J. Functional Programming **4** (1): 65–75, January 1994

Worst case scheduling for parallel functional programs[1]

F. WARREN BURTON[2] AND V. J. RAYWARD-SMITH[3]

[2] *School of Computing Science, Simon Fraser University, Burnaby, British Columbia, Canada V5A 1S6*
(*e-mail*: burton@cs.sfu.ca)

[3] *School of Information Systems, University of East Anglia, Norwich NR4 7TJ, UK*
(*e-mail*: vjrs@sys.uea.ac.uk)

Abstract

Many of the details that a programmer must manage when programming in a procedural language are handled by the implementation in a functional language. In a parallel functional language, we would expect the assignment of processes to processors and the scheduling of processes to be handled by the implementation. The partitioning of a program into processes may be handled by the implementation as well.

We will assume that a parallel functional program may create processes dynamically, and that processes may synchronize in an arbitrary manner. It does not matter whether processes are defined in the source program or generated by the implementation.

On parallel systems where each processor has some local memory, it is common practice not to move processes once they have started to execute. We will show that if each process must be executed on a single processor, then no fully automatic scheduling strategy can ensure good performance.

We also will show that if all processes are sequential processes (i.e. do not contain internal parallelism), and if a process may be moved whenever it resumes execution following synchronization with another process, then good performance can be assured, at least in theory.

Capsule review

Task scheduling is the quintessential analytical problem of parallel computation. Functional programs allow a larger number of feasible schedules than do imperative ones because they entail no hidden inter-task dependencies. The scheduling problem is to determine, by a static analysis of a functional program, a mapping of tasks to processing resources that respects inter-task dependencies and makes reasonably economical use of the available processing resources.

This paper shows that in the absence of *a priori* information as to resource requirements of tasks, good schedules for parallel execution of tasks on distributed-memory multiprocessors cannot be found. The results are obtained by considering a pair of simply constructed, yet hard examples of dependency graphs.

[1] This work was supported by the Natural Science and Engineering Research Council of Canada.

1 Introduction

Parallel functional programs differ from parallel procedural programs in several ways. A functional program is, by definition, deterministic. (Our results also apply to languages permitting non-determinism, but do not depend on the presence of non-determinism.) In addition, functional programs tend to be highly dynamic in nature, so a parallel functional language implementation must support dynamic process creation and arbitrary process synchronization. The high level nature of functional programs suggests that details such as the assignment of processes to processors and the scheduling of processes should be handled by the language implementation rather than the programmer.

On parallel systems where each processor has some local memory, it is common practice not to move processes once they have started to execute. We will use the term *distributed program* to refer to a program in which a process may not be moved once it has started execution.

We will show that fully automatic scheduling cannot be done in a way that will ensure good performance for distributed programs. Depending on how processes are scheduled, a program may have a speed-up of m, where m is the number of processors, or a speed-up arbitrarily close to 1 (i.e. no effective parallelism). We will argue that a practical system will not have sufficient information to be able to choose a good schedule over a poor schedule.

In addition, the problem of determining a good assignment of processes to processors is impossible with the information that is likely to be available to a system at the time the assignment must be done.

These problems are avoided if a process may be moved following synchronization with another process, provided processes contain no internal parallelism. In this case, any scheduling strategy that avoids idle processors while there is work to be done is guaranteed to be within a factor of 2 of an optimal schedule.

Our results may apply to some procedural languages, provided the assumptions stated above are satisfied.

2 Model, assumptions and terminology

We will model a *deterministic distributed program on m processors* by an ordered triple $(A, \subset, f)$, where A is a set of atomic actions, simply called actions hereafter, $\subset$ is a partial order defined on those actions, and $f: A \to \{1,2,\ldots,m\}$ is a total function. If $a_1 \subset a_2$ then a_1 must be performed before a_2. The function f is an assignment of actions to processors.

In the remainder of this paper we will assume that each action can be performed by a processor in one unit of time. This is not a serious restriction in our model, but simplifies the presentation. A process or task in a high level programming language corresponds to a collection of actions. A sequential process corresponds to a chain of actions (i.e. a collection in which for any two actions, a_1 and a_2, either $a_1 \subset a_2$ or $a_2 \subset a_1$).

https://doi.org/10.1017/S0956796800000940 Published online by Cambridge University Press

A *schedule* is a total function $s: A \to \{1, 2, \ldots\}$, subject to the restrictions that

1. If $f(a_1) = f(a_2)$ and $s(a_1) = s(a_2)$ then $a_1 = a_2$.
2. If $a_1 \subset a_2$ then $s(a_1) < s(a_2)$.

A schedule assigns actions to times when they are to be performed, subject to the restriction that no two actions are performed at the same time on the same processor and the partial order $\subset$ is respected. If other than unit execution time actions are allowed, this definition must be generalized.

We define $length(s, (A, \subset, f)) = \max\{s(a) \mid a \in A\}$ to be the *length of a schedule*. We will write $length(s)$ for $length(s, (A, \subset, f))$ when $(A, \subset, f)$ is clear from context. An optimal schedule for $(A, \subset, f)$ is a schedule s, such that $length(s) \leqslant length(t)$ for any other schedule t for $(A, \subset, f)$.

We call a schedule *work conserving* if a processor never idles when it could work. Clearly, there is a work conserving schedule of optimal length so we will restrict all our discussion to such schedules.

For the present, we will assume that the number of processes in a distributed program equals the number of processors, with one process per processor. Hence, we will use the term *process* to mean the set of actions assigned to a given processor. A process may contain internal parallelism. That is, the actions within a process do not need to be totally ordered with respect to $\subset$. If a reader likes to think of a process as a totally ordered set of actions (i.e. a sequential process) then he or she may wish to regard the collection of actions assigned to a given processor as a collection of processes that happen to be assigned to the same processor. In Section 5 we will consider the problem of assigning processes to processors for the general case where there are more processes than processors.

Many algorithms in classical scheduling theory require complete information, at the time scheduling is done, on the number of actions and how actions synchronize. In a practical system for distributed computing, a scheduler is likely to have only limited information. For example, if a system could determine how much time a process will require (i.e. how many actions the process contains), it could solve the halting problem. Therefore, a processor is unlikely to know how many actions are assigned to it, although it can know how many runnable actions it currently has. Similarly, if process synchronization operations are contained in conditional expressions then a system cannot predict how actions will synchronize. Hence, we make the following assumption:

Assumption All runnable actions appear the same to a scheduler.

In job shop scheduling (e.g. see Coffman, 1976) partially ordered unit execution time sets of actions are assigned to processors. The basic model differs from our model in that there is no predetermined assignment; any action can be processed on any machine. Jaffe (1980) refined this basic model to include types of action and corresponding types of processor. He showed that with k types of action and m_i identical processors for actions of type i, the length ω of any work conserving schedule satisfies $\omega \leqslant [(k+1) - 1/\max\{m_1, m_2, \ldots, m_k\}] \times \omega_{opt}$, where ω_{opt} is the length

https://doi.org/10.1017/S0956796800000940 Published online by Cambridge University Press

of the optimal schedule. Our model thus corresponds to a special case of Jaffe's model where $k = m$, the number of processors, and $m_1 = m_2 = \ldots = m_m = 1$.

3 Worst case performance for distributed programs

In this section we will represent deterministic distributed programs on m processors by diagrams. Each circle or oval will represent an action. The partial order $\subset$ is the transitive closure of the relation defined by arrows between actions. Finally, we denote the set of actions assigned to a single processor by drawing a box around those actions. (The processor identity numbers are not relevant, so are not specified.)

Perhaps the simplest example of a program that is highly sensitive to the order in which actions are scheduled is the one given in Fig. 1, which we will refer to as

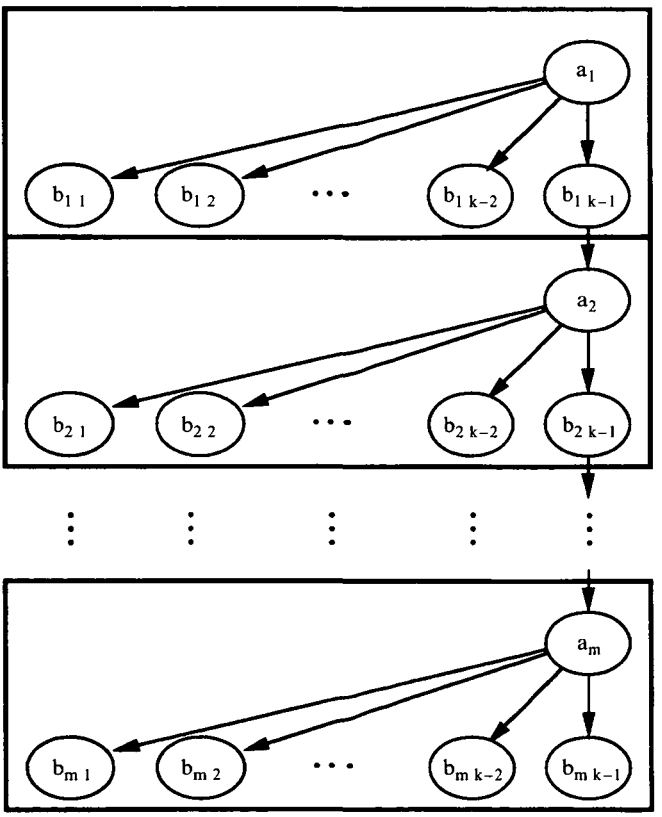

Fig. 1. Example 1.

Example 1. If each processor schedules runnable actions from left to right, then this program is clearly sequential. There will never be two processors running at the same time. The execution time for m processors each with k actions will be mk. On the other hand, if actions are scheduled from right to left, then the actions in the right hand column and the bottom row will run sequentially. By the time these have finished all

https://doi.org/10.1017/S0956796800000940 Published online by Cambridge University Press

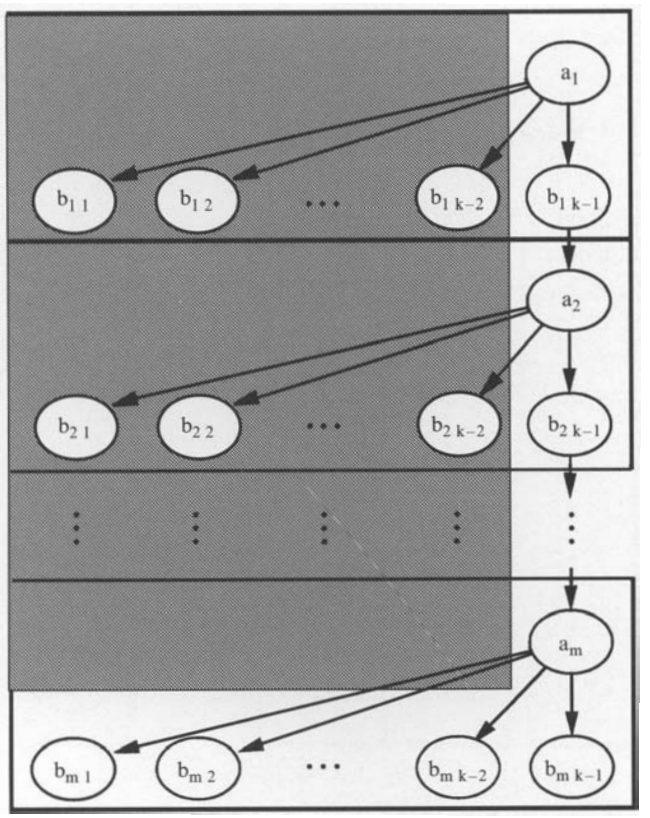

Fig. 2. Example 1 best case.

the other actions (in the shaded square in Fig. 2) will have completed, since the bottom processor will be the last to start and the last to finish. This will result in an execution time of $2m+k-2$. As the number of actions per processor increases, the speedup will approach the number of processors, since

$$\lim_{k\to\infty} \frac{mk}{2m+k-2} = m.$$

The problem here is that each processor contains a single critical action. All other processors are blocked waiting for this action to run. Since we assume that a processor cannot determine which process is critical, we cannot insure good performance.

We have found a wide variety of program structures that result in the same problem, and have not been able to see any reasonable restrictions that will avoid critical actions delaying other processors. Another example that avoids the high degree of branching found in Example 1 but has more synchronization between processors, is given as Example 2 in Fig. 3. This situation might result if the actions

https://doi.org/10.1017/S0956796800000940 Published online by Cambridge University Press

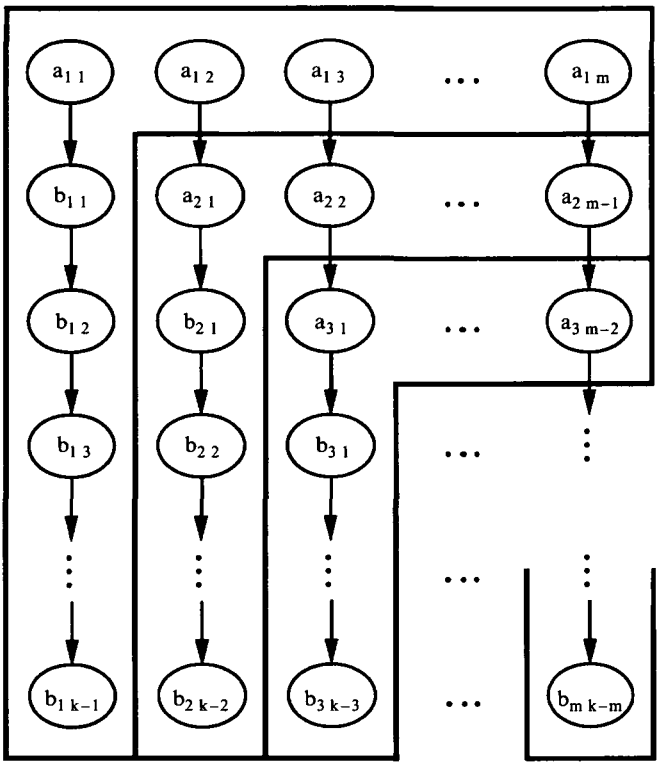

Fig. 3. Example 2.

assigned to each processor consist of a number of processes that happen to be placed on the same processor. In the best case actions are scheduled from right to left. All the actions on a given diagonal may be executed in a single step as shown in Fig. 4. This results in an execution time of $m+k-1$ as illustrated in Fig. 4. Since there are mk actions, the sequential execution time would be mk. Hence the best case parallel execution time produces a speedup of $mk/(m+k-1)$, which goes to m in the limit as k becomes large. On the other hand, if processes are scheduled from left to right, then the actions outside the grey triangle in the upper right corner of Fig. 5 will be executed sequentially, taking time $mk-(m-2)(m-1)/2$. As k becomes large the speedup drops to one in the limit.

These examples raise two questions. First, can we locate processes on processors more sensibly and reduce the problem? Second, is the expected performance better than the worst case performance illustrated here? The first of these questions is addressed in section 5. The second is considered in the following section.

4 Expected performance for distributed programs

It is unrealistic to make any assumption about the 'average structure of a distributed program'. That is, we cannot define a probability space for all possible programs that is going to meaningfully relate to programs found in the real world.

https://doi.org/10.1017/S0956796800000940 Published online by Cambridge University Press

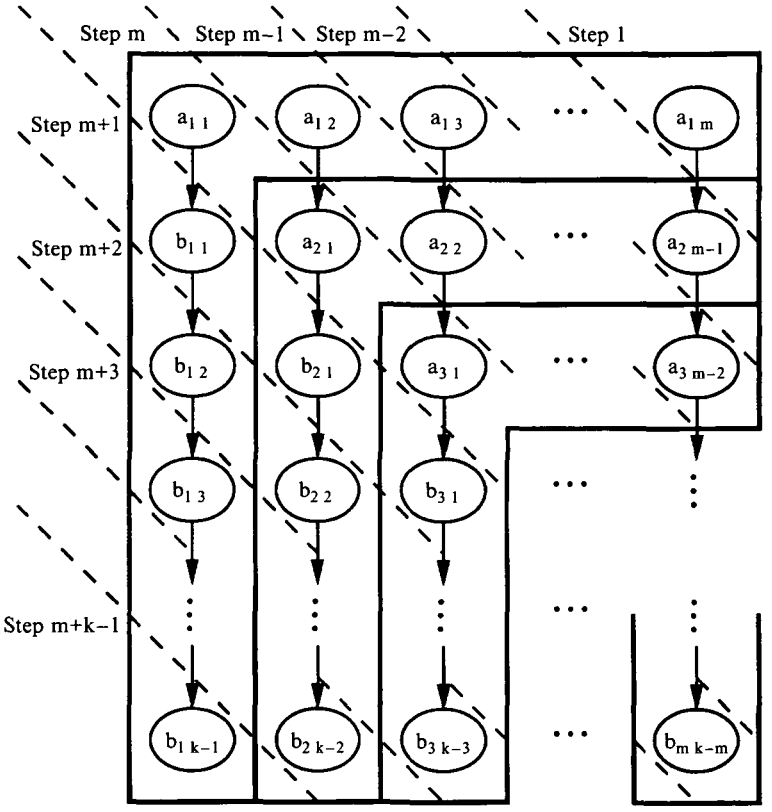

Fig. 4. Example 2 best case.

However, we can assume that actions that cannot be distinguished are executed in a random order, and design a scheduler to do this. However, if we reconsider Example 1 in Fig. 1, we see that on average half the noncritical actions will be executed before the critical one, on each processor. This would result in a speedup on m processors of 2 rather than 1 in the limit as k becomes large, which is not satisfactory. This is the best we can hope to do in this example without some information about what actions will do.

5 Process placement

So far we have assumed that all actions are preassigned to specific processors. In this section we will assume that actions are grouped into processes, that all actions in a given process must be assigned to the same processor, but that different processes may be assigned independently. We will show by example that it is not possible to place processes wisely without knowledge of what is yet to happen.

We will remove the restriction that each action requires exactly one unit of time to execute. Consider the collection of m^2 processes, to be run on m processors, illustrated in Fig. 6. Each process consists of two actions, an a action and a corresponding b action. No b action can be performed until all a actions are finished. We will assume that each a action requires one unit of time, but b actions are of two types. *Nice* type

https://doi.org/10.1017/S0956796800000940 Published online by Cambridge University Press

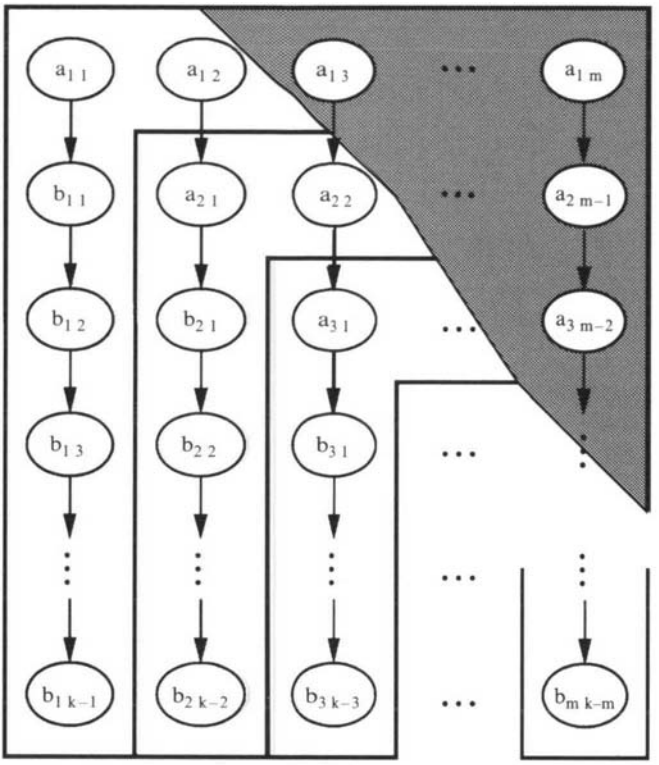

Fig. 5. Example 2 worst case.

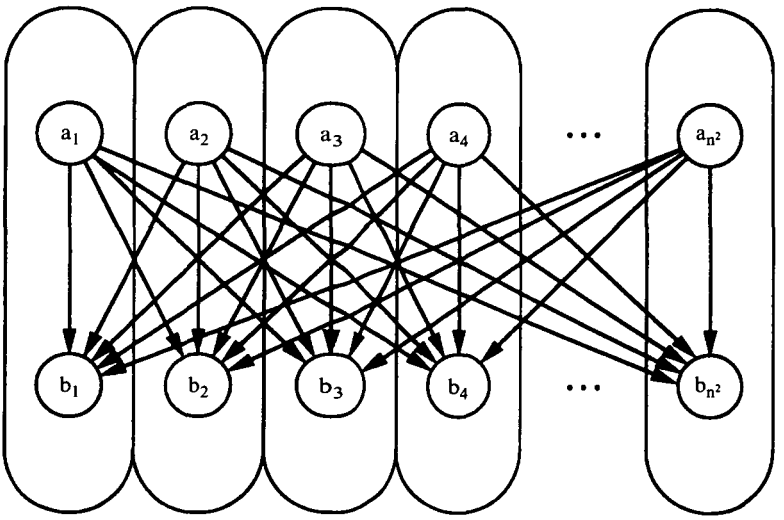

Fig. 6. Synchronizing processes.

b actions require one unit of time, but *nasty* type *b* actions require *k* units of time, for some arbitrary, large, value *k*. (If the reader wishes to continue to think in terms of unit execution time actions, replace each nasty type *b* action with a chain of *k* unit execution time actions.) A process containing a nice or nasty action will be called a

https://doi.org/10.1017/S0956796800000940 Published online by Cambridge University Press

nice or nasty process, respectively. There are m nasty processes and $m(m-1)$ nice processes.

We will assume that a computer system must assign each process to a processor before that process starts execution. Furthermore, we will assume that the system cannot distinguish a nasty process from a nice one before the type b action starts executing. (Again, if we could tell how much time a computation was going to require in advance, we would solve the halting problem.) Since all type a actions must be performed before any type b action, all processes must be assigned to processors before the character of any process is known.

In the best case, each processor will be assigned 1 nasty process and $m-1$ nice processes. The resulting schedule will require $2m+k-1$ units of time (2 units for each nice process and $k+1$ units for the nasty process). All processes will be busy until all work is done, so the speed-up will be m.

On the other hand, with m^2 processes, at least one processor must have at least m processes, no matter how processes are assigned to processors. If one processor happens to be assigned all m nasty processes, then the schedule length will be $m(k+1)$, giving a speed-up of only $m(2m+k-1)/(m(k+1))$. As k becomes large, nasty processes become increasingly dominant and the speed-up drops to arbitrarily close to 1. Hence, with process placement, as with scheduling, there are cases where programs may have a speed-up equal to the number of processors, virtually no speed-up, or anything in between. In this case, if processes were assigned to processors in a completely random fashion, we would not expect to do too badly.

The problem of assigning processes to processors is considered in more detail elsewhere (Warren Burton *et al.*, 1992).

6 Scheduling on a shared memory machine

The problems we have experienced in the previous sections result from each action being tied to a particular processor. In all of the bad examples at least one processor had a choice of actions to perform while other processors sat idle.

There are two sources for scheduling problems with distributed programs, both of which result from the requirement that a process must run on a single processor. First, if a process contains internal parallelism, as in Example 1 in Fig. 1, then a number of actions that could otherwise be performed in parallel must run sequentially (in any order). Second, if a process may be delayed waiting for a process on another processor, then a processor may accumulate a large amount of work to be done in the future. The example in Section 5 resulted from the fact that processors had long term commitments of unknown size. That is, a processor with no runnable actions may have an arbitrary amount of delayed work.

So far, we have assumed that each process must run on a single processor. If we remove this restriction, and allow any processor to run any process at any time, then all of our problems vanish. The worst possible work conserving schedule takes less than twice as long as the best possible schedule. In fact, the schedule length can be bounded in terms of the number of processors and the average parallelism in the problem (that is, the speed-up that would be possible with an unbounded number of

processors). Eager *et al.* (1989) have shown that if p is the average parallelism, T_1 is the time required by the program on one processor, T_m is the time required on m processors, and $S_m = T_1/T_m$ is the speed-up on m processors, then $\min(p,m)/2 \leqslant mp/(m+p-1) \leqslant S_m \leqslant \min(p,m)$. If either m or p is very large compared to the other value, then the speed-up approaches the smaller of these two values. For example, if the average parallelism in a program exceeds the number of processors, then a speed-up equal to at least half the number of processors is guaranteed, and the guaranteed speed-up must approach the number of processors as the average parallelism becomes large. On the other hand, the speed-up cannot be greater than the minimum of the average parallelism and the number of processors. This result suggests that we do not need to worry much about scheduling.

The above result assumes that any processor can perform any action. However, it is sufficient to allow processes to be moved following a synchronization operation in which the processes have been suspended waiting for action by another process, provided each process is sequential. If each process, once started, runs until either it terminates or it is suspended, then at any point in time at most one action is tied to a particular processor. That action is the one that follows the action most recently executed in the current process. Any other runnable action must have become runnable following a synchronization operation, and therefore may be moved to another processor.

The above strategy is most practical on a shared memory multiprocessor, where moving a process to another processor does not involve moving data. The above theoretical results ignore overheads in moving processes and maintaining global information on runnable processes and/or idle processors. Even if the policy is not followed exactly, it may provide guidance in the design of a parallel functional language implementation. For example, in a small diameter network such as a hypercube, processes might be transferred only to adjacent processors. Alternatively, processes might be moved only when less than half the processors are idle.

7 Conclusion

We have seen that it is not safe to rely on automatic process placement or scheduling for arbitrary distributed programs. If the goal for a computer system is to support general purpose parallel processing at a high level, with dynamic process creation and arbitrary process synchronization, then we believe that a system allowing processes to be moved cheaply is essential for guaranteed performance. However, there are several other possibilities that should be mentioned.

It has been observed that in practice even fairly simple load balancing algorithms tend to give good performance for certain types of distributed computing applications (Eager *et al.* 1986). If we take reasonable care in the type of program we write, and are willing to take our chances with no guaranteed speedup, then we are likely to be in luck most of the time. This approach should not be considered for safety critical real-time programs.

Letting the programmer manage process placement and scheduling appears to be another solution. This may be practical in an application where there is a fixed

number of processes, with this number depending on the number of processors. This is not likely to be practical for highly dynamic programs, and hence is an unsuitable approach for a general purpose functional language.

Since the scheduling problem appears to be more critical than the problem of assigning processes to processors, a system might place processes randomly and require a programmer to assign priorities to processes to guide a scheduler.

It is tempting to regard the problems we have discussed as a basic limitation of distributed systems. Moving processes is easier in a shared memory system, or a system where remote memory accesses are cheap. However, even with shared memory parallel computers there can be analogous problems. With fine grain parallelism, memory contention becomes the problem. If we view actions as accessing a particular memory module rather than being performed on a particular processor, then the order in which actions access memory modules is analogous to the order in which actions are executed on a processor. In fact, the same problem arises with any kind of resource where each action is required to use a particular instance of the resource.

While we have considered only the effect of scheduling on time in deterministic programs, there are other issues that should be mentioned. Scheduling decisions may make spectacular differences in the space requirements of a parallel program (Warren Burton, 1988). If a program is nondeterministic, then any difference in the schedule can completely alter the behaviour of a program. For example, in combinatorial search algorithms, such as branch-and-bound, minor scheduling differences may make major differences in the time required by a program (Warren Burton *et al.*, 1981; Lai and Sprague, 1985).

Acknowledgement

We would like to thank Roger Wainwright of the University of Tulsa for pointing out the problem with memory contention on a shared memory computer.

References

Warren Burton, F. (1988) Storage management in virtual tree machines. *IEEE Trans. Comput.*, **37** (3): 321–328, March.

Warren Burton, F., McKeown, G. P. and Rayward-Smith, V. J. (1992) On process assignment in parallel computing. *Info. Process. Lett.*

Warren Burton, F., McKeown, G. P., Rayward-Smith, V. J. and Sleep, M. R. (1981) Parallel processing and combinatorial optimization. *Proc. Conf. on Combinatorial Optimization*, Stirling, Scotland, August, 19–36.

Coffman, E. G., Jr., editor, (1976) *Computer and Job-Shop Scheduling Theory*. John Wiley and Sons.

Eager, D. L., Lazowska, E. D. and Zahorjan, J. (1986) Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. Softw. Eng.*, **12** (5): 662–675, May.

Eager, D. L., Zahorjan, J. and Lazowska, E. D. (1989) Speedup versus efficiency in parallel systems. *IEEE Trans. Comput.*, **38** (3): 408–423, March.

Jaffe, J. M. (1980) Bounds on the scheduling of typed task systems. *SIAM J. Comput.*, **9**: 541–551.

Ten-Hwang Lai, T.-H. and Sprague, A. (1985) Performance of parallel branch-and-bound algorithms. *IEEE Trans. Comput.*, **34** (10): 962–964, October.