Compilation et optimisation de langages parallèles

Lossing Nelson

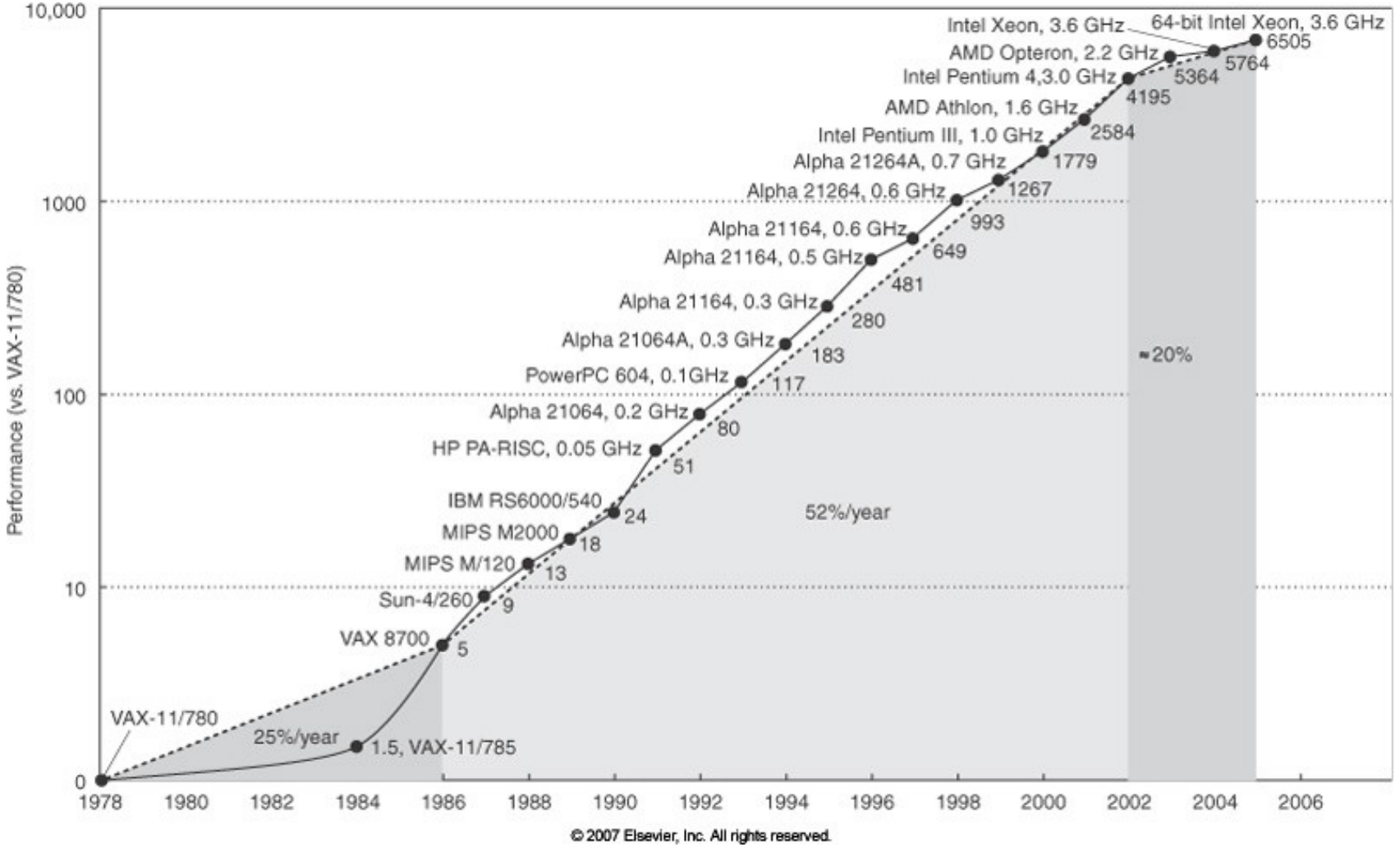
Maître de thèse : Ancourt Corinne
Directeur de thèse : Irigoin François
Centre de Recherche en Informatique
MINES ParisTech

26 mai 2014

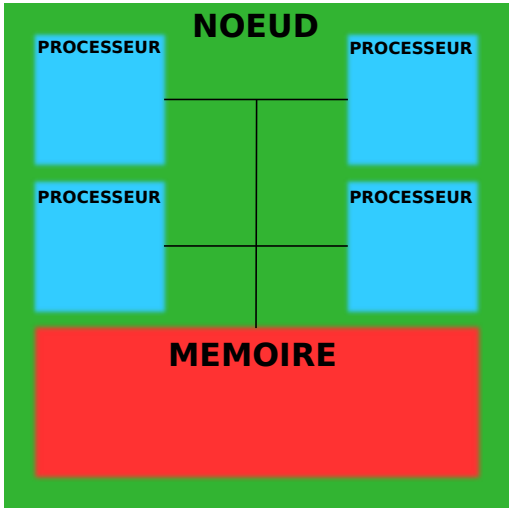
Plan

Plan

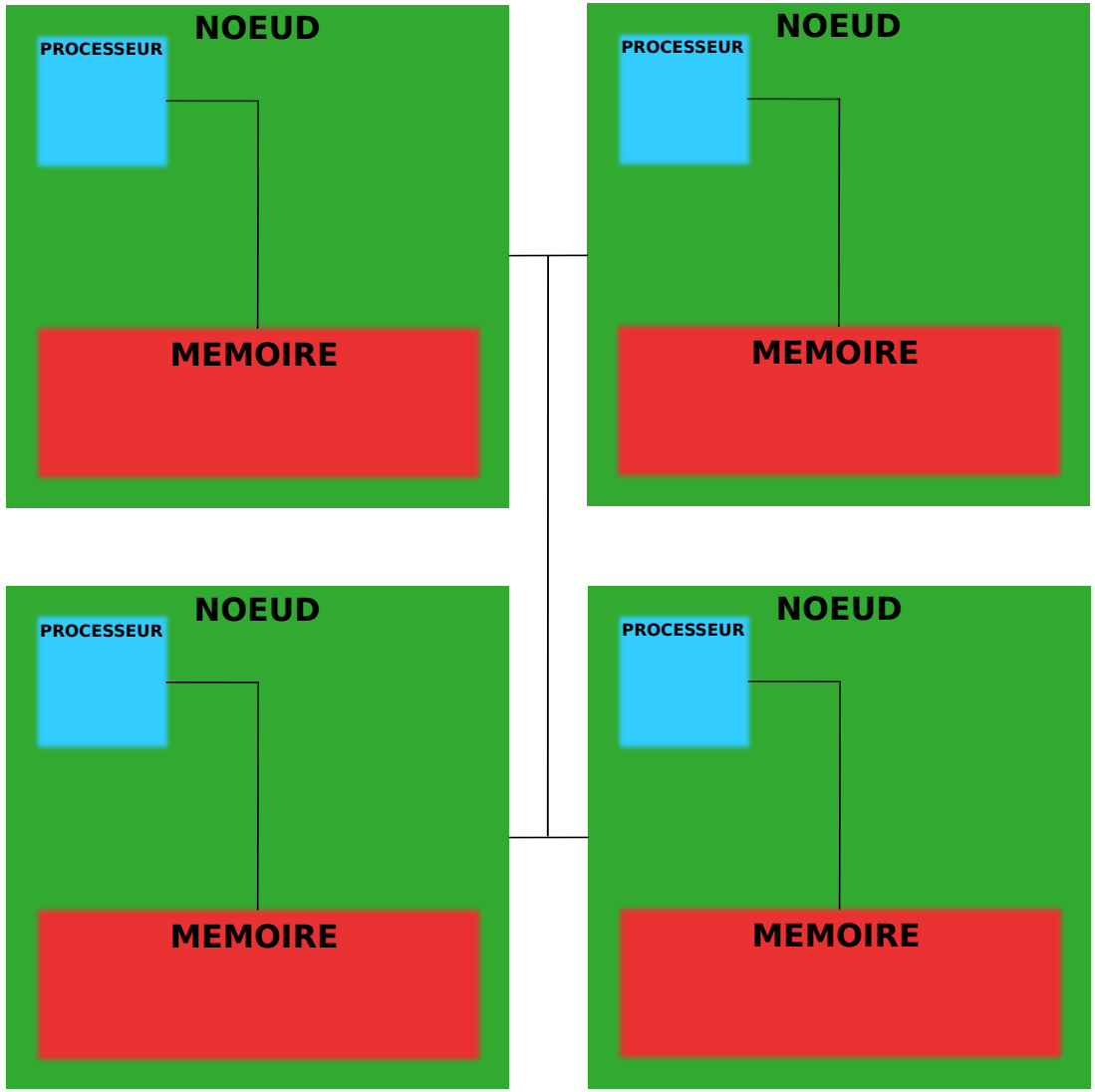
Loi de Moore



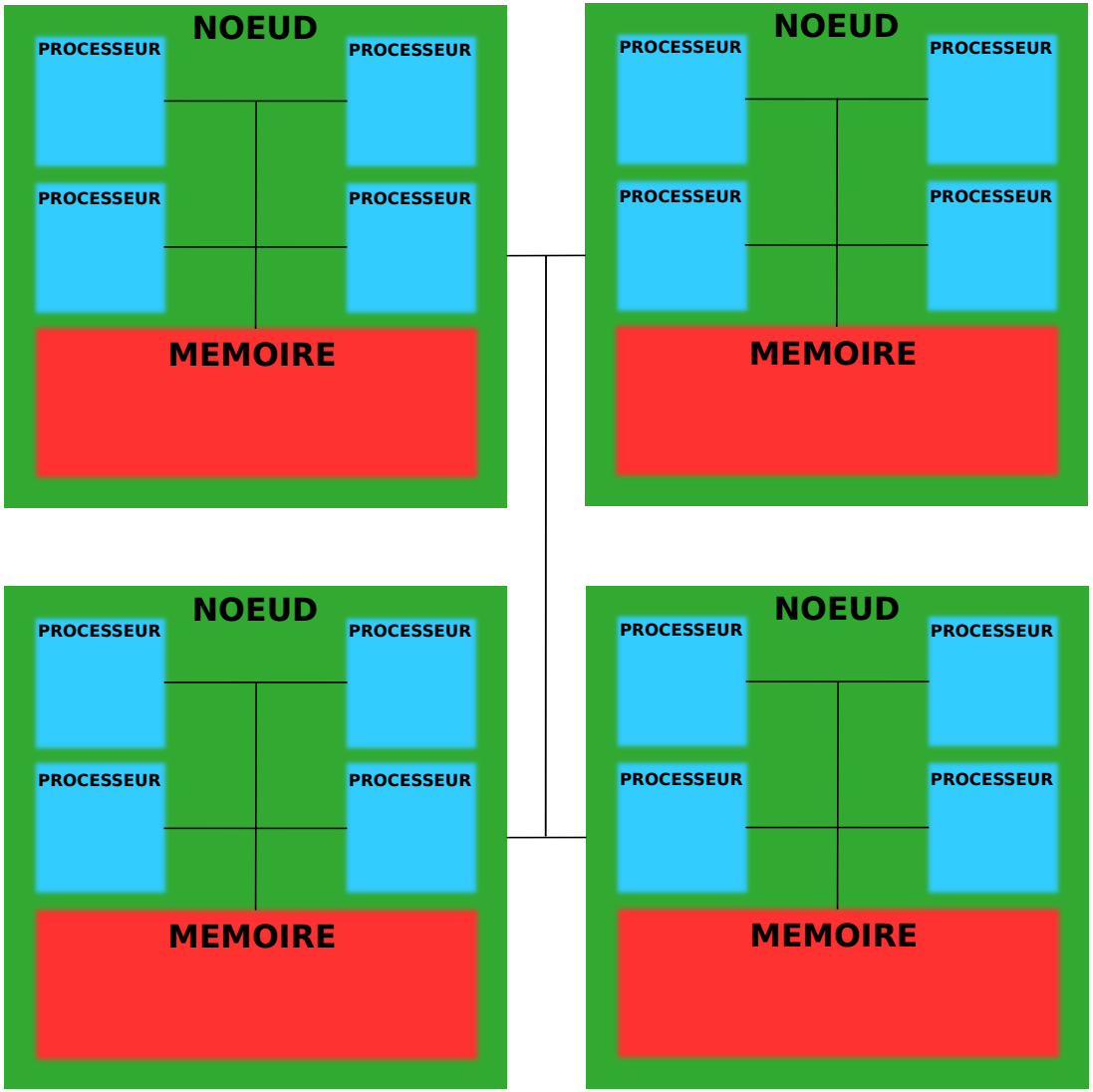
Architectures parallèles



Architectures parallèles



Architectures parallèles



Problème étudié

- Exécution d'applications `C` en parallèle
- Parallélisation de tâches
- Génération automatique d'un programme pour machine à mémoire distribuée
- Validation des transformations

Structure

- Fortran code
- C code

```
int main() {
  int i=10, j=1;
  int k = 2*(2*i+j);

  return k;
}
```

- Static analyses
- Instrumentation/ Dynamic analyses
- Transformations
- Source code generation
- Code modelling
- Prettyprint

- Fortran code
- C code

```
//PRECONDITIONS
int main() {
// P() {}
   int i = 10, j = 1;

// P(i,j) {i==10, j==1}
   int k = 2*(2*i+j);

// P(i,j,k) {i==10,
      j==1, k==42}
   return k;
}
```

Passes

- Mise au point d'un ordonnancement
- Analyse des effets mémoire
- Analyse des régions
- Calcul de transformeurs et préconditions
- Simplification et élimination de code mort
- ...

Plan

État de l'art

- *STEP : a distributed OpenMP for coarse-grain parallelism tool*, D. Millot, A. Muller, C. Parrot et F. Silber-Chaussumier (2009)
- *Automatic Scaling of OpenMP Beyond Shared Memory*, Okwan Kwon (2011)
- *Parallélisation automatique et statique de tâches*, Dounia Khaldi (2013)

Approche

- Réaliser des transformations incrémentales simples
  - Ajout/Suppression d'instructions
  - Déplacement d'instructions
- S'assurer de l'équivalence du code après chaque transformation
- Retarder au maximum la dépendance à l'environnement distribué cible

Prérequis

- Fonction séquentielle correcte à paralléliser.
- Nombre de processeurs numériquement connu.
- Un placement des calculs déjà défini.
- Pas d'écriture sur les arguments de la fonction traitée.
- Absence d'aliasing.
- Pas d'initialisation dans les déclarations.
- Pas de variables globales.
- Pas de pointeurs.

Syntaxe

$$
\begin{array}{lcll}
\langle \textit{function} \rangle & ::= & \langle \textit{declaration} \rangle \text{ (} \langle \textit{declaration} \rangle^* \text{) \{ } \langle \textit{statements} \rangle \text{; [return id ;] \}} & \\
\langle \textit{statements} \rangle & ::= & \varnothing \mid \langle \textit{statement} \rangle \text{; } \langle \textit{statements} \rangle & \\
\langle \textit{statement} \rangle & ::= & \langle \textit{declaration} \rangle & \\
 & \mid & \langle \textit{affectation} \rangle & \\
 & \mid & \text{if } \langle \textit{expression} \rangle \text{ then } \langle \textit{statements} \rangle \text{ [else } \langle \textit{statements} \rangle \text{]} & \\
 & \mid & \text{while } \langle \textit{expression} \rangle \text{ do } \langle \textit{statements} \rangle & \\
 & \mid & \text{for (} \langle \textit{affectation} \rangle \text{; } \langle \textit{expression} \rangle \text{; } \langle \textit{affectation} \rangle \text{) } \langle \textit{statements} \rangle & \\
 & \mid & (\langle \textit{statements} \rangle, \text{ p}) & \text{p} \in \mathbb{N} \\
 & \mid & \langle \textit{statements} \rangle & \\
\langle \textit{declaration} \rangle & ::= & \langle \textit{type} \rangle \text{ id} & \\
\langle \textit{affectation} \rangle & ::= & \langle \textit{variable} \rangle \leftarrow \langle \textit{expression} \rangle & \\
\langle \textit{expression} \rangle & ::= & \langle \textit{variable} \rangle & \\
 & \mid & \text{constant} & \\
 & \mid & \text{true} \mid \text{false} & \\
 & \mid & \langle \textit{unary-op} \rangle \langle \textit{expression} \rangle & \\
 & \mid & \langle \textit{expression} \rangle \langle \textit{binary-op} \rangle \langle \textit{expression} \rangle & \\
\langle \textit{variable} \rangle & ::= & \text{id} & \\
 & \mid & \langle \textit{variable} \rangle \text{. id} & \\
 & \mid & \langle \textit{variable} \rangle \text{[} \langle \textit{index} \rangle \text{]} &
\end{array}
$$

Équivalences

Trois niveaux d'équivalence :

$equiv_1(f_1, f_2) \equiv \forall \sigma, eval(f_1, \sigma) = eval(f_2, \sigma)$

```
int main() {
  int i, r;
  i = 0;
  r = i;
  return r;
}
```

```
int main() {
  int i, r;
  i = 0;
  r = 0;
  return 0;
}
```

$$equiv_2(f_1, f_2) \equiv \forall \sigma, \forall id \in obs(f_1), (eval(f_1, \sigma))(id) = (eval(f_2, \sigma))(id) \wedge \forall \sigma, \forall id \in obs(f_2), (eval(f_2, \sigma))(id) = (eval(f_1, \sigma))(id)$$

$$equiv_3(f_1, f_2) \equiv \forall \sigma, \forall id \in obs(f_1), (eval(f_1, \sigma))(id) = (eval(f_2, \sigma))(permut(id)) \wedge \forall \sigma, \forall id \in obs(f_2), (eval(f_2, \sigma))(id) = (eval(f_1, \sigma))(permut(id))$$

Équivalences

Trois niveaux d'équivalence :

$$equiv_1(f_1, f_2) \equiv \forall \sigma, eval(f_1, \sigma) = eval(f_2, \sigma)$$

$$equiv_2(f_1, f_2) \equiv \forall \sigma, \forall id \in obs(f_1), (eval(f_1, \sigma))(id) = (eval(f_2, \sigma))(id) \wedge \forall \sigma, \forall id \in obs(f_2), (eval(f_2, \sigma))(id) = (eval(f_1, \sigma))(id)$$

```
int main() {
  int i, r;
  i = 0;
  r = 0;
  return r;
}
```

```
int main() {
  int i, r;
  r = 0;
  return r;
}
```

$$equiv_3(f_1, f_2) \equiv \forall \sigma, \forall id \in obs(f_1), (eval(f_1, \sigma))(id) = (eval(f_2, \sigma))(permut(id)) \wedge \forall \sigma, \forall id \in obs(f_2), (eval(f_2, \sigma))(id) = (eval(f_1, \sigma))(permut(id))$$

Équivalences

Trois niveaux d'équivalence :

$$equiv_1(f_1, f_2) \equiv \forall \sigma, eval(f_1, \sigma) = eval(f_2, \sigma)$$

$$equiv_2(f_1, f_2) \equiv \forall \sigma, \forall id \in obs(f_1), (eval(f_1, \sigma))(id) = (eval(f_2, \sigma))(id) \wedge \forall \sigma, \forall id \in obs(f_2), (eval(f_2, \sigma))(id) = (eval(f_1, \sigma))(id)$$

$$equiv_3(f_1, f_2) \equiv \forall \sigma, \forall id \in obs(f_1), (eval(f_1, \sigma))(id) = (eval(f_2, \sigma))(permut(id)) \wedge \forall \sigma, \forall id \in obs(f_2), (eval(f_2, \sigma))(id) = (eval(f_1, \sigma))(permut(id))$$

```
int main() {
  int i, r1;
  i = 0;
  r1 = 0;
  return r1;
}
```

```
int main() {
  int i, r2;
  r2 = 0;
  return r2;
}
```

Principe général

1. Préparer la fonction pour la parallélisation.
2. Optimiser localement la fonction.
3. Optimiser globalement et
   Traduire le code séquentiel en code distribué.

Exemple

```
void foo() {
  int x;
  int y;
  int i;
  int j;
  int a[10];

  j=0;

  for (i=0; i<10; i++) {
    j++;
    a[i]=j;
  }

  x=9;
  y=x*10;
  x=49;

  for (i=0; i<10; i++) {
    a[i]=a[i]+y-x-i;
  }
}
```

Préparer la fonction à paralléliser

1. Ajouter des variables pour chaque processeur.
2. Assurer que les variables ajoutées ont la même valeur entre les processeurs.
3. Supprimer les variables d'origine.

Ajouter des variables pour chaque processeurs

```
int i;


int j;


int a[10];


j=0;
for (i=0; i<10; i++) {
  j++;
  a[i]=j;
}
```

Avant

```
int i;
int i_0;
int i_1;
int j;
int j_0;
int j_1;
int a[10];
int a_0[10];
int a_1[10];
j=0;
for (i=0; i<10; i++) {
  j++;
  a[i]=j;
}
```

Après

Vérifier les valeurs des variables entre les processeurs

```
int i;
int i_0;
int i_1;
int j;
int j_0;
int j_1;
int a[10];
int a_0[10];
int a_1[10];
j=0;


for (i=0; i<10; i++) {
  j++;


  a[i]=j;


}
```

```
int i;
int i_0;
int i_1;
int j;
int j_0;
int j_1;
int a[10];
int a_0[10];
int a_1[10];
j=0;
j_0=j;
j_1=j;
for (i=0; i<10; i++) {
  j++;
  j_0=j;
  j_1=j;
  a[i]=j;
  a_0[i]=a[i];
  a_1[i]=a[i];
}
i_0=i;
i_1=i;
```

Supprimer les variables d'origine

```
int i;
int i_0;
int i_1;
int j;
int j_0;
int j_1;
int a[10];
int a_0[10];
int a_1[10];
j=0;
j_0=j;
j_1=j;
for (i=0; i<10; i++) {
  j++;
  j_0=j;
  j_1=j;
  a[i]=j;
  a_0[i]=a[i];
  a_1[i]=a[i];
}
i_0=i;
i_1=i;
```

```
int i_0;
int i_1;

int j_0;
int j_1;

int a_0[10];
int a_1[10];
j_1=0;
j_0=j_1;
j_1=j_1;
for (i_0=0; i_0<10; i_0++) {
  j_0++;
  j_0=j_0;
  j_1=j_0;
  a_0[i_0]=j_0;
  a_0[i_0]=a_0[i_0];
  a_1[i_0]=a_0[i_0];
}
i_0=i_0;
i_1=i_0;
```

Optimiser localement

1. Supprimer les copies redondantes entre processeurs.
2. Regrouper les copies.

Optimiser localement

```
x_1=9;
x_0=x_1;
x_1=x_1;
y_1=x_1*10;
y_0=y_1;
y_1=y_1;
x_1=49;
x_0=x_1;
x_1=x_1;
```

Avant

```
x_1=9;
y_1=x_1*10;
x_1=49;

y_0=y_1;
x_0=x_1;
```

Après

Transformer un code séquentiel en sa version distribuée

1. Supprimer les copies inutiles entre les processeurs.
2. Traduction du code séquentiel dans un langage parallèle.
3. Génération de fonctions propres pour chaque processeur.

Transformer un code séquentiel en sa version distribuée

```
void foo() {
  int x_0, y_0, i_0, j_0, a_0[10];
  int x_1, y_1, i_1, j_1, a_1[10];
  j_1=0;
  j_0=j_1;

  int temp_00;
  for (i_0=0; i_0<10; i_0++) {
    j_0++;
    a_0[i_0]=j_0;
  }
  j_1=j_0;
  for (temp_00=0; temp_00<10;
       temp_00++) {
    a_0[temp_00]=a_1[temp_10];
  }
  x_1=9;
  y_1=x_1*10;
  x_1=49;
  y_0=y_1;
  x_0=x_1;

  int temp_01;
  for (i_0=0; i_0<10; i_0++) {
    a_0[i_0]=a_0[i_0]+y_0-x_0-i_0;
  }
  for (temp_01=0; temp_01<10;
       temp_01++) {
    a_0[temp_01]=a_1[temp_01];
  }
}
```

Transformer un code séquentiel en sa version distribuée

```
void foo() {
  int numprocs, rank;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank==0) {
    foo_0();
  }
  else if (rank==1) {
    foo_1();
  }

  MPI_Barrier(MPI_COMM_WORLD);
  MPI_Finalize();
}
```

Transformer un code séquentiel en sa version distribuée

```
void foo_0() {
  int x_0, y_0, i_0, j_0, a_0[10];
  MPI_Status status;


  MPI_Recv(j_0, 1, MPI_INTEGER, 1,
       0, MPI_COMM_WORLD, &status);

  for (i_0=0; i_0<10; i_0++) {
    j_0++;
    a_0[i_0]=j_0;
  }
  MPI_Recv(y_0, 1, MPI_INTEGER, 1,
       1, MPI_COMM_WORLD, &status);
  MPI_Recv(x_0, 1, MPI_INTEGER, 1,
       2, MPI_COMM_WORLD, &status);

  for (i_0=0; i_0<10; i_0++) {
    a_0[i_0]=a_0[i_0]+y_0-x_0-i_0;
  }
}
```

```
void foo_1() {
  int x_1, y_1, j_1;
  MPI_Status status;

  j_1=0;
  MPI_SEND(j_1, 1, MPI_INTEGER, 0,
       0, MPI_COMM_WORLD);

  x_1=9;
  y_1=x_1*10;
  x_1=49;

  MPI_SEND(y_1, 1, MPI_INTEGER, 0,
       1, MPI_COMM_WORLD);
  MPI_SEND(x_1, 1, MPI_INTEGER, 0,
       2, MPI_COMM_WORLD);


}
```

Conclusion

- Mise en place d'un langage simplifié
- Formalisation de l'algorithme général
- Formalisation détaillée de la première partie de l'algorithme
  - description de l'algorithme
  - propriétés à prouver

Plan

Cours professionnalisants suivis

- Point de départ. (14h)
- La conduite d'un projet doctoral. (7h)
- La publication scientifique : stratégies, outils et optimisation de sa recherche. (13h)
- Lecture rapide. (21h)

Total du nombre d'heures pour la catégorie Cours professionnalisants : 55 h

- Journée d'accueil des doctorants.
- Journée d'évaluation des doctorants de première année.

Cours scientifiques

- Polyhedral School, ENS Lyon. (5j)

  13-17 Mai 2013

- École d'été *"Introduction to High-Performance Computing"* de KTH. (10j)

  18-29 Août 2014

- TOEIC : 775/990 Mai 2011

Conférences, séminaires et thèses

Séminaires :

- *7e Rencontres de la communauté française de compilation*

  Présentation

- *Adversary-Oriented Computing*, Rachid Guerraoui

- *Understanding and manipulating multi-dimensional loops : A short tour in the world of polyhedral techniques*, Alain Darte

Soutenance de thèses :

- *Parallélisme de tâches et localité de données dans un contexte multi-modèle de programmation pour supercalculateurs hiérarchiques et hétérogènes*, Jean-Yves Vet.

- *Parallélisation automatique et statique de tâches*, Dounia Khaldi.

- *Compiling for a multithreaded dataflow architecture : algorithms, tools, and experience*, Feng Li.

Bibliographie

- *The denotational description of programming langages*, Michael J.C. Gordon (1979)
- *Supercompilers for parallel and vector computers*, Hans Zima (1990)
- *Compilers : principles, techniques, & tools*, A. Aho, M. Lam, R. Sethi, J. Ullman (2007)

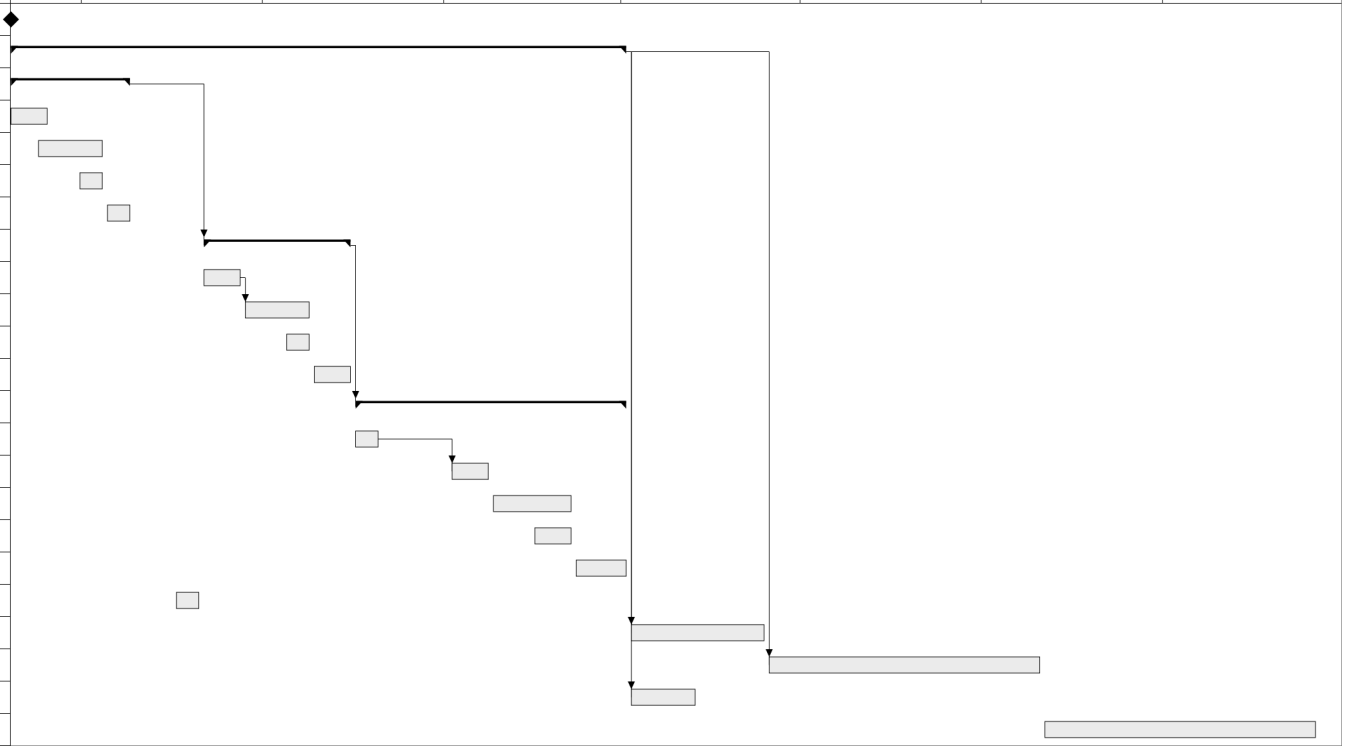
Plan

1re année

- Mise en place d'un langage simplifié
- Formalisation de l'algorithme général
- Formalisation détaillée de la première partie de l'algorithme
  - description de l'algorithme
  - propriétés à prouver
- 55/60 heures de cours professionnalisants validées
- Participation à plusieurs séminaires et soutenances
- Perfectionnement dans les langages parallèles

Planning

Nom	Travail
rapport première année	
Algorithme	**215j**
étape 1	**60j**
Formaliser l'input	15j
Implémenter	25j
Tester l'implémentation	10j
Prouver la première étape	10j
étape 2	**65j**
Formaliser l'algorithme	15j
Implémenter	25j
Tester l'implémentation	10j
Prouver la deuxième étape	15j
étape 3	**90j**
Etudier différents langages cibles	10j
Formaliser l'algorithme	15j
Implémenter	30j
Tester l'implémentation	15j
Prouver la deuxième étape	20j
Ecole d'été	10j
Etudier les pointeur	50j
Etudier la parallélisation hybride	100j
Rédaction d'un article	25j
rédaction du manuscrit de thèse	100j



Compilation et optimisation de langages parallèles

Lossing Nelson

Maître de thèse : Ancourt Corinne
Directeur de thèse : Irigoin François
Centre de Recherche en Informatique
MINES ParisTech

26 mai 2014