1990

Communication and Control in SPMD Parallel Numerical Computations

Dan C. Marinescu

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

E. A. Vavalis

Report Number:
90-981

COMMUNICATION AND CONTROL IN SPMD PARALLEL NUMERICAL COMPUTATIONS

Dan C. Marinescu
John R. Rice
E. A. Vavalis

CSD-TR-981
May 1990

COMMUNICATION AND CONTROL IN SPMD PARALLEL NUMERICAL COMPUTATIONS*

D.C. Marinescu
J.R. Rice
and
E.A. Vavalis

Computer Sciences Department
Purdue University
Technical Report CSD-TR-981
CAPO Report CER-90-23
May, 1990

* Work supported in part by ARO grant DAAG03-86-K-0106.

Communication and Control in SPMD Parallel Numerical Computations*

Dan C. Marinescu, John R. Rice, and Emmanuel A. Vavalis

Computer Sciences Department
Purdue University
West Lafayette, IN 47907, USA

May 8, 1990

Abstract

This paper investigates communication and control in SPMD parallel computations, and introduces the Events/Threads of control, E/T, model which allows qualitative and quantitative analysis of parallel execution. The principal component of the E/T model is the characteristic function $g(P)$ which relates the number of events to the number P of threads of control (usually a processor). Many properties of a computation follow from the behavior of $g(P)$, such as limits on the potential speedup. The model includes the effects of reads and writes in communication, algorithmic blocking, work intensity, etc. It is most appropriate for SPMD (Single Program Multiple Data) computations that are common in scientific applications. An experiment is described briefly which relates the the detailed behavior of a parallel computation, observed through monitoring, with the high level characterization provided by the E/T model.

1 Overview

The E/T 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 perform 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.

Modeling and analysis of numerical problems which lend themselves to the Same Program Multiple Data, SPMD, paradigm are the focus of our investigation [11], [12]. Communication and control latency can strongly influence the performance of these computations and we use the E/T model to analyze this influence. Informally a *SPMD computation* is performed whenever all processing elements, PE's of a parallel machine execute the same program on different data. SPMD computations lead to a collection of similar threads of control therefore their modeling and analysis seems an easier task than the analysis of non-homogeneous computations with a large number of unrelated threads of control.

The E/T model can used for *qualitative analysis* of a parallel computation C, an analysis based upon the study of the *characteristic function* g, which relates the number of events, E, and the number of threads of control P, $E = g(P)$ of C. An *optimal* parallel computation with P threads of

*Work supported in part by ARO grant DAAG03-86-K-0106.

control is a computation characterized by a linear function $g(P)$. If $g(P)$ cannot be expressed as a polynomial then there is little hope that $\mathcal{C}$ will ever be performed efficiently. Consider two parallel computations $\mathcal{C}_1$ and $\mathcal{C}_2$ with P threads of control which represent two different implementation of an algorithm $\mathcal{A}$ or implementations of two different algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$ which perform the same task. If the characteristic functions of $\mathcal{C}_1$ and $\mathcal{C}_2$ are in the relation $g_1(P) < g_2(P)$ for $P_1 \leq P \leq P_2$ then we have a high degree of confidence that $\mathcal{C}_1$ performs better than $\mathcal{C}_2$ in the range $P_1 \leq P \leq P_2$ for a wide variety of parallel architectures.

Whenever more information about a parallel computation $\mathcal{C}$ is available, for example when the sequence of events occurring in a thread of control can be identified or when the characteristics of the parallel machine executing $\mathcal{C}$ are known then the E/T model is capable of providing more accurate assessments about the expected performance of $\mathcal{C}$. A *quantitative analysis* can only be carried out if the threads of control exhibit some form of invariance to data dependencies, in other words if data dependencies can only alter the timing but not the order of events in a thread of control.

A first type of quantitative analysis is a *static analysis*. This is an analysis of the *mapping* from a directed acyclic graph, $\mathcal{D}$ to a parallel computation $\mathcal{C}$. The E/T model is used to determine the computation and communication workload. The computation workload can be analyzed at different levels, e.g., the amount of computation between two consecutive events, the workload per thread of control, and the total workload of $\mathcal{C}$. Similarly, communication workload can be characterized by the amount of data transferred during a single event, the amount of data transferred per thread of control, and the total amount of data transferred at the computation level. The effects of synchronization and blocking are not captured by the static analysis.

A second type of quantitative analysis is the *dynamic analysis* concerned with *schedules* which associate times with events. At this stage a detailed knowledge of the hardware is necessary in order to determine the time required to perform computations and the time to send and receive data. Alternatively, performance monitors and execution traces for a selection of data may provide sufficient knowledge to carry out a dynamic analysis. This analysis reveals the effects of synchronization and blocking.

Static analysis is often susceptible of an analytical approach but the dynamic models only seldom lead to a tractable analysis. Often dynamic models can be constructed only through monitoring the actual execution of $\mathcal{C}$ on a particular parallel system.

2 Qualitative Analysis Of Parallel Computation in the E/T Model

2.1 Basic assumptions

We propose a model for parallel computing based upon events and threads of control, the *E/T model*. A parallel computation $\mathcal{C}$ 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 $\mathcal{C}(1)$ with one thread of control has to be performed by one of the threads of control of $\mathcal{C}(P)$, 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 $\mathcal{C}$. 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

$$W_{cc}(P) \geq \theta \times E = \theta \times g(P) \tag{2.1}$$

which 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

$$W(P) = W_{cons} + W_{cc}(P). \tag{2.2}$$

A parallel computation C with P threads of control is considered to be *optimal* iff $E = O(P)$, the number of events in C is linear in P.

Some of the algorithms we have encountered exhibit a convex characteristic function $g(P)$. We show that if the characteristic function $g(P)$ is convex then: the speedup has a maximum for some finite $P = P_{smax}$ and it is a concave function for $P > P_{smax}$.

2.2 Threads of control and events

The basic idea of the model is to describe a parallel computation C in terms of *threads of control* and *events*. 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

$$W = \int_0^T w(t)dt. \tag{2.3}$$

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

$$w^i(t) = w^i_{cons}(t) + w^i_{cc}(t) \tag{2.4}$$

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 intensity 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

$$W^i = \int_0^T w^i(t)dt = \int_0^T w^i_{cons}(t)dt + \int_0^T w^i_{cc}(t)dt. \tag{2.5}$$

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

$$W(P) = \sum_{i=1}^{P} W^i = \sum_{i=1}^{P} \int_0^T w^i(t)dt. \tag{2.6}$$

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 1 (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*, [14]. Clearly

$$\nu(t) = \nu_{act}(t) + \nu_{cc}(t) \tag{2.6a}$$

and

$$1 \leq \nu(t) \leq P \text{ for } 0 \leq t \leq T(P). \tag{2.7}$$

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 1. Note that the workload intensities associated with the thread ϕ^i exhibit the following behavior

$$\begin{array}{lll} w^i_{cons}(t) > 0 & \text{for} & t \leq t_{suspend} \text{ and } t \geq t_{reactivate} \\ w^i_{cons}(t) = 0 & \text{for} & t_{suspend} < t < t_{reactivate} \\ w^i_{cc}(t) > 0 & \text{for} & t_{suspend} < t < t_{block} \text{ and } t_{resume} < t < t_{reactivate} \\ w^i_{cc}(t) = 0 & \text{for} & t_{block} \leq t \leq t_{resume} \end{array} \tag{2.8}$$

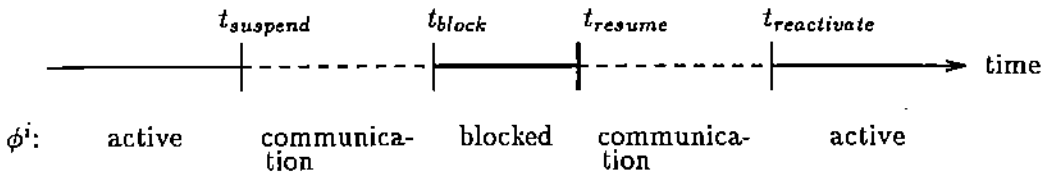

FIGURE 1: The states of the thread of control ϕ^i when an event $e_{t_{suspend}}$ occurs.

The additional work for communication and control, θ in (2.1) 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, $O(n^2/p^{2/3})$ communication steps, for two $n \times n$ matrices, using p processors [1], does not translate easily into statements concerning communication time.

2.3 W,T characterization of a parallel computation

Statements about a computation C can be made when the amount of work, W and the time T required by C are known. To simplify the discussion, let us assume that the work intensity associated with thread ϕ^i is constant when the thread is not blocked,

$$w^i(t) = \begin{cases} I & \text{if } \phi^i \text{ is active or communicating} \\ 0 & \text{if } \phi^i \text{ is blocked.} \end{cases} \tag{2.9}$$

In this case if C is performed using a serial execution, i.e., as $C(1)$, with only one thread as shown in Figure 2a, then there is a clear relationship between $W(1) = W_{cons}$ and $T(1)$, the execution time with one thread only:

$$W(1) = T(1) \cdot I. \tag{2.10}$$

The relationship between W and T is less obvious in case of multiple threads of control, as shown in Figures 2b and 2c, where two alternative computations $C^A(2)$ and $C^B(2)$ are used to perform C. We observe that

$$W(1) < W^{(A)}(2) < W^{(B)}(2), \tag{2.11}$$

but

$$T^{(B)}(2) < T^{(A)}(2). \tag{2.12}$$

Even the simple question of which one of the two variations of $C(2)$, $C^A(2)$ or $C^B(2)$, is better cannot be answered unambiguously, as $C^B(2)$ requires less time, but more work than $C^A(2)$.

The relationship between $W(P)$ and $T(P)$ is explored next. Consider the case described by equation (2.9). Then the work intensity can be expressed as

$$w(t) = w^i(t) \cdot \nu(t) = I \cdot \nu(t) \tag{2.13}$$

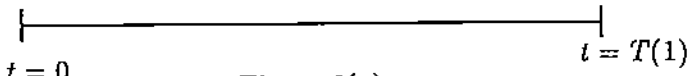

Figure 2(a)

Figure 2(b)

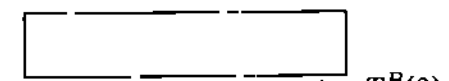

Figure 2(c)

FIGURE 2: (a) Sequential computation $C(1) = < W(1), T(1) >$. (b) Computation reorganized to be $< C^A(2) > = < W^A(2), T^A(2) >$ with two threads of control. (c) Computation reorganized in a different way to be $C^B(2) = < W^B(2), T^B(2) >$ with two threads of control. Solid lines represent work periods and dotted lines blocked periods.

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

$$W(P) = \int_0^{T(P)} w(t)dt = I \int_0^{T(P)} \nu(t)dt. \tag{2.14}$$

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

$$\bar{\nu}(P) = \frac{1}{T(P)} \int_0^{T(P)} \nu(t)dt. \tag{2.15}$$

From (2.14) and (2.15) it follows that

$$T(P) = \frac{W(P)}{I} \frac{1}{\bar{\nu}(P)} . \tag{2.16}$$

Similarly

$$w_{cons}(t) = w^i_{cons}(t) \cdot \nu_{act}(t) = I \cdot \nu_{act}(t) \tag{2.17}$$

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

The work W_{cons} can be expressed as

$$W_{cons} = \int_0^{T(P)} w_{cons}(t)dt = I \int_0^{T(P)} \nu_{act}(t)dt. \tag{2.18}$$

Define the *expected number of threads active at time t* as

$$\bar{\nu}_{act}(P) = \frac{1}{T(P)} \int_0^{T(P)} \nu_{act}(t)dt. \tag{2.19}$$

Then we have

$$W_{cons} = IT(P)\bar{\nu}_{act}(P). \tag{2.20}$$

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

$$\frac{T(1)}{T(P)} = \bar{\nu}_{act}(P). \tag{2.21}$$

2.4 The expected amount of work per thread of control

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

$$\bar{w}(P) = \frac{W(P)}{P}. \tag{2.22}$$

Consider first computations $\mathcal{C}$ 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

$$\bar{w}(P) = \frac{W_{cons}}{P} + \mathcal{O}(1). \tag{2.23}$$

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.

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

$$\bar{w}(P) = \frac{W(P)}{P} = \frac{W_{cons}}{P} + \mathcal{O}(P). \tag{2.24}$$

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 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.

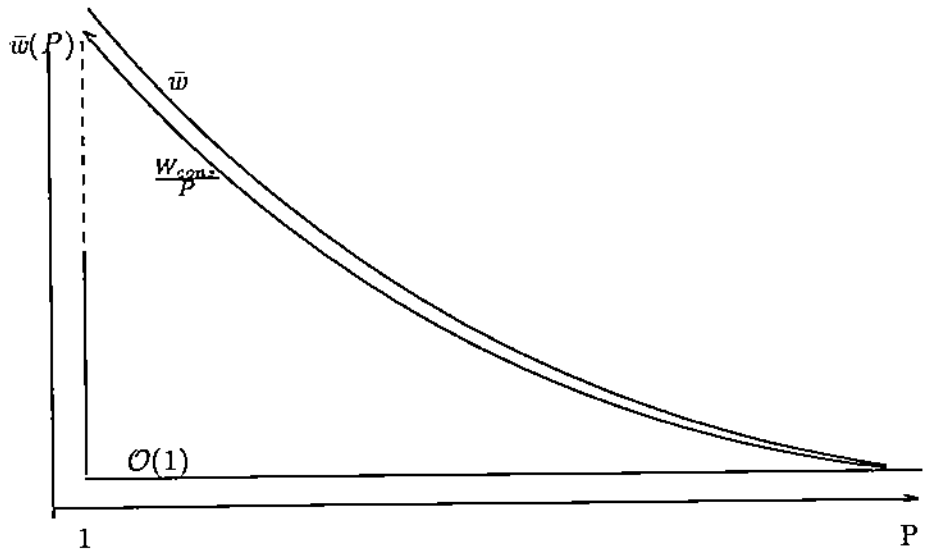

FIGURE 3: The expected work per thread of control $\bar{w}(P)$ function of the number of threads of control, P, for a parallel computation of a fixed problem size with $E = \mathcal{O}(P)$ according to equation (2.23).

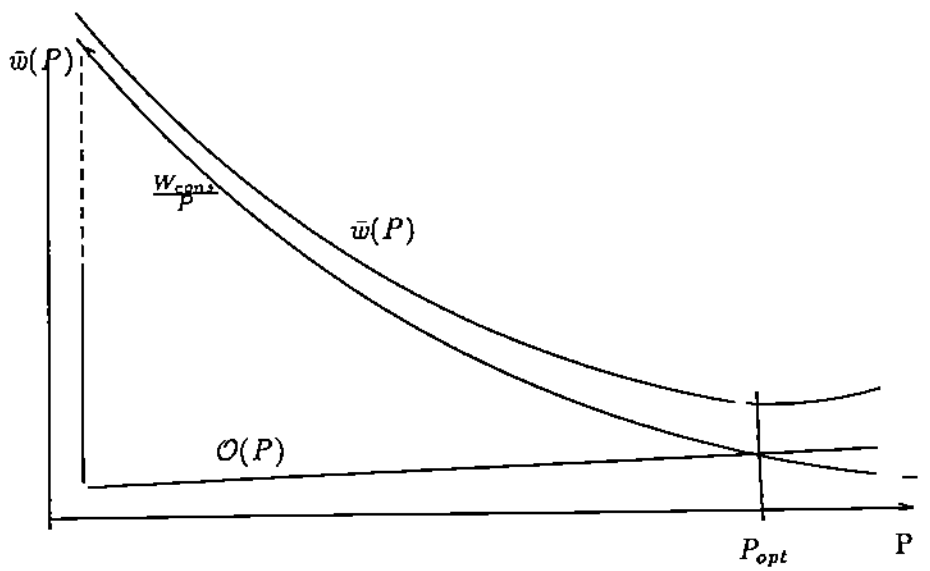

FIGURE 4: The expected work per thread of control $\bar{w}(P)$ function of the number of threads of control, P, for a parallel computation of fixed problem size with $E = \mathcal{O}(P^2)$ according to equation (2.24).

2.5 The speedup

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

$$S(P) = \frac{T(1)}{T(P)} . \tag{2.25}$$

First observe, that according to (2.21),

$$S(P) = \bar{\nu}_{act}. \tag{2.26}$$

Similar results have been reported, see for example [3], but 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

$$S(P) \leq P. \tag{2.27}$$

Consider now the asymptotic behavior of $S(P)$. From (2.16) and (2.20) it follows that

$$S(P) = \frac{W(1)}{W(P)}\bar{\nu}(P). \tag{2.28}$$

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

$$b(P) = \frac{W(P)}{PW(1)} . \tag{2.29}$$

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

$$b(P) \leq 1/P. \tag{2.30}$$

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

$$a(P) = \frac{\bar{\nu}(P)}{P} \quad 0 \leq a(P) \leq 1. \tag{2.31}$$

Then we have

$$S(P) = \frac{a(P)}{b(P)} . \tag{2.32}$$

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 $b(P) = 1/P + \textit{constant}$ for large P and hence $S(P) < \textit{constant}$ 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 [5] where the computation size increases linearly with the number of processors (threads of control) used, namely

$$W(1) = \mathcal{O}(P)$$
$$T(1) = \mathcal{O}(P). \tag{2.33}$$

Scaled speedup $SS(P)$ is defined for scaled execution by equation (2.25). The quantities $a(P)$ and $b(P)$ are analogously defined and relations (2.27) through (2.32) 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

$$SS(P) = P + \mathcal{O}(1). \tag{2.34}$$

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,

$$\begin{array}{lcl} b(P) & = & \log P/P \\ T(P) & \leq & T(1) \times 2 \log P \\ \bar{\nu}(P) & \leq & \text{constant } \times P \\ SS(P) & = & \mathcal{O}(P/\log P). \end{array} \tag{2.35}$$

In [3] 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.

2.6 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 2.1 *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*

$$P = [\alpha + g(P)]/g'(P). \tag{2.36}$$

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

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

$$\frac{W(1)/I}{(W(1) + \theta \times g(P))/(PI)} . \tag{2.37}$$

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

$$S'(P) = \frac{\alpha + g(P) - Pg'(P)}{(\alpha + g(P)^2} . \tag{2.38}$$

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

$$S'(P) = \frac{\alpha - P^2h'(P)}{(\alpha + g(P))^2} = [\alpha + Ph(P) - P(h(P) + P^2h'(P))]/(\alpha + Ph(P))^2. \tag{2.39}$$

Since $h'(P)$ is positive, (2.39) is zero exactly once. A manipulation of (2.38) allows one to obtain (2.36) 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.

2.7 Additional workload due to algorithmic effects

To characterize the work required by a computation C with P threads of control, we have identified two components so far, the intrinsic work, W_{cons} assigned to the threads by virtue of the conservation law and $W_{cc}(P)$ the work for communication and control. However, the transformation from a computation $C(1)$ with one thread of control only to a computation $C(P)$ with P threads of control, often introduces additional work called in the following *algorithmic workload* and denoted by $W_{alg}(P)$, i.e., we have

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

$$W(P) = W_{cons} + W_{cc}(P) + W_{alg}(P). \tag{2.40}$$

We study alternative parallel ways to do the same work and exclude from considering complete changes of algorithms. However, we do allow the work to be transformed in various ways using simple equivalences of operations. Specifically,

$$C(1) = 4 * 5 * 9 * 412$$

is equivalent to $C^A(2)$ defined by

Thread 1	*Thread 2*
4 * 5 = 20	9 * 412 = 3707
20 * 3707 = 74160	

but is not equivalent to $C^B(2)$ defined by

Thread 1	*Thread 2*
log 4 + log 5 = 1.301	log 9 + log 412 = 3.569
Blocked	1.301 + 3.569 = 4.871
$10^{4.871} = 74160$	

Similarly

C(1) = sort (list1), sort (list2), sort (list3), concatenate (list1,list2,list3)

is equivalent to $C^A(2)$ defined by

Thread 1	*Thread 2*
sort (list1)	sort (list2)
sort (list3)	concatenate (list1,list2) = list4
concatenate (list4,list3)	

The question of when two computations are equivalent is a subtle one, which we do not attempt to make precise here because most parallelizations of algorithms introduce new work, called $W_{alg}(P)$ above. However, this concept is useful in heuristic discussions of the work of different algorithms for the same task. The range of possible effects of parallelization of an algorithm are very large, but there seem to be two common ones. Later we examine realistic algorithms in some detail, but here we consider a simple algorithm.

(a) *Algorithmic overhead associated with individual events.* An algorithm may have internal information that needs to be updated when new external information is received. For example, an iteration on one domain receives values from an adjacent part of another domain. These values affect the error estimates along the domain boundary, which in turn, affect the estimate of convergence ratio and relaxation factors. All these estimates and factors must be recomputed when new information is received. A very high rate of events could distort the computation until most of the work done is recomputing these estimates and factors, instead of carrying out the iteration. This example of algorithmic overhead behaves like communication and control work (indeed, the update computations are to control the numerical behavior of the iterations).

It is plausible to merge this work into W_{cc} even though it appears to be algorithmic work. There seems to be no advantage in carrying along two independent sources of work proportional to the number of events.

(b) *Algorithmic overhead associated with (long) event free periods.* The computation in one thread might profitably use information from other threads to reduce its algorithmic work. For example, a search party of 10 men covering an area will be forced to have all 10 men search the entire area unless there is communication between them about which areas have already been searched. Many analogs of this simple situation exist in computational search algorithms. Another example occurs when long periods between events force one thread to save data for future communications, buffers or queues can become full, requiring extra work to save data in special ways, or the thread can even become idle (introducing an unnecessary event) waiting to empty space for saving data. A more subtle example is a set of parallel iterations where long periods of no communication means that the iterations are running but not accomplishing anything useful.

The *average event interval* is $T(P) \cdot P/E$ and this type of algorithmic overhead is modeled by

$$W_{alg}(P) = \mathcal{O}\left(\frac{T(P) \cdot P}{E}\right). \tag{2.42}$$

Thus, $W_{alg}(P)$ is proportional with the expected time between successive events. The larger this interval, the more likely it is that a thread will perform unnecessary work, or it will duplicate work done by another thread.

This informal presentation shows that the avoidance of additional work for the algorithm occurs only in embarrassingly parallel computations.

Note that long event-free periods are usually associated with other types of undesirable effects besides work duplication, effects related to practical implementation of partial computations on real machines with finite memory. Whenever the *lifetime of partial results* defined as the interval between the instance partial results are produced by thread ϕ^i and the instance they are consumed by thread ϕ^j, is large, then the memory requirements for $\mathcal{C}$ become substantial as observed in [10].

The algorithmic workload and the lifetime of partial results discussed in this section are difficult to be captured and they will be largely ignored in this study. But it is conceivable to assume that a second conservation law of the type

$$W_{cc}(P) + W_{alg}(P) = f(P) \tag{2.43}$$

is valid and captures the effect that for a given P the sum of the work for communication and control and algorithmic work is constant and any gain obtained by reducing the number of events is compensated by increasing the level of work duplication.

2.8 Extensions of the model to non-SPMD computations

The SPMD parallel computations are homogeneous, all threads of control ϕ^i, $1 \leq i \leq P$, exhibit similar behavior. In the framework of the E/T model, this translates into the fact that the characteristic functions of all threads of control $g^i(P)$ are identical

$$g^i(P) = \frac{1}{Pg(P)} \quad 1 \leq i \leq P. \tag{2.44}$$

In case of non-SPMD computations, the dissimularities among different threads of control is reflected by a partial ordering of threads, based upon the number of events associated with each thread. Call ϕ^ℓ the thread which has the largest number of events. In such a case, the E/T model requires one to identify the thread ϕ^ℓ and to study its characteristic function $g^\ell(P)$. Of course, things can become even more complex when the thread with the largest number of events is data dependent or its behavior changes widely with the data.

3 Quantitative Analysis of SPMD Parallel Computations

In this section we focus upon quantitative analysis of a parallel computation $\mathcal{C}$ and attempt to estimate measures of performance such as speedup, execution time, processor utilization, etc. Such an analysis is possible only if $\mathcal{C}$ exhibits only limited data dependencies. Dynamic computations in which the actual sequence of events in every thread of control are data dependent lead to an intractable analysis. Fortunately, most SPMD computations satisfy this condition, while the timing of different events in a thread of control may change depending upon the data, the actual sequence of computations and events occurring in a thread of control is invariant to input data.

First, we consider a *static analysis* which is an analysis of the *mapping* from a directed acyclic graph, $\mathcal{D}$ to $\mathcal{C}$. The computation and communications workloads can be analyzed at different levels as discussed in the overview. The effects of synchronization and blocking are not captured by

the static analysis. The *dynamic analysis* is concerned with *schedules* which associate times with events. At this stage a detailed knowledge of the hardware is necessary.

The transition from the static to the dynamic stage of the quantitative analysis is difficult. A "superposition" property which would allow the extension of results obtained in stage one to stage two would be desirable but it seems that the occurrence of such a property is an exceptional event. Some of the problems encountered in this area are due to the difficulties to estimate the communication time between two processors, π_i and π_j. This time depends upon factors as

(a) The architecture of the system $\mathcal{H}$ and the number of processors in the system n. For example if we call τ the communication delay for a message of unit length then $\tau(n)$ is of the order of $\sqrt{n}$ for a grid, $\log_2 n$ for a hypercube and n for a ring interconnection network.

(b) The communication software, the communication protocols, the routing strategy, etc.

Effects unrelated to the parallel computation $\mathcal{C}$ but determined by the need to share resources in a multiuser system can only be considered during the dynamic analysis. Such effects are: fragmentation in communication, the need to split a large message into a number of packets, or processor sharing, in case of multiuser systems. These effects are extremely difficult to be captured by any model.

While static analysis is often susceptible to an analytical approach the dynamic models only rarely lead to a tractable analysis. Often dynamic models can be validated only through the process of monitoring $\mathcal{C}$. Static analysis may be useful to make decisions concerning scheduling in a multiuser environment. For example if the thread ϕ^i expects a large message from the thread ϕ^j then the operating system of the node where ϕ^i runs may suspend it, run another process and return to it after the message has been received. Performance measures as *the average degree of parallelism*, [3] are clearly related to stage one in our approach and can be used successfully to make scheduling decisions as pointed out in [14].

3.1 Static analysis – Mapping in the E/T model

Mapping in the context of the E/T model is the process of deciding which computations and which problem data are assigned to every thread of control of $\mathcal{C}$. The two aspects of mapping, computation mapping and data mapping are closely related.

Let us for the moment consider the general case when a parallel algorithm $\mathcal{A}$ is given as a directed acyclic graph, DAG, $\mathcal{D} = (V, A)$ whose nodes $d_i \in V$ are computational tasks with given workload requirements and whose arcs, $a_i \in A$, represent both temporal and functional dependencies. In addition, the arcs have associated with them communication load requirements. The mapping of the DAG $\mathcal{D}$ to a parallel computations $\mathcal{C}$ is a process of deciding how many threads of control should $\mathcal{C}$ have and how different nodes of $\mathcal{D}$ will be assigned to the threads of control. The *computation mapping* consists of the following steps

(a) Choose the number P of threads of control ϕ^i, $1 \leq i \leq P$.

(b) Group together a number of nodes of $\mathcal{D}$ and assign them to ϕ^i.

The *data mapping* is the process of assigning problem data to different threads of control. Data mapping follows computation mapping and allows us to

(c) Define the sequence of actions performed by every thread ϕ^i, $1 \leq i \leq P$.

(d) Determine the sequence of events associated with every thread ϕ^i, $1 \leq i \leq P$.

Note that the SPMD execution corresponds to the case when the computation mapping assigns all nodes of $\mathcal{D}$ to every thread of control ϕ^i. The data mapping makes the execution of the P threads of control different.

Formally, the computation mapping is described as follows. Denote by N the number of nodes of $\mathcal{D} = (D, A)$, $N = |D|$. Then the work associated with the DAG, $\mathcal{D}$ is characterized by a $N \times 1$ vector $\mathbf{W}_{\mathcal{D}}$

$$\mathbf{W}_{\mathcal{D}} = [w_j] \tag{3.1}$$

with w_j the work associated with node $d_j \in V$. The mapping from $\mathcal{D}$ to $\mathcal{C}$ is characterized by the $(P \times N)$ *computation mapping matrix* $\mathcal{M}_{\mathcal{D}}$

$$\mathcal{M}_{\mathcal{D}} = [m_{ij}] \tag{3.2}$$

with

$$\begin{aligned} &m_{ij} = 1 \quad \text{if } d_j \textit{ is mapped into } \phi^i,\ 1 \leq i \leq P \text{ and } 1 \leq j \leq N \\ &m_{ij} = 0 \quad \textit{otherwise.} \end{aligned} \tag{3.3}$$

The integer $m_i = \sum_{j=1}^{N} m_{ij}$ is called the *grouping factor of* ϕ^i and represents the number of nodes of $\mathcal{D}$ assigned to ϕ^i, with $1 \leq i \leq P$.

The data mapping associated with a domain $\mathcal{B}$ decomposed into K subdomains, b_j, is characterized by the $(P \times K)$ *data mapping matrix* $\mathcal{Q}_{\mathcal{B}}$

$$\mathcal{Q}_{\mathcal{B}} = [q_{ij}] \tag{3.4}$$

with

$$\begin{aligned} &q_{ij} = 1 \quad \text{if } b_j \textit{ is mapped into } \phi^i,\ 1 \leq i \leq P \text{ and } 1 \leq j \leq N \\ &q_{ij} = 0 \quad \textit{otherwise.} \end{aligned} \tag{3.5}$$

The work performed by $\mathcal{C}(P)$ is characterized by a $P \times 1$ vector $\mathbf{W}_{\mathcal{C}}$,

$$\mathbf{W}_{\mathcal{C}} = [w^i] \tag{3.6}$$

with w^i the work assigned to ϕ^i by the mapping $\mathcal{M}$. Clearly,

$$\mathbf{W}_{\mathcal{C}} = \mathcal{M}_{\mathcal{D}} \mathbf{W}_{\mathcal{D}} \tag{3.7}$$

$$w^i = \sum_{j=1}^{N} m_{ij} w^j. \tag{3.8}$$

We consider a nondeterministic model and assume that the work process of $\mathcal{D}$ is stationary and the random variables w_j, $1 \leq j \leq N$ have mean μ_D and variance σ_D. The assumption of a stationarity process is common in the analysis of stochastic processes and it is necessary in order to promote tractability of our model. In general, the w_i are dependent random variables and their dependence is characterized by the *work covariance matrix of* $\mathcal{D}$, $\sigma_{\mathcal{D}}$

$$\sigma_{\mathcal{D}}^2 = [(\sigma_D^2)_{ij}] \tag{3.9}$$

with

$$(\sigma_D^2)_{ij} = \mathrm{Cov}(w_i, w_j). \tag{3.9'}$$

If we assume that the work process associated with the DAG $\mathcal{D}$ is stationary, it follows immediately that the work process associated with the mapping $\mathcal{C}$ is also stationary. The random variables w^i with $1 \leq i \leq P$ have a distribution with mean μ^i and variance σ^i such that

$$\mu^i = m_i \mu_D \tag{3.10}$$

$$(\sigma^i)^2 = m_i \sigma_D^2 + \sum_{j=1}^{m_i} \sum_{k=1}^{m_j} \mathrm{Cov}\ (w^j, w^k). \tag{3.11}$$

The work covariance matrix of $\mathcal{C}$ is $\sigma_{\mathcal{C}}^2$ defined by

$$\sigma_{\mathcal{C}}^2 = [(\sigma_{\mathcal{C}}^2)_{ij}] \tag{3.12}$$

with

$$(\sigma_{\mathcal{C}}^2)_{ij} = \mathrm{Cov}(w^i, w^j). \tag{3.13}$$

An important aspect of mapping is related to load balance. Intuitively, it seems desirable to assign to every thread of control an equal amount of work with the hope that such a load balanced mapping will eventually lead to the shortest possible execution time and to the best possible utilization of resources. An SPMD computation seems an ideal case from the load balance point of view since in this case all threads of control have assigned to them identical workload. But data mapping makes the execution different, the actual instruction execution sequence in each thread of control is different due to data dependencies and a perfect load balance is unlikely to be achieved. For this reason we consider a nondeterministic analysis and regard the workload associated with any thread of control as a random variable.

In addition to such *algorithmic load imbalance effects* there are *non-algorithmic* causes such as hardware failures and retrys, message retransmission, etc. To characterize the load imbalance associated with any given mapping we introduce a load imbalance factor, Δ defined in the following. Denote by Y the workload associated with the most heavily loaded thread of control.

$$Y = \max(w^1, w^2, \ldots, w^i, \ldots, w^P). \tag{3.14}$$

Call $\bar{Y}$ the expected value of the random variable Y and call $\bar{w}$ the mean value of μ^i defined as

$$\bar{w} = \frac{1}{P} \sum_{i=1}^{P} \mu^i. \tag{3.15}$$

Then Δ is defined implicitly as

$$\bar{Y} = \bar{w}(1 + \Delta) \tag{3.16}$$

In this expression $\bar{w}$ is the expected workload per thread of control and $\bar{Y}$ is the expected workload of the most heavily loaded thread of control.

To conclude this section we summarize the parameters which may be used to characterize statically a parallel computation $\mathcal{C}$, in addition to P, E and $W(P)$, which have been discussed previously.

κ^i – The number of events in the thread of control ϕ^i, $1 \leq i \leq P$.

α_j^i – The amount of work performed by the thread ϕ^i between two consecutive events e_j^i and e_{j+1}^i, for $1 \leq i \leq P$, $1 \leq j \leq \kappa^i$.

$\bar{\alpha}^i$ – The expected amount of work associated with an event in thread ϕ^i, computed as the mean value of α_j^i, $1 \leq i \leq P$, $1 \leq j \leq \kappa^i$.

W^i – The total workload associated with the thread ϕ^i, $1 \leq i \leq P$.

μ^i – The expected workload associated with the thread ϕ^i, $1 \leq i \leq P$.

$\bar{w}$ – The expected workload per thread of control.

β_j^i – The amount of data transferred when the event e_j^i in thread ϕ^i occurs, $1 \leq i \leq P$, $1 \leq j \leq \kappa^i$.

$\bar{\beta}^i$ – The expected amount of data transferred per event in thread ϕ^i, computed as the mean value of β_j^i, $1 \leq j \leq \kappa^i$.

β^i – The total communication load associated with thread ϕ^i, $1 \leq i \leq P$.

$\bar{\beta}$ – The expected value of $\bar{\beta}^i$.

Δ – The load imbalance factor.

Finally, we stress again that without the detailed information required to compute a schedule, the performance of a parallel computation can only be estimated. Such estimates may be used to compare different mappings $\mathcal{M}_i$ of a given algorithm $\mathcal{A}$ but no definite statements about the actual execution time, the speedup and/or the efficiency can be made at the stage. Two examples applying these ideas follow.

3.2 Examples

3.2.1 A parallel algorithm for matrix multiplication

Consider a parallel algorithm for matrix multiplication with P threads of control. This algorithm is described and analyzed in [1]. Let A and B be two $(n \times n)$ matrices. The algorithm requires the partitioning of A and B into q^2 disjoint submatrices A_{ij} and B_{jk}. Each submatrix is of size $(m \times m)$. The values of q and m are given by

$$\begin{aligned} q &= P^{1/3} \\ m &= \frac{n}{q} = \frac{n}{P^{1/3}} \,. \end{aligned} \tag{3.17}$$

Without loss of generality assume that $P = 2^a$ and we have $n = mq$ with a and m positive integers. The algorithm proceeds as follows. For every triplet (i, j, k) compute

$$\ell(i, j, k) = (i - 1)q^2 + (k - 1)q + j. \tag{3.18}$$

Clearly $1 \leq \ell(i, j, k) \leq P$ when $(i, j, k) \in [1, q]$. There are two steps:

A. Let the thread $\phi^{\ell(i,j,k)}$ perform the following actions

(A1) – Read the submatrix A_{kj} with m^2 elements.

(A2) – Read the submatrix B_{jk} with m^2 elements.

(A3) – Compute the submatrix $C_{ijk} = A_{im} \times B_{jk}$. This requires $\mathcal{O}(m^3)$ operations.

B. Organize the threads ϕ^ℓ with $1 \leq \ell \leq P$ as q^2 complete binary trees such that $\phi^{\ell(i,1,k)}$ will compute

$$C_{ik} = \sum_{j=1}^{q} C_{ijk} \quad 1 \leq i \leq q,\ 1 \leq k \leq q. \tag{3.19}$$

This addition is done in a pipelined manner. Each group of q threads $\phi^{\ell(i,j,k)}$ with i and k fixed and with $1 \leq j \leq q$ computes the corresponding C_{ik}.

An example with $P = 2^6$ and $n = 12$ is shown in Figure 5. Figure 6 shows the q threads of control used to compute C_{11} for this example.

For this algorithm the total number of events is

$$E = 3P + \frac{P}{q}(2^q - 1) = 3P + P^{\frac{2}{3}}(2^{P^{\frac{1}{3}}} - 1). \tag{3.20}$$

Note the number of events per thread of control ranges from

$$\begin{array}{lll} \kappa^\ell = \mathcal{O}(\log_2 q) & \text{when} & \ell = \ell(i, l, k) \ \text{ to} \\ \kappa^\ell = \mathcal{O}(1) & \text{when} & \ell = \ell(i, q, k). \end{array} \tag{3.21}$$

The expected active period between two consecutive events is the same for all threads and it is equal to the workload required to multiply (add) two submatrices of size $m \times m$

$$\bar{\alpha}^\ell = \mathcal{O}(m^3). \tag{3.22}$$

The expected duration of an event varies from

$$\begin{array}{lll} \beta^\ell = Q(m^2) & \text{when} & \ell = \ell(i, l, k) \ \text{ to} \\ \beta^\ell = Q(m^2 q) & \text{when} & \ell = \ell(i, q, k). \end{array} \tag{3.23}$$

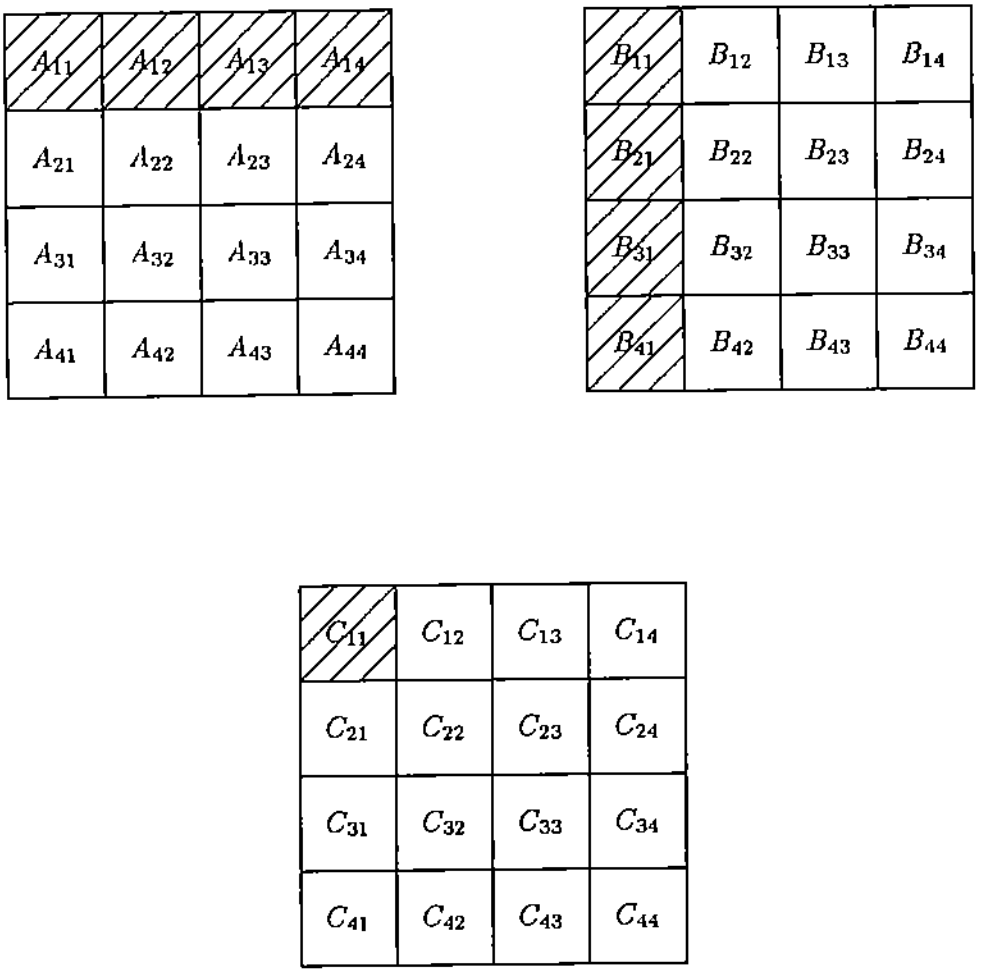

FIGURE 5: A parallel algorithm for matrix multiplication, with P threads of control, ($P = 64$, $n = 12$, $q = P^{1/3} = 4$, $m = \frac{n}{p} = 3$).

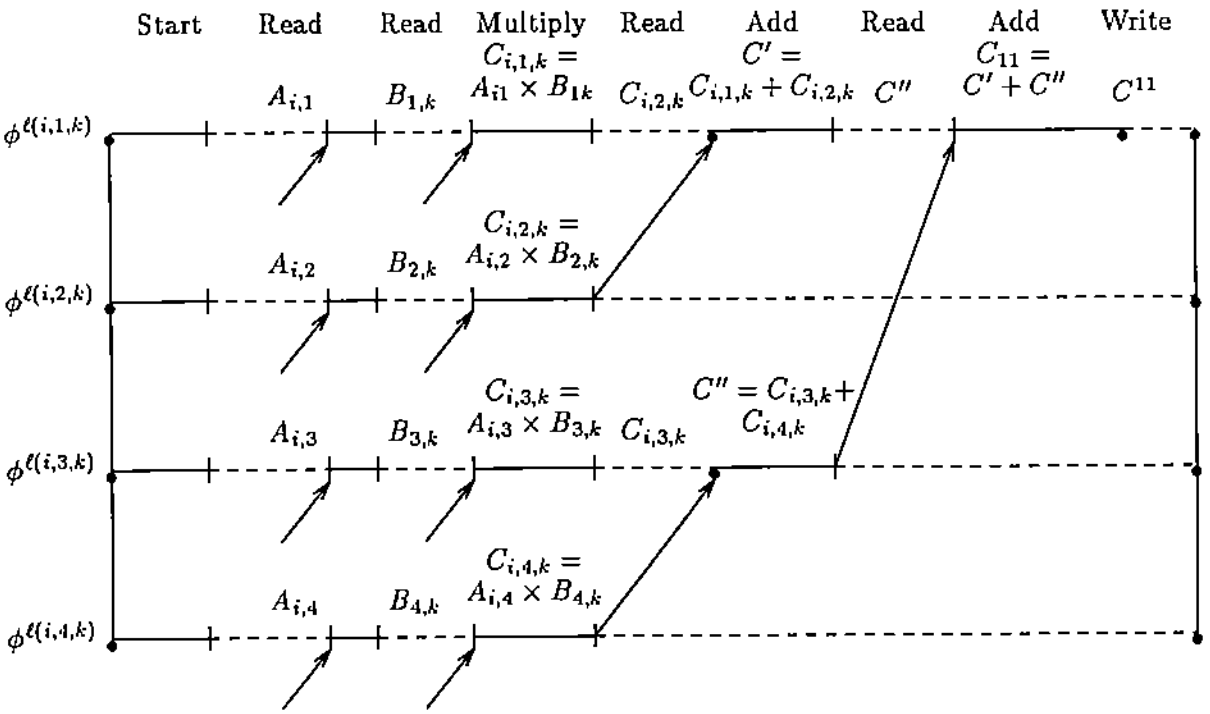

FIGURE 6: The pipelined algorithm for matrix multiplication. The time series of events are shown for $q = 4$ threads of control which compute C_{11}.

3.2.2 A parallel algorithm for banded matrix LU decomposition

The next example is from a concurrent algorithm for banded matrix LU decomposition. A detailed discussion of this algorithm can be found in Chapter 20 of [4] where its implementation on a hypercube is analyzed. The solution of many PDE's can be formulated as the solution of the linear systems of equations

$$AX = B \tag{3.24}$$

where A is a banded $M \times M$ matrix with bandwidth b and where X and B are $M \times n$ matrices of n solutions and free terms, respectively. Only the case when

$$1 << b << M \tag{3.25}$$

is considered here. To solve (3.24) a window of size $m \times m$ ($m = 2b + 1$) is defined and an iterative algorithm is used as follows. At iteration k the window covers the submatrix consisting of rows k to $k + m - 1$ and column k to $k + m - 1$. The process of solving (3.24) consists of three steps.

1. An LU decomposition of A is performed. During the k-th iteration, the following transformations are made:

   Set

   $$L_{k,k} = 1,$$

   and

   $$L_{k+1,k} = \frac{A_{k+i,k}}{A_{k,k}}, \quad 1 \leq i \leq m\ . \tag{3.26}$$

   $$U_{k,k} = \frac{1}{A_{k,k}},$$

   $$U_{k,k+j} = A_{k,k+j}, \quad 1 \leq j \leq m$$

   and

   $$U_{k+i,k+j} = A_{k+i,k+j} - L_{k+i,k} \cdot U_{k,k+j}, \quad 1 \leq i \leq m-1, \quad 1 \leq j \leq m-1.$$

2. A forward reduction of B is performed by

   $$B_{k+i,j} = B_{k+i,j} - B_{k,j} L_{k+i,k} \text{ for } 1 \leq i \leq m,\ 0 \leq j \leq n-1. \tag{3.27}$$

3. A back-substitution from last to first now generates the solution by

   $$X_{k,j} = \frac{B_{k,j}}{U_{k,k}}, \quad 0 \leq j \leq b-1,$$
   $$B_{k-i,j} = B_{k-i,j} - X_{k,j} U_{k-i,j}. \tag{3.28}$$

In this presentation, only the LU decomposition (Step 1) is analyzed. The LU decomposition takes a square matrix A and performs an in-place transformation. The L matrix overlaps the lower triangular part and the U matrix overlaps the upper triangular part of A, including the diagonal. The $m \times m$ window slides along the diagonal such that at step k the k-th row forms the upper side of the window and the k-th column is the left hand side of the window. To simplify the presentation of the algorithm, assume that the current iteration k satisfies the condition

$$k \bmod m = 0 \tag{3.29}$$

and that the matrix element $A_{k+i,k+j}$ is assigned to the thread $\phi^{i_1 q + j_1}$ with

$$i_1 = i \bmod q, \; j_1 = j \bmod q, \tag{3.30}$$

where $q = \sqrt{P}$. This assignment of the matrix elements corresponds to a scatter decomposition as described in [4]. Our analysis corresponds to the case when the size of the window m is a multiple of q and when no pivoting is necessary. In this case $n = m^2/q^2 = m^2/P$ elements of A are handled by every thread of control ϕ^i.

The k-th iteration proceeds as follows:

Step 1. The main diagonal element is updated by ϕ^0 as

$$U_{k,k} = \frac{1}{A_{k,k}} . \tag{3.31}$$

Step 2. The k-th row of A is updated by

$$U_{k,k+i} = A_{k,k+i}, \; 1 \leq i \leq m-1. \tag{3.32}$$

This requires no computation and no communication.

Step 3. The k-th row is transmitted. Each ϕ^i with $0 \leq i \leq q-1$ multicasts $\sqrt{n}$ elements to $(q-1)$ related threads ϕ^{i+jq} with $1 \leq j \leq q-1$. This stage requires $q(q-1)$ events and no computation.

Step 4. The k-th column of A is updated. The threads ϕ^{iq} with $1 \leq i \leq q-1$ compute

$$L_{k+j,k} = A_{k+j,k} \cdot U_{k,k} \text{ with } 1 \leq j \leq m-1. \tag{3.33}$$

Here $(m-2)$ computations are performed. The total amount of work per iteration is $\alpha = 1 + (m-1) + 2(m-1)^2 = 2m^2 - 3m + 3$. The total number of events per iteration is

$$\kappa = 2q(q-1) + q^2 = 3q^2 - 2q = 3P - 2\sqrt{P}. \tag{3.34}$$

The expected amount of work per event is

$$\bar{\alpha}_e = \frac{2m^2 - 3m + 3}{3P - \sqrt{P}} . \tag{3.35}$$

The expected amount of data transferred per event is $\bar{\beta} = \sqrt{\frac{m}{P}}$. Since M iterations are needed and P events are involved in the start-up process, we have

$$E = M\left(3P - \sqrt{P}\right) + P = P(3M+1) - \sqrt{P}M. \tag{3.36}$$

3.3 The dynamic analysis – Schedules in the E/T model

During the previous stage of analysis only static estimates of the performance of parallel computation $\mathcal{C}$ can be obtained since so far the model does not include the concept of time. To extend the model we have to consider a parallel hardware $\mathcal{H}$ with n processors, π_i, $1 \leq i \leq n$ such that $n \geq P$. A *schedule* in the sense of the E/T model is a mapping of every thread of control $\phi^{(i)}$ to a processor π_i.

If detailed information concerning the architecture $\mathcal{H}$ is available then a schedule can be constructed and accurate performance data can be obtained. Following [11] a schedule means to decide for every node of the DAG associated with the computation, $\mathcal{C}$ the processor and the time when the node could execute. In the context of the E/T model, creating a schedule means to determine when every event will occur and how long it will take. The information about the processing speed will allow us to map the amount of work between two consecutive events in a thread of control into the corresponding execution time. The information about communication speed will allow us to map the volume of data to be transferred into a communication time. In this case the time when every event e^i_j occurs is well determined.

Note that at the time a schedule in the E/T sense is constructed, in addition to the *algorithmic events*, the events required by the mapping process, new events may need to be considered. Among the classes of new events we recognize

(a) Events related to the monitoring of $\mathcal{C}$. The next section will discuss in detail this class of events.

(b) Events related to different functions of the operating system. Is is conceivable that in the future more sophisticated operating systems for parallel machines will allow multiprocessing. In this case events related to processor scheduling, memory allocation, etc., will affect the performance of any computation $\mathcal{C}$.

Consider a thread of control ϕ^i assigned to processor π^i and define the following quantities related to ϕ^i, $1 \leq i \leq P$, and to $\mathcal{C}$

α^i_j – The interval of time ϕ^i is active, following the jth event e^i_j in thread ϕ^i.

$\bar{\alpha}^i$ – The expected active time of ϕ^i between two consecutive events.

$\bar{\alpha}$ – The expected active time of $\mathcal{C}$ between two consecutive events.

β^i_j – The duration of the jth event e^i_j in thread ϕ^i.

$\bar{\beta}^i$ – The expected duration of an event in ϕ^i.

$\bar{\beta}$ – The expected duration of an event in $\mathcal{C}$.

γ^i – The expected ratio, $\frac{\bar{\beta}^i}{\bar{\alpha}^i}$, of suspended time to active time in thread ϕ^i.

γ – The expected ratio, $\frac{\bar{\beta}}{\bar{\alpha}}$, of suspended time to active time in $\mathcal{C}$.

If $\bar{\kappa}$ is the expected number of events per thread of control then the average fraction $\eta(P)$ of the lifetime of a thread of $\mathcal{C}$ devoted to computation is given by

$$\eta(P) = \frac{(\bar{\kappa}-1)\bar{\alpha}}{(\bar{\kappa}-1)\bar{\alpha}+\bar{\kappa}\bar{\beta}} \approx \frac{1}{1+\gamma(P)}\,. \tag{3.37}$$

As expected $\eta(P) \to 1$ when $\gamma(P) \to 0$, i.e., when $\bar{\beta} << \bar{\alpha}$, or suspended time is much less than active time. In the case of a one-to-one mapping from threads of control to processors η represents the average processor utilization for $\mathcal{C}$.

The speedup is given by (2.25). We have $T(1) \leq P(\bar{\kappa}-1)\bar{\alpha}$ with equality when $W_{alg}(P) = 0$. Clearly we have

$$T(P) = (\bar{\kappa}-1)\bar{\alpha}+\bar{\kappa}\bar{\beta} \approx \bar{\kappa}(\bar{\alpha}+\bar{\beta}). \tag{3.38}$$

and hence we have the bound

$$S(P) = T(1)/T(P) \leq P\eta(P). \tag{3.39}$$

The last inequality shows that a low processor utilization leads to low speedup, as expected.

3.4 Monitoring parallel computations

A model of a physical system (process or phenomena) is an abstraction which distinguishes between essential and non-essential aspects of the system being modeled and attempts to predict the behavior of the system using a small subset of essential parameters. To validate a model means to compare predictions of the system performance obtained through the analysis of the model for a particular set of input parameters, with actual data gathered from observations of the real system.

In this section some of the issues pertinent to the validation of the E/T model are discussed and it is argued that the parameters necessary for the validation of the model of a parallel computation $\mathcal{C}$ can be obtained easily through monitoring. A detailed discussion of monitoring as well as a formal model for the monitoring process and an architectural model for software and hardware tools for monitoring parallel and distributed software is presented in [8]. Here we discuss only the relationship between monitoring and validation of the E/T model.

The E/T model characterizes a parallel computation $\mathcal{C}$ at two levels, at the *thread of control level* when information about one thread of control is necessary for the model and at the *global level*, when the entire collection of P threads of control are taken into account. At the first level, the total number κ^i of events in thread ϕ^i, as well as the mean time $\bar{\alpha}^i$ between two consecutive events, and the mean duration $\bar{\beta}^i$ of an event, are the parameters of the model. How κ^i, $\bar{\alpha}^i$ and $\bar{\beta}^i$ can be obtained directly by monitoring the execution of ϕ^i is shown later.

At the global level data concerning all threads of control must be gathered. While the global level characterization of $\mathcal{C}$ is not qualitatively different at the thread of control level, the validation of the model through monitoring does require a much larger volume of data to be collected and analyzed. Hence this type of characterization becomes more difficult to validate for massively parallel computations when a large number of threads of control must be monitored and analyzed.

Note also that in case of SPMD parallel computations, all threads of control exhibit quasi-identical behavior and monitoring a sampling of threads (or even one) can provide enough information to estimate accurately the global level characterization of $\mathcal{C}$. For other types of parallel computations (non-SPMD) it is rather difficult to extrapolate the knowledge acquired through monitoring one thread of control to the global characterization of $\mathcal{C}$.

Monitoring a parallel computation $\mathcal{C}$ is the process of recording the events of interest which occur during the lifetime of processes running different threads of control of $\mathcal{C}$. A *monitoring event* is defined [8] as a change of state of a process. An event is "of interest" depending upon the goals of the monitoring process. Monitoring the execution of a parallel computation $\mathcal{C}$ is necessary for debugging a particular implementation of $\mathcal{C}$ and for performance evaluation. During monitoring each *event of interest* generates a trace record which contains all relevant information concerning the thread, e.g. the type of event, the time of the event, the state of the process, etc.

The definition of a monitoring event is more general than the one discussed so far in the context of the E/T model. For example "$I = 5$" can be defined as an event of interest for monitoring, it is indeed a change of state of the corresponding process since a new value is assigned to the variable I, but it is not an event in the sense of the E/T model since the corresponding thread of control does not change its state as a result of this assignment. However, as soon as a change of state of a process is designated as an event of interest and it is decided to monitor it, then this monitoring event becomes also an event in the sense of the E/T model since monitoring means an interruption of the original flow of control done to record the pertinent trace data. On the other hand, any event in the context of the E/T model corresponds to a change of state of the process which embeds the corresponding thread of control, hence it can be monitored.

A first important conclusion of this discussion is that there is a one to one mapping between monitoring events and the events defined in the context of the E/T model. In other words the E/T model is "observable" through monitoring, all the parameters required by the E/T model can be obtained as part of the trace data gathered through monitoring. A second conclusion is that monitoring a parallel computation may affect the timing of events as well as the total number of events in the original computation. This is an undesirable effect associated with every process of measurement.

3.5 Monitoring the performance of iterative methods on a distributed memory system

An experiment to study the performance of iterative methods on a distributed memory system is described in detail in [9]. The experiment uses the parallel ELLPACK, PELLPACK system developed at Purdue [6], running on a 128 processor NCUBE. The TRIPLEX tool set [7] is used to monitor the execution and to collect trace data.

The purpose of the experiment was to collect detailed information concerning the execution of a particular SPMD application, to study how this data relates to the high level characterization of parallelism in the framework of the E/T model, and to investigate how similar or dissimilar the behavior of the threads of control of an SPMD computation are.

The experiment monitors the execution of the code implementing a Jacobi iterative algorithm for solving a linear system of equations, an important component of a parallel PDE solver. To ensure a load balanced execution, the domain decomposer, part of the PELLPACK environment, attempts to assign to every PE an equal amount of computation. A careful selection of the interface points of the neighboring domains is also necessary in order to achieve a balanced communication. The experiment was conducted by taking a problem of a fixed size and repeating the execution with a number of PEs ranging from 2 to 128.

The detailed behavior of all threads of control was captured by recording all the events, marking changes of state for every thread. For every event the TRIPLEX tool creates a trace record, which contains the pertinent information about the event, type, time stamp, PE, amount of data transferred, etc. All the measurements reported are based upon a clock with resolution of 0.1 msec. To minimize the volume of trace data, only events related to communication and control were recorded. Even so, the trace data collected during a single experiment with 128 PEs amounted to about 25 Mbytes.

The raw data were processed in several stages. First, the events outside the scope of Jacobi iterations were filtered out. Then a preprocessing to gather the data required by the E/T model was performed. The active time between events, the duration of an event (read/write) and the length of a blocking period, were obtained by correlating local events, events occurring in the same thread (on the same PE). The time for communication and control was computed as the difference between the duration of an event and the length of its blocking period. To compute the algorithmic blocking (defined as the interval from the instance a read is issued until the corresponding write takes place) it was necessary to correlate non-local events, events involving more than one thread. Finally, a statistical processing was performed in order to obtain data as described in Section 3.3.

Preliminary results indicate that in spite of all the precautions to achieve a well balanced communication, the behavior of threads of control can be considerably different. The number of events, the active time, the total read time per thread, may be within a factor of two from one thread to another as shown in Figure 7 for the expected active time. Figure 8 represents the characteristic function $g(P)$ of the parallel computation, which indicate a $\mathcal{O}(P^2)$ behavior. This happens, since at the end of every iteration a global communication implemented as broadcast-collapse takes place in order to communicate values between threads of control. There is also a global exchange of information every few iterations to obtain information for convergence control. While this could be done by fan-in, fan-out communication in principle, the NCUBE system forces the use of broadcasting which is another source of $\mathcal{O}(P^2)$ events. One familiar with Jacobi iteration would not expect $g(P) = \mathcal{O}(P^2)$. This behavior arises primarily because the NCUBE system does not provide adequate communication utilities, one must use a broadcast when one actually wants to do a multicast to just a handful of "nearby" processors. Using the system provided "supposed" multicast facility actually increases the communication time, because it is implemented using a broadcast. The allocation of threads of control to actual processors can also affect $g(P)$, while the optimal allocation for this problem is NP-hard in general, there are heuristic algorithms which keep the distances between actual processors to a reasonable level. The expected active time per event decreases linearly (Figure 9), while the write time is essentially constant (Figure 10).

The read time per event experiences a sharp increase (Figure 11) and most of it is due to the algorithmic blocking (Figure 12). The active time fraction of the total non-blocked time decreases, due to the $\mathcal{O}(P^2)$ of the number of events per thread (Figure 13).

These results, though preliminary, seem to indicate that the point of view taken by the E/T model, namely, that the work for communication and control is essential in understanding the behavior of parallel computation is well motivated. The measured speedup of a parallel computation C may be disappointingly low, even when a high PE utilization is observed, simply due to the overhead associated with communication and control. This overhead is difficult to measure, but it can be estimated when $g(P)$ is known.

Further experiments are necessary to establish a sound relation between the algorithmic blocking and $g(P)$. The present data show a strong interdependence between the two.

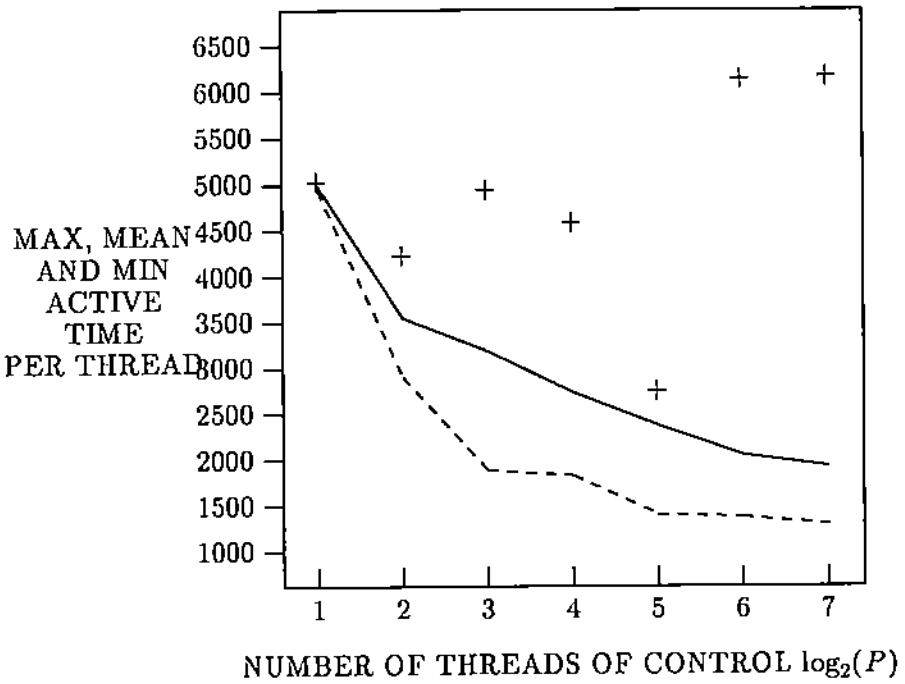

FIGURE 7: The minimum (dashed) the average (solid) and the maximum (plus) *active* time for a thread of control. Irregular domain, 33 × 33 grid.

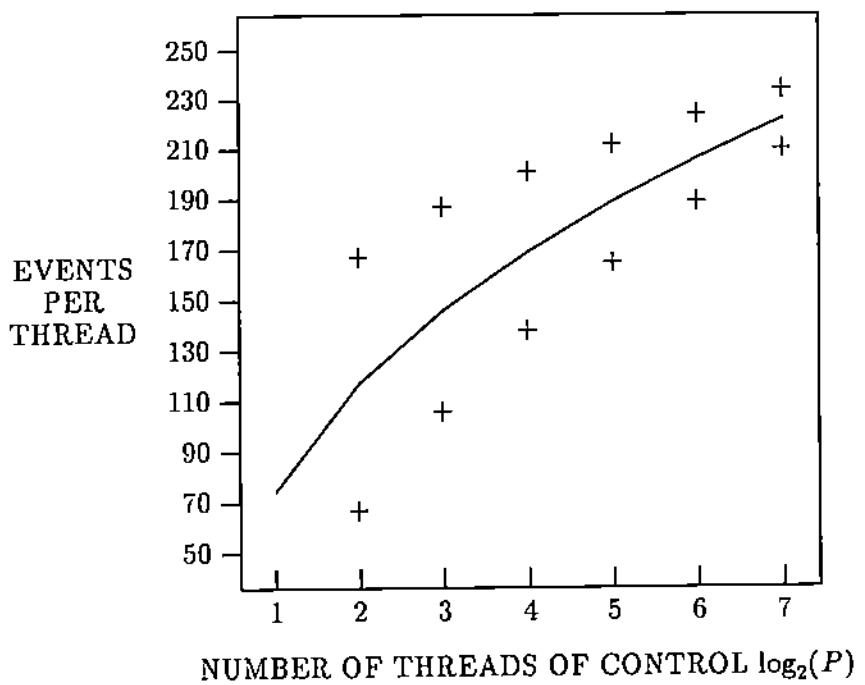

FIGURE 8: The expected number of events per thread of control (solid line) and a 95 percent confidence interval for it.

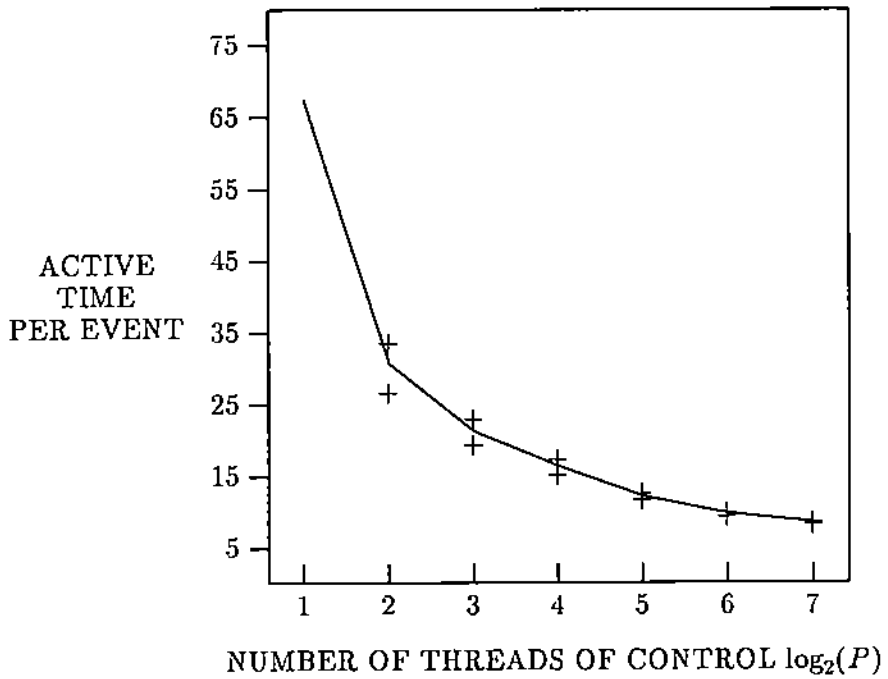

FIGURE 9: The expected time of an *active* period, between two consecutive events (solid line) and a 95 percent confidence interval for it.

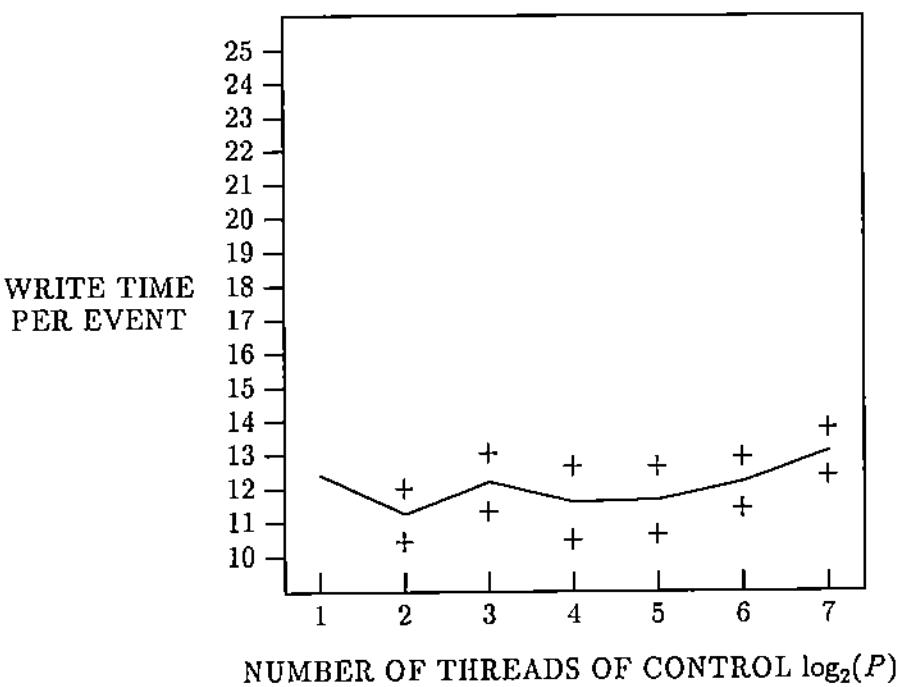

FIGURE 10: The expected time for a single *write* operation (solid line) and a 95 percent confidence interval for it.

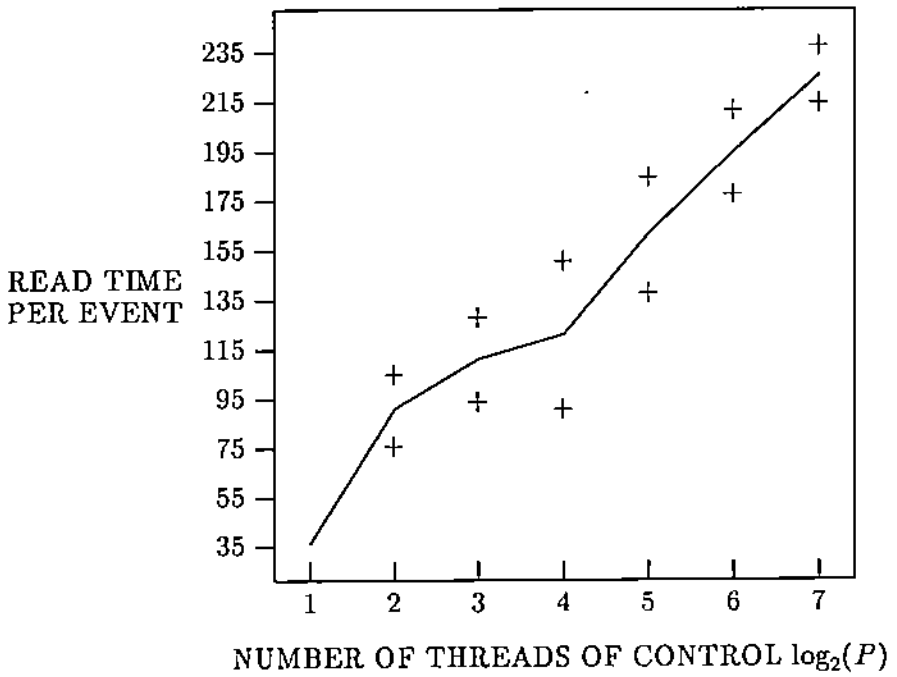

FIGURE 11: The expected time for a single *read* operation (solid line) and a 95 percent confidence interval for it.

4 Conclusions

It is extremely difficult to provide concise characterization of parallel computations, invariant to problem size and especially to the architecture of parallel systems. The E/T model of parallel execution is best suited to the important class of SPMD applications and provides little or no insight for dynamic computations where data dependencies affect the sequence and possibly the number of events in a thread of control.

The characteristic function $E = g(P)$ defined by the E/T model, is less sensitive to the architecture of the parallel system and the problem size than other types of high level characterizations of parallel computations. It allows a uniform treatment for both message passing and shared memory paradigms. The average workload per thread of control, $\bar{w}(P)$ provides a signature of a parallel computation and allows interesting conclusions concerning the asymptotic behavior of different classes of parallel computations.

The E/T model allows quantitative characterizations of the model as well. The static characterization is based upon the computation and data mapping and it is insensitive to timing characteristics (instruction execution rate, communication speed) of the parallel machine, but it does not capture the effects of blocking and synchronization.

A final strength of the E/T model is its closeness to the experiment. Monitoring tools typically provide precisely the data required by the model. The measurements reported in the previous section capture the detailed behavior of all threads of control of a parallel computation in a PDE solver. A preliminary analysis of the measurements shows the effects of the additional work for communication and control and of blocking. For a problem of a given size, the fraction of the time a PE is active, working on W_{cons} to its total running time, including work for communication and

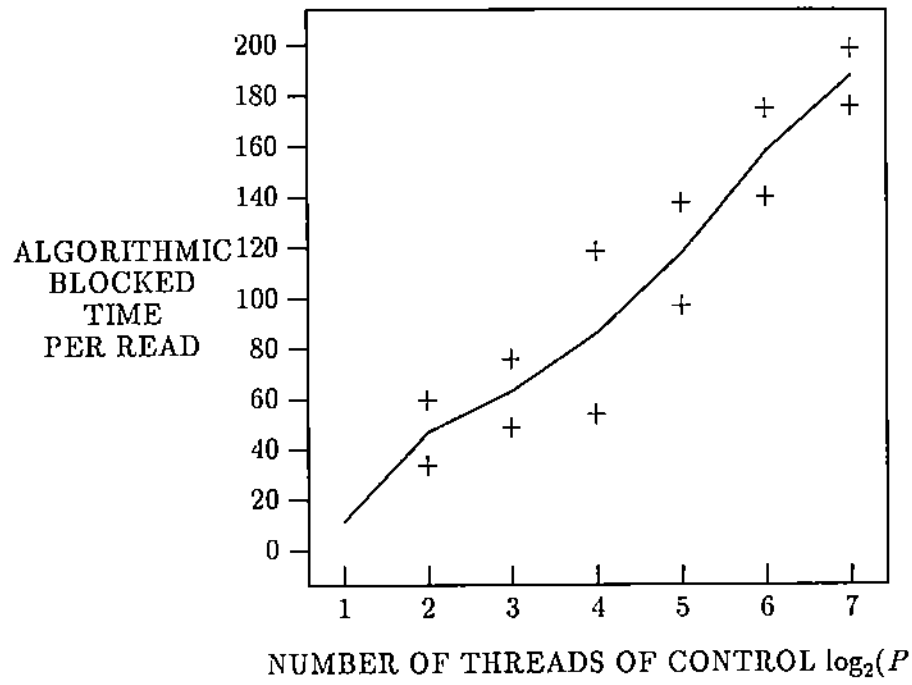

FIGURE 12: The expected *algorithmic blocking time* during a read operation (solid line) and a 95 percent confidence interval for it. The algorithmic blocking is defined as the interval from the instance a *read* is issued until the corresponding *write* is initiated.

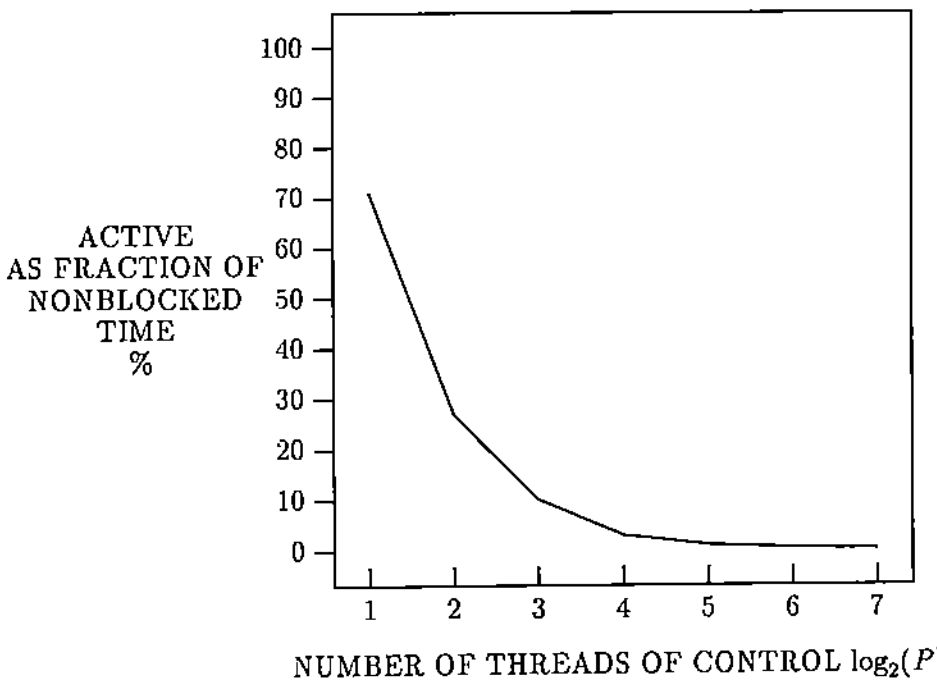

FIGURE 13: The expected *active time* fraction of the non-blocked time per thread.

control, decreases when the number of PEs increases under these conditions. The actual shape of both functions seems closely related to $g(P)$. Further investigations of other parallel computations and comparisons of results obtained from running the same computation on distributed memory and shared memory systems, are necessary to gain insight into the relationship between the characteristic function $g(P)$ and other measures of performance.

Acknowledgements

The authors express their thanks to Mo Mu for his helpful comments.

5 Literature

[1] A. Aggarwal, A.K. Chandra and M. Snir, "Communication complexity of PRAMs", Research Report Rc 14998, IBM Research, February 1989.

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

[3] 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.

[4] G. Fox et al., "Solving problems on concurrent processors", Prentice Hall, 1988.

[5] J.L. Gustafson, "Reevaluating Amdahl's law", CACM 31, 5, pp. 532-533, May 1988.

[6] E.N. Houstis and J.R. Rice, "Parallel ELLPACK: An expert system for parallel processing of partial differential equations", Math. Comp. Simulation, vol. 31, pp. 497-507, 1989.

[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. E. Lumpp, T.L. Casavant and H.J. Siegel "Models for monitoring and debugging tools for parallel and distributed software", Journal of Parallel and Distributed Computing, June 1990, (to appear).

[9] 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.

[10] Mo Mu and J.R. Rice, "The structure of parallel sparse matrix algorithms for solving partial differential equations on hypercubes", CSD-TR 976, Computer Sciences Department, Purdue University, May, 1990.

[11] 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.

[12] J.R. Rice and D.C. Marinescu, "Analysis of a two level asynchronous algorithm for PDEs", in *Aspects of Computations on Asynchronous Parallel Processors*, (M. Wright ed), North Holland, pp. 23-33, 1989.

[13] J.R. Rice and D.C. Marinescu, "Multi-Level asynchronous PDEs", in *Iterative Methods*, (D.R. Kincaid and L.J. Hayes eds), Academic Press, pp 193-212, 1990.

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