Thèse

présentée à

L'Université Paris VI

pour obtenir le titre de

Docteur de l'Université Paris VI

Spécialité :

Systèmes Informatiques

par

M. Lei ZHOU

Sujet de thèse :

Analyse Statique et Dynamique de la Complexité des Programmes Scientifiques

Soutenue le 14 septembre 1994 devant le jury composé de :

M.	**CLAUDE**	**GIRAULT**	Président
M.	**PAUL**	**FEAUTRIER**	Directeur de Thèse
M.	**JEAN-MICHEL**	**FOURNEAU**	Rapporteurs
M.	**SERGE**	**PETITON**	
Mme.	**CORINNE**	**ANCOURT**	Examinateurs
M.	**FRANÇOIS**	**IRIGOIN**	
Mme.	**NADIA**	**TAWBI**	

Thèse préparée à :

l'Ecole Nationale Supérieure des Mines de Paris

Rapport EMP/CRI/A-255

Résumé

Le langage Fortran est largement utilisé dans le domaine du calcul scientifique. Pour choisir les meilleurs optimisations pour un programme Fortran réel, nous devons comparer les temps d'exécution des différentes versions optimisées de ce même programme. Comme les programmes Fortran réels peuvent s'exécuter pendant des heures sur des machines très chères, il serait intéressant de prévoir les temps d'exécution précis de ces programmes au moyens d'analyses statiques. C'est le propos de notre travail : prévoir le temps d'exécution d'une application sur une machine donnée sans exécuter ce programme sur cette machine pour toutes les entrées possibles.

Cette thèse présente une nouvelle approche pour prédire les performances, qui a été implantée et expérimentée. Elle est appelée l'analyse de complexité dans l'environnement de programmation de PIPS. Nous montrons comment et avec quelle précision notre model de complexité, basé sur des approximations polynomiales, permet d'estimer les temps d'exécution des programmes Fortran. Différentes travaux théoriques et pratiques sont présentés :

1. approximation des temps d'exécution par des polynômes.
2. modélisation d'une machine séquentielle.
3. implémentation d'un estimateur automatique.
4. validation sur une machine séquentielle.
5. modélisation d'une machine parallèle.

Dans cette thèse nous montrons d'abord comment une librairies de polynomes peut être utilisée pour construire un estimateur statique de temps d'exécution. Nous présentons ensuite de nombreuses expériences pour vérifier que les effets du compilateur et des caractéristiques de la machine peuvent être résumés à l'aide d'une simple liste de coefficients. L'estimateur a été implanté dans un environnement de programmation parallèle - PIPS -, et les coefficients ont ensuite été utilisés pour valider cette approche pour les machines séquentielles.

Nous avons remarqué durant nos expériences que l'essentiel des caculs de programmes Fortran réels a lieu à l'intérieur des boucles, au delà de 90% du temps d'exécution. De plus, pour de gros programmes scientifiques, en supposant que la probabilité des branchements conditionnels est de 50%, et en gardant les variables inconnues telles que, nous avons estimé les temps avec succès les temps d'exécution de programmes réels. Pour le module EFLUX de FLO52, un programme du Perfect Club, les différences entre les temps d'exécutions réels et estimées sont en dessous de 5% pour une large plage de paramètres d'entré.

Enfin, notre dernier but est de faire fonctionner notre estimateur pour des machines parallèles. Pour pouvoir étendre les bons résultats obtenus sur machines séquentielles aux machines parallèles, nous avons fait une série de tests sur une multiprocesseurs MIMD à mémoire partagée, le BBN TC2000. Nous avons montré que les résultats obtenus sur machines séquetielles peuvent être étendus aux machines parallèles avec des modifications mineures de l'algorithme. Plusieurs algorithmes empiriques ont été dérivés des résultats expérimentaux pour modéliser les boucles parallèles.

Mots clés: temps d'exécution, performance, complexité, prediction, estimation, modelisation, parallélisme.

Abstract

Fortran programs are widely used in the scientific computing area. In order to choose the best optimized version of a real Fortran program, we have to compare the execution times of different versions of the same Fortran program. Since real scientific Fortran programs can run for hours on expensive machines, it would be useful to be able to perform a static analysis that accurately predicts execution times. This is the purpose of our work: predict the execution time of an application on a given machine without running that program on the machine for any possible input.

This thesis presents a new performance prediction approach which was implemented and experimented. It is called complexity analysis in the PIPS programming environment. We show how and how well our complexity model, which is based on polynomial approximations, estimates the execution times of Fortran programs. Different theoretical and practical studies were carried out:

1. approximation of execution times by polynomials.
2. modeling of a sequential machine.
3. implementation of an automatic estimator.
4. validation on a sequential machine.
5. modeling of a parallel machine.

In this thesis we first show how a library of polynomials can be used to construct a static execution time estimator. We then present many experiments to check that compiler effects and machine characteristics could be summarized in a simple table of coefficients. The estimator was implemented in a parallel programming environment — PIPS, and the coefficient table was then used to validate the approach for sequential machine.

We have noticed during our experimentations that major parts of calculations of real Fortran programs are in loops. It is said that it accounts for more than 90% of execution time. Therefore, for large scientific programs, while assuming branch statement probabilities are fifty-fifty and keeping unknown variables as they are, we successfully estimated a suite of real programs. Particularly, for module EFLUX of FLO52 program from Perfect Club, the differences between estimated and measured execution times are smaller than 5% for a large range of input parameters.

Finally, our ultimate purpose is to make our estimator work for parallel machines. In order to expand this performance estimator from successful sequential program evaluation to parallel program evaluation, we have carried out several tests on a shared-memory multiprocessor MIMD machine — BBN TC2000. We have found that the experience obtained from sequential machines can be easily extended to parallel machines with minor modifications of the algorithm. Several empirical algorithms have been deduced from the experimental results to model parallel loop.

Keywords: execution time, performance, complexity, prediction, estimation, modelization, parallelism.

Acknowledgements

First of all, I would like to thank Mr. Claude Girault for accepting to be the president of the jury, and Mr. Jean-Michel Fourneau, Mr. Serge Petiton, Ms. Corrine Ancourt, Mr. Paul Feautrier, Mr. François Irigoin and Ms. Nadia Tawbi for accepting to be the members of the jury.

It was pretty difficult to complete my doctorate work. Fortunately, there were so many people who helped me along the way.

I would like to express my sincere appreciation to my advisor, Mr. François Irigoin, vice-director of C.R.I. Ecole des Mines de Paris, for his patience, guidance, insight, and encouragement which made the completion of this thesis possible. I will always be extremely grateful for the many hours of conversation he and I have had over the past three years. I would also like to thank former director of C.R.I. Ecole des Mines de Paris, Mr. Michel Lenci, now director of I.S.I.A. (Institut Supérieur d'Informatique et d'Automatique), who spared time to interview me and accepted me at C.R.I. He was never far away when I needed help. And equally, I wish to thank the new director, Mr. Robert Mahl, for his encouragement and mental support. I am indebted to Mr. Pierre Jouvelot, I enjoy very much the talks with him, as well as adventure tour with him in his private plane.

I would like to thank the members of the PIPS group, particularly the following people: Mr. Pierre Berthomier, who initiated the work of complexity program; Mr. Bruno Baron, for his patience and willingness of explanation, he greatly helped me get my work started during my first several months at the center; Ms. Danny Durand, visiting scholar of the center, who helped me initiate the work on parallel machine — BBN TC2000, and along all the experiments on the machine. She also helped me correct my article of CONPAR'92 to make it more *American*; Mr. Fabien Coelho, Ms. Béatrice Creusillet and Mr. Laurent Vaucher who helped me translate the abstract and introduction into French.

I would like to thank Ms. Corinne Ancourt, Ms. Yi-Qing Yang, Mr. Ronan Keryell, Mr. Vincent Dornic, Ms. Nahid Emad, Ms. Kathryn McKinley, Ms. Jacqueline Altimira, Ms. Annie Pech-Ripaud, Mr. Jean-Marie Kessis and the other members of C.R.I. for their help.

Most importantly, I would like to thank my family, particularly my parents, for their help throughout my life. They have always encouraged me to do my best. Also, I do not think I would have survived this process without the continuous support of my wife Catherine. She helped me keep my body and soul together through the struggles of this Ph.D.

Finally, I would like to thank all members of the school soccer team, for the friendly matches with them on Tuesdays make the life more enjoyable.

I could not have achieved this without them. Thank you all!

Table des matières

1 Introduction **29**

1.1 Motivation 29

1.2 Hypothèses et objectifs 30

1.3 Performance et complexité 32

1.4 Notre approche pour l'estimation des performances 38

1.5 Contributions 45

1.6 Plan de la thèse 46

2 Estimation de la complexité des programmes dans PIPS **49**

2.1 Présentation de PIPS 49

2.2 Modélisation de la complexité 55

2.3 Algorithmes de calcul de la complexité pour des programmes séquentiels structurés . 60

2.4 Algorithmes de calcul de la complexité lorsque le graphe de flot de contrôle est non-structuré 67

2.5 Calcul symbolique p

Avant-Propos

Les supercalculateurs ont maintenant atteint une phase de maturité. Ils sont certes plus coûteux, mais aussi plus rapides et disposent de plus de mémoire que les systèmes traditionnels. Ils constituent donc potentiellement une importante source de puissance. Un des enjeux principaux de l'évaluation des performances est la mise en valeur de cette réserve de puissance.

1 Motivations

L'analyse et l'estimation des performances d'un programme jouent un rôle important dans le domaine du calcul parallèle. En effet, les temps d'exécution des programmes scientifiques réels peuvent être très importants, même sur des machines multiprocesseurs. Dans ces conditions, il est intéressant de pouvoir prédire statiquement et de manière précise les temps d'exécution de ces applications. Ceci permettra, par exemple, de comparer différentes versions optimisées pour différentes machines cibles.

Prédire le temps d'exécution d'une application sur un ordinateur est, dans le cas général, un problème impossible à résoudre. On peut toutefois se demander si c'est également vrai dans le cadre des programmes scientifiques réels écrits en langage Fortran. Pour cela, deux types de méthodes peuvent être envisagés :

- **Méthodes Dynamiques**
- **Méthodes Statiques**

Les méthodes dynamiques sont les plus faciles à mettre en œuvre. Elles consistent à exécuter le programme considéré sur différentes machines (Cray, réseau de Tranputeurs, CM-5, ...) avec différentes tailles de problèmes et différents paramètres d'entrée. Ceci n'est pas toujours possible, car il faut disposer d'une version différente du programme pour chaque machine. De plus, ce n'est pas la meilleure méthode. Elle est très coûteuse, et les relations qui existent entre les différents paramètres et les temps d'exécution ne peuvent être établies avec certitude qu'après un grand nombre d'expériences.

Les méthodes statiques sont moins coûteuses, mais ne donnent pas toujours des résultats satisfaisants. Par exemple, il n'est pas possible de prédire lors de l'analyse statique le comportement d'une instruction de branchement ou d'un `while`. Le résultat doit alors être pondéré par des probabilités.

Ni les méthodes dynamiques, ni les méthodes statiques ne sont à elles seules suffisamment puissantes pour résoudre le problème de la prédiction des performances d'un programme. Nous pensons qu'il serait intéressant de combiner ces deux méthodes pour tirer parti de leurs avantages respectifs. Mais nous ne sommes pas parvenu à éviter leurs inconvénients, qui sont pour la première le coût d'exécution, et pour la seconde des estimations parfois grossières.

Le problème général peut être formulé de la façon suivante :

> Étant donné un programme et un supercalculateur, peut-on prédire le temps d'exécution du programme sur cet ordinateur sans l'exécuter réellement ?

Dans cette thèse, nous apportons une réponse positive à cette question : nous avons déterminé une méthode qui résoud ce problème de manière efficace.

2 Hypothèses et Objectifs

Le principal objectif de cette thèse est donc de développer une méthodologie permettant de prédire le temps d'exécution d'un programme, quels que soient ses paramètres d'entrée. Cette information ne pourra pas toujours être obtenue de manière exacte, mais nous voudrions pouvoir au moins comparer les temps d'exécution de deux versions d'un même programme, obtenues à l'aide de méthodes différentes.

Le coût des entrée/sortie des programmes scientifiques est difficile à estimer, car ces opérations mettent en jeu le système d'exploitation et des caractéristiques temps réel. Une estimation du temps moyen d'une opération d'entrée/sortie pourrait être utilisée, mais nous supposons qu'elles sont effectuées séparément. Nous ne traiterons donc pas ce problème dans cette thèse.

Nous avons implanté notre estimateur de performances au sein du Paralléliseur Interprocédural de Programmes Scientifiques (PIPS) développé à l'École des Mines de Paris. Le programme est tout d'abord analysé et représenté sous une forme interne propre à PIPS. Puis notre analyseur de complexité utilise le résultat de ces analyses (structure de contrôle du programme, opérations utilisées dans les instructions) ainsi qu'une modélisation de la machine cible sous forme d'une *table de coûts* pour calculer la complexité du programme considéré. Le résultat est inséré dans le code source sous forme de commentaires, de la même manière que les résultats des autres phases de PIPS.

3 Notre Approche pour Performance Estimation

Dans cette section, nous présentons plus en détail différents problèmes qui vont nous permettre de préciser le cadre de cette étude : (1) à quel niveau de langage devons-nous effectuer notre analyse ? (2) quelle méthode statique, dynamique, ou une combinaison des deux, devons-nous utiliser pour estimer le temps d'exécution ? (3) quelle est l'unité de code atomique (la plus petite) que nous considérerons ? (4) sous quelle forme afficherons nous le résultat ?

3.1 Quel niveau langage choisir

Le choix du type de langage qui va pouvoir être traité par l'estimateur de performance n'a pas encore été évoqué. Il s'agit de choisir entre un langage de haut niveau, pour lequel l'estimateur devra tenir compte du fonctionement du compilateur, et le code assembleur, pour lequel il n'y a pas d'assembleur à considérer.

Niveau Assembleur

Si on choisit de faire l'analyse sur le code assembleur, on peut s'attendre à une estimation très fine, car on sait exactement quelles instructions vont être exécutées (accès à la mémoire, aux registres, ...) et combien de cycles elles prennent. De plus, on peut analyser n'importe quel programme, quel que soit le langage dans lequel il a été initialement écrit. Cependant, il existe des désavantages importantes à une telle approche:

- Premièrement, on devient dépendant de la machine sur laquelle on travaille, puisque chaque machine possède son propre assembleur. Le programme assembleur est aussi nettement plus gros que le programme source (jusqu'à un rapport douze d'après les tests de Berthomier [Bert90]).
- Deuxièmement, le code assembleur est en général bien plus difficile à interpréter qu'un programme en langage évolué comme Fortran ou C. Cela explique d'ailleurs le fait que l'on programme de préférence en langage de haut niveau. Les optimisations réalisées à la compilation compliquent encore les choses, et il devient très difficile de prévoir le comportement du programme d'après les résultats analytiques.
- Troisièmement, certaines informations importantes, comme le nombre d'itérations dans une boucle, ne sont plus disponibles au niveau de l'assembleur.

En outre, le fait que le code assembleur soit peu structuré ajoute encore à la compléxité de l'analyse.

Niveau Langage Source

Étant donné les nombreux inconvénients dus au code assembleur, on a choisi de faire les analyses au niveau du langage source. Comme de plus c'est à ce niveau que se font les développements et la maintenance du code, cela permet de mieux percevoir la correspondance entre le code écrit et l'analyse de performances. Il est en effet difficile de raisonner à un niveau avec des résultats provenant d'analyses faites à d'autres niveaux. Il serait aussi assez maladroit de faire l'analyse en temps à un niveau différent de l'analyse logique du programme. D'autres informations, qui se révèlent la plupart du temps décisives pour évaluer le temps d'éxécution et que l'on ne peut prélever qu'à un haut niveau sont la compléxité de l'algorithme et les relations entre procédures.

Les avantages de l'analyse au niveau du langage source sont nombreux et évidents. Les programmes sont souvent bien structurés, de nombreuses informations sont disponibles au sujet des algorithmes, etc.. Le problème à ce niveau est que le compilateur va générer un code objet qui ne repectera pas

forcément bien la construction initiale du programme, des techniques d'optimisation de code ayant été appliquées En conclusion, on peut dire qu'il est difficile d'estimer précisément quelle forme va prendre une construction du code source dans le code assembleur. Nous avons effectué un ensemble d'expériences sur SUN SPARCstation 2 pour se rendre compte à quel point on arrive à prendre en compte de manière statique les optimisations. Les résultats sont donnés dans le chapitre 5. Des architectures plus complexes ont aussi pu être modélisées de façon satisfaisante [Wang93].

Le programme d'évaluation pour langage de haut niveau perd quelques informations, mais travaille sur des fichiers bien plus petits et devrait donc fonctionner nettement plus rapidement.

Option choisie: Langage de haut niveau

En se fondant sur notre discussion, nous avons choisi l'analyse de langage de haut niveau.

3.2 Méthodes d'Évaluation

Le but de notre méthode est d'exprimer la complexité sous forme d'expressions symboliques, ce qui permet d'éviter les problèmes de l'analyse statique traditionnelle, et donc de conserver la puissance de prédiction de notre estimateur. Dans cette section, nous présentons différentes manières de comparer ou d'évaluer les performances de programmes Fortran.

Si un utilisateur désire comparer deux versions d'un même programme, il doit tout d'abord s'assurer que leur sémantique est bien équivalente. Notre but n'est pas de le vérifier, mais de calculer les performances relatives de plusieurs versions différentes d'un même code, afin de les comparer, et d'aider ultérieurement au placement et à l'ordonnancement des tâches de la version choisie.

Mesure dynamique des temps d'exécution

Au lieu de faire une comparaison "différentielle" de deux programmes, nous évaluons chacun d'eux séparément de manière "absolue" et ensuite nous comparons leur évaluation.

Unix fournit des outils pour la mesure dynamique de performance. Il y a principalement deux méthodes pour récupérer des profils d'éxécution : *prof* et *tcov*. La première technique consiste à échantillonner périodiquement le compteur ordinal de programme et d'en déduire à partir de sa valeur quelle fonction ou procédure la machine est en train d'exécuter. On peut ensuite calculer le pourcentage de temps passé dans chaque fonction. L'autre technique consiste à ajouter un compteur avant chaque instruction du programme à analyser. Toutes les valeurs des compteurs sont ensuite écrites dans un fichier lorsque l'exécution du programme se termine.

L'inconvénient principal de cette mesure dynamique est qu'elle ne donne que des valeurs dépendant du jeu de données. Si on veut connaître le comportement de son programme en fonction des paramètres principaux (la taille des données ou le nombre de processeurs pour un programme parallèle), on doit estimer la fonction à partir de telles mesures ponctuelles. Nous verrons que l'évaluation statique peut fournir directement la fonction de complexité lorsque le programme n'est pas trop compliqué.

Évaluation statique symbolique

Nous avons dit que le principal avantage de l'évaluation statique comparée à l'évaluation dynamique est son faible coût d'exécution. Cependant il reste des difficultés liées à l'évaluation statique du langage Fortran. Il y a quelques situations qui ne peuvent pas être prises en compte telles que :

une instruction de branchement : lorsqu'on rencontre un IF, il est souvent impossible d'approximer la complexité moyenne de l'instruction toute entière. On ne peut pas faire mieux que de supposer un choix équiprobable pour les deux branches.

Mais certains cas sont utilisés seulement pour vérifier des conditions courantes ou détecter des anomalies et il est facile de les détecter car ils mènent à des STOP, ABORT ou EXIT. On peut supposer que cette branche n'est que rarement prise et donc lui donner une probabilité d'exécution de 0.

Néanmoins, la plupart du temps, cette probabilité ne peut être calculée statiquement ;

le nombre d'itération : il est ici crucial de l'évaluer puisque généralement plus de 90% du temps est passé dans des boucles. L'analyse sémantique peut être utilisée pour exploiter l'information disponible au niveau des variables du programme. En particulier les variables symboliques peuvent être évaluées alors que sans elle seules les bornes de boucles constantes peuvent être utilisées, ce qui n'est pas très intéressant. Mais l'analyse symbolique ne résoud pas tous les problèmes.

Un inconvénient de l'évaluation statique comparée à l'évaluation dynamique est qu'elle nécessite une modélisation de la machine sur laquelle le code analysé est sensé tourner. Appliquée à un programme écrit dans un langage de haut niveau, cette méthode perd de la précision à cause de toutes les transformations effectuées par le compilateur comme cela a été expliqué précédemment.

Puisque c'est une méthode statique, aucune récursion n'est autorisée à moins qu'un algorithme de point fixe soit utilisé sur le graphe d'appel.

Combinaison d'une estimation statique et de mesures dynamiques

Cette méthode se décompose en trois étapes :

1. on commence par une première passe d'évaluation statique pour accumuler de l'information sur les points où l'évaluation statique échoue : la position de tests IF où les probabilités de branchement ne peuvent pas être calculées (presque tous en fait) et la position des boucles DO dont les bornes n'ont pas été calculées exactement, comme le sont les boucles WHILE. Pour ce qui est des parties de programme non structurées, PIPS les convertit en plusieurs blocs structurés.
2. dans un deuxième temps, une analyse dynamique est utilisée pour résoudre les points marqués par la première passe. Une copie du programme est faite avec ajout de compteurs à tous les endroits où cette première passe a échoué. La version modifiée du programme est ensuite exécutée avec un certain nombre de jeux de données. À la fin de ces exécutions les valeurs des compteurs sont écrites dans un fichier qui est exploité par la seconde passe de l'analyse statique. Des outils tels que *tcov* pourraient être utilisés ;

3. la dernière étape consiste à refaire une phase d'évaluation statique en utilisant les valeurs des compteurs générées lors de l'étape précédente afin d'obtenir la complexité globale du programme.

Un avantage de cette méthode est que la mesure du profil d'éxécution peut être effectuée sur n'importe qu'elle machine (ayant néanmoins toujours la même représentation interne des nombres flottants). Comme la méthode a besoin de beaucoup moins d'information de profil que l'approche dynamique et que sa sortie d'évaluation est paramétrique, ses résultats sont moins sensibles au choix des ensembles de données fournis pour l'échantillonnage des profils d'exécution.

En outre, la seule utilité de la première étape d'évaluation de la complexité est d'insérer des compteurs seulement où cela est nécessaire afin de gagner du temps sur l'exécution du programme modifié. En pratique il peut être plus avantageux de sauter cette étape et de choisir de rajouter des compteurs au niveau de chaque IF et de chaque DO plutôt que de faire la première phase qui peut prendre plus de temps de décision que de temps effectivement gagné par la suite.

Bien entendu, comme pour les autres approches, il y a un certain nombre d'inconvénients :

- puisque l'estimation statique est utilisée il n'y a pas de récursion possible à moins encore d'utiliser un algorithme de point fixe sur le graphe d'appel des fonctions et procédures;
- la correction ne peut pas être prouvée. À supposer que les complexités de programme soient approximées par des polynômes, il est évident qu'une telle estimation de complexité ne peut donner de bons résultats pour tous les algorithmes. Par exemple la complexité des algorithmes de Quicksort et de FFT ont une composante logarithmique qui ne sera pas devinée par les mesures dynamiques.

Option choisie: Une approximation purement statique

Il semble impossible de combiner de manière sûre des informations dynamiques précises sur le comportement du programme recueillies pour un jeu d'entrée donné, et les informations statiques inférées pour n'importe quel jeu d'entrée valide. Les mesures dynamiques pourraient s'avérer utiles pour détecter les branchements du programme à faibles probalités d'exécution, comme les tests d'erreur. Mais d'après Wang [Wang94], une analyse statique du programme est suffisante pour détecter de tels cas. Par ailleurs, les bornes de boucle ainsi mesurées sont peu concluantes : dans le cas d'une boucle `while` par exemple, qui vérifie un critère de convergence, la durée peut dépendre fortement des données, et ne pas être significative.

Par contre, l'utilisation de variables “spéciales”, reliées d'une manière ou d'une autre à l'origine même du problème d'estimation rencontré, dans le code source, est nettement plus sympathique pour l'utilisateur. La correction de la formule dérivée peut être démontrée, et l'utilisateur peut fournir s'il le désire, et s'il en dispose, des informations complémentaires qui seront facilement intégrées dans les résultats par simple substitution de ces variables spéciales.

Des expériences sont nécessaires pour vérifier sur des programmes réels qu'un nombre raisonnable de telles variables sont introduites par l'analyse, afin de garantir la lisibilité des résultats de l'analyse de complexité. De telles expériences ont été menées et des résultats encourageants dans ce sens ont été obtenus. Ils sont présentés au chapitre 5.

3.3 Choix de la classe de fonctions pour l'analyse de complexité

Le choix de la classe de fonctions mathématiques pour l'analyse de complexité est dicté par différents critères parfois contradictoires.

1. D'abord, ces objects doivent être suffisamment complexes pour pouvoir accumuler et propager l'information recueillie au différents niveaux de l'analyse.
2. Il doivent cependant être suffisamment simples pour supporter des opérations comme la somme, le produit ou l'intégration de manière interne.
3. De plus, si l'on veut pouvoir comparer les temps d'exécution de deux programmes, deux objets doivent être comparables.
4. Enfin, ils doivent convenir à l'utilisateur final, à savoir une personne humaine doit pouvoir les exploiter : la lisibilité du résultat est donc un critère important.

Certains de ces critères sont contradictoires, ils ne peuvent donc être tous satisfaits à la fois. Il faut faire un chois, et le meilleur possible. Voici quelques classes de fonctions envisageables pour l'analyse de complexité :

- Évaluer la complexité au moyen d'une simple constante satisfait à tous les critères de choix exposés, sauf le premier : si par exemple l'information calculée à l'intérieur d'une procédure dépend de paramètres passés à cet procédure, *i.e.* des constantes symboliques, cette information est perdue.
- Les expressions syumboliques en général ne perdent pas d'information, mais certaines opérations comme l'intégration ne sont pas toujours possible car il n'existe pas de formule générale pour ce faire.
- Les polynômes (à coefficients constants) permettent l'intégration, peuvent représenter des temps d'exécution paramétriques, et sont aisément manipulables. Cependant, les complexités exponentielles ou logarithmiques ne sont pas exprimables du fait de la constance des exposants.
- Une paire de polynômes bornant inférieurement et supérieurement le temps d'exécution est envisageable. Mais cette information n'est pas toujours significative. Ainsi, pour une boucle `while`, les bornes sont 0 et l'infini... Park [Park92] utilise néanmoins cette approche et donne un intervalle de prédiction.
- Une forme polynômiale conditionnelle, comme par exemple :

  $$\begin{cases} N^2 + 10 \cdot N & \text{if } N > 10 \\ N^2 + 20 \cdot N & \text{if } N >= 10 \end{cases}$$

  permettrait d'inclure une information booléenne dans l'expression de la complexité au lieu de les évaluer directement. Mais une telle forme n'est pas pratique, car elle peut croitre exponentiellement en fonction de la profondeur du nid de boucles incluant un test, et l'intégration entre des bornes polynomiales n'est pas chose aisée.

Option choisie: Les polynômes comme classe de fonctions

Finallement, les polynômes ont été choisi d'une part pour leur simplicité et parce qu'il permettent de traiter la plupart des programmes scientifiques, qui opérent simplement sur des tableaux. Des variables spéciales marquant les valeurs inconnues ont été introduites pour gérer les problèmes de bornes statiquement inconnues.

3.4 Visualisation des formules de complexité

Les formules de complexité dérivées pour un programme peuvent être utilisées directement par un compilateur pour choisir entre différentes techniques de compilation, mais elles peuvent également être proposées à l'utilisateur par l'environnement de programmation.

La manière de visualiser les résultats de l'analyse de complexité n'est en rien évidente. Comment présenter les résultats de manière claire et concise est aussi difficile. Une personne humaine ne peut pas appréhender et exploiter facilement des formules complexes. Le présentation doit être à la fois correcte, claire et concise.

Option choisie: Visualition sous forme de commentaires

Nous avons décidé de présenter les formules de complexité sous la forme de commentaires, pour éviter un développement excessif pour l'interface utilisateur. Cependant, le nombre moyen de lignes de programme des *benchmarks* du Perfect Club est de 4000, et 4000 formules de complexité ne présentent pas beaucoup d'intérêt pour l'utilisateur. Plus de travail est nécessaire sur cette interface.

4 Contribution

Au cours de ma thèse, j'ai réalisé un outil permettant d'estimer la complexité statique de programmes FORTRAN dans l'environnement de programmation parallèle PIPS. Les différentes étapes de son élaboration ont été décrites dans cette dissertation.

Les nouvelles fonctionnalités que j'ai implantées dans PIPS ainsi que les différentes caractéristiques de notre outil d'évaluation sont maintenant détaillées:

Description de la machine: L'utilisateur peut choisir parmi un ensemble de descriptions de machine, appelées *tables de coûts*, celle qui est la plus appropriée à son choix. Pour le moment, trois tables de coûts sont disponibles. Deux correspondent à des modèles théoriques: l'une modélise une machine où toutes les opérations ont un coût unitaire, et l'autre ne tient compte que des opérations flottantes. La dernière modélise une SUN SPARCstation 2 et donne les coûts mesurés des différentes opérations de base de cette station de travail. Tous les coûts inhérents à chacune des opérations de la machine sont stockés dans un fichier.

Paramètres symboliques: Dans certains cas, des variables du polynôme décrivant la complexité du programme restent symboliques et ne sont pas évaluées. Par exemple, PIPS peut identifier

les variables du programme qui ne doivent pas être évaluées parce qu'elles correspondent à des paramètres libres du module (paramètres formels d'un appel de procédure ou paramètres d'entrée dont la valeur n'est connue que lors de l'exécution). De plus, l'utilisateur peut spécifier les variables qu'il désire conserver symboliquement dans le résultat.

Préconditions Interprocédurales: PIPS utilise des préconditions pour caractériser les informations connues à la compilation sur les variables scalaires du programme avant l'exécution d'une instruction. Une fois que ces préconditions sont calculées, elles sont stockées dans un fichier pour chacun des modules et peuvent être utilisées a posteriori par les autres phases du calcul.

Résultats Simplifiés: Une analyse est effectuée de manière à minimiser le nombre de paramètres libres qui sont locaux à un module. De plus, les variables intermédiaires sont éliminées du résultat final.

Trace du calcul: Il est possible d'obtenir une trace des différentes étapes de calcul qui sont effectuées au cous du processus d'évaluation de la complexité du programme. Cette possibilité s'avère utile notamment : (1) pour clarifier la structure du programme et (2) pour faciliter le deboguage (détection des endroits qui soulèvent des problèmes).

De manière à valider mon approche, plusieurs expériences ont été menées sur une machine séquentielle (SUN SPARCstation 2) et sur un machine parallèle (BBN TC2000):

Machines Séquentielles: Comparées aux résultats obtenus manuellement par Emad [Emad91], nos estimations donnent de bons résultats. La principale différence réside dans l'évaluation des probabilités d'exécution de certaines branches du programme. Nous avons utilisé notre outil pour estimer le temps d'exécution du programme FLO52 extrait du benchmark du Perfect Club. Nous avons comparé le résultat obtenu pour la sous routine EFLUX de ce programme avec le temps réel de son exécution sur une SUN SPARCstation 2. La différence est relativement faible, moins de 5%, quelques soient les intervalles de valeurs pris par les différents paramètres d'entrée.

Machines Parallèles: Nous avons effectué plusieurs expériences sur la BBN TC2000 (multiprocesseur MIMD à mémoire partagée) avec de petits programmes de tests. Notre étude montre qu'il est possible moyennant des modifications légères de nos algorithmes d'étendre les travaux effectués pour les machines séquentielles en vue de leur exploitation pour les machines parallèles.

5 Plan de la thèse

Le chapitre 1 introduit cette thèse. Les arguments qui ont motivé mon travail et le but que je mettais fixé y sont présentés. Les concepts généraux de performance et de complexité d'un programme, utiles à la présentation de notre approche dans l'environnement PIPS, sont introduits. Enfin, ce chapitre précise ce qu'il est possible de prédire de manière générale en terme de performance et de complexité avant d'exposer notre contribution.

Le chapitre 2 décrit les algorithmes classiques utilisés pour estimer le temps d'exécution des programmes. Les caractéristiques, grandes lignes et structures de données de PIPS, qui ont influencées l'implantation et les résultats de nos algorithmes, sont présentées. Nous proposons ensuite une approximation du temps d'exécution d'un programme par une formulation mathématique en deux étapes. La première consiste à masquer les problèmes de bas niveaux liés au fonctionnement de la machine (les effets du compilateur, du système d'exploitation, du matériel tel que la mémoire cache, etc...). La seconde sert à l'approximation du comportement dynamique de l'exécution du programme, simplifié par la première étape, par une représentation mathématique statique. Les choix des approximations effectuées par ces deux étapes sont justifiés au cours de leur présentation.

Le chapitre 3 décrit la modélisation d'une machine cible. Il présente comment caractériser par un ensemble de temps de base le comportement de la machine et de son environnement (compilateur, run-time, système d'exploitation, microprocesseur,...). Dans un premier temps, les machines séquentiels sont étudiées et les paramètres importants utiles à la modélisation de la machine sont présentés . Nous proposons ensuite une extension de ce modèle aux machines parallèles. La dernière section décrit les expériences qui ont été menées pour valider cette modélisation et montre comment la SUN SPARCstation 2 peut être précisément modélisée.

Le chapitre 4 détaille l'implantation de nos algorithmes dans PIPS. L'environnement de programmation général de PIPS est présenté, et précède les structures de données développées pour nos algorithmes ainsi que celles de PIPS directement réutilisées comme le graphe des appels, le graphe de contrôle interprocédural, les préconditions et les effets.

Le chapitre 5 présente les expériences effectuées pour les machines séquentielles permettant de valider nos résultats de complexité sur la SUN SPARCstation 2. Son but est de montrer que notre outil permet généralement l'obtention de bons résultats. Ces expériences ont été utilisées pour valider notre approche et nos algorithmes sur des programmes réels. Nous comparons nos résultats avec ceux obtenus manuellement et vérifions la robustesse de notre approche sur un benchmark du Perfect Club par une comparaison de nos prédictions avec les temps d'exécution réels mesurés pour l'un de ces modules. Les résultats de cette comparaison sur machine séquentielle sont encourageant. Comme l'on pouvait s'y attendre, un outil automatique ne fait pas certaines petites erreurs qu'un humain pourrait faire. Toutefois, il requiert un certain contrôle de l'utilisateur sur les hypothèses simplificatrices qui sont faites, et notamment sur les probabilités utilisées pour les branchements, qui peuvent conduire à des approximations très mauvaises. De manière à valider les résultats, nous avons mesuré les temps d'exécution de la sous routine EFLUX provenant du programme FLO52, et comparé les résultats avec nos prédictions. La différence est inférieure à 5%.

Le chapitre 6 présente les expériences effectuées sur la machine parallèle BBN TC2000. L'un des buts de cette thèse était de développer un outil permettant l'estimation de la complexité d'un programme sur machines parallèles. Ce chapitre est un premier pas vers cet objectif. Il contient une description des expériences sur la BBN destinées à caractériser le comportement des boucles parallèles. Nous avons fait une série de programme de test de manière à extraire des formules permettant de prédire ce comportement. Ces équations pourront être utilisées ultérieurement pour étendre nos travaux aux machines parallèles.

Enfin le chapitre 7 conclut cette thèse. Un rappel des caractéristiques de notre outil est tout d'abord exposé. Les différents travaux dans le domaine de l'évaluation de performance y sont ensuite détaillés. Nous introduisons les différentes techniques les plus connues utilisées par les outils de prédiction des performances et distinguons quatre catégories: évaluation manuelle, simulation,

analytique et par mesures. Nous classifions aussi les outils existant selon la nature de la machine ciblée: monoprocesseur, multiprocesseurs, multiprocesseurs avec mémoire partagée ou distribuée. Enfin, nous comparons ces outils avec notre contribution avant de présenter les différents travaux qu'on envisage pour compléter notre étude.

L'appendice A contient la représentation intermédiaire des programmes qui est utilisée par PIPS.

L'appendice B contient les temps d'exécution d'opérations élémentaires écrites en C pour une SUN SPARCstation 2. Ces temps d'exécution sont utilisés au chapitre 5 .

L'appendice C contient les résultats complets pour la sous routine CALCG du programme TMINES.

Contents

1 Introduction **29**

1.1 Motivation . . . 29
1.2 Research Goal . . . 30
1.3 Performance and Complexity — Our Model . . . 32
1.3.1 Performance of Systems . . . 32
1.3.2 Complexity of an Algorithm . . . 34
1.3.3 Program Execution Model . . . 35
1.3.4 Measurement Model . . . 37
1.3.5 Summary . . . 38
1.4 Our Approach to Performance Estimation . . . 38
1.4.1 Language Level Issue . . . 38
1.4.2 Evaluation Method . . . 40
1.4.3 Mathematical Functions for Complexity . . . 43
1.4.4 Visualization of Complexity Formulae . . . 44
1.5 Contributions . . . 45
1.6 Dissertation Outline . . . 46

2 Complexity Estimation in PIPS **49**

2.1 PIPS Overview . . . 49
2.1.1 Motivation for Automatic Parallelization . . . 49
2.1.2 PIPS Historical Goals . . . 50
2.1.3 PIPS General Design . . . 51
2.1.4 PIPS Data Structures . . . 52
2.1.5 Control Graph . . . 52
2.1.6 Effects . . . 53
2.1.7 Transformers . . . 54
2.1.8 Preconditions . . . 54
2.1.9 Summary of PIPS . . . 55
2.2 Complexity Model . . . 55
2.2.1 Dynamic Model . . . 57

2.2.2 Static Model 58
2.2.3 Problem with the Dynamic Model 59
2.2.4 Model Summary 60
2.3 Complexity Algorithm for Structured Sequential Programs 60
2.3.1 Complexity for Primitives 60
2.3.2 Complexity of an Expression 61
2.3.3 Complexity of a Statement 62
2.3.4 Complexity of a Structured IF 62
2.3.5 Complexity of a Sequential DO 64
2.3.6 Complexity of a Block 66
2.4 Complexity Algorithm for Unstructured Control Flow Graph 67
2.4.1 Control flow graph and PIPS programs representation 67
2.4.2 Complexity of an Unstructured Control Flow Graph 67
2.4.3 Example 69
2.5 Symbolic Calculation of Polynomial Summation 70
2.5.1 The Formula of Symbolic Calculation 70
2.5.2 Problems of Symbolic Calculation 72
2.6 Summary 74
2.6.1 Top-Down vs. Bottom-Up Analysis 74
2.6.2 Proper Complexity and Cumulated Complexity 74

3 Target Machine Model 75
3.1 Sequential Machines 75
3.1.1 Sequential Machine Model 75
3.1.2 Multiple Processing Units 77
3.1.3 Memory Issues: Register, Cache and Virtual Memory 78
3.2 Parallel Machines 79
3.2.1 Parallel Machine Architecture 79
3.2.2 Classification for Parallel Machines 81
3.3 Two Trial Models 82
3.3.1 All-One Cost Table 82
3.3.2 Floating-Point-One Cost Table 84
3.3.3 Summary of Trial Cost Tables 85
3.4 Experimental Machine Model 85
3.4.1 The Limit for Machine Modelization Accuracy 85
3.4.2 A Simple Program to Start 85
3.4.3 Automating the Elementary Measurements 87
3.4.4 Elementary Fortran costs on SUN SPARCstation 2 89
3.5 Empirical Cost Tables on SUN SPARCstation 2 90

3.5.1 Assumptions 90
3.5.2 Operation Costs 90
3.5.3 Memory Access Costs 96
3.5.4 Intrinsic Costs 96
3.5.5 Reference Costs 97
3.5.6 Loop and Call Overheads 100
3.5.7 Summary 104
3.6 Discussion 104
3.6.1 Problems with Compiler 104
3.6.2 Fortran vs. C 105
3.7 Summary 106

4 Implementation 107

4.1 Environment 107
4.1.1 NewGen 107
4.1.2 PIPS 108
4.2 Complexity in PIPS 108
4.2.1 Polynomial Form 108
4.2.2 Counters 110
4.3 Prerequisites for Complexity 112
4.3.1 Call Graph 112
4.3.2 Interprocedural Control Flow Graph 114
4.3.3 Preconditions 115
4.3.4 Effects 116
4.4 Description of Complexity Evaluation Code 116
4.4.1 C File Descriptions 116
4.4.2 Abstract Syntax Tree Scan 117
4.4.3 Expression to Value 117
4.4.4 Unstructured Control Flow Graph 117
4.5 Interfaces to Complexity Estimator 117
4.5.1 Terminal Interface 118
4.5.2 X-Window Interface: wpips 119
4.5.3 Development Interface 120
4.6 Parameters of Estimator 122
4.6.1 Variables 122
4.6.2 Cost Tables 123
4.6.3 Early Evaluation vs. Late Evaluation 123
4.6.4 Other Printout and Debugging Parameters 125
4.7 Features of Estimator 125

4.7.1 Preconditions 125
4.7.2 Simplifying Complexity Results 126
4.7.3 Locating Unknown Loop Bound 128
4.7.4 Tracing of Complexity Estimation 130
4.7.5 Debug 130
4.8 Summary 130

5 Experiments on Sequential Machines 133
5.1 Comparisons with Manual Calculation 133
5.1.1 Mailla Subroutine 134
5.1.2 Poltri Subroutine 135
5.1.3 Calmat Subroutine 135
5.1.4 Resul Subroutine 136
5.1.5 Prepcg Subroutine 136
5.1.6 Romat Subroutine 137
5.1.7 Des Subroutine 138
5.1.8 Rep Subroutine 138
5.1.9 Prod Subroutine 138
5.1.10 Calcg Subroutine 139
5.1.11 Tmines Program 141
5.1.12 Summary and Discussion 142
5.2 Results of a Perfect Club Program FLO52 142
5.3 EFLUX of FLO52 145
5.4 Summary 149

6 Experiments on Parallel Machine 151
6.1 Introduction 151
6.2 Background and Terminology 153
6.3 The Machine Description 155
6.3.1 Basic Characteristics of TC2000 155
6.3.2 Architecture of TC2000 156
6.3.3 nX Operating System 156
6.3.4 Uniform System Library 156
6.3.5 Xtra Programming Environment 156
6.4 Gist Performance Analyzer 157
6.4.1 Logging Events with the Event Logging Library 157
6.4.2 Output of Event Logs with Gist 159
6.4.3 Analyzing Event Logs with the Gist 160
6.4.4 Summary of the Analyzer 161
6.5 Experimentations 164

6.5.1 Explanation of Test Program 164
6.5.2 Startup Time 164
6.5.3 Finish Time 167
6.6 Algorithm for Parallel Machines 168
6.7 Verification of Formulas 172
6.8 Discussion 173
6.8.1 Processor 175
6.8.2 Intertask Time of Triangle Form 175
6.9 Summary 175

7 Conclusion 177
7.1 PIPS Complexity Estimator 177
7.2 Related Work 178
7.2.1 Performance Evaluation Methods 178
7.2.2 Machine Modelization 180
7.2.3 Classification of Performance Estimators 181
7.3 Future Work 184

Bibliography 187

A Intermediate Representation in PIPS 197

B Elementary C Costs on SUN SPARCstation 2 199

C Calcg of Tmines 201
C.1 Original CALCG Subroutine 201
C.2 Complexity CALCG Subroutine with Statement Ordering 203
C.3 Probability Matrix of CALCG Subroutine 208
C.4 Simplification Table of CALCG Subroutine 209
C.5 Simplified Probability Matrix of CALCG Subroutine 210

List of Figures

1.1 The Overall Complexity Scheme 31
1.2 Influences on Performance 36

2.1 Two-Step Approach 55
2.2 The Original Fortran Program 69
2.3 The Unstructured Control Flow Graph 70

3.1 Von Neumann Computer Model 76
3.2 Loop Overhead Detecting Program 101

4.1 Data Structure of Vector in PIPS 109
4.2 Data Structure of Monomial in PIPS 110
4.3 Data Structure of Polynomial in PIPS 111
4.4 Calling relation of ESSAI 113
4.5 Cholesky decomposition subroutine 118
4.6 Cholesky Complexity Output with Display 119
4.7 The Control Window of Wpips 120
4.8 Snapshot of Cholesky Complexity Result 121
4.9 An Example **sample.f** 122
4.10 Complexity with Default Properties 122
4.11 Complexity with Unevaluated Variable 123
4.12 Complexity with fp-1 Cost Table and Unevaluated Variable 124
4.13 Complexity with Late Evaluation 124
4.14 Example of Late Evaluation 125
4.15 Example of Preconditions 126
4.16 Precondition of Example 2 127
4.17 Example 1 of Simplification 128
4.18 Example 2 of Simplification 128
4.19 Example 2 of Simplification 129
4.20 Complexity with Unknown Loop Bound 129
4.21 An Example **trace.f** 130
4.22 Tracing of Sample Subroutine 131

5.1 Calcg Simplified Control Flow Graph 140
5.2 Original Eflux Subroutine of FLO52 146

6.1 Example 1: Source Code 152
6.2 Parallelism Profile of an Application 153
6.3 Brief Format of Gist Output 160
6.4 Gisttext Format of Gist Output 161
6.5 Awk script to analyze the gisttext output 162
6.6 The Complete Procedure of Analysis 163
6.7 Data of Test Program on 8 nodes 164
6.8 Data of Test Program on 16 nodes 165
6.9 Data of Test Program on 24 nodes 165
6.10 Startup Time for 8, 16 & 24 Nodes 166
6.11 Startup Time and Finishing Time for 8 Nodes 167
6.12 Startup Time and Finishing Time of Theoretical Model 168
6.13 Startup Time and Finishing Time 171
6.14 Triangle Form 171
6.15 Triangle Formulas Verification 174

List of Tables

3.1 Relative High and/or Low Memory 79
3.2 Unity Cost for Operation 82
3.3 Unity Cost for Operation for Memory Access 83
3.4 Unity Cost for Array Index 83
3.5 Unity Cost for Intrinsic Transcendental Functions 83
3.6 Unity Cost for Intrinsic Trigonometrical Functions 84
3.7 Floating-Point Cost for Memory Access 84
3.8 Floating-Point Cost for Operation 84
3.9 Elementary Fortran Cost on SUN SPARCstation 2 91
3.10 Operation Costs on SUN SPARCstation 2 96
3.11 Memory Costs on SUN SPARCstation 2 96
3.12 Intrinsic costs on SUN SPARCstation 2 97
3.13 Trigonometric Costs on SUN SPARCstation 2 98
3.14 Array Index Costs - Locally Declared 99
3.15 Array Index Costs - Passed by Arguments 99
3.16 Array Index Costs - Passed by Global Memory 100
3.17 Summary of Array Reference Costs on SUN SPARCstation 2 101
3.18 Call Overheads on SUN SPARCstation 2 103
3.19 Call Subroutine Overheads 103
3.20 Overhead on SUN SPARCstation 2 104

5.1 Complexity Results Using **all-1** 143
5.2 Complexity Results Using **fp-1** 144
5.3 Verification of Eflux subroutine of FLO52 150

6.1 Trapezoid Formulas Verification 173
6.2 Triangle Formulas Verification — Extracted 173

Chapter 1

Introduction

Supercomputers have come of age. While there is a general awareness that supercomputer systems are faster, costlier and have larger memory hierarchies than other computer systems, such characteristics merely imply the existence of great potential power. How much of that power can be harnessed productively is the central theme of performance evaluation.

1.1 Motivation

Performance plays an important role in parallel computing. The users would often like to know the execution time of their application prior to the execution. Although this has been proven theoretically impossible in the general case, can they generally do it for real scientific applications?

In order to predict the execution time of an application and to choose the best optimized version of a real Fortran program for a given machine, we have to know the execution times of different optimized versions of the same Fortran program, or at least, we have to be able to order these executions times. Since real scientific Fortran programs can run for hours on expensive machines, it would be very useful to be able to perform a static analysis and to accurately predict execution times.

Several methods can be used to predict the execution time of a given Fortran program. They belong to one of two classes:

- **Dynamic Methods**
- **Static Methods**

Dynamic methods are the easiest way to go. The program is run on real parallel machines such as a Cray, Transputer or CM-5, with varying number of processors and varying problem sizes. However, this is not always possible because different versions of the program have to be written for each machine. This is not the best way, not only because it is the most expensive one, but also because the relationship between the result and parameters cannot be obtained for sure unless many experiments are carried out.

Static methods are cheaper, but it is not computable sometime. As there is no way to predict the outcome of a branch or a while statement in static evaluation, we are forced to guess or choose forced probabilities.

Neither static nor dynamic evaluations are sufficiently powerful to solve performance prediction problem. It would be interesting to combine the advantages of both methods of evaluation and to get rid of their respective drawbacks as much as possible. We believe that a method using static and dynamic evaluations would be good way to evaluate the execution time of a program but we did not manage to avoid combining their respective disadvantages, expensive profiling and risky extrapolation.

Now the general question is:

> Given a program and a supercomputer, could we predict the execution time of the program on the supercomputer? Or could we know the execution time of the application on the supercomputer without running it on the machine?

Our answer to this question is positive. We think we have found a way to effectively deal with this problem.

1.2 Research Goal

The goal in this thesis is to develop a methodology for predicting the execution time of an application for all its possible inputs on a supercomputer with, presumably, a very expensive CPU-hour price. At least we would like to be able to order the execution times of two versions of the same application, obtained using different transformations. A very important point in the second case is that we do not need exact execution time of a program, which can only be obtained by really executing the application on the specific supercomputer.

Since supercomputers are generally used to perform scientific calculations, the costs for Input/Output are often of secondary importance. Also, they are very difficult to estimate because the operating system and real time issues are involved. An average throughput could be used but we chose to assume that I/O could be done off-line. So we will not discuss this issue in this thesis.

The functional structure of the estimator is shown in Figure 1.1. By **pipsize**, we mean that the original program is analyzed and represented in PIPS internal form. Our complexity analyzer then combines information about the program control structure and the operators used, as well as information about the target machine, the so-called *cost table*. Its output is then merged as comments in the source code as is usually done in the PIPS parallel programming environment.

The reader may wonder why the schema in Figure 1.1 is kept in an unsymmetrical way. Since the left part of the schema is supposed to be running on a supercomputer, while right part is supposed to run on a relatively cheap machine, which is the major part of the PIPS estimator, so keeping right part in axis shows the central work of PIPS estimator.

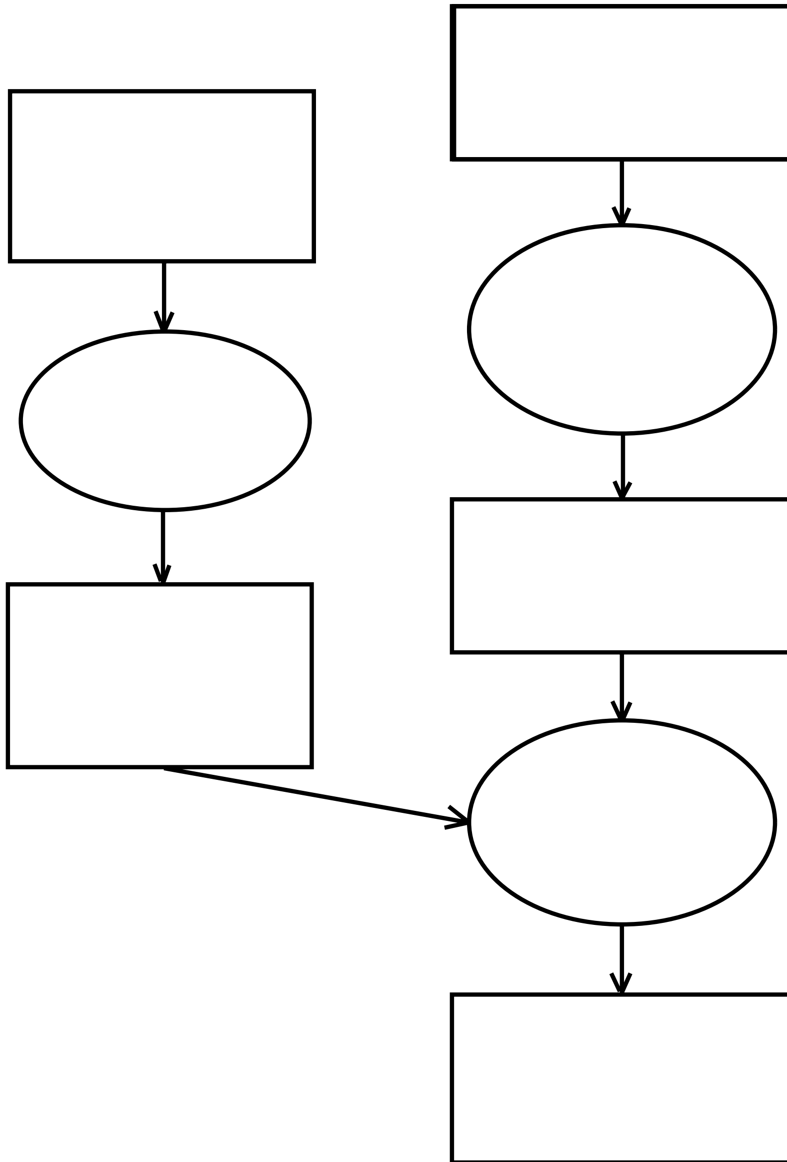


Figure 1.1: The Overall Complexity Scheme

1.3 Performance and Complexity — Our Model

Manufacturers often give "Peak Performance" numbers (MIPS, MFLOPS) that are considerably higher than the performance numbers based on real applications (e.g. Linpack MFLOPS) and, as the peak performances announced by the supercomputer manufacturers are getting higher and higher, the performance percentage that an actual application can reach (its sustained performance) is getting lower and lower. So there is a disharmony between the peak performance and the performance that could be reached. Therefore, "Peak Performance" is an ideal description of the machine under ideal circumstances and is almost meaningless for the users of these machines.

The term "performance" is often used without much thought as to what it really means. In this section, we will clarify what is meant by performance, how the performance of a system is determined and what performance means for a program; and then we will discuss what is meant by complexity and complexity analysis.

1.3.1 Performance of Systems

Performance is an abstract idea that can be applied to any system. We first introduce performance in this general context and then discuss how these concepts apply to programs.

A Definition of Performance

Performance can be defined as the property of a system that makes that system valuable to its users [FSZ79, FSZ83]. In other words, performance indicates how well a system does the task it was designed for. From this definition, performance provides a somewhat subjective means of system evaluation. What one user considers to be important might be of no value to another. If performance is to be used to evaluate the effectiveness of a system, or to compare the relative worth of two systems, a more quantitative assessment is required.

This is achieved by representing the performance of a system by a set of one or more *performance indexes*. Each index has a value which is a measurement aspect of the system of its behavior. When more than one performance index is used, the performance of the system is described by:

$$P = \sum p_i \cdot w_i \tag{1.1}$$

where P is the performance of the system, p_i is the value of i_{th} performance index and w_i is i_{th} weight which indicates the relative importance of each performance index with respect to overall system performance. The w_i are defined such that $\sum w_i = 1$.

For the purpose of this dissertation, all performance indexes will be assumed to take on numeric values. This eliminates subjective indexes such as "programmability" from consideration unless they can be assigned a numeric value. In examining the characteristics of program execution, one is primarily interested in the indexes that reflect how a system uses resources; for example, how it takes to complete an operation or how many processors are being used. Clearly, indexes such as these can be assigned numeric values.

Selecting an appropriate set of performance indexes can be difficult. For example, the value of a computer system may be determined by its computational power. One measure of computing power is the speed at which the computer can perform basic operations. A number of different indexes can be used to represent this notion of performance, including:

- MFLOPS: Millions of FLoating-point Operations Per Second.
- MIPS: Millions of Instructions Per Second.
- Bandwidth: maximum number of bytes transferred per second between the processing component and the memory.
- Delay or Latency: the time taken to service a request, deliver a message, etc.
- MHz: the frequence of internal clock of the processor.
- Benchmarking: the amount of time required to complete the execution of a benchmark program. Such as: Perfect Club [Cybenko91], LINPACK (LINear PACKage), SPEC (System Performance Evaluation Consortium), etc.

This set of performance indexes could be further extended by considering whether the value of an index is determined by the system's best case behavior, or the behavior of the system over a range of activities, its sustained performance.

The users of a system must ultimately decide which indexes best represent their requirements. If multiple indexes are used, then the relative weighting among them must be determined as well. It is important to realize that no combination of performance indexes might fully capture the true value of the system. However, by selecting a specific set of performance indexes, we have a basis from which an objective system evaluation can be made. In this dissertation, the word *performance* is used to mean *the value of the weighted sum of performance indexes being used to represent a system's value*, as in Equation (1.1).

But the only way through which we are interested in performance in this dissertation is execution time. Above indexes can have more or less impact on this execution time. Before we try to define the performance, let us have a look at Figure 1.2 to see what factors can effectively influence the execution time.

The performance of a computer is complicated issue and a function of many interrelated factors. These factors include: the application, the algorithm, the size of the problem, the high-level language, the implementation, the level of human effort used to optimize the program, the compiler's ability to optimize, the maturity of the compiler, the operating system, the architecture of the computer, and the hardware characteristics. In a word, performance depends on many things to which we refer as *performance indexes.*

Methods of Performance Evaluation

The performance of a system is determined by a *performance evaluation study.* Such a study consists of two parts. First, we identify the performance indexes to be used to represent system performance

and second, we determine the value of these indexes. There are two types of performance evaluation studies — performance analysis and performance measurement.

In *performance analysis*, the values of performance indexes are derived from system parameters using mathematical models. Many different tools for performance analysis exist. Of particular importance to parallel programs and parallel computers are queuing models and models based on Petri nets. Analytic methods are most useful when you want to predict performance for a large range of inputs or when the system has not yet been built, or when the cost to perform experiments on the actual system is prohibitive.

Alternatively, if the system has already been built, the values of performance indexes can be obtained directly by taking measurements as the system executes a given task. Obtaining the value of performance indexes in this manner is called *performance measurement.* While this approach is clearly useful, it has the same drawbacks as debugging: the test data may not cover the domain of interest, and measurements may not be feasible without setting up an actual production environment. Another common and useful technique is to simulate the target system using some simulation. The problem here is that the results may not accurately reflect the target, since the simulation model is only an approximation to the real system.

There is an important distinction between the values obtained through performance analysis and the values obtained through performance measurement. Performance analysis methods generally determine the value for a performance index via statistical abstraction. For example, the index values obtained might be representative of a class of system inputs whose size varies according to a Poisson distribution with a specific mean.

The values obtained through performance measurement differ in that they are based on one specific sequence of actions taken by the system in response to one specific set of inputs. Generalizations beyond this sequences of system actions require that the input to the system used for performance measurement be representative of the range of inputs the system will have during actual use.

When performance evaluation is conducted via measurement, the system is instrumented and measurements are made while the system operates. At some point during the development of the system a decision is made to enter a performance evaluation phase and to perturb its normal behavior. An alternative is to make measurement overhead a permanent part of the system, but this is not yet widely supported because hardware support is required to reduce the overhead at an acceptable level. Cray computers and IBM Power-2 architecture support non-intrusive measurements.

1.3.2 Complexity of an Algorithm

An algorithm is a method for solving a class of problems on a computer. The complexity of an algorithm is the *cost*, given in abstract steps, or the amount of storage, given in abstract units of memory, or whatever units are relevant, of using the algorithm to solve any of those problems. To say that a problem is solvable algorithmically means, informally, that a computer program can be written that could produce the correct answer for any input if it is run long enough and is allocated as much storage space as is needed. Before the advent of computers, mathematicians worked very actively to formalize and study the notion of an algorithm, which was then interpreted informally to mean *a clearly specified set of simple instructions to be followed to solve a problem or compute a function.* One of important negative results, established by Alan Turing, was the proof of the

unsolvability of the "halting problem", which is to determine whether an arbitrary given algorithm (or computer program) will eventually halt while working on a given input. No computer program can solve this problem. So if our evaluator encounters this problem, it just uses approximations to deal with unbounded while loops. Some very simple cases of non-halting programs are directly recognized.

A Definition of Complexity

When talking about the complexity, clearly the time and space requirements of a program are of practical importance.

Aho et al. [ASU86] define the *time complexity* as the time needed by an algorithm expressed as a function of the input size of the problem; *asymptotic time complexity* as the limit behavior of the complexity as input size increases. Besides above definitions, analogous definitions have been made for *space complexity* and *asymptotic space complexity*.

Space complexity only concerns memory, e.g., insertion sort is an *in-place* sort while quicksort is not, which means no more memory is needed when insertion sort algorithm is run and the contrary for quicksort algorithm. In this thesis, we are only interested in execution time, so only *time complexity* is taken into account, and we will simply call it *complexity*. It is the complexity of an algorithm that determines the increase in problem size that can be achieved with an increase in computer speed.

From the definition, we can see that it has nothing to do with the hardware. Complexity is only linked to algorithms, not to particular instances of them, such as computer programs.

1.3.3 Program Execution Model

Many factors contribute to the runtime behavior of a parallel program, as the Figure 1.2 shows. In this figure, we can see that execution time can be influenced by lots of factors, notably instruction set architecture, compiler technology, CPU implementation and cache memory hierarchy. Each of these factors can have different impact on the different components of execution time. For example, compiler technology has big influence on instruction set architecture, CPU cycles per instruction and reference per instruction. While cache memory hierarchy has only impact on cycles per reference and cycle time. The particular characteristics of a program's execution result from the interaction of the program with components of the sequential or parallel computer on which it executes. In addition to the program itself, the compilers, operating system and underlying parallel hardware, etc. all have an impact on a program's execution time. The details of these interactions determine the values of the execution time for the program execution.

The value of a performance index is determined by conducting a performance experiment during which the measurements are collected. Before proceeding further, we define the performance model as a five-element tuple:

$$(S, N, P, D, E)$$

where:

$$\begin{aligned}
\begin{matrix}\textit{Execution}\\\textit{Time}\end{matrix} &= \begin{matrix}\textit{Instruction}\\\textit{Count}\end{matrix} \times \begin{matrix}\textit{Cycles per}\\\textit{Instruction}\end{matrix} \times \begin{matrix}\textit{Cycle}\\\textit{Time}\end{matrix}\\
&= \begin{matrix}\textit{Instruction}\\\textit{Count}\end{matrix} \times \left(\begin{matrix}\textit{CPU Cycles}\\\textit{per Instruction}\end{matrix} + \begin{matrix}\textit{Memory Cycles}\\\textit{per Instruction}\end{matrix}\right) \times \begin{matrix}\textit{Cycle}\\\textit{Time}\end{matrix}\\
&= \begin{matrix}\textit{Instruction}\\\textit{Count}\end{matrix} \times \left[\begin{matrix}\textit{CPU Cycles}\\\textit{per Instruction}\end{matrix} + \left(\begin{matrix}\textit{Reference}\\\textit{per Instruction}\end{matrix} \times \begin{matrix}\textit{Cycles per}\\\textit{Reference}\end{matrix}\right)\right] \times \begin{matrix}\textit{Cycle}\\\textit{Time}\end{matrix}
\end{aligned}$$

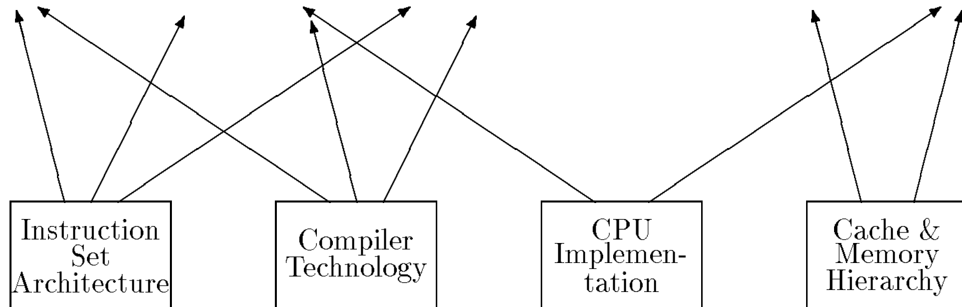


Figure 1.2: Influences on Performance

S: is the target system including a machine, especially its architecture, operating system and compiler.

N: is the number of processors of the machines.

P: is the program in question.

D: denotes the data needed for the execution of the program.

E: describes external factors such as user requests. Note that we do not consider Input/Output operations at all.

Of course, a program P is the most important factor because it defines the possible execution behavior. The domain of data values D decides executable cases among ones defined by the program. A target system S determines the time actually needed to run the program, and external factor E determines interference that happens during the program execution. So execution time of a program is determined as a function of the above five elements as follows:

$$T = \Gamma(S, N, P, D, E) \tag{1.2}$$

where $\Gamma()$ is a generic function to compute an execution time. So until now, if given the above five elements, we should *theoretically* be able to compute the execution time for a given input, with an interpreter.

Our purpose is to leave D out and to compute complexity C independently of any input as:

$$T = \Gamma(S, N, P, D, E) = C(S, N, P, E)(D) \tag{1.3}$$

1.3.4 Measurement Model

We now turn our attention to the actual process of measurement and how measurements are related to a performance index. The relationship between measurement and performance indexes is established by defining a measurement model.

We model abstract program executions as sequences of sets of states, σ. A sequence of state is associated with each program execution. Each state transition corresponds to the program performing a specific activity. For example, a state transition might be associated with the execution of a procedure. Program execution is modeled by 1) the sequence of states that a program is sent through and 2) the amount of time spent in each transition. A measurement can be made at each state transition. The value of a performance index is derived from the measurement sequence:

$$Md = (\sigma_1, s_1, t_1), (\sigma_2, s_2, t_2), \ldots, (\sigma_n, s_n, t_n) \tag{1.4}$$

where $\sigma_i \in \Sigma$ is i_{th} state of the program, s_i is the of i_{th} statement of the program and t_i is the amount of time the program spents in σ_i while executing statement s_i to move to state σ_{i+1}. Measurements of this type are said to be generated by a *state/duration* measurement model.

We notice that statement s_i along with its state σ_i will determine the state σ_{i+1} of the next statement s_{i+1}. We give a small example here:

```
      ......
10    N=10               s1
      ......
70    N=30               s2
80    DO 90 I = 1, N     s3
        ......
90    CONTINUE
      ......
```

Let us assume the statement 10 (`N = 30`) is s_1 and statement 70 (`N = 30`) is s_2 and DO-loop s_3 and N is intact between statement 10 and 70. If we evaluate separately the DO-loop, what is the upper bound of the loop? 10 or 30? The answer is 30 apparently because the statement s_2 modifies the value of N. This explains why the state is kept as well as statement in modeling the evaluation of Fortran code.

By the way, the statement could easily be made part of the state, but we choose to keep it explicit to make things easier to explain.

Certainly, each statement modifies the state of the next statement, but it can have different impact in terms of prediction. In the above example, if s_2 is `M = 30` instead of the `N = 30`, although statement s_2 does modify the state, it will have no impact on the execution time. In other words, s_2 itself won't influence the overall execution time.

If we want to know the execution time spent on this sequence, we can just add all of t_i together[1].

[1] We assume that no superscalar architecture is used. If it is not true, the combination of elementary instructions is much more complicated. See [Wang93] for more details.

$$T = \Sigma_{i=1}^{n} t_i \tag{1.5}$$

As you can see, if the sequence is a program or subroutine, T is the time spent on that program or subroutine.

1.3.5 Summary

In this section, we have formally defined the concepts of performance and complexity.

Since we are interested in input independent estimation, we choose the complexity rather than performance. Because our polynomial estimator can produce results like $N^2 + 2 \cdot N + 10$, it is complexity-like result. If the user gives the real value of the parameter, we can get the performance. Therefore, we see performance as a special case of the complexity. i.e., when all the parameters have the specific values.

We decided to use the measurement model in Equation (1.4) to approximate the function Γ in Equation (1.2).

From now on in this thesis, there is no difference between complexity, execution time estimation and performance evaluation.

1.4 Our Approach to Performance Estimation

In the previous section, we presented our evaluation model. Here we will explain how performance estimation is performed. We discuss several issues which must be explored to refine our proposal: (1) from which language level should we begin our experiment? (2) which method, static and/or dynamic, should we use to approximate the execution time? (3) the granularity to be considered? i.e., the atomic unit we want to time. (4) and how do we display our results?

1.4.1 Language Level Issue

Up to now, the language used to describe input programs has not been specified. Broadly speaking, performance estimation may be based on a high level programming language and have to factor-in compiler effects, or it may be based on assembly code to avoid compiler interferences.

Assembly Language Level

The benefit of assembly language analysis is that we do not need to consider the compiler any more since the analysis starts after the compilation. Any operations performed by a compiler, including code optimizations are irrelevant of our analysis. We can expect to get more precise estimation. e.g., you can see exactly how many cycles needed for an operation and how many memory accesses and/or register accesses. Therefore, in this approach, we can even analyze a program written in assembly language or in any high-level language.

But the disadvantages are significant too, mainly because of the gap between the programming and analysis levels.

- First, the independence with respect to the input language is replaced by a dependence on the target machine. Each machine uses a different assembly language. The assembly language code is much larger than the corresponding source program, Berthomier [Bert90] tested that it can be 12 times larger than the source code.
- Second, interpreting the assembly is difficult just as we generally use high-level language such as Fortran, C rather than assembly language. And due to the lack of correspondence between a source program and its compiled program, especially after code optimization, it is difficult to reason about the timing behavior of a program with analysis results.
- Third, a lot of information available at source language level cannot be used at this level. For example, the loop iteration count.
- Finally, the analysis is complex because an assembly program is less structured.

Source Language Level

Given all the drawbacks of assembly analysis, we decided that timing properties of a program should be analyzed at the source language level, where it is written, analyzed, debugged and maintained. As mentioned above, it is difficult to reason about a program with analysis results made at a different level. It is also awkward to analyze timing behavior and logical behavior at different levels. Source level analysis also gives a chance to exploit high-level information such as the complexity of an algorithm and inter-relations among statements or procedures. Frequently, this sort of information is decisive in predicting the execution time of a program.

The advantages of analysis at the source language level are obvious in a programming environment. Programs are usually well-structured, more algorithmic information is available, etc.. But the inherent problem of this level is that it is incompatible with some of the code optimization techniques of a compiler. Since an optimized compiler generates code without respecting source constructs, there may not be straight one-to-one relation between a source construct and its object code. In some cases, the sequence of the object code does not match with the sequence of the source program. As a result, it is difficult in general to predict the accurate code of a source construct in a program. We designed a set of experiments on SUN SPARCstation 2 to see how well we can statically cope with optimizations. Results are given in Chapter 5. More complex architecture have also been successfully modelized statically [Wang93].

The high-level language evaluator loses some information but possesses much smaller files, so it should run much faster than the low-level one.

Decision: High-Level Language

The assembly level solution is potentially more accurate. It is closer to the target machine. So with a detailed modelization of its various processing units, the evaluator should be able to predict elementary instruction duration more precisely. Furthermore, it occurs after all preprocessing,

compilations and optimizations that transform a high-level source code into an assembly-language one: the latter contains more information about the execution process than the former.

On the other hand, it is harder for a human to exploit the output of an assembly-language evaluator, for comparison or for interpretation, as we can think from a simple Fortran example.

As we said above, code optimization always causes problems for prediction, but fortunately, not all code optimizations are problematic. Any optimization can be handled, as long as it is made inside the boundary of a source construct. By adjusting the unit of a source level construct, we can cover most optimization effects. Our standpoint is that code optimization by a compiler should be used in a restrictive way.

Optimization makes execution efficient but prediction difficult. Assembly level analysis does not suffer this problem, but it has other problems. One practical solution is to disallow some problematic optimizations in favor of predictability. Many compilers provide a user with switchs that turn on/off optimizing options.

Based on our discussion, we choose high-level language.

1.4.2 Evaluation Method

The aim of our method is to get rid of static evaluation problems, while keeping the predictability capability of the symbolic expression of complexity. We discuss in this section different ways to compare and/or evaluate the Fortran source codes.

If the user wants to compare two versions of the same program, it is his responsibility to ensure that they are semantically equivalent. It is not our purpose to check it. Furthermore, we want to offer the possibility to compare the relative performance of completely different versions of code in the optic of helping task scheduling, as a side effect of the chosen solution.

Dynamic Execution Time Measurements

Instead of "differentially" comparing two programs, we separately evaluate each of them "absolutely", then compare the evaluations.

Unix provides tools for dynamic performance measurements. There are two main ways to collect profile data: *prof* and *tcov*. The first technique consists in periodic sampling of the program counter and deducing from its value the function currently running, its output is the percentage of time spent in each function. The other technique is to insert a counter before each instruction to peek in the program to analyze; all counters are written to a file when the program ends.

The Unix tool *tcov* produces a test coverage analysis and statement by statement profile of a C or Fortran program, if it is compiled with the proper option, with **-a** option on SUN SPARCstation 2. The output is the source file listing, each line of which has been prefixed with the number of times it was executed.

The program counter sampling method directly gives the performance of a program, function by function, but it requires to integrately be run on the target machine, which is often expensive. Of

course, we can obtain an approximation of the real timings by running and profiling the Fortran source code on another relatively cheaper machine such as a workstation, the more likely the two machines structures, the smaller the gap.

When *tcov* is used, the program can be run on any machines since we are only interested in the number of times instructions are executed instead of their duration. Those instructions numbers are the same on the two machines, provided that the internal number representations are identical; otherwise, loop ending with a convergence test might not iterate as many times in both cases. The problem of this method is that instead of giving a performance evaluation, it returns a list of counters which must be transformed in execution durations with the help of a machine's modelization. In the case of two slightly different program versions, one could base a comparison only on the interpretation of those counters; in the case of completely different tasks (for a scheduler) one must first convert them into execution times. *tcov* gives precious information about tests and loops.

Another tool: *prof* produces an execution profile of the source program. It can report the percentage of time spent executing between symbols as well as the number of times that routine was invoked and how much time per call. A simple floating-point comparison gives the time gained in each function.

The main drawback of runtime measurement is that it returns only numbers related to one input data set. If one wants to know the behavior of his program versus its main parameters (the size of data, or the number of processors for a parallel program), he must estimate the function with a number of such punctual measures. We will see that static evaluation can directly return the equation of the curve, when the program is not too complex and that the prediction closely matches the measurements in Chapter 5.

Static Symbolic Evaluation

We have said that the biggest advantage of static evaluation is its potential efficiency, compared to the dynamic evaluation. Its own complexity is a function of the program size, not of the program execution time.

But difficulties are obvious for a Fortran language static evaluator. There are a few situations that cannot be coped with.

- Branch Statement: When an IF statement is encountered, in most cases, it is impossible to approximate the average complexity of the whole instruction. We cannot do a better job than to assume that boolean is statistically equiprobable, i.e. equal probabilities, half-true and half-false.

  For some tests, which are used only to check the current conditions, or to detect anomalies, it is easy to detect because they lead to STOP, ABORT or EXIT statements. It should be possible to assume that the branch is not likely to be taken and to give it a 0 probability [Krall94].

  For some tests, the condition only depends on the surrounding loops induction variables and loop constants. It is then possible to exactly evaluate the taken/not taken probability [HP92].

  But most of the time, it cannot be calculated statically.

- Loop Iteration Count: A crucial point here is to correctly evaluate the loop iteration counts, for over nine-tenths of processing time is spent in loops. There exists a technique – *semantic analysis*, which can be used to exploit the information available for variables. With semantic analysis, symbolic variables can be evaluated; without semantic analysis, only constant loop ranges can be calculated, which is not interesting at all. Even with the semantic analysis, there are some problems left over.

A drawback of the static evaluation, compared to the dynamic one, is that it needs a precise modelization of the machine on which the analyzed code is supposed to run. Applied to a high-level language program, this method loses precision because of all the transformations made by the compiler, as explained before.

Because it is static method, no recursion is allowed unless some fixpoint algorithm is used on the call graph.

Combination of Static Estimation and Dynamic Measurement

This method runs in three steps.

1. It starts with a first pass of static evaluation to accumulate information about static failure points: locations of the IF-tests where the branch probabilities could not be computed (almost all), and the locations of DO-loops whose ranges were not exactly computed, so are those of WHILE-loops. As far as it is concerned with UNSTRUCTURED, PIPS provides a technique to convert the unstructured to several structured blocks. These are only counters needed to complete the static complexity evaluation.

2. In the second step, dynamic analysis is used to address these shortcomings detected by the first step. A copy of the program is made, where counters have been inserted at all places where a failure occurred during the first static pass. The modified program is executed for some input sets. At the end of the runs, the counters are written in a file that is exploited by the second pass of static evaluation. Some tool like tcov could be used.

3. The last step is to rerun the static evaluation using the counters generated by the second step to get the overall complexity of the program.

An advantage of this method is that the profiling execution can run on any machine (once again, having the same internal floating-point representation). As it requires much less profiling information than the dynamic approach, and because the output evaluation is parametric, its results are less sensitive to the choice of the data set provided for this particular sample run.

Furthermore, the only use of the first pass of complexity evaluation is to insert counters only where it is necessary, to gain time on the execution of the modified program. Actually, it may be more interesting to skip it and choose to measure every IF-test probability and every DO-loop range width.

Of course, just like the other approachs, there are some disadvantages:

- Since static estimation is used, no recursion is allowed unless some fixpoint algorithm is used on the call graph.
- Correctness cannot be proved. Assuming that program complexities are approximated by polynomials, it is obvious that such a complexity estimator cannot give good results for most algorithms. For instance, Quicksort and FFT complexities have a logarithmic component which won't be retrieved by the dynamic measurements.

It seems impossible to use multiple measurements and fitting approximations to guess missing coefficients in a large class of functions.

Decision: Purely Static Approximation

It seems impossible to combine safely dynamic exact information obtained for one input data set and approximate static information valid for any data set. Dynamic measurements might be useful to detect unlikely branches like error checking tests but these branches seem to be statically detectable according to Wang [Wang94]. Loop ranges obtained by measurements are unlikely to be useful, even if only scientific programs are analyzed. For instance, WHILE loops designed to reach some convergence criterion are very data dependent.

The use of special variables, pointing somehow to the origin of the estimation problem in the source code, in the static complexity function is much more user-friendly. The formulae can be proved correct and user information can easily be added when it is needed by replacing such special variables with the appropriate function.

Experiments are needed to see if the number of such variables is small enough to be usable. Encouraging results of such experiments are given in Chapter 5.

1.4.3 Mathematical Functions for Complexity

The choice of the mathematical objects to represent a complexity is a trade-off between many conflicting requirements.

1. On the one hand, this object must be complex enough to keep all the information we collect during evaluation;
2. On the other hand, it must be simple enough to allow basic computations like addition, multiplication and integration.
3. Besides, if one wants to automatize the comparison of program execution times, then two objects of this form must be comparable.
4. Furthermore, it must be adapted to its final use: if a human is to exploit it, the more readable, the better.

Because some of them are conflicting, we cannot find a means to satisfy all of them at the same time. What we are going to do is to find best trade-off. Here are some classes of functions we thought of for complexity estimations:

- The evaluation to a constant satisfies all requirements but the first: If the information you collect about a subroutine contains some parameters, i.e., symbolic constants, this one would fail.

- The general symbolic expression never loses information, but does not either allow general range integration because there is no formula to do it.

- The polynomial form (with constant exponents) allows range length integration, can give parametric execution times and involves easy computations. However, the fact that exponents are constants forbids exponential and logarithmic complexity expression.

- A pair of polynomials, one for the minimal execution time and one for the maximal execution time. Unfortunately, the minimal and maximal execution times do not carry much information. For instance, a WHILE loop may execute 0 times or forever. However Park [Park92] chose to give the range of prediction results.

- A conditional polynomial form. For example:

$$\begin{cases} N^2 + 10 \cdot N & \text{if } N > 10 \\ N^2 + 20 \cdot N & \text{if } N >= 10 \end{cases}$$

  would allow to include some test booleans in the complexity expression instead of roughly evaluating them. But this is not a practical form for integration between given polynomial bounds and its size may grow exponentially with the test and loop nesting depth.

Decision: Polynomial as Mathematical Function

Finally, polynomials were chosen because they are able to handle many scientific programs based on array computations. Special *unknown* variables are used to handle uncertainties.

1.4.4 Visualization of Complexity Formulae

Complexity formulae can be used directly by a compiler to choose between different compilation tactics but it may also be displayed to the user by a programming environment.

How to view the prediction results is a big challenge. How to present the results clearly, concisely is difficult too. Human being cannot exploit very complex formulae. The display must be correct, clear and short.

Decision: Comment Line Display

We decided to use the comment lines to present prediction results to avoid excessive development in the user interface. However, the average number of lines of the Perfect Club Benchmark is 4000 lines and it is doubtful that 4000 complexity formulae make much sense for a casual user. Some more work is needed in this area.

1.5 Contributions

During my Ph.D work, I have made the following contributions for evaluation tool in PIPS parallel programming environment. They are described in my dissertation with more or less emphasis.

I have implemented many new functionalities in PIPS and its complexity estimator:

Machine Description: The user can potentially choose from a set of machine descriptions called *cost tables*. So far three two cost tables are available. Two theoretical: one models all operations as unit cost; and the other only takes floating-point operation into account. One practical for SUN SPARCstation 2 which models all operations on the machine.

This is the real machine modelization. The machine is modeled by the cost table. The inherent cost of each operation on the machine is recorded in a file.

Symbolic Parameters: In some cases, the variables in the polynomial describing the complexity remain unevaluated. For example, PIPS identifies variables which must be kept unevaluated because they are free parameters of a module, e.g. formal parameters in a subroutine call or dynamic input data. In addition, the user can specify that certain variables remain unevaluated.

Interprocedural Preconditions: PIPS uses preconditions, that is information acquired before the current statement, to resolve analysis problems. As it is obtained, precondition information is stored in a library for later use.

Simplified Results: An analysis is performed to minimize the number of free parameters which are local to the module. In addition, intermediate variables are removed from final result after use.

Program Tracing: The program may be traced while the complexity evaluation proceeds. This tracing facility serves two purposes: (1) clear display of the program structure, e.g., whether the program is well structured or how it is structured, and (2) PIPS debugging, when something goes wrong, we can easily see where the problem is.

I have also validated my approach with several kinds of experiments:

Sequential Machines: Our estimator generates very good complexity results by comparing with the result obtained manually by Nahid Emad [Emad91]. The difference is within the expectation. We have executed our estimator on a benchmark program FLO52 from PERFECT Club Benchmark Suite to show our complexity results. Then, we timed a subroutine **EFLUX** from FLO52 on SUN SPARCstation 2, and compare it with complexity result obtained by our estimator. The time difference is pretty small within 5 %, despite the large ranges of the variables.

Parallel Machines: We have carried out several experiments with different sample tests on a shared-memory multi-processor MIMD machine — BBN TC2000. We have found out that the scheme designed for sequential machines can be easily expanded to such parallel machines, with only minor modifications of the algorithm. Meanwhile, several empirical algorithms have been extracted from the test results and proven to be correct by the experimentation.

1.6 Dissertation Outline

Chapter 1 is a general introduction to the thesis. It lists the motivations for this work, the research goals as they were defined and the contributions made. The rest of the Dissertation is organized as follows.

Chapter 2 talks about general concepts of the performance and complexity. It paves the way for later explanation of Complexity in PIPS. It states what we think about performance and what we could do to predict the complexity. Chapter 2 contains a description of the algorithms used to estimate the execution time of programs for any input. This work is rooted in the PIPS project. PIPS goals, general design and data structures are presented first because they had a very strong influence on the complexity algorithms design. These algorithms cannot be understood without some knowledge about PIPS. Then the choice of a two-step approach for execution time approximation is justified. The first step is used to hide the low-level details of the processor behavior (compiler effects, operating system bias, special hardware, e.g. cache memory, etc...). The second step is used to approximate the dynamic behavior of the program execution, simplified by the first step, by a static mathematical representation. In other words, the first step converts the actual execution of the program for a given input into a sequence of commands which belong to a finite set and which take a constant amount of time to execute. The second step summarizes all these ideal traces for all possible inputs in a mathematical formula. The second step can be proved correct. The first step can only be experimentally validated. This is exactly the purpose of Chapter 3. It also gives some proofs for the algorithms. Symbolic calculation algorithm is also introduced in the chapter.

Chapter 3 deals with target machine model. How and how well can the behavior of a machine and its environment (compiler, run-time, operating system, microprocessor,...) be characterized by a set of basic times? Sequential machines are studied first from a theoretical point of view. We will enumerate some parameters that could reasonably be part of a machine model, then keep only the most important ones for a simple implementation and use. We then extend the model to parallel machines. In a third section, two very simple time tables, called *cost tables*, are presented. The last and most important section presents experiments carried out to validate the approximation of a machine by a finite set of elementary times and shows how a SUN SPARCstation 2 workstation can be precisely approximated. But we delay the real time verification to Chapter 5.

Chapter 4 explains how complexity programs work. It contains a detailed description of the implementation performed in PIPS. The PIPS programming environment is presented first. Then the data structures specifically developed for the complexity estimator are shown, as well as PIPS data structures directly used by the estimator, i.e. the prerequisites for complexity, including call graph, interprocedural control flow graph, preconditions and effects. The relevant C files are also listed. The user interfaces, as well as parameters and options of the complexity estimator, as it is implemented in PIPS.

Chapter 5 talks about the experimentation done on sequential machines, mainly on SUN SPARCstation 2. In this chapter, we are going to show that our estimator can generally generate good results. The experimentation was used to validate the approach and the algorithm on real programs. We present comparisons between results obtained with our prototype and results manually obtained by other researchers. We also checked the robustness of the approach on a Perfect Club benchmark and compared the predicted and measured execution times of one of its module for a large range of input values. Very encouraging results are obtained for sequential machines. As

expected, an automatic estimator does not make small mistakes a human would make, but it does require some user control as simplifying hypotheses about branch probabilities may lead to very bad approximations. Of course, the estimator provides information about the amount of guessing performed. Then, in order to validate the result, we will time a subroutine *EFLUX* from FLO52, and compare it with complexity result obtained by our estimator. We will show that the time difference is pretty small, despite the large ranges of two variables.

Chapter 6 is about our experiments on a real parallel machine BBN TC2000. The final goal of our work was to develop an estimator for parallel machines. Chapter 6 is a first step. It contains a description of experiments carried out on a BBN TC2000 to characterize the behavior of parallel loops on a parallel machine. In order to validate our model of supercomputer, we need to have a look at what is really happened in parallel machine. We have made a bunch of test programs to extract the empirical formulae about how to predict the execution time on the parallel machine. We show how and how well experimental results can be modeled by very simple equations. These equations could be used in the future to extend our sequential estimator into a parallel estimator.

Chapter 7 is the conclusion of the thesis. It first summarizes PIPS complexity estimator by listing its advantages and drawbacks. It then surveys related works in the field of performance evaluation. We elaborate the most common techniques applied in performance prediction tools. We distinguish four approaches: manual evaluation, simulation, analytical and measurement. We classify the existing performance estimators according to the nature of the target machines: single processor machine, general multiprocessor machine (can be both shared or distributed), shared memory multiprocessor machine and distributed memory multiprocessor machine, and we compare them with our own contribution. Finally, we give some future works envisioned.

Appendix A contains the Intermediate Representation of programs used by PIPS.

Appendix B contains the elementary C times for a SUN SPARCstation 2 as well as some explanations. These times were used in Chapter 5.

Appendix C contains the whole stuff of CALCG subroutine from TMINES program.

Chapter 2

Complexity Estimation in PIPS

The complexity estimator described in this dissertation is based on the PIPS[1] environment. Some information about this environment is needed to understand how complexity estimation algorithms work. Let us first have a look at PIPS so as to help understand complexity in PIPS. And then we will provide the algorithms for calculating complexity and prove them correct. Finally, we will mention the different approaches we have tried to generalize the complexity, though they are all dead-ends. Symbolic calculation algorithm is also introduced.

2.1 PIPS Overview

In this section, we present PIPS: its motivation, general design, data structures and semantical analysis. The later three components are used to handle procedure calls, to exploit control flow information recursively and to compute loop ranges or conditional expressions with symbolic information.

2.1.1 Motivation for Automatic Parallelization

There are many scientific programs which are currently running all over the world, most of them are Fortran programs. As the supercomputers have found their ways into industrial and academic centers, a problem rises: the old sequential programs are becoming obsolete.

For the economical reasons, rewriting these programs is completely out of the question:

- That will cost too much;
- There would be a need to rewrite them again each time a new machine or a new environment is introduced.

[1] PIPS is French acronym which stands for Paralléliseur Interprocédural de Programmes Scientifiques, developed at Centre de Recherche en Informatique (CRI) de l'Ecole Nationale Supérieure des Mines de Paris (ENSMP).

The appearance of new languages and new supercomputers makes old scientific programs perform very poorly in comparison with what could be obtained by a complete rewriting. Besides this effective but expensive method, end users who usually have neither time nor enough knowledge to rewrite, prefer to submit their program to an automatic program restructurer, especially when the goal is to run the application on a parallel computers. So this leads researchers to design new techniques to transform old scientific programs to fit new environments. The techniques are supposed to hide the complexity to the programmer and to provide portability.

PIPS is an attempt to address this problem. PIPS is a source-to-source compiler, i.e. from old-fashioned programs to appropriate ones suitable to be run on parallel machines with different structures, e.g., shared or distributed memory.

2.1.2 PIPS Historical Goals

Detecting the maximum level of parallelism in sequential programs requires a thorough understanding of their behavior. Although numerous vectoring and parallelizing compilers for scientific programs exist in both academic (PTRAN, ParaScope, ...) and industrial (VAST, FORGE) worlds, few are able to address the question of parallelism detection on a global basis, i.e. coupled with interprocedural information and comprehensive semantical analysis. The main goal of the PIPS project is to address this very issue and explore its effectiveness on real programs written in a real language.

PIPS is a source-to-source parallelizing compiler that transforms Fortran 77 programs by replacing parallelizable nests of sequential DO loops with either Fortran 90 vector instructions or DOALL constructs. It is not targeted towards any particular supercomputer, although only shared memory machines have been considered. The principal characteristics of PIPS are:

1. Interprocedural parallelization. It is at the core of PIPS: every step of the parallelizing process is able to cope with interprocedural information. A striking example of this drastic point of view is the parser, which encodes every assignment statement as a function call to a built-in "=" procedure with two arguments, the right and left hand side expression. There is no assign node in the abstract syntax tree of programs and a user `CALL` would be encoded in the same way.

2. Interprocedural analysis. To be most effective, a parallelizing compiler needs to gather as comprehensive as possible information about the behavior of program. This is done in PIPS by a set of sophisticated interprocedural semantical analysis phases that compute side-effect (SDFI), regions and predicates.

3. Relative efficiency. PIPS is fast enough to be used on real-life programs. PIPS has been tested on several standard benchmarks such as the PERFECT Club.

See [Baron90, IJ91, IJT91] for additional information.

2.1.3 PIPS General Design

Since PIPS is mainly a research project, its architecture is kept to be modular and evolutive enough to adapt to the wide variety of people and to the requirements of interprocedural analysis and interactivity. This decides a structure in *phases*, all of which manipulate a common intermediate representation (IR) of programs that has been carefully designed. Although this architecture is hardly new, a major issue is then the determination of the order according to which these different phases of the parallelizing process have to be scheduled to maintain consistency. PIPS adopts a relatively uncommon approach to this problem; the ordering is demand-driven, dynamic and automatically deduced from the declarative specifications of the dependencies among phases, by a program called *pipsmake*. Users are allowed to send requests to *pipsmake*, either to get a particular data structure or to apply a specific function.

All data structures used, produced or transformed by a phase in PIPS, such as the abstract syntax tree (AST), the control flow graph, etc. are considered to be *resources* which are managed by a database manager, called *pipsdbm*. *pipsdbm* can optimize the transfers according to the availability of current memory. Some data structures destroyed in memory by side-effect of some phases can be retrieved if they are still consistent. That means the user can make full use of the available information, not necessarily from the very beginning.

Any phase, called by *pipsmake*, begins by requesting some resources, via the `db_get` function provided by *pipsdbm*, performs some computation and declares the availability of its result via `db_put`. Every data structure is linked to its associated program unit.

Every phase in PIPS, such as parsing of a source program, denotes a function; it may use and/or produce some resources. Every phase that exists in PIPS is declared by production rules; these are stored in a configuration file *pipsmake.rc* which is used by *pipsmake* to schedule the execution of each function, according to what the user and each phase (recursively) request. We give an excerpt of this configuration file about the complexity in PIPS:

```
complexities > MODULE.complexities
        < PROGRAM.entities
        < MODULE.code MODULE.preconditions
        < CALLEES.summary_complexity

summary_complexity > MODULE.summary_complexity
        < PROGRAM.entities
        < MODULE.code MODULE.complexities
```

Here, for instance, the phase `complexities` computes the approximation of statement complexities (execution time) for all statements of the `MODULE`. (A `MODULE` in PIPS could be a Fortran program, subroutine and/or function.) It needs the definition of all the entities of the `PROGRAM`, the code of the `MODULE`, the preconditions of the `MODULE` and summary complexity of its `CALLEES`. (All modules that could be called by the `MODULE`). The output of phase `complexities` is the final estimation, i.e., summation of all the statement complexities of the corresponding `MODULE`. For the `summary_complexity`, it computes the approximation of a module complexity (execution time), i.e., the summation of module complexities.

This architecture is incremental, since data structures are computed only when needed by some

phase; and it is flexible because each phase does not have to worry about the others and it only needs to request resources. This organization permits the interproceduralitу to be almost transparent, so if a phase is to be produced, the programmer can neglect how resources are computed, leave all production work to *pipsmake*.

This prototype is exploited in the complexity estimator to cope almost effortlessly with procedure calls.

2.1.4 PIPS Data Structures

PIPS is organized around a core data structure that implements the abstract syntax tree (AST) of Fortran 77. Since Fortran 77 is a little bit outmoded, the Intermediate Representation (IR) in PIPS (see Appendix A) is carefully designed to avoid sticking too closely to Fortran 77 idiosyncrasies.

The data structure that implement the IR used in PIPS are defined with the software engineering tool NewGen [JT89], developed by Pierre Jouvelot and Rémi Triolet. NewGen allows the definition of data *domains* in terms of basic ones, like `integer` or `float`. From these definitions, expressed in a simple language, NewGen generates a set of functions on objects of these domains.

The following is the definition of `expression` used in PIPS:

```
expression = syntax x normalized ;
syntax = reference + range + call ;
normalized = linear:Pvecteur + complex:unit ;
reference = variable:entity x indices:expression* ;
range = lower:expression x upper:expression x increment:expression ;
call = function:entity x arguments:expression* ;
```

Here, we can see that a node for a `call` expression, representing a Fortran 77 `SUBROUTINE` or `FUNCTION` call, includes (1) the `function` entity and (2) the list of `arguments` expression. For `syntax`, it can be a `reference`, or a `range` or a `call`; the `syntax` is only one of these three at a time. For `range`, it consists of three kinds of expressions: lower, upper and increment. In PIPS, an assignment is encoded in the way that it is treated as a function call to = and it has two arguments denoting the left hand side and right hand side of the assignment. This is possible since the implicit call mechanism is “call-by-reference”.

The data structures used by the complexity estimator are based partly on NewGen and partly on data structures defined for an linear algebra library, called C3 libraries.

2.1.5 Control Graph

The control flow graph is implicitly defined by the abstract syntax tree when the code is structured. An explicit control flow graph is only used for unstructured code. Structured statements are either basic commands named calls altering the store, sequences, tests or DO loops. The NewGen declaration, slightly simplified for the purpose of exposition, is:

```
statement = label:entity x number:int x ordering:int x comments:string x instruction ;

instruction = block:statement* + test + loop + goto:statement + call + unstructured ;
test = condition:expression x true:statement x false:statement ;
loop = index:entity x range x body:statement x label:entity x execution
       x locals:entity* ;

unstructured = control x exit:control ;
control = statement x predecessors:control* x successors:control* ;
```

All the analysis phases are defined by induction on the `statement` domain. For example, the code of a subroutine is a `statement`, which contains an `instruction`. This `instruction` could be a `block`, in this case, we go back to the `statement` domain because `block` is defined as a list of `statements`; or it could be a `test`, which is a IF statement; or it could be `loop`, which is iteration statement; or it could be `call`, which is an invocation of a subprogram. In all these cases, as you can see clearly, we go back to `statement` domain recursively. But if this `instruction` is `unstructured`, it will cause some problems because the control flow graph has to be followed using the pointers in control. But finally, as `unstructured` is defined as a list of `control`, we can go back to `statement` too.

We have developed a technique to display the program structures, interested readers will find more information in Section 4.7.4.

The complexity is recursively estimated on the hierarchical control flow graph like other analyses. It is computed bottom-up by aggregating complexities of components in a more global complexity.

2.1.6 Effects

Effects describe the memory operation performed by a given statement. Besides the reference on which the operation is performed and its kind (read or write), PIPS distinguishes between effects that are always performed (MUST) and the ones that may be performed (MAY).

Proper effects are memory references local to a given statement, such as a write on the left-hand side of an assignment or on the index variable of a DO loop construct. Cumulated effects take into account all effects of a given statement, including those of its substatements. (Recall that abstract syntax tree is recursive in PIPS, cf. Appendix A).

The proper effects of a call statement are derived from the cumulated effects of the body of the callee. Proper and cumulated effects are computed by an automatic bottom-up scan of the call tree, as can be seen in the following excerpt of the production rules for effects:

```
proper_effects                  > MODULE.proper_effects
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.summary_effects

cumulated_effects               > MODULE.cumulated_effects
        < PROGRAM.entities
        < MODULE.code MODULE.proper_effects
```

```
summary_effects                 > MODULE.summary_effects
        < PROGRAM.entities
        < MODULE.code MODULE.cumulated_effects
```

The effects are used to decide how to simplify the result of complexity and which symbolic variable should be dropped or could be kept. An advanced option is provided to let the user override the effect information and force the use of a symbolic variable in complexity formulae. See Section 4.6.1 for more information.

2.1.7 Transformers

Although these transformers are not used directly in complexity evaluation, they are used directly by preconditions, which is to be introduced later. Hence transformers are briefly introduced here for the unique purpose of completeness. We give the production rule in the following:

```
transformers_inter_full         > MODULE.transformers
        < PROGRAM.entities
        < MODULE.code MODULE.cumulated_effects MODULE.summary_effects
        < CALLEES.summary_transformer

summary_transformer             > MODULE.summary_transformer
        < PROGRAM.entities
        < MODULE.transformers MODULE.summary_effects
```

Effects, code and entities, which are introduced in the earlier sections, are required to compute transformers. So we suppose that transformers are available when we need in preconditions. Transformers could have been used instead of preconditions to introduce fewer unknown variables.

2.1.8 Preconditions

Precondition is information about the store holding before a statement is executed. Only scalar integer variables are analyzed but they fortunately are the most useful ones for complexity estimation. We give the production rule in the following:

```
preconditions_inter_full        > MODULE.preconditions
        < PROGRAM.entities
        < MODULE.cumulated_effects
        < MODULE.transformers
        < MODULE.summary_precondition

summary_precondition            > MODULE.summary_precondition
        < PROGRAM.entities
        < CALLERS.preconditions
```

All the required components are computed on a demand-driven basis before at pipsmake request. So preconditions are at our disposal by simply declaring them as inputs in pipsmake rules about complexity estimation.

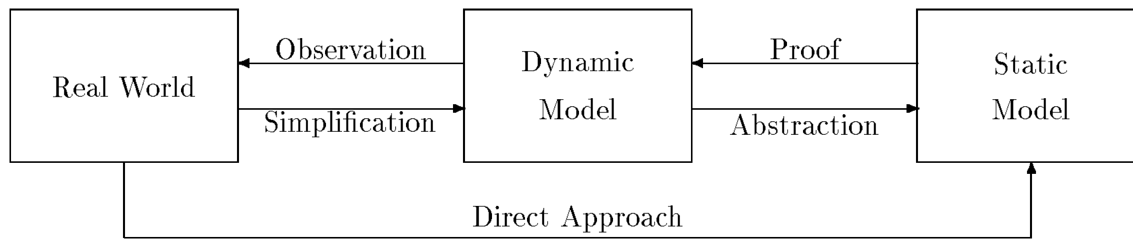


Figure 2.1: Two-Step Approach

2.1.9 Summary of PIPS

In this section, we have introduced the PIPS programming environment and some underlying elements for PIPS complexity which are all required by complexity evaluation in PIPS. In next section, we will introduce the very part of PIPS complexity.

2.2 Complexity Model

When a program is run on a real machine, i.e., in the real world, we might think that there exists a model that corresponds exactly to the execution of the program. For example, the model should be able to predict how long the execution lasts. Or what the probability of each branch statement is. Practically speaking, we can calculate everything we need to know about the program, but it is only after the execution of the program for one given input. We call this model which corresponds to the execution of the program *real world model* or *execution model*.

In order to calculate the complexity of the program statically for all its possible input, we need to firstly set up a dynamic model that could generally correspond to the *real world model*; and secondly, set up a static model that could be used to make the approximation to the dynamic model.

We chose to make a two-step approach to break the difficulty: (1) an approximation from the real world model to the dynamic model, and (2) an approximation from the dynamic model to the static model. The two corresponding reverse arrows correspond to validation phases. Some experimental observations are needed to show that the simplifications made are acceptable. A mathematical proof of consistency between the static and dynamic models is also desirable. Figure 2.1 illustrates the relationship among the three models and two-step approach.

Before we set up our models, we introduce several definitions:

Definition 2.1 *Execution Time Function T(S): This is a mapping from the a general statement to its execution time on a specific machine for a given environment σ.*

For a simple basic language construct S, $T(S)$ is the specific number of time units required to execute S on a specific machine. For example, on SUN SPARCstation 2, the Fortran floating-point addition statement `S: f1 = f2 + f3` is completed in 42 clicks (0.7 μsec), so we think that on

SUN SPARCstation 2:

$$T(S) = T(f1 = f2 + f3) = 0.7$$

Note that $T()$ is a function of not only the statement, but also the current environment σ of the statement S, though above statement has nothing to do with its environment[2]. Imagine that the statement is a DO-loop with a variable upper bound N. So the execution time of statement S under the environment σ can be denoted as $T(S) \cdot \vec{\sigma}$. We also use $T(S)$ to represent the execution time spent in S when the context is clear.

Before we introduce our complexity function $C()$, we need to define a special variable set, called Unknown Variable Set.

Definition 2.2 *Unknown Variable Set U: This is a set of variables whose values are needed in the evaluation procedure and cannot be predicted statically. These variables are not bound to the environment. They can take any value in their domain.*

Generally, when we encounter the loop bounds which depend on the dynamic input data, such an unknown variable is introduced in the complexity formulae.

Definition 2.3 *Complexity Function $C(S)$: This is a mapping from the high-level construct to its complexity c. The complexity c is a mapping from the initial store σ to the execution time of S*

$C()$ is not usually a number. It can include some symbolic variables. For example, a loop construct on the SUN SPARCstation 2:

```
      DO 10 I = 1, N
            f1 = f2 + f3
10    CONTINUE
```

We can consider that the complexity for this construct is:

$$C(DO) = 0.7 \cdot N + overhead$$

Note that $C()$ is a little more complicated than $T()$ since $C()$ is not only a function of a statement and its variables, but also of a set of unknown variables. We chose to keep them among polynomials.

Take the above example, $C(DO) = 0.7 \cdot N + overhead$, if N happens to be a variable which we want to keep, e.g., dummy argument, outer loop induction variable and etc. We are allowed to keep because only its value in the initial store is needed, we do not need to evaluate N to its effective value. We just keep it as it is. But if N is impossible to be evaluated, e.g., it is a random number or simple input datum, N cannot be known statically. The worse case is that we need to know the real value right now, we will be stuck here. We have two solutions: (1) keep the unevaluatable variable as it is, or (2) represent it with a symbol, called UNKNOWN VARIABLE, chosen in unknown variable set.

The first solution is dangerous because we may encounter many these variables in a module, or same variable with different values; and sometimes, keeping variable literally is not always possible because of our polynomial expression, as shown in the following:

[2] This is an approximation. IEEE NaN values may change the execution time, as well as cache or register effects.

```
S1                      DO 10 I = 1, 2**N
                             Body
                     10 CONTINUE
```

Here, N cannot be used in the complexity formulae associated to $S2$ since its value in the initial store is irrelevant:

```
S2                      READ *, N
                        DO 10 I = 1, N
                             f1 = f2 + f3
                     10 CONTINUE
```

Here, N can be used in the complexity formulae associated to $S3$ because its value in the initial store is relevant for the first loop. However, an unknown value symbol is also needed for the second loop:

```
S3                      DO 10 I = 1, N
                             f1 = f2 + f3
                     10 CONTINUE
                        READ *, N
                        DO 10 I = 1, N
                             f1 = f2 + f3
                     10 CONTINUE
```

The second solution is not very appealing since formulae with undefined values are almost meaningless at first glance. However, if such variables are not used too often and if the user is given some way to relate each unknown variable to a particular code construct or variable, such formulae may convey useful information. Also, unknown variables have to be introduced to prove the correctness of our scheme.

$C()$ is a function of statement which produces a function of the unknown variable set U and of the current environment σ. The complexity of statement S, under environment σ and unknown variable $\vec{u}$ is denoted as $C(S)\cdot(\vec{\sigma},\vec{u})$. We also use $C(S)$ and/or $C(S)\cdot\vec{\sigma}$ to represent the complexity of statement S when no ambiguity arises because no unknown values are required.

2.2.1 Dynamic Model

This is the approximation of the real world made to hide some issues like operating system or cache interferences. The dynamic model is used to prove the correctness of the static model. We introduce two new definitions here:

Definition 2.4 *Let Begin(S) be the Beginning Time of statement S for a given execution of S on a given machine.*

Definition 2.5 *Let End(S) be the Ending Time of statement S.*

Both are specific numbers, which refer to a fixed past time, like C library function time(3v). We have made the following assumptions from observation:

1. A program is a sequence of statements. The statement is run one after the other.
   - No latency between two statements. i.e., $S1$;$S2$ means that $S2$ will be executed immediately after $S1$ finishes.
     Begin($S2$) $<=$ End($S1$)
   - No overlapping between two statements. i.e., $S1$;$S2$ means that $S2$ will be executed only after $S1$ finishes.
     Begin($S2$) $>=$ End($S1$)
2. The execution of a statement is determined by two factors: language semantics and its implementation.

From item 1, we can think that two consecutive statements $S1$ and $S2$ can be predicted as the sum of the prediction for each statement as following:

$$T(S1; S2) = T(S1) + T(S2)$$

Since the boundaries are well defined, there exists a clear-cut between the two statements, albeit in practice, it is difficult or sometimes highly unlikely to get once an optimizer has scrambled machine instructions.

To predict the execution time of a statement, we depend on the item 2.

A timing schema for a high-level language construct is a formula computing an execution time of the construct. The formula describes how an execution of a statement is computed with the execution time of its components. For example, for an assignment statement: `S: var = exp` ; its timing schema can be interpreted as following:

$$T(S) = T(var) + T(exp) + T(=)$$

where $T(var)$, $T(exp)$ and $T(=)$ is execution time of the statement component `var`, `exp` and `=` respectively.

In summary, a real program is considered to be composed of a sequence of statements, although the number is not definite. There exist in practice some constructs like tests, loops and unstructured that they can simple generate sequences of statements *dynamically*, but statically, they will bump into problems because the number of the statements cannot be determined statically. In order to solve this problem, we have to decide these unknowns artificially, although it can cause big deviation in performance evaluation, i.e., probability is fifty-fifty for tests.

2.2.2 Static Model

Static model means that the program is analyzed statically and that approximations have to be made for tests and fix-points (loops and unstructured) and I/Os, so as to cope with any possible input data.

1. A program is a sequence of statements. The complexity of the program is the sum of the complexities of the statements associated with its adequate state. We assume that the number of statements is definite, which is the difference between dynamic model and static one.

2. Each statement consists of a number of atomic operations. (can also be called primitives) such as $+, -$. Each atomic operation is assigned a cost. The complexity of the statement is the sum of the costs of all the atomic operations.

   That is, each statement can be divided convergently into a number of atomic operations (primitives)

The item 1 implies that a statement is a good unit in decomposing a program and/or subroutine in question. In other words, the complexity of the module (program, subroutine or function in Fortran) can be predicted as the summation of predictions of its statements. We can think that two consecutive statements $S1$ and $S2$ can be predicted as the sum of the complexities for each statement as following:

$$C(S1; S2) = C(S1) + C(S2)$$

The item 2 describes how a complexity of a statement is computed with the complexities of its components. For example, for an assignment statement: `S: var = exp` ; its complexity can be calculated as following:

$$C(S) = C(var) + C(exp) + C(=)$$

where $C(var)$, $C(exp)$ and $C(=)$ is complexity of the statement component `var`, `exp` and `=` respectively.

In other words, the complexity of a complex statement is estimated recursively according to its structure, as it is defined in PIPS intermediate representation. See Appendix A for more.

2.2.3 Problem with the Dynamic Model

From now on, we will stick to our set-up static model to evaluate the program. But there is a big gap between the two theoretical models, let alone the real world model. Let us examine in more detail the approximations made between the real world and the dynamic model.

In our dynamic model, we can see that the execution time of the two instructions $S1$ and $S2$ below:

```
S1: i1 = i2 + i3
S2: i1 = i1 + i2
```

should be same. There are two fetches, one add and one write. But the experiment result shows that they are not same at all: $S1$ takes about 0.18 μsec on SUN SPARCstation 2 while $S2$ only needs 0.10 μsec. It is simply because `i1` is already in the memory, very likely in register, so $S2$ needs less time than $S1$.

This is the kind of problem we cannot tackle with in our model. Fortunately, we only need an approximately accurate execution time, we can calibrate our time table a little bit to make the whole estimation accurate enough.

2.2.4 Model Summary

In this section, we have introduced how we gradually obtained the static model from the real world model by approximation. Although we know that there are some intrinsic problems in our static model. From now on, we will stick to our set-up static model whenever we need to predict the execution time of program statically.

In the next two sections, we will talk about the complexity algorithms.

2.3 Complexity Algorithm for Structured Sequential Programs

Now we assume that we know how to compute elementary instruction execution times with the dynamic models. Their associated complexities are not data dependent but constant. Let us combine them to evaluate the complexity of structured sets of instructions. We use the word *statement* in the general, recursive sense of PIPS: a statement is either a block of statements, a test, a loop, a function call, or an unstructured control flow graph of statements. For example, a whole DO loop is considered as one statement, whereas the Fortran instruction DO I=1,N is not.

Now we are considering complexity of structured program, which means the program contains neither `GOTO` statement to jump out nor statements where `GOTO` statement can jump in. Let us describe the bottom-up recursive analysis.

2.3.1 Complexity for Primitives

We initially started the evaluation of the complexity in terms of floating-point operations; then it gradually appeared that integer operations could be equally important, and also that after all, memory accesses could as well be the real bottlenecks, so we decided to count them all.

Cost of Fortran predefined functions

The execution time evaluation is performed bottom-up, cumulating elementary operation duration or “costs”. Cost is misleading word but it has been kept for historical reasons. A cost is assigned to each predefined function, that may be:

- a numerical operation: $+, -, *, EXP, MOD, .LT...., T(+) = C(+)$
- a logical operation: $.AND..NOT.$
- a memory read: $N, I.$
- a memory write with assignment “=”

The user must provide the cost table associated with his target machine dynamic model. He specifies the cost file directory using a special properties (See Section 4.6.2 for more information) if the machine model exists. Else he should build files describing a new dynamic model.

Overloaded Operators

The EXP operation hides at least three different functions according to the type of the operand which can be floating-point real, double-precision, complex. The complex one: $e^{a+ib} = e^{a}(cosb + i * sinb)$ can be five times longer than the simple floating-point one, as there is a sin, a cos and an exponentiation and two multiplications to perform instead of just an exponentiation. Thus, for a particular operation, the cost table contains an entry for each argument type: integer, floating-point, double, complex, double-complex. According to the Fortran semantics, an operation that has two arguments of different types is given the cost associated with the "heaviest" type, even though in some cases, it may be wrong: the sum of an integer and a complex number may require only a single precision floating-point addition.

Cost of Memory Accesses

Our dynamic target machine model includes neither cache nor virtual memory, nor does it account for registers. Accordingly, a memory access has a unique cost for each data type (cf. Section 3.5.3), we have $T(=) = C(=)$. However, we distinguish the memory read and the memory write. The former is used for every variable name which appears in the right-hand side of assignment instruction "=", in function calls to arguments, or in ranges of loops: more generally, in every expression. The latter is used for every variable name that appears in the left-hand side of the assignment. Experiments on a SUN SPARCstation 2, presented in Chapter 5, show that this simplification is acceptable.

Cost of Array Element Address Computation

When the variable name is an array name followed by indices, additional time is consumed to compute the address of the array element accessed; namely, integer multiplications and additions. This time should also be accounted for with the chosen machine model. Although this depends on the code optimizer very much (strength reduction, common subexpression computation, ...), we found out that a finite set of elementary "instructions" is sufficient to model the combination of compiler, optimizer, runtime and operating system on the execution time. We distinguish array access costs among locally declared, globally declared and formally passed. See Section 3.5.5 for information about these differences on SUN SPARCstation 2.

2.3.2 Complexity of an Expression

To know the cost of an operator, one must first know the types of its arguments to determine which kind of operation takes place, then find the relative cost for the operator in the cost table *operation*.

The evaluation of an expression cost is complicated by the overloading of most arithmetic operators. To know the cost of an addition, one must first know the type of its arguments, because the floating-point operation costs more than that of integer. So we evaluate the expression bottom-up, beginning with the leafs of the syntax tree (constants, variables or function calls) whose types are known. The types of sub-expressions are propagated upwards towards the root. For example, $IM * JM$

will have cost of 3 according to a uniform all-one cost table: 2 units for memory reads and 1 unit for multiplication operator.

2.3.3 Complexity of a Statement

We have different complexity constructors for different statement constructors. According to control graph introduced in Section 2.1.5, statement complexity is the instruction complexity, which is one of block, test, loop, call and etc. They will be introduced separately in the following.

Complexity of a Call

The complexity of a call is exactly the summary complexity of the corresponding subroutine or function translated into the environment of the caller, plus the calling overhead. Fortunately, Fortran is a language using call-by-reference to pass the arguments, so we do not need to know the argument types, since only their addresses are involved. So we can think that the overhead of calling procedures depends only upon the number of arguments. As the number of arguments increases, the overhead augments proportionally. We will discuss this assumption in Section 3.5.5 and show its limit.

Complexity of an assignment

In PIPS intermediate representation, an assignment is a simple case of call We refer to it as *expression* = *expression*, so the total complexity equals the sum of the two expressions, because the value obtained for the left-hand side is produced in a register that has just to be store. For example, the following assignment has the cost 5.

```
C                                                                5 (STMT)
              JJ = I+J-2
```

There are two memory reads, one plus operator, one minus operator and one memory write, so the above statement has constant complexity of 5. In PIPS, the assignment is considered as a call. Because the memory access has been charged same cost, the assignment is considered to have zero cost.

2.3.4 Complexity of a Structured IF

Although IF construct is one of statement, we prefer to give it a full subsection because of its importance in execution time prediction.

$$\text{IF } cond \text{ THEN } S_{true} \text{ ELSE } S_{false} \text{ ENDIF}$$

This statement is structured if there is no `GOTO` jumping in or out of S_{true} and/or S_{false}. Let us call p the probability that *cond* is true, q the probability that it is false, $p+q=1$. We assume that

the overheads for both branches are equal, because there will be two assembly instructions less in evaluating the condition but two more after the true branch statements in true case than in false case. See below the assembly cost produced by the GNUS gcc compiler for a SUN SPARCstation 2:

```
        Evaluation of condition
        bl      L_True
        nop
        b       L_False
        nop
L_True:
        True Branch Statements
        b       L_End
        nop
L_False:
        False Branch Statements
L_End:
```

We use the following probabilistic definition of the complexity:

$$C(IF...) = C(cond) + p \cdot C(S_{true}) + q \cdot C(S_{false}) + C_{ovhd}$$

The correctness of such a complexity formula with respect to the execution time $T(S) \cdot \vec{\sigma}$ defined by the dynamic model is that for any given store $\vec{\sigma}$, $\exists \vec{u} \in \vec{U}$, such that $T(S) \cdot \vec{\sigma} = C(S) \cdot (\vec{\sigma}, \vec{u})$, where $\vec{U}$ is a set of unknown values which are required to cope with the non-computability of the execution time. In other words, our complexity function maps a statement and a store to a set of possible execution times. The actual execution time must belong to this set for the complexity to be correct.

Proof We make several intuitive hypotheses first:

1. $\forall \vec{\sigma} \in \vec{\Sigma}, \exists \vec{u_c} \in \vec{U}$ such that:
   $T(cond) \cdot \vec{\sigma} = C(cond) \cdot (\vec{\sigma}, \vec{u_c})$
   which means that, provided a given state σ, when the condition is evaluated on the real machine, we can always find a set of variables u_c satisfying the equation.

2. $\forall \vec{\sigma} \in \vec{\Sigma}, \exists \vec{u_1} \in \vec{U}$ such that:
   $T(S_{true}) \cdot \vec{\sigma} = C(S_{true}) \cdot (\vec{\sigma}, \vec{u_1})$

3. $\forall \vec{\sigma} \in \vec{\Sigma}, \exists \vec{u_2} \in \vec{U}$ such that:
   $T(S_{false}) \cdot \vec{\sigma} = C(S_{false}) \cdot (\vec{\sigma}, \vec{u_2})$

Now suppose that the condition evaluation does not change the environment, and suppose that overhead is included in the complexity formula, we consider two different cases:

Case 1: if condition is true, we have:

$$\begin{aligned} T(IF...) \cdot \vec{\sigma} &= T(IF\ cond\ THEN\ S_{true}\ ELSE\ S_{false}\ ENDIF) \cdot \vec{\sigma} \\ &= T(cond) \cdot \vec{\sigma} + T(S_{true}) \cdot \vec{\sigma} \end{aligned}$$

Applying Hypothesis 1 and Hypothesis 2, we can find $\vec{u_c}$ and $\vec{u_1}$, so we have:

$$T(IF...) \cdot \vec{\sigma} = C(cond) \cdot (\vec{\sigma}, \vec{u_c}) + C(S_{true}) \cdot (\vec{\sigma}, \vec{u_1}) \tag{2.1}$$

Case 2: if condition is false, we can get:

$$\begin{aligned} T(IF...) \cdot \vec{\sigma} &= T(IF\ cond\ THEN\ S_{true}\ ELSE\ S_{false}\ ENDIF) \cdot \vec{\sigma} \\ &= T(cond) \cdot \vec{\sigma} + T(S_{false}) \cdot \vec{\sigma} \end{aligned}$$

Applying Hypothesis 1 and Hypothesis 3, we can find $\vec{u_c}$ and $\vec{u_1}$, so we have:

$$T(IF...) \cdot \vec{\sigma} = C(cond) \cdot (\vec{\sigma}, \vec{u_c}) + C(S_{false}) \cdot (\vec{\sigma}, \vec{u_2}) + overhead \tag{2.2}$$

Let p be probability of true case, then $q = 1 - p$ for false case, from Equation (2.1) and Equation (2.2), we have:

$$\begin{aligned} T(IF...) \cdot \vec{\sigma} &= p \cdot [T(cond) \cdot \vec{\sigma} + T(S_{true}) \cdot \vec{\sigma}] + (1-p) \cdot [T(cond) \cdot \vec{\sigma} + T(S_{false}) \cdot \vec{\sigma}] \\ &= T(cond) \cdot \vec{\sigma} + p \cdot T(S_{true}) \cdot \vec{\sigma} + (1-p) \cdot T(S_{false}) \cdot \vec{\sigma} \\ &= C(cond) \cdot (\vec{\sigma}, \vec{u_c}) + p \cdot C(S_{true}) \cdot (\vec{\sigma}, \vec{u_1}) + (1-p) \cdot C(S_{false}) \cdot (\vec{\sigma}, \vec{u_2}) \end{aligned}$$

So there must exist a certain $\vec{u} \subseteq \vec{U}$, such that $\vec{u} \in \vec{u_c} \otimes \vec{u_1} \otimes \vec{u_2}$, satisfying $\vec{u_c} \subseteq \vec{u}$, $\vec{u_1} \subseteq \vec{u}$ and $\vec{u_2} \subseteq \vec{u}$. Now we combine two cases:

$$\begin{aligned} T(IF...) \cdot \vec{\sigma} &= C(cond) \cdot (\vec{\sigma}, \vec{u_c}) + p \cdot C(S_{true}) \cdot (\vec{\sigma}, \vec{u_1}) + (1-p) \cdot C(S_{false}) \cdot (\vec{\sigma}, \vec{u_2}) \\ &= C(cond) \cdot (\vec{\sigma}, \vec{u}) + p \cdot C(S_{true}) \cdot (\vec{\sigma}, \vec{u}) + (1-p) \cdot C(S_{false}) \cdot (\vec{\sigma}, \vec{u}) \\ &= (C(cond) + p \cdot C(S_{true}) + (1-p) \cdot C(S_{false})) \cdot (\vec{\sigma}, \vec{u}) \\ &= (C(IF...)) \cdot (\vec{\sigma}, \vec{u}) \end{aligned}$$

If *cond* is not particular, as a first coarse approximation, we use the values $p = q = 1/2$ in the current implementation and we maintain another kind of correctness by keeping track of the number of approximations made. See counters in Section 4.2.2 for more information.

To keep it tractable, we assume that branch probabilities are statistically independent, so that we can use the two same constants p and q for every execution if the probability of the test cannot be detected easily.

Now the problem is to correctly evaluate them as often as possible.

2.3.5 Complexity of a Sequential DO

Do loops are the key control structures to estimate the execution time of scientific programs. Although DO construct is one of statement, as IF, we prefer to give it a full subsection because of its importance in time-predicting.

Suppose that we have a loop like:

```
DO index = lower, upper, increment
        body
ENDDO
```

The evaluation of lower, upper, increment may call functions, whose complexity must be added to the overall execution time as loop entrance overhead. The complexity of the Do loop *body* may depend on the index, so we must integrate it on the *index* rather than multiply it by the iteration count. We hereafter include in C_{body} the looping overhead of the loop index test and index increment.

We assume that the overheads for lower bound, upper bound and increment are constants. (See Section 3.5.6 for explanation.) These overheads include the evaluation of the these three expressions. $C(loopheader)$ is used to denote the addition of these three, i.e., $C(loopheader) = C(lower) + C(upper) + C(increment)$. And we also assume that loop initialization overhead and branching overhead are constant too. We use C_{init} and C_{branch} respectively.

The complexity of the loop is given below:

$$C(DO...) \cdot \vec{\sigma} = C_{init} + C(loopheader) \cdot \vec{\sigma_i} + \sum_{i=lower}^{upper} (C(body) \cdot \vec{\sigma_i} + C_{branch}) \tag{2.3}$$

Proof We know that the environment is changing all the time as the loop induction variable changes. It is not our purpose to calculate the $C(body)$ as index changes, we want to find out a unique formula to calculate $C(DO)$ once. Generally, we cannot obtain a formula that can be applied to all cases, that is to say, we have to add some restrictions on loop body without changing the actual behavior.

This formula applies if we are able to properly evaluate *lower*, *upper* and *increment* as polynomials and to prove that $(upper - lower) * increment > 0$, i.e., at least the loop is executed.

By induction we assume that $C(body)$ is already obtained.

Case 1: $C(body)$ is not dependent on loop header.
This is the simplest case since complexity is fixed. That is to say, the environment does not affect the complexity of the loop body and we can consider $C(body)$ is a constant. Assuming that the loop iteration is N_{loop}. Since both $C(body)$ and C_{branch} are constants, we have:

$$\begin{aligned} N_{loop} \cdot (C(body) + C_{branch}) &= \sum_{i=lower}^{upper} (C(body) \cdot + C_{branch}) \\ &= \sum_{i=lower}^{upper} (C(body) \cdot \vec{\sigma_i} + C_{branch}) \end{aligned} \tag{2.4}$$

and

$$C(loopheader) = C(loopheader) \cdot \vec{\sigma_i} \tag{2.5}$$

From the loop definition and Equation (2.4) and (2.5), we have the following formula:

$$
\begin{aligned}
C(DO...) &= C_{init} + C(loopheader) + N_{loop} \cdot (C(body) + C_{branch}) \\
&= C_{init} + C(loopheader) + \sum_{i=lower}^{upper} (C(body) \cdot \vec{\sigma_i} + C_{branch}) \\
&= C_{init} + C(loopheader) \cdot \vec{\sigma_i} + \sum_{i=lower}^{upper} (C(body) \cdot \vec{\sigma_i} + C_{branch}) \qquad (2.6)
\end{aligned}
$$

Case 2: $C(body)$ is a function of loop header, it can be the function of lower bound, upper bound, increment and/or any combination of these variables.
Now we add our restriction: $C(body)$ is a function of the loop headers, where the function takes the form of polynomial. With the algorithm introduced in Section 2.3 later in this chapter, we have the chance to calculate them once and for all.

It is the only case we can use the Equation (2.3) to calculate the complexity of DO loop.

For sequential loops, the statistics of the loop are computed by adding those of the expressions $lower, upper, increment$, and *body*. For parallel loops, it would be easier to have *body* with worst execution time.

2.3.6 Complexity of a Block

A basic block is a group of statements with unique entry and exit points such that if the entry statement executes, all other statements in the basic block will execute.

$$
\begin{aligned}
\sigma_0 &: \\
\sigma_1 &: \quad statement_1 \\
\sigma_2 &: \quad statement_2 \\
& \quad \dots \\
\sigma_n &: \quad statement_n
\end{aligned}
$$

Once the control graph is computed, the construction of PIPS internal representation of programs guarantees that there is no `GOTO` jumping in or out of the middle of a block, so that the n statements are always executed sequentially. The complexity of the block is simply as below:

$\forall \vec{\sigma}, \exists \vec{u}$ such that:

$$
C(block) \cdot (\vec{\sigma_0}, \vec{u}) = \sum_{i=1}^{n} (C(statement_i) \cdot (\vec{\sigma_i}, \vec{u}))
$$

Proof The proof is by induction on the statement number n. We know that for two statements, it is correct by definition. And for 3, 4, it will be correct too, until n. The only part is to update a complexity with respect to σ_i, into a complexity with respect to σ_0. To make it easier the implemantation uses σ_0, the initial store as unique reference, and preconditions to update the

local complexity. The use of transformers would let us decrease the number of unknown variables introduced in complexity formulae at the expense of implemantation simplicity.

The statistics of the whole block is the sum counter to counter of the statistics of its statements. PIPS detects only DO loop parallelism (nor `COBEGIN ... COEND` neither `FORK, JOIN`).

2.4 Complexity Algorithm for Unstructured Control Flow Graph

In this section, we will talk about the unstructured programs. Unstructured codes require a more complex technique.

2.4.1 Control flow graph and PIPS programs representation

Any Fortran program, even those containing `GOTO`s, can be represented by a graph whose nodes represent elementary instructions, and whose edges stand for `GOTO` jumps. Most of nodes have only one outgoing edge, `IF-GOTO-ELSE-GOTO` nodes have two, and the computed and assigned `GOTO` can have more. This graph is called a control flow graph (cf. [ASU86]); Internal representation of PIPS uses such graphs. Little unstructured pieces of code are encapsulated into one node in such graphs and viewed from the rest of the program as if they were single, structured blocks of instructions.

2.4.2 Complexity of an Unstructured Control Flow Graph

Let us now define a complexity for an unstructured program represented by its control flow graph. Let $\{S_1, S_2, \ldots, S_n\}$ be the set of the graph nodes, and S_1 the entry node. We are sure there is only one, because of the definition of the graph: if there were more than one entry point, that would mean there would be `GOTO`s reaching from outside of the graph into the middle of it; the origin vertex would actually be a part of it.

Let $c_1, c_2, \ldots, c_n$ be the complexities associated with $S_1, S_2, \ldots, S_n$, inductively assumed known. Let us call p_{ij} the probability of going to node S_j from node S_i. Apparently, the probability to go to any node, from node S_i, is 1, so $\sum_{j=1}^{n} p_{ij} = 1$. We will at last associate to each node S_i the average complexity g_i of the code still to be executed between S_i and the exit of the graph (g_i is a sort of "global cost"). Our goal is the evaluation of g_1 as it is the average complexity of the code executed between the first node S_1 and the exit node.

Here is a recursive definition of the g_i: the global cost of a node is its proper cost c_i plus the sum of the global costs of its successors g_j weighted by the associated probabilities p_{ij} to jump from node S_i to node S_j, as shown in Equation (2.7).

$$g_i = c_i + \sum_{S_j \text{ successor of } S_i} p_{ij} \cdot g_j \tag{2.7}$$

Naturally, $p_{ij} = 0$ if S_j is not a successor of S_i , so the sum can be expressed more regularly:

$$g_i = c_i + \sum_{i=1}^{n} p_{ij} \cdot g_j \tag{2.8}$$

There is a such an equation for each node. We can obtain a system of n equations with n unknown, the g_i, which can be written in a matrix form:

$$\begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_n \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} + \begin{bmatrix} p_{11} & p_{12} & \cdots & p_{1n} \\ p_{21} & p_{22} & \cdots & p_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ p_{n1} & p_{n2} & \cdots & p_{nn} \end{bmatrix} \cdot \begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_n \end{bmatrix} \tag{2.9}$$

Consequently, we have:

$$\begin{bmatrix} 1-p_{11} & -p_{12} & \cdots & -p_{1n} \\ -p_{21} & 1-p_{22} & \cdots & -p_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ -p_{n1} & -p_{n2} & \cdots & 1-p_{nn} \end{bmatrix} \cdot \begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_n \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} \tag{2.10}$$

We assume $G = \{g_1, g_2, \cdots, g_n\}$ and $C = \{c_1, c_2, \cdots, c_n\}$. Finally, we get

$$(I - P) \cdot G = C$$

Is $(I - P)$ always invertible? It should be possible to prove that, as soon as the program is not pathological. An infinite loop, `Si: GOTO Si` generates a horizontal null line in the matrix $(I - P)$ and makes it no invertible. But fortunately, $(I - P)$ generally is invertible in most cases.

Suppose A is the inverse of matrix $(I - P)$, that is:

$$A = (I - P)^{-1} \tag{2.11}$$

and suppose a_{ij} is the i_{th} line and j_{th} column element of A, we have:

$$G = (I - P)^{-1} \cdot C = A \cdot C \tag{2.12}$$

that is:

$$\begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_n \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \cdot \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} \tag{2.13}$$

We extract from Equation (2.13) the complexity of the whole graph:

```
1    S = S + A(I)
2    IF(I.EQ.N) GOTO 4
3    I = I + 1
     GOTO 1
4    MOY = S/N
```

Figure 2.2: The Original Fortran Program

$$g_1 = a_{11} \cdot c_1 + a_{12} \cdot c_2 + \cdots + c_n \cdot a_{1n} \tag{2.14}$$

The computation of the p_{ij} involves the following cases:

- Node S_j is always executed immediately after node S_i, which means $p_{ij} = 1$
- Node S_j is never executed immediately after node S_i, which means $p_{ij} = 0$
- Node S_i has more than one successor. That is to say there are k nodes, namely, $S_{ij_1}, S_{ij_2}, \ldots, S_{ij_k}$ and any of them can be executed immediately after node S_i, and the probabilities from node S_j to these k nodes are $p_{ij_1}, p_{ij_2}, \ldots, p_{ij_k}$ respectively, such that $\sum_{m=1}^{m=k} p_{ij_m} = 1$.

So the problem becomes how to correctly calculate the matrix inversion. Press et al. [PFTV86] provided the algorithm which is used in our implementation.

2.4.3 Example

Now we present an example to illustrate the Equation (2.7). Let us try the unstructured program in Figure 2.2. Figure 2.3 represents its control flow graph.

We assume that p is the probability from $S2$ to $S4$. The transition probability matrix of this flow graph is:

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1-p & p \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

We have

$$I - P = \begin{bmatrix} 1 & -1 & 0 & 0 \\ 0 & 1 & p-1 & -p \\ -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and consequently

$$(I-P)^{-1} = \frac{1}{p} \cdot \begin{bmatrix} 1 & 1 & 1-p & p \\ 1-p & 1 & 1-p & p \\ 1 & 1 & 1 & p \\ 0 & 0 & 0 & p \end{bmatrix}$$

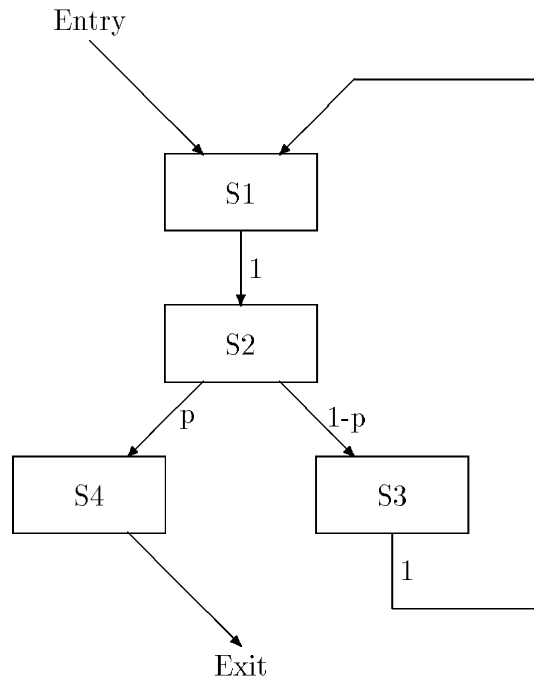


Figure 2.3: The Unstructured Control Flow Graph

Finally, we can get the complexity of the graph g_1:

$$g_1 = \frac{c_1}{p} + \frac{c_2}{p} + \frac{(1-p) \cdot c_3}{p} + c_4$$

If $p = 0$, the complexity of the graph is not defined because $S1$, $S2$ and $S3$ form an infinite loop. The program execution will never reach $S4$.

If $p = 1/n$: then we have:

$$g_1 = n \cdot (c_1 + c_2) + (n-1) \cdot c_3 + c_4 \tag{2.15}$$

$S1$, $S2$ are executed n times, and $S3$ $(n-1)$ times, $S4$ once. This matches the intuition of a DO loop.

2.5 Symbolic Calculation of Polynomial Summation

In this section, we introduce how to calculate symbolic polynomial summation, and problem that arises from the formula in practice.

2.5.1 The Formula of Symbolic Calculation

In the complexity evaluation, we need to calculate polynomial summation symbolically. The problem looks like:

$$C = \sum_{i_1=a_1}^{b_1} \sum_{i_2=a_2}^{b_2} \cdots \sum_{i_n=a_n}^{b_n} (exp) \tag{2.16}$$

where a_k and b_k ($k = 1, 2, \cdots, n$) may be the numerical constants or linear function of i_j, provided $j < k$, and exp is a linear function of i_k, and C is polynomial result which may contain a_k and b_k and must not contain i_k.

The calculation of Equation (2.16) is done from right to left. The above problem becomes the calculation of the expression of the following form:

$$C = \sum_{i=l}^{u} (c_0 + c_1 i + c_2 i^2 + \cdots + c_m i^m) \tag{2.17}$$

where c_k ($k = 1, 2, \cdots, m$) are polynomial which do not contain i, that means we can consider c_k as constant coefficients. So the problem in Equation 2.17 becomes

$$C = \sum_{i=l}^{u} c_0 + \sum_{i=l}^{u} c_1 i + \sum_{i=l}^{u} c_2 i^2 + \cdots + \sum_{i=l}^{u} c_m i^m = c_0 \sum_{i=l}^{u} 1 + c_1 \sum_{i=l}^{u} i + c_2 \sum_{i=l}^{u} i^2 + \cdots + c_m \sum_{i=l}^{u} i^m \tag{2.18}$$

In order to calculate the Equation (2.18), we only need to calculate $\sum_{i=1}^{n}(i^p)$. Fortunately, Knuth [Knuth73] and Spiegel [Spiegel74] provided the formula:

$$\sum_{i=1}^{n}(i^p) = \frac{n^{p+1}}{p+1} + \frac{1}{2}n^p + \frac{B_1 p n^{p-1}}{2!} - \frac{B_2 p(p-1)n^{p-2}}{4!} + \ldots \tag{2.19}$$

where p and n are non-negative integers, the series terminates at n or n^2 according to that p is odd or even. If $n = 0$, the result is 0. The B_k are Bernouilli number defined by the series:

$$\frac{e}{e^x - 1} = 1 - \frac{x}{2} + \frac{B_1 x^2}{2!} - \frac{B_2 x^3}{4!} + + \frac{B_3 x^6}{6!} + \cdots |x| < 2\pi$$

and

$$1 - \frac{1}{2} cot \frac{1}{2} = \frac{B_1 x^2}{2!} + \frac{B_2 x^3}{4!} + + \frac{B_3 x^6}{6!} + \cdots |x| < \pi$$

Until now, we can claim that we are generally able to calculate symbolic polynomial. Apparently, we have no problem to perform operations like addition, subtraction, multiplication of two polynomials, to substitute a symbolic variable with an expression or a constant, to perform $\sum$ operation between two bounds, because we have:

$$\sum_{i=inf}^{sup} exp(i) = \sum_{i=1}^{sup} exp(i) - \sum_{i=1}^{inf} exp(i) + exp(inf) \tag{2.20}$$

The only general problem is division operation. Generally speaking, we have no way to perform division operation for any two arbitrary polynomials. In our implementation, if we encounter this

case, our estimator will report error and exit. But there is only one exception, i.e., when divider polynomial is a monomial, that is, one term polynomial. Because we only need to convert the division to multiplication of its reciprocal. For instance, divided by $3m^2$ means multiplied by $1/3 \cdot m^{-2}$.

2.5.2 Problems of Symbolic Calculation

In general, we cannot know the static values of symbolic variables before the execution. This raises some issues in practice.

Tawbi [Tawbi90] describes a problem related to the symbolic calculation. She uses the following example:

```
      DO 10 i = 1, n
         S1
         DO 10 j = 2, i
            S2
            DO 10 k = j, m-2
10             S3
```

In this case, assuming M_i denotes the iteration number of S_i when $I = 1, 2, \cdots,$, we have the following:

$$M_1 = n$$

$$M_2 = \sum_{i=1}^{n} (i-1)$$

The calculation of M_3 is a little difficult and tricky and cannot be derived directly by the standard formula:

$$M_3 = \sum_{i=1}^{n} \sum_{j=2}^{i} (m - j - 1) \tag{2.21}$$

because if $m - j - 1 \geq 0$ it is correct for all the values of i and j. But if not the error can be enormous. For example, take $n = 300$ and $m = 100$. The result when using Equation (2.21) is -104650. It is obviously wrong since execution time cannot be negative. The real number of iteration of `S3` is 964859. But what is wrong with the Equation (2.21)?

In Fortran, the loop body is not executed if the upper bound is less than lower bound. In the above example, the loop on j is not executed, since $i = 1$, but it is included in the Equation (2.21); and the calculation on k will be executed for all the values of j only if $m - 2 \geq n$. If $m - 2 < n$, the loop nest iteration number of `S3` i.e., M_3 in fact is divided into two parts as following:

```
      DO 10 i = 2, m-2
         DO 10 j = 2, i
            DO 10 k = j, m-2
10           S3

      DO 20 i = m-1, n
         DO 20 j = 2, m-2
            DO 20 k = j, m-2
20           S3
```

and M_3 can be calculated as:

$$\text{If } m-2 \geq n : M_3 = \sum_{i=2}^{n} \sum_{j=2}^{i} (m-j-1) \tag{2.22}$$

$$\text{If } m-2 < n : M_3 = \sum_{i=2}^{m-2} \sum_{j=2}^{i} (m-j-1) + \sum_{i=m-1}^{n} \sum_{j=2}^{m-2} (m-j-1) \tag{2.23}$$

The Equation (2.22) and Equation (2.23) will give the right answer no matter what values m and n take.

We have taken notice that in order to well calculate the number of iteration of loop, we have to vary the counter of loop according to different cases to assure the right answer. Therefore, in order to calculate the number of instantiations of the loop body `S`, we need to:

1. split the loop body into several chunks where the number of iterations should not be negative. This is to avoid the errors occurred in the calculation of M_3 above.

2. calculate M_i while making symbolic expressions calculations

This adds a lot of extra work to the simple static evaluation based on a discrete integration, for we cannot evaluate values of symbolic variables prior to execution. In the example, the values of m and n are required. And the most important, sometimes we even have no ideas at all about the real values of symbolic variables. As you can see, for this small piece of Fortran code, the right answer is already complicated, let alone the real Fortran program which are generally several thousands lines. The answer for real program could be too long to be meaningless.

In order to avoid this kind of problem, we decide that it is up to user to check the loop bound problem because most programs do not contain dead loops or dead iteration. Such cases seem to happen when there is mistake or when some complicated loop pattern is generated by a compiler. W. Pugh and D. Wonnacott [PW92] discussed the similar problem. W. Pugh [Pugh94] describes methods to count the number of integer solutions to selected free variables of a Presburger formula, or sum a polynomial over all integer solutions of selected free variables of a Presburger formula.

2.6 Summary

In this chapter, we have firstly introduced PIPS, on which complexity in PIPS is based, and the complexity itself and its algorithms to calculate and proven them correct.

2.6.1 Top-Down vs. Bottom-Up Analysis

The complexity evaluation is performed one module at a time in topological order. This is possible because of the non-recursive nature of Fortran. In this way, each time a subroutine or function is encountered, its complexity has already been evaluated. So we use bottom-up analysis. When the complexity of a sub-module is required, it is assumed to be available, somewhere in the database, managed by *pipsdbm*. A top-down complexity analysis could be performed. The information computed would then be the number of time each piece of code is executed.

2.6.2 Proper Complexity and Cumulated Complexity

The proper complexity and cumulated complexity are similar to those of effects (cf. Section 2.1.6). Proper complexity is related to the complexity of the next statement, while cumulated complexity is related to the complexity of next block. The extreme case of cumulated complexity is summary complexity, which is the complexity of the whole module. No use was formed for proper complexity.

Chapter 3

Target Machine Model

Each computer manufacturer has its own way to deal with the numerical computations. As computer technology advances, the supercomputer is getting more powerful. In order to know the performance of these supercomputers, we need to know the approximate cost for each operation, memory accesses, etc.

We are interested in execution time evaluation, but various optimization techniques that exist at different levels: multiple ALU, cache, pipeline, etc. make this task very difficult. The question is to determine which machine features have to be taken into account and which do not.

In this Chapter, let us first have a look at sequential machines. In section 3.1, we enumerate some parameters that could reasonably be part of a machine model, keep only the most important ones for a simple implementation and use. Then we will discuss the characteristics of parallel machines: its architecture and classification in section 3.2. We will present two simple models: **all-1** and **fp-1**, that are implemented in our prototype in Section 3.3. Finally, we will show how we obtain the real cost table for SUN SPARCstation 2. But we postpone the experimental verification to Chapter 5.

3.1 Sequential Machines

Sequential machines are the starting point for our machine model, because they are simpler ones. In this section, we present the impact of computation models, and then the impacts of memory and processing elements.

3.1.1 Sequential Machine Model

We are going to list some models of computation which have the impact on the performance.

Sequential Von Neumann Model

For more than 40 years, the *Von Neumann Principles* have dominated computer architectures. This computational model was first used to build uniprocessors. A classical von Neumann computer

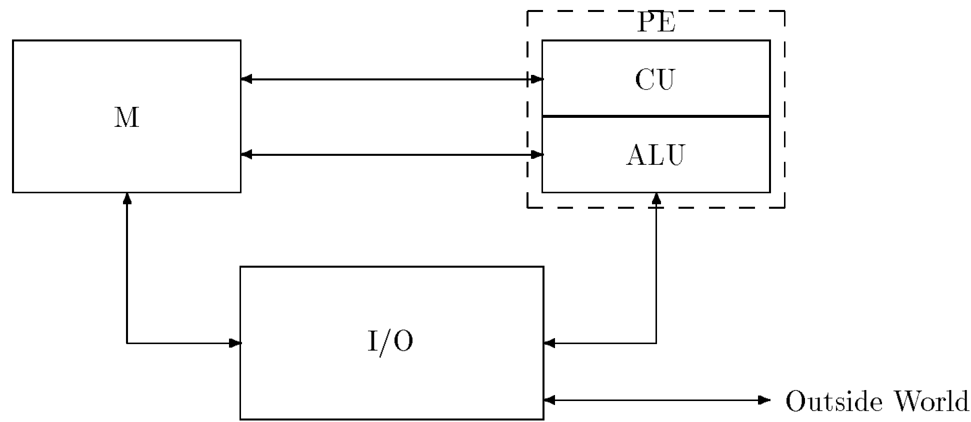


Figure 3.1: Von Neumann Computer Model

consists of a program control unit (CU), an arithmetic logic unit (ALU), an input/output (I/O), and memory (M), as shown in Figure 3.1. The CU and the ALU collectively make up the processing element (PE).

The von Neumann model is based on the following principles:

- A single processing element separated from memory by a communication bus;
- Linear organization of fixed-size memory cells;
- Low-level machine language with instructions performing simple operations on elementary operands; and
- Sequential centralized control of computations.

These principles are simple and well understood; considerable progress has been made with them and confidence in this model has grown over the years. One reason for the success of the von Neumann computer is the flexibility of its implementation. For example, communication between processor and memory may be achieved with separate or multiplexed data and address buses. Also, the von Neumann model leaves room for a variety of processor and memory organizations, such as hierarchical memory.

RAM Model

The conventional uniprocessor can be abstracted as a Random-Access Machine (RAM) model. This theoretical model is useful for algorithm development and for complexity studies. It also constitutes the starting point for the theoretical parallel models.

The *worst time complexity* or just *complexity* of a RAM program is the function $f(n)$ which is the maximum, over all inputs of size n, of the sum of the time taken by each instruction executed. Since each step takes a fixed amount of time, the RAM model can be used as our dynamic model which was introduced in Section 2.2.1.

Pipelining

The process of pipelining divides a task into many subtasks and assigns them to a chain of processing stations. These stations are called pipeline segments. Parallelism is achieved when several segments operate simultaneously. It is possible to use the principle of pipelining either at the *instruction* level or at the *arithmetic* level.

This implementation improvement may be hidden in our dynamic model if no pipeline bubble exists or if bubble can be statistically taken into account as a slowdown coefficient.

Vector Processors

Vector processors are specially designed to handle computations formulated in terms of vectors. A vector processor has a set of instructions that treat vectors as single operands. The vector processor could conceivably be implemented by replicating the number of scalar ALUs to the size of a vector. Since vectors are one-dimensional arrays and the same sequence of operations is repeated for every vector element, vector processing is ideally suited to pipeline arithmetic.

Often uniprocessor computer systems are equipped with vector co-processors in order to speed up vector computations. Sequential code can sometimes be vectorized. This simple and inexpensive technique usually provides significant speed improvements over a purely scalar operation, and meanwhile causes a big problem for evaluation. Since vector processing is quite close to parallel processing, vector units are not taken into account in our sequential model.

3.1.2 Multiple Processing Units

All CPU manufacturers introduce in their chips some parallelisms that high level programmers do not need to know and, at the same time, cannot control. For example, integer and floating-point computations, and address calculations are often performed in the meantime, or at least overlapped in superscalar and/or superpipelined machines. This makes exact static time evaluation of user programs almost impossible or at least very expensive [Jonkers92, Wang93]. We cannot determine approximate overall time by counting only the most costly operations (especially floating-point), since loads of the different units, on supercomputing machines, are generally well balanced. Ideally, the integer computation unit runs as much as the floating-point one, and so does the CPU as a whole, compared to the memory ports activity. If this remark applies for real programs, it could inspire to measure only one of three activities, or to take the average.

For a coarse and fast approximation, one can compute separately the time spent in the different types of units: T_{cpu} in CPU, T_{fpu} for floating-point computations, etc. then divide each time by the number of appropriate units available, and take the maximum of all quotients. That solution gives its best results when the work for all unit types is homogeneously mixed in the program.

A finer simulation of the overlapping may be obtained by maintaining some sort of reservation table throughout the complexity evaluation [Wang93].

3.1.3 Memory Issues: Register, Cache and Virtual Memory

The memory system of a computer is sometimes dubbed as "the von Neumann bottleneck" because of the critical role it plays in affecting performance of the computer. In this subsection, we show the impacts of the memory hierarchy from register to cache memory and virtual memory.

Scalar and Vector Registers

Registers are memory cells used by CPU, exist at one level under the cache. It is a source of error in the evaluation of memory reference time: some frequently used data are kept in scalar or vector registers, and their access time is about three times faster than cache. As a first approach, the loop indices can be supposed to be in scalar registers, the other variables staying in the cache or main memory.

Cache and Virtual Memory

Virtual-memory systems, as they exist today, fulfill a role similar to cache memories for CPU, except that virtual-memory systems manage a different portion of the memory hierarchy. Cache-management algorithms attempt to make optimum use of a high-speed memory for which main memory serves as a backup buffer. Active items tend to move from main memory to cache, and inactive items tend to migrate back to main memory.

Stone [Stone87] defines the effective access time as:

$$t_{eff} = h \cdot t_{cache} + (1 - h) \cdot t_{main} \tag{3.1}$$

where h is cache hit's probability and $1 - h$ is cache miss' probability which is also the probability for main memory access.

Virtual-memory systems attempt to make optimum use of main memory, while using an auxiliary memory, usually a rotating magnetic disk memory, for backup. Therefore, to the first order of approximation, the high-speed buffer memory of a cache system corresponds to main memory of a virtual-memory systems, and main memory of a cache system corresponds to an auxiliary memory of a virtual-memory systems.

The principles that govern the behavior of cache and virtual memory systems are largely the same. Namely:

- Keep active items in the memory that has the higher speed;
- As items become inactive, migrate them back to the lower-speed memory; and
- If the management algorithms are successful, the performance will tend to be close to the performance of the higher-speed memory, and the cost will tend to be close to the cost per bit of the lower-speed memory.

Comparison	High-Speed Memory	Low-Speed Memory
CPU	Register	Cache Memory
Cache	Cache Memory	Main-memory
Virtual	Main-Memory	Auxiliary-Memory

Table 3.1: Relative High and/or Low Memory

Memory access delays are comparable to CPU execution times, depending upon the number of banks, access ports, ALUs and FPUs, etc. So it is important to include them in the complexity evaluation. For example, we can first make the rough assumption that a memory access takes a certain amount of time on the average; but the fact that the cache hit access time may be at least four and up to twenty times faster than the cache miss leads to consider cache memory [Stone87]. Can one statically decide that a certain reference will be a cache-hit or cache-miss? Though determining whether arrays accessed by arbitrary functions access the same memory location is undecidable [Ber66], an approximating method can be used. We suggest to use an average hit ratio and to let the user adjust it according to his knowledge.

The virtual memory raises the same problem as the cache memory, except that for the users interested in CPU time, page faults are invisible. A page fault penalty can be 1000 to 10,000 as costly as a page hit. However, as the supercomputer client pays for the product *memory space* $*$ *cpu time*, he is not interested in an evaluation of elapsed time. So this feature does not have to be handled although the delays generated by the page fetching are several orders of magnitude longer than all sorts of operations we count.

Summary of Memory Issues

Table 3.1 shows the memory comparison. For CPU, the high-speed memory is register while low-speed memory is cache; but for cache memory, the cache becomes the high-speed and main memory is low-speed; finally for virtual memory, main memory is high-speed while auxiliary memory is low-speed.

3.2 Parallel Machines

Parallel machines are considered to be much more complicated than sequential ones. All issues discussed in the previous section apply here. And besides that, parallel machines have their own characteristics. Here we only briefly introduce two items related to parallel machines: architecture and classification. In Chapter 6, more characteristics are taken into account.

3.2.1 Parallel Machine Architecture

Parallel computer architectures may be characterized by the following criteria: processor complexity, mode of operation, memory structure, interconnection network, and number of processors and memory size.

Processor Complexity

Processor complexity refers to the computing power and the internal structure of each processing element. Homogeneous systems are those in which all processors have identical capabilities, e.g. BBN TC2000 which was used to carry out our experiments. Heterogeneous systems are those in which processors are not identical, as for instance when they are specialized for performing certain functions. Processor complexity varies from one architecture to another. Homogeneous systems are more common and our target machine is assumed homogeneous.

Besides the homogeneity of processors, the computer's power also hinges on the power of its processors. We can roughly distinguish them by splitting them into three categories: large-grain, medium-grain and fine-grain architecture. (1) Large-grain architectures have few, but powerful processors. The Cray X-MP has from one to four extremely powerful processors. (2) Medium-grain architectures have an intermediate number of processors, with intermediate processing power. For example, Sequent multiprocessor. (3) Fine-grain architectures employ a large number of smaller processors. The Connection Machine CM-2 has 65536 bit-serial processors.

Mode of Operation

Mode of operation is a general term referring to both instruction control and data handling. The traditional mode of operation is command-flow, so called because the flow of events is triggered by commands derived from instruction sequences. Another method is to trigger operations as soon as their operands become available, as in data-flow, in which computations take place only if their results are requested by other computations. A von Neumann control flow machine is assumed.

Memory Structure

Memory structure refers to the mode of operation and the organization of computer memory. Memories can be accessed by using either addresses or data content (as in associative memories). In some new computer model, such as neural networks, memory consists of interconnection weights that indicate how easily connections can be made. A standard shared-memory is assumed.

Interconnection Network

Interconnection network refers to the hardware connecting processors together or processors and memories. There are generally three types:

- **Bus** is communication path shared by the processors, connecting them to the memory unit.
- **Crossbar** will connect each processor directly with every memory modules, so there is no memory contention.
- **Multilevel** is a connection between two points via a number of intermediate switching stages.

The effect of interconnection network makes the memory hierarchy more difficult to modelize and we chose to ignore this factor.

Number of Processors

Number of processors and memory size simply indicate how many processors the parallel system contains and how large the main memory is.

As part of automatic parallelization, the complexity estimator has to consider the number of processors of the machine. This parameter alone is good enough to simulate coarse-grain parallelism, but for little tasks, the cost of process creation must be considered, as it becomes comparable to the task cost. The operating system behavior is more important in the model than a simple processor model.

3.2.2 Classification for Parallel Machines

In 1972, Flynn [Flynn72] proposed a first classification scheme for parallel computer architectures. The classification is based on the distinction between multiple instruction streams and one instruction stream, and between multiple data streams and one data stream. By this approach Flynn obtained four classes of computer architectures:

- Single Instruction stream — Single Data stream (SISD)
  SISD is standard sequential machine model and it comprises all conventional uniprocessor systems. Vector computers, which have instructions that operate on entire arrays rather than on scalar entities, are also included here. This kind of machines is what we talked about in the previous section.

- Single Instruction stream — Multiple Data stream (SIMD)
  SIMD denotes any parallel computer architectures with only one instruction stream and synchronized processing elements. It consists of multiple ALUs under supervision of a single control unit (CU). The CU synchronizes all the ALUs by broadcasting control signals to them, but on different data that each of them fetches from its own memory. So it is also called *data parallel* or *array processors*. Actually, the SIMD machines are the CM-2 from Thinking Machines Corp., MP-1 from MasPar, TC2000 from BBN, etc.

- Multiple Instruction stream — Single Data stream (MISD)
  MISD is not incorporated in any existing parallel machines, so it is of no real use practically.

- Multiple Instruction stream — Multiple Data stream (MIMD)
  MIMD refers to any parallel computer architectures with multiple instruction streams and mostly unsynchronized processing elements. MIMD machines are Paragon from Intel Corp., CM-5 from Thinking Machines Corp., etc.

The above classification leaves us with two essential classes of parallel computer architectures, viz. SIMD and MIMD. Although these two terms are widely, frequently used, they only give a very vague distinction between the various kinds of parallel computer architectures.

The two classes can be modelized in a similar way, with a special handling of test for SIMD machines. To keep things manageable, we restrain the scope of our estimator to MIMD machines exploited in SPMD mode, which stands for Single Program Multiple Data.

#	int	float	double	complex	double-complex
+	1	1	1	1	1
−	1	1	1	1	1

Table 3.2: Unity Cost for Operation

3.3 Two Trial Models

Before we use operation costs for the dynamic model of real machine, we first present two cost tables which are used to obtain purely theoretical results. We find them equally important because they give users some interesting information about their programs.

Each cost table is divided into six files: index, memory, operation, transcend, trigo and overhead.

3.3.1 All-One Cost Table

In the work reported here, we assume that in the ideal machine, very similar to the RAM model used for theoretical complexity, each arithmetic operation, memory read, memory write and intrinsic function invocation consume one unit of time. All other activities, including I/O operation are assumed to be instantaneous. In this way, the result is in much more machine-independent manner. For example, the overhead of synchronization and/or communication is a function of the target machine and not of the program. Therefore, we decided to ignore that time.

We detail the complexity cost in the following categories:

Operation Costs of All-1

Since it is called all one, each operation has an execution time of 1 regardless of the operation types. For example, an excerpt of all-1 operation cost table is shown in Table 3.2, the plus operator always has unit cost 1, so does minus operator for any kind of operand.

Memory Costs of All-1

The assignment "=" is performed in no time because the cost is considered to be included in memory access. The memory access (including load and store) is equally given unit cost 1, as shown in Table 3.3.

Array Index Costs of All-1

When an element of array is accessed, there is some extra costs. For example, we need to take the dimension of the array into account and calculate the offset of multi-dimensional arrays. Obviously, there is a big difference of access between the array reference with constant indexes and the one with varying indexes.

#	int	float	double	complex	double-complex
MEMORY-LOAD	1	1	1	1	1
MEMORY-STORE	1	1	1	1	1
=	0	0	0	0	0

Table 3.3: Unity Cost for Operation for Memory Access

Dim	Case	Local	Formal	Global
1	$*$	1	1	1
2	$**$	1	1	1
3	$***$	1	1	1

Table 3.4: Unity Cost for Array Index

In our all-1 theoretical cost table, we do not distinguish between these cases, we think of them all as equivalent. For example, A3 is a three-dimensional array, $A3(2,3,4)$ has the same cost as $A3(i,j,k)$ where i,j and k are varying, e.g., loop induction variables.

The cost is shown in Table 3.4 where asterisk sign means that it can be a constant or varying index.

Transcendental Costs of All-1

All the transcendental functions are given equal unit cost 1. An excerpt of that table is shown in Table 3.5. This is not realistic as transcendental execution times are likely to be data dependent.

Trigonometric Costs of All-1

Trigonometric functions are also given equal unit cost 1. We also include hyperbolic functions, such as sinh. An excerpt of that table is shown in Table 3.6. This is another approximation sine these operation execution time are data dependent.

#	int	float	double	complex	double-complex
MIN	1	1	1	1	1
MAX	1	1	1	1	1
EXP	1	1	1	1	1
ALOG	1	1	1	1	1

Table 3.5: Unity Cost for Intrinsic Transcendental Functions

#	int	float	double	complex	double-complex
SIN	1	1	1	1	1
DSIN	1	1	1	1	1
CSIN	1	1	1	1	1
COS	1	1	1	1	1
SINH	1	1	1	1	1
DSINH	1	1	1	1	1

Table 3.6: Unity Cost for Intrinsic Trigonometrical Functions

#	int	float	double	complex	double-complex
MEMORY-LOAD	0	0	0	0	0
MEMORY-STORE	0	0	0	0	0
=	0	0	0	0	0

Table 3.7: Floating-Point Cost for Memory Access

Overheads of All-1

The loop initialization overhead is taken as 1. And no time is charged for loop branching overhead, return overhead and call overhead.

3.3.2 Floating-Point-One Cost Table

The Floating-Point-One Cost Table **fp-1** is almost the same thing as **all-1** except that it takes only floating-point operations including load and store into account while integer operations are excluded. For example, memory accesses are not taken into account. So we give them zero cost, as is shown in Table 3.7.

And in Table 3.8, floating-point operations are accounted. Although others, such as double precision operations, may cost more than floating-point operations, we also include them in the table with the same weight.

#	int	float	double	complex	double-complex
+	0	1	1	1	1
−	0	1	1	1	1

Table 3.8: Floating-Point Cost for Operation

3.3.3 Summary of Trial Cost Tables

In this subsection, we introduced the **all-1** and **fp-1** cost tables. Although both remain ineffective in practice, it can give some nice results of the program. For example, how many operations are executed in a program.

3.4 Experimental Machine Model

In this section, we present experiment to define the elementary costs for a SUN SPARCstation 2, and then fill the cost tables for that machine.

3.4.1 The Limit for Machine Modelization Accuracy

There are a few good reasons for restraining ourselves from over-sophisticating our machine model – the dynamic model. Firstly, the static evaluation of complexity itself contains intrinsic uncertainty due to the problems described in Section 2.2. The second reason is our ignorance of the assembly code that compiler generates, though we need the assembly code to determine the real machine execution time in Section 3.5. Last, the more sophisticated the model, the more complicated the evaluator, and the longer the evaluation. The duration of the evaluation time must also be weighted against the accuracy gained and its usage. So we need a simple, effective method to obtain elementary costs of target machine.

Bentley et al. [BKV91] described how to make an elementary C cost model for a list of machines.

3.4.2 A Simple Program to Start

If you want to estimate the cost for floating point division, for instance, a trivial program does the job. Here is an example to get the time of floating-point division on a machine.

```
main()
{
int i,n;
float f1,f2,f3;
n = 1000000;
f2 = f3 = 3.0;
for ( i=0; i < n; i++ )
    f1 = f2 / f3;
}
```

On UNIX systems, it is easy to time programs with the **time** command. Generally, three kinds of information will be available at the end of the run:

- **User time**: describes the time spent in user mode, where instructions the compiler generated on user's behalf get executed, in addition to any subroutine library calls linked with your program. (Cache-miss is buried here too.)

- **System time:** is a measure of time spent in kernel mode where operating system intervenes the program.
- **Elapsed time:** is a measure of the actual (wallclock) time that has passed since the program was started.

Taken together, user time and system time are called *CPU time*:

$$CPU\ time = user\ time + system\ time$$

If you have all the machine resource at your disposal, you can run the program and obtain an approximation. For the programs that spend most of their time computing, the elapsed time should be very close to the CPU time. For instance, if the program takes the 25 seconds to do a million floating-point divides, then you know that each one costs about 25 microseconds. Of course, we have included the loop overhead in 25 microseconds, we can run that program once on the null loop to subtract out that time. The tight loop guarantees that both code and data will be in any registers the machine might have, i.e., an arbitrary division operation in a real program might take a lot longer. Or a smarter compiler might notice that the expression doesn't change at all in the loop body, and therefore push the division outside the loop and then not even execute the empty loop body. Fortunately, this experiment gives useful approximations for many compilers and machines. We assume the execution time not to be data dependent.

Alternatively, we can do this job more automatically, i.e., use the UNIX system C library function *times*. This function gets process times, i.e., it returns time-accounting information for the current process and for the terminated child processes of current process. All times are in 1/HZ seconds, where HZ is 60 on SUN SPARCstation 2, which is the clicks per second.

The **buffer** points to the following structure:

```
struct tms {
     clock_t   tms_utime;          /* user time */
     clock_t   tms_stime;          /* system time */
     clock_t   tms_cutime;         /* user time, children */
     clock_t   tms_cstime;         /* system time, children */
};
```

What we need are only `tms_utime` which is the CPU time used while executing instructions in the user space of the calling process and `tms_stime` which is the CPU time used by the system on behalf of the calling process. Because instructions we need to time do nott contain any procedure call, so we do not need `tms_cutime` which is the sum of the `tms_utime`s and `tms_cutime`s of the child processes, and `tms_cstime` either which is the sum of the `tms_stime`s and `tms_cstime`s of the child processes. Upon successful completion, times() returns the elapsed real time, in 60ths of a second, since an arbitrary point in the past. This point does not change from one invocation of times() to another within the same process.

So we only need to add a small procedure *counter* between what we need to time. We built a counter procedure in C as in the following:

```
#include <stdio.h>

#include <sys/types.h>
#include <sys/times.h>

int
counter_()
{
    struct tms buffer;
    int t;

    times(&buffer);
    t = (int)( buffer.tms_utime + buffer.tms_stime );
    return (t);
}
```

Careful readers could notice that there is an underscore after the procedure name *counter*, that is because our ultimate goal is to time Fortran program. We prefer the time function in C to these available in Fortran, simply because this time function in C provides more precise timing figures, that is, one-sixtieth second, while time functions available in Fortran can only provide seconds elapsed. If we want to extend its usage to Fortran program, we have to tackle the interface problem between Fortran and C, and this underscore is strictly demanded by the C–Fortran interface protocol. See [SUNF88] for details.

3.4.3 Automating the Elementary Measurements

We want to find the cost of a few operations, it is easy to modify a tiny program with a text editor and run it by hand a few times. But as soon as you time more than a few operations, it pays to have a program do the work.

J. Bentley et al. [BKV91] proposed a C program namely **mintime.c** which is used to time a few C operations:

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/times.h>
#define CLICKSPERSEC 60

int jobclicks()
{
    struct tms buffer;
    times(&buffer);
    return (int)(buffer.tms_utime + buffer.tms_stime);
}

#define loop1(CODE) start = jobclicks(); \
    for ( i=0; i<n; i++) { CODE; } \
    clicks = jobclicks() - start; \
```

```
    sumclicks += clicks; \
    printf("%5d", clicks);

#define loop(CODE) sumclicks =0; \
    printf("  %-15s", "CODE"); \
    loop1(CODE) loop1(CODE) loop1(CODE) \
    loop1(CODE)  loop1(CODE) \
    printf("%10.2f\n", sumclicks*1.0e6/  \
   (CLICKSPERSEC * n * 5));

main()
{
    int n,i,i1,i2,i3;
    int clicks,sumclicks,start;
    n = 1000000;
    printf("Null Loop\n");
    loop( {} )
    printf("Int Operations\n");
    i2 = i3 = 5;
    loop(i1 = i2 + i3)
    loop(i1 = i2 - i3)
    loop(i1 = i2 * i3)
    loop(i1 = i2 / i3)
}
```

The listing shows a minimal timing program which produces the cost of several C instructions shown in the following. For each operation, we see the cost of five experiments measured in the basic system time unit of "clicks" (1/60th of a second on SUN SPARCstation 2), followed by the average cost in microseconds.

The first dozens lines give us access to the system timing routines. This is the part of the problem that varies most from system to system — the **time** function gives less accurate times but is more portable. The middle part of the program contains the two macros that are the heart of the timing experiment: **loop1** runs a single experiment, while **loop** performs a group of five and prints out the operation name and average cost. These are macros rather than functions because we pass executable code to them as parameters, which could reduce the function call overhead considerably. The **main** function invokes the desired experiments.

The output of **mintime.c**:

```
Null Loop
   {}               27   26   27   29   27       0.45
Int Operations
  i1 = i2 + i3      40   40   39   40   44       0.68
  i1 = i2 - i3      42   41   39   40   43       0.68
  i1 = i2 * i3      75   77   73   75   74       1.25
  i1 = i2 / i3     121  118  117  120  118       1.98
```

As a side-product, we made a complete program which times almost all C operations on SUN SPARCstation 2. The interested readers can see the full output in Appendix B of the thesis.

The complete program has the same basic structure as **mintime.c** but adds several bells and

whistles, e.g., the average number of microseconds taken by the null loop is subtracted from later operation (but the raw time clicks is still printed). Some changes are cosmetic and some changes are invisible: some dummy operations are added between the inner loops in an attempt to avoid identical cache alignments. We also added a simple test for statistical robustness. The only substantial change, however, is a large set of operation.

3.4.4 Elementary Fortran costs on SUN SPARCstation 2

What we did in the previous subsection is to obtain the elementary C operation cost on SUN SPARCstation 2. Now we turn our attention to elementary Fortran operation cost.

We use the *counter.c* procedure provided in the previous subsection, and write a small Fortran program.

```
      program ftime
      real f,f1,f2
      integer i,n
      integer start, end
      integer count
c
      n = 1000000
      f1 = 1.2
      f2 = 3.4
      start = counter()
      do 10 i = 1, n
 10   continue
      end = counter()
      print *, end-start
      start = counter()
      do 20 i = 1, n
         f = f1 + f2
 20   continue
      end = counter()
      print *, end-start
      end
```

and in the meantime, provide a **Makefile** below:

```
ftime  :  counter.o ftime.o
          f77 counter.o ftime.o -o ftime
```

As you can see that, we use SUN SPARCstation 2 f77 compiler with default option. The rest is for machine, SPARCstation 2 knows how to link the object files together to create a executable file which can produce the cost of Fortran operation while using C library function.

In this way, we can get the click numbers for empty loop and for a statement. In order to be more precise, we perform the test five times and take the arithmetic average as the final execution time.

With the same reasoning, we can virtually get all execution time on SUN SPARCstation 2. We show the Fortran cost on SUN SPARCstation 2 in Table 3.9.

Note that in Table 3.9, T_i $(i = 1, 2, \cdots, 5)$ means the click numbers of each operation executing one million times. *Raw Time* means the seconds while executing one million times or microseconds while executing once, but with loop overhead (loop initilization overhead and loop branching overhead). Because we execute the statement one million times, the loop initializtion overhead can be ignored, so *Raw Time* indicates the time of the statement in question and loop branching overhead.

3.5 Empirical Cost Tables on SUN SPARCstation 2

In this section, we present the empirical cost table on SUN SPARCstation 2.

3.5.1 Assumptions

We now assume that basic assembly codes as **ld, sub, add, st, tst, mov, nop, bge, b, sll, sethi** etc. have the unit complexity cost 1. According to our experiments on SUN SPARCstation 2, it is a reasonable assumption and good tradeoff, for we do not need to get a little more precise while making enormous efforts. Based on that assumption, we can roughly claim that each above assembly operation costs 0.05 microseconds (μsec). Of course, we will show how we get that result immediately in the following Section 3.5.2. And equally, we prefer to give another result here: loop branching overhead is about 7, which is obtained in Section 3.5.6. We prefer to state the results here because we will use them throughout this section. In the next six subsections, we are going to figure out the costs on SUN SPARCstation 2: operation, memory, index, trigo, transcend and overhead.

3.5.2 Operation Costs

In Table 3.9, we have presented the elementary Fortran costs on SUN SPARCstation 2. Now in order to get precise costs, we need to have a look at the assembly code generated for each operation by the compiler f77 with default options.

Integer Costs on SUN SPARCstation 2

For integer add and minus, besides the branching overhead 7 instructions, there are only four instructions as loop body:

```
L50:
        ld      [%l7+-0xf9c],%o0
        ld      [%l7+-0xf98],%o1
        add     %o0,%o1,%o2
        st      %o2,[%l7+-0xfa0]
```

	$T1$	$T2$	$T3$	$T4$	$T5$	$Raw\ Time$
Null Operation						
	16	17	17	17	17	0.28
Integer Operation						
$i1 = i2 + i3$	30	31	30	31	30	0.51
$i1 = i2 - i3$	31	30	31	30	31	0.51
$i1 = i2 * i3$	69	66	68	65	69	1.12
$i1 = i2/i3$	64	61	63	61	59	1.03
$i1 = i2 - -i3$	30	30	30	30	31	0.50
$i1 = i2 * *i3$	58	57	57	57	58	0.96
Real Operations						
$r1 = r2 + r3$	36	37	36	36	37	0.61
$r1 = r2 - r3$	36	35	36	33	35	0.58
$r1 = r2 * r3$	38	37	37	37	35	0.61
$r1 = r2/r3$	56	57	57	55	54	0.93
$r1 = r2 - -r3$	37	38	39	37	38	0.63
$r1 = r2 * *r3$	431	430	432	430	431	7.18
Double Precision Operations						
$d1 = d2 + d3$	46	46	46	47	48	0.78
$d1 = d2 - d3$	48	46	45	46	45	0.77
$d1 = d2 * d3$	50	48	49	54	53	0.85
$d1 = d2/d3$	85	81	82	85	81	1.38
$d1 = d2 - -d3$	63	64	63	63	64	1.06
$d1 = d2 * *d3$	1037	1036	1037	1051	1042	17.34
Complex Operations						
$c1 = c2 + c3$	132	133	128	128	129	2.17
$c1 = c2 - c3$	129	128	128	129	128	2.14
$c1 = c2 * c3$	178	179	180	188	184	3.03
$c1 = c2/c3$	236	235	232	232	231	3.89
$c1 = c2 - -c3$	148	149	148	147	148	2.47
$c1 = c2 * *c3$	2304	2305	2302	2305	2304	38.40
Double Complex Operations						
$x1 = x2 + x3$	123	122	124	121	123	2.04
$x1 = x2 - x3$	122	123	121	126	124	2.05
$x1 = x2 * x3$	174	168	175	178	176	2.90
$x1 = x2/x3$	266	282	272	277	275	4.57
$x1 = x2 - -x3$	197	196	196	196	196	3.27
$x1 = x2 * *x3$	3040	3039	3049	3050	3049	50.76

Table 3.9: Elementary Fortran Cost on SUN SPARCstation 2

```
L62:
        ld      [%l7+-0xf9c],%o0
        ld      [%l7+-0xf98],%o1
        sub     %o0,%o1,%o2
        st      %o2,[%l7+-0xfa0]
```

As we can see clearly, there are two loads `ld`, one store `st` and the operation whose execution time we want to know. According to our assumption above, they are identical in term of cost, so there are 11 instructions in total. And from Table 3.9, we know they need 0.51μsec, so we can consider that each instruction needs about approximately 0.05μsec. Which means integer addition or subtraction costs one complexity unit, about 0.05μsec. So are other basic operations like **ld, st, tst, nop**, etc. This is the result we claimed above in Section 3.5.1. Until now, we think we justify that result. With this weapon in hand, we have no problem to obtain result for other operations.

For integer multiplication, the assembly codes are the following:

```
        ld      [%l7+-0xf9c],%o0
        ld      [%l7+-0xf98],%o1
        call    .mul,2
        nop
        st      %o0,[%l7+-0xfa0]
```

So from Table 3.9, we know that it needs 1.12 μsec with the loop branching overhead 7 units, 1.12 μsec is about 22 or 23 units, thus we can think that integer multiplication has the cost of 12 or 13 units. Meanwhile, we can conclude that the cost of `call     .mul,2` is about 12 units.

For integer division, the assembly codes are:

```
        ld      [%l7+-0xf9c],%o0
        ld      [%l7+-0xf98],%o1
        call    .div,2
        nop
        st      %o0,[%l7+-0xfa0]
```

With the same reasoning as for integer multiplication, and with a timing of 1.03 μsec, the cost for integer division is about 10.

For minus minus, the assembly codes are:

```
        ld      [%l7+-0xfc8],%o0
        sub     %g0,%o0,%o1
        ld      [%l7+-0xfcc],%o2
        sub     %o2,%o1,%o3
        st      %o3,[%l7+-0xff8]
```

We can see that except two loads `ld` and one store `st`, there are two `sub`s. So it has cost of two `sub`, so the cost is 2. But from Table 3.9, we know it needs 0.50μsec, the same as one `sub`, i.e., one complexity cost. It exists some discrepancy. So we believe its cost is between 1 and 2, we chose 2 unit costs.

And for power operation with 2 as power, the assembly codes are:

```
add     %l7,-0xfac,%o0
add     %l7,-0xfa8,%o1
call    _pow_ii,2
nop
mov     %o0,%l1
```

So with the same reasoning as stated above, the cost is about 9 units.

Floating-Point Costs on SUN SPARCstation 2

When we talk about floating-point operation, we mean that they are real operands, in Fortran, they are likely to be REAL*4. For addition and subtraction in floating-point operations, we have almost the same kind of assembly codes, so we consider them together as follows:

```
ld      [%l7+-0xf90],%f24
ld      [%l7+-0xf8c],%f25
fadds   %f24,%f25,%f26
st      %f26,[%l7+-0xf94]

ld      [%l7+-0xf90],%f31
ld      [%l7+-0xf8c],%f0
fsubs   %f31,%f0,%f1
st      %f1,[%l7+-0xf94]
```

They have the same amount of execution time, about 0.60 μsec, total 12 instructions, including loop branching overhead 7 units, 2 loads `ld`, one store `st` and `fadds`. So `fadds` is about 2, and so is `fsubs`.

For floating-point multiplication, we can get assembly code as:

```
ld      [%l7+-0xf90],%f6
ld      [%l7+-0xf8c],%f7
fmuls   %f6,%f7,%f8
st      %f8,[%l7+-0xf94]
```

The execution time is 0.63 μsec, about 12 units, so with the same reasoning, `fmuls` has about 2 unit costs.

For real division, assembly code is as follows:

```
ld      [%l7+-0xf90],%f13
ld      [%l7+-0xf8c],%f14
fdivs   %f13,%f14,%f15
st      %f15,[%l7+-0xf94]
```

The execution time is 0.93 μsec, about 18 units, so `fdivs` is about 8 units.

For double subtraction, assembly code is as follows:

```
ld      [%l7+-0xfa0],%f4
fnegs   %f4,%f5
ld      [%l7+-0xfa4],%f6
fsubs   %f6,%f5,%f7
st      %f7,[%l7+-0xfa8]
```

We can see clearly that cost for double subtraction is cost of `fsubs` i.e., 2, plus cost of `fnegs` i.e., 1. Then the final result is 3.

For floating-point power, assembly code is as follows:

```
add     %l7,-0xfa4,%o0
add     %l7,-0xfa0,%o1
call    _pow_rr,2
nop
st      %f0,[%l7+-0xfa8]
```

From Table 3.9, we know the raw time is 7.18 μsec, about 143 units, so 140 units for floating-point power.

Double Precision Costs on SUN SPARCstation 2

For double precision, we can think that they need twice as much time as single precision, so we can assume that `ld2` has double cost of `ld`, that is `ld2` is 2, and so is `st2`. So we will put these values as memory access cost, which will be introduced in the next subsection.

For addition and subtraction, the assembly code for double precision is as follow:

```
ld2     [%l7+-0xfb8],%f20
ld2     [%l7+-0xfa8],%f22
faddd   %f20,%f22,%f24
st2     %f24,[%l7+-0xfc8]
```

Reasonably, the raw execution time is about 0.78 μsec, about 15 units cost, we need to subtract the loop branching overhead 7, so we get that `faddd` is about 2. Same thing for `fsubd`, which is about 2.

For multiplication, the assembly code is

```
ld2     [%l7+-0xfb8],%f8
ld2     [%l7+-0xfa8],%f10
fmuld   %f8,%f10,%f12
st2     %f12,[%l7+-0xfc8]
```

The raw execution time is 0.86μsec, about 17 units, so `fmuld` is about 4.

For division, the assembly code is:

```
ld2     [%l7+-0xfb8],%f18
ld2     [%l7+-0xfa8],%f20
fdivd   %f18,%f20,%f22
st2     %f22,[%l7+-0xfc8]
```

The raw execution time is 1.38 μsec, about 27 units, so `fdivd` is about 14.

For double subtraction, the assembly code is:

```
ld2     [%l7+-0xfb0],%f12
fnegd   %f12,%f12
ld2     [%l7+-0xfc0],%f14
fsubd   %f14,%f12,%f16
st2     %f16,[%l7+-0xfd0]
```

Here we think it is the addition of two costs `fnegd` and `fsubd`, So double precision double subtraction cost is 8 units.

For power operation with 2 as power, the assembly code is:

```
add     %l7,-0xfc0,%o0
add     %l7,-0xfb0,%o1
call    _pow_dd,2
nop
st2     %f0,[%l7+-0xfd0]
```

With same reasoning, so double precision power cost is 342 units.

Complex Costs on SUN SPARCstation 2

For complex addition, the assembly code is:

```
add     %sp,LP1,%o0
add     %l7,-0xfc0,%o1
add     %l7,-0xfb0,%o2
call    __Fc_add,3
nop
add     %sp,LP1,%o0
add     %l7,-0xfd0,%o1
ld      [%o0],%o2
ld      [%o0+4],%o4
st      %o2,[%o1+0]
st      %o4,[%o1+4]
```

The raw execution time is 2.17 μsec, about 43 units. Here we do not need to calculate the cost for call, we can think that there two `ld2` and one `st2`, along with complex add. Hence we get $2 * ld2 + st2 + X + 7 = 43$, and $X = 30$. So we think the cost of complex addition is about 30.

The same reason for complex minus, it is about 32. We do not need to list all of these assembly codes, we only give the final results. So complex multiplication is about 47, complex division is about 64, complex double minus is 36 and complex power is 755.

#	int	float	double	complex	double-complex
+	1	2	2	32	28
−	1	2	2	32	28
*	12	2	4	47	45
/	10	8	14	64	78
−−	2	3	8	36	50
**	9	140	342	755	1000

Table 3.10: Operation Costs on SUN SPARCstation 2

#	int	float	double	complex	double-complex
MEMORY-LOAD	1	1	2	2	2
MEMORY-STORE	1	1	2	2	2
=	0	0	0	0	0

Table 3.11: Memory Costs on SUN SPARCstation 2

Double Complex Costs on SUN SPARCstation 2

We only give results here. The raw execution time for double complex add is 2.04 μsec, about 41 units, so we use the same reason as above, the double complex add is about 28, so is minus. Double complex mult is about 45, and double complex division is about 78. Double complex minus 2 is 50 and power is 1000.

Operation Cost Summary

In this subsection, we have obtained the operation costs on SUN SPARCstation 2. The operation cost file of the machine is shown in Table 3.10.

3.5.3 Memory Access Costs

The memory access costs include memory load and store. It is pretty easy to get the cost on SUN SPARCstation 2. In fact, we did get these costs in the previous subsection as by-products. The results are shown in Table 3.11.

3.5.4 Intrinsic Costs

We use the same technique here and we can get the costs on SUN SPARCstation 2. If the type of argument cannot be passed by the SUN Fortran compiler, such as complex argument for MIN, we just ignore it and use a dash sign to denote that this cost is not available. Of course, it does not include all the functions available, we can add more functions in test as needed. The costs are shown in Table 3.12.

#	int	float	double	complex	double-complex
MIN	16	18	26	-	-
MAX	16	18	26	-	-
SQRT	-	122	152	358	494
DSQRT	-	-	140	-	-
CSQRT	-	-	-	358	-
EXP	-	56	88	440	370
DEXP	-	-	86	-	-
CEXP	-	-	-	382	-
ALOG	-	100	-	-	-
DLOG	-	-	66	-	-
CLOG	-	-	-	348	-
ALOG10	-	176	-	-	-
DLOG10	-	-	160	-	-

Table 3.12: Intrinsic costs on SUN SPARCstation 2

The costs for trigonometric are shown in Table 3.13.

3.5.5 Reference Costs

When an element of an array is accessed, the way the compiler generates code to access that element depends on the array declaration, which means whether the array is declared locally, passed as argument of the subprogram or passed as a global memory, i.e., in "common" declaration. We will investigate these three cases one by one and give the verification results on SUN SPARCstation 2 with SunOS f77 compiler.

In this subsection, we use c to indicate constant in the *Case* column of Tables, and i to indicate varying index. We are always using SUN Fortran compiler f77 with default options.

Locally Declared Array

For the one-dimension floating-point array $a1$, we have three kinds of overheads.

1. index is a constant, as $a1(4)$. We can find out that the instruction $a1(4) = 1.2$ needs 4 assembly instructions.
2. index depends on loop induction variable i, as $a1(i)$; the memory access is about 5 units.
3. index is a fixed number but expressed in symbolic variable; i.e., $k = 10$ is set, while we stick to using $a1(k)$ to access that memory instead of $a1(10)$. In this case, the memory access is about 6.

I do not think that the third case is representative, so I only consider the first two cases because they happen much more frequently in the real code. The result is shown in Table 3.14, for the first two rows.

#	int	float	double	complex	double-complex
SIN	-	92	108	736	-
DSIN	-	-	108	-	-
CSIN	-	-	-	736	-
COS	-	92	131	742	-
DCOS	-	-	132	-	-
CCOS	-	-	-	742	-
TAN	-	110	147	-	-
DTAN	-	-	147	-	-
ASIN	-	300	280	-	-
DASIN	-	-	339	-	-
ACOS	-	401	356	-	-
DACOS	-	-	358	-	-
ATAN	-	31	36	-	-
DATAN	-	-	82	-	-
ATAN2	-	151	233	-	-
DATAN2	-	-	230	-	-
SINH	-	310	290	-	-
DSINH	-	-	290	-	-
COSH	-	172	152	-	-
DCOSH	-	-	152	-	-
TANH	-	307	288	-	-
DTANH	-	-	288	-	-

Table 3.13: Trigonometric Costs on SUN SPARCstation 2

For multidimensional arrays, the memory locations used are also consecutive. In fact, it is the first subscript which varies most rapidly, so-called "left-most" or "column-wise" array. The result is shown in Table 3.14.

In the Table 3.14, we list until three dimensional array. Of course, we can give result of array with more dimensions if needed. Note that in the Table 3.14, *Dim* means dimension, *Case* means whether the reference is constant or not, and with the order, i.e., *ci* means the first reference is constant while the second one is varying reference, likely a loop induction variable. Number of core instruction means the how many assembly instructions are needed to access that array element. For instance, because of Fortran "column-wise" characteristics, access to $a2(3,i)$ is about 11 instruction units while $a2(i,3)$ is only 5, that accounts for the importance of the reference order. *Estimated* means that, if we adopt the number of core instruction, the theoretical result of that array element access time. Note that we use 10 as exterior loop counts, the same is true for Experiment values. *Experiment* means the real clock time, the unit is μsec. And finally we give the *Difference* between the *Estimated* and *Experiment*. We need to calibrate the result we get, so we add the *Corrected No.* to mean the calibrated number of core instructions, which is used as standard cost table elements.

Dim	Case	N of Core Ins.	Estimated	Experiment	Difference(%)	Corrected No.
1	c	4	6.10	6.07	0.49	4
1	i	5	6.55	6.94	-5.62	5
2	cc	4	64.55	69.87	-7.61	5
2	ci	11	99.55	89.50	11.23	10
2	ic	5	69.55	74.37	-6.48	6
2	ii	13	109.55	94.48	15.95	11
3	ccc	5	699.55	694.47	0.73	5
3	cci	13	1099.55	954.00	15.26	11
3	cic	11	999.55	908.68	10.00	10
3	cii	18	1349.55	1050.68	28.45	14
3	icc	5	699.55	745.68	-6.19	6
3	ici	16	1249.55	995.68	25.50	13
3	iic	15	1199.55	996.00	20.44	13
3	iii	20	1449.55	1100.33	31.74	14

Table 3.14: Array Index Costs - Locally Declared

Dim	Case	N of Core Ins.	Estimated	Experiment	Difference(%)	Corrected No.
1	c	4	6.90	7.28	-5.22	5
1	i	6	7.90	8.06	-1.99	6
2	cc	13	110.90	107.26	3.39	13
2	ci	22	155.90	157.76	-1.18	22
2	ic	10	95.90	99.19	-3.32	10
2	ii	22	155.90	158.52	-1.65	22
3	**c	25	1802.90	1975.24	-8.73	25
3	**i	38	2552.90	2561.34	-0.33	38

Table 3.15: Array Index Costs - Passed by Arguments

Array Passed by Argument

If the size is hidden, e.g., the size is passed by the argument, in the local subroutine, the compiler knows nothing about the array effectively declared and its size, so it will need more time to compute the offset. The result is shown in Table 3.15.

In Table 3.15, the column meaning is the same as explained for the "Locally Declared" reference. The difference is that for three-dimensional array, only the third reference of array can make difference. In the table, we use $*$ to denote both c and i. That is, no matter what the reference is, the result remains the same.

Dim	Case	N of Core Ins.	Estimated	Experiment	Difference(%)	Corrected No.
1	c	4	6.60	7.28	-9.34	5
1	i	5	7.10	8.06	-11.91	6
2	cc	4	65.25	70.66	-7.66	5
2	ci	11	100.25	90.83	10.37	10
2	ic	4	65.25	75.66	-13.76	5
2	ii	13	110.25	100.30	9.92	12
3	ccc	4	650.40	706.67	-7.96	5
3	cci	15	1200.40	1003.99	19.56	13
3	cic	11	1000.40	902.33	10.87	9
3	cii	16	1250.40	1015.66	23.11	13
3	icc	5	700.40	751.66	-6.82	5
3	ici	18	1350.40	1078.99	25.15	14
3	iic	15	1200.40	1012.33	18.58	13
3	iii	18	1350.40	1084.67	24.50	14

Table 3.16: Array Index Costs - Passed by Global Memory

Array Passed by Global Memory

If the array is passed by global memory, such as COMMON in Fortran, then the cost for array access is shown in Table 3.16.

In Table 3.16, the column's meaning is the same as explained for the "Locally Declared" reference earlier.

Index Cost Summary

In this subsection, we have obtained the index costs on SUN SPARCstation 2. The index cost file of the machine with SUN Fortran compiler f77 along with default options, is shown in Table 3.17. The difference between cost in Table 3.17 and the measurement is less than 25%.

3.5.6 Loop and Call Overheads

In order to begin our experiments, we need to use loop to time certain operations. So we need to know the loop overheads first. Let us see the example shown in Figure 3.2.

There are two kinds of loop overheads, one is the loop initialization overhead, the other is the loop branching overhead. The loop initialization overhead is encountered only once when the loop is reached, and loop branching overhead is the overhead when the loop body reaches its end and goes over to the very beginning where the condition-checking is performed to see whether to execute the loop body one more time or not.

Dim	Case	Local	Formal	Global
1	c	4	5	5
1	i	5	6	6
2	cc	5	13	5
2	ci	10	22	10
2	ic	6	10	5
2	ii	11	22	12
3	ccc	5	25	5
3	cci	11	38	13
3	cic	10	25	9
3	cii	14	38	13
3	icc	6	25	5
3	ici	13	38	14
3	iic	13	25	13
3	iii	14	38	14

Table 3.17: Summary of Array Reference Costs on SUN SPARCstation 2

```
subroutine matinit(c, size, l1, l2, n, length)
      integer size
      real c(size,size)
      integer count
      integer l1,l2
      integer tsum, start, t, lend, length(5)
      integer j,i,k1,k2
C
C     loop overhead
      tsum = 0
      do j = 1, 5
         start = count()
         do i = 1, n
            do k1 = 1, l1
               do k2 = 1, l2
                  c(k1,k2) = 0.
               enddo
            enddo
         enddo
         lend = count()
         t = lend - start
         length(j) = t
      enddo
      end
```

Figure 3.2: Loop Overhead Detecting Program

Loop Initialization Overhead

There are nested loops here, from outside to inside. But they have one thing common, they have same initialization overhead. We present the assembly code here:

```
L26:
        ld      [%fp+0x4c],%o0
        ld      [%o0],%o1
        st      %o1,[%fp+-0x24]
        mov     0x1,%l5
        ld      [%fp+-0x24],%o0
        sub     %o0,%l5,%o0
        st      %o0,[%fp+-0x24]
        ld      [%fp+-0x24],%o0
        tst     %o0
        bge     L38
        nop
        b       L32
        nop
```

We consider that the loop should have several iterations, so branching to L32 is little probability event. Generally it is highly unlikely. Because we already assume all these instructions have the unit cost 1, so we consider that the loop initialization overhead is 11, we just exclude the absolute branch statement to L32 and following no-operation instruction.

Loop Branching Overhead

```
L39:
        add     %l5,0x1,%l5
        ld      [%fp+-0x24],%o0
        sub     %o0,0x1,%o0
        st      %o0,[%fp+-0x24]
        tst     %o0
        bge     L33
        nop
        b       L32
        nop
```

Same thing again here. The branching instruction to L32 only happens once as exit of the loop, we do not take it into account. So the loop branching overhead is 7.

Call Overhead

When there is a call in Fortran program, it needs extra costs to pass the arguments, initiate the subroutine and/or function. So we want to know these costs.

Fortunately, Fortran is call-by-reference language, only reference is passed, that is, when arguments are passed, only their addresses are transferred. Hence, we do not need to consider the types of the

	T1	T2	T3	T4	T5	Raw	Net
Null Operation							
{}	17	16	17	16	17	0.277	0.000
call 0	37	38	37	38	38	0.627	0.350
call 1	44	44	44	44	44	0.733	0.457
call 2	60	49	50	59	49	0.890	0.613
call 3	76	58	57	58	57	1.020	0.743
call 4	88	70	69	69	70	1.220	0.943
call 5	104	78	79	83	79	1.410	1.133
call 6	115	90	91	91	91	1.593	1.317
call 7	127	117	100	127	101	1.907	1.630

Table 3.18: Call Overheads on SUN SPARCstation 2

Number of Argument	Unit Cost
0	7
1	9
2	12
3	15
4	19
5	23
6	26
7	37

Table 3.19: Call Subroutine Overheads

argument, only pointers are taken into account. We present our result of call invocation overhead in Table 3.18.

The overhead was measured for difference number of arguments. Note in Table 3.18 that *Raw* means execution time along with the loop overhead, while Net is the time without loop overhead. We also found that no simple formula can cope with the sudden increase when 7 arguments are passed. This is probably due to SPARC register window. We did not find many calls with more than 7 arguments in real program. We show all loop overhead in Table 3.19.

Return Overhead

There are always two assembly instructions for Fortran *return* statement. So we think that it has 2 complexity units.

Overhead Summary

In this subsection, we have obtained the overhead costs on SUN SPARCstation 2. The overhead cost file of the machine is shown in Table 3.20.

#	Cost
LOOP-INIT-OVERHEAD	7
LOOP-BRANCH-OVERHEAD	11
CALL-ZERO-OVERHEAD	7
CALL-ONE-OVERHEAD	9
CALL-TWO-OVERHEAD	12
CALL-THREE-OVERHEAD	15
CALL-FOUR-OVERHEAD	19
CALL-FIVE-OVERHEAD	23
CALL-SIX-OVERHEAD	26
CALL-SEVEN-OVERHEAD	33
RETURN-OVERHEAD	2

Table 3.20: Overhead on SUN SPARCstation 2

These results are consulted when our estimator encounters the construct whose overhead is listed above.

3.5.7 Summary

In this section, we have described the dynamic cost tables on SUN SPARCstation 2. Based on SUN SPARCstation 2, we present a way of how to combine the execution time and assembly codes to obtain the corresponding cost tables, finally a set of complete cost tables on SUN SPARCstation 2 is presented.

We will verify the predictive power of these empirical cost tables in Chapter 5. They were obtained experimentally. They showed that the present implementation in PIPS is much too coarse but I did not have time to reprogram the estimator interface with the cost tables.

3.6 Discussion

Our model has drawbacks related to the compiler and language.

3.6.1 Problems with Compiler

It is a fact of life that virtually all programs that are run on today's supercomputers are written in a high-level language, and in almost every case that language is Fortran for scientific computing. Since underlying model of Fortran (single type of storage, sequential execution) is so unlike that of any modern supercomputer (multiple-level memory hierarchy, multiple processors, vector operations, etc.), the job of fitting the program to the machine falls onto the compiler. There is a hierarchy of levels at which optimizations can be performed on these programs. At the lowest level, single operations or statements are examined. The subsequent steps include optimization within basic blocks, optimization of the flow graph, and interprocedural analysis. We can cope with the former by multiplying the entries in cost tables, but we cannot take the later into accounts.

3.6.2 Fortran vs. C

During the experiments, we have observed an interesting situation: the same program is running considerably faster when it is coded in Fortran than in C. For instance, we tested empty loops and floating-point additions:

```
main()
{
      int i,n;
      float f1,f2,f3;
      int start, end;
      n = 1000000;
      f2 = 1.2;
      f3 = 3.4;

      start = counter_();
      for ( i=0; i<n; i++ )
            ;
      end = counter_();
      printf("click is %d\n", end-start);

      start = counter_();
      for ( i=0; i<n; i++ )
            f1 = f2 + f3;

      end = counter_();
      printf("click is %d\n", end-start);
}
```

```
      program fna
      integer counter
      integer i,n
      real f1,f2,f3
      integer start,end
      n = 1000000
      f2 = 1.2
      f3 = 3.4

      start = counter()
      do 10 i = 1, n
10    continue
      end = counter()
      print *, 'click is ', end-start

      start = counter()
      do 20 i = 1, n
           f1 = f2 + f3
20    continue
      end = counter()
      print *, 'click is ', end-start
      end
```

After running the program, we get for C program, 26 and 60 clicks, for null loop and floating point addition (with null loop overhead), for Fortran program 17 and 42 clicks. In order to explain the difference, let us have a look at the assembly codes of both programs. excerpts:

The C program is 99 lines long. The two loops are:

```
L22:
        ld      [%fp+-0x4],%l0
        ld      [%fp+-0x8],%l1
        cmp     %l0,%l1
        bge     L21
        nop
        ld      [%fp+-0x14],%f2
        fstod   %f2,%f4
        ld      [%fp+-0x10],%f6
        fstod   %f6,%f8
        faddd   %f8,%f4,%f10
        fdtos   %f10,%f11
        st      %f11,[%fp+-0xc]
L20:
        ld      [%fp+-0x4],%l2
        add     %l2,0x1,%l2
```

```
        st      %l2,[%fp+-0x4]
        b       L22
        nop
L21:
        call    _counter_,0
```

The Fortran program: 223 lines long and the assembly codes are:

```
L34:
L29:
        ld      [%l7+-0xff4],%f0
        ld      [%l7+-0xff0],%f1
        fadds   %f0,%f1,%f2
        st      %f2,[%l7+-0x1000]
L27:
        add     %l6,0x1,%l6
        ld      [%fp+-0x30],%o0
        sub     %o0,0x1,%o0
        st      %o0,[%fp+-0x30]
        tst     %o0
        bge     L29
        nop
        b       L28
        nop
L28:
        call    _counter_,0
```

Although the assembly code of Fortran program is larger than the one of C program, but Fortran program runs relatively faster than C program. In this example, the floating point addition in Fortran takes about 25 clicks, about 0.42μsec while the same code in C needs 34 clicks, about 0.57μsec, 35% more time is needed. That accounts more or less for usefulness of Fortran in the scientific programming world.

3.7 Summary

In this chapter, we have introduced a methodology to represent target machine characteristics.

We first looked at the characteristics of sequential and parallel machines. Then we presented our two implemented *all-1* and **fp-1** models. Finally we also introduced how to get the elementary operations costs on machines, and specifically, we obtained experimentally the elementary Fortran costs on SUN SPARCstation 2 as well as some elementary C costs.

Meanwhile, we also mentioned some problems with SUN compilers.

Chapter 4

Implementation

The approach used in this implementation is to analyze each module (i.e., program, subroutine, function) once and to accumulate elementary execution time for operations, memory accesses, intrinsic functions, etc. We chose this approach because it is both efficient enough to allow us to run it since its own complexity is approximately linear with respect to the module size, and machine-independent since dependencies are factored in the cost tables as explained in Chapter 3.

We first introduce the PIPS programming environment, and later we detail the components of our complexity estimator. The PIPS data structures used by the complexity estimator include the call graph, control flow graph, preconditions and effects. Then we explain how the complexity program works, its interfaces with terminal and X-windows, and finally its main features.

4.1 Environment

PIPS is written principally in C language under UNIX environment. On top of the UNIX operating system, NewGen and various utilities from the PIPS programming environment are also used.

4.1.1 NewGen

NewGen is a software engineering tool that helps programmers define and implement sophisticated data types. Data domains are defined with a high level specification language, called DDL (Domain Definition Language) [JT89] that allows user define domains to be built over *basic* domains with operators like Cartesian product, union, list or array.

NewGen analyzes a set of specifications and produces a collection of macros, functions, predicates, and pre-defined constants that enable programs to create, initiate, access, update and delete objects of these types as if the programming language had been especially tailored to manipulate these data types.

NewGen is used as the basis of PIPS. All major data structures of PIPS are declared via NewGen.

4.1.2 PIPS

The whole complexity estimator program is a part of PIPS project. All work is done in PIPS programming environment.

We have defined several environmental variables needed in PIPS programming environment. There are two files: *pipsrc.sh* for the standard shell and *pipsrc.csh* for the C-shell environment. COSTDIR environment variable is set there, it contains the name of the directory where cost tables are stored. In that directory, several kinds of cost tables which are detailed in Section 4.6.2 must be formed for the complexity analyzer to work.

An input program is a set of user source files. They are split by PIPS during the program initialization phase to produce a PIPS-private source file for each procedure, subroutine or function. Each module is analyzed once by the phase. A set of modules is processed in a bottom-up order of the call tree.

4.2 Complexity in PIPS

After this quick introduction to the PIPS development environment, we present the implementation of our static prediction method as a PIPS phase, called *complexity* in PIPS environment.

We now describe the data structures used to represent the complexity of a set of instructions. In this Chapter, we interchangeably use the words complexity, execution time and costs, although we can tell the difference: cost is an elementary execution time while complexity is a formula mapping an environment σ to a set of possible execution time.

The complexity data structure of PIPS is composed of two elements: a polynomial and statistics counters about unknown information. We introduce them in the following.

4.2.1 Polynomial Form

We choose to internally express complexities as symbolic polynomials over program variables and unknown values as we explained earlier in Section 1.4.3.

The polynomial library has been developed by several people of CRI, especially Pierre Berthomier. It is a component of C3 library, which has been under development since 1986 at CRI with CNRS C3 funding. It is made from the monomials, which are made from vectors. The vector library is the essential part of C3 library.

We define a polynomial as a list of monomials and monomial as a list of vectors. Now we have a tour to see these basic data structures.

Vector Data Structure

The vector is composed of three elements, a variable, a value and a pointer points to its own data structure. We give the definition:

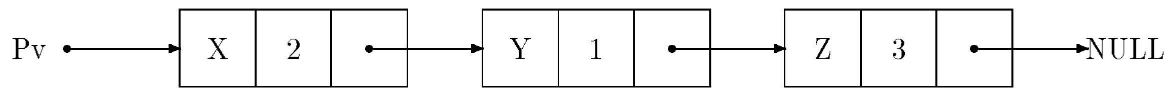


Figure 4.1: Data Structure of Vector in PIPS

```
typedef struct Svecteur {
    Variable var;
    Value val;
    struct Svecteur *succ;
} Svecteur,*Pvecteur;
```

Generally speaking, `Variable` is defined as a pointer to char, and `Value` is defined as an integer.

Example:

The vector $\vec{V} = 2 \cdot \vec{X} + \vec{Y} + 3 \cdot \vec{Z}$ is shown in Figure 4.1 where Pv is a pointer to the vector $\vec{V}$.

Monomial Data Structure

Monomial consists of two elements, one is coefficient of the monomial, and the other is a pointer to the corresponding vector. We present the definition below:

```
typedef struct Smonome {
    float coeff;
    Pvecteur term;
} Smonome, *Pmonome;
```

There is a small difference between the vector of monomial and vector itself. In vector, the `Value` is coefficient of the variable, but in monomial, it becomes the exponent. So the data structure representing vector $\vec{V} = 2 \cdot \vec{X} + \vec{Y} + 3 \cdot \vec{Z}$ is interpreted as the monomial X^2YZ^3.

Example:

The monomial $2 \cdot X^2YZ^3$ is represented in Figure 4.2. Pm is a pointer to this monomial.

Polynomial Data Structure

Polynomial consists of two elements, one is a pointer to monomial, the other is a pointer to another polynomial. We implement polynomials as lists of monomials. The C definition is as follows:

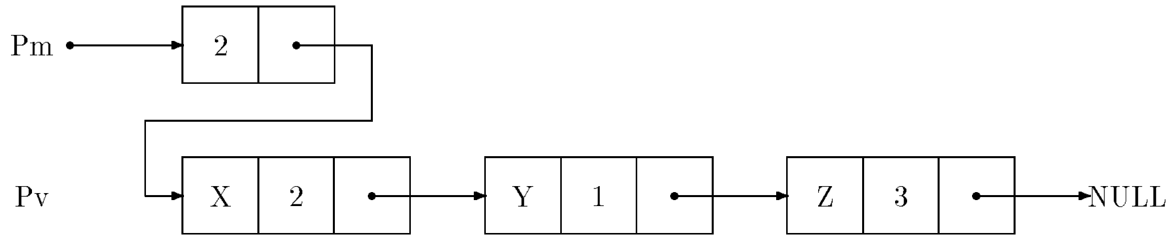


Figure 4.2: Data Structure of Monomial in PIPS

```
typedef struct Spolynome {
    Pmonome monome;
    struct Spolynome *succ;
} Spolynome, *Ppolynome;
```

Example:

Given a polynomial $2 \cdot X^3YZ^2 + 6XY^2Z^3 - 3$, see Figure 4.3 where Pp is the pointer to the polynomial, Pm is the pointer to the monomial and Pv is the pointer to the vector.

In the Figure 4.3, there is a special variable TCST, which stands for CONSTANT TERM. Consequently, the exponent is 1.

4.2.2 Counters

Although it is almost impossible to get the precise execution time of a given set of instructions, we can expect a good approximation. The information of the accuracy of the computed polynomial, with respect to the real complexity of statement, is included in the complexity data structure. The statistics counters contain three kinds of counters to summarize the different sorts of approximations which can be performed during the evaluation: variable, range and if counters.

Variable Counters

When evaluating the complexity of an expression, we do not care about the values of the variables, but only about the operation costs. On the contrary, when scanning loop bounds, we want to know the value a variable can have in loop expressions, in order to evaluate the number of iterations performed. We distinguish four counters:

1. **symbolic** counts the variable which appears literally and needs not to be evaluated. What we need to know is its data type, e.g., integer or real. Symbolic variables include the variables selected by users in order to keep them symbolic, e.g., parameters read from a disk configuration file, loop indices and formal parameters of a function.

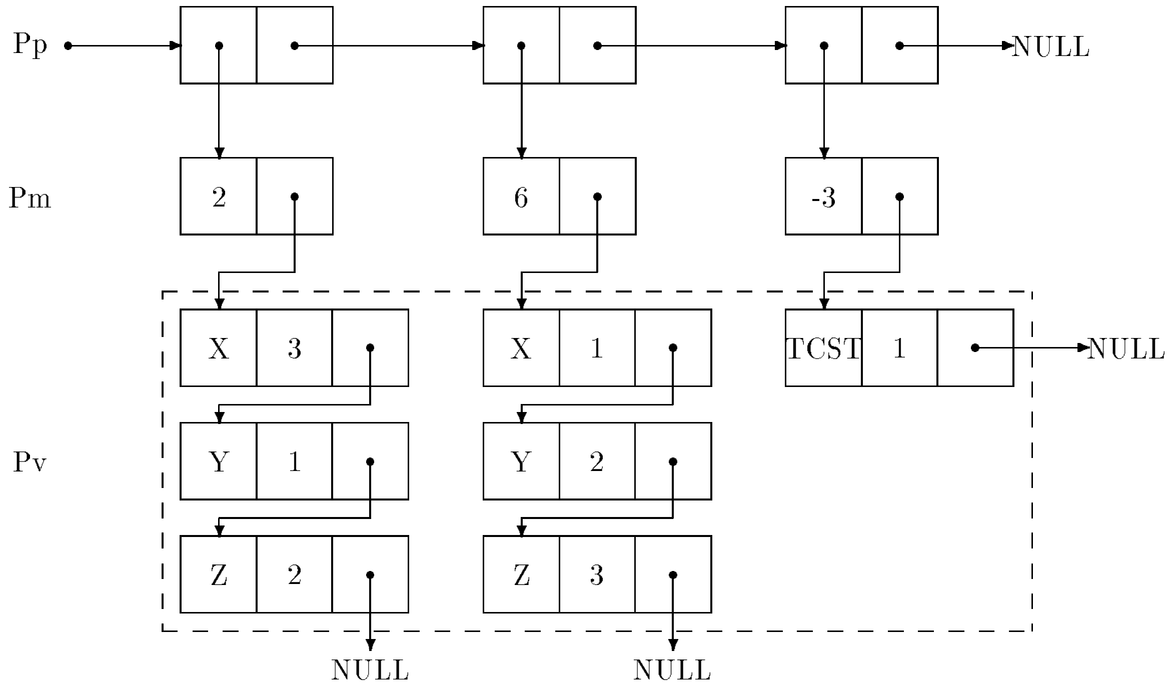


Figure 4.3: Data Structure of Polynomial in PIPS

2. **guessed** counts the variables which had to be evaluated and whose value could not be found.
3. **bounded** counts the variable which has to be evaluated and whose value cannot be found exactly, where the prediction are used. In this case, we chose the worse estimation.
4. **unknown** counts the variables which are totally unknown.

Range Counters

A bad evaluation of the range of a loop can lead to catastrophic results. The following counters give an idea about it, especially the percentage of *unknown* range:

1. **profiled** counts the loop range whose range is measured with some profiling. (not implemented)
2. **guessed** counts the loop range whose bound contains variable of the kind *symbolic* and *guessed*.
3. **bounded** counts the number of loop ranges whose bound contain variables of the kind *bounded*.
4. **unknown** counts the number of loop ranges which are totally unknown.

If Counters

A wrong evaluation of a test probability can also lead to a completely wrong result. For example, the WHILE structure implemented with a backward `GOTO`. Typically boolean could be a convergence test in a numerical program. So the way those probabilities were evaluated must included in the complexity information:

1. **profiled** counts the test whose probability was measured. (not implemented)
2. **computed** counts the test whose probability was computed. This may happen if the boolean expression is a function of the indices of surrounding loops; in this case, the probability is the ratio of two polyhedral volumes.
3. **halfhalf** counts the test that we know nothing about and whose probability was supposed to be fifty-fifty.

Summary of Counters

These counters are not displayed in the current implementation. They are not necessary if unknown variables are introduced each time, they are theoretically required. They are useful if the user is willing to control the approximations made by because he wants a simple formulae or a numerical value as complexity.

4.3 Prerequisites for Complexity

Complexity program cannot work without the following supporting data structures: call graph, interprocedural control flow graph, preconditions and effects. We introduce them one after the other in greater detail below.

4.3.1 Call Graph

Call-graph is the tree[1], which describes the calling relations between the procedures. There is the definition of a call graph:

Definition 4.1 *The Call Graph G of a set of modules N is a directed graph G=(N,E), where each subprogram is denoted by a node, N denotes the set of nodes and E denotes the set of edges. There is one-to-one correspondence in G between N and Q, and (p,q) $\in$ E and p,q $\in$ Q iff subprogram p contains a call to q, and execution of p may lead to the activation of subprogram q.*

Example:

[1] No recursivity is allowed in standard Fortran

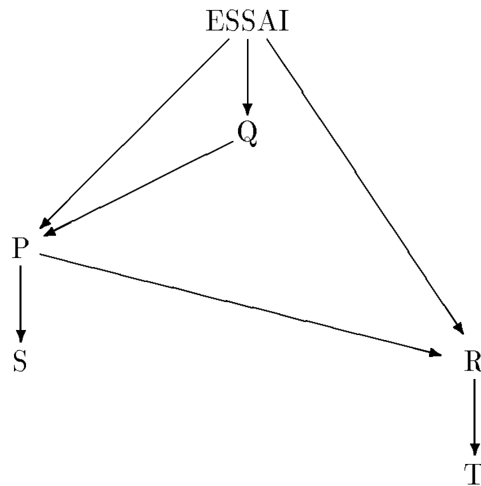


Figure 4.4: Calling relation of ESSAI

```
      program essai
      integer i, j, k
      do i=1, 1000
      if (i.gt.500) call P(i)
      enddo
c
      call P(10)
c
      j=1
      call Q(j)
c
      k=1
      call R(k)
c
      end
```

```
      subroutine P(x)
      call R(x)
      x = 1 + S(0,1)
      end

      subroutine Q(y)
      call P(y+1)
      end

      subroutine R(z)
      do k=1, 100
           z=z*2
      enddo
      call T(z)
      end
```

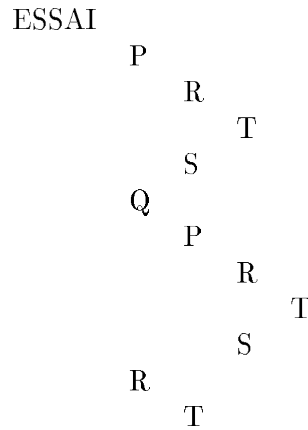
```
      function S(a,b)
      S = (a + 3) / b
      end

      subroutine T(w)
      w = w*w
      end
```

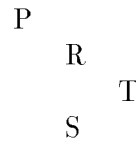
Since P, Q and R are called by ESSAI, so they are all callees of ESSAI. Hence, R and S are callees of P, and in the meantime P is callee of Q and ESSAI, R has only one callee T while S has none.

The calling relation of ESSAI is shown in Figure 4.4.

The PIPS's display ESSAI's callgraph is:



If you prefer see them separately, for example, if you just want to see the callees of P, it is feasible and call trees are calculated automatically.



This shows clearly that if we want to calculate the complexity of P, we first need to know the complexity of R and S. likewise T's complexity is required in order to know the complexity of R.

4.3.2 Interprocedural Control Flow Graph

Interprocedural Control Flow Graph (henceforth abbreviated ICFG) is a little complicated than Call-graph, because loop and branch statements are involved. Each call will produce an edge from caller to callee. It is defined as follows:

Definition 4.2 *The Interprocedural Control Flow Graph G of a Program Q is a directed graph G=(N,E,α), where Q may contain several subprograms and each subprogram is denoted by a node, N denotes the set of nodes and E denotes the set of edges, α is the initial node. There is a path from α to every node of G. There exists an edge $(p,q) \in E$ and $p,q \in N$ iff subprogram p contains a call to q, and execution of p may lead to the activation of subprogram q.*

We use the same example above to illustrate the difference between the call graph explained above and ICFG. If we only want to ICFG without loop and branch. The three kinds of ICFGs are shown in the following:s

ICFG only:

```
ESSAI
    P
        R
            T
        S
    P
        R
            T
        S
    Q
        P
            R
                T
            S
    R
        T
```

ICFG with loop:

```
ESSAI
    DO
        P
            R
                T
            S
    ENDDO
    P
        R
            T
        S
    Q
        P
            R
                T
            S
    R
        T
```

ICFG with control:

```
ESSAI
    DO
        IF
        THEN
            P
                R
                    T
                S
        ELSE
        ENDIF
    ENDDO
    P
        R
            T
        S
    Q
        P
            R
                T
            S
    R
        T
```

The call graph and ICFG display facilities in PIPS were implemented by the author. They are routinely used by people who used to take a guide look at an unknown application. However, only the interprocedural control flow graph is used to recursively evaluate the complexity of a module. The interprocedural ordering is managed by pipsmake. See Section 2.1.3 for more.

The difference between call graph and ICFG is obvious: call graph is concerned only with calling relations among the modules, while for ICFG, in addition to calling relations, it is also concerned with calling times and structure. As shown in the both graphs of our example, ESSAI calls P twice, this relation is displayed in ICFG, while not in call graph. If we use ICFG with loop and/or control, we can have more information.

4.3.3 Preconditions

We have mentioned preconditions in Section 2.1.8, we want to introduce it a step further. PIPS's precondition is information about scalar variables available before a statement is executed. The variable can be defined prior to the current statement, or the value of variable can be passed by the COMMON variable or even by the arguments of a subprogram. For example, a variable is assigned to a certain constant somewhere in the program, and is not modified again in the rest of the program. We can assume that this variable is known for each statement in the rest of program. This information is carried by preconditions. A precondition is computed for each statement.

We will show how we exploit that information in Section 4.7.1.

4.3.4 Effects

We have mentioned effects in Section 2.1.6. Effects are used to decide where we must evaluate the variable at some time, especially when trying to delay the evaluation to reduce the number of unknown variables used in complexity.

We will show how that information is exploited in Section 4.6.3.

4.4 Description of Complexity Evaluation Code

The complexity evaluation process is a bottom-up traverse of the PIPS abstract syntax tree of the program. It recursively computes the complexity associated with each program sub-structure, from the expressions and assignments up to the main program level, through all the possible structures, such as loop, test, unstructured control flow graph, etc.

4.4.1 C File Descriptions

There are nine files in the complexity development directory:

`comp_scan.c` This file contains all the major scan functions using the Abstract Syntax Tree of a program to analyze the module and count elementary operations.

`main.c` This is only used for debugging purposes, complexity user interface in **tpips** or **wpips**. See Section 4.5 for more.

`comp_prettyprint.c` This file controls how to prettyprint the complexity estimation format of output.

`comp_expr_to_pnome.c` Here is the place where expression needs to be evaluated or connected as polynomial. e.g., loop bound expressions, if we are obliged to know the value of the expression, it may be a constant or be denoted by some symbolic variables chosen to be kept. See Section 4.4.3 for more.

`comp_math.c` The math library for complexity is here. e.g., addition integration are defined there.

`comp_unstr.c` When unstructured control flow graphs are encountered, procedures in that file are used.

`comp_util.c` Some useful routines that do not belong to others. e.g., complexity verification and tracing procedure.

`polynome_ri.c` This file gathers some functions interfacing polynomial library and the PIPS internal representation.

`comp_matrice.c` matrice inversion procedures are there. They are designed for floating point operations. The algorithm used is a LU decomposition.

4.4.2 Abstract Syntax Tree Scan

The evaluation program starts with a module name, a main, a subroutine or a function. The evaluation of the complexity is performed by walking up the abstract syntax tree (henceforth abbreviated AST) and combining the call costs using the rules defined in Appendix A.

4.4.3 Expression to Value

In `comp_expr_to_pnome.c`, we need to transform some expressions to values (symbolic or constant), for example, the loop bounds. Statements are either call to functions or statement built with one constructor, sequence, if or loop. Unstructured statements are dealt with in Section 2.4. The expression has almost the same structure as AST, and goes over the `expression` until the atomic variable. The trickiest procedure is `evaluate_var_to_complexity`, which decides when and how a variable can be used via complexity polynomial, or if it fails to be evaluated in the evaluation procedure, an `UNKNOWN` variable will be introduced.

For the conversion, the C3 library is used. `simplify_sc_to_complexity(ps,var)` makes the full use of `Psysteme` ps to simplify the variable `var`. For example, `M1 - M2 = 1` where `M1` is formal parameter and `M2` is an induction variable, the procedure returns `M2 = M1 - 1` packed in the polynomial of the complexity. The statistics counters of this complexity are all zero because no approximation is made in that case.

4.4.4 Unstructured Control Flow Graph

When unstructured control flow graphs are encountered, procedures in `comp_unstr.c` file are used. The algorithm introduced in Section 2.4.2 about unstructured control flow graph is implemented there. The algorithm is based on a matrix inversion implemented in `comp_matrice.c` file. The algorithms for unstructured are introduced in Section 2.4.

4.5 Interfaces to Complexity Estimator

In PIPS interactive environment, there are high-level objects and functions which are user-visible. Objects can be viewed and functions can be activated by one of PIPS interfaces. We have two standard interfaces: **pips**, the tty style interface which is encapsulated by shell procedure, and **wpips**, the X-window based XView interface. Besides these two, we have a development interface which is used to debug the complexity estimation program.

In order to show the different interfaces, we give a small Fortran program. We chose Cholesky decomposition as the demonstration example. Cholesky decomposition is the LL^T decomposition of a symmetric positive definite matrix, and uses an algorithm much like LU decomposition. The example subroutine is *cholesky.f* which is shown in Figure 4.5 in Page 118.

```
      subroutine cholesky(a,n)
      real a(n,n)
      integer i,j,k,n
c
      do k = 1,n
         a(k,k) = sqrt(a(k,k))
         do i = k+1,n
            a(i,k) = a(i,k)/a(k,k)
            do j = k+1,i
               a(i,j) = a(i,j) - a(i,k)*a(j,k)
            end do
         end do
      end do
c
      end
```

Figure 4.5: Cholesky decomposition subroutine

4.5.1 Terminal Interface

There are several commands in PIPS, you can use either **pips** directly with different options, or separate commands, actually they are shell scripts, such as **Init**, **Display**, **Delete**, etc. We have to make it clear that these shell script commands will ultimately call **pips** associated with appropriate options. So we only explain some of these commands which are documented by manual pages in PIPS programming environment.

Init creates a workspace or/and set the current workspace. It creates a new workspace when a readable **file.f** is provided and sets it as current workspace. Otherwise, it sets its argument as current workspace.

Display creates various prettyprints and supports browsing. It can be used on any workspace. It is the simplest command to analyze a program with pips and browse results of any text types. It calls **pips** with appropriate arguments, and runs a pager on the pretty-printed results.

Delete removes a (set of) workspace(s).

After the simple introduction to these commands, we will see how to use these commands. First, we create the workspace using:

```
shell> Init -f cholesky.f cspace
```

which means that we take `cholesky.f` as input Fortran program and create a workspace called `cspace`. Then, we want to see the the complexity analysis result of this subroutine, we use:

```
shell> Display -w cspace -m cholesky -v comp
```

which means that we want to see the complexity analysis result (`comp` is an abbreviation for complexity) of module `cholesky` in workspace `cspace`. The workspace `cspace` does not have to

```
C                                                  3*N^3 + 8*N^2 + N + 1 (SUMMARY)
      SUBROUTINE CHOLESKY(A,N)
      REAL A(N,N)
      INTEGER I,J,K,N
C
C                                                  3*N^3 + 8*N^2 + N + 1 (DO)
      DO K = 1, N                                                        0002
C                                                                      9 (STMT)
         A(K,K) = SQRT(A(K,K))                                           0003
C                                9*K^2 - 18*K.N + 9*N^2 - 25*K + 25*N + 3 (DO)
         DO I = K+1, N                                                   0005
C                                                                     13 (STMT)
            A(I,K) = A(I,K)/A(K,K)                                       0006
C                                                        18*I - 18*K + 3 (DO)
            DO J = K+1, I                                                0008
C                                                                     18 (STMT)
               A(I,J) = A(I,J)-A(I,K)*A(J,K)                             0009
            ENDDO
         ENDDO
      ENDDO
C                                                                      0 (STMT)
C
      END
```

Figure 4.6: Cholesky Complexity Output with Display

be specified if the `Init` has been used recently. So we can remove `-w cspace` in the example. The `-v` option is the verbose option, which causes more information printed out while processing. The output with all-1 cost table is shown in Figure 4.6.

Note that in the Figure 4.6, `SUMMARY` means the summary complexity of the module, `STMT` means the statement complexity and `DO` means the loop complexity, including the complexity of loop body.

When it is no longer needed, the workspace is deleted by:

```
shell> Delete cspace
```

This shell interface is described in [Baron90].

4.5.2 X-Window Interface: wpips

When **wpips** is invoked in X-Window System, the user use the mouse and buttons to perform what he wants. The on-line **Help** is never far away from the user, although it is not very useful for the moment because it is not context sensitive.

At any shell prompt of X-window with PIPS programming environment, type **wpips**, the control window will appear. The control window of **wpips** is shown in Figure 4.7. We first click the *Select*, choose the workspace and module, as shown in upper part of Figure 4.8, and then click to *Props*

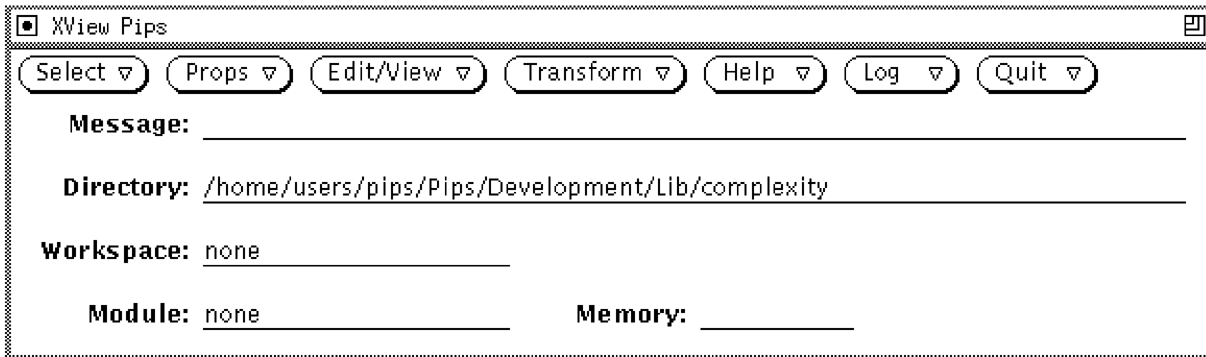


Figure 4.7: The Control Window of Wpips

button to require sequential views of *Statements & Complexities*, and finally, click the *Edit/View* button to display result. **wpips** will automatically open another window to show the result when it is ready, as shown in lower part of Figure 4.8.

We can see from this figure that statement number 0009 has the complexity result 2. That is because we chose the **fp-1** cost table, so only floating point operations are taken into account. In that statement, only two operations are present, $-$ and $*$, so it has result 2. This explanation holds for statement number 0003 and 0006.

More complex analyses require more selection in the *Props* menu, e.g., to use intra or inter procedural preconditions as input for the complexity phase. And of course, more time is needed.

Finally, we can click *Quit* to quit the wpips.

4.5.3 Development Interface

Besides these two standard interfaces, we have another interface unique to the complexity phase. It is reserved exclusively for the complexity evaluation program developers and not for the users.

Using this interface, the developer can obtain additional information unavailable through the standard interfaces. For example, we can get all the trace calls, which will be introduced in Section 4.7.4. At the same time, we can see all the intermediate complexities as well as matrices for the unstructured piece of code.

We add a small explanation for the Figure 4.6. Because we used the *all-1* cost table, so each access and operation all costs one unit of complexity. But there is an exception, we add one more overhead for arrays that are more than two dimensions, because there is a calculation of the address to be accessed. So for $A(I,J)$, one unit cost for A,I and J, and one for overhead. Hence, the complexity for $A(I,J)$ is 4. In statement 0009, there are four accesses of two-dimensional array A and two operations $-$ and $*$. So the complexity result of statement 0009 is $4*4+2=18$.

```
XView Pips
Select ▽  Props ▽  Edit/View ▽  Transform ▽  Help ▽  Log ▽  Quit ▽
Message: PRINTED_FILE made for CHOLESKY.
Directory: /home/users/pips/Pips/Development/Lib/complexity
Workspace: cspace
Module: CHOLESKY        Memory:

Xview Pips Display Facility
C                                     1/3*N^3 + 1/2*N^2 + 1/6*N (SUMMARY)
      SUBROUTINE CHOLESKY(A,N)
      REAL A(N,N)
      INTEGER I,J,K,N
C
C                                     1/3*N^3 + 1/2*N^2 + 1/6*N (DO)
      DO K = 1, N                                                0002
C                                                              1 (STMT)
         A(K,K) = SQRT(A(K,K))                                   0003
C                                 K^2 - 2*K.N + N^2 - 2*K + 2*N (DO)
         DO I = K+1, N                                           0005
C                                                              1 (STMT)
            A(I,K) = A(I,K)/A(K,K)                               0006
C                                                      2*I - 2*K (DO)
            DO J = K+1, I                                        0008
C                                                              2 (STMT)
               A(I,J) = A(I,J)-A(I,K)*A(J,K)                     0009
            ENDDO
         ENDDO
      ENDDO
C                                                              0 (STMT)
C
      END
Sequential View                                       Module: CHOLESKY
```

Figure 4.8: Snapshot of Cholesky Complexity Result

```
SUBROUTINE SAMPLE
REAL A(100)
N = 10
DO I = 1, N
    A(I) = A(I) + 1.0
ENDDO
```

Figure 4.9: An Example **sample.f**

```
C                                                        52 (SUMMARY)
      SUBROUTINE SAMPLE
      REAL A(100)
C                                                         1 (STMT)
      N = 10                                                 0001
C                                                        51 (DO)
      DO I = 1, N                                            0003
C                                                         5 (STMT)
         A(I) = A(I)+1.0                                     0004
      ENDDO
C                                                         0 (STMT)
      END
```

Figure 4.10: Complexity with Default Properties

4.6 Parameters of Estimator

For the time being, only a pure static evaluator has been implemented and only sequential Fortran programs are taken into account. In this section, we present the current implementation and describe advanced features, which are not available through the two standard PIPS user interfaces. These features can be activated by changing special global variables known as "properties" in PIPS.

Default properties are always available, all current properties can be printed out by the command `properties`. But if the user prefers to modify some or all of them, he can overwrite the default values which are stored in the file `properties.rc` at standard PIPS development directory. The user can override the default properties, some or all of them, by creating a file with the same name at the current working directory. We have elaborated this in [ZI92].

Example *sample.f*, shown in Figure 4.9 is used to explain the complexity properties.

The default complexity result of the example in Figure 4.9 is shown in Figure 4.10.

4.6.1 Variables

If the user decides to keep some variables un-evaluated no matter what they are, he can specify the variable names in the file *properties.rc* of current directory. For example, they want to keep `N` and `M` un-evaluated, just modify the variable as `COMPLEXITY_PARAMETERS "N M"`, which guarantees that `N, M` remain as they are.

The complexity result with all-1 cost table of sample program in Figure 4.9 is shown in Figure 4.11.

```
C                                                         5*N + 2 (SUMMARY)
      SUBROUTINE SAMPLE
      REAL A(100)
C                                                               1 (STMT)
      N = 10                                                      0001
C                                                         5*N + 1 (DO)
      DO I = 1, N                                                 0003
C                                                               5 (STMT)
         A(I) = A(I)+1.0                                          0004
      ENDDO
C                                                               0 (STMT)
      END
```

Figure 4.11: Complexity with Unevaluated Variable

A more realistic use of this facility would affect if `N=10` was changed into `READ *, N`.

In addition, PIPS may identify variables which may remain unevaluated. If COMMON variables are read in the module, that means they are assigned somewhere else. These variables are kept in the module's complexity result. So are unassigned formal parameters.

4.6.2 Cost Tables

We provide a way to choose the cost table desired, or to build a new one if needed. For example, if you only want floating-point operations to be accounted for, you can select a non-standard COMPLEXITY_COST_DIR, which is supposed to save a set of cost tables. you put the line `COMPLEXITY_COST_TABLE "all_1"` into the file *properties.rc* of current directory, which means you want to use the indicated cost tables save at indicated directory. The result of sample program in Figure 4.9 with **all-1**, the uniform cost table, is shown in Figure 4.11. You can also use the floating-point only cost table, called **fp-1**, which produces a new result. The result of sample program in Figure 4.9 with **fp-1** cost table is shown in Figure 4.12. As you can see, only floating-point operation is counted.

If user wants to print out that table, he can set `COMPLEXITY_PRINT_COST_TABLE TRUE` to check.

4.6.3 Early Evaluation vs. Late Evaluation

If the user want to see the result more clearly, they will prefer to postpone the variable evaluation as lately as possible. There is `COMPLEXITY_EARLY_EVALUATION` property which can be set to `TRUE`.

Figure 4.13 shows the late variable evaluation. Although the estimator knows that N has the value 10, but the user prefers to see $5*N+1$ instead of the number 51, but finally as the late result, N is replaced by its proper value 10.

The scheme may also reduce the number of unknown variable. For example, in Figure 4.14, n is an unknown variable and is not changed between loop 10 and loop 20. If we use default complexity

```
C                                                         N + 1 (SUMMARY)
      SUBROUTINE SAMPLE
      REAL A(100)
C                                                             0 (STMT)
      N = 10                                                    0001
C                                                         N + 1 (DO)
      DO I = 1, N                                               0003
C                                                             1 (STMT)
         A(I) = A(I)+1.0                                        0004
      ENDDO
C                                                             0 (STMT)
      END
```

Figure 4.12: Complexity with fp-1 Cost Table and Unevaluated Variable

```
C                                                            52 (SUMMARY)
      SUBROUTINE SAMPLE
      REAL A(100)
C                                                             1 (STMT)
      N = 10                                                    0001
C                                                       5*N + 1 (DO)
      DO I = 1, N                                               0003
C                                                             5 (STMT)
         A(I) = A(I)+1.0                                        0004
      ENDDO
C                                                             0 (STMT)
      END
```

Figure 4.13: Complexity with Late Evaluation

```
          n = f(..)
          do 10 i = 1, n
             ...
10        continue
          ...
          do 20 i = 1, n
             ...
20        continue
```

Figure 4.14: Example of Late Evaluation

evaluation, i.e., early evaluation, we will get complexity result with two unknow variables `UU_10` and `UU_20`. See Section 4.7.3 for explanation of these symbols.

But if we defer the evaluation of variables, only one unknown variable is present. That means we effectively reduce the number of unknown variables.

4.6.4 Other Printout and Debugging Parameters

Besides what we have mentioned above, there are some other parameters related to complexity program. They are stored in the same place *properties.rc* and can be overridden by the local ones. We introduce them briefly in the following:

`COMPLEXITY_PRINT_STATISTICS 0` means that the approximate counters option — one of the two components of our complexity — is turned off. (cf. Section 4.2)

`COMPLEXITY_INTERMEDIATES FALSE` means we do not want to see the intermediate complexities while the evaluation program is running.

`COMPLEXITY_TRACE_CALLS FALSE` means we do not want to see all the procedure calls while the evaluation program works. This tracing facility is introduced in Section 4.7.4.

4.7 Features of Estimator

Besides the properties we could turn on or off, there are some intrinsic features of our estimator. We will list and introduce them in the following.

4.7.1 Preconditions

PIPS uses preconditions, information about the input store of statements, to sharpen various analyses. All the preconditions are built at top of the C3 library, which is also used to extract the information from the preconditions. One major source of precondition information is constant: An example is given in Figure 4.15.

In this loop, `n` is set to 10 initially. So the loop upper bound `n` of loop 10 is replaced automatically by 10. However, it is important to make sure that the preconditions are up to date. Consider the second part of the example, `n` is added 20. In this case, we need to use the new value of `n` instead of

```
      subroutine prec2
      n = 10
      do 10 i = 1, n
         u = 1.0
10    continue
      n = n + 20
      do 20 i = 1, n
         t = 1.0
20    continue
      end
```

Figure 4.15: Example of Preconditions

old one. The precondition output is shown in Figure 4.16. The result is produced by **pips** terminal interface command **Display** while set prettyprint option to **prec** for preconditions.

Note that `n` has been changed between the two loops. We have to use different values for each loop. The precondition knows how to get the right value, e.g., in the Figure 4.16, after statement `N = N+20`, `N` is added 20. So `N` is 30 as shown in Figure 4.16.

Preconditions are heavily used by the result simplification, which is the topic of next subsection. As you will see from the example, all kinds of simplifications are based on the program's preconditions.

4.7.2 Simplifying Complexity Results

When a subroutine is called, maybe several formal parameters have been passed. The complexity results should contain the formal parameters in the polynomial but not the *intermediate* variable(s), which are local to the subroutine. Look at the first example of simplification shown in Figure 4.17.

For this example, the summary complexity of the module should contain `n` instead of `k`, for `k` is a private integer variable and is not known by the outside world. We call such variables *local* variables.

Please note that no matter how complex the function is , as long as `k` is a linear function of `n`, the final complexity result always contains `n` as its component. Let us look at a more complex example shown in Figure 4.18.

There are three embedded loops here loop 10, loop 20 and loop 30 respectively. For the innermost loop, its lower bound `jj + 10` is dependent on the index of loop 20, which is determined by the two indices of the outermost loop. Remember that we use a bottom-up method to accumulate the complexity. Confined in this innermost loop, we cannot know the relation between this loop and outer loop. So we must obtain what we need in the innermost loop. Naturally preconditions come into use.

We give the output of complexity result for this example shown in Figure 4.19.

Our method is to put each induction variable into a hash table when encountering a loop, and delete that induction variable when leaving the loop. This allows us to save a copy of the variable, which has not been evaluated by the complexity program. So for the loop 30, the complexity information contains only the outer loops' induction variables `i, j` and formal parameter `m`. For loop 20, the

```
C  P() {}
      SUBROUTINE PREC2
C  P() {}
      N = 10                                                          0001
C  P(N) {N==10}
      DO 10 I = 1, N                                                  0003
C  P(I,N) {1<=I, I<=N, N==10}
         U = 1.0                                                      0004
C  P(I,N) {I<=N, 1<=I, N==10}
10       CONTINUE                                                     0005
C  P(I,N) {N==10}
      N = N+20                                                        0006
C  P(I,N) {N==30}
      DO 20 I = 1, N                                                  0008
C  P(I,N) {1<=I, I<=N, N==30}
         T = 1.0                                                      0009
C  P(I,N) {I<=N, 1<=I, N==30}
20       CONTINUE                                                     0010
C  P(I,N) {N==30}
      END
```

Figure 4.16: Precondition of Example 2

```
      subroutine sub1(a,n)
      real a(1000)
      integer m,n
      k = 3 * n + 2
      do 10 i = 1, k
         a(i) = a(i) + 1.0
10    continue
      return
      end
```

Figure 4.17: Example 1 of Simplification

```
      subroutine sub2(m)
      integer m
      do 10 i = 1, m
         ii = i + 2
         do 20 j = ii, m + 2
            jj = 3*i - 2*j - 2
            do 30 k = jj + 10, 100
               t = t + 1.0
               u = u + 1.0
30          continue
20       continue
10    continue
      return
      end
```

Figure 4.18: Example 2 of Simplification

complexity result is given in terms of `i` and `m`, for loop 10, the result contains merely the formal parameter `m`.

According to our complexity algorithm, shown in (2.3) of Section 2.3, the total complexity for the loop is `2 + 2 + 0 + 6*( (M+10) - (JJ+2) + 1 )` which equals `-18*I + 12*J + 6*M + 83` when substituting `JJ` with the expression `JJ = 3*I-2*J-2` . Here 6 is loop body complexity, 1 is added because `K` is read each time in the loop, and 4 is the range complexity, that is, 2 for the expression `JJ+2` (variable `JJ` and plus sign, each has one complexity unit.) and the same thing for the expression `M+10`.

4.7.3 Locating Unknown Loop Bound

Our estimator provides another facility to locate the unknown loop bound with respect to its statement number, preceded by `UL_` for unknown lower bound, `UI_` for unknown increment and `UU_` for unknown upper bound. For example, the complexity result of program sample of Figure 4.9 is shown in Figure 4.20.

If the number is not present, it won't be there in the final result.

```
C                                            4*M^3 + 57*M^2 + 60*M + 2 (SUMMARY)
      SUBROUTINE SUB2(M)
      INTEGER M
C                                            4*M^3 + 57*M^2 + 60*M + 2 (DO)
      DO 10 I = 1, M                                                   0002
C                                                                    3 (STMT)
         II = I+2                                                      0003
C                        12*I^2 - 24*I.M + 12*M^2 - 126*I + 126*M + 118 (DO)
         DO 20 J = II, M+2                                             0005
C                                                                    7 (STMT)
            JJ = 3*I-2*J-4                                             0006
C                                              -18*I + 12*J + 6*M + 83 (DO)
            DO 30 K = JJ+2, M+10                                       0008
C                                                                    3 (STMT)
               T = T+1.0                                               0009
C                                                                    3 (STMT)
               U = U+1.0                                               0010
C                                                                    0 (STMT)
30             CONTINUE                                                0011
C                                                                    0 (STMT)
20          CONTINUE                                                   0012
C                                                                    0 (STMT)
10       CONTINUE                                                      0013
C                                                                    0 (STMT)
      END
```

Figure 4.19: Example 2 of Simplification

```
C                                                        5*UU_10 + 1 (SUMMARY)
      SUBROUTINE SAMPLE
      REAL A(100)
C                                                        5*UU_10 + 1 (DO)
      DO I = 1, N                                                      0002
C                                                                    5 (STMT)
         A(I) = A(I)+1.0                                               0003
                                                                     0 (STMT)
10       CONTINUE                                                      0004
C                                                                    0 (STMT)
      END
```

Figure 4.20: Complexity with Unknown Loop Bound

```
SUBROUTINE SAMPLE
REAL A(100)
N = 10
DO I = 1, N
    A(I) = 1.0
ENDDO
```

Figure 4.21: An Example **trace.f**

4.7.4 Tracing of Complexity Estimation

We still use an even simpler program shown in Figure 4.21, we shown the program tracing output in Figure 4.22.

The tracing output begins at the left part of the figure and continues in the right part. Vertical bars in the figure help locate the indent level. At the very beginning, TRACE bumps into a statement with ordering (0,1), which is interpreted as an instruction as well. And this instruction is considered unstructured according to PIPS intermediate representation shown in Appendix A. This unstructured is interpreted as a statement, which is a block, which is comprised of several statements.

4.7.5 Debug

There is an environment variable called `COMPLEXITY_DEBUG_LEVEL` for debugging, which is set 0 as default. This variable indicates which debug level we prefer. It can vary from 0 to 9, the higher the level, the more information. For example, if it is set to 9, we can have maximum information available, including all matrix results.

4.8 Summary

This chapter contains a detailed presentation of our execution time estimator and its implementation. The PIPS programming environment is briefly introduced. The data structures designed to estimate execution times are shown as well as the PIPS data structures used in our algorithms. We end up the implementation part with a list of C files and some explanations about the key functions. The complexity user interfaces are presented from a functional point of view. They are made of the two standard PIPS user interfaces plus a set of parameters that can be set by advanced users only. Some special features can also be activated.

```
complexities TRACE
  >statement @, ordering (0 1)
    >instruction
    | >unstructured
    |   >statement @, ordering (1 1)
    |     >instruction
    |     | >block
    |     |   >statement @, ordering (1 2)
    |     |     >instruction
    |     |     | >call TOP-LEVEL:=
    |     |     |   >arguments
    |     |     |     >expression
    |     |     |     | >syntax
    |     |     |     |   >reference TRACE:N
    |     |     |     |     >indices
    |     |     |     |     <
    |     |     |     |   <
    |     |     |     | <
    |     |     |     <
    |     |     |     >expression
    |     |     |     | >syntax
    |     |     |     |   >call TOP-LEVEL:10
    |     |     |     |   <
    |     |     |     | <
    |     |     |     <
    |     |     |   <
    |     |     | <
    |     |     <
    |     |   <
    |     |   >statement @, ordering (1 3)
    |     |     >instruction
    |     |     | >loop TRACE:I label TOP-LEVEL:@
    |     |     |   >range
    |     |     |     >expression
    |     |     |     | >syntax
    |     |     |     |   >call TOP-LEVEL:1
    |     |     |     |   <
    |     |     |     | <
    |     |     |     <
    |     |     |     >expression
    |     |     |     | >syntax
    |     |     |     |   >reference TRACE:N
    |     |     |     |     >indices
    |     |     |     |     <
    |     |     |     |   <
    |     |     |     | <
    |     |     |     <
    |     |     |     >expression
    |     |     |     | >syntax
    |     |     |     |   >call TOP-LEVEL:1
    |     |     |     |   <
    |     |     |     | <
    |     |     |     <
    |     |     |   <
    |     |     |   >statement @, ordering (1 4)
    |     |     |     >instruction
    |     |     |     | >call TOP-LEVEL:=
    |     |     |     |   >arguments
    |     |     |     |     >expression
    |     |     |     |     | >syntax
    |     |     |     |     |   >reference TRACE:A
    |     |     |     |     |     >indices
    |     |     |     |     |     | >expression
    |     |     |     |     |     |   >syntax
    |     |     |     |     |     |     >reference TRACE:I
    |     |     |     |     |     |     | >indices
    |     |     |     |     |     |     | <
    |     |     |     |     |     |     <
    |     |     |     |     |     |   <
    |     |     |     |     |     | <
    |     |     |     |     |     <
    |     |     |     |     |   <
    |     |     |     |     | <
    |     |     |     |     <
    |     |     |     |     >expression
    |     |     |     |     | >syntax
    |     |     |     |     |   >call TOP-LEVEL:10.
    |     |     |     |     |   <
    |     |     |     |     | <
    |     |     |     |     <
    |     |     |     |   <
    |     |     |     | <
    |     |     |     <
    |     |     |   <
    |     |     |   >expression -> pnome
    |     |     |     >normalized -> pnome
    |     |     |     | >pvecteur -> pnome max -1
    |     |     |     | <
    |     |     |     <
    |     |     |   <
    |     |     |   >expression -> pnome
    |     |     |     >normalized -> pnome
    |     |     |     | >pvecteur -> pnome max 1
    |     |     |     |   >var TRACE:N->pnome, max 1
    |     |     |     |   <
    |     |     |     | <
    |     |     |     <
    |     |     |     >expression -> pnome
    |     |     |     | >normalized -> pnome
    |     |     |     |   >pvecteur->pnome max -1
    |     |     |     |   <
    |     |     |     | <
    |     |     |     <
    |     |     |   <
    |     |     | <
    |     |     <
    |     |     >statement @00000, ordering (1 5)
    |     |     | >instruction
    |     |     |   >call TOP-LEVEL:RETURN
    |     |     |     >arguments
    |     |     |     <
    |     |     |   <
    |     |     | <
    |     |     <
    |     |   <
    |     | <
    |     <
    |   <
    | <
    <
  <
  <summary complexity TRACE
<
```

Figure 4.22: Tracing of Sample Subroutine

Chapter 5

Experiments on Sequential Machines

In this chapter, we show that our estimator can generate exact results by comparing its output for a real application **tmines.f** with the results obtained manually by Nahid Emad [Emad91]. Then, in order to validate our result, we will time a subroutine **EFLUX** from **FLO52** program of the Perfect Club Benchmark Suite [Cybenko91], and compare its effective execution time with its predicted execution time according to the complexity result obtained by our estimator. The time difference is pretty small, less than 5% despite the large ranges of values of two variables $I2$ and $J2$.

5.1 Comparisons with Manual Calculation

We use the program **tmines.f** from ONERA[1], which was written by Marc Bredif. Because Nahid Emad [Emad91] studied the floating-point complexity of all 11 modules of **tmines.f** during her internship at CRI, so we can take advantage of her results in order to compare. We only need to select the cost table **fp-1** to restrict the complexity analysis to floating point. The program resolves the potential equations and calculates the irrotational isentropic flow in a convergent tube with varying rectangular section. The call graph of TMINES is shown below:

```
TMINES
    MAILLA
    POLTRI
    CALMAT
    RESUL
    ROMAT
        PREPCG
    CALCG
        DES
        REP
        PROD
    RESUL
```

The convergence loop is not shown because it is seen as an unstructured piece of code by PIPS. In Emad's paper, it is observed that variables ni, nj and nk are parameters used in the program

[1] A French Aerospace Research Center

as array dimensions and as control variable for many loops. In order to reduce her complexity formulae, Emad introduced 5 intermediate variables:

$$\begin{aligned}
\alpha &= ni * nj * nk \\
\beta &= nj * nk \\
\zeta &= ni * nk \\
\tilde{\alpha} &= (ni-1)*(nj-1)*(nk-1) \\
\tilde{\gamma} &= (ni-1)*(nj-1)
\end{aligned}$$

Actually, the COMMON parameters used in **tmines.f** are IM, JM and KM. The initialization code insures that $ni = IM$, $nj = JM$ and $nk = KM$. We use $C_m(S)$ to denote the manual complexity obtained by Nahid Emad for statement S and $C_e(S)$ to denote the estimated complexity of our estimator for statement S, and if both happen to be the same, we use simple $C(S)$ to denote both of them.

In this section, we compare the results for each module. Note that *Loop Label* in all tables means that the corresponding loop, while *Module* means the final complexity of the module, it is a kind of combination of its containing loops' complexity, not necessarily the sum of them.

5.1.1 Mailla Subroutine

The results are:

MAILLA Subroutine		
Loop Label	*Manual Complexity*	*Estimator Complexity*
1	$3/2 * ni - 3$	$3/2 * IM - 3$
2	ni	IM
3	$12 * ni$	$12 * IM$
20	$4 * \alpha$	$4 * IM.JM.KM$
200	$(ni-1)(3 * nj * nk + 419)$	$3 * IM.JM.KM - 3 * JM.KM + 419 * IM - 419$
12	$12 * (ni - 1)$	$12 * IM - 12$
Module	$7 * \alpha - 3 * \beta + 543 * ni - 517$	$7 * IM.JM.KM - 3 * JM.KM + 434 * IM - 417$

Despite several test statements in Loop 3, MAILLLA is a well-structured subroutine. We suppose statically the probabilities are all fifty-fifty for those tests as usual. As a matter of fact, the probabilities of these test statements cannot be determined statically by a human, so the best bet is fifty-fifty. This leads to the same result for Loop 3. There are no discrepancies for the loops, but for the whole procedure are different. We looked for the reason why there is a difference in the estimator but could not find any. We manually checked the result, confirming that the result of our estimator is correct. We think that the difference probably came from Emad's miscalculation.

5.1.2 Poltri Subroutine

The results are:

POLTRI Subroutine		
LoopLabel	*Manual Complexity*	*Estimator Complexity*
111	8	8
1	351	351
2	6912	6912
Module	7277	7277

The number of floating point operation is numerically known, because the loop ranges are constants. The two results are equal.

5.1.3 Calmat Subroutine

This subroutine builds the matrix used in the calculation. It contains 170 lines of code. The results are:

CALMAT Subroutine		
LoopLabel	*Manual Complexity*	*Estimator Complexity*
2	$144 * (ni - 1)$	$144 * IM - 144$
205	$ni - 1$	$IM - 1$
4	$3 * (ni - 1)$	$3 * IM - 3$
5	$12 * (ni - 1)$	$12 * IM - 12$
3	$48 * (ni - 1)$	$48 * IM - 48$
207	$32 * (ni - 1)$	$32 * IM - 32$
8	$468 * (ni - 1)$	$468 * IM - 468$
30	$5536 * (ni - 1)$	$5536 * IM - 5536$
40	$126 * (ni - 1)$	$126 * IM - 126$
45	$2 * (ni - 1)$	$2 * IM - 2$
46	$8 * (ni - 1)$	$8 * IM - 8$
10	$5912 * \tilde{\gamma}$	$5672 * IM.JM - 5672 * IM - 5672 * JM + 5672$
1	$5912 * \tilde{\alpha}$	$5672 * IM.JM.KM - 5672 * IM.JM - 5672 * IM.KM$ $-5672 * JM.KM + 5672 * IM + 5672 * JM + 5672 * KM - 5672$
Module	$5912 * \tilde{\alpha}$ $+\beta$	$5672 * IM.JM.KM - 5672 * IM.JM - 5672 * IM.KM$ $-5672 * JM.KM + 5672 * IM + 5672 * JM + 5672 * KM - 5672$

If we simplify our estimated result, it should be $5672 * \tilde{\alpha} + \beta$. After investigation, we finally figure out that the difference is due to miscalculation of original author. Because $C(L10)$ should be

$$C(L10) = (nj - 1) * [C(L30) + C(L40) + C(L45) + C(L46)] = 5672 * \tilde{\gamma}$$

and not all of those above. The author probably took extra loop costs into accounts, very likely that of 2, 205, 4, 5, 3 and 205. As we can see:

$$C_m(L10) = (nj - 1) * [C(L30) + C(L40) + C(L45) + C(L46) + C(L2)$$

$$
\begin{aligned}
& +C(L205)+C(L4)+C(L5)+C(L3)] \\
= \;& 5912 * \tilde{\gamma}
\end{aligned}
$$

This modified formula matched the manual result. This is the kind of mistakes that a human can easily make, because humans are not reliable, consummate. On the other hand, the machines are not prone to commit that kind of mistakes.

5.1.4 Resul Subroutine

This subroutine prints out the result. The complexity results are:

RESUL Subroutine		
LoopLabel	*Manual Complexity*	*Estimator Complexity*
10	36	36
100	$63 * ni - 63$	$63 * IM - 63$
Module	$63 * ni - 54$	$63 * IM - 54$

The two results are identical.

5.1.5 Prepcg Subroutine

The complexity results are:

PREPCG Subroutine		
LoopLabel	*Manual Complexity*	*Estimator Complexity*
2	$33 * nj - 18$	$24 * JM$
3	$40 * nj$	$40 * JM$
5	$32 * (nj - 1)$	$32 * JM - 32$
6	$24 * (nj - 1)$	$24 * JM - 24$
20	$ni * (129 * nj - 74)$	$120 * IM.JM - 56 * IM$
1	$129 * \alpha - 74 * \zeta$	$120 * IM.JM.KM - 56 * IM.KM$
Module	$129 * \alpha - 74 * \zeta$	$120 * IM.JM.KM - 56 * IM.KM$

As we can see, there is a difference in Loop 2. After investigation, we find out that there is a test statement with obvious probability, the true-ratio is one over loop count, as the following:

```
      DO 2 J = 1, JM
      S1
      IF (J.EQ.JM) THEN
      ELSE
         S2
      ENDIF
2     CONTINUE
```

Where $C(S1) = 15$ and $C(S2) = 18$. We can see clearly that false-ratio is $(JM-1)/JM$. As we have said earlier, our estimator supposes half-half, so we have:

$$C_e(L2) = JM * (C(S1) + 1/2 * C(S2)) = 24 * JM$$

while manual result is:

$$C_m(L2) = JM * (C(S1) + (JM-1)/JM * C(S2)) = 33 * JM - 18$$

Therefore the probability makes that difference. In this subroutine, the manual result is correct because it takes right probability. It is no surprise to see the final result deviates. If we instruct the estimator to take that probability at that test statement, we can get right result for Loop 20 and final result. We have mentioned earlier in Section 1.4 about the dynamic estimation of probability.

5.1.6 Romat Subroutine

This subroutine computes, among the others, an incomplete decomposition of Cholesky, to produce a prerequisite for the conjugate gradient method. The complexity results are:

ROMAT Subroutine		
LoopLabel	*Manual Complexity*	*Estimator Complexity*
11	$126*(ni-1)$	$126*IM-126$
23	$2*(ni-1)$	$2*IM-2$
33	$16*(ni-1)$	$16*IM-16$
24	$4*(ni-1)$	$4*IM-4$
28	$7*(ni-1)$	$7*IM-7$
25	$2*(ni-1)$	$2*IM-2$
26	$3*(ni-1)$	$3*IM-3$
29	$108*(ni-1)$	$108*IM-108$
27	$11*(ni-1)$	$11*IM-11$
91	$14*(ni-1)$	$14*IM-14$
92	$14*(ni-1)$	$14*IM-14$
93	$16*(ni-1)$	$16*IM-16$
99	$28*(ni-1)$	$28*IM-28$
50	$16*(ni-1)$	$16*IM-16$
2	$371*\tilde{\gamma}$	$371*IM.JM-371*IM-371*JM+371$
1	$371*\tilde{\alpha}$	$371*IM.JM.KM-371*IM.JM-371*IM.KM$ $-371*JM.KM+371*IM+371*JM+371*KM-371$
Module	$371*\tilde{\alpha}+21$ $+129*\alpha-74*\zeta$	$491*IM.JM.KM-371*IM.JM-427*IM.KM$ $-371*JM.KM+371*IM+371*JM+371*KM-350$

In fact, the complexity of ROMAT is easy to calculate interprocedurally as:

$$C_e(Romat) = 371*\tilde{\alpha} + 21 + C_e(Prepcg)$$

$$C_m(Romat) = 371*\tilde{\alpha} + 21 + C_m(Prepcg)$$

Because ROMAT calls PREPCG and there is a small difference in PREPCG, so it is normal that there is a small difference here in ROMAT. But for this difference, the results are the same.

5.1.7 Des Subroutine

The complexity results are:

DES Subroutine		
LoopLabel	*Manual Complexity*	*Estimator Complexity*
10	$3*(nj-1)$	$3*JM-3$
11	$8*nj$	$8*JM$
12	$8*(nj-1)$	$8*JM-8$
13	$8*(nj-1)$	$8*JM-8$
1	$ni*(27*nj-18)$	$27*IM.JM-18*IM$
100	$27*\alpha-18*\zeta$	$27*IM.JM.KM-18*IM.KM$
Module	$27*\alpha-18*\zeta$	$27*IM.JM.KM-18*IM.KM$

The two results are identical.

5.1.8 Rep Subroutine

The complexity results are:

REP Subroutine		
LoopLabel	*Manual Complexity*	*Estimator Complexity*
21	$8*nj$	$8*JM$
22	$8*(nj-1)$	$8*JM-8$
23	$8*(nj-1)$	$8*JM-8$
30	$3*(nj-1)$	$3*JM-3$
20	$ni*(27*nj-18)$	$27*IM.JM-18*IM$
200	$27*\alpha-18*\zeta$	$27*IM.JM.KM-18*IM.KM$
Module	$27*\alpha-18*\zeta$	$27*IM.JM.KM-18*IM.KM$

The two results are identical.

5.1.9 Prod Subroutine

The complexity results are:

PROD Subroutine		
LoopLabel	*Manual Complexity*	*Estimator Complexity*
10	$18*ni$	$18*IM$
11	$24*(ni-1)$	$24*IM-24$
12	$12*(ni-1)$	$12*IM-12$
2	$nj*(54*ni-36)$	$54*IM.JM-36*JM$
1	$54*\alpha-36*\beta$	$54*IM.JM.KM-36*JM.KM$
Module	$54*\alpha-36*\beta$	$54*IM.JM.KM-36*JM.KM$

The two results are identical.

5.1.10 Calcg Subroutine

This subroutine applies conjugate gradient method with the previously decomposed matrix LDL^t. The decomposition is performed with Cholesky's algorithm in *romat*. The complexity results are:

CALCG Subroutine		
LoopLabel	*Manual Complexity*	*Estimator Complexity*
21	$2 * \alpha$	$2 * IM.JM.KM$
30	$2 * \alpha$	$2 * IM.JM.KM$
50	$2 * \alpha$	$2 * IM.JM.KM$
41	$2 * \alpha$	$2 * IM.JM.KM$
51	$2 * \alpha$	$2 * IM.JM.KM$
Module	$nmax * (118 * \alpha$ $-36 * \beta - 36 * \zeta + 4)$	$63.38 * IM.JM.KM - 22.15 * IM.KM$ $-16.62 * JM.KM + 3.38$

Again, this estimation is interprocedural:

$$\begin{aligned} C_m(Calcg) &= nmax * [10 * \alpha + C(Des) + C(Rep) + C(Prod) + 4] \\ &= nmax * [10 * \alpha + (27 * \alpha - 18 * \zeta) + (27 * \alpha - 18 * \zeta) + (54 * \alpha - 36 * \beta) + 4] \\ &= nmax * (118 * \alpha - 36 * \beta - 36 * \zeta + 4) \end{aligned}$$

There is a big difference here between the two results. Let us first have a look at the original program in Appendix C.1. Original *calcg* has a loop labeled 1 (Loop 1) with 1 as lower bound and *nmax* as upper bound. It contains all the five loops cited above and three subroutines. But in this module, there are some error-checking statements, something like $IF()$ *return*, which cause great problem for complexity estimation.

Figure 5.1 shows the simplified control flow graph. According to the control analyzer of PIPS, Loop 1 is converted to IF statement with GOTO statement at the end of IF scope to go back to the beginning of IF statement, i.e., from $S8$ to $S2$ in the Figure. And we have seen equally that inside the Loop 1, there are two paths the lead to irregular exits, i.e., $S2 \rightarrow S12 \rightarrow S7$ and $S3 \rightarrow S9 \rightarrow S11 \rightarrow S7$.

Because these test statements are simply error-checking, so they are very unlikely to happen, we should give them zero probability without hesitation as suggested in [Wang93]. Due to several occurrences of these error-checking tests and the half-half policy used in the implementation, we obtained this wrong result. In this case, manual result is correct simply because a human is more intelligent than machine. But a look at the complexity counters would warn the user.

Now let us assume that our estimator has extra knowledge that those error-checking tests are small probablily events. We categorily give them zero probability and remove them from the control flow graph. As shown in Figure 5.1, we remove the part of graph that is surrounded by the discontinuous lines. After that, we can find that the graph resembles the example we have

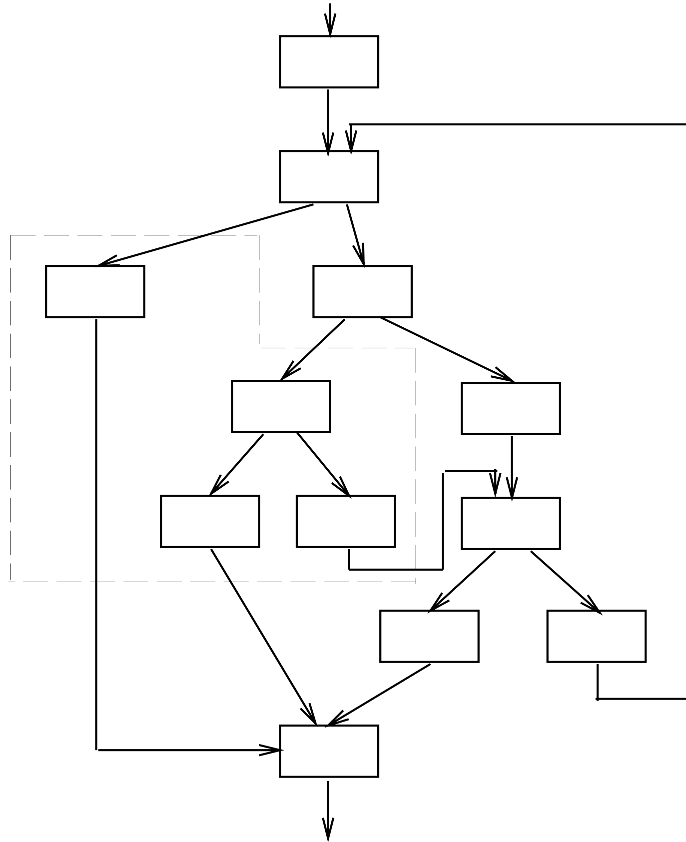


Figure 5.1: Calcg Simplified Control Flow Graph

presented in Section 2.4.3. So we can reuse the Equation 2.15. We have no problem to get the right result:

$$\begin{aligned} C(Calcg) &= C(S1) + nmax * [C(S2) + C(S3) + C(S4) + C(S5) + C(S8)] + C(S6) + C(S7) \\ &= nmax * [10 * \alpha + C(Des) + C(Rep) + C(Prod) + 4] \\ &= nmax * [10 * \alpha + (27 * \alpha - 18 * \zeta) + (27 * \alpha - 18 * \zeta) + (54 * \alpha - 36 * \beta) + 4] \\ &= nmax * (118 * \alpha - 36 * \beta - 36 * \zeta + 4) \\ &= C_m(Calcg) \end{aligned}$$

We give the original subroutine and its corresponding complexity result in Appendix C. The interested user can find the corresponding complexity of each statement in Appendix C.4.

5.1.11 Tmines Program

Tmines module is main program of **tmines.f** program. Note that both $nmaxn1$ and $nmax$ are set to 50 at the beginning of the program. We can easily obtain as worse case manual complexity:

$$\begin{aligned} C_m(Tmines) &= C_m(Mailla) + C(Poltri) + C_m(Calmat) + C(Resul) \\ &\quad + nmaxn1 * [C_m(Romat) + C_m(Calcg) + C(Resul)] \\ &= 7 * \alpha + 5912 * \tilde{\alpha} - 2 * \beta + 606 * ni + 6706 \\ &\quad + 50 * [129 * \alpha + 371 * \tilde{\alpha} - 74 * \zeta + 50 * (118 * \alpha - 36 * \beta - 36 * \zeta + 4) + 63 * ni - 33] \\ &= 7 * \alpha + 5912 * \tilde{\alpha} - 2 * \beta + 606 * ni + 6706 \\ &\quad + 50 * [6029 * \alpha + 371 * \tilde{\alpha} - 1800 * \beta - 1874 * \zeta + 63 * ni + 167] \\ &= 301307 * \alpha + 24462 * \tilde{\alpha} - 89998 * \beta - 36 * \zeta + 3756 * ni + 15056 \end{aligned}$$

And for estimated complexity:

$$\begin{aligned} C_e(Tmines) &= C_e(Mailla) + C(Poltri) + C_e(Calmat) + C(Resul) \\ &\quad + nmaxn1 * [C_e(Romat) + C_e(Calcg) + C(Resul)] \\ &= 5679 * IM.JM.KM - 5672 * IM.JM - 5672 * IM.KM - 5675 * JM.KM \\ &\quad + 6169 * IM + 5672 * JM + 5672 * KM - 6203 \\ &\quad + 50 * (554.38 * IM.JM.KM - 371 * IM.JM - 449.15 * IM.KM \\ &\quad - 387.62 * JM.KM + 434 * IM + 371 * JM + 371 * KM - 400.62) \\ &= 33398 * IM.JM.KM - 24222 * IM.JM - 28129.55672 * IM.KM \\ &\quad - 25056 * JM.KM + 27869 * IM + 24222 * JM + 24222 * KM - 26234 \end{aligned}$$

TMINES Main Program		
Module	*Manual Complexity*	*Estimator Complexity*
Mailla	$7*\alpha-3*\beta+543*ni-517$	$7*IM.JM.KM-3*JM.KM+434*IM-417$
Poltri	7277	7277
Calmat	$5912*\tilde{\alpha}$ $+\beta$	$5672*IM.JM.KM-5672*IM.JM-5672*IM.KM$ $-5672*JM.KM+5672*IM+5672*JM+5672*KM-5672$
Resul	$63*ni-54$	$63*IM-54$
Romat	$371*\tilde{\alpha}+21$ $+129*\alpha-74*\zeta$	$491*IM.JM.KM-371*IM.JM-427*IM.KM$ $-371*JM.KM+371*IM+371*JM+371*KM-350$
Calcg	$nmax*(118*\alpha$ $-36*\beta-36*\zeta)$	$63.38*IM.JM.KM-22.15*IM.KM$ $-16.62*JM.KM+3.38$
Tmines	$301307*\alpha+24462*\tilde{\alpha}$ $-89998*\beta-36*\zeta$ $+3756*ni+15056$	$33398*IM.JM.KM-24222*IM.JM$ $-28129.55672*IM.KM-25056*JM.KM+27869*IM$ $+24222*JM+24222*KM-26234$

As we have noted the difference is too big to be used, and we think that it is normal. The difference in CALCG is amplified by $nmaxn1$. i.e. 50 in TMINES. Therefore, so much big difference is the correct inference.

5.1.12 Summary and Discussion

In this section, we have compared all results of 11 modules of **tmines.f** program.

Among these, the results for POLTRI, RESUL, DES, REP and PROD are completely identical; the results for MAILLA, CALMAT, PREPCG and ROMAT are almost the same; and results for CALCG is a failure, and so is TMINES.

Although there is so big difference in final module TMINES, we only need to do one thing to minimize the discrepancy: instruct the estimator the right probability in CALCG, then everything will be alright.

5.2 Results of a Perfect Club Program FLO52

The Perfect Benchmarks are a collection of benchmark applications which were contributed by various large system vendors and which have been used to characterize supercomputer performance.

In this section, we present experimental results of our complexity program on a real Fortran program. We chose a medium-sized one — *TFS.f* from the PERFECT Club (cf. [Cybenko91]). It has almost 2000 lines and its application area is computational fluid dynamics. In order to focus on the most important functions, we have gotten rid of the irrelevant routines, such as plotting a graph and timing. This leaves 27 routines, and we succeeded in evaluating 23 of them. The analysis were performed using both the **all-1** and **fp-1** models. The results are shown in Table 5.1 and Table 5.2 respectively.

IL and JL are COMMON variables, so we can keep them in the complexity results. $I2$, $J2$, $II2$, $JJ2$, $LPRNT$ and $NRES$ are formal parameters passed by the corresponding callers. In the

$Module$	$Complexity$
$ADDX$	$159*I2.J2+32*II2.JJ2+95*IL.U_RANGE$ $+158*IL+11*J2+2*JJ2-4*U_RANGE+22$
$BCFAR$	$240*IL-101$
$BCWALL$	$253*IL-113$
$CIRCLE$	$21*IL.JL+IL+2$
$COLLC$	$60*I2.J2+94*IL.JL+62*IL+4*J2+56*JL+103$
$COORD$	$13*IL+16*JL+7$
$CPLOT$	$36*IL+308$
$DFLUX$	$995*IL.JL-775*IL-553*JL+387$
$DFLUXC$	$459*IL.JL-326*IL-305*JL+197$
$EFLUX$	$490*IL.JL-445*IL-315*JL+281$
$FORCF$	$87*IL-65$
$GRAPH$	$31*IL+32$
$GRID$	$32*IL.JL+14*IL+23*JL+14$
$INIT$	$33*I2.J2+J2+1$
$INTPL$	458.93
$MESH$	230
$METRIC$	$52*IL.JL+14*I2-52*IL-37*JL+39$
$PRNTFF$	$93*I2.J2.LPRNT\hat{}(-1)+186*I2-91*I2.LPRNT\hat{}(-1)$ $-93*J2.LPRNT\hat{}(-1)+91*LPRNT\hat{}(-1)$
$PRNTXY$	$17*IL.JL.LPRNT\hat{}(-1)+2*IL.LPRNT\hat{}(-1)+4$
$PSMOO$	$400*IL.JL-416*IL-416*JL+478$
$RPLOT$	$37*NRES+51$
$STEP$	$244*IL.JL+28*I2-244*IL-213*JL+222$
$XPAND$	$5*IL.JL+5*IL+7*JL+9$

Table 5.1: Complexity Results Using **all-1**

$Module$	$Complexity$
$ADDX$	$28 * I2.J2 + 2 * II2.JJ2 + 12 * IL.U_RANGE + 43 * IL - 12 * U_RANGE - 3/2$
$BCFAR$	$36 * IL + 25 * IL - 36$
$BCWALL$	$55 * IL - 38$
$CIRCLE$	$5 * IL.JL$
$COLLC$	$4 * I2.J2 + 10 * IL.JL + 6 * IL$
$COORD$	$4 * IL + 5 * JL + 4$
$CPLOT$	$8 * IL$
$DFLUX$	$106 * IL.JL - 96 * IL - 83 * JL + 78$
$DFLUXC$	$48 * IL.JL - 34 * IL - 34 * JL + 23$
$EFLUX$	$70 * IL.JL - 64 * IL - 41 * JL + 35$
$FORCF$	$25 * IL - 19$
$GRAPH$	$8 * IL + 8$
$GRID$	$8 * IL.JL + 2 * IL + 4 * JL$
$INIT$	$4 * I2.J2$
$INTPL$	88
$MESH$	80
$METRIC$	$8 * IL.JL - 8 * IL - 8 * JL + 8$
$PRNTFF$	$17 * I2.J2.LPRNT\hat{}(-1) + 34 * I2 - 17 * I2.LPRNT\hat{}(-1)$ $-17 * J2.LPRNT\hat{}(-1) + 17 * LPRNT\hat{}(-1)$
$PRNTXY$	0
$PSMOO$	$48 * IL.JL - 56 * IL - 56 * JL + 78$
$RPLOT$	$13 * NRES + 8$
$STEP$	$48 * IL.JL - 48 * IL - 48 * JL + 48$
$XPAND$	0

Table 5.2: Complexity Results Using **fp-1**

routine ADDX, U_RANGE which is short for UNKNOWN_RANGE appears. This is due to the variable J1, which serves as upper loop bound. It depends on the input, so we cannot know it at compile-time. But it could be kept in the formulae because it is constant over the program. The user should declare it constant with properties file `properties.rc`.

5.3 EFLUX of FLO52

We chose **eflux.f** from FLO52 to verify our results because most computations of FLO52 happen there. FLO52 is one of 13 Benchmark programs from Perfect Club. We give the original **eflux** program in Figure 5.2.

We can expect that the complexity result will contain both IL and JL because they are both global variables. In order to check our estimated complexity, we need to obtain reasonable ranges of the both variables IL and JL.

1. From the original program FLO52 of Perfect Benchmark Club, we could observe from line 81, 155, 202, 316, 338 and 441 where IL and JL are modified, that there are consecutive lines as following:

```
IL = NX + 1
JL = NY + 1
I2 = NX + 2
J2 = NY + 2
```

   and nowhere else. So we conclude that $IL = I2 - 1$ and $JL = J2 - 1$.

2. In subroutine eflux, declarations are:

```
COMMON/LIM/ IL,JL
COMMON/ADD/ DW(194,34,4)
COMMON/FLX/ FS(193,34,4)
DIMENSION   W(I2,J2,4),P(I2,J2),X(I2,J2,2)
```

   and in the loop labeled 30, there is

```
DO 30 J = 2, JL
DO 30 J = 2, IL
.....
FS(I,J,4) = QSP*(W(I,J+1,4)+P(I,J+1))+ ...
```

   so we can see that the first index of FS is 193 and cannot be surpassed by IL, so $IL \leq 193$; for $P(I, J+1)$ reason $JL \leq 33$.

3. From above item, $IL \geq 2$ and $JL \geq 2$.

Now let us have a look at the output of the complexity evaluation program obtained with **Sun-SPARC** cost table:

```
      SUBROUTINE EFLUX (I2,J2,W,P,X)
C     EULER FLUXES
      COMMON/LIM/ IL,JL
      COMMON/ADD/ DW(194,34,4)
      COMMON/FLX/ FS(193,34,4)
      DIMENSION   W(I2,J2,4),P(I2,J2),X(I2,J2,2)
      DO 10 J=2,JL
      DO 10 I=1,IL
      XY        = X(I,J,1)  -X(I,J-1,1)
      YY        = X(I,J,2)  -X(I,J-1,2)
      PA        = P(I+1,J)  +P(I,J)
      QSP       = (YY*W(I+1,J,2)  -XY*W(I+1,J,3))/W(I+1,J,1)
      QSM       = (YY*W(I,J,2)  -XY*W(I,J,3))/W(I,J,1)
      FS(I,J,1) = QSP*W(I+1,J,1)  +QSM*W(I,J,1)
      FS(I,J,2) = QSP*W(I+1,J,2)  +QSM*W(I,J,2)  +YY*PA
      FS(I,J,3) = QSP*W(I+1,J,3)  +QSM*W(I,J,3)  -XY*PA
      FS(I,J,4) = QSP*(W(I+1,J,4)  +P(I+1,J))  +QSM*(W(I,J,4)  +P(I,J))
   10 CONTINUE
      DO 20 N=1,4
      DO 20 J=2,JL
      DO 20 I=2,IL
      DW(I,J,N) = FS(I,J,N)  -FS(I-1,J,N)
   20 CONTINUE
      DO 25 I=2,IL
      XX        = X(I,1,1)  -X(I-1,1,1)
      YX        = X(I,1,2)  -X(I-1,1,2)
      PA        = P(I,2)    +P(I,1)
      FS(I,1,1) = 0.
      FS(I,1,2) = -YX*PA
      FS(I,1,3) =  XX*PA
      FS(I,1,4) = 0.
   25 CONTINUE
      DO 30 J=2,JL
      DO 30 I=2,IL
      XX        = X(I,J,1)  -X(I-1,J,1)
      YX        = X(I,J,2)  -X(I-1,J,2)
      PA        = P(I,J+1)  +P(I,J)
      QSP       = (XX*W(I,J+1,3)  -YX*W(I,J+1,2))/W(I,J+1,1)
      QSM       = (XX*W(I,J,3)  -YX*W(I,J,2))/W(I,J,1)
      FS(I,J,1) = QSP*W(I,J+1,1)  +QSM*W(I,J,1)
      FS(I,J,2) = QSP*W(I,J+1,2)  +QSM*W(I,J,2)  -YX*PA
      FS(I,J,3) = QSP*W(I,J+1,3)  +QSM*W(I,J,3)  +XX*PA
      FS(I,J,4) = QSP*(W(I,J+1,4)  +P(I,J+1))  +QSM*(W(I,J,4)  +P(I,J))
   30 CONTINUE
      DO 40 N=1,4
      DO 40 J=2,JL
      DO 40 I=2,IL
      DW(I,J,N) = DW(I,J,N)  +FS(I,J,N)  -FS(I,J-1,N)
   40 CONTINUE
      RETURN
      END
```

Figure 5.2: Original Eflux Subroutine of FLO52

```
C                              1878*IL.JL - 1711*IL - 999*JL + 1033 (SUMMARY)
C     NAME=EFLUX
      SUBROUTINE EFLUX (I2,J2,W,P,X)
C     EULER FLUXES
      COMMON/LIM/ IL,JL
      COMMON/ADD/ DW(194,34,4)
CSRD  COMMON/FLX/ FS(193,33,4)
      COMMON/FLX/ FS(193,34,4)
      DIMENSION   W(I2,J2,4),P(I2,J2),X(I2,J2,2)
C                                    699*IL.JL - 699*IL + 18*JL - 7 (DO)
      DO 10 J = 2, JL                                              0002
C                                                     699*IL + 11 (DO)
         DO 10 I = 1, IL                                           0004
C                                                          54 (STMT)
            XY = X(I,J,1)-X(I,J-1,1)                               0005
C                                                          54 (STMT)
            YY = X(I,J,2)-X(I,J-1,2)                               0006
C                                                          48 (STMT)
            PA = P(I+1,J)+P(I,J)                                   0007
C                                                          95 (STMT)
            QSP = (YY*W(I+1,J,2)-XY*W(I+1,J,3))/W(I+1,J,1)         0008
C                                                          92 (STMT)
            QSM = (YY*W(I,J,2)-XY*W(I,J,3))/W(I,J,1)               0009
C                                                          72 (STMT)
            FS(I,J,1) = QSP*W(I+1,J,1)+QSM*W(I,J,1)                0010
C                                                          78 (STMT)
            FS(I,J,2) = QSP*W(I+1,J,2)+QSM*W(I,J,2)+YY*PA          0011
C                                                          78 (STMT)
            FS(I,J,3) = QSP*W(I+1,J,3)+QSM*W(I,J,3)-XY*PA          0012
C                                                         121 (STMT)
            FS(I,J,4) = QSP*(W(I+1,J,4)+P(I+1,J))+QSM*(W(I,J,4)+P(I,J   0013
     &      ))                                                     0013
C                                                           0 (STMT)
10          CONTINUE                                               0014
C                                  208*IL.JL - 208*IL - 136*JL + 219 (DO)
      DO 20 N = 1, 4                                               0016
C                                     52*IL.JL - 52*IL - 34*JL + 45 (DO)
         DO 20 J = 2, JL                                           0018
C                                                      55*IL - 41 (DO)
            DO 20 I = 2, IL                                        0020
C                                                          45 (STMT)
               DW(I,J,N) = FS(I,J,N)-FS(I-1,J,N)                   0021
C                                                           0 (STMT)
20             CONTINUE                                            0022
C                                                    167*IL - 156 (DO)
      DO 25 I = 2, IL                                              0024
C                                                          54 (STMT)
         XX = X(I,1,1)-X(I-1,1,1)                                  0025
C                                                          54 (STMT)
         YX = X(I,1,2)-X(I-1,1,2)                                  0026
C                                                          23 (STMT)
         PA = P(I,2)+P(I,1)                                        0027
C                                                           5 (STMT)
         FS(I,1,1) = 0.                                            0028
C                                                          10 (STMT)
         FS(I,1,2) = -YX*PA                                        0029
C                                                           9 (STMT)
         FS(I,1,3) = XX*PA                                         0030
```

```
C                                                                5 (STMT)
         FS(I,1,4) = 0.                                             0031
C                                                                0 (STMT)
25       CONTINUE                                                   0032
C                                    699*IL.JL - 699*IL - 681*JL + 692 (DO)
      DO 30 J = 2, JL                                               0034
C                                                        699*IL - 688 (DO)
         DO 30 I = 2, IL                                            0036
C                                                               54 (STMT)
            XX = X(I,J,1)-X(I-1,J,1)                                0037
C                                                               54 (STMT)
            YX = X(I,J,2)-X(I-1,J,2)                                0038
C                                                               48 (STMT)
            PA = P(I,J+1)+P(I,J)                                    0039
C                                                               95 (STMT)
            QSP = (XX*W(I,J+1,3)-YX*W(I,J+1,2))/W(I,J+1,1)          0040
C                                                               92 (STMT)
            QSM = (XX*W(I,J,3)-YX*W(I,J,2))/W(I,J,1)                0041
C                                                               72 (STMT)
            FS(I,J,1) = QSP*W(I,J+1,1)+QSM*W(I,J,1)                 0042
C                                                               78 (STMT)
            FS(I,J,2) = QSP*W(I,J+1,2)+QSM*W(I,J,2)-YX*PA           0043
C                                                               78 (STMT)
            FS(I,J,3) = QSP*W(I,J+1,3)+QSM*W(I,J,3)+XX*PA           0044
C                                                              121 (STMT)
            FS(I,J,4) = QSP*(W(I,J+1,4)+P(I,J+1))+QSM*(W(I,J,4)+P(I,J   0045
     &      ))                                                      0045
C                                                                0 (STMT)
30          CONTINUE                                                0046
C                                    272*IL.JL - 272*IL - 200*JL + 283 (DO)
      DO 40 N = 1, 4                                                0048
C                                        68*IL.JL - 68*IL - 50*JL + 61 (DO)
         DO 40 J = 2, JL                                            0050
C                                                          68*IL - 57 (DO)
            DO 40 I = 2, IL                                         0052
C                                                               61 (STMT)
               DW(I,J,N) = DW(I,J,N)+FS(I,J,N)-FS(I,J-1,N)          0053
C                                                                0 (STMT)
40             CONTINUE                                             0054
C                                                                2 (STMT)
      RETURN
C                                                                0 (STMT)
      END
```

We have got the complexity of the subroutine **eflux.f** with our complexity estimator:

$$C(eflux) = 1878 * IL.JL - 1711 * IL - 999 * JL + 1033 \quad (5.1)$$

Now we explain how our estimator obtain that result. In **eflux** subroutine, there are five outer loops, whose loop labels are 10, 20, 25, 30 and 40 respectively. We do not need to look into every one, we just need to pick up one as example. We chose the very first loop, i.e., Loop 10. There are 9 statements in the loop body of Loop 10. In order to explain clearly, we use the statement number shown in the complexity result of **eflux**. For the first statement of the loop body, which is numbered 0005 in the complexity result output, has 54 as cost of the statement. Because X is passed as argument and from Table 3.17, we know that the cost for iic is 25. So $X(I,J,1)$ is 25

and $X(I, J-1, 1)$ is 26 since there is an integer subtraction inside. From Table 3.10, we know that floating-point subtraction has 2 costs and from Table 3.11 we know that memory access to a floating point variable is 1. Finally, we can add them together $1 + 25 + 2 + 26 = 54$.

With same reasoning, we can get the results of all 9 statements. The sum of their costs is 692. For the inner loop of Loop 10, and from Table 3.20, we know the loop init is 11 and branching is 7, we have the Equation (2.4), so the inner loop cost is $11 + IL * (692 + 7) = 699 * IL + 11$. For the outer loop, we have $11 + (JL - 1) * [(699 * IL + 11) + 7] = 699 * IL * JL - 699 * IL + 18 * JL - 7$.

With the above inference in mind, we have no problem to get the result of other four loops, we add them together and get the final result shown in Equation (5.1).

Now we turn our attention to verification. Verification results are shown in Table 5.3.

Table 5.3 contains results from 50 tests. The first column is the test number; the second and third are values that two variables IL and JL could take; the next is the estimated number of executed instructions, calculated by the Equation (5.1); the fifth column is the estimated number of microseconds, obtained by assuming each instruction takes about 0.05μsec; the sixth is the number of microseconds we measured; the final is the difference in percentage.

For example, for test 14, we introduce the real values of IL and JL into the Equation (5.1). We get the result 7478510, which is the number of instruction cycles. As we have assumed in Section 3.5.1, each instruction cycle is about 0.05μsec on SUN SPARCstation 2. Hence, we have 373925μsec as prediction of execution time for corresponding IL and JL. We timed the code with same IL and JL, the real time is 364199μsec. The difference is about 2.67%.

5.4 Summary

In this chapter, we have first compared the result generated by our static estimator with the one obtained manually by a researcher. It showed that in most cases, our estimator works well, and if we can instruct it a little bit, e.g., indicate probabilities for some statements, we could get very satisfactory result.

We chose a benchmark program FLO52 from PERFECT Club Benchmark Suite to validate our complexity results obtained from our evaluation program.

Then, we timed a subroutine **EFLUX** from FLO52 program of Perfect Club, and compared it with complexity result. Despite the large range of two variables explored, we got very good result with a very simple cost table **SunSPARC**, the prediction error is less than 5%. It is necessary for the performance evaluator to be aware of the relative amount of time required by different operations. For instance on the IBM RISC System/6000 computer, handfuls of floating-point multiplication and additions can be performed in the time it takes to do a single fixed-point multiply or to service a cache miss or even to get the result of a comparison to the branch unit. More advanced techniques are needed for the superscalar superpipelined machines.

Test	IL	JL	Est. Cyc.	Estimated (μsec)	Measured (μsec)	Difference(%)
1	193	33	11598825	579941	567799	-2.13
2	193	30	10514460	525723	513299	-2.42
3	193	27	9430095	471505	459133	-2.69
4	193	24	8345730	417286	404766	-3.09
5	193	21	7261365	363068	352999	-2.85
6	183	33	10996195	549810	535633	-2.64
7	183	30	9968170	498408	489666	-1.78
8	183	27	8940145	447007	440033	-1.58
9	183	24	7912120	395606	388533	-1.82
10	183	21	6884095	344205	338599	-1.65
11	173	33	10393565	519678	511999	-1.49
12	173	30	9421880	471094	458733	-2.69
13	173	27	8450195	422510	410733	-2.86
14	173	24	7478510	373925	364199	-2.67
15	173	21	6506825	325341	318299	-2.21
16	163	33	9790935	489547	479299	-2.13
17	163	30	8875590	443779	433599	-2.34
18	163	27	7960245	398012	388966	-2.32
19	163	24	7044900	352245	344233	-2.32
20	163	21	6129555	306478	300066	-2.13
21	153	33	9188305	459415	448499	-2.43
22	153	30	8329300	416465	406699	-2.40
23	153	27	7470295	373515	365399	-2.22
24	153	24	6611290	330564	323433	-2.20
25	153	21	5752285	287614	280599	-2.50
26	143	33	8585675	429284	419566	-2.31
27	143	30	7783010	389150	381033	-2.13
28	143	27	6980345	349017	341366	-2.24
29	143	24	6177680	308884	302433	-2.13
30	143	21	5375015	268751	262299	-2.45
31	133	33	7983045	399152	390599	-2.18
32	133	30	7236720	361836	353899	-2.24
33	133	27	6490395	324520	317766	-2.12
34	133	24	5744070	287203	280366	-2.43
35	133	21	4997745	249887	244266	-2.30
36	123	33	7380415	369021	361599	-2.05
37	123	30	6690430	334521	327233	-2.22
38	123	27	6000445	300022	293233	-2.31
39	123	24	5310460	265523	259699	-2.24
40	123	21	4620475	231024	225166	-2.60
41	113	33	6777785	338889	331466	-2.23
42	113	30	6144140	307207	301266	-1.97
43	113	27	5510495	275525	269699	-2.16
44	113	24	4876850	243842	238333	-2.31
45	113	21	4243205	212160	207199	-2.39
46	103	33	6175155	308758	303166	-1.84
47	103	30	5597850	279892	274199	-2.07
48	103	27	5020545	251027	246233	-1.94
49	103	24	4443240	222162	217499	-2.14
50	103	21	3865935	193297	189366	-2.07

Table 5.3: Verification of Eflux subroutine of FLO52

Chapter 6

Experiments on Parallel Machine

We have shown that good complexity estimation could be obtained statically. We now want to extend these results to parallel machines which are hot topics in performance debugging today.

Precise prediction of an application execution time on a parallel computer is important in performance debugging. In this chapter, we present experimental results obtained with several sample tests on a NUMA shared-memory multiprocessor MIMD machine, the BBN TC2000[1]. We have found that the experience obtained from sequential machines could easily be extended to parallel machines with minor modification of the algorithm to handle parallel loops. Several empirical models have been built from the test results and proven to be correct by the experimentation. These models could be used to characterize the execution time of parallel loops.

After having gathered some experience on execution time prediction with sequential machines, we got some first-hand experience on parallel machines. We ran many test programs and traced the activities of the parallel machines. Because we did not have much time to observe the activities of parallel machines, we only executed a small group of sample programs to get some ideas about the behavior of parallel machines.

In this chapter, we first briefly present parallel machines, why they are difficult to handle. Then we introduce some definitions that are used in the chapter. Later, we present BBN TC2000, its architecture, operating system and performance analyzer. We present our experimental tests with varying number of processors. Finally, we extract the algorithms from the experimentation and verify them by comparing the results.

6.1 Introduction

Performance is an important aspect of sequential programming, and even more so far for parallel programming. When one sits down to write a parallel program, although with much more difficulties than sequential one, it is expected that the parallel program will execute faster in parallel than in sequential. Furthermore, there is an expectation that increasing the number of processors will result in a commensurate decrease in execution time. Unfortunately, it is generally not the case.

[1]We thank sincerely CERFACS for allowing us use their BBN TC2000 computer.

```
/* Multiprocessor "Hello World" program */

#include <us.h>

long  *  nodecount;

PrintHello (dummy, index)
    int dummy, index;
{   printf ("Index=%d: Hello World from node #%d (= h/w node #%x)\n",
         index, PhysProcToUsProc(Proc_Node), Proc_Node);
    xmatomadd31( nodecount, -1 );
    while ( *nodecount != 0 ) LockWait (0);
}

main ()
{   InitializeUs ();
    nodecount = (long *) UsAlloc (sizeof (short));
    * nodecount = TotalProcsAvailable ();
    printf ("\nThere are %d nodes in this cluster\n\n",
           * nodecount);
    Share (& nodecount);
    GenOnI (PrintHello, * nodecount);
}
```

Figure 6.1: Example 1: Source Code

We have a very simple multiprocessor program, see Figure 6.1, it is a multiprocessor version in C of the “Hello World” program in [KR88] and is only a little more complicated.

This program causes each processor to print out the greeting message:

```
Index=n: Hello World from node #m (= h/w node #%x)
```

The “Hello World” program uses **UsAlloc** to reserve space in globally shared memory for *nodecount*, a variable used for bookkeeping by the processors. *nodecount* is initialized with the number of processors available to the current execution of the program, a number obtained via **TotalProcsAvailable**. After using **Share** to propagate the location of *nodecount* to other processors, the program then uses **GenOnI** to generate tasks that print the “Hello” message for each processor. The only tricky part is ensuring that each processor performs exactly one task. In general, without some form of coordination, some processors could get more than one task and others might get none. For this program, the coordination is simple. After printing its message, each processor atomically decrements a counter maintained in globally shared-memory (*nodecount*) with **xmatomadd31** and then waits until the counter indicates that all messages have been printed. This guarantees that no processor finishes its task until all messages have been printed. Therefore, all tasks are generated before any processor finishes. The program is shown in Figure 6.1.

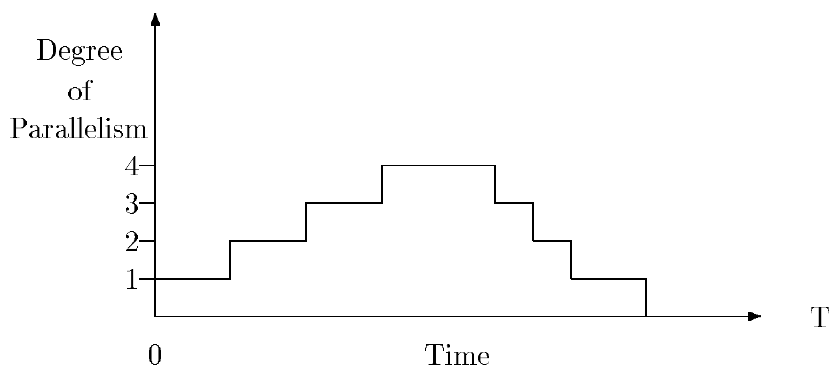


Figure 6.2: Parallelism Profile of an Application

An interesting fact is that, if we eliminate the busy waiting coordination part of the program shown in Figure 6.1, only three processors participate the work. One gets 6, one gets 4 and one 2. The important fact is that it is 12 times faster than the one with coordination. This illustrates that if the user does not have much work to do on a parallel machine, the more processors the worse performance.

6.2 Background and Terminology

Generally speaking, researchers consider three main degradations of parallelism, *load imbalance*, *communication cost* and *runtime overhead*. The first one is due to uneven distribution of workload among processors, and is application-dependent. The second one is due to the communication processing and latency, and it depends on both the application and underlying parallel computer. The last one is caused by setting up the working environment, i.e., the function stack, etc. To give an accurate performance measurement, all degradations need to be considered. Load imbalance is measured by *degree of parallelism*. We give the definition below:

Definition 6.1 *Effective Degree of Parallelism: the maximum number of processors busy computing at a particular instant in time.*

Apparently, the parallelism degree is function of time. All processors cannot start to work at the same time. We can imagine that the processors get to work one by one until no more processor is available on the machine, and when there are no more tasks, the processors gradually stop working. By drawing the degree of parallelism over the execution time of an algorithm, a graph can be obtained, see Figure 6.2. We refer to this graph as the parallelism profile.

Definition 6.2 *Parallelism Profile: The parallelism degree of a program over time.*

Communication cost is an important factor contributing to the complexity of a parallel algorithm. Unlike the parallelism degree, which depends on the application as well as on runtime support and the operating system, communication cost is machine-dependent. It depends on the communication

network, the routing scheme, and the adapted switching technique, the runtime support and the operating system.

In our testing suite, we are considering a relatively simpler case to deal only with the parallelism degree and to ignore communication costs.. We now introduce some definitions here.

Definition 6.3 *Time of Task Duration Tt: The execution time needed to finish one task.*

Also, we chose identical tasks for our first experiment so as to distinguish between runtime and operating system influence on one hand and load imbalance on the other.

Definition 6.4 *Interstate Time Ti: The wasted time between the execution of two consecutive tasks on the some processor.*

All processors are identical, so we can assume that interstate time is same on any processor. On some machines, like BBN TC2000, we need to designate the maximum number of processors for a session, we give the definition:

Definition 6.5 *Number of Allotted Processor Np: The number of processors available for a certain job, which is allotted before the job gets started.*

Np can be allotted by **cluster** (see [BBNXtra]) and its value can be obtained by **TotalProcsAvailable** during the execution. Of course, it can not be larger than the number of processors available on the machine. If you try to allot more processors, the machine will warn you and allot the maximum processors it actually has. For example, BBN TC2000 machine in Toulouse has maximum 64 processors, and if some other sessions use 30 processors, only 34 processors are available, if you want 40, you will only get 34, because only 34 processors are available.

When processors are allotted, we must know how many of them actively participate the job, we introduce the number of effective processor definition:

Definition 6.6 *Number of Effective Processor Ne: The number of processors that fully participate, i.e., execute at least two tasks of the program.*

Ne in general case is equal to Np when the program is large and parallel enough. But when the job is not big enough, all processors available cannot be all exploited, only a small part of them can be used, as explained before with the parallel "Hello world" program. So there are two kinds of "idle" processors, one is formal processor which symbolically does one job, and the other is null processor, to its turn, it is even worse because it does nothing at all. We give both definitions in the following:

Definition 6.7 *Number of Formal Processor Nf: The processor that only does one task during the work.*

The processor participates to the execution but it executes only one task and quits. In this case, more processors in the execution can only deteriorate the performance because of the heave start-up overhead. This overhead may be much larger than the task execution time.

Definition 6.8 *Number of Null Processor Nn: Given the amount of tasks, the number of processors that do not participate the job at all.*

Nn is the number of processors that are allotted before the real run, but do not execute any job at all because of task is not heavy. So we have the following equation:

$$Np = Ne + Nf + Nn \tag{6.1}$$

Definition 6.9 *Number of Optimal Processor Nd: Given the amount of tasks, the number of processors that can be effectively exploited to execute the job.*

As long as Np is less than Nd, the more processors, the shorter the execution time is; i.e., the formal processor can be fully eliminated. And when Np is greater that Nd, the more processors, the longer the execution time is.

6.3 The Machine Description

Before we present our test program, we briefly introduce the machine used for the experiment — BBN TC2000 and its programming and performance debugging environment.

6.3.1 Basic Characteristics of TC2000

The TC2000 computer is a powerful multiprocessor [BBNInside]. It builds upon BBN's experience in design of parallel processor and extends the state-of-the-art of parallel machine. It has the following features:

1. It employs a number of Motorola 88K **microprocessors**, each executing individually on the users' tasks in a controlled and coordinated way.
2. It employs **shared memory** to store information. All main memory of the machine is accessible to every processor, but with varying execution time (i.e., NUMA machines).
3. The TC2000 processors access the shared memory through an interconnection network called the **Butterfly switch.** The switch provides a fast, efficient and effective access path.
4. The TC2000 design is **modular** and **scalable.** Processors, memory and IO capacitty can be added board by board, as needed by the user.
5. The TC2000 has a **balanced** architecture. The integer computation, floating point computation, memory and input/output capabilities are approximately equal in power.

6.3.2 Architecture of TC2000

The TC2000 architecture consists of **function boards** interconnected by a high performance **Butterfly Switch.** In addition, the **Test and Control System** (TCS) monitors the entire machine.

Each **function board** contains some or all of the following: a processor, cache and memory management unit, memory, a VMEbus interface, a switch interface, TCS circuitry and power supplies. The **processor** is Motorola 88000 chip group comprised of an 88100 CPU chip and at least two 88200 cache/memory management unit (CMMU) chips. One or two 88200 chip(s) handle instruction reference, and one other 88200 chip handles data reference. The **memory** is dynamic RAM, with parity, addressable as byte, halfword (2 bytes) or word (4 bytes), aligned on boundaries of the size being addressed.

6.3.3 nX Operating System

It is the operating system base on UNIX that runs on the TC2000 to support parallel multiprocess applications.

There is a big difference between nX operating system and the classical one like SunOS. nX, just as you can imagine, is supposed to be an n-processors X-Window operating system, so-called **nX**. The user needs to specify how many processors (called nodes in BBN jargon) are allotted to a certain application.

6.3.4 Uniform System Library

The TC2000 hardware and nX operating system are a foundation on which a variety of software structures may be built. The Uniform System Library contains subroutines that can be used with either C or Fortran programs. The approach to memory management used by the Uniform System Library is based on two principles:

- Use a single address space shared by all processes to simplify programming; and
- Scatter application data across all memories of the machine to reduce memory contention.

The benefit of this strategy is that you can treat all processors as identical workers, each able to do any application task, since each has access to all application data. This greatly simplifies programming the machine, a benefit that far outweighs the modest cost in average time.

The TC2000 multiprocessor can work very efficiently with individual tasks a few milleseconds long; if necessary, it can also work on tasks in the hundred of microseconds range. For shorter tasks, various overheads begin to severely deteriorate performance.

6.3.5 Xtra Programming Environment

The X Tools for Runtime Analysis, called **Xtra**, programming environment is a software development environment, based on the X-Window System, for debugging, analyzing, and tuning the

performance of parallel programs. See [BBNXtra] for more information. The Xtra environment includes (1) the TotalView debugger; (2) the ELOG library for generating user event logs; and (3) the Gist performance analyzer for analyzing event logs. The Xtra package also includes special macros and the *gisttext* tool for generating text event logs. This package is designed to help users develop parallel programs on the TC2000 system and port serial applications to the parallel environment of the TC2000 system.

TotalView Debugger

The principal advantage of the **TotalView** debugger over conventional UNIX debuggers (e.g. *dbx*) is that it was designed to debug the multiple processes of parallel programs. Furthermore, since the **TotalView** debugger runs under the X Window System, its graphic interface is easy to learn and ideal for conveying the concepts of parallel programs. I did not use the debugger much because my test program were simple and because I only had remote access to the machine.

The ELOG library and Gist performance analyzer

I have used the other two components of the Xtra environment — the ELOG library (for Event LOG) and Gist performance analyzer. They are designed to tune the performance of multiprocess programs. The performance analyzer also has a graphic interface which is based on the X-Window System.

We need to present this utility in more detail because all the data we gathered about the BBN TC2000 were produced with it.

6.4 Gist Performance Analyzer

This utility is provided by the manufacturer. We explain how to create the events, obtain data and analyze them.

6.4.1 Logging Events with the Event Logging Library

An *event log* is a file containing a record of events for **each process** in a program. Each event recorded for a process contains:

- A time stamp, which is the real-time clock value when the event was recorded. The resolution is XXX microsecond (μsec).
- An event code, which is an integer identifying the event.
- A user-defined 32-bit data item, which is used by the Gist programs to analyze the log file.

In order to create an event log, we need to complete the following steps:

1. Decide what events should be logged from the program.
2. Understand the library of event logging routines.
3. Instrument the program with the appropriate event logging routines.
4. Compile and link the program with the appropriate compiler switches.
5. Generate the event log by running the program.

We will elaborate them one by one in the following.

Deciding on the Events to be logged

The event logging routines can be used for two purposes: 1) analyzing the performance of a program and 2) debugging a program. We need to pay attention to several important points here:

- Log events for the major components of user program, such as procedures, functions and subroutines.
- Log events for any parts of user program that use multiprocess programming.
- If the application requires user input, gather it before user start logging.

Understanding the Routines

The event logging library, **/usr/lib/libelog.a** consists of six routines. These routines initialize an event log, allocate space for individual processes to record events, specify what to do if a buffer overflows, define events to be logged, log events, and write the recorded events to the event log file. You must link your program to the **libelog.a** library in order to call these routines.

It takes about 6 to 7 microseconds on the TC2000 to log an event. The amount of runtime overhead incurred by the event log instrumentation depends on how the event logging routines are used. The instrumentation in this work uses event-logging macros, which are activated if the pseudo-variable ELOG is defined. If not, program will incur **no runtime overhead** due to the presence of the event log instrumentation.

Instrumenting a Program

We created a sample C program, **simple.c** which generates tasks that simply use up time in the processes busy waiting. We can specify the number of tasks, the size of each task (e.g. the number of iterations of the task), and the number of event log files among other things.

Compiling and Linking the Program

For C programs, we compile in different ways depending on whether or not we define the symbol ELOG for expansion of the event logging macros. To compile the program with macros, do as follows:

```
cc -DELOG -o simple.c -lus -leleog simple.c
```

where

1. -DELOG defines the symbol ELOG (required when the C macros are used)
2. **-lus** links the prograam to the Uniform Systems library (required when you use the Uniform System)
3. **-lelog** links the program to the event logging library (required all the time).

Initializing the Event Log with giststart

giststart is a program that initializes the event log file from the command line. Use this method when it is difficult or inconvenient to instrument user program with a call to ELOG_INIT.

6.4.2 Output of Event Logs with Gist

If **gist** does not provide a function that needed to analyze your event log, you can write the event log in a text file. And then you can analyze the text file with standard operating system utilities, such as **awk, sed**, etc. or write your own program to perform the analysis desired. We use this method for our simple tests.

Two kinds of output formats are available. One is brief format, a relatively simpler format, and the other is **gisttext** format. The two formats are simply presented below.

Brief Format

A text file for an event log in *brief* format contains four columns of information for each event:

1. The process label (in hexadecimal)
2. The time the event occurred, in microseconds
3. The event codes
4. The datum logged with the event

Figure 6.3 contains an example of the standard output of brief format. But this format is less useful than the following one, so we did not use it.

```
0 0 1 200
0 750 3 160
c 1563 3 116
. .... . ...
. .... . ...
```

Figure 6.3: Brief Format of Gist Output

Gisttext Format

A text file in **gisttext** format contains the same seven columns as the output produced by the **gisttext** program. They are:

1. The line type (always E, for Event)
2. The processor and process for the process trace
3. The time the event occurred, in microseconds
4. The elapsed time since the previous event, in microseconds
5. The unique integer(event code) identifying the event
6. A text string identifying the type of event
7. The formatted data item for the event

Figure 6.4 is the standard output of **gisttext** format. We will analyze this format in the following section.

6.4.3 Analyzing Event Logs with the Gist

This section explains how to analyze event logs that were generated by the event logging library introduced in the previous section.

Gist can display more information about each event in a process trace. Specifically, **Gist** can display (1) the time the event occurred in microseconds, (2) the event code, (3) the event name and (4) the data that is logged with the event. Now let us look at Figure 6.4, for the Node 0x0, at the real time 0 microseconds, the work starts and lasts 0, after that there is an interstate (waiting time) 186μsec and worker task #0 starts there, at 598 μsec worker task #0 finishes, that is after 412μsec. Node 0x0 begins the work again, at 625μsec for the worker task #1 after 27μsec interstate time and ends at 1031μsec. During this time, another Node 0xe starts the worker task #2 and finishes at 1411μsec. Node 0x0 starts at 1049μsec for the worker task #3 and ends at 1455μsec. And the same things for other nodes.

```
E 0x0        0     0  1  Start Generator
E 0x0      186   186  2  Start Worker Task #0
E 0x0      598   412  3  End Worker Task #0
E 0x0      625    27  2  Start Worker Task #1
E 0xe      998   373  2  Start Worker Task #2
E 0x0     1031    33  3  End Worker Task #1
E 0x0     1049    18  2  Start Worker Task #3
E 0xe     1411   362  3  End Worker Task #2
E 0xe     1432    21  2  Start Worker Task #4
E 0x0     1455    23  3  End Worker Task #3
E 0x0     1473    18  2  Start Worker Task #5
E 0xe     1838   365  3  End Worker Task #4
E 0xe     1854    16  2  Start Worker Task #6
E 0x0     1879    25  3  End Worker Task #5
E 0x0     1898    19  2  Start Worker Task #7
E 0xe     2260   362  3  End Worker Task #6
E 0xe     2277    17  2  Start Worker Task #8
```

Figure 6.4: Gisttext Format of Gist Output

So actually, we have every details about execution performed by the node, i.e., for each node, how many tasks it has executed, its overall start-up time and finish time, even start-up time and finish time for every task it has executed. We know exactly when a task starts and finishes, and the interstate between two consecutive tasks.

It is convenient to write a **awk** script to analyze the output of **gisttext**. Figure 6.5 shows the **awk** script which is used in the next section to analyze the traces produced with gist.

6.4.4 Summary of the Analyzer

Up to now, we have every information necessary to analyze the trace.

Figure 6.6 is the complete procedure for trace analysis. We have got two sorts of outputs after a bbn-ized program is run: (1) standard result of the program on issue, but it is out of our consideration, we simply ignore this output, and (2) tracing side effects of the program. While the standard result is produced, logging file with events is formed too. As we have mentioned earlier, all information about each node is available in that file.

```
BEGIN { maxnu = 0 }
# $2 is the processor number
{ c=0 # convert proc number from hex to decimal
        for (i=3;i<=length($2);i++)
        {
        a=substr($2,i,1)
        if (a=="a") a=10
        if (a=="b") a=11
        if (a=="c") a=12
        if (a=="d") a=13
        if (a=="e") a=14
        if (a=="f") a=15
        a=a*1
        c=16*c+a
        }
# $3 is the time stamp
# event #5=start of task, event #7=end of task, c = proc number
# each time a processor finishes a task, compute it is duration
        if ($5==2) { debut[c]=$3 }
        if ($5==3)
{
line += 1 # finished number of the task
duree[c]=$3-debut[c]  # dural time of the task
   proc = c
   if ( length(begin[proc]) == 0 ) begin[proc] = line
   if ( end[proc] < line ) end[proc] = line
   total[proc] = total[proc] + duree[c]
   count[proc] += 1
   if ( maxnu < proc ) maxnu = proc
}
}
END { printf("\t Proc \t Ntask \t First \t Last \t Average Time Per Task\n")
      for ( i=0;i<= maxnu;i++)
printf("\t %d \t %d \t %d \t %d \t %f\n",\
i,count[i],begin[i],end[i],total[i]/count[i])
      for ( i=0;i<= maxnu;i++) {
sum_count += count[i]
sum_squa += count[i]*count[i]
      }
      av_count = sum_count/(maxnu+1);
      sigma = sqrt(sum_squa/(maxnu+1) - av_count*av_count)
      printf("For Task Distribution: %d tasks per processor\n",av_count)
      printf("The absolute standard deviation is %f\n\
              The relative standard deviation is %f %\n",\
              sigma, (sigma*100/av_count) );
}
```

Figure 6.5: Awk script to analyze the gisttext output

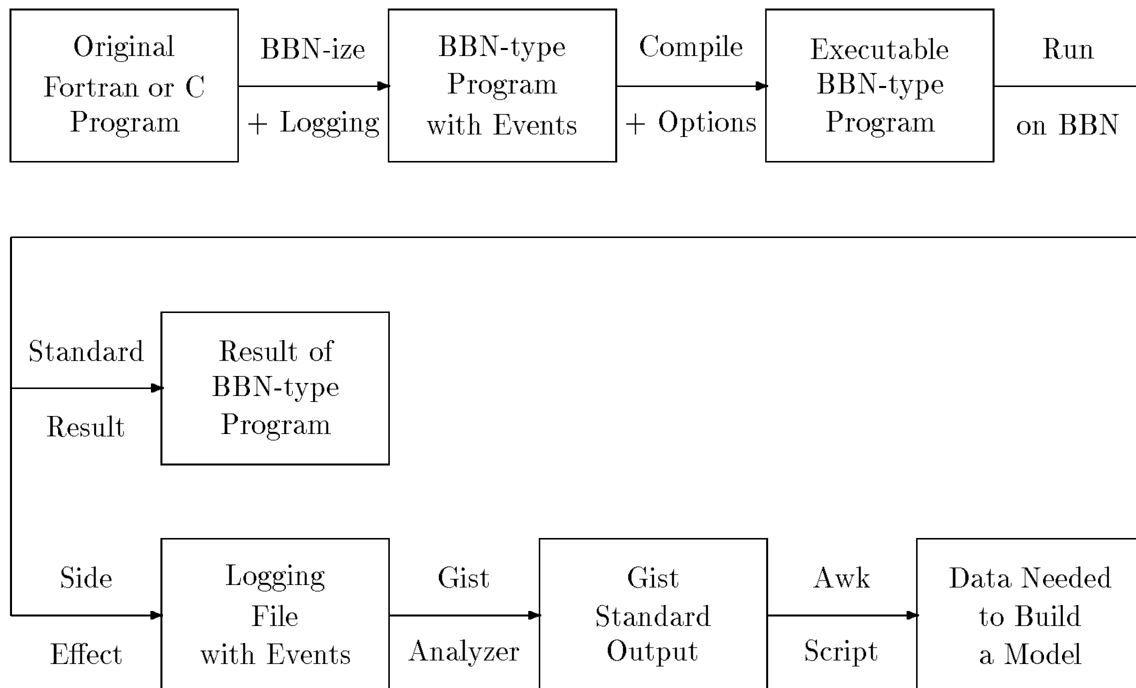


Figure 6.6: The Complete Procedure of Analysis

Proc	Ntask	StartTime	FinishTime	AverageTimePerTask	AverageInterval
0	1006	4370	4104378	4059.95	15.62
1	995	49942	4105011	4059.36	16.10
2	1000	25758	4101243	4059.93	15.57
3	996	44032	4103198	4059.88	15.61
4	999	31689	4103061	4059.84	15.62
5	1005	11605	4102828	4060.00	10.88
6	1002	19427	4102786	4059.37	15.86
7	997	37893	4101113	4059.60	15.86

Figure 6.7: Data of Test Program on 8 nodes

6.5 Experimentations

In this section, we present the experiments we have on the BBN TC2000 machine. First we explain the sample program, then introduce the startup time and finish time.

6.5.1 Explanation of Test Program

We implemented the **simple.c** program to obtain the data for each processor. Recall that this program generates tasks simply use up a certain amount of processing time. We can specify the number of tasks, the size of each task (e.g. the number of iterations of the task), and the number of the event log files among other things.

To save space, we do not need to present all data obtained from the experiments. We only present the three representative ones, when number of processors is 8, 16 and 24 respectively.

In Figure 6.7, Figure 6.8 and Figure 6.9, note that the first processor is numbered 0 instead of 1.

6.5.2 Startup Time

From the data shown in Figure 6.7, Figure 6.8 and Figure 6.9, we can display all start-up times in a graph, shown in Figure 6.10. From the picture, we see that start-up times are a linear function of processor number.

We give the definition of that line:

Definition 6.10 *Start-up line: The line formed by start-up points of all processors.*

Because we can suppose that start-up makes a straight line which starts from the origin, in order to facilitate the stating, we introduce the following definition:

Definition 6.11 *Start-up angle α: The angle formed by start-up line and processor-axis.*

In Figure 6.12, we show the angle α. So tgα has the physical meaning, the time to start another processor, so it introduces another definition:

Proc	Ntask	StartTime	FinishTime	AverageTimePerTask	AverageInterval
0	1014	4672	4137212	4059.70	15.80
1	998	66113	4136782	4063.27	15.58
2	997	72894	4136147	4059.62	15.87
3	1008	29133	4137078	4059.50	15.85
4	1002	51676	4135487	4059.82	15.85
5	1013	12436	4136878	4060.34	11.18
6	994	86167	4136968	4059.50	15.77
7	989	104488	4135148	4059.86	15.65
8	988	110525	4137049	4059.48	15.97
9	1004	44618	4136409	4059.53	15.97
10	992	92379	4135544	4059.72	16.07
11	1000	58835	4134372	4060.05	15.50
12	1010	21324	4137709	4059.75	15.89
13	990	98309	4133768	4060.38	15.86
14	1006	37256	4137738	4060.02	16.02
15	995	79539	4135101	4060.20	15.75

Figure 6.8: Data of Test Program on 16 nodes

Proc	Ntask	StartTime	FinishTime	AverageTimePerTask	AverageInterval
0	1021	4305	4166433	4059.61	16.93
1	990	135410	4170263	4059.78	15.84
2	982	167444	4169620	4059.96	15.59
3	1010	51743	4168428	4060.09	15.85
4	985	155121	4169089	4059.24	15.87
5	1020	13455	4166355	4060.80	10.68
6	995	114219	4169168	4059.70	15.64
7	1008	59747	4167856	4059.90	15.62
8	1002	83067	4166893	4059.93	15.76
9	986	148870	4167895	4060.04	16.06
10	981	170496	4168890	4059.83	16.02
11	1000	90978	4166817	4060.05	15.81
12	988	141954	4168847	4060.13	15.69
13	999	98534	4170290	4060.11	15.74
14	983	161295	4167995	4059.74	16.27
15	991	128607	4167651	4059.65	16.09
16	1012	44208	4168585	4059.95	15.54
17	1017	21664	4166695	4059.92	15.83
18	1016	29221	4169859	4059.64	15.81
19	1004	75595	4167422	4059.80	15.74
20	1006	67424	4167617	4059.83	15.92
21	993	121806	4168712	4059.55	15.90
22	1014	36647	4169501	4059.80	16.01
23	997	106553	4170137	4059.96	15.87

Figure 6.9: Data of Test Program on 24 nodes

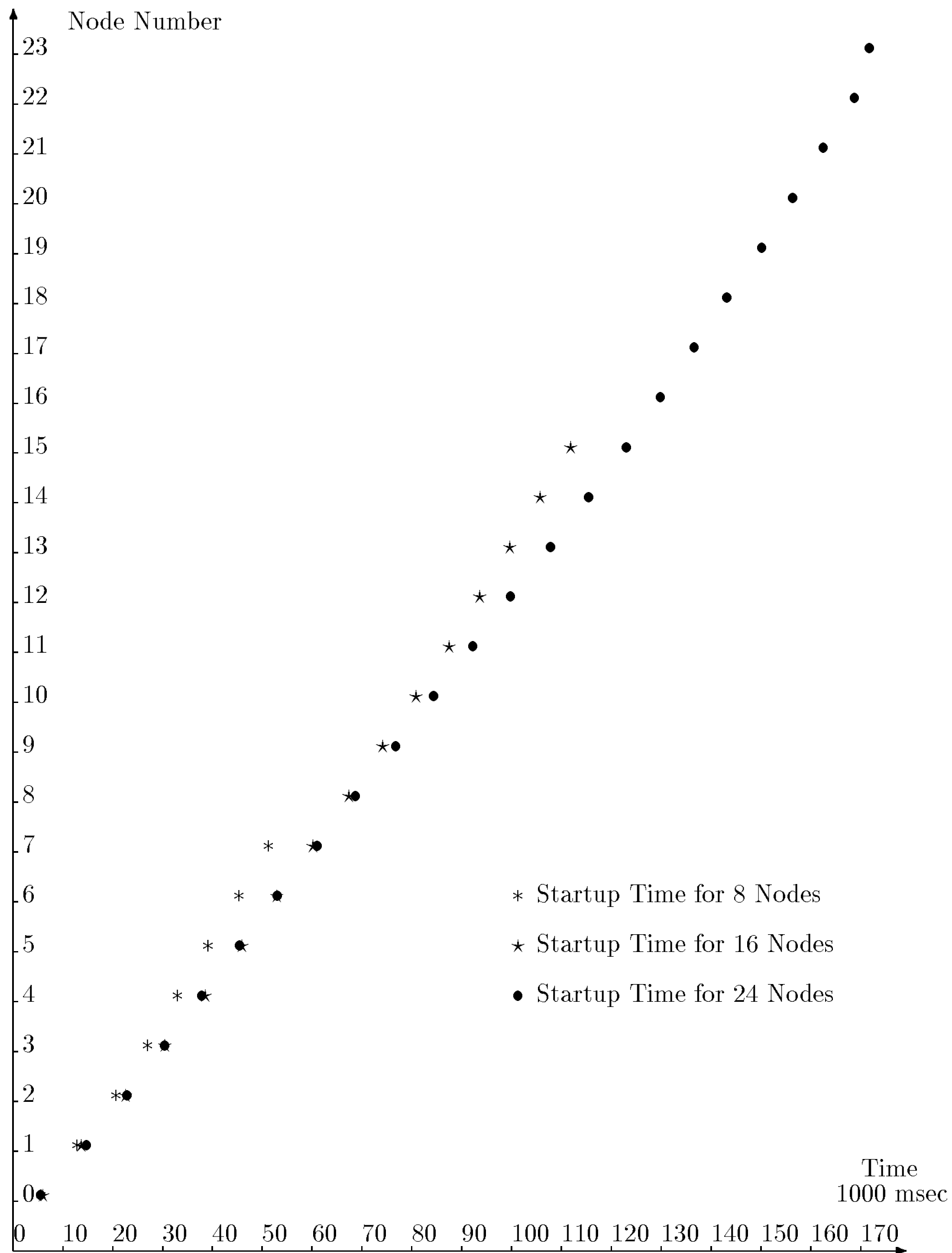


Figure 6.10: Startup Time for 8, 16 & 24 Nodes

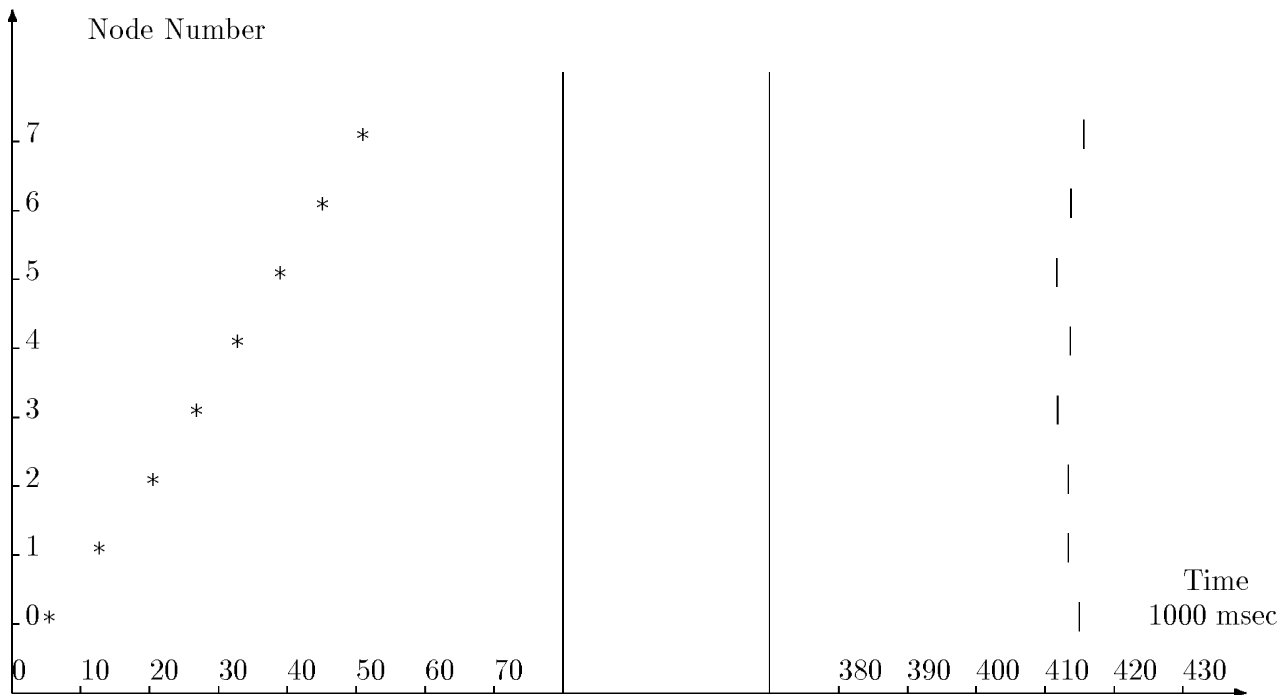


Figure 6.11: Startup Time and Finishing Time for 8 Nodes

Definition 6.12 *Start-up Slope Ta: The time needed to start up the processor.*

Here, from the data presented, we can conclude that a start-up slope of 7300 msec/processor is a good approximation of the BBN TC2000 behavior. We see in Figure 6.10 that the maximal error is less than 10%.

6.5.3 Finish Time

Now let us turn our attention to the finishing times of processors. We reuse the data for 8 nodes.

From Figure 6.11, which represents the whole execution trace with the stationary behavior we can observe that there is no idle processor (node) as long as there is a task in the waiting queue. i.e., the time between the node which finishes the job the first and the one which does the last is equal or smaller than average task time. So from the observation, we can simply assume that difference between the processor which finishes the work first and the one which does the last is exactly one task time Tt. Meanwhile, because we are only concerned with the surface surrounded by the start-up line and finish time, so we could sort the finish time in ascending order and we can also assume that finish time forms a line and difference between the processor which finishes the work first and the one which does the last is exactly one task time Tt. Figure 6.12 is a theoretical model for the parallelism profile.

We introduce another definition:

Definition 6.13 *Finish line: The theoretical line which simulates the curve formed by finishing points according to ascending order. The difference the processor which finishes the work first and*

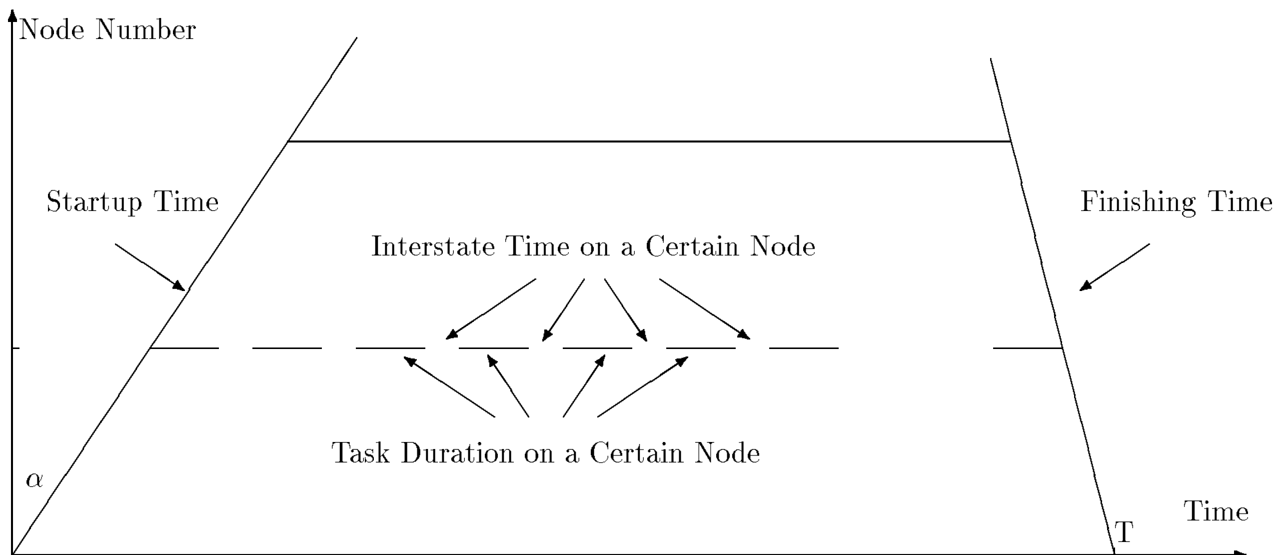


Figure 6.12: Startup Time and Finishing Time of Theoretical Model

the one which does the last is exactly one task time Tt.

The total execution time shown in Figure 6.11 is such that the surface delimited by stars and bars is proportional to the amount of work to execute. This display can be simplified by forgetting the relationship between starting and finishing times and by sorting finishing time in ascending order as shown in Figure 6.12. The reader can easily check that this shuffling does not modify the enclosed surface, i.e., the amount of work performed. However, the new display makes it easy to derive a symbolic formula linking the amount of work to the observed execution time.

6.6 Algorithm for Parallel Machines

Before we introduce our algorithms for the parallel machines, we summarize the assumptions we have made earlier.

1. Start-up is a straight line from the origin.
2. Start-up slope is a constant for a fixed BBN TC2000 machine.
3. Finish time form a line.
4. The difference between the processor which finishes the work first and the one which does the last is exactly one task time Tt.
5. All tasks are identical.

Under all these assumptions, we introduce the symbols used in our model of parallel loop execution time.

Np Number of processors available

Ne Effective Number of processors which participate

Nt Number of tasks

Tt Time for a task

Ti Time of interstate or waiting time.

Ta Start-up slope tg(α)

Depending on the amount of work available, $Ne * Tt$, with respect to the number of processors Np, two different cases must be considered:

1. There is enough work to use efficiently all processors.
2. There is not enough work.

So they will produce different shapes.

Algorithm for Trapezoid Form

First we consider a general case. We assume that all processors effectively participate the work, i.e. the number of tasks is greatly larger than the number of processors — $Nt \gg Np$; so all nodes participate i.e. $Np = Ne$. We assume that amount of time spent on a task and amount of waiting time be identical. So the parallelism profile takes the shape of trapezoid.

Under the above assumptions, for a node j, it has Nj tasks to execute, it has $(Nj - 1)$ interstates, one less than the number of tasks, the time is $Tj = Nj \cdot Tt + (Nj - 1) \cdot Ti = Nj \cdot (Tt + Ti) - Ti$.

Now consider that the trapezoid as a whole, we can imagine that $\sum_{j=1}^{Np} Nj = Nt$ and $\sum_{j=1}^{Np} 1 = Np$. So the sum of execution times for all the processors is:

$$\begin{aligned}
\sum_{j=1}^{Np} Tj &= \sum_{j=1}^{Np} (Nj \cdot Tt + (Nj - 1) \cdot Ti) \\
&= \sum_{j=1}^{Np} (Nj \cdot (Tt + Ti) - Ti) \\
&= (\sum_{j=1}^{Np} Nj) \cdot (Tt + Ti) - (\sum_{j=1}^{Np} 1) \cdot Ti \\
&= Nt \cdot (Tt + Ti) - Np \cdot Ti
\end{aligned}$$

It means that there are Nt tasks and because of there are Np processors, so there are $Nt - Np$ interstates, Np is largely less than the number of overall tasks, i.e., $Np \ll Nt$. Because the height for each Node is a unit, so the total height is the number of Nodes. From this point of view, we think the surface of the trapezoid is:

$$S1 = Nt \cdot (Tt + Ti) - Np \cdot Ti \tag{6.2}$$

The standard formula to get the surface is $Surface = Height \cdot (UpperLength + LowerLength)/2$. For the this trapezoid , we have $Height = Np$, which is the number of processors allotted (of course, $Nt \gg Np$); $LowerLength = T$, which is overall finishing time; and $UpperLength = T - Tt - Np \cdot Ta$

Here we have

$$\begin{aligned} S2 &= Height \cdot (UpperLength + LowerLength)/2 \\ &= Np \cdot ((T - Tt - Np \cdot Ta) + T)/2 \end{aligned} \tag{6.3}$$

Both $S1$ in Equation (6.2) and $S2$ in Equation (6.3) denote the same amount of trapezoid surface from two different points of view, so they are equal: $S1 = S2$. Hence we have:

$$Nt \cdot (Tt + Ti) - Np \cdot Ti = Np \cdot ((T - Tt - Np \cdot Ta) + T)/2 \tag{6.4}$$

And this equation can be simplified to:

$$T = \underbrace{\frac{Np \cdot Ta}{2}}_{part1} + \underbrace{\frac{Nt}{Np} \cdot (Tt + Ti) - Ti}_{part2} + \underbrace{\frac{Tt}{2}}_{part3} \tag{6.5}$$

This equation has an intuitive meaning shown in Figure 6.13. Part 1 in Equation (6.5) is half of start-up line projected on the time-axis. Part 2 is the average length of upper length and lower length. And Part 3 is half of finish line projected on time-axis. Obviously, this equation is intuitively proven.

Algorithm for Triangle Form

Now we consider the critical case, i.e., when not enough work is available for the processors. First we set up our session to 12 processors, which is introduced in Section 6.2, and then we gradually increase the number of tasks while keeping the task size same as shown in Figure 6.14. Finally the parallelism profile takes the shape of triangle.

Here we need to introduce another definition:

Definition 6.14 *Critical Finish Time Tc: The finish time when parallelism profile takes the shape of triangle and all processors allotted are effective. It is a function of the number of processors.*

It means that if we increase a little bit amount of work, parallelism profile will become a trapezoid, and if we decrease a little bit amount of work, not all processors are effective.

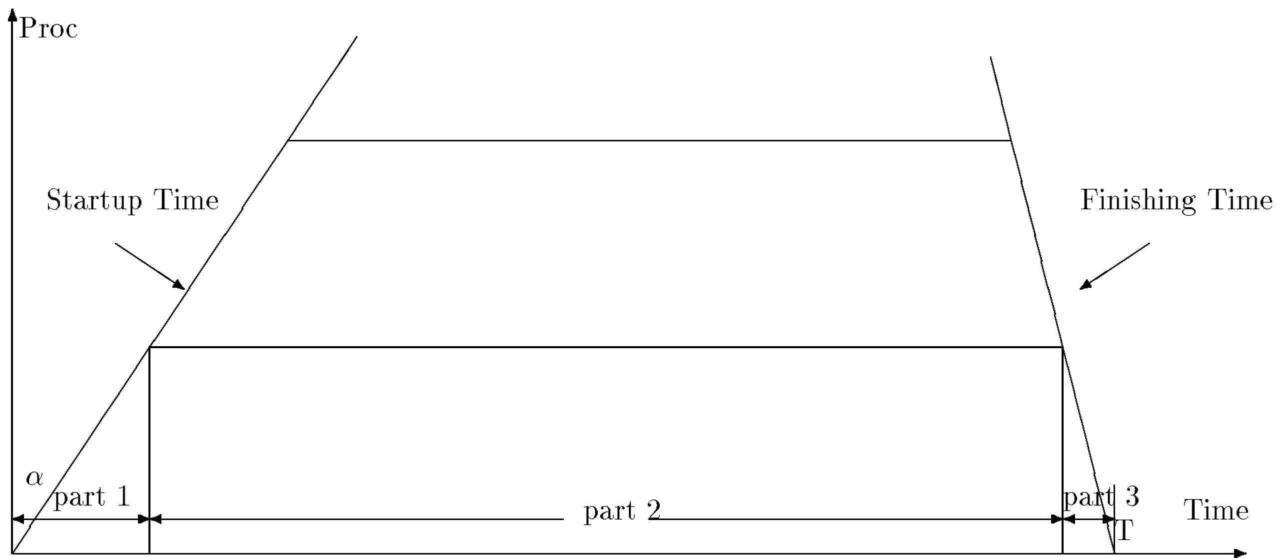


Figure 6.13: Startup Time and Finishing Time

Proc
12
11
10
9
8
7
6
5 Startup Time
Moving Finishing Time
Critical Finishing Time
Final Finishing Time
4
3
Increasing Tasks
Increasing Tasks
2
1
α
Tm
Tc
T
Time

Figure 6.14: Triangle Form

We use two different ways to compute the surface of the triangle. Using the standard formula, the surface of critical triangle S3 is:

$$S3 = \frac{Tc \cdot Ne}{2} \tag{6.6}$$

and from the point of view of execution, the surface of critical triangle S4 is:

$$S4 = Nt \cdot (Tt + Ti) - Ne \cdot Ti \tag{6.7}$$

Certainly, $S3 = S4$ because both represent the same surface. We have:

$$\frac{Tc \cdot Ne}{2} = Nt \cdot (Tt + Ti) - Ne \cdot Ti \tag{6.8}$$

with

$$Ta \quad = \quad \frac{Tc - Tt}{N} \tag{6.9}$$

After substitution, we have:

$$Tc^2 + (2 \cdot Ti - Tt) \cdot Tc - 2 \cdot Nt \cdot (Tt + Ti) \cdot Ta - 2 \cdot Ti \cdot Tt = 0 \tag{6.10}$$

Because, the total finishing time cannot be negative, so the negative solution does not make any sense. Finally, we have a unique solution:

$$Tc = \frac{-(2 \cdot Ti - Tt) + \sqrt{(2 \cdot Ti - Tt)^2 + 8 \cdot Nt \cdot (Tt + Ti) \cdot Ta + 8 \cdot Tt \cdot Ti}}{2} \tag{6.11}$$

Besides the final finishing time, we can also compute the number of processors which minimize the computation, as following:

$$Nd = \frac{(Tc - (Ti + Tt))}{Ta} \tag{6.12}$$

This is the desired number for the computation, so d is used as subscript of Nd.

6.7 Verification of Formulas

In order to check the adequacy of our model with respect to the real world, we use the data shown in Section 6.5.

Np	Experiment	Formula	Difference
8	4105011	4106084	0.03%
16	4137738	4135614	0.05%
24	4170290	4164614	0.14%

Table 6.1: Trapezoid Formulas Verification

Ne	Experiment	Formula	Difference
3	24190	22836	5.93%
4	27716	26430	4.87%
5	34605	34003	1.74%
6	42669	41258	3.42%
7	50867	49382	3.01%
8	53975	51719	4.36%
9	65658	62359	5.29%
10	73287	73045	0.33%

Table 6.2: Triangle Formulas Verification — Extracted

Verification for Trapezoid Formulae (General Case)

We only choose three cases when Np is 8, 16 and 24 for simplification.

In Table 6.1, we see that the difference between prediction quality is too satisfactory to believe. In effect, it is because task size is too small, so for a larger task size, bigger difference is expected.

Verification for Triangle Form

Now let us consider the triangle case. In Figure 6.15, we gradually increment the total tasks number, `40*p` means the total task number is 40 times the number of processors allotted, (in the example, $Np = 12$). MeasuredFT stands for Measured finishing time, EffectP for the effective number of processor, FormalP for the formal number of processor, FTFormula1 for finishing time when using formula 1, which is Tc is Equation (6.11), FTFormula2 for finishing time of formula2, which is T in Equation (6.5), DesiredP for desired number of processor, which is Nd in Equation (6.12).

Maybe the Figure 6.15 is a little bit misleading, let us just extract some data from that figure and compare these data in Table 6.2.

6.8 Discussion

During the experimentation, we have observed something that we cannot explain.

TotalTask	MeasuredFT	EffectP	FormalP	FTFormula1	FTFormula2	DesiredP	InterTime	TaskTime
40*p	14538	2	4	13596.48	44861.88	1.86	16.62	9.80
50*p	18051	2	7	16430.20	45348.77	2.25	20.99	9.88
60*p	18781	3	3	17861.55	45629.07	2.44	20.61	9.79
70*p	26292	3	9	24038.95	47112.18	3.29	37.49	9.76
80*p	24156	2	10	20097.85	46113.62	2.75	19.25	9.61
90*p	24190	3	9	22836.00	46786.21	3.12	23.24	9.89
100*p	28127	3	9	24610.48	47267.56	3.37	24.66	9.96
110*p	27247	3	9	24928.46	47356.88	3.41	22.34	9.95
120*p	27716	4	8	26430.93	47797.92	3.62	23.31	9.96
180*p	34605	5	7	34003.53	50412.94	4.65	26.41	10.30
190*p	35368	5	7	34663.08	50671.40	4.74	26.12	10.02
200*p	38801	4	8	33858.11	50355.09	4.63	22.90	9.85
210*p	39325	5	7	36478.14	51408.86	4.99	25.98	10.23
220*p	42143	4	8	35797.46	51126.75	4.90	23.51	9.77
230*p	39076	4	8	36471.70	51404.83	4.99	22.59	10.45
240*p	40526	5	7	38637.38	52334.78	5.29	24.95	10.59
250*p	43103	5	7	38797.55	52405.13	5.31	23.96	10.44
260*p	42669	6	6	41258.31	53531.47	5.65	26.86	10.55
270*p	46249	5	7	40884.16	53355.00	5.60	24.79	10.58
280*p	50867	7	5	49381.67	57743.07	6.76	39.24	10.54
290*p	48599	6	6	43308.60	54521.31	5.93	26.06	10.89
300*p	49289	6	6	46043.14	55918.41	6.30	29.64	10.73
310*p	46818	6	6	44695.75	55218.43	6.12	26.24	10.58
320*p	48375	7	5	49060.71	57558.68	6.71	32.09	10.88
330*p	53975	8	4	51719.31	59090.78	7.08	35.49	10.83
340*p	51953	7	5	50070.46	58129.85	6.85	31.26	10.87
350*p	53042	5	7	48394.65	57185.44	6.62	27.60	10.63
360*p	56590	7	5	53865.54	60384.62	7.37	34.83	11.22
370*p	94815	4	8	43990.61	54857.99	6.02	19.93	9.95
380*p	61136	8	4	58249.23	63194.87	7.97	40.31	10.72
390*p	55396	8	4	56179.14	61838.79	7.69	35.19	11.05
400*p	59231	7	5	53208.79	59979.76	7.28	29.55	10.88
410*p	59816	5	7	51022.77	58676.29	6.98	25.62	10.65
420*p	65658	9	3	62359.30	66027.05	8.54	41.84	11.07
430*p	67213	6	6	57051.22	62401.07	7.81	32.85	10.39
440*p	63268	8	4	57691.89	62820.62	7.90	32.37	10.85
450*p	73360	11	1	72766.81	74069.86	9.96	55.66	11.59
460*p	73287	10	2	73045.55	74301.15	10.00	54.63	11.66
470*p	69647	9	3	65186.14	68085.26	8.92	40.64	11.02
480*p	75210	11	1	74671.37	75673.21	10.22	55.09	11.30
490*p	74598	10	2	68117.77	70318.88	9.32	43.02	11.08
500*p	78052	11	1	74955.26	75913.97	10.26	52.94	11.27
510*p	76573	11	1	76685.03	77413.55	10.50	54.63	11.27
520*p	77898	11	1	74682.08	75678.12	10.22	50.15	11.15
530*p	86900	12	0	86509.37	86584.54	11.84	69.18	11.53
550*p	102887	12	0	99080.03	99935.51	13.56	91.88	10.18

Figure 6.15: Triangle Formulas Verification

6.8.1 Processor

There is always at least one processor which needs less waiting time considerably and in the meantime does more job than the rest. As you can see from Figure 6.7 and Figure 6.8, the 5th (logical number) processor in both experiments. We have asked the machine owner in Toulouse, they replied that all processors are identical. We once tried to locate physical number of the fastest processor in one suite, and tried the same experimentation again without that processor. It turned out that another processor emerged as the fastest one. This effect is probably due to the operating system or to the runtime library (including the logging system).

6.8.2 Intertask Time of Triangle Form

We cannot explain why waiting time of triangle is going up steadily while incrementing the number of tasks.

We are guessing that more communication cost is needed when more tasks are involved.

6.9 Summary

From the experiments we have done, we can see that, if:

1. Time for each task is identical on each processor.
2. Waiting time between the consecutive two tasks is identical on each processor. i.e., the interstate between the adjacent two tasks is same.
3. Starting times for nodes can be approximated by a line.
4. There are no idle processors as long as there is a task in the waiting queue. The time between the mode which finishes the job the first and the one which does the last is equal or smaller than the average task time.

These let us build a model of DOALL execution on the BBN TC2000. The formulae derived from the model could be used by an automatic static estimator to predict execution times on such a machine.

As we have stated several times, this experimentation is only the first step towards parallel machine modelization. Therefore all these observations support the idea that performance on parallel machines can be predicted and assure that the experience we got from the sequential machines can be expanded to parallel ones. Static analysis looks very promising.

Chapter 7

Conclusion

We started this dissertation by introducing our goal and explaining how we related an approach to performance evaluation in Chapter 1. In Chapter 2, we described the sequential algorithms we have developed to compute statically the execution time of program on a sequential machines. Then in Chapter 3 we introduced a machine model and the way to represent it. We presented the implementation of our complexity in Chapter 4. Experimental results with sequential machines are presented in Chapter 5. They showed a good agreement between prediction measurement on a SUN SPARCstation 2. Finally, we showed how parallel loop execution times can be modeled for a BBN TC2000 in Chapter 6. Formulae are derived from the model. They could be used in a static estimator for parallel machines.

In this chapter, first we summerize our contributions to static execution time estimation. Then we survey related work in the field of performance evaluation. Finally we suggest some future developments.

7.1 PIPS Complexity Estimator

We start with an ideal *real-world model* which describes exactly the way applications are executed on a specific machine. This information can only be obtained by running the application for a given input. In order to approximate the *real-world model*, we have introduced two models. The first one is a dynamic model, through which we abstract the application running on a simplified machine. The second one is a static model, which is used to describe the application from a static point of view, regradless of any specific input.

The goal in this thesis is two folds: (1) to develop a methodology for predicting the execution time of an application on a supercomputer, with presumably very expensive CPU-hour; (2) to compare the execution times between two versions of the same application with different transformations. A very important point is that we do not need exact execution time of a program, which can only be obtained by really executing the application on the specific supercomputer.

This thesis presents a new performance prediction approach, which was implemented and called "complexity" in the PIPS programming environment. The complexity is composed of a library of polynomial models of program performance and dynamically-derived program statistics, to present

running time predictions of Fortran programs.

Besides the above three steps, as we have noticed along with our experimentation, major part of calculations of real Fortran programs are in loop, it is said that it accounts for more than 90% of execution time. Therefore, for large scientific programs, while assuming branch statement probabilities are fifty-fifty and keeping unknown variables as they are, we successfully estimated a suite of real programs. Particularly, EFLUX from Perfect Club, the difference between estimated and measured is within 5%.

Finally, our ultimate purpose is to make our estimator run on the parallel machines. In order to expand this performance estimator from successful sequential program evaluation to parallel program evaluation, we have experimented several sample tests on shared-memory multi-processor MIMD machine — BBN TC2000. We have found that the experience obtained from sequential machines can be easily expanded to parallel machine with minor modification of the algorithm. At the same time, several empirical algorithms have been extracted from the test results and proven to be correct by the experimentation.

7.2 Related Work

In this section, we survey the most common techniques applied in performance prediction tools. Then we classify the existing techniques and we compare them with our own contribution.

7.2.1 Performance Evaluation Methods

We distinguish four approaches: manual evaluation, simulation, analytical and measurement.

Manual Method

The most essential and primitive method is to analyze the real program manually.

N. Emad [Emad91] detected all parallelism for a test suite from ONERA (A French Space Research Center), particularly, TMINES program. Meanwhile, she obtained the manual complexity results. So we used these results to check the results produced by our estimator. The comparison showed that human being can fail on some coefficients and that an automatic approach was very accurate most of the time, but could also dramatically fail. Information about the estimation reliability is fortunately provided.

Simulation Method

While CPU of supercomputers are too expensive to use, people can always find ways to get around that problem by using simulation. In order to evaluate a program's performance, simulation techniques [FSZ79] such as emulation, Monte Carlo or discrete event simulation are widely used.

J. Bruner et al. [BCVY91] presented an approach which aims at employing the Perfect Club

benchmarks to evaluate machine performance. The tool called MaxPar determines the maximum available parallelism in a program by instrumenting each program.

Simulation requires a lot of efforts from the users. The program has to be modeled. In some way, this is exactly what we do automatically at the instruction level.

Analytical Method

The analytical method analyzes the program behavior and underlying architecture by a set of mathematical equations and/or formulas.

There are several people who are active in this field.

P. Flajolet et al. [FSZ88] presented an automatic analyzer. This analyzer conducts a middle-cost analysis on the operating algorithms on top of the decomposable database. The dynamic phase is assured by an algebraic analyzer while compiling the algorithm specifications on generic functions. The static phase — analytic analyzer extracts the asymptotical information on the generic series coefficients, i.e., on the ascending time rate. The information computed is much richer than ours, but very few program can be analyzed. Procedure calls are not supported.

V. Dornic [Dornic92] presented in his thesis three systems that use the notion of effect systems to estimate complexity. Introduced for the FX language, his framework is general and flexible. It allows additional information to be associated with program expressions (e.g. side effects in FX and time in our thesis). Their systems are conceptually new. The programming language analyzed is powerful and the systems are proved correct. However the expression power of the output language is paid for by returning very limited execution time information. Programs have either a constant execution time or an undefined execution time. Our estimator does compute symbolic polynomials based on input parameters to provide some prediction capability.

B. Stramm and F. Berman [SB92] proposed a model called *Retargetable Program-Sensitive Model* (RPS Model) which predicts the performance of static, data-independent parallel program mapped to message-passing multicomputers. The behavior of multiprocessor systems at the application level is analyzed using an analytical model. They show that their RPS model accurately (within 10%) predicts the performance of programs by comparing predicted and actual execution times.

J-Y. Berthou and P. Klein [BK92] showed that the discrepancy generated by the target machine has a non-negligible influence on the sequential as well as parallel execution times. They investigated the effects on efficiency, which depends on these execution times. They presented a set of criteria for estimating the effective performance of parallelization, while studying the machines generated drop in efficiency.

A. Yurik [Yurik93] presents *micro-analysis* approach to analyze program performance. Its goal is to construct, with the help of a computer, a symbolic formula, so-called *time-formula* which expresses a program's execution time as a function of *time variable*. Once a time-formula for a program is determined, actual time can be computed by replacing the time variable by the specific values. He claim that his approach can determine the execution time of a program without explicitly running. We believe that PIPS uses the same technique as his at the very beginning, i.e., to construct a formula to represent the execution time. But what he called *time variable* means the execution time of the basic operations, such as addition, memory access. While in PIPS, we use *cost table*

to present the basic cost. He does not deal with the constructs like test, loop at all. He shows his approach on sort algorithms while supposing the worst cases.

Measurement Method — Profiling

This class of methods is linked to benchmarking. Benchmarking is generally used to test machines. A bundle of programs from various fields can form a Benchmark test suite, such that Perfect Benchmark Club [Cybenko91]. It has become a new and very good performance prediction approach to support the program optimization effort. The corresponding performance measurements are not only used to show the machine performance, but also reflect the performance behavior of real programs.

V. Sarkar [Sarkar89A, Sarkar89B] presented a general framework for the determining average program execution and their variance, based on the program's interval structure and control dependence graph. Average execution times and variance values are computed using frequency information from an optimized counter-based execution profile of the program.

C. Kesselman [Kessel91] in his thesis investigated methods for understanding and improving the performance of parallel programs. There were two main aspects in his work: measurement and presentation. His approach was to base performance measurement on extending execution profiling to parallel programs. He developed a set of low overhead techniques into a parallel programming system, called PCN. They also developed a performance visualization tool called *Gauge*.

A. D. Malony [Malony90] has discussed several methods for performance measurement, concentrating specifically on mechanisms for timing and tracing. He shows how minor hardware and software modifications can enable better measurement tools to be built and describe results from a prototype hardware-based software monitor developed for the Intel iPSC/2 multiprocessor. He develops two models of performance perturbation to understand the effects of instrumentation intrusion: time-based and event-based. The time-based models use only measured time overheads of instrumentation to approximate actual execution time performance. He shows that the model can give accurate approximate for sequential execution and for parallel execution with independent execution ordering. He uses event-based model to quantify the perturbation effects of instrumentations of parallel executions with ordering dependencies. His results show that the model can be applied in practice to achieve accurate approximations. He also discusses the limitations of the time-based and event-based models and gives several examples where performance visualization techniques have been effectively applied. In general, these techniques are basically different from ours since they provide one measurement point for one input. However, they are useful to validate our symbolic prediction.

7.2.2 Machine Modelization

We all know that development of parallel applications is seriously hampered by insufficient knowledge of the influence of the machine architecture on parallel program performance. In order to improve application development time, there is a growing need for performance model. We have mentioned in Section 3.1 that our model uses a set of cost tables for sequential machines. We also mentioned how to deal with parallel DO loops in Chapter 6. Now we present some other ideas

about this topic.

H. Jonkers [Jonkers92] proposes a performance modeling technique. That technique seeks to incorporate both parallel processing and machine concurrency into one modeling paradigm with an emphasis on the derivation of reusable machine models, called *generic machine model* in the `ProcMod` subproject of `parTool` project. These generic models, together with the model counterpart of the specific parallel program, are input to the performance evaluation. They started with data model of parallel architecture. The data model is divided into *static* and *dynamic*. Typical example of static data are the scalar speed of a processor, the number of processors and memories and the topology of an interconnection network. Typical examples of dynamic data are the duration of a vector operation, etc. We can believe that it resembles more or less our cost table in PIPS. They only investigated the static model.

7.2.3 Classification of Performance Estimators

We classify the existing performance estimators according to the nature of the target machines: single processor machine, general multiprocessor machine (can be both shared or distributed), shared memory multiprocessor machine and distributed memory multiprocessor machine.

Single Processor Machine

K. Kennedy et al. [KMM92] developed a pure static performance estimation tool in the ParaScope programming environment. It is similar to our tool in terms of static part. But when no information is available to the estimator, wild guessing is used. For example, 50 is default loop iteration range. We do better in our estimator because preconditions are used.

C. Y. Park [Park92] addressed the problem of analyzing the timing behavior of a real-time system by predicting the deterministic execution times of the program comprising a system. Their approach is to predict a time interval, covering all execution cases, with an analytic method at the source program level. For tighter prediction they analyzed the dynamic behavior of a program using information provided by the user. They introduced a formal path model where both a program and user information are represented by a set of program paths described by an extended regular expression. Infeasible paths are eliminated by intersecting two path sets. In PIPS, we use the same technique to predict the execution time of the program, but we do not give the range, we give only one expression, probably the worst case. And we deal with the symbolic estimation, but he did not.

N. Tawbi [Tawbi90] studied the generation of efficient parallel programs in her thesis while preserving the same semantics of the initial sequential program. She requested that the parallelism in the sequential program be detected in a preliminary step. She used a list scheduling algorithm, for this, she experimented execution time tests by summing the statement execution times together while paying attention to loop iteration number.

T. Ball [BL94] present a program-based branch predictor that performs well for a large and diverse set of programs written in C and Fortran. In addition to using natural loop analysis to predict branches that control the iteration of loops, they focus on heuristics for predicting non-loop branches, which dominate the dynamic branch count of many programs. Although accuracy is

major problem, it reaches a level of usefulness.

T.A. Wagner et al. [WMGH94] present a static estimator for program optimization. They use quantitative metrics to compare estimates from static analysis to those derived from profiles, which bear resemblance with PIPS cost table. They combine function-level information with a Markov model of control flow over the call graph to produce arc and basic block frequency estimates for the entire program. They succeed in determining 76% of the most frequently executed call sites for SPEC92 Benchmark Suite.

A. Krall [Krall94] presents code replication technique that improves the accuracy of semi-static branch prediction to a level comparable to dynamic branch prediction schemes. His technique uses profiling to collect information about the correlation among different branches, and among the subsequent outcomes of a single branch. We can think that it resembles the idea in PIPS half-dynamic and half-static strategy so as to better estimate branches.

N.B. MacDonald [MacDo93] uses a micro-analysis technique to deriving estimates of the sequential execution time of code fragments written in a subset of Fortran 77. He considers the execution of each code fragment to involve the certain number of various basic operations, and predicts the execution time of the fragment using expected execution times for the basic operation, which can be derived automatically. We think that we use the same idea in PIPS to build our cost tables for different machines.

General Multiprocessor Machine

Because there are no many tools in this category, we only present one tool from E. Gabber et al. of Tel-Aviv University.

E. Gabber et al. [GAY93] developed P^3C, a fully automatic, portable parallelizing Pascal compiler for scientific code, which is characterized by loops operating on regular data structures. P^3C can compile same source code to all target machines without modification, yet it can generate parallel programs for MIMD machines. P^3C is separated into two parts: (1) parallelizer, which performs the machines-independent task of extracting parallelism and data distribution, and (2)VMMP for Virtual Machine for MultiProcessors, which hides the specific implementation details of the target machine. PIPS uses the same technique but targets the Fortran language instead of Pascal of P^3C.

P^3C can also give an accurate estimate about whether the parallel code will actually reduce execution time over serial code, taking into account the associated overhead. It derives the estimate by statically analyzing the program at compile time, referring to a table of the target machine's parameters. We believe that this table must resembles the cost table of PIPS. They have implemented P^3C on two shared-memory multiprocessors, as well as on a distributed-memory network of 8 transputers.

Shared Memory Multiprocessor Machine

W. Abu-Sufah and A.Y. Kwok [AK85] present a set of machine-specific performance prediction tools which are developed for the Cedar multiprocessor computer. It includes analytical and simulation techniques incorporating guessing for program unknows. Their approach lacks portability because

it is base on the Cedar compiler assembly code.

V. Sarkar [Sarkar89B, Sarkar89A] derives estimated runtime information for shared memory machines. He presents a general framework for determining average program execution times and their variance, based on the program's interval structure and control dependence graph. Average execution times and variance values are computed using frequency information from an optimized counter-based execution profile of the program. He uses guesses for loop ranges and tests.

K. Gallivan [GJMW89] present some techniques which involve a combination of empirical observations, architectural models and analytical techniques. They used Lawrence Livermore Loops as a test case to verify their approach.

A. Goldberg and J. Hennessy [GH89] present the MTOOL, a method to isolate memory bottlenecks in shared memory multiprocessor programs. First, they estimate the execution time using the same method as PIPS does, that is, supposing the memory access performs ideally. Then they time the difference with the real run. They observe where actual measured execution time differs from the time predicted given a perfect memory system, and then inexpensively detect the memory bottlenecks so as to isolate them.

T. E. Anderson and E. D. Lazowskaet [AL90] presented the Quartz, a tool for tuning parallel program performance on shared-memory multiprocessors. The philosophy underlying Quartz was inspired by that of the sequential UNIX tool **gprof.** The principal metric of Quartz is normalized processor time: the total processor time spent in each section of code divided by the number of other processors that are concurrently busy when that section of code is being executed. Of course, Quartz is dynamically tuning the performance. Apparently, Quartz cannot deal with symbolic variables because everything depends on runtime measurement.

D. Atapattu and D. Gannon [AG89] present an interactive tool designed for performance prediction of parallel programs. Their prediction system built as a sub-system of a larger interactive environment uses a parser, dependence analyzer, database and an X-window based front end in analyzing programs. They propose a simple analytical model as an attempt to predict performance degradation due to data references in hierarchical memory systems. The authors obtain estimated runtimes for parallel Fortran programs in order to support the application of program transformations. When there are unknowns involved, such as loop bounds, the output is an algebraic expression in terms of these variables.

M. Haghighat and C. Polychronopoulos [HP92] propose a program flow analysis framework for parallelizing compilers. Within their framework, they use symbolic analysis as an abstract interpretation technique to solve many of the flow analysis problems in a unified way. Some of these flow analysis problems are constant propagation, global forward substitution, detection of loop invariant computations and induction variable substitution. They also present a systematic method for generalized strength reduction. We believe that their symbolic analysis is like PIPS' precondition. They use their analysis to optimize the compiler while we use the precondition to do the same job, e.g., our complexity estimation is a good example.

K. Wang [Wang93, Wang94] proposes a framework to predict the static performance for superscalar processors. This performance prediction framework combines several innovative approaches. (1) The framework employs a detailed, architecture specific, but portable, cost model that can be used to estimate the cost of instruction sequence. The cost model looks like the cost table in PIPS, but contains two steps. The first one is like a machine independent assembly language. The second

one models the sequenced execution. (2) Aggregated cost of loops and conditional statements are computed and manipulated symbolically as in PIPS. (3) enable selective incremental performance update.

Distributed Memory Multiprocessor Machine

V. Balasundaram et al. [BFKK91, BFKK92] described a performance estimator to select a data distribution strategy based on estimated runtime information. The estimator statically evaluates the relative efficiency of different data partitioning schemes for any any given program on any given distributed memory multiprocessor. It is limited to programs utilizing the loosely synchronous communication model.

T. Fahringer [Fahringer93] presents in his thesis a performance prediction approach which has two major components: profiling and performance parameters. He incorporates a profile run to derive program unknown data for loop iteration counts, frequency information and true ratios. Then he develops a tool called P^3T [FBZ92, FZ93], *Parameter based Performance Prediction Tool*, which is a part of Vienna Fortran Compilation System (VFCS), a compiler that automatically translates Fortran programs into message-passing programs for massively parallel architectures. P^3T statically computes a set of optional parameters that characterize the behavior of the parallel program, including work distribution, the number of transfers, the amount of data transferred, transfer times, etc. These parameters can be selectively determined for statements, basic blocks, loops, subroutines and the whole program. But he only takes the message-passing machines into account.

B. Corwin and R. Braddock [CB92] present a strategy for using operational performance metrics to characterize and monitor distributed systems. They can capture the performance metrics during system modeling.

J. Yang et al. [YAG93] present a framework to address the issue of execution time estimation on heterogeneous supercomputer architectures. They propose the techniques to characterize applications and architectures of the machine. Based on code profiling and analytical benchmarking, these techniques provide a detailed architecture-independent application characterization.

7.3 Future Work

Although impressive results have already been obtained, many improvements are still possible, especially with regards to the current implementation. First of all, we showed that the pitfall observed on tmines in Chapter 5 was due to the use of guessed probabilities instead of unknown probabilities introduced in Chapter 2 to prove the correctness of the second approximation step. These symbolic probabilities should be introduced in the implementation by default and numerical default probabilities should only be used on user request. Unstructured and while loops could be handled with a symbolic calculus tool like Maple or Mathematica, since they have the capability to inverse matrices.

Unknown probabilities could also be guessed for efficiency if error cases were detected. For instance, it should be safe to assume that control paths leading to an exit outside of the main module are error

handling paths and that they are taken with probability 0 since users usually are not interested in failures.

For programming environments, another kind of complexity could be computed. Users are primarily interested in the relative weight of each statement with respect to the total execution time. This information is not carried by the complexity presently computed since it only estimates the time to execute a piece of code once by a bottom-up evaluation. The combination of a second top-down phase is necessary to estimate how many times each piece of code is executed and to multiply the elementary complexity by this number of times. All data structures are available, as well as the top-down interprocedural extension. A minimal programming effort would be required to add this functionality to PIPS.

Finally, lots of additional experiments are required to see better what the shortcomings of our present scheme are. In particular, we would like to see how the simple but successful model of the SPARCstation 2 could be extended to handle superscalar and superpipelined RISC architectures which are now used in every new computer.

Bibliography

[AK85] W. Abu-Sufah and A.Y. Kwok *Performance Prediction Tools for Cedar: A Multiprocessor Computer*, Proceedings of the 12th International Symposium on Computer Architecture, pp 406-413, 1985

[ASU86] A.V. Aho, R. Sethi, J.D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesly, 1986.

[AKSR93] K.K. Aggarwal, M. Pavan Kumar, Vinay Santurkar and Radha Ratnaparkhi *Towards a Weighted Operational Profile*, ACM SIGSOFT, Software Engineering Notes, Vol. 18 No.1 Jan. 1993

[AC] Vishwani D. Agarwal and Srimat T. Chakraadhar *Performance Estimation in a Massively Parallel System*, 306-313

[ABGS91] Carme Alvarez, Jose L. Balcazar, Joaquim Gabarro and Miklos Santha *Parallel Complexity in the Design and Analysis of Concurrent Systems*, PARLE'91 Parallel Architectures & Languages Europe Proceedings Vol 1

[AL90] Thomas E. Anderson and Edward D. Lazowska *Quartz: A Tool for Tuning Parallel Program Performance* Department of Computer Science and Engineering, University of Washington Proceedings of the 1990 SIGMETRICS Conference on Measurement & Modeling of Computer Systems Boulder, Colorado, USA. May 22-25, 1990, pp 115-125.

[AG89] Daya Atapattu and Dennis Gannon *Building Analytical Models into an Interactive Performance Tool* Department of Computer Science, Indiana University at Bloomington Proceedings of Supercomputing 1989, Reno, Nevada, USA. November 13-17, 1989, pp 521-530

[BBNInside] *Inside the TC2000 Computer*, Revision 1.0 BBN Advance Computers Inc. 1990

[BBNXtra] *Using the Xtra Programming Environment*, Revision 2.0 BBN Advance Computers Inc. 1990

[BBNC] *Uniform System Programming in C*, Revision 2.0 BBN Advance Computers Inc. March 1990

[Baase88] Sara Baase *Computer Algorithms — Introduction to Design and Analysis* Second Edition, Addison-Wesley Publishing Company

[BL94] Thomas Ball and James R. Larus *Branch prediction for free* Computer Science Department, University of Wisconsin-Madison. Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation, pp 300-313. Albuquerque, New Mexico.

[BFKK91] Vasanth Balasundaram, Geoffrey Fox, Ken Kennedy and Ulrich Kremer *A Static Performance Estimator to Guide Data Partitioning Decisions* pp 213-223, Third ACM Sigplan Symposium on Principles and Practice of Parallel Programming (PPoPP), Williamsburg, VA. April 21-24 1991.

[BFKK92] Vasanth Balasundaram, Geoffrey Fox, Ken Kennedy and Ulrich Kremer *A Static Performance Estimator in the Fortran D Programming System* Language, Compilers and Run-Time Environments for Distributed Memory Machines, 1992. pp 119-138

[BZ93] David A. Barrett and Benjamin G. Zorn *Using Lifetime Predictors to Improve Memory Allocation Performance* Department of Computer Science, University of Colorado at Boulder. SIGPLAN'93 Conference on Programming Language Design and Implementation, Albuquerque, NM, June 23-25, 1993. pp 187-196.

[Baron90] Bruno Baron *PIPS User Guide Version 1.0*, Ecole des Mines de Paris, EMP-CRI/A Sep. 1990

[Ber66] A.J. Berstein, *Analysis of programs for parallel processing*, IEEE Transactions on Electronic Computers, Vol.15, No 5, OCT66.

[BK92] Jean-Yves Berthou and Philippe Klein *Estimating the Effective Performance of Program Parallelization on Shared memory MIMD Multiprocessors*, Laboratoire de Méthodologie et Architecture des Systèmes Informatiques MSAI 92.53 Sep. 1992.

[BP] Carl J. Beckman and Constantine D. Polychronopoulos *The Effect of Scheduling and Synchronization Overhead on Parallel Loop Performance*, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign.

[Bert90] P. Berthomier *Static Comparison of Different Program Versions*, DEA systèmes Informatiques - Université PARIS VI, Parallélisation et Vectorisation Automatiques. Sep. 1990 Document EMP-CAI-I 130

[BKV91] J. Bentley, B. Kernighan and C. Van Wyk *An Elementary C Cost Model* UNIX Review, Feb. 1991.

[BL] David K. Bradley and Jone L. Larson *Fine-Grained Measurements of Loop Performance on the CRAY Y-MP* Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign.

[BCVY91] John Bruner, Hoichi Cheong, Alexander Veidenbaum and Pen-Chung Yew *Chief: A Parallel Simulation Environment for Parallel Systems* Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign. CSRD Report No. 1050 5th International Parallel Processing Symposium, Anaheim, CA, April 30 - May 2, 1991.

[CZ91] Edwin K.P. Chong and Wei Zhao *Performance Evaluation of Scheduling Algorithms for Imprecise Computer Systems* The Journal if Systems, Vol 15, No.3 July 1991.

[CP91] Doreen Y. Chong and Douglas M. Pase *An Evaluation of Automatic and Interactive Parallel* Programming Tools Supercomputing'91, pp 412-423.

[Collins91] William J. Collins *Estimating Execution Times: A Laboratory Exercise for CS2* Computer Science Department, Lafayette College. ACM SIGCSE Bulletin, Vol 23, No. 1, March 1991.

[CH90] Thomas M. Conte and Wen-mei W. Hwu *Benchmark Characterization for Experimental System Evaluation* Center for Reliable and High-Performance Computing, University of Illinois. Proceedings of the 23th Annual Hawai International Conference on System Sciences 1990 Vol. 1

[CB92] Bert N. Corwin and Robert L. Braddock *Operational Performance Metrics in a Distributed System: Part I: Strategy* Proceeding of the 1992 ACM SIGAPP Symposium on Applied Computing, pp 867-873.

[Cybenko91] G. Cybenko *Supercomputer Performances Trends and the Perfect Benchmark*, Supercomputing Review, April 1991

[Dornic92] Vincent Dornic *Analyse de Complexité des Programmes : Vérification et Inférence* Ph.D Dissertation. 1992 Rapport CRI/A/212 Ecole des Mines de Paris

[Dowd93] Kevin Dowd *High Performance Computing* O'Reilly & Associates, Inc. 1993.

[DMKJ91] M.D. Durand, T. Montaut, L. Kervella and W. Jalby *Modeling the Impact of Memory Contention on Dynamic Scheduling* Submitted to IEEE Transaction on Parallel and Distributed Systems.

[DMKJ93] M.D.Durand, T. Montaut, L.Kervella and W.Jalby *Impact of Memory Contention on Dynamic Scheduling on NUMA Multiprocessors* Proceedings of the 1993 International Conference on Parallel Processing August 16-20, 1993.

[Emad91] Nahid Emad *Détection de Parallélism et Production de Programmes Parallèles* CRI report Juin 1991. Document EMP-CRI-E/147.

[FBZ92] Thomas Fahringer, Roman Blasko and Hans P. Zima *Automatic Performance Prediction to Support Parallelization of Fortran Programs for Massively Parallel Systems* Proceedings of the 1992 International Conference on Supercomputing 1992, pp 347-356.

[FZ93] Thomas Fahringer and Hans P. Zima *A Static Parameter based Performance Prediction Tool for Parallel Programs* Department of Computer Science, University of Vienna, Austrian Center for Parallel Computation, Technical Report Series ACPC/TR 93-1 January 1993.

[Fahringer93] Thomas Fahringer *Automatic Performance Prediction for Parallel Programs on Massively Parallel Computers* Ph.D Dissertation, Vienne, Austria. Sep. 1993.

[FSZ79] Domaenico Ferrari, Giuseppe Serazzi, Alessandro Zeigner *Computer System Performance Evaluation* Prentice-Hall Inc. 1979.

[FSZ83] Domaenico Ferrari, Giuseppe Serazzi and Alessandro Zeigner *Measurement and Tuning of Computer Systems* Prentice-Hall Inc. 1983.

[FSZ88] P. Flajolet, B. Salby and P. Zimmermann *Lambda-Upsilon-Omerga: An Asistant Lagorithm Analyzer* Rapport de recherche INRIA 876 and Proceedings of AECC'6, Lectures Notes in Computer Science, July 1988.

[Flynn72] M.J. Flynn *Some Computer Organizations and their Effectiveness* IEEE Trans. Computers, C-21, pp 948-960, 1972.

[GAY93] Eran Gabber, Amir Averbuch and Amiram Yehudai *Portable, Parallelizing Pascal Compiler* Tel-Aviv University, Israel. IEEE Software, March 1993, pp 71-81.

[GJMW89] Kyle Gallivan, William Jalby, Allen Malony and Harry Wijshoff *Performance Prediction of Loop Constructs on Multiprocessor Hierarchical-Memory Systems* Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign International Conference on Supercomputing 1989. Pages 433-42 June 5-9, 1989. Crete, Greece.

[Gemund93] Arjan J.C. van Gemund *Compile-time Performance Prediction with PAMELA* Faculty of Electrical Engineering, Delft University of Technology. Proceedings of 4th International Workshop on Compilers for Parallel Computers. pp 428-435. 13-16 Dec. 1993. Delft University of Technology, The Netherlands.

[GH89] Aaron Goldberg and John Hennessy *MTOOL: A Method for Isolating Memory Bottlenecks in Shared Memory Multiprocessor Programs* Computer System Laboratory, Stanford University Proceedings of the 1989 International Conference on Supercomputing Crete, Greece. June 5-9, 1989. pp 433-442.

[Gordon79] Michael J.C. Gordon *An Introduction to the Denotational Description of Programming Languages* Springer-Verlag 1979

[Gordon88] Michael J.C. Gordon *Programming Language Theory and its Implementation* Prentice Hall, 1988

[GS92] Michael M. Gutzmann and Klaus Steffan *PEPSIM-ST: A Simulator Tool for Benchmarking* CONPAR'92 / VAPP V, Sep.1-4, 1992. Lyon France. Institut für Mathematische Maschinen und Datenverarbeitung VII, Universität Erlangen-Nürnberg, Martensstr. 3, D-8520 Erlangen, Germany.

[HP92] M. Haghighat and C. Polychronopoulos *Symbolic Program Analysis and Optimization for Parallelizing Compilers* University of Illinois at Urbana-Champaign. Proceedings of 5th International Workshop on Languages and Compilers for Parallel Computing, New Haven, Connecticut, USA, August 1992. pp 538-562.

[HC91] Williams Ludwell Harrison III and Jyh-Herng Chow *Dynamic Control of Parallelism and Granularity in Executing Nested Parallel Loops* CSRD Report Np.1167 The Third IEEE Symposium on Parallel and Distributed Processing, Dallas, Texas Dec. 1991

[HK90] Jukka Helin and Kimmo Kaski *Performance Analysis of High-Speed Computers in Scientific/Engineering environment* Tampere University of Technology, Finland Proceedings of the 23th Annual Hawai International Conference on System Sciences 1990 Vol. 1

[Hill] Mark D. Hill *What is Scalability?* Computer Science Department, University of Wisconsin at Madison 18-20

[HKT92] Seema Hiranandani, Ken Kennedy and Chau-Wen Tseng *Evaluation of Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines* International Conference on Supercomputing 1992.

[Hochney91] Roger Hochney *Performance Parameters and Benchmarking of Supercomputers* Computer Science Department, Reading University, UK Parallel Computing, Vol.17 No. 10 & 11, Dec. 1991 1111-1130

[HIM91] Jeffrey K. Hollingsworth, R.B. Irvin and B. P. Miller *The Integration of Application and System Based Metrics in a Parallel Program Performance Tool* Computer Science Department, University of Wisconsin-Madison Proceedings of the 1990 ACM SIGPLAN Symposium on Principals and Practice of Parallel Programming Williamsburg, Virginia, USA. April 21-24, 1991, pp 189-200.

[HG93] Seongsoo Hong and Richard Gerber *Compiling Real-Time Programs into Schdulable Code* University of Maryland. SIGPLAN'93 Conference on Programming Language Design and Implementation, Albuquerque, NM, June 23-25, 1993. pp 166-176.

[IAM87] I. Ion, R. Arhire, M. Macesann, *Program complexity: comparative analysis, hierarchy, classification*, SIGPLAN NOTICES, Vol.22, No 4, AVR87.

[IJ91] François Irigoin and Pierre Jouvelot *PROJET PIPS: Manuel Utilisateur du Paralléliseur (version 2.3)* Centre d'Automatique et d'Informatique - Section Informatique: Rapport interne, 1991. Document EMP-CAI-I E144

[IJT91] François Irigoin, Pierre Jouvelot and Rémi Triolet, *PIPS overview*, ICS'91 Paris, France

[IM93] R. Bruce Irvin and Barton P. Miller *Multi-Application Support in a Parallel Program Performance Tool*, Computer Science Department, University of Wisconsin-Madison

[Jonkers92] Henk Jonkers *Data Modelling of Parallel Computer Architectures*, Delft University of Technology, Faculty of Electrical Engineering, Lab of Computer Architecture and Digital Techniques. Report No. 1-68340-44(1992)02, Febuary, 1992.

[JT89] P. Jouvelot, R. Triolet, *NewGen: A Language-Independent Program Generator*, Rapport interne EMP-CAI-I A/191, Mines de Paris, JUL89.

[KME92] A. Kapelnikov, R.R. Muntz and M.D. Ercegovac *A Methodology for Performance Analysis of Parallel Computations with Looping Constructs*, Journal of Parallel and Distributed Computing 14, 105-120 1992.

[KG91] H. Kellerer and Graz *Bounds for Nonpreemptive Scheduling of Jobs with Similar Processing Times on Multiprocessor Systems Using the LPT-Algorithm*, Computing 46,183-191, 1991

[KMM92] Ken Kennedy, Nathaniel McIntosh and Kathryn S. McKinley *Static Performance Estimation in a Parallelizing Compiler*, Department of Computer Science, Rice University

[Kessel91] Carl Kesselman *Tools and Techniques for Performance Measurement and Performance Improvement in Parallel Programs*, June 1991, Ph.D Thesis, University of California at Los Angeles, CSD-910015

[KR88] Brian W. Kernighan and Dennis M. Ritchie *The C Programming Language*, Second edition 1988 Prentice Hall

[Knuth73] D.E. Knuth. *The Art of Programming*, Vol 1, Addison-Wesley, 1973.

[Krall94] Andreas Krall *Improving Semi-static Branch Prediction by Code Replication* Institut für Computersprachen, Technische Universität Wien. ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Pages 97-106, Orlando, Florida, June 20-24, 1994.

[KG91] Vipin Kumar and Anshul Gupta *Analyzing Scalability of Parallel Algorithms and Architectures*, Department of Computer Science, University of Minnesota Tech Report 91-18, May 1991.

[LB] James R. Larus and Thomas Ball *Rewriting Executable Files to Measure Program Behavior*, Computer Science Department, University of Wisconsin-Madison.

[LS90] Calvin Lin, Lawrence Snyder *A Comparison of Programming Models for Shared Memory Multiprocessors*, Proceedings of 1990 Int. Conf. on Parallel Processing. August13-17, 1990.

[MacDo93] Neil B. MacDonald *Predicting the Execution Time of Sequential Scientific Code*, Proceedings of International Workshop on Automatic Distributed Memory Parallelization 1993, Automatic Data Distribution and Parallel Performance Prediction, Universität des Saarlandes, Saarbrücken, Germany. March 1993.

[Malony90] Allen Davis Malony *Performance Observability*, Ph.D Dissertation. Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign CRSD Rpt. No. 1034 Dissertation, September 1990.

[MLR91] Allen Davis Malony, John L. Larson and Daniel A. Reed *Tracing Application Program Execution on the CRAY X-MP and CRAY-2*, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign The Journal of Supercomputing, 5, 137-162. 1991.

[MAD91] William Mangione-Smith, Santosh G. Abraham and Edward S. Davidson *Architectural vs. Delivered Performance of the IBM RS/6000 and the Astronautics ZS-1*, Proceedings of the 24th Annual Hawai International Conference on Systems 1991.

[Martin88] Joanne L. Martin *Performance Evaluation of Supercomputers*, IBM Corporation T.J. Watson Research Center and Data Systems Division, Yorktown Heights, NY 10598.

[MHL91] Dror E. Maydan, John L. Hennessy and Monica S. Lam *Efficient and Exact Data Dependence Analysis*, Computer Systems laboratory, Stanford University ACM SIGPLAN Notices, Vol. 26 No. 6, June 1991.

[McGeoch92] Catherine McGeoch *Analyzing Algorithms by Simulation: Variance Reduction Techeniques and Simulation Speedups*, Department of Mathematics and Computer Science, Amherst College. ACM Computing Surveys. Vol. 24, No. 2 June 1992.

[MA] Daniel Menasce and Virgilio Almeida *Cost-Performance Analysis of Heterogeneity in Supercomputer Architectures*, Departamento de Informatica Pontificia Universidade Catolica, Brasil

[Monro82] Donald M. Monro *Fortran 77*, Imperial College of Science and Technology, University of London 1982 Edward Arnold (Publishing) Ltd.

[Patt90] Yale N. Patt *Methodologies for Experimental Research in Computer Architecture and Performance Measurement* Department of Electrical Engineering and Computer Science, University of Michigan Proceedings of the 23th Annual Hawai International Conference on System Sciences 1990 Vol. 1

[Park92] Chang Yun Park *Predicting Deterministic Execution Times of Real-Time Programs* Ph.D Dissertation. Department of Computer Science and Engineering, University of Washington, Seattle Technical Report 92-08-02

[Petersen93] Paul Marx Petersen *Evaluation of Program and Parallelizing Compilers Using Dynamic Analysis Techniques* Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign. CSRD Report No. 1273 January 1993.

[Port89] A.K. Porterfield, *Software Methods for Improvement of Cache Performance on Supercomputer Applications*, Rice University, Computer Science Technical Report, Rice COMP TR89-93, MAY89.

[PFTV86] William H. Press, Brian P. Flannery, Saul A. Teukolsky and William T. Vetterling *Numerical Recipes — The Art of Scientific Computing* Cambridge University Press, 1986.

[Pugh94] William Pugh *Counting Solutions to Presburger Formulas: How and Why*, Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, College Park. ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Pages 121-134, Orlando, Florida, June 20-24, 1994.

[PW92] William Pugh and David Wonnacott *Static Analysis of Upper Bounds on Parallelism*, Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, College Park. Technical Report UMIACS-TR-92-125, CS-TR-2994, Nov. 1992.

[SUNF88] Sun Microsystems *Sun FORTRAN Programmer's Guide* Revision A of 6 May 1988.

[SUNAS3] Sun Microsystems *Sun-3 Assembly Language Reference Manual* Revision A of 27 March, 1990.

[SUNAS4] Sun Microsystems *Sun-4 Assembly Language Reference Manual* Revision A of 27 March, 1990.

[Schm86] David A. Schmidt *Denotational Semantics – A Methodology for Language Development* Kansas State University, Allyn and Bacon, Inc. 1986.

[SB92] Bernd Stramm and Francine Berman *Predicting the Performance of Large Programs on Scalable Multicomputers* University of California at San Diego Scalable High Performance Computing Conference Williamsburg, Virginia, USA. April 26-29, 1992, pp 22-29.

[Sarkar89A] Vivek Sarkar *Determining Average Program Execution Times and Their Variance*, Proceedings of the SIGPLAN 89 Conference on Programming Language Design and Implementation. Portland, July 1989. pp 298-312.

[Sarkar89B] Vivek Sarkar *Partitioning and Scheduling Parallel Programs for Multiprocessors*, Ph.D Dissertation. 1989. IBM Thomas J. Watson Research Center

[Spiegel74] M.R. Spiegel *Formules et Tables de Mathématiques*, McGraw Hill Paris, Série Schaum, 1974

[Stone87] Harold S. Stone *High-Performance Computer Architecture*, IBM Thomas J. Watson Research Center and Courant Institute, New York University

[SG91] Xian-he Sun and John L. Gustafson *Toward a Better parallel Performance Metric*, Parallel Computing, Vol.17 No. 10 & 11, Dec. 1991 1093-1109

[WMGH94] Tim A. Wagner, Vance Maverick, Susan L. Graham and Micheal A. Harrison *Accurate Static Estimation for Program Optimization*, Computer Science Division, University of California, Berkeley. ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Pages 85-96, Orlando, Florida, June 20-24, 1994.

[Wilf86] Herbert S. Wilf *Algorithms and Complexity*, Thèse à l'Université de Nice–Sophia Antipolis, 1992.

[Syska92] Michel Syska *Communications dans les architectures à mémoire distribuée*, Thèse à l'Université de Nice–Sophia Antipolis, 1992.

[Tawbi90] Nadia Tawbi, *Parallélisation Automatique : Estimation des Durées d'Exécution et Allocation Statique de Processeurs*, Thèse à l'uNiversité Paris VI. 1990.

[TC88] R.H. Thomas and W. Crowther *The Uniform System: A approach to runtime support for large scale shared memory parallel processors*, ICPP'88

[TI90] Rémi Triolet and François Irigoin *PIPS High-Level Environment*, Ecole des Mines de Paris, August 1990.

[WLSH91] Mingfang Wang, Ben Lee, Behrooz Shirazi and A.R. Hurson *Accurate Communication Cost Estimation in Static Task Scheduling* , Proceedings of the 24th Annual Hawai International Conference on Systems, 1991.

[Wang93] Ko-Yang Wang *A Framework for Static, Precise Performance Prediction for Superscalar-based Parallel Computers* IBM TJ Watson Research Center, PO Box 704, Yorktown Heights, NY 10598, USA. Proceedings of 4th International Workshop on Compilers for Parallel Computers. pp 413-427 13-16 Dec. 1993. Delft University of Technology, The Netherlands.

[Wang94] Ko-Yang Wang *Precise Compile-Time Performance Prediction for Superscalar-Based Computers* IBM TJ Watson Research Center, PO Box 704, Yorktown Heights, NY 10598, USA. ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Pages 73-84, Orlando, Florida, June 20-24, 1994.

[Wijs89] Harry A.G. Wijshoff *Data Organization in Parallel Computers*, Kluwer Academic Publishers, 1989.

[Worlton91] Jack Worlton *Toward a Taxonomy of Performance Metrics*, Parallel Computing, Vol.17 No. 10 & 11, Dec. 1991 1073-1092

[YAG93] Jaehyung Yang, Ishfaq Ahmad and Arif Ghafoor *Estimation of Execution Times on Heterogeneous Supercomputer*, Proceeding of the 1993 International Conference on Parallel Processing, 16-20 August, 1993. P219-226

[Yurik93] Aline Yurik *Micro-Analysis of Sequential and Parallel Programs*, Ph.D dissertation, Nov. 1993. Michtom School of Computer Science and Center for Complex Systems, Brandais University.

[ZC91] H. Zima and B. Chapman *Supercompilers for Parallel and Vector Computers* ACM Press, Frontier Series 1991.

[Zhou91] Lei Zhou, *Enhanced Static Evaluation of Fortran Programs in PIPS Environment* Report EMP-CRI E/160/CRI, Dec. 1991.

[Zhou92] Lei Zhou, *Complexity Estimation in the PIPS Programming Environment* CONPAR'92 / VAPP V, Sep.1-4, 1992. Lyon France.

[Zhou93] Lei Zhou, *First Experience on Shared-Memory Multi-processor MIMD Machine — BBN TC2000* Report EMP-CRI E/176/CRI, Sep. 1993.

[ZI92] Lei Zhou and François Irigoin *Properties — Low Level Tuning of PIPS* Internal Report EMP-CRI, Ecole des Mines de Paris, 1992.

Appendix A

Intermediate Representation in PIPS

```
-- External domains
-- ----------------
external Psysteme ;
external Pvecteur ;

-- Domains
-- -------
action = read:unit + write:unit ;
approximation = may:unit + must:unit ;
area = size:int x layout:entity* ;
basic = int:int + float:int + logical:int + overloaded:unit + complex:int + string:value ;
call = function:entity x arguments:expression* ;
callees = callees:string* ;
code = declarations:entity* x decls_text:string ;
constant = int + litteral:unit ;
control = statement x predecessors:control* x successors:control* ;
dimension = lower:expression x upper:expression ;
effect =  persistant reference x action x approximation x context:transformer ;
effects =  effects:effect* ;
execution = sequential:unit + parallel:unit ;
expression = syntax x normalized ;
formal = function:entity x offset:int ;
functional = parameters:parameter* x result:type ;
instruction = block:statement* + test + loop + goto:statement + call + unstructured ;
loop = index:entity x range x body:statement x label:entity x execution x locals:entity* ;
mode = value:unit + reference:unit ;
normalized = linear:Pvecteur + complex:unit ;
parameter = type x mode ;
predicate = system:Psysteme ;
ram = function:entity x section:entity x offset:int x shared:entity* ;
range = lower:expression x upper:expression x increment:expression ;
reference = variable:entity x indices:expression* ;
statement = label:entity x number:int x ordering:int x comments:string x instruction ;
storage = return:entity + ram + formal + rom:unit ;
symbolic = expression x constant ;
syntax = reference + range + call ;
tabulated entity = name:string x type x storage x initial:value ;
test = condition:expression x true:statement x false:statement ;
transformer = arguments:entity* x relation:predicate ;
type = statement:unit + area + variable + functional + unknown:unit + void:unit ;
unstructured = control x exit:control ;
value = code + symbolic + constant + intrinsic:unit + unknown:unit ;
variable = basic x dimensions:dimension* ;
```

Appendix B

Elementary C Costs on SUN SPARCstation 2

Operation	Clicks for each trial					Mics/N
Null Loop (n=1000000)						
{}	26	26	26	25	27	0.43
Int Operations (n=1000000)						
i2	26	26	26	26	26	0.00
i1++	36	37	37	36	37	0.18
i1 += 1	37	36	36	37	36	0.17
i1 = i1 + 1	37	36	37	37	37	0.18
i1 = i2	33	34	33	34	33	0.12
i1 = 1	32	32	33	32	32	0.10
i1 = i2 + 1	37	36	37	37	36	0.18
i1 = i2 + i3	39	40	40	40	39	0.23
i1 = i2 + i3 + 1	41	41	42	41	41	0.25
i1 = i2 + i3 + i4	45	46	46	45	46	0.33
i1 = i2 + i3 + i4 + 1	47	47	47	47	46	0.35
i1 = i2 + i3 + i4 + i5	52	52	51	52	52	0.43
i1 = i2 + i3 + i4 + i5 + i6	58	59	58	58	58	0.54
i1 = i2 + i3 + i4 + i5 + i6+i7	64	65	65	64	65	0.64
i1 = i2 - i3	39	40	40	39	40	0.23
i1 = i2 * i3	73	73	73	73	74	0.79
i1 = i2 / i3	116	116	116	115	116	1.50
i1 = i2 % i3	117	115	116	115	117	1.50
Float Operations (n=1000000)						
f1 = f2	33	33	33	34	33	0.12
f1 = f2 + f3	60	59	60	59	60	0.56
f1 = f2 + f3 + f4	71	72	71	73	71	0.76
f1 = f2 + f3 + f4 + f5	84	84	84	83	84	0.96
f1 = f2 + f3 + f4 + f5 + f6	96	96	96	96	96	1.17
f1 = f2 - f3	59	60	60	60	59	0.56
f1 = f2 * f3	62	62	63	62	63	0.61
f1 = f2 / f3	93	93	92	93	93	1.11
Numeric Operations (n=1000000)						
i1 = f1	41	41	40	41	41	0.25
f1 = i1	49	49	49	49	48	0.38

```
Integer Vector Operations (n=1000000)
  v[i] = i                       52   53   53   53   52      0.44
  v[v[i]] = i                    59   59   60   59   58      0.55
  v[v[v[i]]] = i                 71   69   70  398   70      1.83
Control Structure (n=1000000)
   if ( i == 5 ) i1++            35   35   34   35   36      0.15
   if ( i != 5 ) i1++            45   46   46   46   47      0.33
   while ( i < 0 ) i1++          35   35   35   35   34      0.15
  i1 = sum1(i1)                  59   57   58   59   57      0.53
  i1 = sum2(i2, i3)              71   72   72   78   75      0.79
  i1 = sum3(i2, i3, i4)          90   90   93   88   89      1.07
Input/Output (n=10000)
  fputs(s, fp)                    3    6    5    5    5      7.57?
  fgets(s, 9, fp)                 4    4    4    3    3      5.57
  fprintf(fp,sdn, i)             18   20   17   19   18     30.23
  fscanf(fp, sd, &i1)            26   27   26   26   24     42.57
Malloc (n=10000)
  free(malloc(8))                 8    7    7    8    8     12.23
  push(i)                         4    4    4    4    3      5.90
  i1 = pop()                      2    1    1    2    1      1.90?
String Functions (n=100000)
  strcpy(s, s0123456789)         17   17   17   16   17      2.37
  i1 = strcmp(s, s)              16   18   17   17   17      2.40
  i1 = strcmp(s, sa123456789)    10   11   11   10   11      1.33
String/Number Conversion (n=10000)
  i1 = atoi(s12345)               3    2    2    2    1      2.90
  sscanf(s12345, sd, &i1)        27   26   27   28   27     44.57
  sprintf(s, sd, i)              16   18   18   17   16     27.90
  f1 = atof(s123_45)            159  157  157  157  157    261.90
  sscanf(s123_45, sf, &f1)      141  141  142  142  142    235.57
  sprintf(s, sf62, 123.45)      143  142  141  140  141    235.23
Math Functions (n=10000)
  i1 = rand()                     4    2    3    2    3      4.23
  f1 = log(f2)                    2    3    3    3    3      4.23
  f1 = exp(f2)                    3    3    3    4    3      4.90
  f1 = sin(f2)                    5    4    5    4    5      7.23
  f1 = sqrt(f2)                   4    5    5    5    5      7.57
```

The first line shows that each execution of the null loop required 25 to 27 clicks, or slightly less than half a second; the average cost of a single iteration of the null loop was 0.43 microseconds. That time is subtracted from all subsequent times. The program prints a question mark after the mean if the range of values is greater than some fixed multiple of the mean value. The listing contains several question marks next to dubious values, i.e., fputs and pop, while in others there are too few clicks for statistical accuracy. The question mark points out spurious values, but be sure to study the raw clicks before placing too much faith in the summary values.

Appendix C

Calcg of Tmines

C.1 Original CALCG Subroutine

```
      subroutine calcg(im,jm,km,phi,q,r,t,b,a)
c
      dimension phi(*),t(*),q(*),r(*),b(*),a(*)
c
      common/calcgx/eps,nmax,resnl
      common/resulx/imp,niter,numtot
c
c
c
      rtn=1.
      ro=0.
      num=0
      npmax=im*jm*km
      do 5 l=1,im*jm*km
      t(l)=0.
5     q(l)=0.
c
      do 1 nitl=1,nmax
c
      num=num+1
      numtot=numtot+1
c
      do 21 l=1,im*jm*km
      r(l)=r(l)-ro*t(l)
21    t(l)=r(l)
c
      call des(im,jm,km,t,a)
      call rep(im,jm,km,t,a)
c
      rt=0.
      do 30 l=1,im*jm*km
30    rt=rt+r(l)*t(l)
      rtd=rtn
```

```
      rtn=rt
      dq=rtn/rtd
      if(num.ne.1) go to 3
      rt0=rt*eps*eps
      if(resnl.lt.sqrt(rt/float(npmax))) go to 3
      resnl=-1.
      return
3     rts=sqrt(rt/npmax)
      write(6,1000) num,rts
1000  format(i6,e12.4)
c
      do 50 l=1,im*jm*km
      q(l)=t(l)+dq*q(l)
50    t(l)=0.
c
      call prod(im,jm,km,q,t,b)
c
      qt=0.
      do 41 l=1,im*jm*km
41    qt=qt+q(l)*t(l)
      ro=rt/qt
c
      do 51 l=1,im*jm*km
51    phi(l)=phi(l)-ro*q(l)
c
      if(rt.lt.rt0) return
1     continue
c
      return
      end
```

C.2 Complexity CALCG Subroutine with Statement Ordering

```
C                   63.38*IM.JM.KM - 22.15*IM.KM - 16.62*JM.KM + 3.38 (SUMMARY)
      SUBROUTINE CALCG(IM,JM,KM,PHI,Q,R,T,B,A)
C
      DIMENSION PHI(*),T(*),Q(*),R(*),B(*),A(*)
C
      COMMON/CALCGX/EPS,NMAX,RESNL
      COMMON/RESULX/IMP,NITER,NUMTOT
C
C
C
C (0,1)
C                                                                 0 (STMT)
C (1,2)
      RTN = 1.                                                       0001
C                                                                 0 (STMT)
C (1,3)
      RO = 0.                                                        0002
C                                                                 0 (STMT)
C (1,4)
      NUM = 0                                                        0003
C                                                                 0 (STMT)
C (1,5)
      NPMAX = IM*JM*KM                                               0004
C                                                                 0 (DO)
C (1,6)
      DO 5 L = 1, IM*JM*KM                                           0006
C                                                                 0 (STMT)
C (1,8)
         T(L) = 0.                                                   0007
C                                                                 0 (STMT)
C (1,9)
5        Q(L) = 0.                                                   0008
C                                                                 0 (STMT)
C (1,10)
      NITL = 1
99997 CONTINUE
C                                                                 0 (TEST)
C (2,1)
      IF (NITL.LE.NMAX) THEN
      ELSE
         GOTO 99995
      ENDIF
C                                                                 0 (STMT)
C
C (3,2)
      NUM = NUM+1                                                    0011
C                                                                 0 (STMT)
C (3,3)
      NUMTOT = NUMTOT+1                                              0012
C                                                        2*IM.JM.KM (DO)
C
```

```
C (3,4)
      DO 21 L = 1, IM*JM*KM                                         0014
C                                                                2 (STMT)
C (3,6)
         R(L) = R(L)-RO*T(L)                                        0015
C                                                                0 (STMT)
C (3,7)
21       T(L) = R(L)                                                0016
C                                         27*IM.JM.KM - 18*IM.KM (STMT)
C
C (3,8)
      CALL DES(IM, JM, KM, T, A)                                    0017
C                                         27*IM.JM.KM - 18*IM.KM (STMT)
C (3,9)
      CALL REP(IM, JM, KM, T, A)                                    0018
C                                                                0 (STMT)
C
C (3,10)
      RT = 0.                                                       0019
C                                                     2*IM.JM.KM (DO)
C (3,11)
      DO 30 L = 1, IM*JM*KM                                         0021
C                                                                2 (STMT)
C (3,12)
30       RT = RT+R(L)*T(L)                                          0022
C                                                                0 (STMT)
C (3,13)
      RTD = RTN                                                     0023
C                                                                0 (STMT)
C (3,14)
      RTN = RT                                                      0024
C                                                                1 (STMT)
C (3,15)
      DQ = RTN/RTD                                                  0025
C                                                                0 (TEST)
C (4,1)
      IF (NUM.NE.1) THEN                                            0031
      ELSE
         GOTO 99992
      ENDIF
C                                                                0 (STMT)
C (5,1)
99991 CONTINUE
C                                                                0 (STMT)
C (6,2)
C                                                                2 (STMT)
C (6,3)
3     RTS = SQRT(RT/NPMAX)                                          0026
C                                                                0 (STMT)
C (6,4)
      WRITE (FMT=1000,UNIT=6) NUM,RTS                               0041
C                                                                0 (STMT)
C (6,5)
```

```
1000  FORMAT(I6,E12.4)                                                      0042
C                                                          2*IM.JM.KM (DO)
C
C (6,6)
      DO 50 L = 1, IM*JM*KM                                                 0044
C                                                                    2 (STMT)
C (6,8)
         Q(L) = T(L)+DQ*Q(L)                                                0045
C                                                                    0 (STMT)
C (6,9)
50       T(L) = 0.                                                          0046
C                                                54*IM.JM.KM - 36*JM.KM (STMT)
C
C (6,10)
      CALL PROD(IM, JM, KM, Q, T, B)                                        0047
C                                                                    0 (STMT)
C
C (6,11)
      QT = 0.                                                               0048
C                                                          2*IM.JM.KM (DO)
C (6,12)
      DO 41 L = 1, IM*JM*KM                                                 0050
C                                                                    2 (STMT)
C (6,13)
41       QT = QT+Q(L)*T(L)                                                  0051
C                                                                    1 (STMT)
C (6,14)
      RO = RT/QT                                                            0052
C                                                          2*IM.JM.KM (DO)
C
C (6,15)
      DO 51 L = 1, IM*JM*KM                                                 0054
C                                                                    2 (STMT)
C (6,16)
51       PHI(L) = PHI(L)-RO*Q(L)                                            0055
C                                                                    0 (TEST)
C
C (7,1)
      IF (RT.LT.RT0) THEN                                                   0060
      ELSE
         GOTO 99988
      ENDIF
C                                                                    0 (STMT)
C (8,1)
      GOTO 99987
99988 CONTINUE
C                                                                    0 (STMT)
C (24,1)
99986 CONTINUE
C                                                                    0 (STMT)
C (25,1)
C                                                                    0 (STMT)
C (26,2)
```

```
C                                                                    0 (STMT)
C (26,3)
1     CONTINUE                                                          0061
C                                                                    0 (STMT)
C (27,1)
C                                                                    0 (STMT)
C (28,1)
      NITL = NITL+1
      GOTO 99997
99992 CONTINUE
C                                                                    0 (STMT)
C (19,1)
99982 CONTINUE
C                                                                    0 (STMT)
C (18,1)
C                                                                    0 (STMT)
C (17,2)
C                                                                    2 (STMT)
C (17,3)
      RTO = RT*EPS*EPS                                                  0032
C                                                                 0.62 (TEST)
C (15,1)
      IF (RESNL.LT.SQRT(RT/FLOAT(NPMAX))) THEN                          0037
      ELSE
         GOTO 99978
      ENDIF
C                                                                    0 (STMT)
C (16,1)
      GOTO 99991
99978 CONTINUE
C                                                                    0 (STMT)
C (14,1)
99977 CONTINUE
C                                                                    0 (STMT)
C (13,1)
C                                                                    0 (STMT)
C (12,2)
C                                                                    1 (STMT)
C (12,3)
      RESNL = -1.                                                       0038
C                                                                    0 (STMT)
C (12,4)
      GOTO 99987
99995 CONTINUE
C                                                                    0 (STMT)
C (11,2)
C                                                                    0 (STMT)
C
C (11,3)
      GOTO 99987
99987 CONTINUE
C                                                                    0 (STMT)
C (9,2)
```

```
C                                                                    0 (STMT)
C (9,3)
      END
```

Note that (X,Y) where X,Y are integers means: X is statement ordering number and Y is ordering inside a statement. We can think that X is a block number while Y is the relative statement number inside that block.

We also note that there are 24 control nodes (blocks) while the largest block number is 28. That means there are some holes in the block number list. That is because we assign the number when we initially begin analyzing, but later we find they are useless and try to delete them. We can delete some of them so as to reduce the number of nodes.

C.3 Probability Matrix of CALCG Subroutine

```
average_probability_matrix:  P =


  0  2  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0

  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1

  0  0  0  2  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0

  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0

  0  0  0  0  0  2  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0

  0  0  0  0  0  0  2  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0

  0  0  0  0  0  0  0  1  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0

  0  0  0  0  0  0  0  0  2  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0

  0  0  0  0  0  0  0  0  0  2  0  0  0  0  0  0  0  0  0  0  0  0  0  0

  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0

  0  0  0  0  0  0  0  0  0  0  0  2  0  0  0  0  0  0  0  0  0  0  0  0

  0  0  0  0  0  0  0  0  0  0  0  0  2  0  0  0  0  0  0  0  0  0  0  0

  0  0  0  0  0  0  0  0  0  0  0  0  0  2  0  0  0  0  0  0  0  0  0  0

  0  0  0  0  0  0  0  0  0  0  0  0  0  0  2  0  0  0  0  0  0  0  0  0

  0  2  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0

  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  2  0  0  0  0  0  0  0

  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  2  0  0  0  0  0  0

  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  2  0  0  0  0  0

  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  1  0  0  0

  0  0  0  0  0  2  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0

  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  2  0  0

  0  0  0  0  0
```