Asymptotically Efficient Algorithms for Parallel Architectures *

Paul Feautrier
Laboratoire MASI
Université P. et M. Curie
75252 PARIS CEDEX 05 FRANCE

July 25, 1996

Abstract

This paper gives a general method for the construction of parallel algorithms. Starting from a conventional sequential program, one first constructs a timing function, i.e. a schedule for a paracomputer. It is shown that the ratio of the maximum value of the timing function to the total operation count is a good measure of the degree of parallelism in the original algorithm. In particular, if this ratio tends to zero when the operation count grows large, then there is an asymptotically efficient parallel version of the original algorithm. This implementation is shown to be surprisingly robust in the face of variations, random and otherwise, of the operation execution times. The technique may be used as the starting point of the construction of programs for all kinds of parallel computers: vector, synchronous, asynchronous and distributed architectures.

1 Introduction

Many people have argued that for building programs for parallel computers, some knowledge of the execution time of tasks is necessary. However, experience has shown that one gets quite efficient algorithms in spite of imprecise timing information ([9]). The aim of this paper is to explain this apparent paradox.

Recent research on the automatic construction of parallel programs has centered on the notion of transformation. The rationale is that most programs were written with sequential execution in mind and are not well adapted to parallel execution. Transformations were invented to remove excessive optimization : expansion ([14], [4]), loop reordering and loop splitting ([1]), algebraic expression rearrangement ([2], [10]). As always, if taken naively, the transformation approach leads to combinatorial explosion. For a given program, there is a very large number of valid transformations which interact in a complex way; depending on the target architecture, some of them improve the running time of the program and some do not. Parallelization becomes a search for a minimum in a set of semantically equivalent programs. Beside the mere size of the search space, one may be trapped in local minima.

Mathematicians have a well known method for dealing with this kind of situation : instead of working with individual programs, work with equivalence classes under the selected

*This work has been supported by DRET under contract 87/280 and by PRC C^3 of the CNRS.

transformations. To apply this strategy, one obviously need a notation for one equivalence class; the notation should be simple and compact but carry enough information to enable one to compute the required objective function and to reconstruct the selected optimal representative. This paper shows that, for a restricted class of programs (the static control programs of [4]), the dataflow graph as (DFG) defined in [5] answer at least part of our requirements.

In paragraph 2, we give a more precise definition of static control programs; we review the methods of [5] for computing their DFG. We then introduce the notion of timing function. A timing function simply is a representation of a schedule for a computer with an unllimited number of processors and no memory conflicts. J. T. Schwartz ([17]) has coined the term "paracomputer" for such a machine.

In paragraph 3, we explain how to fold the paracomputer schedule to P processors. We show that the performance of the resulting algorithm is strongly related to the ratio D/S, where S is the size of the DFG and D is the length of its longest path [1]. For many practical algorithms, this ratio tends to 0 as the size of the calculation grows large; as a consequence, the efficiency of the parallel algorithm tends to 1. We shows that the resulting algorithm is surprisingly robust in the face of uncertainties in the execution time of operations. Basically, the reason is that such uncertainties introduce delays of the order of D while the running time is of the order of S/P.

From these remarks to a practical restructuring compiler is still a long way. In the conclusion, we discuss how to attack the remaining problems: construction of linear and non linear timing functions, code generation issues, memory management, extension to distributed systems.

2 Models for algorithms and computers

It seems very difficult to give interesting performance estimates for the parallel execution of arbitrary programs. There is a sharp distinction, for instance, between polynomials algorithms, non-polynomials algorithms, and programs whose termination is not guaranteed. Now parallel program construction involves, among other problems, questions of load balancing and hence performance prediction. Obviously this is easiest for polynomial algorithms, and very difficult for potentially non terminating processes. This has motivated the definition in [5] of *static control programs*, i.e. programs whose operation count may be calculated a priori as soon as the values of *structure parameters* are known.

Briefly speaking, a static control program has `for` loops as the only control instructions. The data processing instructions are simple or guarded assignments and input-output instructions. Furthermore, in a loop nest, the bounds depend only on outer loop counters and structure parameters. Each structure parameter is defined by an input instruction and is not modified anywhere else in the program.

As a matter of technical convenience, we will restrict all array indices and loop bounds to affine integer expressions, and all structure parameters to integer variables. This restriction is enforced because at the present time our main analysis tool is the parametric linear integer

[1] which also is the maximum value of the timing function.

programming algorithm of [6]. It may be possible to extend the class of tractable programs by using formal computation techniques, (see for instance [13]).

An important distinction when discussing parallel programs is the one between instructions and operations. An instruction is a static entity, which may be identified with a syntactic unit in the program text. An operation is one particular execution of an instruction in a given context. Most often, an instruction will be executed many times. Each execution will be considered as a distinct operation. In a static control program, an operation may be identified by specifying the parent instruction and the values of the surrounding loop counters or *iteration vector*. If array indices depends only on loop counters, a knowledge of an operation name allows one to compute the adresses of accessed memory cells, and hence its dependencies to other operations.

In a sequential program, operations are totally ordered in time. In a parallel program, the execution order is partial. Non-comparable operations may be executed simultaneously if sufficient ressources exists. We will postulate that the total running time of a program may be computed simply by summing the duration of elementary operations. Specifically, we will ignore phenomena like pipe-line loading and unloading, cache hit or miss, and memory contention. In fact, we will show in 3.4 how to model these effects as random fluctuations in the execution times.

2.1 The Dataflow Graph

For a static control program, it is possible to analyze the flow of data through the operations and the memory cells. The basic technique is presented in [7] and in more details in [5]; a brief description follow. For each read access in the program, the set of all preceeding write accesses to the same memory cell is characterized and its temporal maximum is computed. The result is the source of the value obtained by the read access. A source is composed of an instruction name and an iteration vector. Both these elements may depend on the iteration vector of the read access. The method of [5] yields a *source function* in the form of a more or less complicated conditional expression.

Consider for instance the following Fortran code which is a part of a Cholesky solver:

```
        program choles
        integer i, j, k
        real x(10), z(10, 10)
        real a(10,10), p(10)
        do 1 i = 1,n
        x(i) = a(i,i)                          {1}
        do 2 k = 1, i-1
2       x(i) = x(i) - a(i,k)**2                {2}
        p(i) = 1.0/sqrt(x)                     {3}
        do 3 j = i+1, n
        z(i,j) = a(i,j)                        {4}
        do 4 k = 1,i-1
4       z(i,j) = z(i,j) - a(i,j) * a(i,k)      {5}
3       a(j,i) = z(i,j) * p(i)                 {6}
1       continue
```

```
end
```

Let us consider statement {6}. This instruction is executed once for each integer point in the set

$$D_6 = \{< i, j > | 1 \leq i \leq n, i+1 \leq j \leq n\}.$$

D_6 is the *execution domain* of {6}. Each operation produces a value for `a(i,j)` and consumes one value for `z(i,j)` and one value for `p(i)`. Let us investigate, e.g., the source of the value for `z(i,j)`. There are two a priori possibilities: statements {4} and {5}. One sees that the last iteration of {6} will destroy all preceding values, *provided this instruction is executed at least once.* This is always true unless $i = 1$, in which case the source is instruction {4}. Let us note $(n, i, ..., k)$ the value which is produced by instruction `n` when the surrounding loop counters have values $i, \ldots, k$. The value of `z(i,j)` in {6} is given by the following conditional:

```
if i >= 2 then (5, i, j, i-1) else (4, i, j)
```

This reasoning may be reproduced (with suitable modifications), for all right-hand side (*rhs*) references in the program. Whe have shown in [5] that the process is completely mechanical, the basic tool being an algorithm for solving parametric linear programing problems in integers ([6]). The corresponding program has been inplemented partly in Lisp and partly in C, and runs on various computers including a Dec Vax, a Sun workstation and a personnal computer.

The result of the analysis may be presented as a graph which we call the *data flow graph* or DFG for short. The DFG has one node per instruction in the original program. There is an edge from instruction s (*the source*) to instruction t (*the sink*) for each rhs reference in t wich uses a value produced by s. Each edge is labelled by the following information:

- The *governing predicate*, which must be true for the value to be really used in the sink instruction. For the exemple above, the governing predicate for the $5 \rightarrow 6$ edge is $i \geq 2$.
- The *sink-to-source transformation* which allows one to compute the iteration vector of the source instruction in term of the iteration vector of the sink instruction. In all practical cases we have encountered so far, this transformation is affine, but there is a possibility of encountering quasi-affine transformations as the result of the source computation (see [4] for a somewhat artificial exemple). In the above case, this transformation is:

$$i' \leftarrow i, j' \leftarrow j, k' \leftarrow i - 1$$

Table 1 gives the complete DFG for the Cholesky program.

The DFG is similar to the usual dependence graph but has several advantages. The edges in the DFG correspond only to true dependences in the program; anti-dependences and output dependences are suppressed. Spurious true dependences (i. e. dependences on values which are destroyed before being used) are suppressed as well. As a consequence, the DFG is insensitive to expansion transformations ([14], [4]). The DFG is sensitive to loop inversion and loop skewing ([19]). However, this effect is very limited : it is simply a renaming of all nodes, with corresponding changes of variables in the governing predicates and source-to-sink transformations.

source	sink	reference	governing predicate	execution domain
$(2,i,k-1)$	$(2,i,k)$	x(i)	$k \geq 2$	$1 \leq i \leq n$
$(1,i)$	$(2,i,k)$	x(i)	$k = 1$	$1 \leq k \leq i-1$
$(6,i,k)$	$(2,i,k)$	a(i,k)	true	
$(2,i,i-1)$	$(3,i)$	x(i)	$i \geq 1$	$1 \leq i \leq n$
$(1,i)$	$(3,i)$	x(i)	$i = 1$	
$(5,i,j,k-1)$	$(5,i,j,k)$	z(i,j)	$k \geq 2$	$1 \leq i \leq n$
$(4,i,j)$	$(5,i,j,k)$	z(i,j)	$k = 1$	$i+1 \leq j \leq n$
$(6,k,i)$	$(5,i,j,k)$	a(i,k)	true	$1 \leq k \leq i-1$
$(5,i,j,i-1)$	$(6,i,j)$	z(i,j)	$i \geq 2$	$1 \leq i \leq n$
$(4,i,j)$	$(6,i,j)$	z(i,j)	$i = 1$	$i+1 \leq j \leq n$
$(3,i)$	$(6,i,j)$	p(i)	true	

Table 1: The DFG of the Cholesky Program

2.2 Expanded Dataflow Graph and Timing Functions

The DFG is a synthetic representation which abstracts over all values of the structure parameters. For given values of the structure parameters, we may construct a more explicit representation by the following expansion process. The expanded DFG will have one node per operation. There will be an edge from $(s,\vec{a})$ to $(t,\vec{b})$ iff there is an edge from s to t in the DFG, and if $\vec{b}$ satisfies its governing predicate, and if $\vec{a}$ and $\vec{b}$ are related by its source-to-sink transformation.

The expanded DFG has no loops; its transitive closure is a partial order $\prec$ which is coarser than the sequential execution order of the program. Let u and v be two operations. $u \prec v$ means that u produces a value which participates more or less directly in the computation of v's output value. Obviously this implies that u must be finished before v may start. There are several paradigms for extracting a parallel program from a task graph. For instance maximal chains with respect to $\prec$ are processes. Conversely, maximal antichains are wavefronts whose operations may be executed simultaneously.

In the classical scheduling approach, one tries to assign to each operation a start time and a processor number in such a way that all constraints are satisfied. As is well known, the scheduling problem is NP-complete unless the task graph is restricted to simple forms which are not likely to be encountered in practice.

Our starting point will be the so-called timing functions, which are nothing more than paracomputer schedules. A timing function should verify the following inequalities:

$$u \prec v \Rightarrow \theta(u) + \partial(u) \leq \theta(v), \tag{1}$$

where $\partial(u)$ is the duration of u.

We will be interested mainly in the case where all operations have the same duration, which may be taken as the time unit. Among all timing functions is a minimal one, which is easily computed by the following formula:

$$\theta(u) = \max\{\theta(v) | v \prec u\} + 1. \tag{2}$$

The algorithm is linear in the number of edges in the expanded DFG. The important point, however, is that in many cases we may obtain simple closed expressions for some timing function; most often, this may be done uniformly in the structure parameters.

> For instance, the reader may care to verify that the above Cholesky program admits the following timing fonctions:
>
> $$\begin{array}{ll} \theta(1,i) = 0 & \theta(2,i,k) = 3k \\ \theta(3,i) = 3i-2 & \theta(4,i,j) = 0 \\ \theta(5,i,j,k) = 3k & \theta(6,i,j) = 3i-1 \end{array} \tag{3}$$
>
> The verification is nothing more than an examination of all edges in table 1. For an edge from s to t with governing predicate π and sink-to-source transform L, verify that the inequality:
>
> $$\theta[t, \vec{a}] \geq \theta[s, L(\vec{a})] + 1 \tag{4}$$
>
> is a consequence of $\vec{a} \in D_t$ and $\pi(\vec{a})$.

The determination of timing functions is outside the scope of this paper. Let us note, however, that the problem is strongly connected to the construction of systolic arrays. In fact, from most systolic designs one may retrieve a (linear) timing function for the underlying algorithm. Conversely, a well known synthesis method ([16]), starts by the construction of a linear timing function from the DFG [2].

Another source of timing functions is the wavefront method ([12]). A wavefront is a set of independent operations. Wavefronts are hyperplanes in the iteration space:

$$F_t = \{\vec{a} | \vec{h}.\vec{a} = t\}. \tag{5}$$

Obviously, $\theta(\vec{a}) = \vec{h}.\vec{a}$ is a timing function.

One should note, lastly, that guessing timing functions for illustration purposes is quite easy. First, construct an expanded DFG and a sample timing function with the help of (2). Next, guess a simple representation of the "experimental" values. Lastly, check that the proposed timing function is compatible with the DFG. The Cholesky timing functions (3) were obtained in this way.

3 Theoretical results

We will suppose that we are given a timing function θ which satisfies (2). Let $S(n)$ be the total operation count (which may be taken as the sequential running time), and $D(n)$ be the maximum value of θ. Let us define:

$$F(t) = \{u | \theta(u) = t\}. \tag{6}$$

[2] Or rather from an equivalent representation as a set of uniform recurrence equations.

$F(t)$ is the set of operations which are executed from time t to time $t+1$ by the paracomputer, and will be called a *front*.

An obvious way to fold the paracomputer schedule into a schedule for P processors is to distribute the operations of $F(t)$ between the available processors. $F(t)$ will now be executed in a time which is roughly proportional to its size. The resulting program may be sketched as follow:

```
for t := 1 to D(n) do
begin
    execute F(t) in parallel on P processors;   {A}
    synchronize;
end;
```

It is an interesting exercise to apply the above recipe to the Cholesky program and its set of timing functions (3). There is obviously a set of initialisations which are done at time 0. Hence the program starts as [3] :

```
      PCASE
         DOALL 10 i = 1,n
   10       x(i) = a(i,i)
      PAR
         DOALL 11 i = 1,n
            DOALL 11 j = 1+1,n
   11          z(i,j) = a(i,j)
      END PCASE
```

An examination of the timing functions (3) shows that the contents of $F(t)$ depends on (t mod 3). In fact, $F(3s-2)$ contains only instances of $\{3\}$, $F(3s-1)$ instances of $\{6\}$ and $F(3s)$ contains both instances of $\{2\}$ and $\{5\}$. This suggests rewriting the original program in the form:

```
    DO 1 s = 1, n
       F(3s-2)
       F(3s-1)
       F(3s)
1   CONTINUE
```

The explicit coding of each front is now a straightforward problem of loop transformation, see [7] for a general solution. The result is:

```
      DO 1 s = 1,n
         p(s) = 1.0/sqrt(x(s))
         DOALL 2 j = s+1, n
   2        a(j,s) = z(j,s) * p(s)
         PCASE
            DOALL 3 i = s+1,n
   3           x(i) = x(i) - a(i,s)**2
```

[3] We will use the parallel programming primitives of [11].

```
            PAR
               DOALL 4 i = s+1, n
                  DOALL 4 j = i+1, n
      4              z(i,j) = z(i,j) - a(i,j) * a(i,s)
            END PCASE
   1     CONTINUE
```

Let us return to a general analysis of {A}. There is no reason for this program to be optimal. It is a well known fact, however, that most super-computers usually operate one or two orders of magnitude below their peak performance. Hence, while optimality would be a very desirable characteristics, a substantial increase in efficiency would be interesting enough.

The efficiency of a parallel programs for P processors is usually defined as:

$$\epsilon = \frac{T_s}{PT_P}$$

where T_s is the sequential elapsed time (which we will identify to the execution time on one processor), and T_P is the elapsed time on P processors.

Another point to keep in mind is that super-computers are specially adapted to large computations. Having good values of ϵ for large values of n is all we need in practice. We will say that a parallel algorithm is asymptotically efficient iff $\epsilon \to 1$ when the structure parameters tend to infinity. Our aim in this paper is to investigate conditions under which algorithm {A} is asymptotically efficient.

3.1 The case of perfect information

As we have said earlier, $T_S = S(n)$. The duration of iteration t of the algorithm is easily seen to be :

$$d_t = \lceil \frac{\text{Card}(F(t))}{P} \rceil + \sigma \tag{7}$$

where σ is the duration of the synchronization operation. Obviously:

$$x \leq \lceil x \rceil < x + 1. \tag{8}$$

Hence :

$$\begin{aligned} T_P &< \sum_{t=1}^{D(n)} (\frac{\text{Card}(F(t))}{P} + 1 + \sigma), \\ &< (\sigma + 1)D(n) + \frac{S(n)}{P}, \end{aligned} \tag{9}$$

which implies that :

$$\epsilon > \frac{1}{1 + (\sigma + 1)P\frac{D(n)}{S(n)}} \tag{10}$$

From this we deduce that:

Theorem 1 *All algorithms such that the ratio $D(n)/S(n) \to 0$ as $n \to \infty$ have an asymptotically efficient parallel implementation, provided that the execution times of all operations are equal.*

The above ratio may thus be seen as a characterization of the degree of parallelism of the source algorithm. Since this caracterization is derived from the DFG, it does not depends on implementation details such as memory management or loop ordering. In the case of the Cholesky algorithm, for instance, this ratio is of order $O(1/n^2)$, which indicates very good parallelism. The same is true for matrix multiplication, Gauss-Jordan elimination and LU factorisation. At the other end of the spectrum are summation algorithms (the dot product is an exemple), for which the ratio is $O(1)$. An interesting situation obtains in the case of the solution of a triangular system, which is $O(1)$ or $O(1/n)$ depending on whether the dot products are computed left-to-right or right-to-left. This indicates that while our theory abstracts from a large number of possible transformations, it still does not take into account algebraic properties of operators like associativity and commutativity. Further work is needed in this direction.

In the sequel we will say that an algorithm has degree of parallelism α if the ratio $D(n)/S(n)$ is $O(1/n^\alpha)$ when $n \to \infty$.

3.2 A Worst Case Analysis

The hypothesis that the duration of an operation is always exactly one time unit is not likely to be realized in practice, with the important exception of strongly synchronous architectures like the Connexion Machine or VLIW computers. We would like to know what happens in the face of such variations. Let the mean duration of an operation be 1 time unit, and let τ be an upper bound. Obviously, the worst case occurs when the longest operations are all executed by the same processor. Since there is a synchronization operation at the end of each iteration, each processor will wait for the slowest one. In this case we may neglect the synchronization time and ceiling effects. The new iteration time is:

$$d_t = \tau \frac{\mathrm{Card}(F(t))}{P}, \tag{11}$$

and the efficiency becomes:

$$\epsilon = \frac{1}{\tau}. \tag{12}$$

The conclusion is that {A} no longer is asymptotically efficient, and that the degradation is directly proportional to the variability in operation time. This result, however, is not very realistic. Arranging for the longest operations to be executed by one particular processor is as difficult a task as distributing them equally among all processors. We will next investigate two more realistic models. In the first one, the so-called self-scheduling approach, the allocation of an operation to a processor is done on the fly at execution time. In the other one, we will suppose that the operation durations are independent random variables and compute the expected value of the parallel running time.

3.3 Greedy Self-scheduling

A greedy scheduler works in the following way. When starting the execution of a front, all its operations are put into a (real or virtual) queue. When a processor terminates an operation, it removes an arbitrary operation from the queue and start executing it. If the queue is empty, it stops until all pending operations are terminated and then start again on the next front. We deduce that under the greedy scheduling policy, no processor is idle as long as there is work to be done.

Let us take as origin the time at which front $F(t)$ start executing. Let d_0 be the time at which the work queue becomes empty, d be the time at which all work on the current front is terminated, and let $f_p \subset F(t)$ be the set of operations which are executed by processor $p, p = 1, P$. Processor p finishes working at time $\sum_{u \in f_p} \partial(u)$ and find the work queue empty. Hence:

$$d_0 \leq \sum_{u \in f_p} \partial(u) \tag{13}$$

and if we sum all such inequalities for $p = 1, P$ we obtain:

$$P d_0 \leq \sum_{u \in F(t)} \partial(u). \tag{14}$$

Now, since from time d_0 to d each processor execute at most one operation, we have $d - d_0 \leq \tau$, and hence:

$$d \leq \frac{\sum_{u \in F_t} \partial(u)}{P} + \tau. \tag{15}$$

Summing on all fronts, and taking care of including the synchronisation time, we get:

$$\begin{aligned} T_P &\leq \frac{\sum_{t=1}^{D(n)} \sum_{u \in F_t} \partial(u)}{P} + D(n)(\tau + \sigma), \\ &\leq \frac{S(n)}{P} + D(n)(\tau + \sigma). \end{aligned}$$

and hence:

$$\epsilon \geq \frac{1}{1 + P\frac{D(n)}{S(n)}(\tau + \sigma)}, \tag{16}$$

which gives us the

Theorem 2 *All algorithms such that $D(n)/S(n) \to 0$ as $n \to \infty$ have an asymptotically efficient parallel implementation provided that the execution time of all operations is bounded uniformly in n.*

In the above analysis, we have neglected the time it takes to extract the next operation from the scheduler queue. On most computers, this must be done from within a critical section. Let κ be the extraction time; the global memory must be considered as one more processor, with a running time of $\kappa S(n)$, to be compared to the total running time which is of the order of $\frac{S(n)}{P}$. This mean that self-scheduling is efficient only if $\kappa \ll 1/P$. For ways of improving this situation by lumping several operations, see [15]. On machines

with recombining networks, like the New York Ultracomputer ([8]), there is no interference between queue accesses, and the extraction time may simply be added to the operation execution time.

3.4 A Probabilistic Analysis

In this section, we will suppose that an element of chance enter in the definition of the operation execution time. If randomness is defined as the interaction of two independent causal sequences then in a complicated architecture, there must be many random events. Furthermore, one may further contribute to the randomness of the process by allocating an operation to a processor according to some pseudo-random or hashing function (see [18] for an exemple).

Let us suppose that the duration of an operation is a random variable with expectation one time unit and variance ν^2. All such variables are deemed independent. Let us first consider a front F. Suppose that operations in F are partitionned equally between P processors. The workload of processor p will be a random variable X_p with expectation m and variance $m\nu^2$, where $m = \mathrm{Card}(F)/P$. The duration of front F is the random variable:

$$Z = \max_{p=1}^{P} X_p, \tag{17}$$

and we are interested in the expectation of Z. We will suppose that the X_p have a common distribution function:

$$\Phi(x) = \mathrm{Prob}\{X_p \leq x\}, \tag{18}$$

and a density [4]

$$\phi(x) = \Phi'(x). \tag{19}$$

Since the event $\{Z \leq z\}$ is equal to $\{X_p \leq z, p = 1, P\}$, the distribution function of Z is $\Phi(z)^P$, and its density is $P\Phi(z)^{P-1}\phi(z)$. The expectation of Z is then:

$$E(Z) = P \int_0^\infty z\Phi(z)^{P-1}\phi(z)dz. \tag{20}$$

The integral may easily be bounded by Schwartz inequality:

$$E(Z)^2 \leq P^2 \int_0^\infty z^2\phi(z)dz \int_0^\infty \Phi(z)^{2P-2}\phi(z)dz. \tag{21}$$

The first integral is $E(X_p^2)$. The second evaluates to $1/(2P-1)$, to give :

$$E(Z) \leq P\sqrt{\frac{E(X_p^2)}{2P-1}}. \tag{22}$$

Let a be any non-random number. From the identity:

$$\max_{p=1}^{P}(X_p) = a + \max_{p=1}^{P}(X_p - a) \tag{23}$$

[4] The reader may care to convince himself that all our conclusions would stand in the case of a discrete distribution function (and even in more complicated cases).

we may deduce a whole family of analogous bounds for $E(Z)$. The most precise one is associated to the minimum value of $E((X_p - a)^2)$, which obtains when $a = m = E(X_p)$. The final result is:

$$E(Z) \leq E(X_p) + P\sqrt{\frac{\mathrm{Var}(X_p)}{2P-1}}. \tag{24}$$

Replacing $E(X_p)$ and $\mathrm{Var}(X_p)$ by their values, we get [5]:

$$d \leq \frac{\mathrm{Card}(F)}{P} + \nu\sqrt{\frac{P}{2P-1}}\sqrt{\mathrm{Card}(F)} \tag{25}$$

for the expected duration of front F. Summing on all fronts, we get:

$$E(T_P) \leq \frac{S(n)}{P} + \nu\sqrt{\frac{P}{2P-1}}\sum_{t=1}^{D(n)}\sqrt{\mathrm{Card}(F(t))} \tag{26}$$

The sum is easily seen to be maximum when all summands are equal, i.e. when

$$\mathrm{Card}(F(t)) = S(n)/D(n).$$

This gives the final bound:

$$E(T_P) \leq \frac{S(n)}{P} + \nu\sqrt{\frac{P}{2P-1}}\sqrt{S(n)D(n)}. \tag{27}$$

$$\epsilon \geq \frac{1}{1 + P\nu\sqrt{\frac{P}{2P-1}}\sqrt{\frac{D(n)}{S(n)}}} \tag{28}$$

From this we deduce the

Theorem 3 *If the operation durations of an algorithm may be represented by random independent variables with finite expectation and variance, then the fact that the ratio $D(n)/S(n) \to 0$ as $n \to \infty$ is a sufficient condition for the algorithm to have an asymptotically efficient parallel implementation.*

One should note that random fluctuations in the execution times effectively halve the degree of parallelism, a not unexpected result ! The factor $\sqrt{P/(2P-1)}$ vary between 1 and $1/\sqrt{2}$. It should not be an important element in the following discussion. With this proviso in mind, to get, say, more than 80% efficiency, one should keep $P\theta/n^\alpha < .25$, where θ is some characteristic time and α is the effective degree of parallelism. This will allow more massive parallelism on synchronous machine and coarser parallelism on asynchronous and distributed systems.

[5] Here again we neglect synchronization time and ceiling effects.

4 Relation to Previous Work

As the reader may have gathered by now, the aim of this paper is to put in perspective a host of existing techniques rather than introduce new ones. In this context, our debt to work on the construction of systolic arrays should be obvious.

A recent paper by Eager et. al. ([3]) is very near to our concerns and results. In [3], one starts with a task graph and finds bounds on its efficiency when executed on a multiprocessor. The authors show that most of the details of the Task Graph may be summarized by the Average Parallelism Measure, which is exactly the same as the ratio D/S of the present paper.

From here on the papers take somewhat divergent directions. Eager et. al. suppose that the task graph has been constructed elsewhere, that the tasks are few and that their durations are large. This allows them to lump the synchronization time with execution time and to use processor sharing as a scheduling policy. The main cause of inefficiency in their task graphs is variation in the workload.

Here on the contrary our main objective is the construction of a task graph from the DFG. Nodes of the DFG are build from a few machine instructions; there is almost always adequate workload (as long as $D/S \to 0$), and the main causes of inefficiencies are the synchronization operations, the impossibility to use processor sharing at the instruction level, and inaccuracies in the operation durations.

As a result we feel that the two papers give complementary information at the two extremities of the granularity spectrum. It is a very striking fact that the main controlling parameter (the average degree of parallelism), is the same in both cases.

5 Conclusion

Let us summarizes what has been achieved so far. When given a (FORTRAN) program, we start by building its dataflow graph by techniques which are described in [5] and have been sketched here. From the DFG we construct a timing function and obtain two crucial informations:

- the sequential operation count, $S(n)$,
- the length of the longest path in the expanded DFG, $D(n)$.

We claim that the ratio $\rho = D(n)/S(n)$ is a fair measure of the inherent parallelism in the original program. If $\rho \to 0$, then there is an asymptotically efficient parallel implementation, and the faster ρ tends to zero, the faster the efficiency tends to one. Conversely, since $D(n)$ is a lower bound on the parallel execution time on a finite number of processors, if ρ does not tend to zero, the advantage to be gained from a parallel execution is limited.

All these results depends on the hypothesis that all operation are executed in unit time. Let us note first that all our results would subsist under the less stringent hypothesis that all operations inside one front have the same duration. In the presence of timing variations inside a front, we have investigated two solutions:

- In self-scheduling, the allocation of operations to processors is done at execution time. Self-scheduling is very efficient but precludes the use of fine grain parallelism.
- Randomization allocates operations at compilation time according to some deterministic approximation of a random process. The method has low overhead but reduces the effective degree of parallelism.

Choosing between the two approaches will obviously depends on details of the source program and object computer structure.

Let us now investigates the technical feasibility of our proposal. There is no problem with the construction of the DFG. Constructing *linear* timing functions also is a well known procedure, but there are programs for which such a function does not exists. An elementary exemple is:

```
      DO 1 i = 1,n
         DO 1 j = 1,i
1           s = s + x(i, j)
```

One may try to look for polynomial timing functions. This does not look a very promising approach, since the next task is to invert the timing function for the construction of fronts. It seems much more promising to search for multidimensional and multiphase timing functions. This problem will be the subject of future research.

The construction of fronts is a problem in loop transformation, for which we have given the elements of a solution in [7]. It is here that a knowledge (even an approximate one), of execution time may be helpful when distributing the workload between processors.

Once the fronts are constructed, one must insert appropriate synchronization instructions in the code. In the skeletal algorithm {A}, this has been done in a systematic, uneconomical way. In some cases one may get away with much less synchronization (consider the case of matrix multiplication), and this will generally be worthwhile as synchronizations operations are always quite costly. In machines which are inherently synchronous, there will be no need for explicit synchronization. As a counterpart, these architectures are only able to execute homogeneous fronts. Fronts with more than one type of operations (like $F(3s)$ in the Cholesky exemple), will have to be split into several sub-fronts. Since the number of sub-fronts will always be finite, this will not decrease the asymptotic degree of parallelism.

Next come memory management. Most of the time, the program obtained in the above way will be incorrect, since we have not taken into account anti- and output-dependencies([14]). Correctness must be restored by minimal scalar and array expansion, and this is a whole research subject in itself.

Let us now examine two open problems. We have said in section 3.1 that the degree of parallelism of the triangular solver algorithm depends on the order in which a dot product is computed. Mathematically speaking, all such orders are equivalent, since addition is associative and commutative, but this is no longer true if rounding errors are taken into account. There are reasons to believe that, with the exception of some pathological cases, the summation order is not critical. Hence, a technique for including such equivalence in the calculation of the timing function would be highly interesting. There should however

be a directive for turning the facility on and off according to the numerical stability of the source algorithm.

While we have used the language of shared memory architectures, this work is equally applicable (in fact, perhaps more so), to distributed systems. A frequently used paradigm for the construction of distributed programs is the graph partitionning approach. Edges in the task graph represent information transfer. They may be valued by the volume of the transfered data. One then proceed to partition the task graph in such a way that the sum of the values of interpartition edges is minimal. What is lacking in this approach is a sense of time. There is a big difference between a transfer which is evenly distributed all along the lifetime of the computation, and one which is concentrated, e.g. at the beginning or end. The graph partitionning approach cannot take these phenomena into account. We think that timing functions will provide the required time scale. Note first that, by construction, there is no information exchange inside a front. All communication occur between different fronts, and the communication overhead will be minimal if communicating operations are allocated to the same processor. This will obviously conflicts with the objective of load equalization; one should try to locate an optimum. This problem will be the subject of future research.

References

[1] J.R. Allen and Ken Kennedy. Automatic loop interchange. *SIGPLAN Notices*, 19, 1984.

[2] Jean-Loup Baer. A survey of some theoretical aspects of multiprogramming. *Computing Surveys*, 5:31–80, March 1973.

[3] Derek L. Eager, John Zahorjan, and Edward D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38:408–423, March 1989.

[4] Paul Feautrier. Array expansion. In *ACM Int. Conf. on Supercomputing, St Malo*, 1988.

[5] Paul Feautrier. *Data Flow Analysis of Scalar and Array References.* Technical Report 282, MASI, April 1989.

[6] Paul Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, September 1988.

[7] Paul Feautrier. Semantical analysis and mathematical programming; application to parallelization and vectorization. In *Workshop on Parallel and Distributed Algorithms, Bonas*, October 1988.

[8] Allan Gottlieb, Ralph Grishman, Clyde P Kruskal, Kevin P. McAuliffe, Larry Rudolf, and Marc Snyr. The nyu ultracomputer : designing an mimd shared memory parallel computer. *IEEE Transactions on Computers*, C32:175–189, February 1983.

[9] William Jalby and Ulrike Meier. *Optimizing Matrix Operations on a Parallel Multiprocessor with a Memory Hierarchy.* Technical Report, CSRD, February 1986.

[10] Pierre Jouvelot. Semantic parallelization, a practical exercise in abstract interpretation. In *ACM-POPL '87*, Munich, 1987.

[11] J. Karp. Programming for parallelism. *IEEE Transactions on Computers*, May 1987.

[12] Leslie Lamport. The parallel execution of do loops. *CACM*, 17:83–93, February 1974.

[13] Alain Lichnewsky and François Thomasset. Introducing symbolic problem solving techniques in the dependence testing phases of a vectorizer. In *ACM Int. Conf. on Supercomputing*, St Malo, France, July 1988.

[14] D. A. Padua and Michael J. Wolfe. Advanced compiler optimization for super computers. *CACM*, 29:1184–1201, December 1986.

[15] Constantine Polychronopoulos and Donald J Kuck. Guided self-scheduling. *IEEE Transactions on Computers*, C-36:1425–1439, December 1987.

[16] Patrice Quinton. Mapping recurrences on parallel architectures. In *3rd Int. Conf. on Supercomputing*, Boston, May 1988.

[17] Julius T. Schwartz. Ultracomputers. *ACM Transactions on Programming Languages and Systems*, 484–521, 1980.

[18] Nadia Tawbi and Paul Feautrier. *Parallélisation automatique de programmes pour ordinateur multiprocesseur à mémoire partagée.* Technical Report 285, MASI, March 1989. Journées algorithmes paralléles et architectures nouvelles.

[19] Michael J. Wolfe. Loop skewing, the wavefront method revisited. *Int. J. of Parallel Processing*, 15, August 1988.