Coscheduling Based on Run-Time Identification of Activity Working Sets

Dror G. Feitelson
IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598
feit@watson.ibm.com

Larry Rudolph
Institute of Computer Science
The Hebrew University of Jerusalem
91904 Jerusalem, Israel
rudolph@cs.huji.ac.il

Abstract

This paper introduces a method for runtime identification of sets of interacting activities ("working sets") with the purpose of *coscheduling* them, i.e. scheduling them so that all the activities in the set execute simultaneously on distinct processors. The identification is done by monitoring access rates to shared communication objects: activities that access the same objects at a high rate thereby interact frequently, and therefore would benefit from coscheduling. Simulation results show that coscheduling with our runtime identification scheme can give better performance than uncoordinated scheduling based on a single global activity queue. The finer-grained the interactions among the activities in a working set, the better the performance differential. Moreover, coscheduling based on automatic runtime identification achieves about the same performance as coscheduling based on manual identification of working sets by the programmer.

Keywords: coscheduling, gang scheduling, on-line algorithms, activity working set.

1 Introduction

The performance of multiprogrammed multiprocessors can be improved without additional burden on the programmer if the sets of closely interacting, fine-grained activities can be identified and scheduled to always execute simultaneously. Some twelve years ago John Ousterhout noticed an analogy between memory management in multiprogrammed uniprocessor operating systems and processor management in multiprogrammed multiprocessor systems [27]. The analogy was based on the observation that parallel applications require a "working set" of processes to execute simultaneously, just like uniprocessor applications require a working set of pages to be memory resident simultaneously. If a parallel application does not receive enough processors, It might thrash due to synchronization constraints among its processes. Executing processes become blocked waiting for responses from those that are currently not executing. The machine could spend most of its time context switching if an additional process is scheduled to execute when its interacting counterpart is descheduled from execution.

We use the term *activity working set* to refer to a set of activities that should be scheduled together. We use the term *activity* rather than *process*, as in Ousterhout's "process working set", to stress the fact that it is one of many components in a parallel processing application, and to avoid overloaded terms such as *task* or *thread.* The context of our discussion is parallel systems where programming is based on control parallelism, and activities can be created dynamically during execution.

The term *coscheduling* is used to describe scheduling algorithms that schedule on the basis of activity working sets, i.e. all the threads of an activity working set are scheduled to execute simultaneously. We shall be comparing our work to *uncoordinated scheduling* in which a single global activity queue (or workpile) is maintained, and each processor removes the next activity from the workpile, executes it for a whole time quantum, and returns it to the end of the workpile — the scheduling of one activity is uncoordinated with the scheduling of any other activity [6, 17].

The subject of this paper is how to identify the activities that constitute a working set. The analogy with uniprocessor memory management is not so helpful here. The memory working set is approximated by using the least recently used (LRU) paradigm [20]. Basically, this assumes that the past is indicative of the future and hence it may be expected that the next pages that the application will need are exactly those that it used most recently. These pages are therefore maintained in the primary memory, and only the least recently used pages are evicted when additional space is needed. This approach does not easily transfer to our case, because activities are active entities; there is no such thing as an "unused" activity that can be "evicted", and because in general there can be a number of (disjoint) activity working sets within an application that each need to be scheduled simultaneously.

Activity working sets should be defined by the pattern of interactions among the activities. But it has usually been assumed that the working sets are defined by the user [27]. For example, the syntactic structure of the language may be used, by defining the set of activities that are spawned together by a single parallel construct to be a working set (Fig. 1) [9].

We suggest an automatic system that can identify the activity working set by using access rates to shared communication objects to gather information about interaction patterns at runtime. The communication objects can be named channels as in Occam [18], named barrier synchronization points, or named multiparty interactions such as scripts [13] or communicators [7]. Their role is to help expose the patterns in which activities interact and to identify the activity working set; activities which use or access the same objects are known to interact, so it is safe to conjecture that they are in the same activity working set. Having identified an activity working set there is positive and negative feedback: coscheduling all the activities in a correctly identified working set improves their performance and increases the likelihood that they will complete even more interactions thereby increasing the confidence in the working set composition; coscheduling activities that do not form a real activity working

```
PAR
  PAR
    proc.1
    proc.2
    proc.3
  proc.4
  PAR
    PAR
      proc.5
      proc.6
      proc.7
    proc.8
```

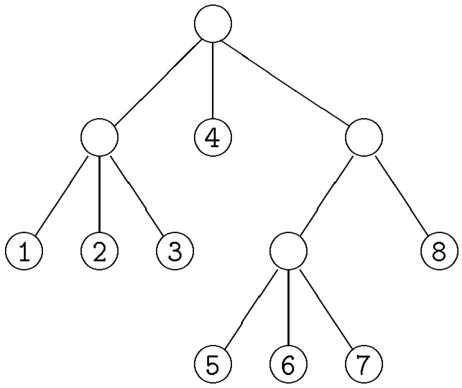


Figure 1: *A code segment (using Occam notation) and its activity tree. Sets of leaf activities with a common parent may be assumed to interact strongly with each other, and can therefore be used to define working sets. In this case there are two such sets:* $\{1, 2, 3\}$ *and* $\{5, 6, 7\}$.

set will likely reduce performance, since the activities that are required for interaction will not be simultaneously executing, leading to a decrease in the identification confidence. Our experimental results are encouraging, showing that using interaction patterns to dynamically determine activity working sets can improve overall performance. In many cases, it is just as efficient as manual identification of the working sets.

The rest of this section gives a short overview of directly related and superficially related work. Section 2 presents the coscheduling algorithm that identifies activity working sets based on the use of communication objects. Experimental results that support this approach are presented in Section 3. Section 4 concludes the paper.

Related Research Results

The terms coscheduling and gang scheduling refer to scheduling strategies that are quite similar. The salient feature of both is that a set of activities are scheduled to execute *simultaneously* on a set of processors, with a *one-to-one mapping* of activities to processors. This is done to ensure that the activities can interact efficiently. Gang scheduling schemes schedule only whole activity working sets — that is, all the activities in the working set (or gang) must be scheduled together (which obviously requires the working sets to be defined in advance). Coscheduling schemes are more flexible, in that a large fraction of the activity working set is scheduled for simultaneous execution if it is impossible to schedule all the activities in the working set. In both cases, activities (or rather, working sets) are *preemptable*, thus supporting a multiprogrammed interactive environment. The difference from simple preemptive scheduling on each processor is that preemption and scheduling are coordinated across all those processors that are executing the working set. If any processors are left over when some working sets are scheduled, they are used to schedule any activity that happens to be ready, with no regard to working sets. This is called *alternative* scheduling [27, 28]. In the sequel, we shall sometimes use "gang" as a synonym for "working set".

Coscheduling was introduced by Ousterhout [27], who also implemented it in the Medusa system on CM*. Since then, gang scheduling was implemented in a number of additional experimental systems [15, 29, 3, 10, 4]. Gang scheduling is also provided by at least five commercial systems: the Alliant FX/8 [31], the Connection Machine CM-5 [21], the Meiko CS-2 [8], the Intel Paragon [8], and the Silicon Graphics Challenge. In these systems, gangs are typically defined to include all the activities (or processes in their terminology) in the application. Off-line algorithms to find an optimal gang schedule were presented by Błażewicz et al. [2]. Feitelson and Rudolph proposed a scalable implementation of gang scheduling based on buddy systems [9, 11, 12]. In these works gangs are also predefined, even though they are not necessarily identical to jobs (i.e., a job can have multiple independent gangs). This paper is the first to consider the possibility of automatically grouping the activities in a job into gangs at run time.

There are many scheduling strategies that are superficially similar to gang and coscheduling in that parts of parallel jobs are scheduled to execute simultaneously on several processors. One basic strategy is to physically partition the parallel machine and statically assign a parallel application job to a partition. Many partitioned systems do not support preemption, effectively limiting the number of jobs that can execute simultaneously [30, 34, 33]. Other systems take this partitioning a step further, and perform dynamic re-partitioning of the processors to reflect changes in the system load. Note that if the number of processors allocated to a job is less than the number of activities in it, then they cannot execute simultaneously. Therefore a two-level scheduling scheme is used: the operating system is responsible for the allocation of processors, while the application itself schedules activities on them [32, 23]. Two-level scheduling with dynamic partitioning has been shown to be an effective scheduling scheme for small-scale uniform memory access (UMA) multiprocessors, provided applications are coded in a style that can tolerate dynamic changes in the number of processors at runtime. Under certain circumstances, its use can be extended to large distributed-memory machines as well [26, 24]. A number of papers have evaluated the performance implications of coscheduling and gang scheduling, and compared them with dynamic partitioning and other scheduling policies for multiprogrammed multiprocessors [10, 28, 22, 16, 35].

The Mach scheduler also allows processors to be allocated to jobs dynamically, but performs the scheduling of activities on these processors in the kernel rather than leaving it to the application [1]. Like the other dynamic partitioning schemes, it does not require a one-to-one relation between activities and processors, thereby violating the original concepts as introduced by Ousterhout for coscheduling. The term "gang scheduling" has nevertheless been applied to this system as well. Despite the identical nomenclature, we shall reserve the terms gang scheduling and coscheduling for policies where the number of processors used is equal to the number of activities. Finally, we note that although Ghosal et al. [14] suggests multiprocessor scheduling algorithms based on a concept they call the "processor working set", their work is totally unrelated to Ousterhout's coscheduling definitions.

2 Runtime Identification of Activity Working Sets

This section describes how our system identifies activity working sets and how they are scheduled. The particular scheduling algorithm clearly effects our experimental results, however, there are many scheduling schemes that can be used for co- and gang scheduling and are compatible with our run-time identification algorithm.

2.1 Preliminaries

There are several technical terms and basic assumptions that need to be defined and specified.

Multi-context-switching. Coscheduling requires the system to support *multi-context-switching*, i.e. simultaneous context-switches on multiple processors. This can be coordinated by a centralized controller that instructs all the processors to switch at the same time, or else it can be done by a distributed coordination algorithm. The choice is independent of our results and to simplify matters, we therefore assume that the scheduling is coordinated by a central controller.

Scheduling Slots and Rounds. The time between consecutive multi-context-switches is called a *scheduling slot*. The duration of a scheduling slot is *a priori* assumed to be the standard scheduling time quantum, but it can be shorter if all the scheduled activities block or terminate. A *scheduling round* is the time period in which all the ready activities are scheduled; it is a sequence of scheduling slots.

Communication Objects. Activities can interact in many ways and various mechanisms can be used to implement interactions between activities. We assume that the interactions are mediated by *communication objects* (CO) that are known to the system, and do not change[1]. In some environments, this assumption is directly valid. For example, the named channels used for message passing in Occam [18], and practically all proposed notations for multiparty interactions, e.g. scripts [13] and communicators [7], are objects known to the system. In systems that provide group communication primitives such as broadcast, multicast, and barrier synchronization, the system calls used to implement these services may be used to identify the interactions (but note that operations in distinct groups must be distinguished) [25]. In systems that only provide basic point-to-point message passing, each sender-receiver pair identifies an interaction.

The identification of working sets is not based directly on a single interaction, e.g. a specific instance of a collective communication operation. Rather, it is based on *interaction schemata* with the COs. An interaction schemata is simply the set of participating activities.

[1]Clearly, a generalization of our work would be to also automatically identify the communication objects, or rather, to identify those that should be used to drive the coscheduling.

Thus different interactions in the same set, e.g. a broadcast and a barrier, are united and represented by a single schemata. In the sequel, "interactions" and COs should be understood as referring to such schemata.

In shared memory systems, activities interact typically through a side effect of access to shared data structures. Such data structures can therefore be identified as COs and used to identify interactions. However, this requires the compiler to generate additional code to monitor the access rates to the different memory objects. It also requires more complex algorithms for identifying working sets, because the set of activities that access the shared data can change dynamically.

2.2 How Interactions Define Working Sets

The COs and the pattern of their use by the activities (the interactions of the activities) will be the major indicator to define activity working sets. Interactions between activities imply synchronization constraints. This is obvious in the case where the interaction itself is a synchronization point, e.g. a barrier synchronization. It is also true in seemingly asynchronous one-way interactions, such as the transfer of data between a producer and a consumer, due to the necessarily bounded size of buffers used to mediate the transfer.

Synchronization constraints imply that in some situations one activity will have to wait for another. Waiting obviously wastes processor cycles. However, the alternative, i.e. blocking, must deschedule the waiting activity and schedule another in its place, a process that might be even more costly: it suffers the context switch overhead, forfeits the activity's cache state, and might cause additional synchronization problems down the road as other activities are obliged to wait for this one. The choice between busy waiting and blocking depends on the granularity of interactions: if the time between interactions is small, so is the expected waiting time, and waiting is better than blocking. Coscheduling is specifically targeted to support such fine-grain interactions [27, 10].

An immediate consequence is that coarse-grain interactions do not require coscheduling. For example, if the interval between successive interactions is longer than the scheduling time quantum, the price for blocking is paid anyway. We therefore start by defining a threshold value, θ, for the interaction rate. θ is expressed in interactions per time quantum; the higher it is, the more fine-grained the interactions, with $\theta = 1$ denoting a rate of once per scheduling quantum. We shall use $\theta = 2$ in the experiments below. Interactions that are performed at a rate lower than θ do not suggest that the associated activities interact in a significant way.

Interactions that are performed at a high rate define an "interacts with" relation on activities. The interpretation of "a high rate" deserves some elaboration as it has several possible definitions. One is that the interaction rate can be taken as the number of times the interaction is performed, divided by the "wallclock time" since execution began. But wallclock time does not reflect the importance of this interaction to making progress by the participating activities. The activities could be held up while other unrelated activities execute for arbitrary periods of time. It is therefore better to use the time accountable to useful computation between interactions. Unfortunately, different participating activities might do different amounts of useful computation. In our algorithm, we use the *slowest* activity to set the rate for the whole set.

Denote the "interacts with" relation by $\leftrightarrow$. Obviously, $\leftrightarrow$ is symmetric. It is important to note that $\leftrightarrow$ is also transitive: if $A_1 \leftrightarrow A_2$ and also $A_1 \leftrightarrow A_3$, than activity A_1 effectively mediates between A_2 and A_3, even if no data is actually forwarded. Coscheduling the set $\{A_1, A_2\}$ is not enough, because A_1 will soon after block trying to interact with A_3, and subsequently A_2 will block trying to interact with A_1. Thus the working set is defined by the *transitive closure* of the "interacts with" relation. In this example, the working set is $\{A_1, A_2, A_3\}$. In general, of course, the activity working set need not include all the activities in an application.

We note in passing that once sets of activities are considered, it might be appropriate to change the definition of interaction rates. Consider the following example. The activities in the set $\{A_i\}_{i \in I}$ interact with each other at a high rate. So do the activities in the set $\{B_i\}_{i \in I}$. Between the two sets, for each i the activities A_i and B_i interact at a low rate. Depending on the pattern in which the interactions are performed, the effective rate of interactions between the sets $\{A_i\}$ and $\{B_i\}$ can be as low as that between any single pair A_i and B_i, or it can be as high as the sum of the rates among all such pairs. We leave such issues for future research.

2.3 The Algorithm for Activity Working Set Identification

The main data structure used by the algorithm is the matrix of usage rates. Columns in this matrix correspond to activities, and rows to COs. The entry in row i and column j, denoted U_{ij}, gives an approximation of the rate in which activity A_j takes part in the interaction associated with CO_i. A zero entry indicates that the activity does not participate in the interaction via CO_i. For a group collective communication CO, all the entries of the activities in the group will be non-zero, although the entries need not be identical since they are rates and not counts.

The data in the usage rate matrix is derived from two other data structures: the usage count vector and the runtimes vector (Fig. 2). Entries in the usage count vector count the number of times each interaction has been performed. The runtimes vector contains the cumulative effective runtime of each activity (i.e. excluding busy waiting time). If an activity participates in an interaction, the usage rate is obtained by dividing the usage count by the activity's runtime; otherwise it is zero. To prevent overflow and adjust to changing interaction patterns, both the counters and the elapsed time are periodically divided by two (i.e. use "exponential aging").

While these three data structures may appear to be centralized and cause much contention, this is in fact not so. For each activity, the corresponding runtime entry and matrix column are only updated by the processor on which the activity executes. Therefore no two elements in either data structure are ever write-shared in the same scheduling slot, and no locks are needed. Each entry in the usage count vector, similarly, is updated by only a single

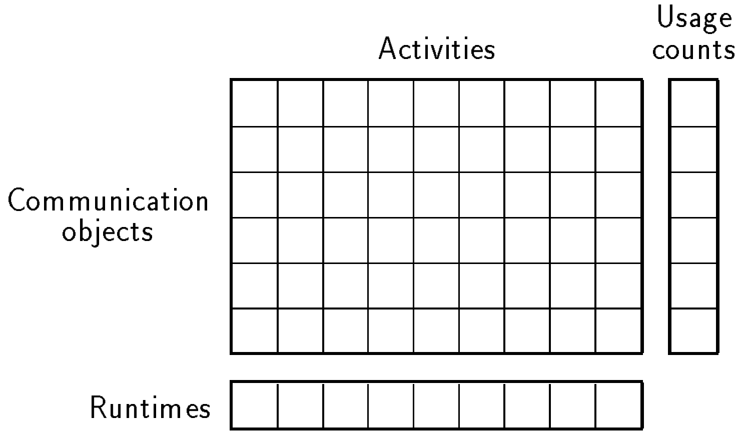


Figure 2: *The main data structures. The row i, column j entry in the matrix approximates the rate at which activity j participates in interactions using CO_i. The* `Usage Count` *column vector records the number of times each CO is used for an interaction and the row vector* `Runtimes` *records the accumulated compute time of each activity.*

designated activity each time the interaction is performed. Thus these data structures can be implemented in a highly distributed fashion.

The algorithm is based on the definition of a preliminary working set of activities for each CO. These preliminary working sets are then combined when taking the transitive closure of interactions through this and other COs. For a given CO_i, the preliminary working set is defined to be the set of all the activities that participate in the interaction, qualified by the requirement that they all interact at a sufficiently high rate. This is verified by making sure that the rate of slowest of the participating activities is at least θ. Formally, this is expressed as

$$WS_i = \begin{cases} \emptyset & \text{if } \exists j : 0 < U_{ij} < \theta \\ \{A_j \mid U_{ij} \geq \theta\} & \text{otherwise} \end{cases}$$

The algorithm is presented in Fig. 3. In each scheduling round, it loops on all COs and creates working sets around them by finding the transitive closure of the interacting activities. This is the loop with index i, where n is the number of COs. The working set is generated in the variable WS. Note that the step of finding the transitive closure is expressed as an inner loop that repeats an operation until convergence. It is not sufficient to loop once on all COs from $i+1$ to n. The working sets are combined using next-fit bin packing subject to the available number of processors. This is done in the variable S. As each slot is filled the activities in it are coscheduled. Alternative scheduling is used for activities that were not assigned to any working set.

Careful bookkeeping is required to ensure that all activities are scheduled correctly in each round. This is done in two levels. First, activities are identified as belonging to some preliminary working set, or not belonging to any working set. Those that belong to some preliminary working set will be coscheduled as part of the transitive closure of

```
set all COs as unmarked
set S = ∅

for i = 0 to n
      if CO_i is marked or WS_i is empty then skip it
      set WS = WS_i
      repeat until convergence
            if there exists an unmarked CO_k such that WS ∩ WS_k ≠ ∅ then
                  mark CO_k
                  WS = WS ∪ WS_k
      if |S| + |WS| > P then
            coschedule activities in S
            S = ∅
      S = S ∪ WS

coschedule activities in S

schedule activities not in any WS_i by dividing them into groups
of P and scheduling each group in successive scheduling slots
(this is the alternative scheduling)
```

Figure 3: *The algorithm for one round of runtime identification and scheduling of activity working sets. n is the number of COs. Indentation is used to specify the scope of* `for`, `if`, *and* `repeat` *constructs.*

the corresponding interaction, while those that do not are left for alternative scheduling. Second, COs that are used in the definition of a working set are marked as part of finding the transitive closure. They are then skipped in subsequent iterations of the loop over all COs, thus preventing the possibility of finding (and scheduling) the same working set again starting from a different CO.

Two simple optimizations are possible to reduce the fragmentation. First, the working sets can be sorted in decreasing size, leading to first-fit-decreasing bin packing, which is more space-efficient [5]. Second, alternative scheduling can be overlapped with coscheduling to the degree possible by PEs that are left over in the coscheduling slots. That is, whenever the activities in S are to be scheduled and $|S| < P$, then $P - |S|$ activities that were not assigned to any preliminary working sets are scheduled also.

Note that there is an implicit assumption that the size of any given working set is not larger than P. If it is, then the working set has to be broken up into smaller sets that can be coscheduled. A simple approach is to forgo the step in which the transitive closure is computed, and just schedule preliminary working sets based on single COs. This can be optimized by computing part of the closure based on usage rates, so as to cluster those activities that interact with each other most frequently. We leave the details for future research.

Finally, we note that the main loop in the algorithm scans the COs, rather than the activities. This is based on the assumption that in systems where multiparty interactions such as barrier synchronization and collective communication are provided, the number of COs can be expected to be smaller than the number of activities. In systems where only pairwise interactions are provided, on the other hand, the number of interactions may far exceed the number of activities. It is then better to reorganize the algorithm to scan the activities instead.

3 Experimental Results

In order to evaluate the performance of the algorithm for runtime identification of activity working sets, the behavior of multiprogrammed multiprocessors under the proposed scheduling discipline was simulated. This section describes the simulator, the associated experiments, and the results.

3.1 The Simulator

The simulator was used for several tasks. It implemented the activity working set identification algorithm described in the previous section. It also simulated several scheduling strategies to enable a comparative evaluation of our scheme. We first describe the three scheduling approaches and then describe the workload and simulation parameters.

3.1.1 Three Scheduling Schemes

The simulator accepts a high-level description of the workload as its input and simulates the execution of this workload for three approaches:

r coscheduling based on runtime identification of activity working sets

m coscheduling based on manually-identified working sets

u uncoordinated scheduling where all the processors share a global activity queue

In both coscheduling algorithms, gangs are grouped together using the next-fit bin packing algorithm as long as the total does not exceed the number of processors, and are then scheduled to run. This is essentially a dynamic version of Ousterhout's matrix algorithm [27], as the mapping of activities to processors may change in each round. Coscheduled activities that arrive at an interaction retain their processor and busy wait. This is justified because activities in gangs are believed to interact frequently, so the waiting time is expected to be short. The coscheduled activities continue to run until either the scheduling time quantum expires or all of them are busy waiting for an interaction, at which time a multi-context switch is performed. After all the gangs have been coscheduled, any remaining activities that have not been scheduled yet in this round are scheduled in an uncoordinated manner. When these activities arrive at an interaction, they block rather than busy waiting. This is justified because as far as the system knows, these activities do not interact frequently, and therefore there is no ground to believe that they will only wait a short while.

The uncoordinated scheduling algorithm, which is used for comparison, implements round-robin scheduling from a shared queue of activities. Only activities that are ready to run are placed in the queue. Each activity is scheduled independently, and receives a full scheduling time quantum for execution. Competitive two-phase blocking is used for the interactions [27, 19]. Thus when an activity waits for an interaction, it first busy-waits for a duration equal to the context-switch overhead, and then it suspends execution and yields its processor. If the activity's time quantum expires while it is busy-waiting, it is suspended at that time. When the awaited interaction is finally performed by another activity, the suspended activity is resumed and immediately returned to the ready queue.

For all three scheduling algorithms, the simulation is event-based. The events represent state changes of the processors. For the coscheduling algorithms, the main events are the end of the think time of a running activity and the end of a scheduling quantum; busy waiting activities may wait indefinitely, so they are not kept in the event queue. Activities using alternative scheduling in the coscheduling algorithm, and all activities in the uncoordinated scheduling algorithm, require additional event types. These include the end of a think time, the end of a busy waiting period, the end of a scheduling quantum, and the end of a context switch.

3.1.2 The Workload Specification

The workload is described by a set of activities. 100 were used in the experiments. Each activity has a synthetic program which lists the interactions in which it participates. It is assumed that this sequence of interactions is interleaved with periods of local work, i.e. think times, and the whole interleaved sequence is repeated indefinitely. The think times between interactions are picked at random from a uniform distribution around a mean value. Both the mean value and the range of the distribution are input parameters. Typical values used in the experiments for the mean are 0.25, 1.0, and 4.0 time units, where the time unit is the context-switch overhead. These values correspond to fine-grain, medium, and coarse-grain interactions. The range used was $[0.8m, 1.2m]$ where m is the mean. The same random number generators are used for the same activities for the three scheduling schemes that were compared, guaranteeing that the input for the three is indeed identical.

Two basic interaction patterns were modeled in the simulations and each of these was further subdivided to give a total of five different patterns ranging from clearly identified independent gangs to almost no gang structure. In the first the activities were actually partitioned into gangs, and the gang size was equal to the number of processors. This pattern had three variations:

gangs The activities were divided into independent gangs, each with its own CO. All interactions always involved all the activities in the gang.

grid The activities were arranged in a grid. Each row of activities had a CO associated with it, and each column also had a CO associated with it. Each activity thus participated in interactions using two different COs, and involving distinct sets of interaction partners. The interactions within rows were executed five times more often than those within columns.

rings The activities were divided into independent gangs, but within each gang, the activities were arranged in a ring. Each activity interacted directly only with its two neighbors in the ring. Therefore the full gangs can only be identified by finding the transitive closure of the interactions.

In the second basic pattern all activities were organized in a single ring, rather than first dividing them into gangs. This pattern had two variations:

ring same each activity alternated between interacting with its two neighbors, and all interactions were performed equally often.

ring pairs activities were arranged in pairs, i.e. the first two activities in the ring are a pair, the next two are another pair, and so on. Interactions within each pair were five times more common than interactions across pairs.

In all patterns, the numbering of activities and interactions was randomized. For example, and first gang in the gangs workload included activities $\{59, 1, 30, 4, 94, 47, 67, 52, 85, 41\}$.

The system was represented by the following simulation parameters: the number of processors (10 in the experiments), the context switch overhead (which serves as the unit of time), and the scheduling time quantum (20 times the context switch overhead). The runtime identification algorithm used a threshold of $\theta = 2$.

3.2 The Results

Each experiment consisted of a sequence of 11 batches of 240 interaction completions for each activity in the system. The first batch was then discarded to account for "warm-up" of the simulation, and the results of the other ten were averaged. The reported results are these averages of the time for a batch; this can be regarded as the average response time for an application that performs 240 interactions. The measurements for different batches typically differ by less than 3%, so confidence intervals are too small to be shown.

3.2.1 Effect of Granularity and Interaction Pattern

The main support for our approach can be found in Fig. 4. The simulated time required for each of the five interaction patterns simulated under each of the three granularities are presented in this figure. Moreover, the figure also displays the three main components of the total run times. The first component is the actual computation time, which is the sum of the think times in 240 interactions. The second component is the busy waiting time, and the third is the time spent context switching. The context switching time was calculated by dividing the total time by 10 (the average load), and subtracting the computation time and the busy waiting time.

The main observation from these simulations is that the performance of coscheduling based on runtime identification of activity working sets is in many cases identical to that of coscheduling based on manual identification by the programmer. In some cases it is only marginally inferior. In addition, coscheduling — either by automatic or manual gang identification — is uniformly better than uncoordinated scheduling for all workloads that are indeed based on gangs of interacting activities. Coscheduling is inferior to uncoordinated scheduling only for workloads with interaction patterns that do not favor any clustering of activities, most notably the `ring` of activities with same interaction rates and medium- or coarse-grain interactions.

With coscheduling, the absolute values of the context switching and busy waiting overheads depend on the workload and the granularity. For the workloads that have strong clustering, most notably the `gangs` and `rings` workloads, the ratio of the overhead to the think time is fixed for all granularities. This leads to uniformly good performance for these workloads. In workloads where such clustering does not exist, like the `ring` with coarse-grain interactions, coscheduling creates false dependencies that increase the relative overhead. When activities form a ring, all interactions are among pairs. These pairs are matched arbitrarily by the next-fit packing to utilize all the processors, and then they are coscheduled together. Pairs that block sooner than others must nevertheless wait for those that manage to do more work. This can only be solved by partitioning the machine and doing gang scheduling independently on the partitions, as suggested in the Distributed Hierarchical Control scheme [9].

With uncoordinated scheduling, the overhead is constant regardless of the granularity of the interactions. For fine-grain interactions, the relative overhead is very large, leading to poor performance. For coarse-grain interactions, the overhead is acceptable and overall performance is similar to that of coscheduling. This supports our assertion that coscheduling is important for fine-grained interaction. The graphs show that the overhead in the `gang` and `grid` workloads is double that in the `ring` patterns. The reason is that in the `gang` and `grid` workloads each interaction involves 10 activities. Therefore most activities only manage one interaction before they block. In the `ring` patterns, interactions involve pairs of activities, so each activity typically performs two successive interactions before blocking.

Another interesting conclusion that can be derived from the graphs concerns the use of two-phase blocking with uncoordinated scheduling. All the graphs for uncoordinated scheduling show that an equal amount of time is spent on busy waiting and on context switching. This implies that the busy waiting practically never paid off, and it would have been better to suspend the activity immediately when it reached each interaction. Checking more carefully, we found that the highest score was a 6.5% success rate (for the ring of pairs workload, with fine- and medium-grain interactions), with all other scores at 4.1% or less. The reason for this result is that in a loaded system, there is very little chance that the

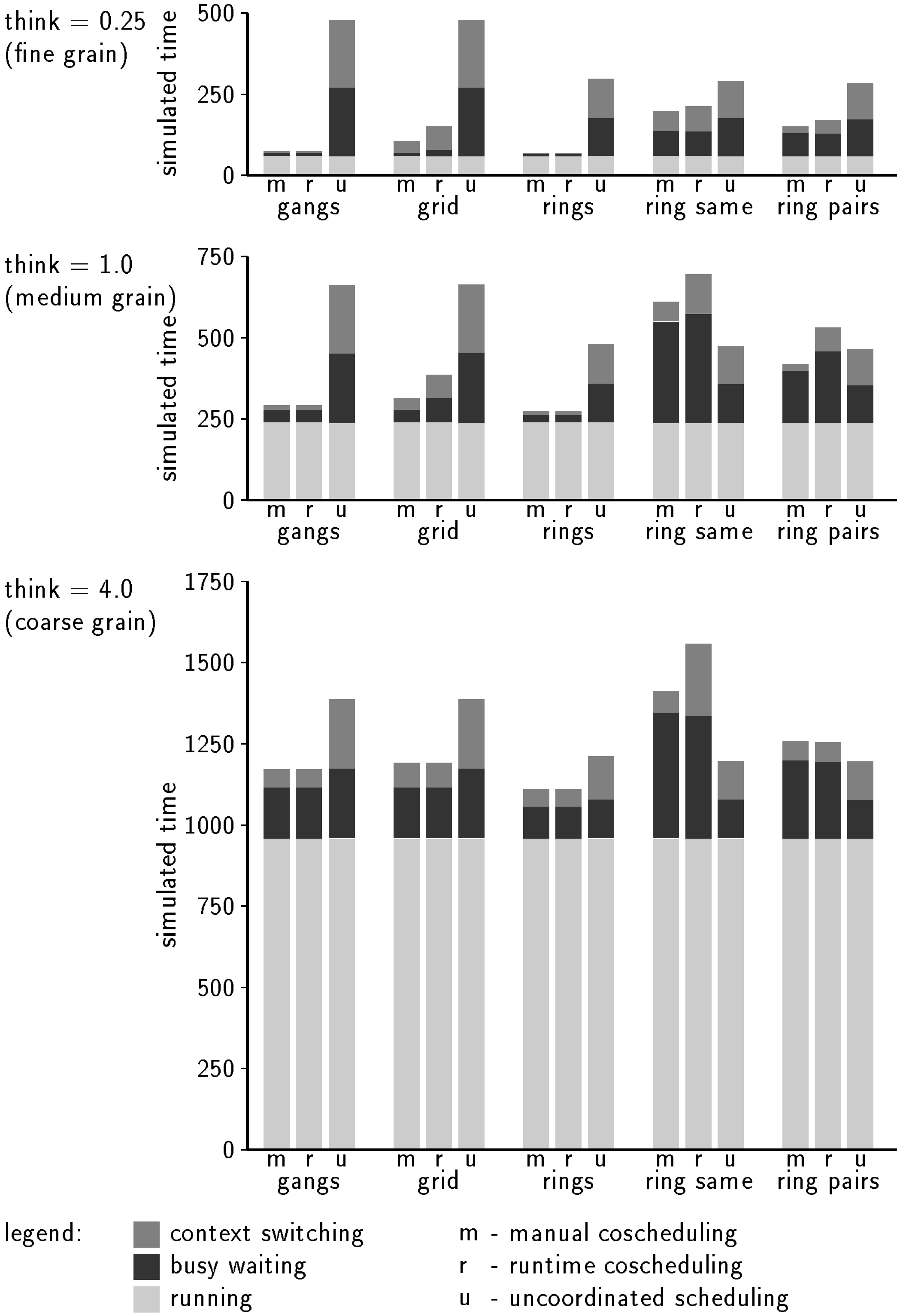


Figure 4: *Average run times and their breakdown for the different interaction patterns and granularities. The performance of coscheduling based on runtime identification of activity working sets is in many cases identical to that of coscheduling based on manual identification by the programmer, and both are superior to uncoordinated scheduling for workloads with clustering of activities.*

activity you are waiting for will actually be running and reach the interaction within the busy-waiting window. Thus waiting for it is a waste of time.

It is not necessary to re-run the experiments to see how an implementation that blocks immediately would perform, because all the information is already contained in the graphs of Fig. 4: simply remove the top segment from each bar representing uncoordinated scheduling. For workloads with gangs and fine grain interactions, the difference in performance would still be significant, in favor of coscheduling. For coarse-grain interactions and the `ring` workloads, coscheduling loses any benefits it had over uncoordinated scheduling.

3.2.2 The Impact of the Value of θ

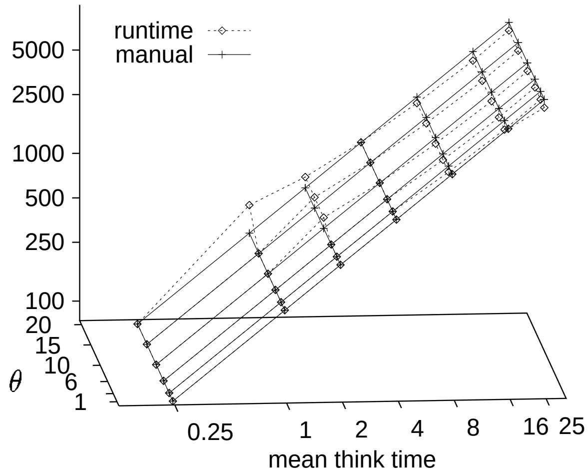
The threshold value, θ, that determines when an interaction rate is considered significant, has an affect on the runtime identification of activity working sets. The experiments reported above used a threshold value of $\theta = 2$. Here we investigate the effect of θ on the performance of this coscheduling scheme. The workload of 10 independent gangs of 10 activities each, running on 10 processors, is used as a test case.

A number of combinations of θ and the mean think time were measured. The results are shown in Fig. 5, where the performance of runtime identification is compared with that of coscheduling based on manual identification of gangs. Note that the mean think time and the simulation time in part (a) are shown with a logarithmic scale.

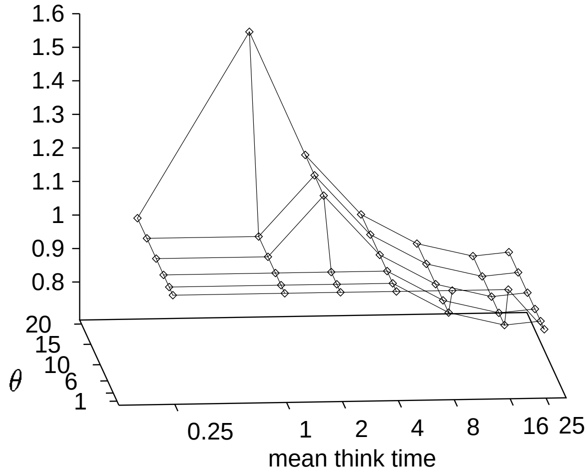
Denote the mean think time by τ and the scheduling time quantum by q (in all the experiments, $q = 20$). The results indicate that the parameter space is divided into two areas, where the dividing line is the hyperbola $\theta = q/\tau$ (Fig. 6). In the area where $q/\tau > \theta$ (both τ and θ are small) the granularity is fine enough so that the number of interactions that are completed on average in a scheduling quantum is larger than θ. Consequently, the runtime identification algorithm succeeds in correctly recognizing the activities in each gang and the performance is identical to that of coscheduling based on manual identification.

If, on the other hand, $q/\tau < \theta$, then gangs are not recognized at runtime, and the performance of the two schemes differs. With manual identification, the runtime is linearly proportional to the *maximal* think time (given that there is some variability among activities in each gang), because all the activities in the gang have to wait for the last one. Obviously, it does not depend on θ. With runtime identification the situation is different. Since the gangs are not recognized, all the activities are scheduled independently and blocking is used to synchronize interactions. The expected time for each interaction is therefore the *mean* think time plus the context switch overhead, just like for uncoordinated scheduling. If the context switch overhead is larger than the variability in think times (as is typical in fine-grain workloads), the runtime algorithm suffers extra overhead. This can be seen for $\theta > 6$ and think times less than 4 in Fig. 5 (b). But if the context switch overhead is smaller than the variability in think times (typically in coarse-grain workloads), independent scheduling can free processors and use them for other activities. Thus the runtime algorithm achieves superior performance for think times greater than 4 in the figure.

For the simple case of this specific workload model, the maximal think time is $\tau\left(1 + \frac{n-1}{n+1}v\right)$,



(a)



(b)

Figure 5: *Dependence of runtime on the combination of* θ *and the granularity, for 10 independent gangs of 10 activities. (a) absolute values, (b) ratio of runtime identification to manual identification of gangs. When the think time and* θ *are small, gangs are identified correctly at runtime. When think times are large, it is actually beneficial not to identify gangs but rather to use uncoordinated scheduling.*

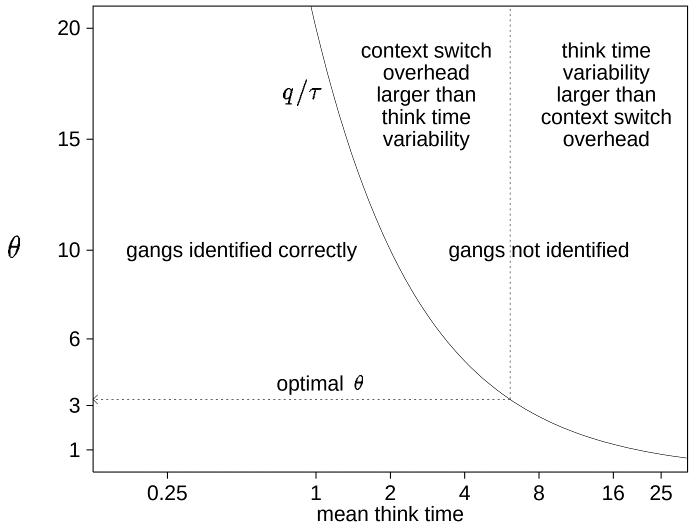


Figure 6: *Phase diagram showing the effect of different combinations of θ and the mean think time. The optimal value of θ is found by the intersection of q/τ and the think time for which the variability balances the context switch overhead.*

where n is the number of activities in each gang (10) and v is the variability in think times (0.2 in our experiments). The think time where the variability exactly counteracts the overhead of a context switch c is given by $\tau\frac{n-1}{n+1}v = c$. The value of θ should be chosen such that for smaller τ gangs are identified, but for larger τ activities are scheduled independently. Using $q/\tau = \theta$, this is given by

$$\theta = \frac{(n-1)qv}{(n+1)c}$$

For the parameters used in the experiment, the optimal value is $\theta = 3.27$ (Fig. 6).

Such an analysis cannot be carried out in the general case, because the workload cannot be characterized by a small set of parameters. However, this example provides insight indicating that θ should be a small number larger than 1, e.g. in the range of 1 to 5.

3.2.3 Sensitivity to Simulation Parameters

The simulations were based on a parametric model of the workload, with parameters such as the granularity of interaction, the variability in think times, the number of processors, and the total number of activities. The results shown in Figs. 4 through 6 are explicit in terms of dependence on the granularity and on θ. Here we shall consider the other parameters.

The dependence on the number of processors is plotted in Fig. 7, for the gangs interaction pattern and medium granularity ($\tau = 1$). With uncoordinated scheduling, the runtime decreases smoothly as processors are added. With coscheduling, we see a jump in performance each time a new gang fits in fully. Adding processors that are not enough for a full gang does not help, and these processors are essentially idle. This holds for both coscheduling

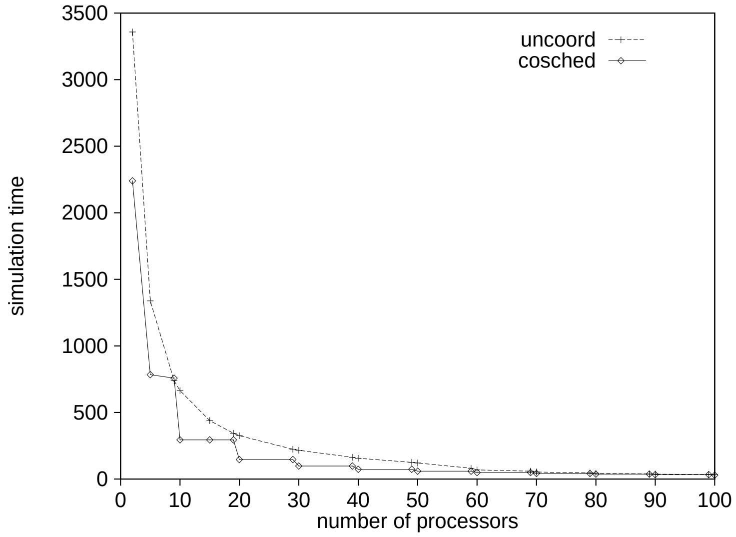


Figure 7: *Dependence of run time on the number of processors. The workload is 10 independent gangs of 10 activities each. With uncoordinated scheduling, the runtime decreases smoothly as processors are added. With coscheduling, we see a jump in performance each time a new gang fits in fully. Adding processors that are not enough for a full gang does not help, and these processors are essentially idle.*

algorithms, and the measured results are practically identical. The measurements shown are for runtime recognition of gangs.

These results are even more prominently displayed when we look at the speedup rather than at the run time (Fig. 8). For small numbers of processors (up to 50), uncoordinated scheduling consistently achieves about 40% of the ideal resource utilization. When the number of processors is above 50, the performance of uncoordinated scheduling improves significantly. The reason is that the total number of activities is fixed at 100, so in this range the relative load on the system becomes very low. Uncoordinated scheduling here is like self scheduling: processors that finish early take a new activity, and activities hardly have to wait because there are very many processors. In this range, two-phase blocking also becomes beneficial: the busy waiting success rate is 21.1% for 50 processors, jumps up to 66.2% for 60, and rises to a peak of 89.4% for 100 processors.

Coscheduling achieves about 80% utilization each time another gang fits in with no processors left over. This is the maximum possible given the 20% variability in think times used in these experiments. Even when the number of processors does not divide the gang size evenly, the performance of coscheduling is always superior to that of uncoordinated scheduling. Similar results were also measured for coscheduling at other granularities ($\tau =$ 0.25 and $\tau = 4.0$). However, uncoordinated scheduling suffered from much lower performance in the fine-grain case ($\tau = 0.25$), whereas its performance was essentially the same as that of coscheduling in the coarse-grain case ($\tau = 4.0$).

The think times for different activities in each gang come from a uniform distribution

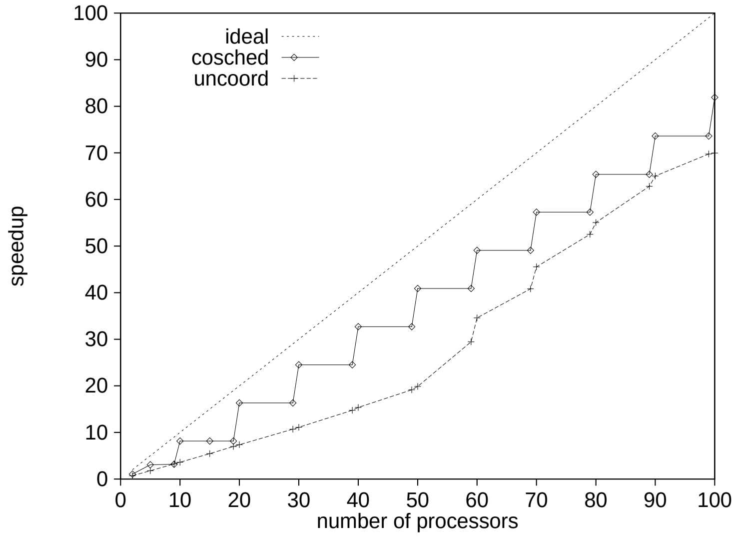


Figure 8: *Dependence of speedup on the number of processors, based on the data in Fig. 7, and an ideal execution time of 2400 time units on a single processor. With coscheduling, a utilization of 80% is achieved each time enough processors are added to accommodate another full gang. Uncoordinated scheduling only achieves good utilization above 60 processors, because then the effective load becomes very low.*

around the mean think time. The variability in think times, i.e. the width of this distribution, determines the expected wait times for the interactions. Thus a bigger variability implies longer wait times. This favors uncoordinated scheduling, because it is more flexible in its use of processors: a processor running an activity that reaches an interaction early can switch to something else rather than waiting for the others to catch up. With coscheduling, the early activities must busy wait. If the variability in think times is longer than the context switch overhead, this ends up wasting more cycles.

The above observations are quantified in Fig. 9 where ten independent gangs are executed on 10 processors, with different think-time distributions. Under uncoordinated scheduling, the run-time does not depend on the variability of the think time at all: it is just the sum of the *average* think time plus twice the context switch overhead for each interaction (this includes busy waiting time due to the use of two-phase blocking). Under coscheduling, the run-time of an activity set (and hence the time of all the activities in the system) is proportional to the *maximum* think time, and therefore also to the variability in think times. In this specific workload, the maximal think time is linearly proportional to the variability, and this is evident in the measured results. For mean think times of 0.25 and 1.0, the performance of coscheduling is nevertheless superior to that of uncoordinated scheduling for all measurements, because even the maximal variability is less than the context switch overhead. For a mean think time of 4.0 there is a crossover, and uncoordinated scheduling is better for workloads with a high variability. For a mean think time of 16.0 the runtime coscheduling algorithm does not recognize gangs, because the interactions occur at a rate lower than θ.

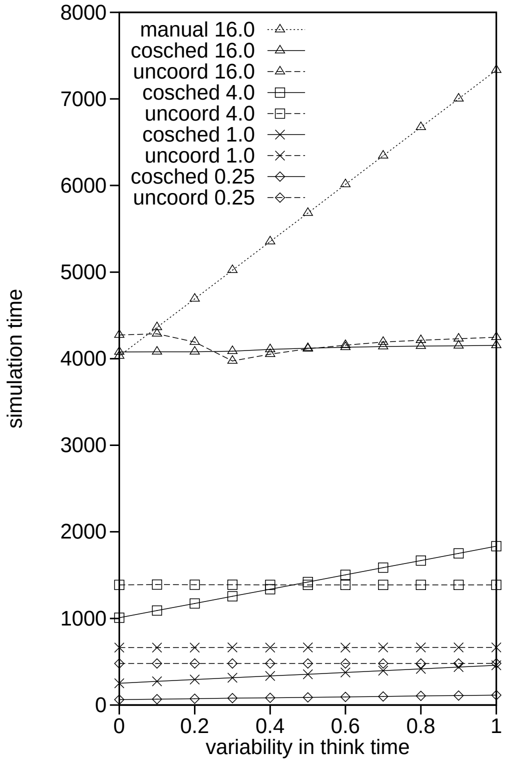


Figure 9: *Dependence of run time on variability in think time, for 10 independent gangs of 10 activities with various granularities of interactions. Under uncoordinated scheduling, the run-time does not depend on the variability of the think time at all: it is just the sum of the average think time plus twice the context switch overhead for each interaction. Under coscheduling, the run-time of an activity set is proportional to the maximum think time, and therefore also to the variability in think times.*

Therefore runtime coscheduling achieves the same performance as uncoordinated scheduling, while oblivious coscheduling based on manual identification of gangs suffers from the high variability.

4 Conclusions

The research reported in this paper is part of a growing endeavor to automate parallel programming tasks, by moving responsibility from the programmer to the runtime system. The idea is to try and find simple, efficient, and general run-time support mechanisms, so that the user need not do everything manually. Coscheduling was originally proposed as a way to reduce the burden of implementing efficient interactions among activities. We have now taken another step, by showing how the interacting activities may be identified. Our experimental results are encouraging, and indicate that in many cases on-line identification can lead to the same performance as using information supplied by the programmer. In some cases it can even be superior, because it can also identify situations in which the activities actually do not interact at a high rate.

The described algorithm comprises a first attempt to solve a difficult and important problem, related to the growing interest and belief in coscheduling. Naturally, there is still much to be done. We have identified a number of issues that require further research, such as recognizing cases where interactions among sets of gangs should be summed up to obtain the cumulative interaction rate, and how to break up gangs that are larger than the number of processors. A long-term goal is to actually implement coscheduling based on runtime identification of working sets in a real system, and test its effectiveness with a real workload. Another direction for research is investigating the degree to which activity working sets can be identified by compile-time analysis, and comparing it with runtime identification.

Acknowledgements

We wish to thank the anonymous referees who made valuable suggestions that improved both the content and the presentation of this paper.

References

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

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

[3] R. H. Campbell, N. Islam, and P. Madany, "*Choices, frameworks and refinement*". *Computing Systems* **5(5)**, pp. 217–257, Summer 1992.

[4] E. M. Chaves Filho and V. C. Barbosa, "*Time sharing in hypercube multiprocessors*". In 4th *IEEE Symp. Parallel & Distributed Processing*, pp. 354–359, Dec 1992.

[5] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson, "*Approximation algorithms for bin-packing — an updated survey*". In *Algorithm Design for Computer Systems Design*, G. Ausiello, M. Lucertini, and P. Serafini (eds.), pp. 49–106, Springer-Verlag, 1984.

[6] J. Edler, A. Gottlieb, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, M. Snir, P. J. Teller, and J. Wilson, "*Issues related to MIMD shared-memory computers: the NYU Ultracomputer approach*". In 12th *Ann. Intl. Symp. Computer Architecture Conf. Proc.*, pp. 126–135, 1985.

[7] D. G. Feitelson, *Communicators: Object-Based Multiparty Interactions for Parallel Programming.* Technical Report 91-12, Dept. Computer Science, The Hebrew University of Jerusalem, Nov 1991.

[8] D. G. Feitelson, *A Survey of Scheduling in Multiprogrammed Parallel Systems.* Research Report RC 19790 (87657), IBM T. J. Watson Research Center, Oct 1994.

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

[10] D. G. Feitelson and L. Rudolph, "*Gang scheduling performance benefits for fine-grain synchronization*". *J. Parallel & Distributed Comput.* **16(4)**, pp. 306–318, Dec 1992.

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

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

[13] N. Francez, B. Hailpern, and G. Taubenfeld, "*Script: a communication abstraction mechanism*". *Sci. Comput. Programming* **6(1)**, pp. 35–88, Jan 1986.

[14] D. Ghosal, G. Serazzi, and S. K. Tripathi, "*The processor working set and its use in scheduling multiprocessor systems*". *IEEE Trans. Softw. Eng.* **17(5)**, pp. 443–453, May 1991.

[15] B. C. Gorda and E. D. Brooks III, *Gang Scheduling a Parallel Machine.* Technical Report UCRL-JC-107020, Lawrence Livermore National Laboratory, Dec 1991.

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

[17] S. F. Hummel and E. Schonberg, "*Low-overhead scheduling of nested parallelism*". *IBM J. Res. Dev.* **35(5/6)**, pp. 743–765, Sep/Nov 1991.

[18] INMOS Ltd., *Occam Programming Manual.* Prentice-Hall, 1984.

[19] A. R. Karlin, K. Li, M. S. Manasse, and S. Owicki, "*Empirical studies of competitive spinning for a shared-memory multiprocessor*". In 13th *Symp. Operating Systems Principles*, pp. 41–55, Oct 1991.

[20] S. Krakowiak, *Principles of Operating Systems.* MIT Press, 1988.

[21] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong, S-W. Yang, and R. Zak, "*The network architecture of the Connection Machine CM-5*". In 4th *Symp. Parallel Algorithms & Architectures*, pp. 272–285, Jun 1992.

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

[23] C. McCann, R. Vaswani, and J. Zahorjan, "*A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors*". *ACM Trans. Comput. Syst.* **11(2)**, pp. 146–178, May 1993.

[24] C. McCann and J. Zahorjan, "*Processor allocation policies for message passing parallel computers*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 19–32, May 1994.

[25] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard.* May 1994.

[26] V. K. Naik, S. K. Setia, and M. S. Squillante, "*Scheduling of large scientific applications on distributed memory multiprocessor systems*". In 6th *SIAM Conf. Parallel Processing for Scientific Computing*, vol. II, pp. 913–922, Mar 1993.

[27] J. K. Ousterhout, "*Scheduling techniques for concurrent systems*". In 3rd *Intl. Conf. Distributed Comput. Syst.*, pp. 22–30, Oct 1982.

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

[29] P. Steiner, "*Extending multiprogramming to a DMPP*". *Future Generation Comput. Syst.* **8(1-3)**, pp. 93–109, Jul 1992.

[30] H. Sullivan, T. R. Bashkow, and D. Klappholz, "*A large scale, homogeneous, fully distributed parallel machine, II*". In 4th *Ann. Intl. Symp. Computer Architecture Conf. Proc.*, pp. 118–124, Mar 1977.

[31] J. A. Test, "*Multi-processor management in the Concentrix operating system*". In *Proc. Winter USENIX Technical Conf.*, pp. 173–182, Jan 1986.

[32] A. Tucker and A. Gupta, "*Process control and scheduling issues for multiprogrammed shared-memory multiprocessors*". In 12th *Symp. Operating Systems Principles*, pp. 159–166, Dec 1989.

[33] D. L. Tuomenoksa and H. J. Siegel, "*Task scheduling on the PASM parallel processing system*". *IEEE Trans. Softw. Eng.* **SE-11(2)**, pp. 145–157, Feb 1985.

[34] A. M. van Tilborg and L. D. Wittie, "*Wave scheduling — decentralized scheduling of task forces in multicomputers*". *IEEE Trans. Comput.* **C-33(9)**, pp. 835–844, Sep 1984.

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