Speedup Limits for Tightly-Coupled Parallel Computations*

Reinhard Schwarz

University of Kaiserslautern, Department of Computer Science,
PO Box 3049, D–67653 Kaiserslautern, Germany

Abstract. In order to reduce the elapsed time of a computation, a popular approach is to decompose the program into a collection of largely independent subtasks which are executed in parallel. Unfortunately, it is often observed that tightly-coupled parallel programs run considerably slower than initially expected. In this paper, a framework for the analysis of parallel programs and their potential speedup is presented. Two parameters which strongly affect the scalability of parallelism are identified, namely the grain of synchronization, and the degree to which the target hardware is available. It is shown that for certain classes of applications speedup is inherently poor, even if the program runs under the idealized conditions of perfect load balance, unbounded communication bandwidth and negligible communication and parallelization overhead. Upper bounds are derived for the speedup that can be obtained in three different types of computations. An example illustrates the main findings.

1 Introduction

Today, parallel and distributed computing is almost ubiquitous. Apart from fault tolerance considerations, one of the main goals of parallel programming is to improve performance. Substantial speedup may be achieved by partitioning a problem into several subproblems, and by solving all subproblems concurrently on parallel and potentially distributed hardware.

In practice, however, it is sometimes difficult to realize performance gains. Certain classes of problems do not lend themselves for efficient parallel execution. Problems which are notoriously "reluctant" to parallelization are those which require frequent interaction between the concurrent threads of control constituting the parallel problem solving algorithm. Such tight coupling tends to be particularly disturbing in a distributed environment, simply because remote interaction is subject to relatively high communication latency and (typically) comparatively low communication bandwidth. Other potential sources of performance losses are, e.g., poor problem partitioning leading to undue load imbalances, additional costs for managing a complex parallel runtime environment, or overhead

* To appear in: K.P. Birman, F. Mattern, and A. Schiper (eds.), *Theory and Practice in Distributed Systems*, LNCS 938, Springer-Verlag, 1995.

caused by the parallelization of a problem solving strategy which is not inherently concurrent. Nevertheless, it is generally assumed that substantial speedup should be feasible, provided that the runtime environment operates efficiently, communication links are fast, and a suitable parallel algorithm leading to even load distribution exists.

Ideally, a programmer could hope for a speedup which is linear in the number of physical processors being used. Note that we do not consider phenomenons leading to so-called "superlinear speedup", caused mainly by an increase in main memory resources or by a reduction of the problem size due to parallelization. Such effects are known to occur, e.g., in branch & bound applications [8]. It is very tempting to assume that linear speedup will inevitably result if the parallelization and communication overhead approaches zero.

Unfortunately, it often turns out that parallel programs run surprisingly slow, although there is no clear performance bottleneck — neither in the application program, nor in the runtime environment. While we were analyzing a runtime package for parallel and distributed programming, we actually found several benchmark applications which seemed to have properties favorable for parallel execution — no additional parallelization overhead, reasonably low communication and synchronization demands, divisibility in perfectly even partitions — but which still did not yield scalable performance on parallel hardware. These observations aroused our curiosity, and led to an analytical study of factors relevant for the performance of parallel programs.

In this paper, we analyze how sporadic processor preemption affects the overall performance of a parallel system. Section 2 presents an abstract model of a tightly-coupled parallel computation which serves as a basis to study some fundamental effects that lead to speedup losses. In Sect. 3, three different execution profiles representing three different classes of parallel programs are analyzed, and speedup limits are derived in each case. The relevance of our study is discussed in Sect. 4, and a case study based on empirical results is presented which illustrates some implications of our analysis. Related work is briefly surveyed in Sect. 5, and Sect. 6 summarizes the main findings.

2 Basic Model of a Tightly-Coupled Parallel Computation

Our study is based on the simple model depicted in Fig. 1. Let the parallel system consist of n processors $P_1, \ldots, P_n$. For the subsequent analysis, we (optimistically) assume that communication between processors is feasible without any latency, and without bandwidth limitations — i.e., essentially at zero cost. Under these assumptions, communication events different from those which are necessary to achieve synchronization may be completely excluded from the analytical model.

We further assume that a parallel program is divided into n subtasks, each running on a separate processor. In order to reflect a close interaction between all subtasks, it is assumed that the computation consists of a large number of successive rounds and that at the end of each round all subtasks perform a global

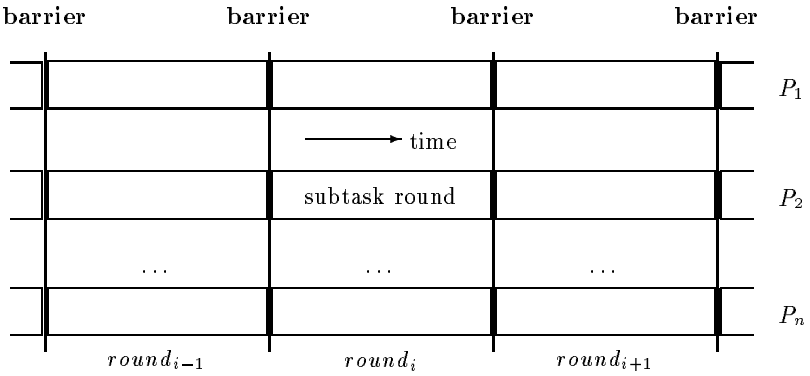

Fig. 1. Idealized model of a parallel computation

barrier synchronization. Such a pattern is frequently found in parallel algorithms, e.g. in iterative numerical computations. An example is presented in Sect. 4.

We also assume that all subtasks require the same amount of processing time in each round, yielding perfect load distribution over all processors. Global barrier synchronization, like communication, is assumed to cause no overhead and no delay. Therefore, synchronization events can be depicted as thin vertical bars in the space-time diagram of Fig. 1.

Let T denote the CPU time required to compute a single subtask round on one processor, and let the overall computation require r rounds on n processors. That is, the computation on n processors requires time $T(n) = rT$. In the absence of any overhead, the same computation could be executed on one processor computing all n subtasks, which would ideally require time nT per round. A single processor would thus require an elapsed time $T(1) = rnT$.

The aim of our analysis is to obtain upper bounds for the achievable speedup of a parallel computation. Let $T(n)$ denote the elapsed time required to finish the computation on n processors. We then define speedup in the usual way as

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

In the ideal scenario above, the speedup is — as expected — linear, i.e. $S(n) = n$. Under realistic conditions, however, we can hardly expect that a computation runs as smoothly as shown in Fig. 1. Instead, we typically observe that different processors require slightly different times to complete their round, even if all of them perform identical computations. Speed differences usually have their origin in the underlying operating system or in the runtime environment. For example, each node may run a timesharing system (a typical situation in a workstation cluster), and may therefore attribute only a small fraction of the available time slices to a specific subtask. Also, the operating system may spend some time on "housekeeping" tasks such as, e.g., dynamic heap defragmentation, swapping, or garbage collection. As a result of these local speed variations, the processors reach

the global barrier at different times. More specifically, all processes have to wait at the barrier for the slowest participant of the computation. The enforced idle times caused by the need to wait for global synchronization lead to performance losses. It is important to note that these delays are not under the control of the application program; they will occur even under the (optimistic) assumption of perfect load distribution.

In order to capture the notion of speed differences in our analytical model, we introduce so-called *time-out intervals*, periods where a processor is effectively not available for the application, for a reason hidden from the programmer. Furthermore, we partition the continuous execution into small, discrete *time units*. According to the assumptions stated above, each round of the application program requires a certain constant amount of time units where the processor is available to carry out the computation. Sporadic processor time-outs appear as time units where the processor is not available for the application subtask. This is reflected in our model by inserting a certain number of additional time-out units in each round. Time-outs on one processor lead to the occurrence of idle time on others at the synchronization barrier. The discrete model that results is depicted in Fig. 2.

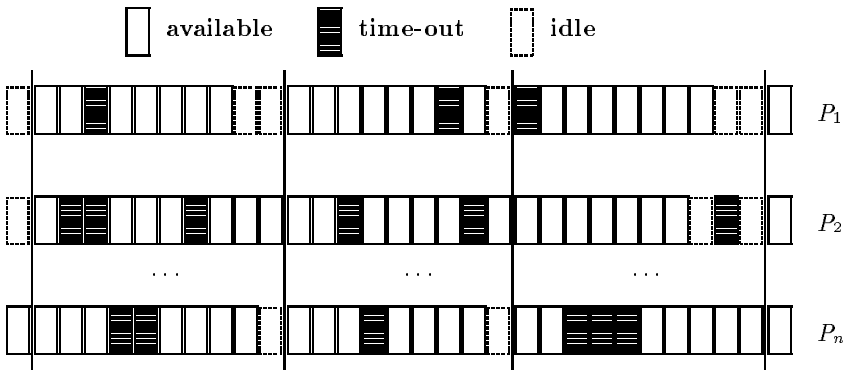

Fig. 2. Discrete model of a computation suffering from reduced availability

3 The Potential for Speedup in Parallel Computations

Based on the abstract model above, the impact of sporadic time-outs leading to speed variations can be analyzed. In this section, we present three different classes of computations, and derive limits for the speedup that is obtainable in each case. The models differ in the ratio between the average duration of a time-out period, denoted t, and the duration of a computation round, denoted T. Model I assumes that t is substantially smaller than T, model II assumes that t is significantly larger than T, and model III considers the case where $t \approx T$.

3.1 Model I: Short Time-out Periods, Long Rounds

If $t \ll T$, then we may choose t as the basic time unit of the discrete model. Without loss of generality, we assume that $t = 1$. At each time unit, the processor may be available — spending its time computing a unit of work for the subtask — or it may not be available, thus costing the subtask an additional time-out time unit without actual progress. As t is considered small in comparison with T, we ignore its variation. Instead, we assume that each time unit (independently of any other) is available for the application with probability a, or that it is a time-out time unit with probability $1 - a$. This implies that time-outs are Bernoulli-distributed with probability $1 - a$.

Each round requires T time units where the processor is available, plus additional k time units where the processor happens to be not available. From a local perspective, a round ends with a final time unit where the processor *is* available (several idle or timeout units may follow until the synchronization barrier occurs, but this must be caused by a different processor whose local subtask round takes longer to complete). Each of the remaining $T - 1 + k$ time units may be either available or not, as shown in Fig. 2. Based on a Bernoulli distribution, the probability that a specific permutation of available and time-out units occurs for a round of length $T + k$ is $a^T(1 - a)^k$, and therefore the probability for the occurrence of any such pattern of length $T + k$ is

$$p_T(a, k) = \binom{T - 1 + k}{k} a^T (1 - a)^k \quad . \tag{2}$$

Consequently, the probability that (locally) a round requires *at most* $T + u$ time units is

$$\hat{p}_T(a, u) = \sum_{k=0}^{u} p_T(a, k) \quad . \tag{3}$$

If n processors with availability a compute their rounds independently, then the probability that they *all* require at most $T + u$ time units is simply the product of the probabilities for each processor, i.e., $\hat{p}_T(a, u)^n$. Thus, the probability $P_T(n, a, u)$ that *exactly* $T + u$ time units are required until the last processor has finished its round — i.e., that all processors require at most $T + u$ time units, but more than just $T + u - 1$ — is given by

$$\begin{aligned} P_T(n, a, 0) &= \hat{p}_T(a, 0)^n \\ P_T(n, a, u) &= \hat{p}_T(a, u)^n - \hat{p}_T(a, u - 1)^n \quad . \end{aligned} \tag{4}$$

The mean elapsed time $T_{\text{act}}(n, a)$ of a round having a net requirement of T time units and being executed on n processors with average availability a is then

$$T_{\text{act}}(n, a) = \sum_{u=0}^{\infty} (T + u) P_T(n, a, u) \quad . \tag{5}$$

According to (1), the actual speedup $S_{\mathrm{I}}(n)$ achieved by an application of r rounds running on n processors is obtained by substituting $T_{\mathrm{act}}(n,a)$ for T, yielding

$$\boxed{\text{Speedup} \quad S_{\mathrm{I}}(n) = \frac{rnT_{\mathrm{act}}(1,a)}{rT_{\mathrm{act}}(n,a)} = n\frac{T_{\mathrm{act}}(1,a)}{T_{\mathrm{act}}(n,a)} \ .} \tag{6}$$

Equation (6) shows that the ideal linear speedup n is reduced by the factor $T_{\mathrm{act}}(1,a)/T_{\mathrm{act}}(n,a)$. The actual speedup that is achievable is depicted in Fig. 3 for some characteristic parameter settings. According to the diagram, this class of computations has a high potential for speedup. Figure 3 shows that if T is sufficiently large, then even a substantial decrease in availability will not lead to major performance degradations. Therefore, a parallelization of such applications should yield good performance.

So far, we only considered the case where all processors have the same availability a. If we assume that there is one processor P_1 whose availability a_1 is substantially smaller than $a_2 = \ldots = a_n$, then (4) has to be adapted. Intuitively, it should be clear that in this case $P_T(n,a,u)$ is dominated by the slowest processor P_1. For brevity, we do not present the corresponding equations that can easily be obtained. Instead, we just state without a formal proof that the following "rule of thumb" characterizes model I: *If a computation of type I runs on a system of n processors $P_1,\ldots,P_n$ with availabilities $a_1,\ldots,a_n$, then the observed availability a_{act} of the overall system is roughly given by $a_{\mathrm{act}} \approx min\{a_1,\ldots,a_n\}$.*

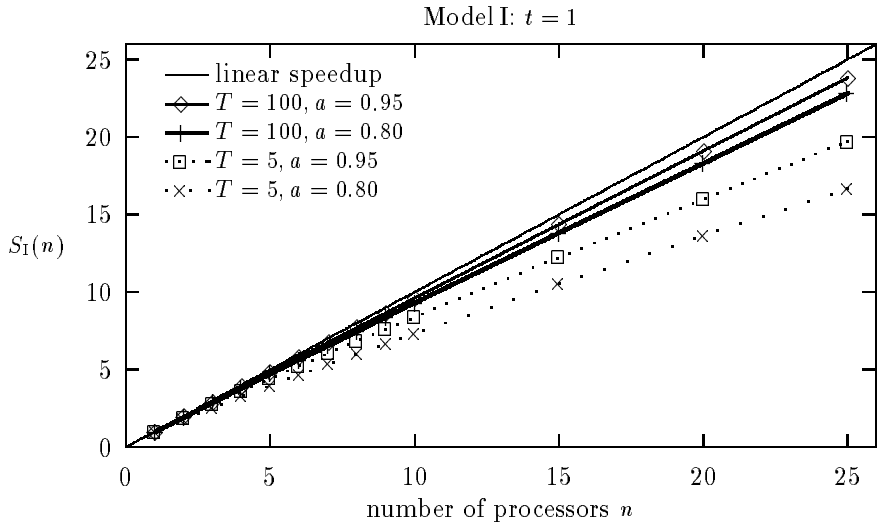

Fig. 3. Speedup of computations where $T \gg t$

3.2 Model II: Long Time-out Periods, Short Rounds

In this model, we assume that $T \ll t$. Therefore, we may safely choose T as the basic time unit, assuming without loss of generality that $T = 1$. Figure 4 shows the corresponding discrete model. Each round requires one time unit of

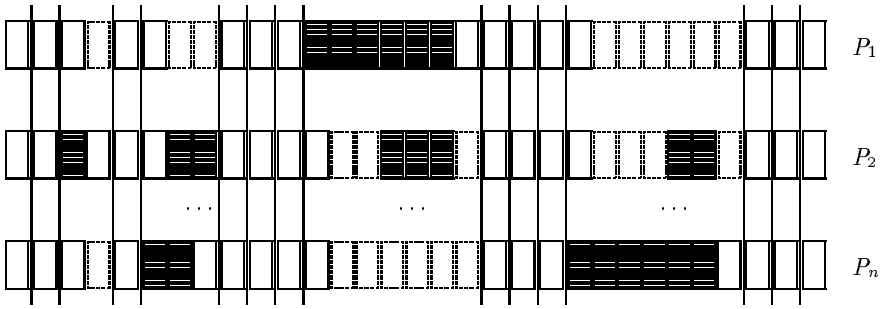

Fig. 4. Discrete model of computation where $T \ll t$

computation, and ideally barrier synchronization should occur once per time unit. However, the frequency of barriers actually observed is reduced by sporadic time-out periods, similar to model I.

Unfortunately, the assumption of time-outs obeying a simple Bernoulli distribution is inappropriate in the case of model II. Note that time-outs — if they are long — correspond to *runs* of time-out time units in our discrete model. Furthermore, it is known from experience that the duration of long time-outs is subject to considerable variation. As a practical example, we studied the time-sharing behavior of a highly loaded workstation running SunOS, and we found that time-out periods due to timeslicing ranged from less than 7 ms to more than 250 ms, with a mean of about 35 ms. Therefore, a more sophisticated distribution scheme for time-out units is required.

One possibility to reflect the assumptions of our model — time-out bursts of considerably varying length — is to base the analysis on a so-called Markov-modulated Bernoulli process (MMBP). In our case, the MMBP consists of two different states and associated transition probabilities which determine the permanence of each state. The two states reflect periods of "availability" and "time-out" of a single processor. The parameters α and β denote the probability that an interval of availability (or time-out, respectively) ends with the next discrete time unit of the computation. Figure 5 shows the MMBP that was used for the subsequent analysis.

Our MMBP model has some interesting properties. In particular, it is easy to show that the mean duration which the MMBP continuously spends in state AVAILABLE (TIME-OUT) is $1/\alpha$ ($1/\beta$, respectively). And the overall probability that the MMBP is in state AVAILABLE is $\beta/(\alpha + \beta)$. Thus, we can adjust the

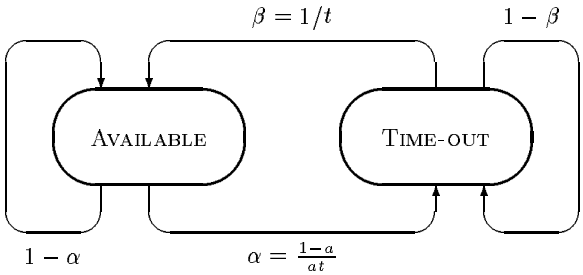

Fig. 5. A Markov-modulated Bernoulli process for model II

MMBP model so that it simulates arbitrary rates of availability $a \in (0, 1)$, and arbitrary mean time-outs $t > 1$, by defining $\beta = 1/t$ and $\alpha = \beta(1 - a)/a$. But besides being easily adaptable to our needs, the model may serve as a reasonable approximation of reality. For example, if we interpret AVAILABLE as "sunshine", TIME-OUT as "rain", and the time unit T as one day, then for $\alpha = 1/4$ and $\beta = 1/3$ the process depicted in Fig. 5 approximates the weather conditions in Tel Aviv between December and February [5].

Assuming that time-outs are distributed according to a Markov-modulated Bernoulli distribution and that each round requires one time unit of computation, our aim is to determine the average frequency f of barrier synchronizations. Note that the reciprocal $1/f$ — the mean time between successive barriers — denotes the expected time T_{act} which elapses per round. In order to obtain f, we convert our model of computation into a Markov chain model, as shown in Fig. 6.

To this end, each discrete step of our original model is characterized by a state with two attributes, n_{t} and n_{w}. Attribute n_{t} denotes the number of processors which are currently in state TIME-OUT according to the MMBP model. State attribute n_{w} denotes the number of processors which have not yet finished their subtask round, i.e., which have been continuously in state TIME-OUT since the beginning of the current round. As demonstrated in Fig. 6, each instant of a computation can be uniquely represented by some state $s = (n_{\text{t}}, n_{\text{w}})$. Consequently, each computation corresponds to a state sequence $s_1, s_2, s_3, \ldots$, where $n_{\text{t}} \in [0,\ n]$ and $n_{\text{w}} \in [0, n_{\text{t}}]$ according to the definition.

Figure 6 shows the Markov chain corresponding to a system comprising 2 processors. In this diagram, an arrow between two states s_i and s_j indicates that a state transition from s_i to s_j may occur with positive probability (transitions from s_i to s_i have been omitted for simplicity of exposition). If two states are not connected by a directed edge, then a state transition from one to the other is impossible according to the definition of the states' attributes. Note that state $s = (2, 0)$ is not feasible for $n = 2$. Similar restrictions apply for $n > 2$ which reduce the number of feasible states in the Markov model.

Knowing n_{t} and n_{w}, it is a simple but tedious task to determine the transition probabilities for each pair (s_i, s_j) of states as a function of α and β — or a and t, respectively. Several transitions are known to be impossible, yielding transition

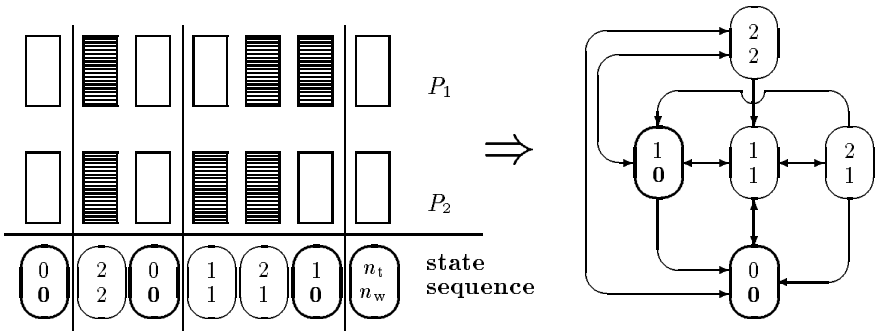

Fig. 6. Converting model II into a Markov chain

probability 0. For example, n_w can only decrease on each state transition, except for the case where $n_w = 0$: If no processor is waiting, then the round is finished, global synchronization can be established, and the next state transition must lead to a feasible initial state of the next round. For this initial state, $n_t = n_w$ must hold by definition. For the probability $p = p(n, n_t, n_w, n'_t, n'_w)$ of a feasible state transition $(n_t, n_w) \rightarrow (n'_t, n'_w)$ in an n processor Markov chain model, the following equations can be obtained with elementary techniques of probability theory. We omit the details of the derivation for brevity. Equation (7) denotes the transition probability p_0 for a transition from a feasible *final* state of a round r_i to the *first* state of the subsequent round r_{i+1}. As noted above it follows in this case that $n_w = 0$ and $n'_t = n'_w$, yielding

$$p_0 = \sum_{k=\max\{0,n_t-n'_t\}}^{\min\{n_t,n-n'_t\}} \binom{n_t}{k} \beta^k (1-\beta)^{n_t-k} \binom{n-n_t}{n'_t-n_t+k} \alpha^{n'_t-n_t+k} (1-\alpha)^{n-n'_t-k} \quad . \tag{7}$$

The general probability p for a feasible state transition *internal* to a round (i.e., a transition where $n_w \neq 0$ and $n'_w \leqq n_w$) is given by

$$p = \binom{n_w}{n'_w} \beta^{n_w-n'_w} \bar{\beta}^{n'_w} \sum_k \binom{n_t-n_w}{k} \beta^k \bar{\beta}^{n_t-n_w-k} \binom{n-n_t}{y(k)} \alpha^{y(k)} \bar{\alpha}^{n-n_t-y(k)}$$

where

$$\bar{\alpha} = 1-\alpha; \bar{\beta} = 1-\beta$$

$$\max\{0, n_t - n'_t + n'_w - n_w\} \leqq k \leqq \min\{n_t - n_w, n - n'_t - n_w + n'_w\}$$

$$y(k) = n'_t - n_t + n_w - n'_w + k \quad . \tag{8}$$

Let $\{s_1, \ldots, s_l\}$ denote the set of feasible states. The Markov chain equivalent of model II is completely determined by the matrix $\mathbf{M} = (m_{ij}), 1 \leqq i, j \leqq l$, with m_{ij} denoting the probability of transition $s_i \rightarrow s_j$, as given by (7) and (8). Each computation conforming to model II may be interpreted as a random walk through the corresponding Markov chain (and vice versa), where the state that

is entered next is chosen according to the probabilities defined by $\mathbf{M}$. From the theory of Markov chains it is known that (for our specific model) the probability q_i that state s_i is entered during a random walk of infinite length is well-defined — regardless of the starting point of the walk; in fact, $q = (q_1, \ldots, q_l)$ is the so-called *steady-state probability distribution* of the Markov chain. It can easily be shown that q is uniquely defined by the following equation (for a proof, the interested reader is referred to any standard textbook on Markov chains):

$$q\mathbf{M} = q \quad . \tag{9}$$

From q, the solution of (9), we can immediately derive the frequency f of global barrier synchronization. Recall that synchronization takes place as soon as the final processor has completed its subtask round. In the Markov chain model this corresponds to a state $s = (n_\mathrm{t}, n_\mathrm{w})$ where $n_\mathrm{w} = 0$. Therefore, it follows that

$$f = \sum_{k \in \{i \mid s_i = (n_\mathrm{t}, 0)\}} q_k \quad . \tag{10}$$

Let $f(n, a, t)$ denote the frequency obtained for a computation on n processors, each being available with probability a, and suffering from time-outs of mean duration t. Recall that the mean elapsed time T_act per round is $1/f$. The speedup achievable in model II is then given by

$$\boxed{\text{Speedup} \quad S_\mathrm{II}(n) = \frac{rnT_\mathrm{act}(1)}{rT_\mathrm{act}(n)} = n\frac{f(n,a,t)}{f(1,a,t)} \quad .} \tag{11}$$

Similar to (6), the potential for linear speedup is reduced by a characteristic factor, $f(n, a, t)/f(1, a, t)$. To compute the corresponding frequencies, (9) has to be solved. This can be done, e.g., by choosing an arbitrary initial probability distribution $q^{(0)}$ and by applying the iteration $q^{(i+1)} = q^{(i)}\mathbf{M}$ which quickly converges. Note, however, that the Markov chain model tends to become rather large even for moderate values of n. The number of components of q is of the order $O(n^2)$, and the size of $\mathbf{M}$ is of the order $O(n^4)$; for $n = 99$, matrix $\mathbf{M}$ comprises about 25,5 million elements.

The speedup diagram corresponding to (11) is depicted in Fig. 7. In contrast to model I, computations belonging to this class are extremely vulnerable by reduced availability of the processors. Even for a small number of processors a moderate decrease in the availability a dramatically reduces the performance of the computation. The negative effects of time-outs are illustrated in Fig. 4. In the example given there, the avarage availability a is about 0.76, but 29 time units on three processors are required to compute only 14 rounds.

Interestingly, the precise value of the average time-out duration t is not a decisive factor in model II, which makes it robust against parameter changes. Provided that a is known, a rough estimate of t suffices to make accurate predictions.

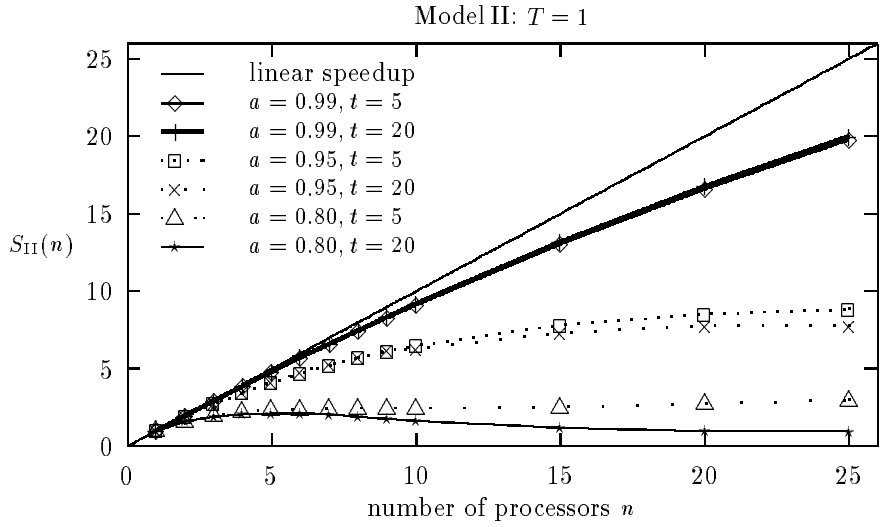

Fig. 7. Speedup of computations where $T \ll t$

3.3 Model III: Time-out Duration Comparable to Round Duration

To bridge the gap between model I and model II above, we next assume that $t \approx T$. Unfortunately, none of the previous models can be extended to cover this situation. If we assume that $t = T = 1$, then model I lacks accuracy because it neglects the variation of t and the fact that a round may be partly finished when a time-out interval occurs. Alternatively, we may try to analytically solve model II, assuming that $t = T = k$ time units for some $k > 0$. Although feasible in principle, this is again not a viable approach, simply because the required number of Markov states would by far exceed our ability to solve (9) numerically.

As we were not aware of a manageable analytical model which satisfactorily solves the case where $t \approx T$, we studied model III by simulating some representative cases. To this end, we chose different values for T from the interval $[20, 100]$, and for t from $[T/2, 2T]$ to cover a wide range of possible parameter settings. A MMBP was used to generate time-out intervals of mean duration t, while T was assumed to be constant. Figure 8 shows the simulation results for some of these experiments. A comparison with Fig. 7 and Fig. 3 reveals that model III is closely related to model II, substantially closer than to model I. This is particularly true for $a \approx 1$. In this case, the value of t — like in model II — is again not decisive, and therefore reliable predictions are feasible even if accurate parameter values are not available. Roughly speaking, for $a \approx 1$ the analytical results that were formally derived for model II under the assumption that $t \gg T$ extend to the case where $T \approx t$. This means that computations of type III are almost as sensitive to reduced availability as computations of type II.

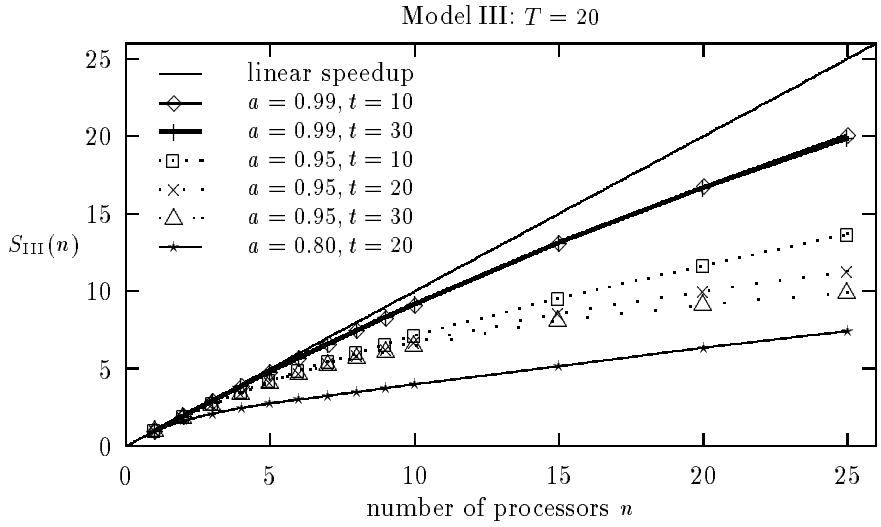

Fig. 8. Speedup of computations where $T \approx t$

3.4 Efficiency of Models I–III

In the previous subsections, we saw that for each model the desired optimal speedup is reduced by a characteristic factor, $S(n)/n$, the so-called *efficiency* of the computation [4]. Efficiency can be interpreted as the fraction of perfect speedup that can be realized in practice. Figure 9 depicts $S(n)/n$ for our three different models, for an availability $a = 0.95$. The diagram emphasizes the strong difference between model I on the one hand, and models II and III on the other. It also shows the close resemblance between the latter two models.

4 Discussion

4.1 Validity and Scope of Results

In order to validate our analysis, we compared our analytical predictions with measurements obtained by simulation. We found that the results matched perfectly which confirms the correctness of (6) and (11). Of course, both simulation and analysis were based on a specific set of assumptions, and the parameters that we chose lack concise empirical justification. To explore the scope of our models, we therefore made further simulation experiments, considerably varying the parameters t and T in each case. We also tested probability distributions which were quite different from a simple Bernoulli or MMBP model. Our models

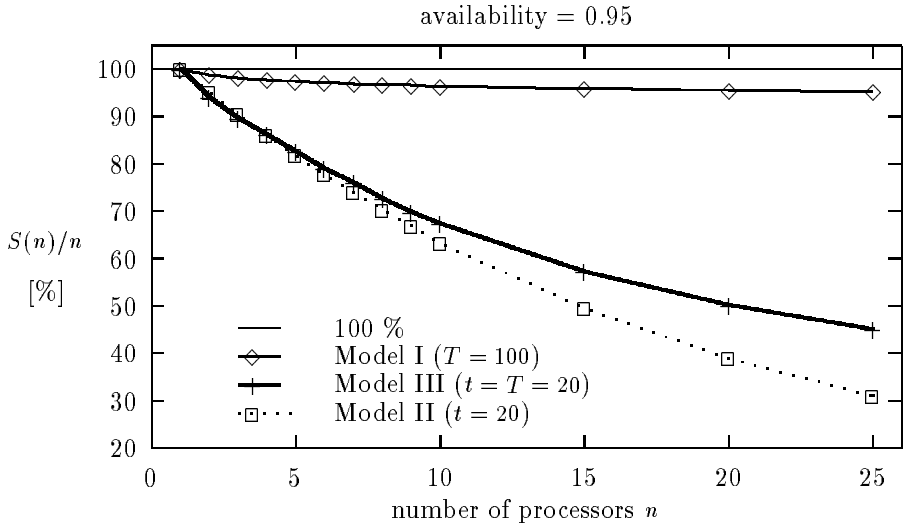

Fig. 9. Efficiency achievable in models I–III

turned out to be rather robust. We found that their general tendency was preserved over a wide range of different scenarios, and the speedup results obtained by simulation were in good accordance with the values predicted by (6) and (11).

Another concern is whether the specific pattern of interaction on which our analysis has been based — a sequence of rounds, each followed by a global barrier synchronization — is frequently found in practice. Admittedly, such a tight coupling is a rather pessimistic assumption, and one might argue that most parallel programs try to restrict synchronization events to only a small subset of all processors, thus escaping from the limitations imposed by our analytical results. To a certain degree, our model compensates for its pessimism by being far too optimistic in other respects. For example, we assumed that parallelization, communication and synchronization causes no additional overhead, and that perfect load balance is preserved throughout the computation. It should be noted that (to a certain extent) load imbalances and parallelization overhead could roughly be interpreted as additional time-outs according to our model: The processor "wastes" additional time while computing the next round of the application program, again leading to the undesirable effect that different processors reach the barrier at different and unexpected times.

Interestingly, the behavior of some parallel programs closely resembles our model of global barrier synchronization, even though processors synchronize only bilaterally. As an example, consider a computation organized as a pipeline of n stages, where each stage is executed by a separate processor. Each stage receives its input from its predecessor, and sends its output to its successor. If the in-

terconnection between successive stages is not able to buffer sufficient data in order to compensate for speed variations, then each processor has to agree with its neighbors on the beginning of the next round of input and output. In this case the whole pipeline essentially runs in lockstep, which is comparable to our assumption of global barrier synchronization. A slightly different, but closely related scenario — a parallel version of successive over-relaxation (SOR) — is discussed below in greater detail.

But even if we assume that large-scale barrier synchronization is not found frequently in parallel programming, one should recall that according to model II substantial performance penalties may already result in small, tightly-coupled groups of less that 10 individuals. Such groups are typical for a large number of applications and protocols such as, e.g., atomic group communication. This observation — together with the relative insensitivity of the analytical results towards parameter changes — strongly suggests that our *specific* model may serve as a representative for a wide range of tightly-coupled parallel programs.

4.2 Case Study: SOR

As a practical application of our theoretical results, we studied the Red/Black Successive Over-Relaxation (SOR) algorithm. SOR is an iterative method for solving discretized Laplace equations on a rectangular grid. During each iteration the algorithm considers all non-boundary points of the grid. For each inner point, SOR computes the average value of its four neighbors in the North, South, East, and West, and updates the point using this value. Conceptually, the grid is colored red and black like the squares of a checkerboard. In the red phase, only the red points are updated, and in the subsequent black phase, only black points are considered.

For a parallel execution of the SOR algorithm, the grid is partitioned into regions of consecutive columms, and each region is assigned to a different processor. At the beginning and at the end of a phase, each processor has to exchange the grid values of its leftmost and rightmost column with the corresponding neighbor. To enforce consistency, synchronization has to ensure that all processors compute the same round of the iteration. This may be achieved either by a global barrier synchronization, or by pairwise synchronization between adjacent grid regions. In any case, all processors are forced to compute the same round and the same color phase concurrently. Therefore, SOR is a good example for a parallel program that fits in our formal model of computation.

SOR is a popular benchmark for parallel programming environments. A number of speedup results have been published for different implementations on various systems. Among these, several experiments were performed on workstation clusters connected by a 10 Mb/s Ethernet, e.g., a SOR implementation in the Orca environment running on MC68030 processors [3] on top of Amoeba, one in the Amber environment running on DEC SRC Firefly shared-memory multiprocessor workstations on top of Topaz [6], and also a realization on top of the Clouds operating system running on Sun3 workstations [1].

Interestingly, all these SOR implementations claim to have good scalability properties, and in fact they achieved almost linear speedup. However, to obtain such favorable results it is essential to ensure high availability of the processors on which the program is executed. Under the conditions of an open workstation cluster running a general-purpose operating system, the only way to exclude noticeable time-outs is to run the application on an otherwise idle network.

SOR might be able to tolerate reduced availability provided that its behavior corresponds to model I of our analysis. In fact, we inspected the (scarce) information given in the literature, and we actually found that this was probably the case for the benchmarks mentioned above. For instance, in all these systems a single round of a 1000×500 SOR problem would have required at the order of ten or more seconds of CPU time if executed sequentially on a single processor. This is substantially longer than the time-out periods that typically occur in workstation clusters, even if we take into account that a parallelization on n processors would reduce the round duration by a factor of $1/n$. Consequently, for reasonably small n these implementations can safely be assumed to correspond to our first model which is not critical with respect to availability.

However, the situation radically changes with improved CPU speed. We implemented SOR in our PANDA environment, running on top of SunOS. The program was executed on lightly loaded SunSparc10 Model 20 workstations connected by a 10 Mb/s Ethernet [2]. In this environment the time required to compute one iteration of the SOR algorithm on one processor was only about one second for a grid of 1000×500 points. That is, a single phase required less than 500 ms CPU time in the sequential case. An execution on 10 processors would have reduced T to less than 50 ms. Note that this is close to the expected duration of time-outs caused, e.g., by timeslicing. As a consequence, the program closely resembled model III and — with increasing load in the network — even model II of our analysis.

Compared to the previously mentioned realizations running on slower hardware and belonging to class I, the *absolute* performance of our program was considerably better, but we achieved only poor scalability due to a CPU availability of only about 95 %. Even with an implementation based on optimized message passing, the best result that we achieved for a 1000×500 problem was a speedup of 4.0 on five processors. This is in accordance with the predictions of models II and III.

The example shows that speedup and scalability of a program strongly depend on the type of computation being executed. Therefore, in order to analyze the properties of a runtime environment it is important to select benchmarks which cover the different models. In model I, the performance of a program is probably dominated by the speed of the sequential parts (i.e., the local subtask rounds) of the computation; a benchmark taken from this class mainly tests the local support for efficient execution, and the system's ability to achieve even load distribution. In contrast to that, benchmarks of classes II–III are very sensitive to time-outs. Their performance is a good indicator for the efficiency of communication, global synchronization, and scheduling — relative to CPU speed.

5 Related Work

There are a number of analytical studies which indicate that — beyond current technological bottlenecks — there are several fundamental reasons why multithreaded, tightly-coupled programs often have poor performance.

Eager et al. [4] relate the efficiency of a parallel execution, $E(n) = S(n)/n$, to the inherent parallelism of the computation on the one hand, and to the available number of processors on the other. In contrast to our study, Eager et al. assume that time-outs are caused by a lack of conceptual and physical parallelism rather than by processor preemption. They derive the optimal number of processors required to maximize $E(n)/T(n)$, leading to both good efficiency and reasonably short completion time of the computation.

Multithreading has been advocated as a means to utilize intervals which are otherwise lost because of communication or synchronization delays. Although appealing in principle, the advantages of multithreading and its potential drawbacks have to be weighed up. In [9], Saavedra-Barrera et al. show how context switch costs, cache performance and communication latency affect the performance of a multithreaded program execution. In accordance with our results — but for different reasons — they argue that optimal performance is typically achieved with only a small number of parallel threads of control per processor.

Multithreading may have a negative impact on performance because each time slice that is given to a specific thread occurs as a time-out period to all the others, thus reducing the effective availability that is observed by each competitor. To overcome this difficulty, the system should try to ensure that a thread, once activated, is not forced to wait for related, but currently preempted activities. Ideally, such latencies should be minimized by scheduling closely cooperating groups of processes — so-called *gangs* — at the same time on different processors, and by preempting the whole gang as soon as the first gang member runs into a waiting condition. For short delays, busy waiting might be superior to preemption as it may help to avoid premature context switching of the whole gang. This is the basic approach discussed by Feitelson and Rudolph [7]. In their paper, the authors analytically determine the relative performance gains of gang scheduling, depending on the grain of synchronization. They reach the conclusion that to some extent, rigorous global management of scheduling activities may help to avoid (or at least to coordinate) the occurrence of time-outs caused by busy waiting, and to eliminate their impact on the parallel computation.

Related results were obtained by Zahorjan et al. In [10] the authors discuss the potential benefits of different scheduling disciplines in shared memory multiprocessor systems employing spin locks. Similar to our analysis, they show that variations in thread execution times may cause substantial spinning overhead if uncoordinated thread scheduling disciplines are used. They study the benefits of various scheduling strategies, and they determine the conditions required to obtain robust parallel execution characteristics.

6 Conclusions

Tightly-coupled parallel programs are notoriously difficult to implement so that reasonable speedup is achieved. The underlying reasons, however, are still not fully understood.

Traditionally, performance losses were mainly attributed to the lack of communication and synchronization speed, particularly in distributed environments. It was generally argued that tight coupling in the computation should be adequately reflected by tightly-coupled hardware architectures — i.e., multicomputers — to satisfactorily solve the problem. In this paper, we showed that such a view is probably too optimistic. We found that certain types of programs are extremely sensitive to sporadic time-outs. They badly fail to achieve linear speedup even if we assume perfect load balance and highly efficient hardware support for communication and synchronization.

Compared with conventional workstation clusters, it is generally assumed that multiprocessor systems offer a higher availability $a \approx 1$. For reasons of cost effectiveness, however, there is a trend to employ multithreading and time-sharing even for the CPUs of multiprocessor machines. While this may help to better utilize idle times caused by communication latencies, it also increases the probability of time-outs which reduces availability. To a certain degree, gang scheduling may help to overcome the difficulties, but its potential is limited by a bounded number of processors and by the general inability to identify the members of dynamically changing gangs. This implies that the ongoing technological progress will probably not lead to a corresponding increase in speedup for tightly-coupled computations. On the contrary: As processors become faster, the duration of the sequential phases of a parallel computation will further decrease, and the impact of processor availability will gain importance, as our SOR example illustrates.

A general guideline can be derived from our analysis. Runtime systems should be designed so that computations of classes II and III can be avoided. In order to ensure that $t \ll T$ holds, preemptive multithreading should — where necessary — be done at a fine granule, even if more frequent context switching leads to more overhead. Furthermore, for memory management and garbage collection on-the-fly strategies should be given preference over stop-and-collect approaches, again even at the expense of increased complexity.

Acknowledgements

The work presented in this paper was inspired by experiments carried out as part of the PANDA project. I would like to thank my colleagues for providing me with a distributed environment that works. Thanks in particular to Peter Buhler who implemented a version of SOR based on a distributed shared memory model. Our joint analysis and discussion of benchmark results helped me to gain better insight into the fundamental issues. Thanks also to Thomas Breitbach for his SOR realization based on message passing which was used for a comparative analysis.

References

1. Ahamad M., John R., Kohli P., and Neiger G.: Causal Memory Meets the Consistency and Performance Needs of Distributed Applications. Proc. 6th ACM SIGOPS European Workshop, Dagstuhl Castle, Wadern, Germany (Sept. 1994) 45–50
2. Assenmacher H., Breitbach T., Buhler P., Hübsch V., Peine H., and Schwarz R.: Parallel Programming in PANDA. Journal of Supercomputing, Special Issue on Trends in Parallel Operating Systems (to appear)
3. Bal H.E., Kaashoek F., and Tanenbaum A.S.: Orca: A Language for Parallel Programming of Distributed Systems. IEEE Transactions on Software Engineering, Vol. 18, No. 3 (March 1992) 190–205
4. Eager D.L., Zahorjan J., and Lazowska E.D.: Speedup Versus Efficiency in Parallel Systems. IEEE Transactions on Computers, Vol. 38, No. 3 (March 1989) 408–423
5. Engel, A.: Wahrscheinlichkeitsrechnung und Statistik, Band 2. Ernst Klett Verlag, Stuttgart 1976, 178–179
6. Feeley M.J., Bershad B.N., Chase J.S., and Levy H.M.: Dynamic Node Reconfiguration in a Parallel-Distributed Environment. Proc. 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM SIGPLAN Notices, Vol. 26, No. 7 (July 1991) 114–121
7. Feitelson D.G. and Rudolph L.: Gang Scheduling Performance Benefits for Fine-Grain Synchronization. Journal of Parallel and Distributed Computing, Vol. 16, No. 4 (Dec. 1992) 306–318
8. Rao V.N. and Kumar V.: On the Efficiency of Parallel Backtracking. IEEE Transactions on Parallel and Distributed Systems, Vol. 4, No. 4 (April 1993) 427–437
9. Saavedra-Barrera R.H., Culler D.E., and von Eicken T.: Analysis of Multithreaded Architectures for Parallel Computing. Proc. 2nd Annual ACM Symposium on Parallel Algorithms and Architectures, Island of Crete, Greece (Juli 1990) 169–178
10. Zahorjan J., Lazowska E.D., and Eager D.L.: The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Systems. IEEE Transactions on Parallel and Distributed Systems, Vol. 2, No. 2 (April 1991) 180–198