1986

ENSMP-CAI-86-E81 Document No. E81

ECOLE NATIONALE SUPERIEURE
DES MINES DE PARIS

CENTRE D'AUTOMATIQUE ET INFORMATIQUE

Supernodes and Alliant FX/8 Minisupercomputer

François Irigoin
Rémi Triolet

August 1986

35, rue Saint Honoré
77305 FONTAINEBLEAU CEDEX
FRANCE

ABSTRACT

This report presents the experimental results obtained on an Alliant FX/8 minisupercomputer with programs restructured according to the supernode partitioning method proposed by F. Irigoin in his thesis (in process). Supernode restructuring increases data locality and seems to improve the speed by 50% when a large amount of data is processed.

Some knowledge about the Alliant's architecture and supernode partitioning is assumed.

Introduction

The Alliant FX/8 minisupercomputer uses a cache memory to speedup vector memory accesses. W. Jalby[2] has shown that variables should be kept in the cache to obtain a reasonable performance. By using a block matrix algorithm for matrix multiplication a speed of 28 MFlops out of 47 MFlops (peak) is reached instead of 9 MFlops with the common algorithm (based on BLAS 1) and 19 MFlops with a matrix-vector algorithm (based on BLAS 2).

In his thesis[1] , F. Irigoin proposes an automatic restructuring technique to minimize synchronization and communication costs when parallelizing perfectly nested DO loops. This theoretically leads to an improved data locality and to a better usage of the cache memory. As most parallelization techniques, this one is based on dependence analysis.

As no interesting multiprocessors are available in France or in Europe for computer science experiments, we were allowed to spend 4 days at the Center for Supercomputing Research and Development, University of Illinois, to gather experimental data confirming our theoretical expectations.

We first present summary outlines of the restructuring technique proposed by F. Irigoin and used in the experiments. We then describe a few problems we had to cope with before getting interesting results and explain why a first version of restructured program failed to improve the performance. Finally satisfactory results were obtained with a new version: the computational speed does not decrease significantly when the data size increases.

1. Supernode Restructuring

The purpose of this program transformation is to use and reuse available values as long as possible. A usual sweep of the iteration domain, as shown at the bottom of figure 1, exhibits a very poor locality since a whole front has to be computed before a value is used again. On the contrary the clustering of basic computation nodes into **supernodes** and the reordering of the whole computation, depicted in the same figure 1, produce a program with a good locality. For example, the supernode size can be chosen small enough to let the related data set fit into the cache and large enough to take advantage of the vector facilities and to keep the extra control overhead small.

This transformation is purely syntactic. It does not rely on commutativity or associativity but only on dependence vectors. As a result, the value histories are not changed and the final results are equal at the bit level to the ones obtained by the usual simple sweep.

Many programs could have been considered for experimenting but we had time only for a simple one. The program chosen is issued from a finite difference resolution of the heat equation. Its scheme uses three points (i.e. values) from the previous iteration to compute a new point. This generates three dependence vectors, $\vec{d}_1$, $\vec{d}_2$, $\vec{d}_3$, which constrain the set of possible supernode shapes. The execution is computation bounded since O(n^2) computations are performed on only O(n) memory elements. Here is the test program, written in Fortran-8X:

```
REAL*8 BARRE(0:LNGBARRE+1)

BARRE = 273.

DO T=1,TFIN
    BARRE(1:LNGBARRE) = (BARRE(0:LNGBARRE-1)+BARRE(1:LNGBARRE)
&        +BARRE(2:LNGBARRE+1))/3.
    BARRE(0) = 273. + 0.1*T
    BARRE(LNGBARRE+1) = BARRE(0)
ENDDO
```

The boundary conditions are computed and stored in BARRE(0) and BARRE(LNGBARRE+1). The bar is heated by both ends in a symmetrical way, from time T=1 to time T=TFIN.

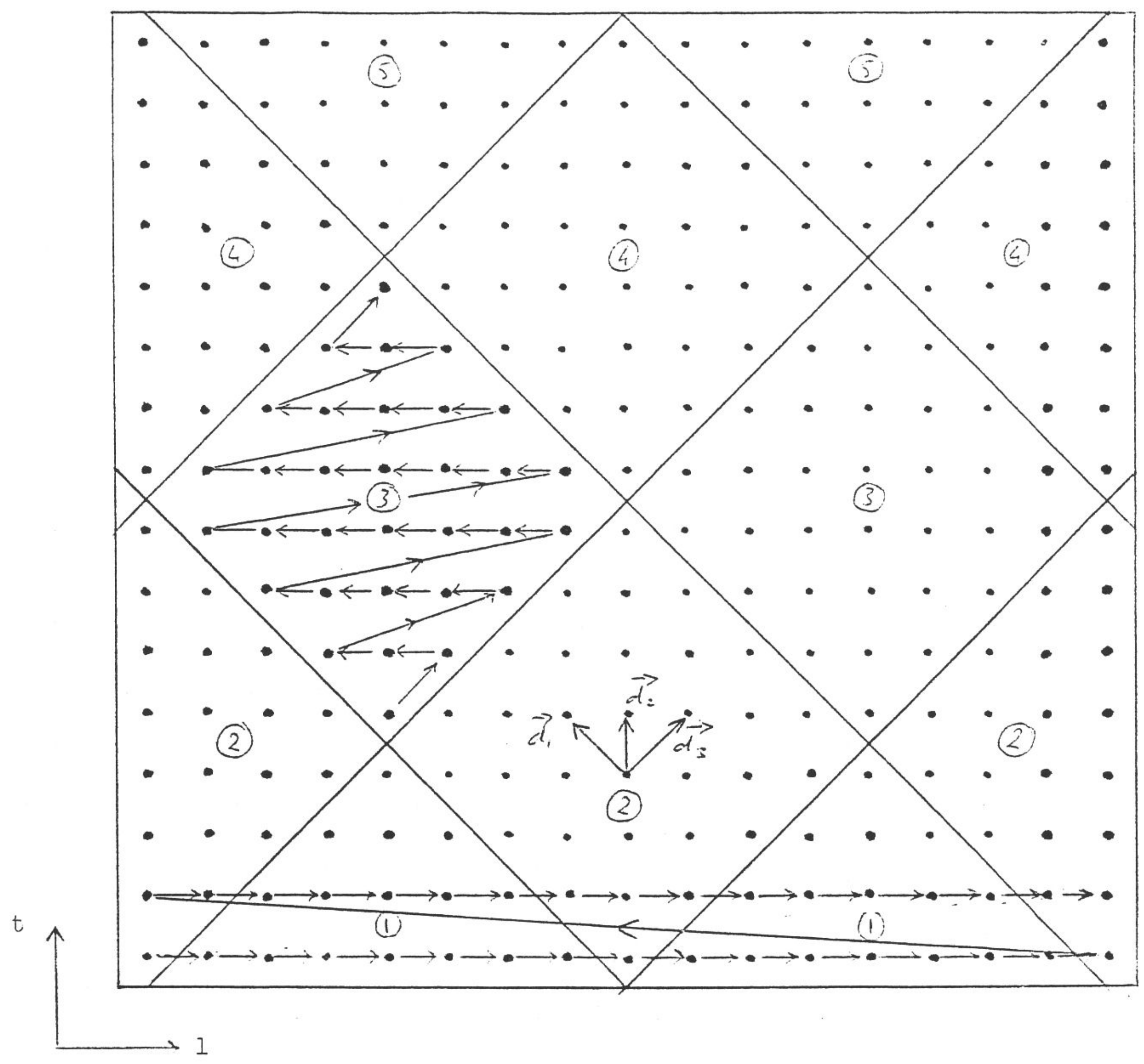

Figure 1. Supernode Partitionning

2. Initial Program

A few elements of the array BARRE are assigned wrong values when this program is executed. More precisely, as the array BARRE is sliced in 32 element pieces, boundary elements are not properly computed. Of course, errors increase and propagate along the slices from iteration to iteration. The Fortran compiler generates a wrong code as if the dependence carried by the vector statement were ignored. Furthermore, the compiler does not transform a division by a constant into a multiplication by the inverse in spite of the huge time cost of divisions on the Alliant.

Thus we rewrote the previous program and used two arrays BARRE1 and BARRE2 to avoid intricate dependences. The division was also hand replaced by a multiplication. And the compiler was still not able to generate an add multiply operation: two adds and one multiply were necessary to compute each node.

Here is the second version of the program. This one is used as a reference for all speedup computations.

```
      REAL*8 BARRE1(0:LNGBARRE+1)
      REAL*8 BARRE2(0:LNGBARRE+1)

      BARRE2 = 273.

      DO T = 1, TFIN, 2
         BARRE1(1:LNGBARRE) = (BARRE2(0:LNGBARRE-1)+BARRE2(1:LNGBARRE)
     &         +BARRE2(2:LNGBARRE+1))/3.
         BARRE1(0) = 273. + 0.1*T
         BARRE1(LNGBARRE+1) = BARRE1(0)

         BARRE2(1:LNGBARRE) = (BARRE1(0:LNGBARRE-1)+BARRE1(1:LNGBARRE)
     &         +BARRE1(2:LNGBARRE+1))/3.
         BARRE2(0) = 273. + 0.1*(T+1)
         BARRE2(LNGBARRE+1) = BARRE2(0)
      ENDDO
```

3. Cache Memory Effect

Before testing supernode restructuring effects, we tried to exhibit a cache size effect and a virtual memory effect by increasing LNGBARRE in the reference program. Virtual memory tests were quickly stopped because they seem to increase UNIX response time in a dramatic way and bother all users of the machine.

Cache size effects were found to be very limited as can be seen with curve I of figure 2, where speeds in MFlops are given as function of the memory size required (the memory size unit is the Kword and a logarithmic scale is used). This seems to contradict W. Jalby's results[2] with matrix multiplication. They show that a careful use of the cache causes a dramatic speed improvement. But W. Jalby had to write his block matrix subroutines in assembly language to suppress Fortran generated code overhead and to give cache misses a sizable part of the total time.

It was not possible in four days to learn how to program the Alliant machine in assembly language and we tried to write Fortran code that would be translated into efficient machine code. By unrolling the recursion once and computing $BARRE_i = f \circ f(BARRE_{i-2})$ instead of $BARRE_i = f(BARRE_{i-1})$, higher performances shown by curve III of figure 2 were obtained. This time, the compiler generated multiply add instructions.

W. Jalby suggested a two point scheme which produces the best cache effect (curve II) by increasing the number of vector loads and stores in relation to the number of vector computations. Unfortunately this scheme does not solve the initial problem.

The choice of Megaflops as unit was a mistake since the unrolling introduced some redundancy in the computations. Expressed in Meganodes/sec., which is the unique useful unit for users, curve III would be closer to curve I but would stay higher.

Because of hardware problems, the number of processing elements varied during the experiments between 6 and 8, being 7 most of the time.

Due to the lack of time, we decided not to use unrolling in supernodes to keep the code simpler. As a consequence, unrolling was also not used in the reference program.

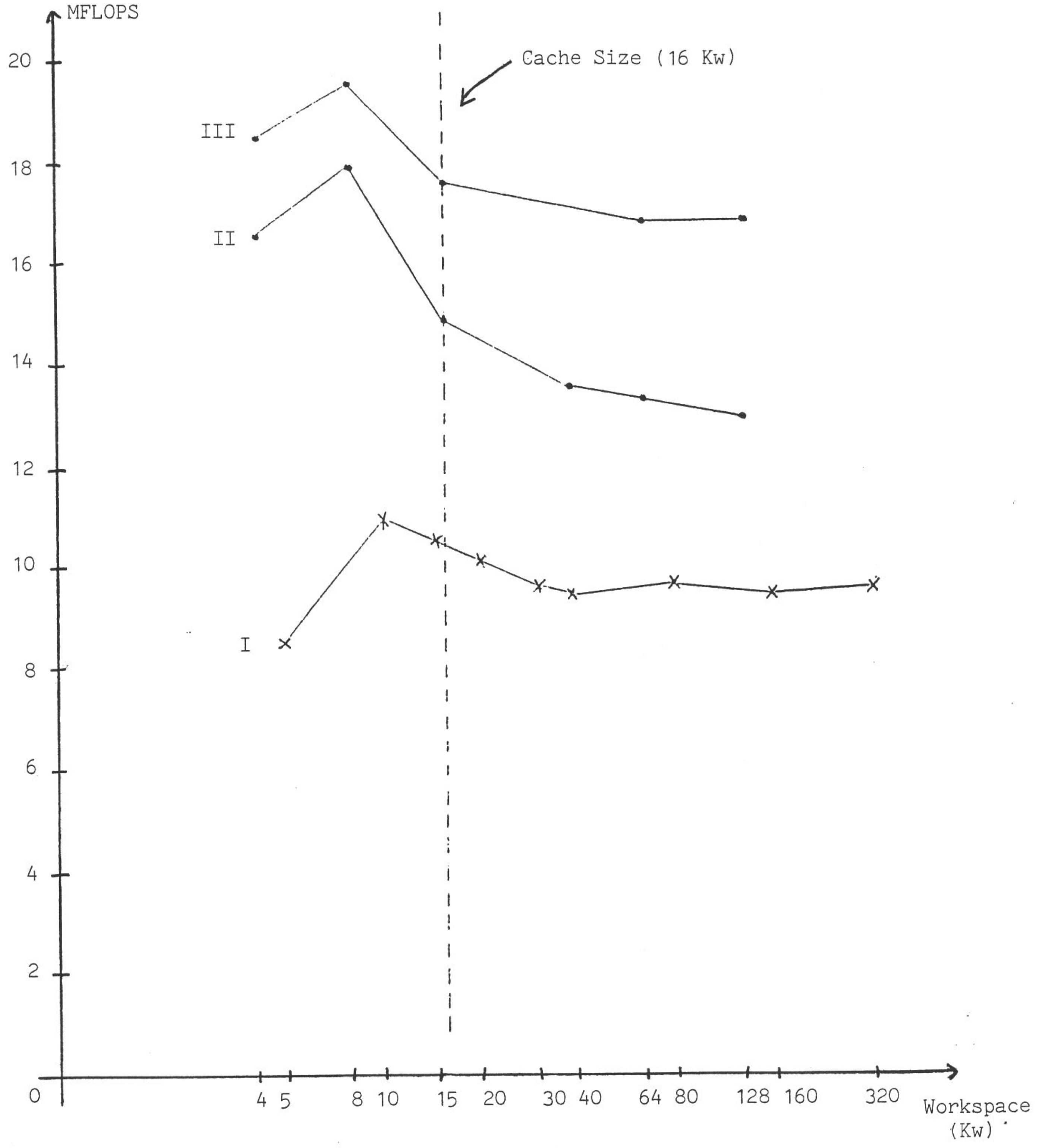

Figure 2. Cache Effect Without Supernodes

4. Supernode Overhead

As the reader can guess from figure 1, supernode restructuring leads to a very intricate code. The first version we developped is not shown here because of its length and because of the poor results we got.

The speed fell from 12 MFlops to 0.82 MFlops! First of all, the compiler was not able to detect and to generate parallel code from the restructured program because of the complex control structure. Second, partial supernodes, like the ones from iterations 1 and 5 in figure 1, were coded in a very simple way, with a complex test to decide for each node whether it belonged or not to the iteration domain. Measurements showed that, on the average, a partial supernode execution time was 15 times longer than a total one. The control overhead was tremendous and the vector arithmetic unit was not used.

To compensate for a penalty of more than an order of magnitude, huge iteration domains had to be used to get a very low ratio of partial to total supernodes, and flat domains were prohibited since their surface (i.e. the number of computations and hence the total number of supernodes) grows like their perimeter (i.e. the number of partial supernodes).

Although it was already day 3 evening, we decided to recode the supernode version with optimal code for partial supernodes during day 4. We imposed relations between the loop bounds and the supernode size to have only four types of partial supernodes. Furthermore, we had to split our code in many subroutines to avoid VAST's devastating effects and to make sure parallel and vector loops would be generated at the right place. This second code with supernodes is listed in annexe 1.

5. Supernode Effect

Although supernode restructuring has a negative effect for very small iteration domains, it allows to keep a high computational speed over a large range of memory size requirement, 50% higher than the non-restructured code. Measurements were done only twice due to lack of time and are shown in figure 3, where the speed in meganodes/sec. is a function of the memory size logarithm. Supernode measure points are represented with x's while circles are used for the reference program.

Only 7 processors were available on the Alliant machine we used. Supernodes with an edge of 64 nodes were chosen since they provided enough vector speed in spite of the varying vector length in one supernode. It seems to imply that startup times of pipeline units are very low.

With our reference program the Alliant Fortran compiler chops the long vectors BARRE1 and BARRE2 into 32 element vectors and the resulting slices are given one at a time to each elementary processing unit by a control unit. The two levels of hardware parallelism are mapped on one level of program parallelism. With the supernode version, each processor has to execute a different supernode (first level of parallelism) and the vector units are used inside a supernode on short vectors (second level). The VAST directives were not sufficient to warranty the desired code generation and we had to create subroutines for total and partial supernodes to control the directive scopes. The parallelism between supernodes was insured by a first set of directives since VAST cannot detect parallelism between CALLs (CVD$ CONCUR and CVD$ CNCALL) and only vector code was allowed for the subroutines (CVD$ NOCONCUR). The subroutines were compiled with the recursive option. The tests were executed during our last night at CSRD and we were alone on the machine most of the time. However we sometimes launched more than one execution at a time.

We also varied the supernode edge from 4 to 4096 for two different iteration domains (LNGBARRE=16384, TFIN=4096 and LNGBARRE=8192 and TFIN=8192) and observed the expected effect: small supernodes increase the control overhead and do not take advantage of the vector units, while large supernodes decrease the parallelism degree and do not leave enough work for the 7 processors. This can be seen on figure 4 where X's represent measure points for the first domain and circles are relative to the second one. The two horizontal stripes give the range of performance reached by the reference program. They also convey an idea of the measure precision.

In both cases supernode coding provided better performances in the case of certain sizes of supernodes. F. Irigoin predicted a flat maximum which does not show up on figure 4 because of the logarithmic scale used and because of the limited size of the iteration domain.

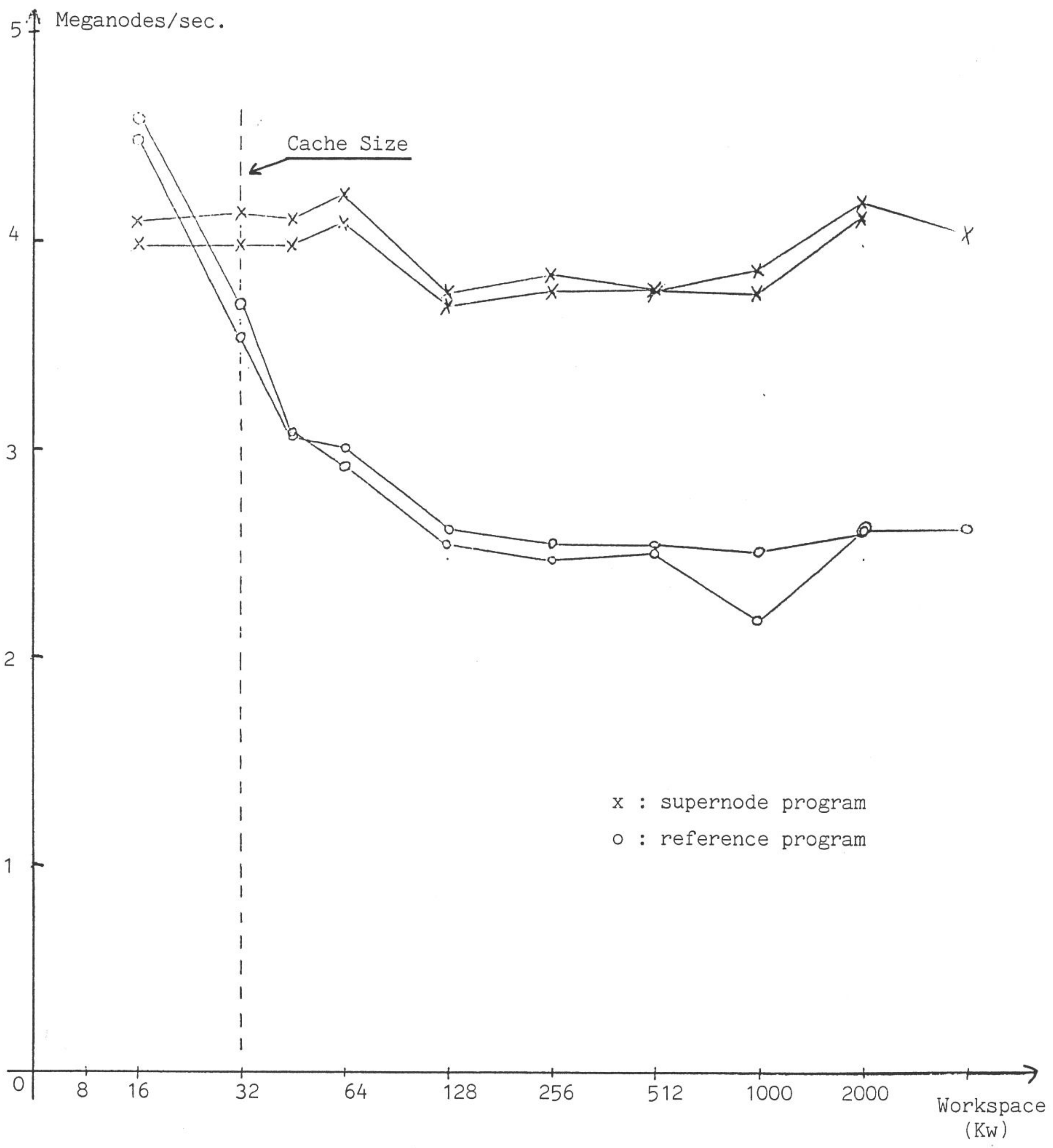

Figure 3: Computational Speed as a Function of the Memory Size Logarithm

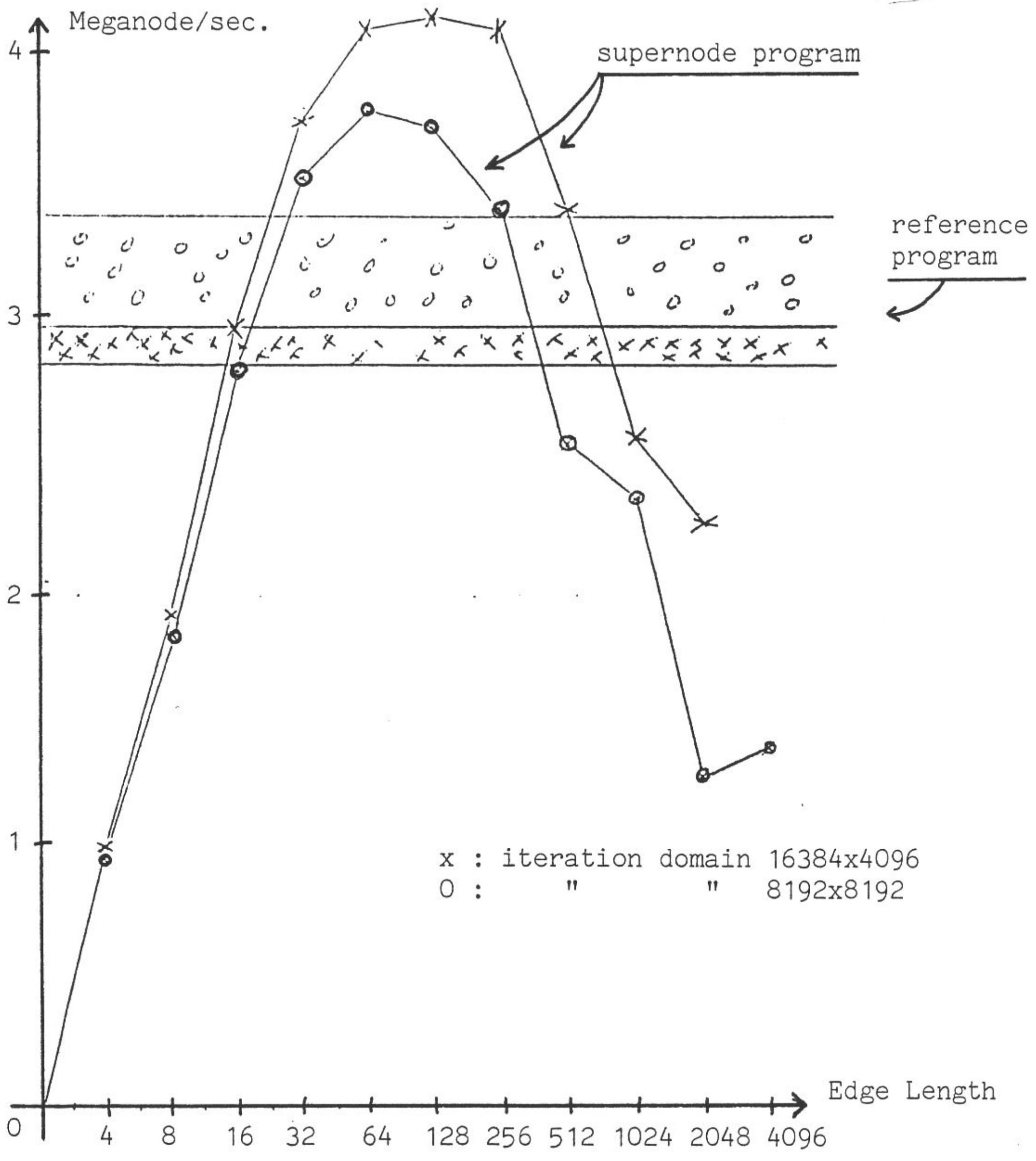

Figure 4: Computational Speed as a Function of Supernode Edge Length Logarithm

Conclusion

It is not possible to draw a conclusion after only 4 days of experiments on a previously unknown machine. Due to the lack of time and to the low quality of the Fortran code generator, we were not able to reach the best possible performance of the machine but we believe with W. Jalby that cache effect would then become even more important.

However, supernode restructuring seems to have a favorable effect on execution time and its study deserves more time. For example this technique should be tried on other machines with a memory hierarchy and a higher vector-to-scalar speed ratio like the Cray 2 to see if supernodes with constant length vectors are of interest. It would also be interesting tō use a set of programs to do measurements.

Moreover the effect of supernodes in a virtual memory environment should also be tested.

Handwriting code with supernodes is tedious and error prone work: the initial one-page program was extended to almost seven pages, mainly to provide optimal code for partial supernodes. F. Irigoin is going to propose an algorithm to perform the transformation automatically in his thesis.

Acknowledgements

We are grateful to CSRD and especially to Dr. Sameh for letting us use two of their FX/8 Alliant without any restriction and for providing us with all the support necessary to get interesting results in only 4 days.

Special thanks are also due to W. Jalby who organized our visit to CSRD and was always available for a piece of advice about the Alliant machine or its software.

References

1. F. Irigoin, "Partitionnement des boucles imbriquées : une technique d'optimisation pour les programmes scientifiques," Thèse de Doctorat d'Université, Université PARIS-VI, PARIS (1987).
2. W. Jalby and U. Meier, "Optimizing Matrix Operations on a Parallel Multiprocessor With a Memory Hierarchy," Technical Report CSRD, University of Illinois at Urbana-Champaign, Urbana-Champaign (Feb. 1986).

A N N E X E   I

```
Dec 31 11:20 1986  barre.txt Page 1


C
C       Evolution temporelle de la temperature d'une barre.
C
        implicit integer (a-z)
C
C       Quelques parametres.
C       Attention, la taille des supernoeuds doit etre paire car les
C       supernoeuds sont prevus pour demarrer avec le tableau b1, et
C       car les supernoeuds voisins en x sont decales de TAILLE
C
        parameter (TAILLE = 8)
        parameter (NITER = 32)
        parameter (DEGPAR = 8)
        parameter (TFIN = TAILLE*2*NITER)
        parameter (LNGBARRE = 1 + TAILLE*2*DEGPAR)
C
        doubleprecision COEF, INCRTEMP
        parameter (COEF =  1.0/3.0)
        parameter (INCRTEMP = 0.1)
C
C       Quelques declarations.
C       Les conditions aux limites sont donnees par des fonctions
C       explicites et calculees quand c'est necessaire
C
        doubleprecision b1(0:LNGBARRE+1), b2(0:LNGBARRE+1)
C
C       Initialisations et verifications
C
        if (mod(TAILLE,2).ne.0) then
                stop 'TAILLE doit etre un nombre pair'
        endif
C
        b2(0:LNGBARRE+1) = 273.0
C
C       Debut de la partie calcul.
C
C       Chronometrage.
C
        call getime(dduse, ddsys)
C
C       Premiere iteration.
C
        t = 1
CVD$ CONCUR
CVD$ CNCALL
        do 10 node = 1, DEGPAR
                call th(t, (2*node - 1)*TAILLE + 1, TAILLE, COEF,
     &                  INCRTEMP, LNGBARRE, TFIN, b1, b2)
10      continue
C
C       Iterations medianes.
C
CVD$ NOCONCUR
CVD$ DEPCHK
CVD$ SYNC
        do 20 iter = 2, 2*NITER
                t = iter*TAILLE + 1
                if (mod(iter,2).eq.0) then
CVD$ CONCUR
CVD$ CNCALL
                        do 30 node = 1, DEGPAR
                                call itpaire(node, t, l, TAILLE,
     &                                  COEF, INCRTEMP, LNGBARRE,
     &                                  TFIN, b1, b2)
30                      continue
                else
CVD$ CONCUR
CVD$ CNCALL
                        do 40 node = 1, DEGPAR
                                l = (2*node - 1)*TAILLE + 1
                                call itimpai(node, t, l, TAILLE,
     &                                  COEF, INCRTEMP, LNGBARRE,
     &                                  TFIN, b1, b2)
40                        continue
                 endif
20         continue
C
C          Iterations finales.
C
           t = TFIN - TAILLE + 1
CVD$ CONCUR
CVD$ CNCALL
           do 50 node = 1, DEGPAR
                 call tb(t,(2*node - 1)*TAILLE + 1, TAILLE, COEF,
     &                 INCRTEMP, LNGBARRE, TFIN, b1 ,b2)
50         continue
C
C          Chronometrage.
C
           call getime(dfuse, dfsys)
C
C          Impression des resultats.
C
           flops = LNGBARRE*TFIN
           duree = (dfuse + dfsys) - (dduse + ddsys)
           print *, 'Longueur de la barre:       ', LNGBARRE
           print *, 'Nombre d iterations:        ', TFIN
           print *, 'Nombre de points calcules: ', flops
           print *, 'Duree totale:               ', duree/100.
           print *, 'Megapoints/seconde:         ', flops*100./duree
C
C          Valeurs finales des temperatures de la barre
C
           write (6, 1000) (i, b2(i), i = 1, LNGBARRE)
1000       format (3(i8, g17.10))
C
C          Fin du programme.
C
           stop
           end

CVD$ NOCONCUR
           subroutine th(t, l, TAILLE, COEF, INCRTEMP, LNGBARRE, TFIN, b1, b2)
C
C          Triangle haut.
C
           implicit integer (a-z)
           doubleprecision COEF, INCRTEMP
           doubleprecision b1(0:LNGBARRE+1), b2(0:LNGBARRE+1)
C
           l1 = l - TAILLE - 1
           l2 = l + TAILLE + 1
C
           do 10 i = TAILLE, 1, -2
                 l1 = l1 + 2
                 l2 = l2 - 2
                 b1(l1:l2) = (b2(l1-1:l2-1) + b2(l1:l2) + b2(l1+1:l2+1))*COEF
C
                 b2(l1+1:l2-1) = (b1(l1:l2-2) + b1(l1+1:l2-1) +
     &                           b1(l1+2:l2))*COEF
10         continue
C
           return
           end

           subroutine tb(t, l, TAILLE, COEF, INCRTEMP, LNGBARRE, TFIN, b1, b2)
C
C          Triangle bas.
C
           implicit integer (a-z)
           doubleprecision COEF, INCRTEMP
           doubleprecision b1(0:LNGBARRE+1), b2(0:LNGBARRE+1)
C
           l1 = l + 2
```

```
        l2 = l - 2
C
        do 10 i = 1, TAILLE, 2
                l1 = l1 - 2
                l2 = l2 + 2
                b1(l1:l2) = (b2(l1-1:l2-1) + b2(l1:l2) + b2(l1+1:l2+1))*COEF
C
                b2(l1-1:l2+1) = (b1(l1-2:l2) + b1(l1-1:l2+1) +
     &                           b1(l1:l2+2))*COEF
10      continue
C
        return
        end

        subroutine chd(t, l, TAILLE, COEF, INCRTEMP, LNGBARRE, TFIN, b1, b2)
C
C       Coin haut droit.
C
        implicit integer (a-z)
        doubleprecision COEF, INCRTEMP
        doubleprecision b1(0:LNGBARRE+1), b2(0:LNGBARRE+1)
C
        if (l.ne.1) stop 'chd:l different de 1'
C
        l2 = TAILLE + 2
C
        do 10 i = TAILLE, 1, -2
                l2 = l2 - 2
                b1(0) = b2(0) + INCRTEMP
                b1(1:l2) = (b2(0:l2-1) + b2(1:l2) + b2(2:l2+1))*COEF
C
                b2(0) = b1(0) + INCRTEMP
                b2(1:l2-1) = (b1(0:l2-2) + b1(1:l2-1) + b1(2:l2))*COEF
10      continue
C
        return
        end

        subroutine cbd(t, l, TAILLE, COEF, INCRTEMP, LNGBARRE, TFIN, b1, b2)
C
C       Coin bas droit.
C
        implicit integer (a-z)
        doubleprecision COEF, INCRTEMP
        doubleprecision b1(0:LNGBARRE+1), b2(0:LNGBARRE+1)
C
        if (l.ne.1) stop 'cbd: l different de 1'
C
        l2 = -1
C
        do 10 i = 1,TAILLE,2
                l2 = l2 + 2
                b1(0) = b2(0) + INCRTEMP
                b1(1:l2) = (b2(0:l2-1) + b2(1:l2) + b2(2:l2+1))*COEF
C
                b2(0) = b1(0) + INCRTEMP
                b2(1:l2+1) = (b1(0:l2) + b1(1:l2+1) + b1(2:l2+2))*COEF
10      continue
C
        return
        end

        subroutine chg(t, l, TAILLE, COEF, INCRTEMP, LNGBARRE, TFIN, b1, b2)
C
C       Coin haut gauche.
C
        implicit integer (a-z)
        doubleprecision COEF, INCRTEMP
        doubleprecision b1(0:LNGBARRE+1), b2(0:LNGBARRE+1)
C
        if (l.ne.LNGBARRE) stop 'chg: l different de LNGBARRE'
C
```

```
        l1 = LNGBARRE - TAILLE -1
C
        do 10 i = TAILLE,1,-2
                l1 = l1 + 2
                b1(LNGBARRE+1) = b2(LNGBARRE+1) + INCRTEMP
                b1(l1:LNGBARRE) = (b2(l1-1:LNGBARRE-1) + b2(l1:LNGBARRE) +
     &                           b2(l1+1:LNGBARRE+1))*COEF
C
                b2(LNGBARRE+1) = b1(LNGBARRE+1) + INCRTEMP
                b2(l1+1:LNGBARRE) = (b1(l1:LNGBARRE-1) + b1(l1+1:LNGBARRE) +
     &                             b1(l1+2:LNGBARRE+1))*COEF
C
10      continue
C
        return
        end

        subroutine cbg(t, l, TAILLE, COEF, INCRTEMP, LNGBARRE, TFIN, b1, b2)
C
C       Coin bas gauche.
C
        implicit integer (a-z)
        doubleprecision COEF, INCRTEMP
        doubleprecision b1(0:LNGBARRE+1), b2(0:LNGBARRE+1)
C
        if (l.ne.LNGBARRE) stop 'cbg: l different de LNGBARRE'
C
        l1 = LNGBARRE + 2
C
        do 10 i = 1,TAILLE,2
                l1 = l1 - 2
                b1(LNGBARRE+1) = b2(LNGBARRE+1) + INCRTEMP
                b1(l1:LNGBARRE) = (b2(l1-1:LNGBARRE-1) + b2(l1:LNGBARRE) +
     &                           b2(l1+1:LNGBARRE+1))*COEF
C
                b2(LNGBARRE+1) = b1(LNGBARRE+1) + INCRTEMP
                b2(l1-1:LNGBARRE) = (b1(l1-2:LNGBARRE-1) + b1(l1-1:LNGBARRE) +

     &                             b1(l1:LNGBARRE+1))*COEF
C
10      continue
C
        return
        end

        subroutine itpaire(node,t,l,TAILLE,COEF,INCRTEMP,LNGBARRE,TFIN,b1,b2)
C
C       iterations paires
C
        implicit integer (a-z)
C
        doubleprecision COEF, INCRTEMP
        doubleprecision b1(0:LNGBARRE+1), b2(0:LNGBARRE+1)
C
        if (node.ne.1)  then
C               l=(2*node - 2)*TAILLE + 1
C               print *,'DBG ', 'call tb ',t,l
                call tb(t,(2*node - 2)*TAILLE + 1,
     &                  TAILLE,COEF,INCRTEMP,
     &                  LNGBARRE,TFIN,b1,b2)
C               print *,'DBG ', 'call th ',t+TAILLE,l
                call th(t+TAILLE,(2*node - 2)*TAILLE + 1,
     &                  TAILLE,COEF,INCRTEMP,
     &                  LNGBARRE,TFIN,b1,b2)
        else
C               print *,'DBG ', 'call cbd ',t,1
                call cbd(t,1,TAILLE,COEF,INCRTEMP,
     &                  LNGBARRE,TFIN,b1,b2)
C               print *,'DBG ', 'call chd ',t+TAILLE,1
                call chd(t+TAILLE,1,TAILLE,COEF,INCRTEMP,
     &                  LNGBARRE,TFIN,b1,b2)
C               print *,'DBG ', 'call cbg ',t,LNGBARRE
                call cbg(t,LNGBARRE,TAILLE,COEF,INCRTEMP,
     &                  LNGBARRE,TFIN,b1,b2)
C               print *,'DBG ', 'call chg ',t+TAILLE,LNGBARRE
                call chg(t+TAILLE,LNGBARRE,TAILLE,COEF,INCRTEMP,
     &                  LNGBARRE,TFIN,b1,b2)
        endif
C
        return
        end

        subroutine itimpai(node,t,l,TAILLE,COEF,INCRTEMP,LNGBARRE,TFIN,b1,b2)
C
C       iterations impaires
C
        implicit integer (a-z)
C
        doubleprecision COEF, INCRTEMP
        doubleprecision b1(0:LNGBARRE+1), b2(0:LNGBARRE+1)
C
C               print *,'DBG ', 'call tb ',t,l
                call tb(t,(2*node - 1)*TAILLE + 1,
     &                  TAILLE,COEF,INCRTEMP,
     &                  LNGBARRE,TFIN,b1,b2)
C               print *,'DBG ', 'call th ',t+TAILLE,l
                call th(t+TAILLE,(2*node - 1)*TAILLE + 1,
     &                  TAILLE,COEF,INCRTEMP,
     &                  LNGBARRE,TFIN,b1,b2)
C
        return
        end
```