Gang Scheduling Performance Benefits for Fine-Grain Synchronization

Dror G. Feitelson* Larry Rudolph
Department of Computer Science
The Hebrew University of Jerusalem
91904 Jerusalem, Israel

Abstract

Multiprogrammed multiprocessors executing fine-grained parallel programs appear to require new scheduling policies. A promising new idea is *gang scheduling*, where a set of threads are scheduled to execute simultaneously on a set of processors. This has the intuitive appeal of supplying the threads with an environment that is very similar to a dedicated machine. It allows the threads to interact efficiently by using busy waiting, without the risk of waiting for a thread that currently is not running. Without gang scheduling, threads have to block in order to synchronize, thus suffering the overhead of a context switch. While this is tolerable in coarse grain computations, and might even lead to performance benefits if the threads are highly unbalanced, it causes severe performance degradation in the fine-grain case. We have developed a model to evaluate the performance of different combinations of synchronization mechanisms and scheduling policies, and validated it by an implementation on the Makbilan multiprocessor. The model leads to the conclusion that gang scheduling is required for efficient fine grain synchronization on multiprogrammed multiprocessors.

1 Introduction

Multiprocessors are often dedicated to running a single application at a time. The program is allowed full control over what happens on each processor, and in fact it might be required to include instructions that regulate the mapping and scheduling of parallel threads. Much experience relating to these issues has been accumulated over the years, and automatic parallelization and compilation techniques have been developed. These techniques allow dedicated processors to be used efficiently by a single application.

In recent years multiprogrammed general-purpose parallel systems have begun to emerge. In such systems, each thread is viewed as a "virtual processor". Threads belonging to the same application cooperate with each other, as before, while threads from different applications compete for system resources. The total number of threads is typically larger than the actual number of processors, and the full details about the system state are not available to any of the individual applications. Therefore the execution cannot be regulated by the programmer or the compiler. Rather, the system software takes care of mapping and scheduling issues. The question is, how should processing resources be divided among the competing jobs?

Run-time systems on parallel machines are typically straightforward modifications of uniprocessor systems. Sometimes each processor simply executes a distinct copy of the system, with various

*Current address: IBM T. J. Watson Research Center, P. O. Box 218, Yorktown Heights, NY 10598.

locks added to provide mutual exclusion where necessary. At best, the processors are considered as just another resource that is allocated upon request. Upon closer inspection, however, it seems that this approach is not entirely satisfactory. The dynamics of program execution on a multiprocessor are fundamentally different from the dynamics on a uniprocessor, due to the fact that multiple threads of control are active simultaneously. The system must acknowledge the interactions and interdependencies between the different threads, and this must also be reflected in the support provided by the system.

The interactions between threads of a parallel application are embodied in the *synchronization* requirements of the application. This includes both explicit synchronization, such as barrier synchronization and mutual exclusion, and implicit synchronization, such as the relation between a thread that produces a data item and the thread that consumes it. The next section examines the effect of the scheduling policy used by the run-time system on the efficiency of synchronization. In the case of *fine-grain* interactions it is shown that it is best for the threads to execute simultaneously on distinct processors, and coordinate their activities with busy waiting. This is achieved by a *gang scheduling* policy, i.e. the scheduling policy that coordinates context switching across a number of processors so as to schedule a "gang" of interacting threads simultaneously. Note the analogy between this and virtual memory management, where a working set of pages must be present simultaneously in order to prevent thrashing [21]. With *coarse-grain* interactions, on the other hand, coordinated scheduling is not necessary. If the variance in computation times between interactions is high, it is then best to block a thread that has to wait for synchronization. This allows the processor to be used by another thread, possibly from another application, while the first thread is waiting.

We have implemented gang scheduling on the Makbilan multiprocessor in order to validate these results. While Makbilan is a shared-memory machine, it should be noted that the results are also applicable to message passing architectures. The implementation is described in section 3, followed by the experiments that were used to validate the model. The model and measurements are then used to give a full characterization of situations in which busy waiting with gang scheduling is preferred, as opposed to situations where blocking is better. The implementation also showed that gang scheduling alone is not enough to support fine-grain interactions: special hardware support is needed to make the interactions fast enough. These and other results are discussed in the conclusions.

Related Work

Gang scheduling was introduced by Ousterhout in the context of the Medusa system on Cm* [21] (actually he suggested a less strict version called *coscheduling*). He explained the intuition behind using gang scheduling to allow efficient use of busy waiting for fine-grain synchronization, and developed three algorithms for its implementation. Similar ideas were suggested by Edler et. al. in the context of the the NYU Ultracomputer project [10]. However, no analysis of the performance implications was done.

Very little work has been done on gang scheduling since then. Błażewicz et. al. developed off-line algorithms for gang scheduling, essentially using dynamic programming [7]. Seager and Stichnoth have simulated gang scheduling on multiprocessor Crays, and conclude that it is a good scheduling discipline for multithreaded supercomputer applications [26]. Gupta et. al. have simulated gang scheduling (and other scheduling policies) on a cached bus-based machine [14]. They conclude that gang scheduling is one of the best approaches, because busy waiting can be used efficiently with it. However, due to the use of only four specific applications, they fail to characterize the exact conditions under which gang scheduling is beneficial, and the specific performance impact it may have. Some results that support gang scheduling were also presented by Lo and Gligor [17], Leutenegger and Vernon [16], and Zahorjan et. al. [30]. We have described a novel control structure, called "distributed hierarchical control", for the implementation of preemptive gang scheduling in large, interactive multiprocessor systems [11, 12]. Some existing systems support space-division sharing to execute applications side by side, which is similar to batch-style gang scheduling [6, 3].

Busy waiting and blocking have also been analyzed elsewhere. The two schemes have been compared by Zahorjan et. al. in two different papers, using queueing models. The first does not consider the benefits of gang scheduling [31], and is therefore biased against busy waiting. The other paper, through their choice of parameters, considers situations which essentially amount to coarse grain computations, and therefore it concludes, not surprisingly, that both schemes are rather similar [30]. Gupta et. al. show that the performance of busy waiting and blocking depends strongly on the scheduling mechanism [14], but they are nonconclusive as to which is ultimately better.

To our knowledge, this paper is the first to analyze the performance implications of gang scheduling, and the interplay between scheduling and synchronization. This allows us to identify the situations in which gang scheduling should be used, namely when the application is based on fine-grain synchronization. It is also the first to report experiments based on a real implementation of gang scheduling on a multiprocessor.

Gang scheduling may cause an effect reminiscent of fragmentation, if the gang sizes do not fit the number of available processors. We have previously shown that under reasonable conditions this may lead to a loss of up to 25% of the computing resources [13]. In this paper we show that for fine-grain computations gang scheduling can more than double the processing capability. Thus the final balance indicates that in certain cases gang scheduling has considerable performance benefits.

Terminology

Different authors tend to use the same terms with slightly different meanings. Therefore a note on our usage is in order. First, we use the term *thread* to denote the parallel light-weight activities that co-exist in and comprise the execution of a parallel program. This is synonymous with *task* in the context of our implementation on the Makbilan, described in section 3.

We use the term *gang scheduling* to denote a scheduling policy, implemented by the run-time system, in which a set of threads is scheduled simultaneously on a set of processors, using a one-to-one mapping. In other words, we insist on a scheduling that matches the intuitive model of parallelism, where spawned threads really execute in parallel with each other. This excludes policies which allocate a block of processors to an application without any regard to the number of threads in it.

Finally, we use the term *blocked*, as in a thread that is blocked, to mean that the tread is actually suspended from execution, and the processor switches to another thread. This is more specific than just implying it has nothing useful to do, so it can either busy wait or suspend.

2 Synchronization and Scheduling

The purpose of this paper is to compare the two basic synchronization mechanisms, busy waiting and blocking, in the context of parallel processing. Busy waiting is not used on uniprocessor multiprogrammed systems because of the obvious waste it entails: one process consumes CPU cycles waiting for another process to advance, but the other process cannot advance because it

```
parfor i := 1 to n
{
    for j := 1 to k
    {
        compute for t_p^ij time
        sync
    }
}
```

Figure 1: *Pseudo-code of application model.*

does not have the CPU. On the other hand, if the threads are actually running in parallel on distinct processors, as is possible in parallel machines, busy waiting seems to be the fastest and most direct way to synchronize. It still might cause waste, however, if there are more threads than processors, because a thread might again wait for another thread that is not running. The resulting performance is therefore strongly dependent on the system's scheduling policy.

Two-phase blocking, in which a thread first busy-waits for a while and then blocks [21], is omitted from this comparison. The reason is that when gang scheduling is used, and the synchronization is fine-grained, only the first phase of two-phase blocking is executed, making it identical to busy waiting. Without gang scheduling, it is almost identical to blocking (but with a larger overhead). Thus examining it would not add any new information.

As the effectiveness of busy waiting depends on whether or not the threads are actually running in parallel, we consider two cases. The first is when *gang scheduling* is used, meaning that interacting threads are always scheduled to run side by side simultaneously. The second is when the scheduling is performed independently on each processor in an uncoordinated manner; tasks are scheduled regardless of the state of any other task. As the combination of gang scheduling with blocking does not make sense, we are left with the following three methods:

- Busy waiting with gang scheduling.
- Busy waiting with uncoordinated scheduling.
- Blocking with uncoordinated scheduling.

The comparison is done by selecting a simple model of application behavior, and calculating the expected run time under the different methods as a function of a number of parameters. These parameters include a characterization of the granularity of the interactions among threads, the load, the gang size, and the scheduling time quantum and the context-switch overhead of the run-time system.

2.1 Model and Assumptions

Application Modeling

The application is modeled as a set of n interacting threads, where n, the gang size, is less than or equal to the number of processors. This is expressed as a `parfor` loop, and is not to be confused with `doall` loops; `doall` specifies that the iterations *may* be done in parallel, while `parfor` specifies that they *should* be done in parallel (Fig. 1). It is assumed that the identity of the interacting threads is known, i.e. the threads are declared to be a gang. The threads are iterative, as in the models of Vrsalovic et. al. [29] or Dubois and Briggs [9]. In each iteration, each thread computes for a certain time, and then all the threads perform a barrier synchronization. The processing time of thread i in iteration j is given by the random variable t_p^{ij}. It is assumed that this computation is local, and in any case it is not influenced by whatever other threads in the system are doing. The number of iterations, k, is assumed to be large enough so that various overheads may be averaged over the iterations disregarding end effects.

This model is representative of many parallel algorithms, which are designed as a large number of parallel computation phases separated by barrier synchronizations or sequential phases. It also provides a good approximation of the performance of divide-and-conquer algorithms, where each recursive level is replaced by a synchronized iteration [18], and pipelines algorithms, especially for systolic arrays. For $n = 2$, the model is reduced to a simple synchronization between two threads. This too is a common situation, which occurs in synchronous message passing, remote procedure calls, rendezvous, etc. It should be noted, however, that this is not meant to be a completely general model — it is just a simple case that is easy to analyze.

The equations to be subsequently derived in section 2.2 give the expected time needed to complete a single iteration, using the different synchronization and scheduling schemes. As this deals with the average time for an iteration, it is convenient to base the equations on the average processing time and the average waiting time. The average processing time, denoted by t_p, is simply

$$t_p = \frac{1}{nk} \sum_{i=1}^{n} \sum_{j=1}^{k} t_p^{ij}. \tag{1}$$

To define the average waiting time, we first define the maximal processing time in a certain iteration j. This is

$$t_p^{\max j} = \max_{1 \le i \le n} t_p^{ij}. \tag{2}$$

If each thread were to execute on a dedicated processor, threads that compute for less time would have to wait for the one that computes the most. The waiting time for thread i in iteration j is therefore

$$t_w^{ij} = t_p^{\max j} - t_p^{ij}, \tag{3}$$

and the average waiting time, denoted by t_w, is

$$t_w = \frac{1}{nk} \sum_{i=1}^{n} \sum_{j=1}^{k} t_w^{ij}. \tag{4}$$

Note that $t_p + t_w = \frac{i}{k} \sum_{j=1}^{k} t_p^{\max j}$, i.e. this is the average of the full iteration times as dictated by the slowest thread in each one.

The barrier synchronization itself also takes some time. Instead of modeling this independently, we observe that this is an added overhead to all the threads. Hence we include it in each thread's value of t_p^{ij}, and therefore also in t_p. Note that the synchronization overhead may depend on the number of processors, so when the model is used to predict performance t_p should be adjusted accordingly.

System Characterization

Recall that we are dealing with general purpose multiprogrammed systems, where there may be many more threads than processors. It is assumed that time sharing is used to service all the threads at the same time, rather then running only a subset to completion. The threads are mapped to processors when they are spawned. It is assumed that the total number of threads is a multiple of the number of processors, and that there is perfect load balancing. The number of threads on each processor is denoted by ℓ. We focus our attention on a single gang of n threads, mapped to n different processors. The character of the other $n(\ell-1)$ threads running on these processors does not interest us in the general case. The performance of the blocking mechanism, however, does depend also on the other threads. In this case we check two possibilities: (i) that they have an identical iterative behavior, and (ii) that they are independent compute-bound threads.

The scheduler is assumed to be perfectly fair, giving the same service to each thread (or each gang). The scheduling time quantum is denoted by τ_q, and the context switching overhead is denoted by τ_{cs}. It is assumed that $\tau_{cs} \ll \tau_q$. Blocking is assumed to cost a factor of α more than a regular context switch, where α is a constant somewhat larger than 1.

Two cases are investigated: that of coarse-grain interactions, in which t_p is large relative to τ_q (i.e. many time quantums are required before a synchronization), and fine-grain interactions, in which it is relatively small. The number of iterations, k, is assumed to be large enough so that $k(t_p + t_w) \gg \tau_q$ in any case. This allows us to average over a number of scheduling rounds. Note that k may have to be very large in the fine grain case, as it is realistic to assume τ_q to be on the order of $10^4 - 10^5$ instructions, and an interaction may occur every $10-100$ instructions. Note also that the granularity relates to the *interaction rate* of the threads, not to their life time.

2.2 Performance Derivation

Busy-Waiting with Gang Scheduling

When gang scheduling is used, the situation with respect to the interactions between the threads is identical to that in which the threads run on dedicated processors. The only difference is that we should also take into account the time allocated to other threads that share the use of the same processors, and the context switches that are involved. The granularity does not have any effect. The total run time for k iterations is therefore given by

$$T = \left(k(t_p + t_w) + \frac{k(t_p + t_w)}{\tau_q} \tau_{cs} \right) \ell. \tag{5}$$

Dividing by k and rearranging, the average time for a single iteration is

$$t = \left(1 + \frac{\tau_{cs}}{\tau_q} \right) (t_p + t_w)\, \ell. \tag{6}$$

In effect, this equation shows that an iteration takes $t_p + t_w$ time on average, meaning that the rate is dictated by the slowest thread. In addition, there is an overhead factor of τ_{cs}/τ_q. In the fine-grain case, this means that the overhead is amortized across a large number of iterations, because many iterations are completed in each scheduling round.

Busy-Waiting with Uncoordinated Scheduling

The behavior of busy waiting with uncoordinated scheduling depends on the granularity. In the coarse-grain case, it is nearly identical to the behavior of busy waiting with gang scheduling. In every scheduling round, each thread executes on its respective processor, but this does not necessarily happen at the same time within the round. Given that $t_p + t_w$ is substantially larger than τ_q, it is obvious that each iteration is spread over a number of scheduling rounds. Specifically, there are $\lfloor (t_p + t_w)/\tau_q \rfloor$ full scheduling rounds, in which the whole time quantum is used, and then a final round in which the synchronization is completed. The difference between the gang scheduling scheme and the uncoordinated scheme is apparent only in the final round; therefore the difference in run times is relatively small. To summarize, equation (6) is a good approximation of the run time for both busy-waiting schemes in the case of coarse-grain interactions.

With fine-grain interactions, however, the situation is more complicated: the fact that the n threads are not running simultaneously might change the waiting time. Specifically, there is a certain probability that the n threads happen to be scheduled simultaneously, even if no explicit measures are taken to ensure gang scheduling. When this happens, many iterations are completed. If, on the other hand, there is no overlap between the executions of any two threads, then the whole gang can only complete a single iteration.

The expected overlap of n segments of length λ that are placed at random in a loop of circumference Λ, where $\Lambda \geq \lambda$, is λ^n/Λ^{n-1}. In our case, $\lambda = \tau_q$ represents the execution of a single thread, and $\Lambda = \ell(\tau_q + \tau_{cs})$ is the duration of a scheduling round. The expected overlap, i.e. the time in which all the threads happen to execute simultaneously, is therefore $\tau_q^n/\ell^{n-1}(\tau_q + \tau_{cs})^{n-1}$. The expected number of iterations that will be completed in this time is $\tau_q^n/\ell^{n-1}(\tau_q + \tau_{cs})^{n-1}(t_p + t_w)$ (assuming this is not less than 1; see below). Hence the number of scheduling rounds needed to complete k iterations is $m = k\ell^{n-1}(\tau_q + \tau_{cs})^{n-1}(t_p + t_w)/\tau_q^n$, and the total run time is

$$\begin{aligned} T &= m\,(\tau_q + \tau_{cs})\,\ell \\ &= \frac{k\ell^n\,(\tau_q + \tau_{cs})^n\,(t_p + t_w)}{\tau_q^n}. \end{aligned} \tag{7}$$

The expected time for a single iteration is then

$$t = \left(1 + \frac{\tau_{cs}}{\tau_q}\right)^n (t_p + t_w)\,\ell^n. \tag{8}$$

Assuming that $\tau_q \gg \tau_{cs}$, this is about a factor of ℓ^{n-1} slower than with gang scheduling. The higher the load, the smaller the probability of being scheduled simultaneously, thus increasing the probability of wasting the rest of the time quantum. As a side note, we observe that two-phase blocking can be used to place a bound on the waste, but it cannot improve the performance to the level achieved by gang scheduling.

If the load is too high or the grain not fine enough, the equations might show that less than one iteration is completed each time. This is of course not true. In the correct equations, the number of completed iterations is the maximum between the expression given above and 1, and the expected time per iteration is the minimum between equation (8) and $(\tau_q + \tau_{cs})\ell$.

Blocking Mechanism

Blocking is an alternative to busy waiting. When a thread must wait for the completion of iteration j, it is suspended until the awaited threads accumulate an additional t_w^{ij} run-time. At this time the waiting thread is moved to the ready queue; it gets to run again on the subsequent round. Note that with blocking the burden of synchronization lies with the operating system. We assume that the overhead incurred is α times that of a regular context switch, where α is a constant larger than one.

The fact that the waiting threads do not consume CPU cycles during their wait does not reduce the duration of the slower computations. Its effect is to reduce the number of threads that compete for processor usage. We therefore have to make some assumption about the behavior of the competing threads. Two possibilities are considered: (i) all the threads in the system have the same behavior, i.e. they all compute and synchronize iteratively, and block when they have to wait; and (ii) the competing threads are independent and compute bound, so they never block.

As usual, both possibilities must be investigated in the coarse-grain and the fine-grain cases. In the coarse-grain case, most of the context switches are the result of a time quantum that expires, and do not involve blocking. The extra overhead in blocking is negligible. Based on the assumption that all the threads display the same behavior, only a fraction $\frac{t_p}{t_p+t_w}$ of them are active at any given moment. The expected total run time is therefore

$$T = \left(k(t_p + t_w) + \frac{k(t_p + t_w)}{\tau_q}\tau_{cs} \right) \frac{t_p}{t_p + t_w}\ell, \tag{9}$$

and the time per iteration is

$$t = \left(1 + \frac{\tau_{cs}}{\tau_q} \right) t_p \ell. \tag{10}$$

Comparing this with equation (6), we find that the effective length of each iteration is reduced from $t_p + t_w$ to t_p. Therefore this result can also be interpreted as an execution of ℓ threads where each iteration takes the average computation time rather than the maximum time.

If we do not assume that all the other $\ell - 1$ threads have the same characteristics, i.e. that some of them also block, then the number of active threads remains ℓ. In this case, the blocked threads gain no advantage. The average time per iteration is again given by equation (6). However, the blocked threads do reduce the load on the system, freeing resources for their competitors. Thus blocking is an altruistic mechanism.

Let us now consider the fine-grain case, where a thread that blocks induces a context switch. In this case the blocking may be said to cause the system to adapt to the workload, by effectively decreasing the size of the scheduling time quantum to fit the typical interaction rate of the application. An important point to notice is that as each iteration is completed, the last thread to arrive at the barrier is not blocked. This thread can immediately continue to the next iteration, without paying the overhead. Thus in each iteration only $n - 1$ of the n participating threads incur the overhead.

Again, we start by assuming that all the threads sharing the use of the processors have the same characteristics. The average run time, which is actually the effective time quantum, becomes t_p. The context switching overhead is multiplied by α, because context switches only occur when a thread is blocked. The total run time for k iterations is therefore

$$T = k \left(t_p + \tfrac{n-1}{n}\alpha\tau_{cs} \right) \ell, \tag{11}$$

and the time per iteration is

$$t = \left(1 + \frac{(n-1)\alpha\tau_{cs}}{n t_p} \right) t_p \ell. \tag{12}$$

Like the coarse grain case, the effective length of each iteration is reduced. However, the overhead is increased relative to the busy-waiting case (equation (6)). First, blocking is more expensive than just switching; this is represented by the factor of α. Moreover, t_p appears in the denominator rather than τ_q, and in fine grained interactions we expect that $t_p \ll \tau_q$. The result is that the overhead per iteration is a constant $\frac{n-1}{n}\alpha\tau_{cs}$, instead of becoming negligible as the grain becomes finer as it does for busy waiting with gang scheduling (or two-phase blocking with gang scheduling).

If we do not assume that the other threads have the same characteristics, then their time quantum stays a whole τ_q. Thus the gang of iterative threads that we are examining still manage only one iteration per scheduling round on average, as in the above derivation, but the scheduling round does not get shorter. The run time in this case is approximately

$$T = k\left(t_p + (\ell - 1)\tau_q + (\ell + \alpha - 1)\tau_{cs}\right). \tag{13}$$

As t_p and τ_{cs} are assumed to be small relative to τ_q, the cycle time of the competing threads dominates; thus the competitors again benefit more than the altruistic blocking gang. Note that this derivation depends on the assumption that the scheduler sticks to a rigid round-robin policy, which causes it to be unfair. A smart scheduler can compensate for this to some degree by giving top priority to a thread that was just resumed; in fact, this is the motivation behind giving higher priority to I/O bound jobs in many operating systems.

3 Implementation and Experiments

In order to validate the model presented in the previous section, a run-time library using a gang scheduling policy was written for the Makbilan research multiprocessor. This section provides some background about the system, and delineates the implementation of gang scheduling. Then the experimental results are described. Note that while Makbilan is a shared memory machine, the model is more general and does not rely on this feature.

3.1 Background

The Makbilan Testbed

The Makbilan research multiprocessor consists of up to 15 processor boards in a Multibus II cage. The experiments reported in the next section were run on a 10-processor configuration. Each board has an Intel 386 processor running at 20 MHz, providing about 4 MIPS. It also has a 387 mathematical co-processor, a message passing co-processor, and 4MB of memory. Memory on remote boards may be accessed through the bus, thus supporting a shared-memory model. As access to on-board memory is faster than access to memory on remote boards, Makbilan is a non-uniform memory access (NUMA) machine [5]. The processors have on-board caches, but they do not cache remote references. Hence there is no issue of cache coherence.

The box also includes one board that acts as a Unix host, a bus controller, a peripherals interface, and a terminals controller. Users log on to the Unix board, and can then load and execute *ParC* programs on all the other boards.

The *ParC* Language

ParC is a superset of the *C* programming language intended to support parallel programming in a shared memory environment [4]. The main additions over *C* are two block-oriented parallel constructs, `parblock` and `parfor`; the first indicates that the constituent sub-blocks execute in parallel, while the second indicates that iterations of the loop body be done in parallel. Each sub-block or iteration is called an *activity*; these are equivalent to threads in the discussion so far.

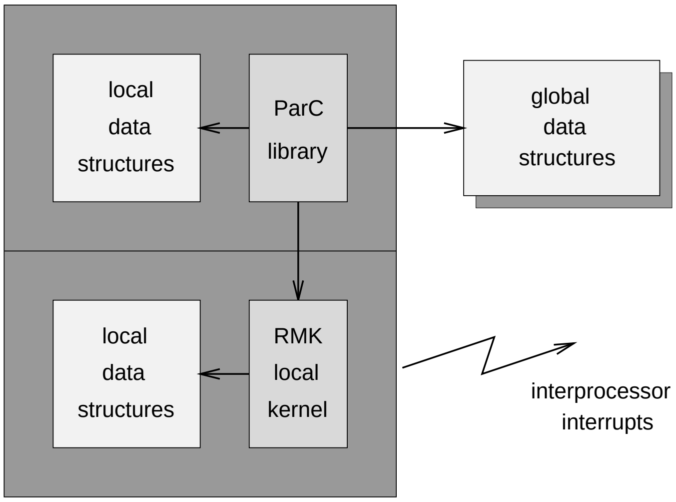

Figure 2: *MAXI system configuration on each board.*

These constructs may be nested in arbitrary ways, creating a tree of activities where all the leaves are executing in parallel. Variables declared within the block of code that defines a certain activity are accessible only by that activity and its descendants. For the work reported in this paper, the set of activities that are spawned together in a single construct is taken as the definition of a gang.

In addition to the parallel constructs, there are three main synchronization mechanisms: fetch-and-add, semaphores, and `sync`. The `sync` instruction implements a barrier synchronization among all the activities created by a certain parallel construct. However, it is not used in the experiments described in section 3.3. Rather, a specially optimized version of barrier synchronization with less overhead is used. This version is based on atomic bitwise logical operations supported by the multibus II. It cannot be used in the general implementation because it limits the number of activities that are involved.

The MAXI System

MAXI is an acronym for the **Mak**bilan **Sy**stem, based on an abuse of the English alphabet. The system may be partitioned into two main layers: A run time library that supports the *ParC* constructs, and a local kernel on each board. The local kernel is Intel's RMK [15], which is a real-time kernel designed to use hardware support provided by the 386 and the Multibus II. This kernel is highly optimized to provide fast task creation, termination, and context switching. Parallel activities are implemented by RMK tasks (which are the RMK equivalents of Unix processes). These tasks embody the threads from the analysis of section 2.

The current version supports only a single user at a time. The system design emphasizes asynchronous distributed operation without unnecessary interdependencies between boards. Thus each board has a local copy of the run-time library, complete with local data structures (Fig. 2). Global data structures in shared memory are used only when activities executing on one board need to influence what happens on other boards, e.g. when new activities are spawned or when a barrier synchronization point is reached.

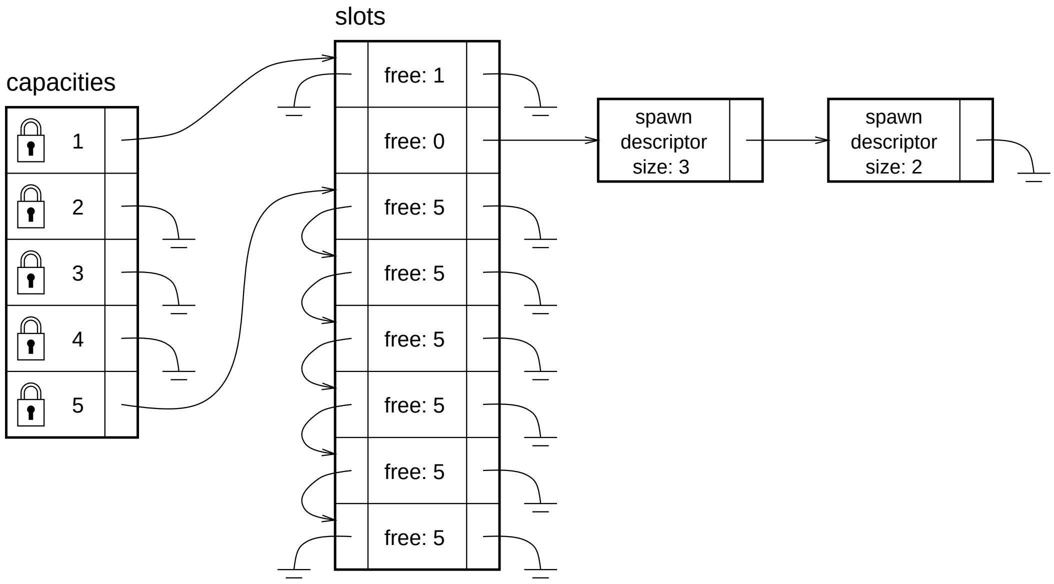

Figure 3: *Global data structures used for gang scheduling.*

3.2 The Gang Scheduling Library

In the default *ParC* run-time library, scheduling is done by the RMK kernel on each board independently. In order to implement gang scheduling, a new library with an external scheduler was written. This scheduler forces RMK to schedule the desired task by raising its priority.

The gang scheduling algorithm and data structures are based on the matrix algorithm developed by Ousterhout [21]. Scheduling space is seen as a matrix, where each row corresponds to a scheduling slot and each column to a processor. Tasks belonging to a single gang are mapped to entries in the same slot. In our implementation, the matrix is not stored explicitly; it is simply the conjunction of the PCB tables[1] on all the processors.

The scheduling is done in round-robin style, by circling through the used slots. Rescheduling is triggered when the alarm on PE #1 goes off, indicating that the time quantum has expired. Alternatively, any processor may trigger a rescheduling if it finds that no gang scheduling is taking place in the current slot. This information is mediated by a shared bit mask, with a distinct bit representing each processor. A set bit indicates that the corresponding processor is not participating in gang scheduling in the current slot: this can happen if a processor was not allocated a task in the slot, if the task is suspended, or if it terminated. A processor that finds all the bits set triggers a rescheduling.

Rescheduling is implemented by a broadcast interrupt, which causes all the processors to switch to the next slot simultaneously. When the broadcast interrupt is received, the interrupt handler reduces the priority of the current task (if any), and raises the priority of the task in the next slot. A hint about the maximal used slot is maintained to indicate when to return to the first slot.

Two global data structures are used to allocate slots to gangs (Fig. 3). The first is an array of pointers to lists of slots with a given capacity. The second is a table of slots. Each slot entry indicates how much free space there is in the slot, and the entries are linked to each other according to this value. The list is locked when a slot is manipulated. A slot can also point to a list of spawn

[1]PCB stands for Process Control Block. This is the data structure used by the run-time system to store information about the task.

	run-time library	
	default	gang sched
scheduling	$0.07 + \frac{0.14}{\ell}^*$	0.2
spawning	1.6	2.5

*The second term is due to overhead experienced once in each scheduling round, and therefore amortized.

Table 1: *Run-time library overheads (in ms).*

descriptors. These are data structures that describe gangs that have been allocated to this slot, but have not commenced yet. The allocation is static and does not change during execution. In particular, slots that are left with a small number of tasks due to the termination of other tasks are not united.

For example, Fig. 3 shows a possible configuration for a system with 5 processors and a maximum of 8 tasks per processor (the real numbers in MAXI are 15 and 512, respectively). The first slot contains active tasks on all the processors except one. The second slot has been allocated to two gangs of sizes 3 and 2, so it has no free space. When it is scheduled, the tasks will be created and the spawn descriptors removed. All the rest of the slots are unused.

The gang scheduling library suffers more overhead than the default library (Table 1) [5]. This has two reasons: first, it requires more coordination through global variables and broadcasts. Second, various functions which would normally be implemented inside the kernel are actually implemented above it, and use a sequence of kernel calls to achieve the desired end result. The next section shows that despite this higher overhead, the gang scheduling version does indeed perform better for fine-grain applications.

3.3 Experimental Results

The experiments used to verify the model of section 2 are based on a synthetic program that simulates interactions with various degrees of granularity. The program spawns gangs with one thread per processor. These threads loop a large number of times and synchronize in each iteration. The average time to complete an iteration and synchronize is measured. Each instance of the program, i.e. each execution, is characterized by three parameters:

- `LOAD` — the number of competing gangs.
- `GRAIN` — the number of instructions in each iteration, excluding the code that implements the synchronization. If different activities in the gang have different granularities, this is the minimal one.
- `VAR` — the difference between the minimal and maximal numbers of instructions in different activities in each iteration. The actual numbers are selected at random from a uniform distribution between `GRAIN` and `GRAIN`+`VAR`.

As the execution times are selected from a uniform distribution, the expected execution time of the longest activity in any iteration is $\texttt{GRAIN} + \frac{n}{n+1}\texttt{VAR}$, and that of the shortest activity is $\texttt{GRAIN} + \frac{1}{n+1}\texttt{VAR}$, where n, the number of activities, is equal to the number of processors. 10 were

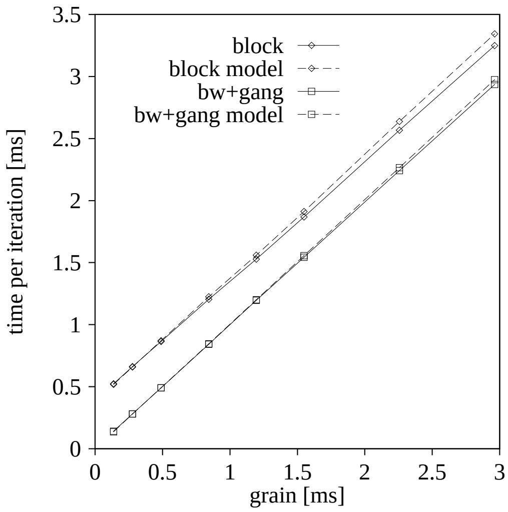

Figure 4: *Results of experiment 1.*

used in the experiments. The system time quantum is 50 ms. Measurements show that each unit of `GRAIN` takes 0.00141 ms, and the interaction at the end of each iteration takes an additional 0.137 ms on 10 processors. Thus the relationship between the experimental parameters and the model parameters used in section 2 is the following:

$$\begin{aligned} t_p &= 0.137 + 0.00141 \cdot (\texttt{GRAIN} + \tfrac{1}{2}\texttt{VAR}) \\ t_w &= 0.00058 \cdot \texttt{VAR} \end{aligned} \tag{14}$$

The above expressions and values, and the overheads shown in table 1, are used to calculate model predictions. Equations (6) and (12) are used. The predictions are then compared with the actual measurements.

The number of iterations that were measured was 30000 or 50000 in most of the experiments. Repeated measurements show that this is large enough so that the accuracy is within 5%, and in most cases even within 1% (except for some of the results of experiments 3 and 4, see below). To prevent situations in which the starting conditions cause the system to settle into a scheduling pattern that affects synchronization performance, a set of skewing threads is generated at the outset. These threads execute for random durations of up to one quantum. Thus they terminate early in the measurement, and serve to cause the different processors to start asynchronously.

There are two versions of the program: one using busy waiting and the other using blocking. A simplified version of blocking was used, where threads just yield the processor but do not join any explicit blocked queue. In effect, the run queue serves as a blocked queue as well. This saves the need for explicit dequeueing. As a result, the blocking overhead is not constant. Rather, it depends on how many times the waiting thread yielded the processor. The program counts this parameter, and its average value is used for α in the model predictions.

Experiment 1

This experiment shows the additive overhead incurred by blocking, and also that blocking cannot achieve any gains when the load is 1 (e.g. on a dedicated machine). `VAR` is set to 0. With busy waiting the time per iteration is essentially the time needed for `GRAIN` instructions plus synchronization. With blocking, each iteration suffers an additional constant overhead. The measured and predicted

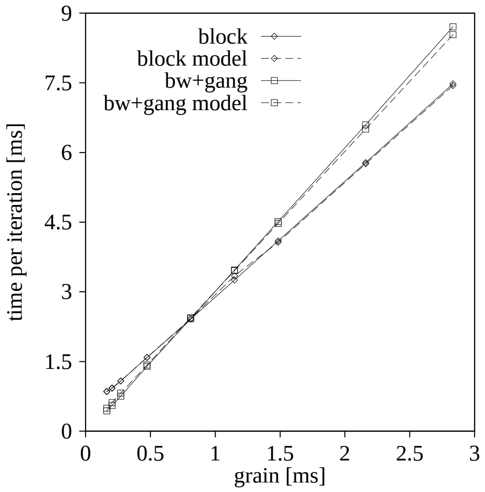

Figure 5: *Results of experiment 2.*

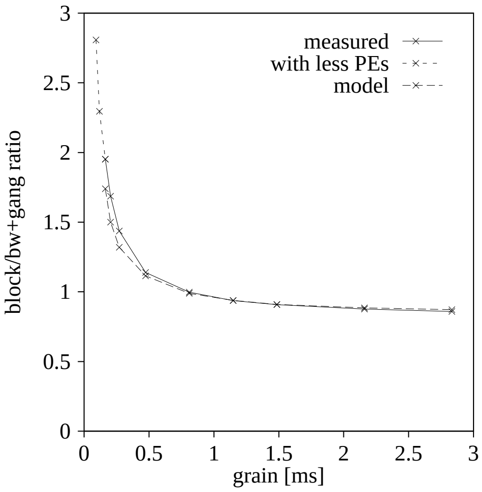

Figure 6: *Ratio of time required by blocking to time required by busy waiting with gang scheduling.*

results are shown in Fig. 4. The x axis is the time needed to complete an iteration on a dedicated machine, $t_p + t_w$. The y axis shows the time that was actually required. The model equations for the fine-grain case were used, and indeed the agreement with the measurements deteriorates slightly as the grain becomes larger.

Experiment 2

This is the main experiment used to verify the performance relations between busy waiting with gang scheduling and blocking with uncoordinated scheduling. `LOAD` is set to 3, and `VAR` is equal to `GRAIN`. As expected, when the granularity is large enough, blocking can use the idle time of waiting activities to execute other activities, thus reducing the average time per interaction (Fig. 5). For fine-grain interactions, however, the blocking overhead dominates the possible gain.

Note that the crossover occurs at a granularity of about 0.8 ms; for smaller granularity, gang scheduling is better. As the execution time of fine grain computations is small, the difference between the two schemes is also small in absolute terms. The relative performance gains, however, are unmistakable. The measurements show that for the most fine-grain interactions that were measured, busy waiting with gang scheduling was twice as fast as blocking (Fig. 6). This was a granularity of about 0.16 ms. The results indicate an obvious increase in the relative performance ratio for smaller granularities. Measurements on a smaller number of processors, where the barrier synchronization overhead is smaller, confirm this trend (dotted line in the figure).

Experiment 3

This experiment demonstrates the altruistic nature of blocking. Both `GRAIN` and `VAR` are set equal to 1000. Two versions of the test program with blocking are used. In one all the competing gangs have the same iterative nature with the same granularity, and block when they try to synchronize. In the other, only one gang has these characteristics. The competing gangs are composed of compute-intensive activities that do not block.

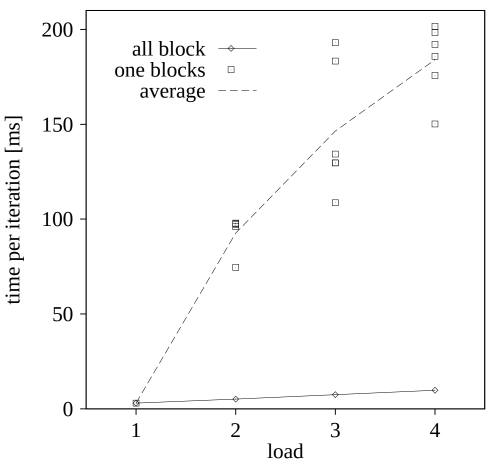

Figure 7: *Results of experiment 3.*

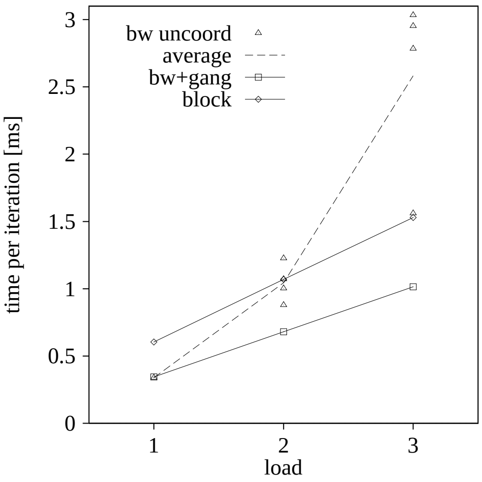

Figure 8: *Results of experiment 4.*

The results indicate that the time per iteration of the blocking gang is approximately the product of the number of competing gangs times the scheduling time quantum, as expected from equation (13). The exact relation depends on the details of the scheduling pattern that the system falls into, hence the large variance in the measured results.

The slightly non-linear characteristic of the curve results from the fact that α changes from about 4 for a load of one down to about 1.2 for a load of four. The reason for this is that at higher loads there is a larger delay until a task is rescheduled, so the rest of the gang has a better chance to reach the synchronization point.

Experiment 4

This experiment shows that busy waiting without gang scheduling is indeed not a viable alternative (Fig. 8). Pairs of threads are used ($n = 2$), with `GRAIN` and `VAR` set to 100. As expected, there is a linear dependence between the required time and the load for busy waiting with gang scheduling and for blocking. While the results for busy waiting with uncoordinated scheduling are not as accurate, because they depend on the exact pattern that the system falls into, it is evident that in this case the dependence is quadratic.

4 Discussion and Conclusions

Based on the model and experiments, we can derive the following conclusions regarding synchronization mechanisms and scheduling policies:

1. When busy waiting with gang scheduling is compared with blocking, the relative performance is a function of the granularity. For fine-grain jobs, busy waiting with gang scheduling is better.

2. Blocking altruistically frees system resources. If *all* the jobs are unbalanced and coarse grained, the run time is reduced. For fine-grained jobs, however, the blocking overhead dominates any possible savings and thus degrades the performance. If competing jobs do not block, a job that does use blocking may receive disproportionally low service.

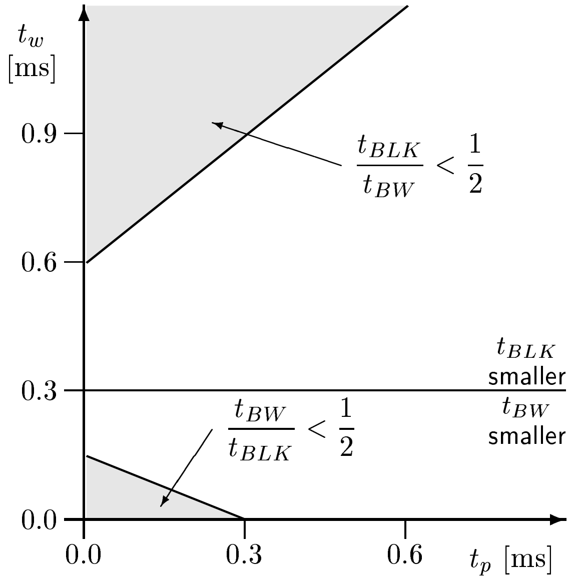

Figure 9: *Phase diagram comparing the relative performance of busy waiting with gang scheduling vs. blocking, on the Makbilan multiprocessor.*

3. Busy waiting with uncoordinated scheduling is wasteful, especially in fine-grained jobs. With very coarse-grain interactions, however, it is basically similar to busy waiting with gang scheduling.

4. Two-phase blocking limits the waste, but does not provide any real benefits unless it is used together with gang scheduling.

When to use busy waiting with gang scheduling

The comparison between busy waiting with gang scheduling and blocking can be quantified in our model in the following manner. Denote the expected time per iteration when using busy waiting and gang scheduling by t_{BW}, and the time when using blocking by t_{BLK}; these are given in equations (6) and (12), respectively. Note that for blocking this rests on the assumption that all the threads in the system display the same behavior and block to save system resources. We would like to know when one is smaller than the other by a certain factor σ. In other words, we would like to find a condition that will cause the inequality $\frac{t_{BW}}{t_{BLK}} \leq \sigma < 1$ to be satisfied. By using the equations and changing sides, it is easy to see that the condition is

$$t_w \leq \frac{\sigma \, \frac{n-1}{n} \, \alpha \, \tau_q \, \tau_{cs}}{\tau_q + \tau_{cs}} + \left(\frac{\sigma \tau_q}{\tau_q + \tau_{cs}} - 1 \right) t_p \qquad (15)$$

This defines a certain area in "interaction space", i.e. the quarter-plane consisting of all possible combinations of t_p and t_w, using coefficients that are a function of system parameters. The area is delimited by a straight line with a negative slope, yielding that corner in interaction space where both t_p and t_w are small. In this area, busy waiting with gang scheduling is at least $1/\sigma$ times faster than blocking.

The same approach may be used to find a condition that guarantees that $\frac{t_{BLK}}{t_{BW}} \leq \sigma < 1$, i.e.

that blocking has the advantage. The condition turns out to be

$$t_w \geq \frac{\frac{n-1}{n}\,\alpha\,\tau_q\,\tau_{cs}}{\sigma(\tau_q+\tau_{cs})} + \left(\frac{\tau_q}{\sigma(\tau_q+\tau_{cs})} - 1\right) t_p \tag{16}$$

This is again a straight line, but now the area of interest is above it.

The two expressions converge when $\sigma = 1$. This gives a line with a negative slope, but assuming that $\tau_{cs} \ll \tau_q$ the slope is very very small. In effect, this line therefore divides the interaction space into two. The lower part, where t_w is smaller than the blocking overhead ($t_w < \alpha\tau_{cs}$), includes all combinations in which busy waiting with gang scheduling has the advantage. The upper part includes those interactions in which blocking is better.

Applying the above formulas to the parameters that characterize the Makbilan system ($\tau_q = 50$, $\tau_{cs} = 0.2$, $\alpha = 1.5$), and using $\sigma = \frac{1}{2}$, we get the phase diagram of Fig. 9. The lower shaded area is the part of interaction space where busy waiting with gang scheduling is at least twice as good as blocking[2]. The shaded triangle at the top left is the part where blocking is twice as good. The white area indicates combinations in which the difference is smaller than a factor of two. This area is neatly divided, and busy waiting has the advantage whenever t_w is smaller than 0.3 ms.

Obviously the advantage of busy waiting with gang scheduling increases for finer grained interactions. The specific area in the figure is defined by the requirement that $t_p + 2t_w \leq 300\mu$s; given that each Makbilan processor is capable of about 4 MIPS, this represents a granularity of up to a few hundred instructions. In a multiprocessor based on modern RISC microprocessors the graularity would be even larger, because the context is bigger. Blocking only has a decided advantage when t_w is larger than t_p by 0.6 ms. This happens only if the computational tasks of the different threads are highly unbalanced.

Note that all the above was done under assumptions that favor blocking, namely that *all* the competing gangs display the same behavior. If this assumption is dropped, equation (13) should be used for t_{BLK} rather than equation (12). This leads to the following condition for the advantage of busy waiting and gang scheduling:

$$t_w \leq \frac{\sigma\,\tau_q(\ell-1)}{\ell} + \frac{\sigma\,\alpha\,\tau_q\,\tau_{cs}}{(\tau_q+\tau_{cs})\ell} + \left(\frac{\sigma\tau_q}{(\tau_q+\tau_{cs})\ell} - 1\right) t_p. \tag{17}$$

This is again the fine-grain corner of interaction space, but due to the first term it is a much larger corner. The other condition changes in a similar manner.

The conclusion from the above is that busy waiting with gang scheduling is a viable and promising method for the implementation of fine-grain parallel systems, a target that has been problematic to date. Its importance lies in the fact that many algorithms are naturally expressed using small parallel blocks of code, and the finer the granularity the larger the degree of parallelism that is exposed [8]. Gang scheduling allows busy waiting to be used for synchronization, which allows fine grain threads to be supported. In addition, busy waiting has an advantage on multiprocessors with caches, as the frequent context switches induced by blocking may make caching much less effective [20, 14]. Moreover, busy waiting can actually utilize the cache coherence mechanism to reduce network load [24, 2].

However, blocking is easier to implement than gang scheduling, so blocking is used in most of the parallel systems existing today. Consequently fine-grained algorithms have to be restructured to run efficiently on contemporary coarse-grain systems (see, e.g., [25, 19, 8]). This places an unnecessary

[2]This does not correspond exactly with the results of experiment 2 because the actual values of α in that experiment were different for different data points, and varied between 1.1 and 2.1.

burden on the programmer and the compiler. In addition, it is imperative that systems that use blocking raise the priority of tasks that become unblocked. If this is not done, the altruistic nature of blocking can cause these tasks to suffer severe performance degradation.

While gang scheduling would provide better support for fine-grain computations, this approach too has its limitations. Specifically, a gang cannot involve more than P threads, where P is the number of processors. This does not mean that applications cannot spawn more threads: it only means that larger groups should not interact simultaneously. In effect, the application is required to display *interaction locality*, which is analogous to the requirement for reference locality in virtual memory. The gangs are actually "thread working sets", and gang scheduling is a means to prevent thrashing [21]. While this requirement may seem restrictive to the point of limiting the usefulness of gang scheduling, this is in fact not so. Scheduling policies that schedule threads in an uncoordinated manner implicitly require the threads to be independent, which is much more restrictive.

It should be emphasized that this definition of interaction locality is completely different from the often mentioned requirement that programs display communication locality. Communication locality refers to cases in which the hardware has a certain topology, and threads are required to communicate with only a small subset of the other threads, so as to facilitate the mapping of threads to processors. Interaction locality means that threads may be grouped into gangs with no more than P threads each, such that the vast majority of the interactions do not cross gang boundaries. However, a thread may interact with all the other members of its gang. This has nothing do to with topology.

Implications

The support of fine-grain computations through busy waiting and gang scheduling requires adaptations in various areas of parallel computing. For example, explicitly parallel programming languages should give the compiler and run-time system information about threads that can be expected to interact strongly, e.g. through the syntactic structure of parallel blocks. Alternatively, compile-time dependency analysis can be used to glean information about interaction patterns and granularity. Automatic parallelization may also produce threads that must interact: for example, this happens with `doacross` loops [22]. If the granularity of interactions is fine enough, the relevant threads should be marked for gang scheduling.

The implementation of gang scheduling also requires additional research. To date, only a small number of parallel operating systems incorporate preemptive gang scheduling [21], while some others support batch-style space-division sharing which has similar features [6, 3]. New algorithms and control structures are needed to support gang scheduling on increasingly larger machines [11, 12]. Hardware support for the operating system may also be needed. For example, in our implementation the ability to broadcast interprocessor interrupts was necessary.

It is conceivable that special hardware support might also be necessary for the efficient implementation of inter-thread interactions. Gang scheduling guarantees that threads will find their interaction partners, but if the interactions themselves take too long they will violate the fine-grain time scale [8]. For example, hardware supporting access to shared variables conditioned on a full/empty status bit can save explicit busy waiting, resulting in faster operation and reducing the load on the communication network; this already exists in a number of systems [27, 1]. Hardware support for barrier synchronization is also advocated [23, 28]. In addition, it would be beneficial to have caching with hardware support for cache coherence, as this can reduce the contention and further reduce the cost of busy waiting [2]. Of course, the opposite point of view should also be remembered. Systems that incorporate hardware support for synchronization will not utilize this support if the synchronizing threads do not execute simultaneously. Thus gang scheduling is needed to ensure the efficiency of hardware mechanisms and justify the investment in them.

Acknowledgments

ParC was developed by Yosi Ben-Asher. Much of the original version of the MAXI system was written by Moshe Ben Ezra and Lior Picherski. We thank the anonymous referees for their comments.

We thank Intel Corp. for their generous equipment donation, without which the Makbilan multiprocessor could not be built. This research was funded in part by the US-Israel Binational Science Foundation grant no. 88-00045. Dror Feitelson was supported by an Eshkol fellowship from the Ministry of Science and Technology, Israel.

References

[1] Alverson, R., Callahan, D., Cummings, D., Koblenz, B., Porterfield, A., and Smith, B. The Tera computer system. *Intl. Conf. Supercomputing.* Jun 1990, pp. 1–6.

[2] Anderson, T. E. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel & Distributed Syst.* **1,** 1 (Jan 1990), 6–16.

[3] Anderson, T. E., Bershad, B. N., Lazowska, E. D., and Levy, H. M. Scheduler activations: effective kernel support for the user-level management of parallelism. *ACM Trans. Comput. Syst.* **10,** 1 (Feb 1992), 53–79.

[4] Ben-Asher, Y., Feitelson, D. G., and Rudolph, L. ParC — an extension of C for shared memory parallel processing. Manuscript, Dept. Computer Science, The Hebrew University of Jerusalem, Oct 1990. Submitted for publication.

[5] Ben-Asher, Y., and Feitelson, D. G. Performance and Overhead measurements on the Makbilan. Technical Report 91-5, Dept. Computer Science, the Hebrew University of Jerusalem, Oct 1991.

[6] Black, D. L. Scheduling support for concurrency and parallelism in the Mach operating system. *Computer* **23,** 5 (May 1990), 35–43.

[7] Błażewicz, J., Drabowski, M., and Węglarz, J. Scheduling multiprocessor tasks to minimize schedule length. *IEEE Trans. Comput.* **C-35,** 5 (May 1986), 389–393.

[8] Chen, D-K., Su, H-M., and Yew, P-C. The impact of synchronization and granularity on parallel systems. 17th *Ann. Intl. Symp. Computer Architecture Conf. Proc.* May 1990, pp. 239–248.

[9] Dubois, M., and Briggs, F. A. Performance of synchronized iterative processes in multiprocessor systems. *IEEE Trans. Softw. Eng.* **SE-8,** 4 (Jul 1982), 419–431.

[10] Edler, J., Gottlieb, A., and Lipkis, J. Considerations for massively parallel UNIX systems on the NYU Ultracomputer and IBM RP3. *EUUG (European UNIX system User Group) Autumn '86 Conf. Proc.* Sep 1986, pp. 383–403.

[11] Feitelson, D. G., and Rudolph, L. Distributed hierarchical control for parallel processing. *Computer* **23,** 5 (May 1990), 65–77.

[12] Feitelson, D. G., and Rudolph, L. Mapping and scheduling in a shared parallel environment using distributed hierarchical control. *Intl. Conf. Parallel Processing.* Aug 1990, vol. I, pp. 1–8.

[13] Feitelson, D. G., and Rudolph, L. Wasted resources in gang scheduling. 5th *Jerusalem Conf. Information Technology.* IEEE Computer Society Press, Oct 1990, pp. 127–136.

[14] Gupta, A., Tucker, A., and Urushibara, S. The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications. *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.* May 1991, pp. 120–132.

[15] Intel Corporation. *iRMK I.2 Real-Time Kernel Reference Manual.* 1988. Order number 462660-001.

[16] Leutenegger, S. T., and Vernon, M. K. The performance of multiprogrammed multiprocessor scheduling policies. *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.* May 1990, pp. 226–236.

[17] Lo, S-P., and Gligor, V. D. A comparative analysis of multiprocessor scheduling algorithms. 7th *Intl. Conf. Distributed Computing Systems.* Sep 1987, pp. 356–363.

[18] Madala, S., and Sinclair, J. B. Performance of synchronous parallel algorithms with regular structures. *IEEE Trans. Parallel & Distributed Syst.* **2,** 1 (Jan 1991), 105–116.

[19] McCreary, C., and Gill, H. Automatic determination of grain size for efficient parallel processing. *Comm. ACM* **32,** 9 (Sep 1989), 1073–1078.

[20] Mogul, J. C., and Borg, A. The effect of context switches on cache performance. 4th *Intl. Conf. Architect. Support for Prog. Lang. & Operating Syst.* Apr 1991, pp. 75–84.

[21] Ousterhout, J. K. Scheduling techniques for concurrent systems. 3rd *Intl. Conf. Distributed Computing Systems.* Oct 1982, pp. 22–30.

[22] Polychronopoulos, C. D., Kuck, D. J., and Padua, D. A. Execution of parallel loops on parallel processor systems. *Intl. Conf. Parallel Processing.* Aug 1986, pp. 519–527.

[23] Polychronopoulos, C. D. Compiler optimizations for enhancing parallelism and their impact on architecture design. *IEEE Trans. Comput.* **37,** 8 (Aug 1988), 991–1004.

[24] Rudolph, L., and Segall, Z. Dynamic decentralized cache schemes for MIMD parallel processors. 11th *Ann. Intl. Symp. Computer Architecture Conf. Proc.* Jun 1984, pp. 340–347.

[25] Sarkar, V. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* MIT Press, 1989.

[26] Seager, M. K., and Stichnoth, J. M. Simulating the Scheduling of Parallel Supercomputer Applications. Technical Report UCRL-102059, Lawrence Livermore National Laboratory, Sep 1989.

[27] Smith, B. J. A pipelined, shared resource MIMD computer. *Intl. Conf. Parallel Processing.* 1978, pp. 6–8.

[28] Stone, H. S. *High-Performance Computer Architecture.* Addison-Wesley, 2nd ed., sect. 7.2.5, 1990.

[29] Vrsalovic, D. F., Siewiorek, D. P., Segall, Z. Z., and Gehringer, E. F. Performance prediction and calibration for a class of multiprocessors. *IEEE Trans. Comput.* **37,** 11 (Nov 1988), 1353–1365.

[30] Zahorjan, J., Lazowska, E. D., and Eager, D. L. The effect of scheduling discipline on spin overhead in shared memory parallel systems. *IEEE Trans. Parallel & Distributed Syst.* **2,** 2 (Apr 1991), 180–198.

[31] Zahorjan, J., Lazowska, E. D., and Eager, D. L. Spinning Versus Blocking in Parallel Systems with Uncertainty. Technical Report 88-03-01, Dept. Computer Science, University of Washington, Mar 1988.