Processor Allocation and Loop Scheduling on Multiprocessor Computers *

Nadia Tawbi
BULL Corporate Research Center
Rue Jean Jaurès
78340 Les Clayes sous Bois
FRANCE
Nadia.Tawbi@frcl.bull.fr

Paul Feautrier
IBP-MASI université Paris VI
4, place Jussieu
75252 Paris Cedex 5
FRANCE
feautrier@masi.ibp.fr

ABSTRACT

This paper is concerned with the automatic exploitation of the parallelism detected in a sequential program. The target machine is a shared memory multiprocessor.

The main goal is minimizing the completion time of the program. To achieve this one has first to distribute the code over the processors, then to schedule the parts of the code in order to minimize the execution time while preserving the semantics. This problem is NP-complete.

Loop scheduling and processor allocation are the main problems. However we are also able to deal with so-called control parallelism. Allocation and scheduling are performed at compile time. For a given processor allocation, we use list scheduling algorithm to compute the elapsed time, which is then optimized by the Tabu heuristic.

The estimationn of each component execution time is based on knowledge of the average execution time of the operators and built-in functions and on the estimation of iteration space size.

Experimentations on the Encore-Multimax machine show that on a representative set of scientific programs, the efficiency we obtained is in almost all the cases greater than 80%, as soon as the problem size is large enough.

1 INTRODUCTION

Automatic parallelization of Fortran programs is an active research area. The reason is the need of a powerful parallel programming environment for the efficient use of multiprocessor computers. Such an environment should allow the programmer to abstract from the architecture details of the machine, and would be especially useful for large scientific programs. These programs are often written in Fortran.

Automatic detection of the parallelism in a program is only the first issue in this direction. Actually, to efficiently execute a program on a multiprocessor machine one has also to find a schedule that makes good use of this parallelism. In this paper we address the second issue.

We suppose that the programs are well structured. They do not involve `goto` statements nor `while` loops. The program can only use: assignement statements, guarded assignment statements and do loops. If necessary the restructuration of the program should be done in a previous step. Even though these assumptions are strong, the generalization of our method would be possible if combined with profiling techniques. This issue is not studied in this work.

Scheduling can either be done at run time (dynamic scheduling) or at compile time (static scheduling). In the former case the overhead can be considerable, while in the latter case the major problem is the lack of knowledge of the execution time of each component in the program. This ignorance is mainly due to the presence of variables in the loop bounds.

There are different ways to execute a parallel program on a multiprocessor machine, depending on how we assign and schedule the available processors. We have to find a solution that gives the shortest completion time of the whole program.

The target machine architecture can be any multiprocessor with shared memory. The shared memory parallel processors model is actually very suitable for the grain of parallelism we are intending to deal with. The low communication cost, compared with the distributed memory model, determines our choice.

We assume that as a result of the dependence analysis, the program is split into many components. Precedence constraints may hold between these components. Any execution of the program must respect these constraints, otherwise the semantics may be altered.

Assuming that the number of available processors is p, our approach consists in allocating to each component of the program some number n of processors, with $n \leq p$, estimating its execution time and constructing a schedule. The generated program is a set of p processes. In each one, synchronization primitives insure that the precedence constraints are respected as well as the serial execution of sequential loops. All the steps in this approach are performed at compile time. In fact, since our approach is quite costly, it is justified only if the added cost can be amortized on several executions of the same program.

*This work is a part of the PAF automatic parallelizer project, supported by DRET under contract 87/280 and by PRC C^3 of the CNRS

ICS '92-7/92/D.C., USA
© 1992 ACM 0-89791-485-6/92/0007/0063...$1.50

Processor allocation and scheduling are both NP-complete problems. We have used heuristics to solve them. To solve the processor allocation problem we tested simulated annealing [Laar 87] and Tabu search method [Glover 85] and found the latter to be more efficient. The list schedule heuristic finds a good schedule of the components when the processor allocation is given [Coffm 76], [Thoma 74].

The main results of this approach are: first, we are able to parallelize the whole program rather than be restricted to deal with each loop nest separately. Second, we have the possibility to deal with more general nested loops, at compile time, than standard techniques. The parallelism can be exploited even if it is nested within sequential structures. Another result is the estimation of the execution time of a loop nest. This is based on the symbolic computation of the loop iteration count of the nest.

The estimation of the execution time in a program involving conditionals is beyond the scope of this work. The presence of conditionals may decrease the efficiency of our target program. The handling of procedure call is not addressed here. We can deal with procedure calls only if we have already an interprocedural data-dependence analysis, and if we have an estimation of the execution time of each call.

Related work is presented in Section 2. Section 3 gives an overview of the whole method. In Section 4, we investigate a method to schedule a loop nest and compute its execution time on p processors. Section 5 presents the Tabu heuristic which is used to solve the processor allocation problem. Section 6 briefly discusses the code generation. Section 7 presents the results of experiences on the Encore machine. Conclusions are drawn in Section 8.

2 RELATED WORK

Related work address, two major issues: run-time scheduling and compile-time scheduling.

Self-Scheduling [Tang 86] is used for the execution of parallel loops. This method consists in distributing the iterations of a `doall` loop to all the processors at run time. The first iteration of the parallel loop not yet executed is assigned to the first available processor. The problem with this approach is the overhead at run time due to the synchronization on the loop iteration variable. In *Guided Self-Scheduling* [Polyc 87], Polychronopoulos tried to reduce this overhead. When a processor is available, more than one iteration is assigned to it. More precisely this method assigns to each free processor $\lceil \frac{R}{p} \rceil$ iterations, where R is the number of loop iterations not yet executed nor allocated, and p is the total number of processors. The number of exclusive accesses to the loop iteration variable is reduced but the overhead still exists.

Compile time scheduling of loops on multiprocessor has been studied in [Polyc 89]. The approach is based on allocating the processors to nested loops according to how efficiently these processors would be used. The limitation of this approach is that the bounds of the loops must be constant and the loops perfectly nested.

Our approach, globally, is rather similar to [Sarka 86]. Sarkar proposed a method to partition and schedule parallel programs. The main idea is to split the program into sequential tasks. This technique is addressed to single assignement languages. Our approach is different in the way we deal with nested parallelism, i.e. sequential loops that embed parallelism.

In [Sarka 89], the estimation of average program execution time is studied. The proposed method is based on profiling the program. This is especially useful when the source code contains conditional statements and/or while loops.

A scheduling algorithm is presented in [Belkh 91]. The program is presented as a directed task graph, where each task can be executed on more than one processor. The problem consists in determining the processor allocation that minimizes the completion time of the program under the precedence constraints. It is solved as a linear optimization problem where each variable represents the number of processors allocated to a specific task. The authors assume that the sequential execution time of each task is given as well as the variation with the number of processors. On the other hand the linear program is solved in the domain of reals, and the result for each variable is the nearest integer value.

3 PRESENTATION OF THE METHOD

Given a graph representation of a sequential program as constructed by a parallelizing compiler and the number of processors of a shared memory machine, our aim is to generate the semantically equivalent parallel program. The execution time of the parallel program must be as short as possible. This section gives an overview of all the steps performed in order to achieve this goal and how they are linked together.

The result of the dependence analysis is the dependence graph `DG` where each node is a statement and each edge is a precedence constraint that holds between two statements. From this graph a directed acyclic graph (`DAG`) is then constructed. The nodes of this `DAG` are the strongly connected components of the DG (the π-blocks in Kuck's terminology). Each component is either a single statement, a sequential loop or a `doall` loop. Each loop body has the same form as the whole program and has its own `DAG`. The `DAG` representing the program is a set of components $E = \{c_1, c_2, \ldots, c_n\}$ and a set of edges connecting the components. The edges correspond to precedence constraints that must be respected in order to preserve the semantics of the original program.

Since most components can be executed on more than one processor, the problem of scheduling E on p processors is different from the classical scheduling problem of a set of sequential tasks with precedence constraints. However, list scheduling can be used if we know how many processors are assigned to each component and the resulting execution time on the assigned processors.

Let S denote a possible allocation of processors to E: S associates each c_i with a number p_i of processors, such that $1 \leq p_i \leq p$. We allocate and free all the p_i processors to c_i at the same time.

Recall that, each component c_i is either a loop or a single statement. We will show how to estimate the execution time of c_i on p_i processors. This time will depend on the syntactic structure of c_i (e.g. whether it is a sequential or parallel loop), on the loop bounds and, ultimately, on the execution time of elementary operations and intrinsic functions (`SQRT`, `COS`, `SIN`, etc), which is determined empirically. From this we may deduce the cost function of the allocation: $C(S)$ is the execution time of the given program provided that the processor allocation is S.

The enumeration of all possible solutions, in order to find the best one, is not practical, since the total number of different allocations is p^n. This number can be reduced if we eliminate all the allocations that keep idle one

or more processors. We will consider only allocations such that $p_i \leq \max(c_i)$, where $\max(c_i)$ is the maximum number of processors that c_i can use. The number of the acceptable allocations remains very large. The use of a heuristic search method is the only acceptable approach.

The method can be briefly described as follows:

1. Preprocessing step:
   - For each loop nest compute the size of the iteration space. This computation is performed symbolically. However, the loop bounds are constrained to be linear functions on the surrounding loop iteration variables and on some other parameters.
   - Get (from the user) the values of the variables in the loop bounds that are not loop iteration variables.
   - For each component c_i compute $\max(c_i)$. These numbers are computed by simulating the execution on an unlimited number of processors. Actually they are used as a characterization of the components inherent parallelism.
2. Repeat until a good enough solution is found (according to the cost function).
   (a) select an allocation of processors $S = \{(c_1, p_1), \ldots, (c_n, p_n)\}$. This selection is guided by the processor allocation heuristic.
   (b) For each component c_i compute the execution time on the p_i allocated processors specified by S.
   (c) Schedule S on p processors using the list-schedule algorithm

The way step (b) is performed deserves more explanation: If c_i represents a single statement then c_i is executed on one processor, and the computation of its execution time is based on the average execution time of the operators and functions that are involved in the statement. On the other hand, if c_i represents a nest of loops then the loop scheduling method is applied; this method will be explained later on. We just give here a brief presentation: we consider the outermost perfectly nested loops and we decide how many processors among the p_i have to be allocated to the body, and how many are going to be used by the iterations of the parallel loops. Then the body is recursively scheduled and its execution time is computed. The execution time of c_i can then be computed using the size of the iteration space.

The result of the above steps is a recursive data structure which associates each c_i with a tuple (p_i, t_i, d_i), where p_i is the number of processors computed by the allocation heuristic, t_i is the start execution time and d_i is the estimated execution time of c_i, t_i and d_i are computed by the list-schedule algorithm. These informations are recursively computed for all the bodies of the perfectly nested loops, represented by DAGs. These values are then used to generate the high-level parallel code.

As an example of programs we can deal with, we give an abstract parallel program. Which is generated after the parallelism detection step from Cholesky triangularization algorithm. (Here p-begin ... p-end denotes a region in which all the components can be executed simultaneously).

Example 3.1

```
      program cholesky
      integer i, j, k
      real x(100), s(100, 100)
      real a(100,100), P(100)
            p-begin
C     component C1
            doall  i = 1, n
              x(i) = a(i, i)
            end do
C    component C2
            doall  i = 1, n
              doall  j = i+1, n
                s(i, j) = a(i, j)
              end do
            end do
          p-end
C         component C3
          do  i = 1, n
            p-begin
C             component C3-1
              do  k = 1, i-1
                x(i)=x(i) -a(i,k)**2
              end do
C             component C3-2
              doall  j = i+1, n
               do  k = 1, i-1
                s(i,j) = s(i,j) - a(j,k)*a(i,k)
               end do
              end do
            p-end
C           component C3-3
            p(i) = 1.0/sqrt(x(i))
C           component C3-4
            doall  j = i+1, n
              a(j,i)=s(i,j)*p(i)
            end do
          end do
      end
```

4 LOOP SCHEDULING

Since parallel loops represent the most profitable parallelism a special attention has been paid to their execution. The underlying idea of our method is to consider the outermost perfectly nested loops. If the parallel loops are not contiguous, a first transformation is applied; it consists in pushing all the outer parallel loops at the level of the innermost parallel one. The iterations of all the parallel loops are then distributed over the processors. Only one synchronization primitive is invoked after an execution of the whole parallel nest. The body of the loop nest is processed in the same fashion as the whole program. Therefore, we have to find out how many processors must be allocated to this body and how many must be dedicated to the parallel iterations.

In order to estimate the execution time of the nest, iteration count of the nest must be known as well as the execution time of the body. The last value is recursively computed.

The problem of counting the number of loop iterations has an obvious solution when the loop bounds are constant. It is much more difficult when the expressions of the bounds are linear functions on the surrounding loop iteration variables and in some other parameters. This issue is discussed in the following section.

4.1 COUNTING LOOP ITERATIONS

Let L be a nest of loops, $L = (L_1, L_2, \ldots, L_n)$, let i_i be the loop iteration variable of loop L_i and l_i (resp. u_i) be the lower (resp. the upper) bound of L_i. Assume that the loop bounds are constant, M_i is the number of iterations of loop L_i, $M_i = u_i - l_i + 1$ if $u_i - l_i \geq 0$ otherwise M_i is zero.

If the loop bounds are constant, then the total number of different executions of the body B_i of the nest $L_1, \ldots, L_i$ is $N_i = \prod_{j=1}^{i} M_i$. Otherwise, if the following holds

$$\forall j, 1 \leq j \leq n, \forall (i_1, i_2, \ldots, i_{j-1}) \text{ we have } u_j - l_j \geq 0 \quad (1)$$

then

$$N_i = \sum_{i_1=l_1}^{u_1} \cdots \sum_{i_i=l_i}^{u_i} (1) \quad (2)$$

We can associate each loop nest with a bounded convex polyhedron. An iteration of the loop nest corresponds to an integer point of the polyhedron. The problem of counting the number of integer points in a bounded convex polyhedron is known to be of high complexity. We can find in [Tawbi 91] the presentation and the proof of an algorithm that computes an approximative value of this number. The estimated value is exact when the bounds of loop iteration variables are linear functions. When the expression of a bound involves *ceiling* or *floor* functions, the algorithm finds a value which is very close to the exact one. This case occurs after loop normalization or polyhedron splitting (cf the following). We give here a brief description of this algorithm.

To compute the value of loop nest iterations number, two steps are performed. The first one splits the whole iteration space of the loop nest in an equivalent set of nests, such that in each nest condition (1) holds. The second algorithm performs the computation of symbolic sums as in (2). The result of this computation is a polynomial which is a function of the parameters. Parameters are the variables involved in the loop bounds that are not loop iteration variables.

The first step is performed by splitting the convex polyhedron which is associated with the iteration space, into a partition of convex polyhedrons. Each one represents a nest of loops in which condition (1) holds. We have developped and proved an algorithm that performs the splitting step. The method is based on keeping for each variable only one upper bound and one lower bound. If one can derive another bound by variable elimination then splitting the polyhedron would be necessary. More details about this algorithm can be found in [Tawbi 91].

The computation of symbolic sums is based on the following formula [Spiegel 74]:

$$\sum_{i=1}^{n} (i^k) = \frac{n^{k+1}}{k+1} + \frac{1}{2} n^k + \frac{B_1 k n^{k-1}}{2!} - \frac{B_2 k(k-1) n^{k-2}}{4!} + \ldots \quad (3)$$

where k and $n \in I\!N$; The power of n in the last term is either 1 or 2 depending on whether k is even or odd. If $n = 0$, the result would be 0. The B_j are Bernoulli numbers [Knuth 73], [Spiegel 74], defined as follows:

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

and

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

The calculation is done step by step from inside outward. At each step ons has to calculate a dum:

$$\sum_{i=l}^{u} P(i),$$

where P is a polynomial in i (and other variables) and l and u are linear in the other variables. $P(i)$ is written as a sum of powers of i; the contribution of each summand is computed according to (3). The result is a polynomial in the other variables, which must be reordered for the next step.

Example 4.1

```
      do 10 i = 1, n
       B1
       do 10 j = 1, i
        B2
        do 10 k = j, m-2
         B3
10     continue
```

N_3 is the number of the different executions of B_3. The splitting step would put N_3 in the following form:

$$N_3 = \begin{cases} \sum_{i=1}^{n} \sum_{j=1}^{i} \sum_{k=j}^{m-2} (1) & \text{if } m-2 \geq n \\ \sum_{i=1}^{m-2} \sum_{j=1}^{i} \sum_{k=j}^{m-2} (1) + & \\ \sum_{i=m-1}^{n} \sum_{j=1}^{m-2} \sum_{k=j}^{m-2} (1) & \text{otherwise} \end{cases}$$

the symbolic sum step yields :

$$N_3 = \begin{cases} \frac{m \times n^2}{2} + \frac{m \times n}{2} - \frac{n^3}{6} - n^2 - \frac{5n}{6} & \text{if } m-2 \geq n \\ -\frac{m^3}{6} + m^2 + \frac{m^2 \times n}{2} - & \\ \frac{11m}{6} - \frac{3m \times n^2}{2} + n + 1 & \text{otherwise} \end{cases}$$

The values of the parameters are not required to perform the previous steps (including dependence analysis), however they are required for the scheduling step. When the values of parameters are known, the N_i will have constant values.

[Shrij 86] is a good reference for the theory of linear programming and the manipulation of convex polyhedron.

4.2 SCHEDULING NESTED LOOPS

The method consists in considering the first perfectly nested loops in the nest. If we know how many processors are assigned to the body of these loops, we can recursively schedule it as we do for the whole program. If the body involves loops whose bounds depend on the surrounding loop iteration variables, the size of the nested iteration space is a function on the surroundig loop variables. In this case, the computation of the execution time is based on the average number of iterations.

In example (3.1), the loop:

```
      do k =1, i-1
```

which corresponds to component C3-1 is within an imperfectly nested loop. Its average number of iterations, when n=100 is 50. This number is used to compute the body average execution time of do i = 1, n (component C3).

If the body of the first perfectly nested loop is sequential then all the processors are assigned to the iterations of these loops. Otherwise, one has to decide how many processors have to be assigned to the first loops and how many to the body.

Let L be the loopnest, $L = (L_1, L_2, \ldots, L_n)$ and B be the body represented by a directed acyclic graph. We assume that the completion times of two consecutive different executions of B are approximatively the same.

4.2.1 SCHEDULING A NEST OF SEQUENTIAL LOOPS

If all the loops in the nest are sequential, the p assigned processors are allocated to B. Scheduling B on p processors is the same problem as scheduling the whole program and it is solved recursively.

The execution time $T(L,p)$ of the nest is $T(L,p) = N_n \times T(B,p)$, where N_n is the total number of iterations of the nest and $T(B,p)$ is the execution time of B on p processors.

4.2.2 SCHEDULING A NEST OF PARALLEL LOOPS

In this case, the nest of loops has the following form:

```
      doall 100 i1 = l1, u1
         ...
         doall 100 in = ln, un
            B
100   continue
```

We assume in the following that the parallelism in the body is not profitable. We study in the last subsection the allocation of processors to the body.

The processors are assigned to the n doall loops. If all the iterations of the nest are numbered from 0 to $N_n - 1$ and the p processors are numbered from 0 to $p-1$, then our method consists in excuting iteration k on the processor $q = k \bmod p$. A first sketch of the code for processor q is the following:

```
      k = 0
      Do 100 i1 = l1, u1
         ...
           Do 100 in = ln, un
             if k mod p = q then B
             endif
             k = k + 1
100   continue
      barrier
```

The barrier instruction after the parallel loops is a sychronization primitive, the role of which is to wait until all p processors have reached this point before proceeding to the rest of the program.

The use of the if statement can be avoided. The above program can be shown to be equivalent to the following more efficient version.

```
      k0 = 0
      Do 100 i1 = l1, u1
         ...
           Do 10 in = ln + (q - k0) mod p, un, p
             B
10         continue
           if (un - ln + 1 > 0) k0 = k0 + un - ln + 1
100   continue
      barrier
```

The execution time of this nest is:

$$T(L,p) = \lceil \frac{N_n}{p} \rceil \times T(B,1) + N_{n-1} \times T_{s_c} + T_{s_b}$$

where N_i is the number of the different executions of the body of the nest $(L_1, L_2, \ldots, L_i)$, $T(B,1)$ is the sequential execution time of B, T_{s_c} is the cost of incrementing k_0 and T_{s_b} is the cost of the barrier.

Experiences show that $T(B,1)$ is approximatively the same for two consecutive iterations. Hence, the distribution of the loop iterations in our method is nearly optimal.

As an example let us consider the following nest:

```
      doall 1 i = 1, n, 1
       doall 1 j = i+1, n
        s(i,j) = a(i,j)*b(i,j)
1     continue
```

If we suppose that $n = 100$ and the number of the processors allocated to this nest is 7 the processor number 3 would execute the following code:

```
      k_0 = 0
      Do 200 i = 1,100,1
          Do 100 j = 1+i+ mod(3-k0,7),100,7
              s(i,j)= a(i,j)*b(i,j)
100       continue
          if (100-i > 0) k0 = k0 + 100-i
200   continue
      barrier
```

Assuming that the execution time of the statement s(i,j) = a(i,j)*b(i,j) is $4u$, the cost of incrementing k_0 is $3u$, and the cost of calling the barrier is 15u then the execution time $T(L,7) = (708 \times 4 + 100 \times 3 + 15)u = 3147u$

4.2.3 SCHEDULING A NEST OF MIXED LOOPS

In this case the parallel loops are pushed inside the sequential ones at the level of the innermost parallel loop. In this way, we get a contiguous parallel loop nest and use all the parallel loops instead of assigning the processors to one of them. It has been shown in [Allen 84] that this transformation preserves the semantics of the program. Thus, the most general case is that of m sequential loops surrounding $n-m$ parallel loops. The body of the parallel loops may be a nest of sequential ones.

```
      Do 100 i1 = l1, u1
        Do 100 i2 = l2, u2
          ...
           Do 100 im = lm, um
             //Do 100 im+1 = lm+1, um+1
                ...
                 //Do 100 in = ln, un
                        B
100     continue
```

If we assume that the parallelism in the body B is not profitable then the processors are used by the parallel loops and B is executed sequentially.The code executed by the processor q is the following:

```
      Do 1000 i1 = l1,u1
       ...
        Do 1000 im = lm,um
             k0 = 0
             Do 100 im+1 = lm+1, um+1
                ...
                  Do 10 in = ln + (q - k0) mod p, un, p
                    B
10                  continue
                    if (un - ln + 1 > 0) k0 = k0 + un - ln + 1
             barrier
1000    continue
```

Let N_p be

$$N_p = \sum_{i_{m+1}=l_{m+1}}^{u_{m+1}} \cdots \sum_{i_n=l_n}^{u_n} (1).$$

Clearly N_p represents the number of parallel loop iterations; this number is a function on the surrounding loop iteration variables. N_i is the number of iterations of the loop nest $(L_1, \ldots, L_i)$. If we do not take into account the overhead due to the barrier and to the counter k_0 then the execution time of the loop nest L is:

$$T(L,p) = N_m \times \left\lceil \frac{N_p}{p} \right\rceil \times T(B,1)$$

where $T(B,1)$ is the execution time of B. Notice that if B is a nest of sequential loops, the execution time of B may depend on the surrounding loop iteration variables. In this case the average execution time is taken into account.

If $N_p \gg p$ and if $T(B,1)$ is approximately the same for two consecutive iteration of the nest, then this method gives nearly optimal results. Actually $\lceil \frac{N_p}{p} \rceil$ would be very close to $\frac{N_p}{p}$ and thus the speed up would be very close to p. Otherwise, one has to select the level into which to push the parallel loops. Since we deal with a relatively small number of processors and with scientific programs the above conditions are met in allmost all the cases.

4.2.4 SCHEDULING LOOPS WITH PARALLEL BODY

The body B of the perfectly nested loop L contains parallelism and L involves parallel loops. Let N_p be the average number of the parallel loop iterations and p be the number of the processors assigned to L . If $N_p \gg p$ then the p processors are allocated to parallel loops. More precisely, if we look to the expression $T(L,p)$ we see that the speed up depends mainly on $\frac{\lceil \frac{N_p}{p} \rceil}{\frac{N_p}{p}}$: the bigger N_p and the closer to 1 is the efficiency. To get an idea, suppose for instance that N_p is 200 and p is 11. The above ratio is 1.04 which garantees a speed up not less than $95\% \times p$. Thus if N_p is big enough it is not worth the cost to allocate more than one processor to the body. Since the number of processors is not very large, if $N_p \not\gg p$ one can enumerate all the factors of p and select the best one to be assigned to the body. Hence, we will have p_1 processors for the parallel loop iterations and p_2 processors to be allocated to B with $p = p_1 \times p_2$.

The processors will be organised in p_1 groups, each of p_2 processors, and each group is responsible for the execution of one iteration of the parallel loop nest.

5 PROCESSOR ALLOCATION

We are given a program P represented by a set E of components $E = \{c_1, c_2, \ldots, c_n\}$ connected by precedence constraints, and p processors to execute them. Each c_i represents a part of code that can be executed on more than one processor.

Let $\mathcal{S} = \{S_1, S_2 \ldots, \}$ be the set of all possible processor allocations to E. A specific allocation S_j associates each component c_i with a number of processors p_i, $1 \leq p_i \leq max(c_i)$ and $p_i \leq p$. Thus, we eliminate all the allocations that keep idle one or more processors. A cost function C associates a solution S with the execution time, which is computed by the list-scheduling algorithm applied to E. When the allocation is S and if we know the execution time $T(c_i, p_i)$ of each component c_i on the p_i allocated processors, $C(S)$ can be determined.

Our aim is to perform an exploration of $\mathcal{S}$ in order to find an allocation S^* that gives a schedule for E which is very close to the shortest one, i.e. $C(S^*)$ is close to the global minimum of C. Actually, it is very hard to find the best allocation because it has been proved that this problem is NP-complete [Coffm 76].

One can imagine an iterative descent method to explore $\mathcal{S}$. This approach should define a neighborhood $N(S)$ for each solution S. The procedure would start, then, from an initial solution S_0. One move consists in finding a new solution S' in the neighborhood $N(S)$ of the current solution S, with $C(S') < C(S)$. The procedure stops if such an S' cannot be found in $N(S)$. Obviously, this method leads to a local minimum. Besides, it depends on the definition of $N(S)$.

The aim of any heuristic, that intend to find better solutions than the descent method, is to do the exploration of $\mathcal{S}$ while avoiding to be trapped in local minima. To achieve this goal two heuristics have been studied: the simulated annealing and Tabu. Since we have found that Tabu is consistently better and faster than simulated annealing, only the latter method is presented here.

5.1 TABU METHOD

Tabu method is a heuristic used to solve hard optimization problems. It was first introduced by Glover [Glover 85] and then used to solve many optimization problems [Glover 89], [deWer 89]. Tabu consists in the exploration of the solutions set, starting with an initial solution S_0. A move from S to S' is allowed even if $C(S') \geq C(S)$. If we are at a solution S, a subset of $N(S)$ is generated, and the best S' in this

subset is then selected. Cycling may of course occur. In order to prevent that, we are not allowed to go back to a solution which has been visited in the last k iterations. A list T, called the Tabu list, of the last k solutions is updated after each move. Thus, a move to S' is allowed only if S' is not in the Tabu list.

Tabu results depend on the choice of certain parameters: the neighborhood function, the enumerated subset in the neighborhood of a solution, the length $|T|$ of the tabu list, the initial solution and the stop criterion.

In our implementation $|T|$ is fixed to 7, and the search stops if in the last 7 iterations we did not improve the best solution yet encountered. The neighborhood of an allocation S is the set of all allocations that associate the same processors to the c_i as S does, except for one component, say c_j. In this neighborhood we enumerate at most n solutions, $(n = |E|)$; each solution S'_j $(j = 1, \ldots, n)$ is associated with specific $c_j \in E$, it keeps the same allocations as S, except for c_j, to which a random number of processors p'_j is associated, $p'_j \neq p_j$, where p_j is the number of processors allocated by S. We denote this subset $V(S)$. The choice of these parameters was made empirically.

At each iteration, $C(S_k)$ is the result of the list schedule algorithm, performed after the computation of the execution times of the components.

The implemented algorithm is the following:

1. start with $k = 0$ and the allocation S_0 that associates to each c_i the maximum allowable number of processors. $T = \{S_0\}$; $S^* = S_0$; $n = 0$.
2. while $n <= 7$:
   (a) $k = k + 1$
   (b) generate $V(S_{k-1})$
   (c) move to S_k which gives the minimum cost $C(S_k)$ among the elements of $V(S_{k-1}) - T$.
   (d) if $C(S_k) < C(S^*)$ then $S^* = S_k$; $n = 0$ else $n = n + 1$;
   (e) put S_k at the head of the list T. If $|T| > 7$ remove the last element of T.
3. the result is S^*

For instance, the allocation on 6 processors for the code of example (3.1) is:

```
C1:6, C2:6, C3:6, C3-1:1, C3-2:5, C3-3:1,
C3-4:6.
```

It was computed by Tabu in 14 seconds on a DPX-1000 station (which is as fast as a SUN3/60).

Tabu method has been successfully used to solve hard optimization problem: job shop scheduling, graph coloring, the travelling salesman problem, ... Our experience does not contradict these results. However, there is no formal convergence proof.

6 CODE GENERATION

The generated parallel code for a program is a set of p processes, where p is the number of processors on which the program will be executed. The code is generated for an Encore Multimax machine with 8 processors and a shared memory.

The serial character of sequential loops execution and the precedence constraints between components must be respected. For this, we use user mode synchronisation primitives in the process code.

The components of the program are sorted according to t_i which is the start execution time. If we assign one processor to each component then the code generation will be a greedy algorithm that places the code of a component in the first available processor (in the code of the correspondig process). After this code, an `event-post` is issued and the corresponding `event-wait` is placed before the code of all of its successors in the DAG representation.

If the component c_i has to be executed on $p_i > 1$ processors then its code is distributed over the first p_i available processors and a `barrier` synchronization is invoked after each part of the code to insure that all p_i processors will be freed at he same time. One of the p_i processors has to invoke an `event-post` just after the barrier call. The `event-wait` is called in every process executing a part of a component which is a successor of c_i.

7 RESULTS

In this section we present the results of the experimentations on some numerical programs. We did not study the optimization in a multi-user environment. We think that dynamic methods would be more suitable in that case. Hence, all our tests are executed under the *gang-scheduling* mechanism which allows the possibility to assign a number of processors to a specific application. At least one processor should be left out of the gang to deal with interruptions and emergencies. That is the reason why we do not give results for more than 7 processors.

We give first a brief description of the programs, and the difference between the estimated and the observed time of the sequential and parallel execution. We then discuss the variation of the efficiency as a function of the problem size and of the number of processors.

7.1 SOURCE PROGRAMS

The programs we parallelize are representative of a class of numerical analysis programs. They do not involve conditional constructions. The data structures they deal with are arrays.

We will give the results of the experimentations on the following programs:

- Matrix product (Prod).
- Gaussian elimination and backward substitution phases (Gauss)
- Cholesky matrix triangularization (Choles)
- Lanczos projection phase (a component of an eigenvalue method [Chate 88]) (Lanczos)
- Prr is another projection method [Emad 91] (Prr)
- Arnoldi iterative method [Saad 80] (Arnoldi)

All the sequential programs have the following form:

```
Sequential Program
 Read data
 t1 = clock-time
 Process the data
 t2 = clock-time
 print (t2 - t1)
 write the results
```

The parallel programs have the following form:

```
Parallel Program
 Read data
 fork p processes
  i is  the process number
  t1 = clock-time
  call process i body
 join
 t2 = clock-time
 print (t2 -t1)
 write the results
```

Input and output time are not taken into account. The reason is that our scheduling method only optimizes the cpu time.

7.2 ESTIMATED AND ACTUAL EXECUTION TIME

The execution time estimation is based on the average execution time of the operators and the estimation of the loop iteration number.

Since the parallelizing process translates the sequential source program into a parallel source program, the execution time estimation does not take into account the optimizations that the compiler would perform.

In the following table we give E_1 and E_6 values where $E_1 = 100 \times abs(\frac{d_1 - d'_1}{d_1})$, d_1 is the actual sequential execution time and d'_1 is the estimated sequential time. $E_6 = 100 \times abs(\frac{d_6 - d'_6}{d_6})$ where d_6 is the actual parallel execution time on 6 processors and d'_6 is the estimated parallel execution time on 6 processors.

Program	E_1	E_6
Lanczos	4,2	1,2
Prr	8,0	6,0
Arnoldi	0,8	2,6
Choles	2,7	5,2
Gauss	9,2	12,4
Prod	5,6	9,6

One can observe that the error is rather small, except for Prr and Gauss programs. This is mainly due to compiler optimizations. Actually in the text of these two programs there are many obvious optimizations that the compiler can perform.

Based on these results, we can extend our observations beyond the range of our target machine. We can estimate for instance the parallel execution time on more than 8 processors and rely on the results.

7.3 THE EFFICIENCY AND THE SIZE OF DATA

The efficiency of a parallel computation is defined as $R = 100 \times \frac{d_1}{p \times d_p}$, where d_1 is the sequential execution time and d_p is the parallel execution time on p processors. For the following experimentations we used 6 processors. The size of data is represented by the dimension of the matrix which is given in the first row. A number in a specific box represents the efficiency of the parallel program when run on 6 processors on a matrix of the given dimension.

	500	400	300	200	100	50
Lanczos	94	93	92	93	87	66
Prr	96	96	96	97	96	86
Arnoldi	91	91	92	91	88	78
Choles	94	93	92	90	78	58
Gauss	96	95	95	93	88	80
Prod	100	100	100	102	100	99

The efficiency increases with the size of the matrix and is nearly always greater than 80%. In the case of the matrix product, we obtain anomalous values. The probable explanation is the following. When comparing execution time on one and six processors, we are not really measuring the same algorithm, since the six processors execution has a much larger cache space than the one processor execution. Usually, this effect is masked by the incidence of synchronization and by coherence problems. For the matrix product, there are very few synchronization and no coherence problem. The program may thus make full use of the increased cache space, hence the anomalous values.

7.4 THE EFFICIENCY AND THE NUMBER OF PROCESSORS

The efficiency is studied here according to the number of processors. All the programs have been run on a 200 x 200 matrix.

The number of processors is given in the first row. The number in a specific box is the efficiency of the program when run on the indicated processors.

	1	2	3	4	5	6	7
Lanczos	100	97	98	96	93	93	90
Prr	100	97	97	97	97	97	96
Arnoldi	100	96	93	93	93	91	91
Choles	100	98	96	94	92	90	89
Gauss	100	97	95	94	94	93	92
Prod	100	103	102	102	102	102	101

The efficiency decreases with the number of processors. The results are good, almost all are over 90%. The same remark as in the last section can be made concerning the Prod program.

8 CONCLUSION

The main result of our approach is that we are able to parallelize the whole program rather than individual nests. Parallelism is exploited at different nesting levels. The experiences show that we get good poerformances.

The experimentation with the allocation processor heuristics shows that the solution S_m that assigns to each component the maximum number of processors it can use is always nearly optimal. This observation leads us to concentrate the research of the optimal solution in the neighbourhood of S_m and thus to be much faster.

Some steps of our method can be performed without knowledge of the *parameters* values, which represent the data size. The processor allocation which uses the list scheduling algorithm needs this information. One can imagine a solution that associates all the processors to the parallel loops. The use of a sophisticated allocation method

is obviously needed when the number of the processors is of the same magnitude as the number of parallel iterations. We can use this result in order to completly parametrize the method.

Another important result is that when we increase the number of processors, the search time does not increase.

The computation of the iteration space volume of a nested loop can be used in other contexts: the estimation of the volume of transfered data in a hierarchical or distributed memory architecture. Another use can be the complexity analysis of programs. The limitation is the over-estimation of execution time for the components which contain conditionals. Another limitation is that we do not deal with procedure calls. For this, the side effect informations are not sufficient. The estimation of the execution time of the procedure call on a specific site is needed.

REFERENCES

[Allen 84] Allen J. R., Kennedy K. *Automatic loop interchange* Proceedings of Sigplan 84, Symposium on compiler construction, Montreal Canada June 17-22, 1984

[Belkh 91] Belkhale K.P., Banerjee P., *A Scheduling Algorithm for Parallelizable Dependent Tasks*, Proceedings of the 5th International Parallel Processing Symposium, April 1991.

[Chate 88] Chatelin F., *Valeurs propres de matrices*, Masson 1988.

[Coffm 76] Coffman E. G. *Computer and Job-sop scheduling Theory* John Wiley and sons, 1987

[deWer 89] de Werra D., Hertz A. *Tabu Search Techniques: a tutorial and an application to neural networks.* Ecole Polytechnique Fédérale de Lausanne, ORWP 89/02, January 1989.

[Emad 91] Emad N., *A new Iterative Projection Method for Large Symmetric Eigenproblem* Yale Technical Repport YALEU/DCS/RR-872, 1991.

[Glover 85] Glover F. *Future paths for integer integer programming and links to artificial intelligence.* CAAI Report 85-8. University of Colorado. Boulder CO (1985).

[Glover 89] Glover F. *Tabu Search, Part I.* ORSA Journal of Computing, Vol. 1, $N^{\underline{o}}$ 3, 1989.

[Knuth 73] Knuth D. E. *The Art of Programming.* Vol. 1, Addison Wesley 1973.

[Laarh 87] Laarhoven P. J. M. Van, Aaarts E. H. L. *Simulated Annealing Theory and applications* Kluwer Academic publishers, 1987.

[Polyc 87] Polychronopoulos C. D. *Guided self scheduling A practical scheduling scheme for parallel supercomputers* IEEE Transactions on Computers, vol C-36, $N^{\underline{o}}$ 12, December 1987

[Polyc 89] Polychronopoulos C. D., Kuck D. J., Padua A. P. *Utilizing Multidimensional Loop Parallelism on Large-Scale Parallel Processor Systems* IEEE Transactions on Computers, vol 38, $N^{\underline{o}}$ 9, September 1989

[Saad 80] Saad Y., *Variations on Arnoldi's method for computing Eigenelements of Large Unsymmetric matrices,* Lin. Alg. Appl., vol. 34, pp. 269-295, 1980.

[Sarka 86] Sarkar V., Hennessy J. *Compile-time partitioning and scheduling of parallel programs* ACM Sigplan Notices, vol 21, $N^{\underline{o}}$ 7, July 1986

[Sarka 89] Sarkar V., *Determining Average Program Execution Times and their Variance* Proceedings of the ACM Sigplan Conference PLDI 89, Portland.

[Schri 86] Schrijver A. *Theory of linear and integer programming.* Wiley, NY, 1986.

[Spiegel 74] Spiegel M. R. *Formules et tables de Mathématiques.* Mc Graw Hill Paris, Série Schaum, 1974.

[Tang 86] Tang P., Chung Yew P. *Processor Self scheduling for multiple nested parallel loops* Proceedings of the 1986 International Conference on parallel processing 19-22 August 1986

[Tawbi 91] Tawbi N., *Parallélisation Automatique: Estimation des Durées d'Exécution et Allocation Statique de Processeurs*, Ph. D. Thesis of Université Pierre et Marie Curie, April, 1991.

[Thoma 74] Thomas L. Adam, Chandy K. M., Dikson J. R. *A comparison of list schedules for parallel processing systems* Communications of ACM, vol 17, $N^{\underline{o}}$ 12, December 1974