1990

On High Level Characterization of Parallelism

Dan C. Marinescu

John R. Rice
Purdue University, jrr@cs.purdue.edu

Report Number:
90-1011

ON HIGH LEVEL CHARACTERIZATION OF PARALLELISM*

Dan C. Marinescu
and
John R. Rice

Computer Sciences Department
Purdue University
Technical Report CSD-TR-1011
CAPO Report CER-90-32
August, 1990

* Work supported in part by the Strategic Defense Initiation through ARO grants DAAG03-86-K-0106, DAAL03-90-0107 and by National Science Foundation grant CCR-8619817.

On High Level Characterization of Parallelism*

Dan C. Marinescu, J.R. Rice
Computer Sciences Department
Purdue University
West Lafayette, IN 47907, USA

August 30, 1990

Abstract

We discuss issues pertinent to performance analysis of massively parallel systems. We first argue that single parameter characterization of parallel software or of parallel hardware rarely provides insight into the complex interactions among the software and the hardware components of a parallel system. In particular, bounds for the speed up based upon simple models of parallelism are violated when a model ignores the effects of communication.

We then present a new model of parallel execution based on threads of control and events, the E/T model. This model is still simplified compared to the possible complexity of parallel computations but it is able to capture the behavior of an important class of parallel computations, those of SPMD (Single Program Multiple Data) type. The key ingredient of this model is the *characteristic function* $g(P)$ which gives the number of events as a function of the number of threads of control. Our experience suggests that the data needed for this model can be collected fairly easily and with minimal perturbation of the computations. General properties of the computation are derived from the nature of $g(P)$. We indicate how to apply the E/T model to do static load balancing for a computation involving several classes of SPMD subcomputations. Finally, we note that classifying the event types as they are recorded provides very useful data for the analysis of parallel computations.

*Work supported in part by the Strategic Defense Initiation through ARO grants DAAG03-86-K-0106, DAAL03-90-0107 and by National Science Foudation grant CCR-8619817.

1 Introduction

Performance analysis of parallel systems is extremely important for the design of parallel applications and poses significant intellectual challenges for Computer Science [1], [2], [8], [12]. Simple, yet powerful and accurate ways to characterize parallelism are needed. Questions as: what is the most suitable architecture for application $\mathcal{A}$, what is the number of processors of the parallel machine $\mathcal{H}$ to be allocated to the application $\mathcal{A}$ in order to have optimal performance in some sense, and how to schedule a group of applications on a given parallel system, can be approached only if the parallelism in an application $\mathcal{A}$ can be detected and characterized. It would be highly desirable to characterize a parallel application by only a small number of parameters and by doing so to be able to make reasonably accurate statements about the level of performance, e.g., speedup and efficiency when the application runs on a particular parallel system.

In Section 3 we present the E/T model which attempts to do this and develop its properties. Sections 4 and 5 discuss applications of the model and suggest that it provides a reasonable compromise between simplicity and usefulness.

2 Communication Latency and Single Parameter Characterization of Parallelism

The effects of communication latency are difficult to be captured by simplified models of parallel execution, and they affect greatly the performance of parallel computations on real machines. Several attempts to provide synthetic characterization of parallelism without taking into account the delays due to communication and control have been made in the past.

In [4], the *average parallelism, A* is proposed as a single parameter characterization of a software structure. A software structure is modeled in [4] as a directed acyclic graph (DAG) in which each vertex corresponds to a subtask with known service demands, but no communication costs are specified.

Bounds on the speedup and efficiency are obtained. It is shown that the speedup with n processors, $S(n)$ of a software structure with average parallelism A is bounded as

$$\frac{nA}{n+A-1} \leq S(n) \leq \min(n, A). \tag{2.1}$$

The average parallelism of a software structure is defined in several different ways and it is argued that the definitions are equivalent. Consider the first two definitions:

1. A is the average number of processors that are busy during the execution time of a software system in question, given an unbounded number of processors.

2. A is equal to the speedup, given an unbounded number of processors.

The definitions of A do not impose any restrictions upon the architecture of the parallel system or upon its hardware characteristics, in particular, upon the communication speed. Thus A seems to be an inherent characterization of the software structure, independent of the hardware. Then the bounds specified by (2.1) would indicate that the efforts towards more sophisticated and efficient parallel architectures is misdirected. For example, if we have an application exhibiting an average parallelism $A = \frac{n}{2}$ and if we can build parallel hardware with n processors, then the speedup will be in the range $\frac{n}{3} \leq S(n) \leq \frac{n}{2}$ (for large n). If so, it seems wasteful to build parallel machines with expensive, fast communication.

At this point we should probably develop a healthy suspicion that A is not only a function of the software structure, but it depends strongly upon the hardware component, in particular, upon the communication delay, τ.

As an example for our arguments, consider a parallel computation $\mathcal{A}$ related to domain decomposition on a hypercube [9]. The computation consists of $(k+1)$ synchronization periods, $\pi_0, ..., \pi_k$. During period π_i, $0 \leq i \leq k-1$, there are $n_i = 2^{k-i}$ subtasks which may run concurrently. All tasks of π_i start at the same time and run for a deterministic length of $t_i = 2^i$ units of time, then pass their results to all the tasks of π_{i+1}. The computation is completed at the end of π_k when only one task runs for $t_k = 2^k$ units of time, as shown in Figure 1.

Call N the number of subtasks of $\mathcal{A}$. Then N is given by

$$N = \sum_{i=0}^{k} n_i = \sum_{i=0}^{k} 2^i = 2^{k+1} - 1. \tag{2.2}$$

Assume that we have a parallel system $\mathcal{H}$ with N identical processors and that each subtask of $\mathcal{A}$, $\tau_{i,j}$ with $0 \leq i \leq k$, $0 \leq j \leq 2^i$ is assigned to one processor of $\mathcal{H}$ and it will run as soon as input data is available. Such a

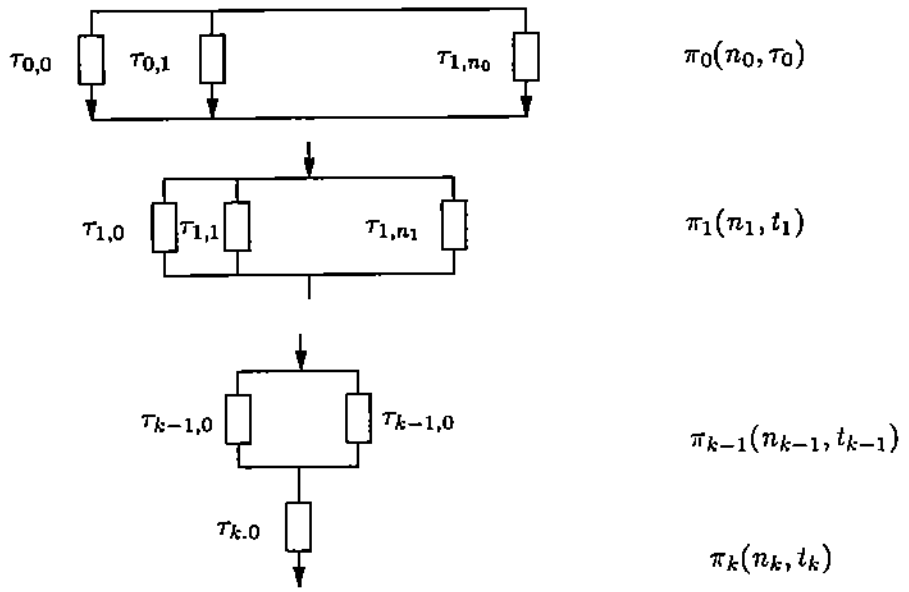

Figure 1: Structure of the example parallel computation $\mathcal{A}$

setup satisfies the conditions of [4], since a work-conserving schedule is used and we have one processor for every task.

Assume first that communication occurs instantaneously. Then the parallel execution time is

$$T(N) = \sum_{i=0}^{k} t_i = 2^{k+1} - 1 = N. \tag{2.3}$$

The serial execution time, assuming that a processor performs one unit of work per unit of time is

$$T(1) = \sum_{i=0}^{k} t_i n_i = (k+1)2^k.$$

The asymptotic speedup S_∞ is then

$$S_\infty = \frac{T(1)}{T(N)} = \frac{(k+1)2^k}{2^{k+1}-1} \cong \frac{k+1}{2} \cong \log_2 \sqrt{N} \text{ for large } N.$$

Due to the equivalence of definitions (a) and (b) for A, it follows that the average parallelism depends only upon the number N of subtasks of the application $\mathcal{A}$

$$A = S_\infty \cong \log_2 \sqrt{N}. \tag{2.4}$$

Now assume that communication delays are taken into account, more precisely that at the end of π_i, $0 \leq i \leq k$, all n_i processors active during π_i communicate their results to the n_{i+1} processors active in the period π_{i+2} in constant time τ. The speedup then becomes

$$S_\infty(\tau) = \frac{(k+1)2^k}{2^{k+1} - 1 + (k\tau)} \tag{2.5}$$

Many current machines have relatively long communication times. When τ increases the speedup S_∞ can be made as small as we like, independent of n. For example

$$S_\infty(\tau) < 1 \quad if \quad \tau > \frac{2^k(k-1)+1}{k}. \tag{2.6}$$

If we insist that the average parallelism A be a characterization only of the software system $\mathcal{A}$, then the relationship of A as given by (2.4) with the speedup $S_\infty(\tau)$ as given by (2.5) becomes loose and the bounds of (2.1) can be easily violated as shown by our example computation.

It follows that the average parallelism has to reflect also the characteristics of the specific parallel hardware used, in particular it should depend upon the communication delays. For our example, one might redefine A for a particular hardware architecture $\mathcal{H}$ with communication delay τ as $A(\tau) = S_\infty(\tau)$ with $S_\infty(\tau)$ given by (2.5). Now $A(\tau)$ will be a characterization of the pair $(\mathcal{A}, \mathcal{H})$. In this case the average parallelism $A(\tau)$ of an application $\mathcal{A}$ can be used for scheduling decisions on the parallel hardware $\mathcal{H}$ as proposed in [12].

This approach could be satisfactory for a system with nearly constant communication costs. But, in general, τ is a function of the architecture of the system, and the number of processors in the system, n. The value of τ ranges from $\log_2 n$ for a hypercube, to $\sqrt{n}$ for a grid, and n for a ring architecture [10]. Call $\tau(n)$ the communication delay in a system with n processors and note that the average parallelism $A(\tau_{(N)})$ is determined considering a configuration with $n = N$ processors. Then $A(\tau_{(N)})$ has the following expression

$$A(\tau_{(N)}) = \frac{(k+1)2^k}{2^{k+1} - 1 + k\tau(N)}. \tag{2.7}$$

Consider now the case when application $\mathcal{A}$ of Figure 1 runs on a two processor system. A simple analysis shows that the parallel execution time $T(2)$, is in this case

$$T(2) = 2^{k-1}(k+2) + \tau(2)(2^k - 1). \tag{2.8}$$

The corresponding speedup is

$$S(2) = \frac{T(1)}{T(2)} = \frac{(k+1)2^k}{(k+2)2^{k-1} + \tau(2)(2^k - 1)}. \tag{2.9}$$

For this case the lower limit specified by (1) is violated when

$$\tau(2) > (2^k - 1)/(2^{k+1} - 2 - k) \tag{2.10}$$

In particular, the lower limit in (2.1) is always violated when $\tau(2) > 1$.

But the communication delay depends upon the size of the parallel machine and not upon the size of the problem. For example consider a hypercube with N processors and denote $\tau(2) = \tau$. Then $\tau(N) \simeq \tau \log_2 N \simeq \tau(k+1)$. Even if only two processors of this hypercube are allocated to our problem, hence we have a speedup, $S(2)$, the average parallelism A will depend upon $\tau_{(N)}$. Indeed messages from the other $N-2$ processors maybe routed through the two nodes allocated to our problem and cause additional communication delays. The lower bound specified by (2.1) will be violated when $S(2) < \frac{2A(\tau_{(N)})}{1+A(\tau_{(N)})}$. This happens when

$$\tau > \frac{1}{(k+1)(2 - \frac{k}{(2^k-1)})}. \tag{2.11}$$

The limitations of the average parallelism as a characterization of the software structure $\mathcal{A}$ were discussed in the context of a very simple model. A more accurate model needs to take into account more intricate aspects of communication and control of a parallel computation. For example, in addition to the system architecture and number of processors in the system, the communication delays depend upon the amount of data transferred. Synchronization effects add to the difficulties of the problem. When random execution times for all processors active in the period π instead of strictly

deterministic execution times are considered, then A depends upon the number of processors active in every period and upon the coefficient of variation of the distribution function of the execution time in that period [9].

The performance issues are even more difficult when nonalgorithmic load imbalance effects like those due to memory failures and retrys, communication failures and so on must be taken into account, [9]. Clearly, simple models of parallel computations which ignore the effects of communication delays are of limited usefulness from a practical standpoint.

It is useful to recall that Amdhal's law [2] which provides another bound on performance based upon a single parameter characterization of a parallel application has been questioned [6]. If f is the fraction inherently sequential of a computation, Amdhal's law states that

$$S(n) < \frac{1}{f + \frac{1-f}{n}}. \tag{2.12}$$

Empirical results reported in [6] indicate considerably larger values for the scaled speedup. We see that the speedup depends upon the problem size, when the problem size increases the fraction of the strictly sequential part of the computation f often decreases.

To characterize independently the parallel application and the parallel hardware seems an elusive, if not an impossible task. As pointed out in [10], given a parallel algorithm as a DAG $D = (V, B)$ whose nodes $v \in V$ are computational tasks with given execution time requirements, and whose arcs $b \in B$ represent both time and functional dependence then performance analysis can be carried out only after a schedule for the algorithm on a particular machine has been constructed. A schedule S of D is a finite set of triples $S \subset V \times P \times T$ with P the set of processors available and T the time such that

1. $\forall v \in V$ then is a triple (v, p, t) which specifies the processor p where v will run, and the starting time t.

2. There are no two triples (v, p, t), $(v', p, t) \in S$ such that $v \neq v'$. Two different tasks cannot run on the same processor at the same time.

3. If $(u, v) \in A$ and $(v, p, t) \in S$ then either there is another triple (u, p, t') $\in S$ with $t' \leq t - \sigma_u$ (u, the predecessor of v may be started on the same processor p at an earlier time, such that it completes by time t; σ_u is the execution time of u) or there is another triple (u, p', t')

$\in S$ such that $t' \leq t - \sigma_u - \tau$ (u has been scheduled on a different processor p' such that by time t, it has completed *and* its results have been transmitted to p).

Communication is the crucial aspect of parallel computing and no model that ignores it is capable of providing insight. Slightly more sophisticated arguments along the same lines can be found in Dasgupta [3].

3 A Model of Parallel Execution Based Upon Threads of Control and Events

In this section a model of parallel execution which takes into account the effects of communication and control is introduced. The model was proposed suitably for homogeneous parallel computations like the ones supported by the SPMD paradigm [9]. The model can be easily extended for the non-homogeneous case when the threads of control perform different computations. The model allows us to identify classes of parallel computations when the asymptotic speedup is non-zero and captures subtle aspects of parallel computations, for example, that a high processor utilization does not always lead to a high speedup, but failes to quantify other aspects like blocking.

3.1 The model: Basic assumptions

The model describes a parallel computation C as a collection of P threads of control and E events. Informally a *thread of control* is an agent capable to perform some work in behalf of C and an *event* is an explicit action performed by a thread of control in order to coordinate its activity with other threads of control. In a wider sense an event is a change of state of a thread of control.

A parallel computation C with P threads of control and E events is described by its *characteristic function* g defined by $E = g(P)$. The model is based upon two assumptions:

(a) *Conservation of work.* Any work required by a computation $C(1)$ with one thread of control has to be performed by one of the threads of control of $C(P)$, the parallel computation with P threads of control.

(b) $W(P)$, *the work required by a parallel computation is an increasing function of the number of threads of control,* P.

The first assumption needs little justification. It is an immediate consequence of the view that a thread of control is an agent performing some work in behalf of C. To carry out a computation with P threads of control simply means to redistribute in some fashion the work which otherwise would be carried out by only one thread. Call this constant amount of work reflecting the work conservation principle W_{cons}.

The second assumption is supported by the following arguments. An event is associated with every communication and control act. Any thread of control needs to communicate with other threads at least at the instance when it is initiated when some work is assigned to it, and at the termination time, when it has to communicate its results. It follows that $g(P)$ is an increasing function of P. Moreover any event requires a small amount of additional work, say θ, to be carried out by the thread of control when an event occurs. Let $W_{cc}(P)$ denote the additional amount of work required by $C(P)$ for communication and control. The previous arguments show that $W_{cc}(P)$ given by

$$W_{cc}(P) \geq \theta \times E = \theta \times g(P) \tag{3.1}$$

is an increasing function of P. Thus, while $W_{cc}(P)$ might not increase monotonically, it is plausible to assume that the variations from the trend are small and that $W_{cc}(P)$ is increasing. But $W(P)$, the work carried by $C(P)$ consists of at least two components the first one, W_{cons}, independent of P and the second one, $W_{cc}(P)$, an increasing function of P

$$W(P) = W_{cons} + W_{cc}(P). \tag{3.2}$$

Important properties of C are its duration T and work intensity $w(t)$. The *work intensity* is the actual measure of work performed as a function of time, e.g., operations per second. The work associated with C is

$$W = \int_0^T w(t)dt. \tag{3.3}$$

In view of the previous discussion the work intensity $w^i(t)$ associated with thread ϕ^i has two components

$$w^i(t) = w^i_{cons}(t) + w^i_{cc}(t) \tag{3.4}$$

where $w^{(i)}_{cons}$ is from the work assigned to the thread by virtue of the work conservation principle, and the second one, $w^i_{cc}(t)$ represents the work inten-

sity for communication and control. Note that $w^i_{cons}(t)$ and $w^i_{cc}(t)$ cannot be non-zero simultaneously.

The duration T of $C(P)$ is expected to depend upon the number P of threads of control of $C(P)$. The work performed by the ith thread, ϕ^i, is

$$W^i = \int_0^T w^i(t)dt = \int_0^T w^i_{cons}(t)dt + \int_0^T w^i_{cc}(t)dt. \tag{3.5}$$

The total work required by $C(P)$ is thus

$$W(P) = \sum_{i=1}^{P} W^i = \sum_{i=1}^{P} \int_0^T w^i(t)dt. \tag{3.6}$$

The thread ϕ^i can be in one of two states at time t: *active* if $w^i_{cons}(t) > 0$, and *suspended* if $w^i_{cons}(t) = 0$. When the thread ϕ^i is suspended then it can be either *communicating* if $w^i_{cc}(t) > 0$, or *blocked* if $w^i_{cc}(t) = 0$, as shown in Figure 2 (which is explained later).

A parallel computation $C(P)$ may have several threads of control ϕ^i active at any given time t. Call $\nu_{act}(t)$ *the number of threads active*, $\nu_{cc}(t)$ *the number of threads communicating* and $\nu(t)$ the *number of threads non-blocked*, either active or communicating at time t. Note that $\nu(t)$ is sometimes called the *profile of the parallelism*, [12]. Clearly

$$\nu(t) = \nu_{act}(t) + \nu_{cc}(t) \tag{3.7}$$

and

$$1 \leq \nu(t) \leq P \text{ for } 0 \leq t \leq T(P). \tag{3.8}$$

We say that the system changes its state at time t if $\nu_{act}(t-\epsilon) \neq \nu_{act}(t+\epsilon)$ for any positive ϵ. To mark the change of state, we say that an *event* $e(t)$ has occurred at time t. If thread ϕ^i has changed state at time t, we denote the event by $e^i(t)$. Note that we make the following convention: an event is associated only with the transition from active to suspended state. The duration of an event is equal to the time spent by the thread in the suspended state.

For the sake of convenience we consider that all P threads of control are created at time $t = 0$ and exist until time $t = T(P)$. In addition, we assume that there are two intervals of time when only one thread of control is active, $\nu(t) = 1$ for $0 \leq t \leq t_s$ and for $T(P) - t \leq t \leq T(P)$. The times t_s and t_e are called *start parallel* and *end parallel* times, respectively. At t_s the thread of control active initially, ϕ^1, explicitly performs an action to assign

a part of work to a thread ϕ^2, which changes its state from suspended to active, ϕ^1 is called a *parent* of ϕ^2. This process has to be repeated at least P times, such that each thread must become active at least once.

In case of a serial computation, only one thread of control is active at any time t. Without loss of generality, we assume that a serial computation, $C(1)$ has only one thread of control active at any time t.

In a parallel computation $C(P)$ changes of state occur due to the need for communication and control. Such communication must take place at least once during the lifetime of ϕ^i, otherwise ϕ^i would not be able to coordinate its work with other threads. Communication between two threads of control, ϕ^i and ϕ^j takes place as the sender, say ϕ^i, performs an explicit action of making available private information, and the receiver, say ϕ^j, performs an explicit action to access this information. The terms *sender* and *receiver* are considered in the sense of information theory and the E/T model is not concerned with the mechanisms used for communication. Sending and receiving may be performed in different ways, such as by message passing or by accessing shared data.

Every time a thread ϕ^i performs an explicit action for communication or control, our model assumes the behavior illustrated in Figure 2. Note that the workload intensities associated with the thread ϕ^i exhibit the following behavior

$$\begin{array}{lll} w^i_{cons}(t) > 0 & \text{for} & t \leq t_{suspend} \text{ and } t \geq t_{reactivate} \\ w^i_{cons}(t) = 0 & \text{for} & t_{suspend} < t < t_{reactivate} \\ w^i_{cc}(t) > 0 & \text{for} & t_{suspend} < t < t_{block} \text{ and } t_{resume} < t < t_{reactivate} \\ w^i_{cc}(t) = 0 & \text{for} & t_{block} \leq t \leq t_{resume} \end{array} \tag{3.9}$$

The additional work for communication and control, θ in (3.1) reflects the work associated with the periods when $w^i_{cc}(t)$ is non-zero. A blocking period may occur only for some events. For example, in a message passing system, an asynchronous write operation does not experience blocking, while a synchronous read may experience blocking if the data has not been received yet. In a shared memory system, both reading and modifying a shared data element may experience blocking.

It is difficult to predict the duration of a blocking period, therefore, knowing that an algorithm for matrix multiplication requires say, $\mathcal{O}(n^2/p^{2/3})$ communication steps, for two $n \times n$ matrices, using p processors [1], does not translate easily into statements concerning communication time.

To simplify the discussion, let us assume that the work intensity associated with thread ϕ^i is constant when the thread is not blocked,

$$w^i(t) = \begin{cases} I & \text{if } \phi^i \text{ is active or communicating} \\ 0 & \text{if } \phi^i \text{ is blocked.} \end{cases} \tag{3.10}$$

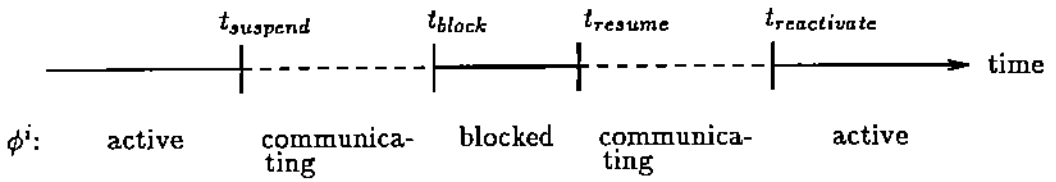

Figure 2: The states of the thread of control ϕ^i when an event $e_{t_{suspend}}$ occurs.

In this case if C is performed using a serial execution, i.e., as $C(1)$, with only one thread, then there is a clear relationship between $W(1) = W_{cons}$ and $T(1)$, the execution time with one thread only:

$$W(1) = T(1) \cdot I. \tag{3.11}$$

The relationship between $W(P)$ and $T(P)$ is explored next. Consider the case described by equation (3.10). Then the work intensity can be expressed as

$$w(t) = w^i(t) \cdot \nu(t) = I \cdot \nu(t) \tag{3.12}$$

with $\nu(t)$ the number of threads of control non-blocked at time t. The work $W(P)$ associated with $C(P)$, can be expressed as

$$W(P) = \int_0^{T(P)} w(t)dt = I \int_0^{T(P)} \nu(t)dt. \tag{3.13}$$

Define the *expected number of threads non-blocked* (active or communicating) as

$$\bar{\nu}(P) = \frac{1}{T(P)} \int_0^{T(P)} \nu(t)dt. \tag{3.14}$$

From (3.13) and (3.14) it follows that

$$T(P) = \frac{W(P)}{I} \frac{1}{\bar{\nu}(P)} . \tag{3.15}$$

Similarly

$$w_{cons}(t) = w^i_{cons}(t) \cdot \nu_{act}(t) = I \cdot \nu_{act}(t) \tag{3.16}$$

with $\nu_{act}(t)$ the number of threads of control active at time t.

The work W_{cons} can be expressed as

$$W_{cons} = \int_0^{T(P)} w_{cons}(t)dt = I \int_0^{T(P)} \nu_{act}(t)dt. \tag{3.17}$$

Define the *expected number of threads active* as

$$\bar{\nu}_{act}(P) = \frac{1}{T(P)} \int_0^{T(P)} \nu_{act}(t)dt. \tag{3.18}$$

Then we have

$$W_{cons} = IT(P)\bar{\nu}_{act}(P). \tag{3.19}$$

But $W_{cons} = W(1) = IT(1)$ hence

$$\frac{T(1)}{T(P)} = \bar{\nu}_{act}(P). \tag{3.20}$$

3.2 The speedup and its asymptotic behavior

The speedup $S(P)$ is defined as the ratio of the computation time with one thread of control to the computation time with P threads, $P > 1$, that is

$$S(P) = \frac{T(1)}{T(P)} . \tag{3.21}$$

First observe, that according to (3.20),

$$S(P) = \bar{\nu}_{act}. \tag{3.22}$$

Note that in the framework of the E/T model, *the speedup is equal to the expected number of threads active, performing work assigned by virtue of the conservation law.* The speedup is less than ν, the expected number of threads running (active or communicating). Since $\nu_{act} \leq \nu \leq P$, it follows that

$$S(P) \leq P. \tag{3.23}$$

Consider now the asymptotic behavior of $S(P)$. From (3.15) and (3.19) it follows that

$$S(P) = \frac{W(1)}{W(P)}\bar{\nu}(P). \tag{3.24}$$

We introduce the *efficiency*, $b(P)$ as the ratio between the expected amount of work per thread of control using P threads, $\bar{w}(P) = W(P)/P$, and the work $W(1) = W_{cons}$ using one thread (sequential execution), that is

$$b(P) = \frac{W(P)}{PW(1)}\,. \tag{3.25}$$

Note that $W(P) \geq W(1)$. Hence

$$b(P) \leq 1/P. \tag{3.26}$$

The expected fraction $a(P)$ of non-blocked threads in $C(P)$ is given by

$$a(P) = \frac{\bar{\nu}(P)}{P} \quad 0 \leq a(P) \leq 1. \tag{3.27}$$

Then we have

$$S(P) = \frac{a(P)}{b(P)}\,. \tag{3.28}$$

The study of the asymptotic behavior of $S(P)$ when P becomes very large is reduced to the problem of the asymptotic behavior of $a(P)$ and $b(P)$. From the definitions of $a(P)$, $b(P)$ and $W(P)$, the following conclusion can be drawn:

(a) For parallel computations with $g(P) = \mathcal{O}(P)$, we have $b(P) = 1/P +$ *constant* for large P and hence $S(P) <$ *constant* for a large number of threads of control.

(b) For parallel computation with $g(P) = \mathcal{O}(P^n)$ with $n \geq 2$, $b(P)$ is an increasing function of P and hence $S(P)$ tends to zero asymptotically.

Let us now consider the case of *scaled* execution [6] where the computation size increases linearly with the number of processors (threads of control) used, namely

$$W(1) = \mathcal{O}(P)$$
$$T(1) = \mathcal{O}(P). \tag{3.29}$$

Scaled speedup $SS(P)$ is defined for scaled execution by equation (3.21). The quantities $a(P)$ and $b(P)$ are analogously defined and relations (3.23) through (3.28) hold. The asymptotic behaviors of $b(P)$ and $SS(P)$ in this case are as follows:

(sa) For parallel computation with $g(P) = \mathcal{O}(P)$, $SS(P)$ is an increasing function of P.

(sb) For parallel computation with $g(P) = \mathcal{O}(P^2)$, $SS(P) <$ *constant* for large P.

(sc) For parallel computation with $g(P) = \mathcal{O}(P^n)$ and with $n \geq 3$, $SS(P)$ tends to zero for large P.

It seems reasonable to question whether scaled execution and parallel computations with $g(P) = \mathcal{O}(P)$ are compatible with one another. A computation is called *embarrassingly parallel* if

$$W(P) = W(1) + \text{ constant}, \nu(t) = P \text{ for } t_0 \leq t \leq T(P) - t_0$$

and

$$E = \text{ constant } \times P \ .$$

This terminology is especially appropriate if the constants involved are small. For these computations we have $a(P) = 1 - 2t_0/T(P)$ which is asymptotically 1 and $b(P) = (W(1) + \textit{constant}/(P \cdot W(1))$ which is asymptotically $1/P$. Thus for embarrassingly parallel computations, we have

$$SS(P) = P + \mathcal{O}(1). \tag{3.30}$$

Such computations arise when the work can be partitioned into P parts at the beginning and then done completely independently by the processors. Thus we can achieve optimal speedup for such computations.

Divide and conquer algorithms may provide scaled speedup nearly as great. Let $P = 2^k$ and assume conservatively that

1. The work after each division of the problem is the same as $W(1)$.

2. The events take place only at dividing the computation up and recombining the results.

Then we see that $W(P) \leq W(1) \log P$, $E = \mathcal{O}(P)$ and we compute that, asymptotically,

$$\begin{array}{lcl} b(P) & = & \log P/P \\ T(P) & \leq & T(1) \times 2 \log P \\ \bar{v}(P) & \leq & \text{constant} \times P \\ SS(P) & = & \mathcal{O}(P/\log P). \end{array} \tag{3.31}$$

In [4] the *average parallelism* was proposed as a high level characterization of software structure. The average parallelism is defined as the speedup, given an unbounded number of processors. The previous discussion shows that there are parallel algorithms, such that $S(P)$ or $SS(P)$ tend asymptotically to zero, hence the average parallelism does not provide a useful characterization of such applications.

3.3 Analysis when E = g(P) is a convex function

The qualitative analysis continues with the case when $g(P)$ is a convex function. Several algorithms we have examined suggest that $g(P)$ is often a convex function of P as well as increasing.

Theorem 3.1 *If $E = g(P)$ is increasing and convex function for $P \geq 1$ then $W(P)$ is also convex. Let P_{smax} be the unique solution of*

$$P = [\alpha + g(P)]/g'(P). \tag{3.32}$$

where $\alpha = W(1)/\theta$. Then $S(P)$ is increasing for $P < P_{smax}$ and decreasing for $P > P_{smax}$.

Proof. We have $W(P) = W_{cons} + \theta \times g(P)$ so $W(P)$ is convex if $g(P)$ is. The speedup $S(P)$ may be expressed by

$$S(P) = T(1)/T(P) = \frac{P}{1 + [\theta/W(1)]g(P)} = \frac{W(1) + \theta \times g(P)}{W(1) + \theta \times g(P)} =$$

$$\frac{W(1)/I}{(W(1) + \theta \times g(P))/(PI)}. \tag{3.33}$$

Combine $\theta/W(1)$ into the constant $1/\alpha$ and differentiate this expression to obtain

$$S'(P) = \frac{\alpha + g(P) - Pg'(P)}{(\alpha + g(P)^2} \ . \tag{3.34}$$

Set $g(P) = Ph(P)$ with $h'(P) \geq 0$ by the convexity assumption. Then we have

$$S'(P) = \frac{\alpha - P^2h'(P)}{(\alpha + g(P))^2} = [\alpha + Ph(P) - P(h(P) + P^2h'(P))]/(\alpha + Ph(P))^2. \tag{3.35}$$

Since $h'(P)$ is positive, (3.35) is zero exactly once. A manipulation of (3.34) allows one to obtain (3.32) as asserted by the theorem.

We may use this result to provide estimates of maximum speedups and the corresponding number of threads of control (processors) for a few cases as given in Table 1. Note that the speedups given are maximums, other factors (e.g., lack of load balancing) can make them smaller.

3.4 The expected amount of work per thread of control

To study the asymptotic behavior of a parallel computation C when, P, the number of threads of control increases, we first investigate the behavior of the function

$$\bar{w}(P) = \frac{W(P)}{P}. \tag{3.36}$$

Consider first computations C with $E = \mathcal{O}(P)$ where each thread of control ϕ^i experiences only a few communication events, in addition to the events to initialize and terminate ϕ^i. An example of such a computation is a plotting computation when each thread operates in isolation upon its private data to create its part of the plot and makes the results available at the end. In this case

$$\bar{w}(P) = \frac{W_{cons}}{P} + \mathcal{O}(1). \tag{3.37}$$

For such parallel computations the expected amount of work per thread of control is a monotonically decreasing function of the number of threads of control as shown in Figure 3.

Table 1: Values of maximum speedups and corresponding P_{smax} for $\alpha = W(1)/\theta = 10^6, 10^4, 10^2$ and $g(P) = P^n$ for $n = 1.5, 2, 2.5$, and 3.

$g(P) = P^{1.5}$	$\alpha = 10^6$	$\alpha = 10^4$	$\alpha = 10^2$
Speedup	5291	245	11
P_{smax}	16000	736	34

$g(P) = P^2$	$\alpha = 10^6$	$\alpha = 10^4$	$\alpha = 10^2$
Speedup	500	50	5
P_{smax}	1000	100	10

$g(P) = P^{2.5}$	$\alpha = 10^6$	$\alpha = 10^4$	$\alpha = 10^2$
Speedup	128	20	3.2
P_{smax}	213	34	5

$g(P) = P^3$	$\alpha = 10^6$	$\alpha = 10^4$	$\alpha = 10^2$
Speedup	53	11.4	2.4
P_{smax}	79	17	4

Consider now parallel computations with $E = \mathcal{O}(P^2)$, for example when each thread of control communicates with every other thread of control during its lifetime. In this case the asymptotically expected amount of work per thread of control is

$$\bar{w}(P) = \frac{W(P)}{P} = \frac{W_{cons}}{P} + \mathcal{O}(P). \tag{3.38}$$

The amount of work per thread of control exhibits a minimum for a certain P_{opt} and it is a monotonically increasing function of P when $P > P_{opt}$. Clearly, P_{opt} increases as W_{cons} increases. For a given δ the range of P such that $W(P) - W(P_{opt}) < \delta$ is usually fairly large, $\bar{w}(P)$ is relatively flat around its minimum. This case is illustrated in Figure 4.

If $g(P) = \mathcal{O}(P^n)$ with $n \geq 3$ then $\bar{w}(P)$ increases rapidly with P and massive parallelism is unlikely to be advantageous unless W_{cons} is enormous.

In conclusion $\bar{w}(P)$ provides a useful *signature* of $\mathcal{C}$. This signature indicates that massive parallelism is truly advantageous only when $E = \mathcal{O}(P)$. In this case the $\bar{w}(P)$ is a monotonically decreasing function of P so that

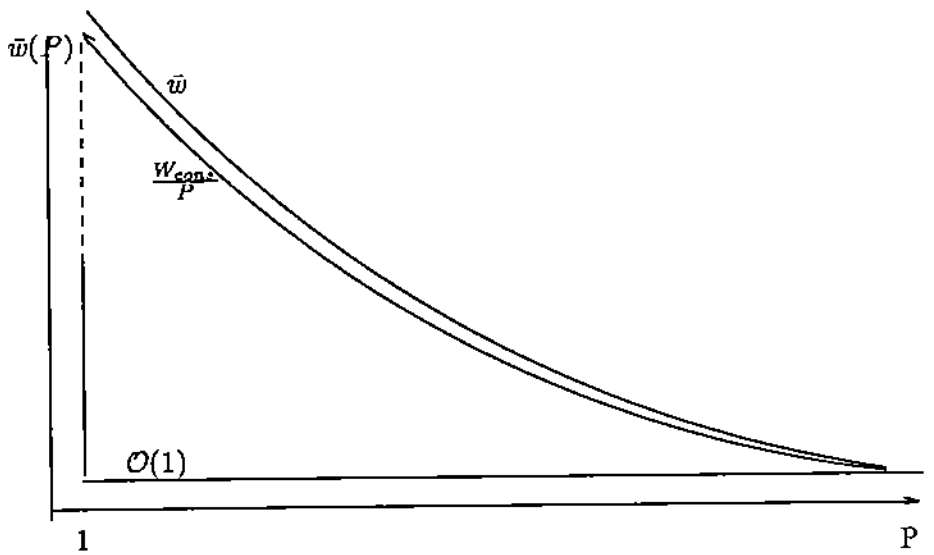

Figure 3: The expected work per thread of control $\bar{w}(P)$ function of the number of threads of control, P, for a parallel computation of a fixed problem size with $E = \mathcal{O}(P)$ according to equation (3.37).

if reasonable load balancing is achieved among the threads of control then the processors are used efficiently. When $E = \mathcal{O}(P^2)$ then there exists an optimum number of threads of control which minimize the expected workload per thread, and $\bar{w}(P)$ is relatively flat around that minimum. If the characteristic function $E = g(P)$ is either a polynomial of degree $n \geq 3$ or similar type of behavior, then $\bar{w}(P)$ exhibits a minimum for a lower value of P_{opt} and $\bar{w}(P_{opt})$ is higher than in the previous case. The efficiency of computations in this class is rather sensitive to the choice of P, the number of threads of control.

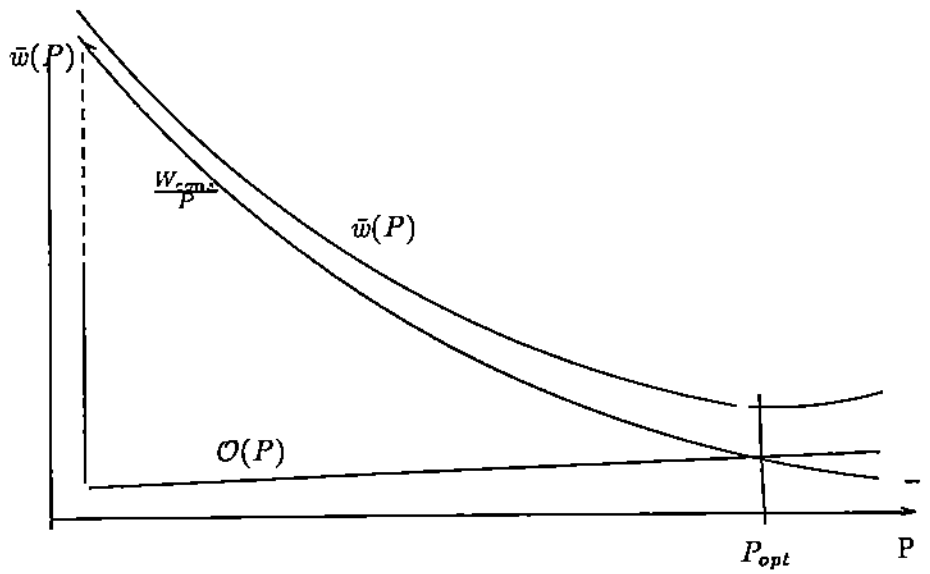

Figure 4: The expected work per thread of control $\bar{w}(P)$ function of the number of threads of control, P, for a parallel computation of fixed problem size with $E = \mathcal{O}(P^2)$ according to equation (3.38).

4 Static Load Balancing Using the E/T Model

The previous analysis is based on the SPMD paradigm involving a collection of computations with similar but not identical characteristics. We now show how the characteristic function $g(P)$ can be used for a computation with several collections of computations, each collection having similar but not identical characteristics. Let k index the K collection, each with its

characteristic function $g_k(P)$ and its work variables W^k_{cons} and $W^k_{cc}(P)$. We assume that θ is the same for each collection (since it is primarily a hardware characteristic) and the work for each collection is

$$W^k(P_k) = W^k_{cons} + \theta \times g_k(P_k) \qquad k = 1, 2, ..., K$$

where P_k is the number of processors allocated to the kth collection.

The goal of load balancing is to make the work per processor the same throughout. Thus if there are P processors, the assignment of processor P_k, $k = 1, 2, ..., K$, to give a balanced load must satisfy, for all k and j,

$$\frac{W^k(P_k)}{P_k} = \frac{W^j(P_j)}{P_j} \tag{4.1}$$

plus

$$P_1 + P_2 + ... + P_k = P \tag{4.2}$$

This is a system of K nonlinear equations in the K variables P_k. The system may be expressed more explicitly as follows:

$$W^1_{cons}/P_1 + \theta \times g_1(P_1)/P_1 = W^k_{cons}/P_k + \theta \times g_k(P_k)/P_k \qquad k = 2, 3, ..., K$$

$$P_1 + P_2 + ... + P_k = P \tag{4.3}$$

This system of equations is rather tractible for commonly occurring characteristic functions $g_k(P)$.

Note that there are implicit constraints on the system (4.3). The solutions P_k must be integers between 1 and P, which means that exact load balancing will rarely be possible. There are some architectures, e.g., bus based machines such the Encore or Sequent machines or the reconfigurable arrays machines such as PASM [11] where these are the only constraints. For other machines such as Cedar or the hypercubes (e.g., Intel IPSC/1 or NCUBE) processors should be allocated in blocks rather than individually.

If all the collections of computations have a linear $g(P)$, then the system (4.3) is linear in the variables $R_k = 1/P_k$ except for the final equation. Thus the system is easily reduced to a single nonlinear equation which is readily solved. We illustrate the technique for two examples with $K = 3$. We may assume that $\theta = 1$ so the systems (4.3) are

$$\begin{aligned} 1000R_1 + 51 &= 4000R_2 + 12 \\ &= 2500R_3 + 25 \\ P_1 + P_2 + P_3 &= 64 \end{aligned} \qquad (4.4a)$$

$$\begin{aligned} 100R_1 + 200 &= 4000R_2 + 10 \\ &= 3000R_3 + 80 \\ P_1 + P_2 + P_3 &= 512 \end{aligned} \qquad (4.5a)$$

These are transformed into

$$1000/(4000R_2 - 38) + 1/R_2 + 25000/(4000R_2 - 13) = 64 \qquad (4.4b)$$

$$100/(4000R_2 - 190) + 1/R_2 + 3000/(4000R_2 - 70) = 512 \qquad (4.5b)$$

The solution of these systems are, respectively,

$$\begin{aligned} P_1 &= 11.088 \sim 11, \quad P_2 = 31.206 \sim 31, \quad P_3 = 21.705 \sim 22 \\ P_1 &= -0.88 \sim 1, \quad P_2 = 52.27 \sim 52, \quad P_3 = 460.61 \sim 459 \end{aligned}$$

In the second case (4.5) the constraint $P_k \geq 1$ was imposed and the other values modified.

5 Experimental Results

We have applied this methodology to a computation involving partial differential equations [8]. Figure 5 shows the spread of the number of events per thread of control as the number of threads of control (processors) increases from 4 to 128. This data supports the basic assumption of the SPMD paradigm that the computations are homogeneous with a reasonably small variation in behavior.

Figure 6 shows further data from the same computation, namely the fraction of time each thread of control is active (doing useful work) when it is not blocked (waiting for data from another thread of control). One can create such figures when capturing event histories by simply identifying the type of event and then post processing the event/thread of control traces. In this case, very poor performance is indicated (which is why the computation was being studied). In [9] further analysis using the E/T model is presented and the causes of the poor performance are identified.

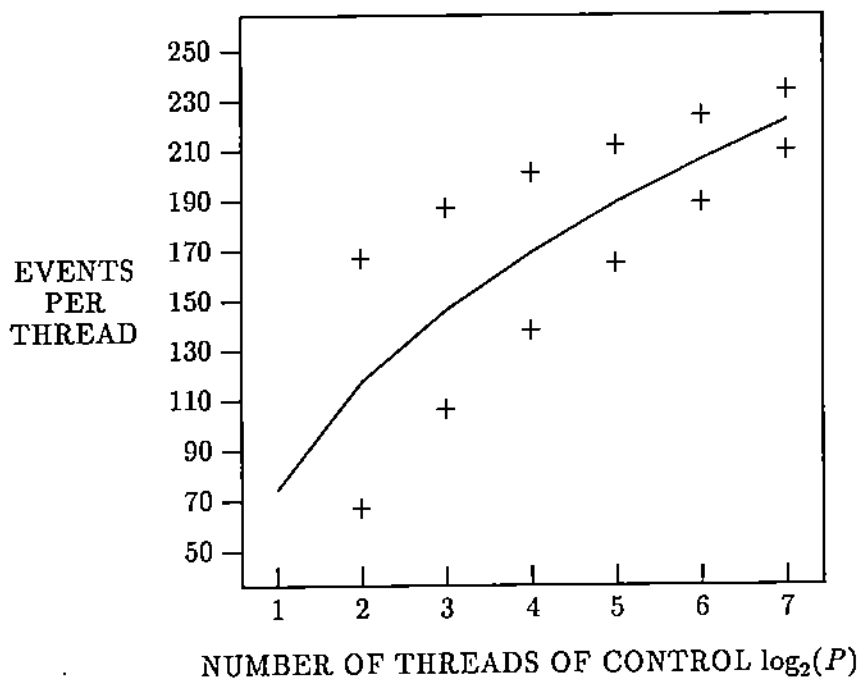

Figure 5: The expected number of events per thread of control (solid line) and a 95 percent confidence interval for it.

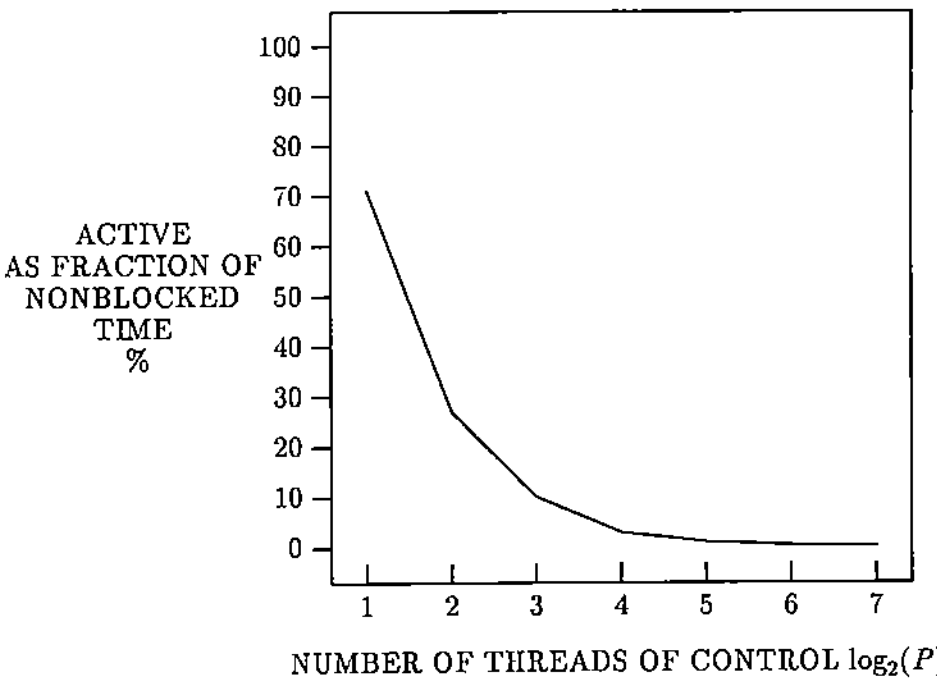

Figure 6: The expected *active time* fraction of the non-blocked time per thread.

6 Literature

1. A. Aggarwal, A.K. Chandra and M. Snir, "On communication latency in PRAM computations", Research Report RC 14973, IBM Research, September 1989.

2. G.M. Amdhal, "Validity of the single processor approach to achieving large scale computing abilities", in Proc. AFIPS, vol. 30, pp. 438–485, 1967.

3. S. Dasgupta, "A hierarchical taxonomic system for computer architectures", IEEE Computer vol. 23, no. 3, pp. 64–74, March 1990.

4. D.L. Eager, J. Zahorjan and E.E. Lazowska, "Speedup versus efficiency in parallel systems", IEEE Trans. on Computer Systems, vol. 38, no. 3, pp. 408–423, March 1989.

5. G. Fox et al., "Solving problems on concurrent processors", Prentice Hall, 1988.

6. J.L. Gustafson, "Reevaluating Amdahl's law", CACM 31, 5, pp. 532–533, May 1988.

7. D.W. Krumme, A.L. Couch and B.L. House, "The TRIPLEX tool set for the NCUBE multiprocessors", Technical Report Tufts University, June 1989.

8. D.C. Marinescu, J.R. Rice and E. Vavalis, "Performance of iteration methods for distributed memory processors", CSD-TR 979, Computer Sciences Department, Purdue University, May 1990.

9. D.C. Marinescu and J.R. Rice, "Synchronization and load imbalance effects in distributed memory multiprocessor systems', CSD-TR-1000, Computer Sciences, Purdue University, July 1990.

10. C.H. Papadimitriou and M. Yannakakis, "Towards an architecture-independent analysis of parallel algorithms", Proc. of 20th Annual ACM Symp. on Theory of Computing, pp. 510–513, May 1988.

11. Seigel, H.J., L.J. Siegel, F.C. Kemmerer, P.T. Mueller, Jr., H.E. Smalley, Jr., and S.D. Smith, "PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition", *IEEE Transactions on Computers*, Vol. C-30, December 1981, pp. 934–947.

12. K.C. Sevcik, "Characterizations of parallelism in applications and their use in scheduling", Proc. of Sigmetrics and Performance'89, Berkeley, PA, May 1989.