

# Coscheduling through Synchronized Scheduling Servers— A Prototype and Experiments

Holger Karl\*

Humboldt-University of Berlin  
Institut für Informatik  
Unter den Linden 6  
10099 Berlin, Germany  
karl@informatik.hu-berlin.de

**Abstract.** Predictable network computing still involves a number of open questions. One such question is providing a controlled amount of CPU time to distributed processes. Mechanisms to control the CPU share given to a single process have been proposed before. Directly applying this work to distributed programs leads to unacceptable performance, since the execution of processes on distributed machines is not coordinated in time. This paper discusses how coscheduling can be achieved with share-controlling scheduling servers. The performance impact of scheduling control is evaluated for BSP-style programs. These experiments show that synchronization mechanisms are indispensable and that coscheduling can be achieved for unmodified programs, but also that a performance overhead has to be paid for the control over CPU share.

## 1 Introduction

In recent years, the potential of interconnected stand-alone workstations for parallel and distributed applications has been widely discussed in research literature (cp. [1] for a classic text). These so-called *networks of workstations* (NOW) exhibit other research issues than dedicated, custom-designed parallel supercomputers. One such issue is predictable runtime of parallel programs.

To achieve predictable runtime, it is necessary to control the allocation of computing resources, e.g. CPU time, of a single machine to a parallel program. Since a workstation might also be used to run interactive applications, parallel programs should not be allowed to interfere with this interactive work arbitrarily, but should leave enough free capacity for interactive users. On the other hand, it might be desirable to dedicate a certain minimum amount of resources to the parallel application. This gives rise to the notion of a contract between (one or

---

\* This research was sponsored by Deutsche Forschungsgemeinschaft (German Research Council), by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0320; and by the National Science Foundation under grant number CCR-94-11590.

several) parallel applications and interactive, local users of a machine, regimenting how resources are allocated to each program and guaranteeing minimum and maximum shares of these resources.

Enforcing such a contract could be done by the operating system, but a middleware approach (not modifying a commodity operating system) is more in accordance with the NOW ideas. Several such middleware systems (often called “scheduling servers”) have been described; this paper presents an implementation for the Linux operating system.

However, all other comparable systems are concerned with non-parallel applications running only on a single machine. The case of parallel applications executed in a distributed fashion on a NOW is more challenging. As experiments will show (cp. Section 4.3), a naïve use of distributed scheduling servers results in unacceptable performance. This is commensurate with the well-known fact that distributed scheduling and the coordination of programs can have an immense effect on program efficiency — as evidenced by work on coscheduling. Therefore there is a need to combine controlled resources with coordinated, distributed scheduling. This paper discusses how a set of scheduling servers can enforce a resource contract while at the same time providing coscheduling of parallel programs, combining guaranteed resources with good performance.

Parallel programs conforming to the bulk synchronous parallel-style (BSP, [15]) are used for evaluations. This choice is due to their close relationship with consensus-based methods which have been suggested as a basis for fault-tolerant network computing [11]. Combining these fault tolerance mechanisms with the guaranteed resource allocation for distributed programs described in this paper allows to give (probabilistic) guarantees on the program execution time, even in the presence of faults. This is a step towards predictable network computing on commercial off-the-shelf systems.

The rest of this paper is organized as follows. Section 2 describes related work in more detail, Section 3 considers limitations imposed by commodity operating systems and explains the design of a prototype, Section 4 gives some initial experimental results, Section 5 outlines possibilities for extending this work and Section 6 finally gives some conclusions.

## 2 Related work

### 2.1 Controlling CPU share

In real-time operating system, controlling CPU share is relatively straightforward (cp. e.g. [10] for real-time Mach). Other approaches target commodity off-the-shelf operating systems and try not to modify them. [13] describes a *Scheduling Server* for NeXTSTEP. This server is a program that runs with the highest available real-time scheduling priority, and cyclically lowers or raises a controlled program’s priority. The controlled program also runs with a high real-time scheduling priority, higher than normal programs, but strictly lower than the scheduling server. The scheduling server itself sleeps practically all the time

and only wakes up to schedule the controlled program. The resource share that is given to the program is tunable via the scheduling server. Similarly, the URSched system introduced in [8] uses the fixed-priority scheduling of SUN's Solaris 2.4 operating system to provide smooth video playback. URSched achieves considerably reduced jitter compared to standard time-sharing scheduling strategies.

A comparison between the user-level approaches [8, 13] and the operating system approaches [10] shows that for the price of modifications to the system kernel, more precise accounting of resource usage (in particular, processing time spent in the operating system on behalf of a process) can be achieved. User-level approaches on the other hand allow better flexibility to implement various advanced scheduling schemes on top of existing operating systems [5].

## 2.2 Coordinated scheduling

The question of coordination of processes belonging to a parallel program has been addressed by work on gang/coscheduling. The original idea of gang scheduling is to schedule distributed threads of a parallel program, running on multiple processors, at the same time [12]. This provides these threads with an environment similar to a dedicated machine and allows them to spin-wait for synchronization and/or communication. This is particularly important for fine-grained programs which would suffer considerably from the context switches entailed by blocking communication. For coarse-grained or highly imbalanced threads, blocking can be beneficial.

Classic gang scheduling research assumes that the parallel machine is dedicated to parallel programs, which basically use it in a time-sharing fashion. This is no longer true for NOWs. In particular, pure spin-waiting is no longer acceptable since this is a waste of resources. [6] and later [2] suggest to use an adaptive two-phase spin-blocking communication, which, in combination with a standard operating system's time-sharing scheduler, results in coordinated scheduling of distributed threads. In this scheme, a communication event (sending or receiving of messages) is considered as an implicit request for coscheduling between sender and receiver. They show that this implicit coscheduling is particularly beneficial for fine-grained programs. The essential property exploited in [6] here is the scheduler's property to give a priority boost to a blocked program after becoming runnable again. [14] argues along similar lines, but uses additional hardware support (a programmable network interface) and system support (a custom-made network driver) to generate additional scheduling events when messages arrive.

[7] gives a performance comparison of gang and coscheduling for data-parallel workloads on a PC cluster. They have implemented gang scheduling via a signal-based distributed user-level scheduler, SCORE-D, which is similar to the prototype presented here except that it does not provide controlled resource allocation.

## 3 Prototype description

The prototype has two main objectives: controlling CPU share with a scheduling server on a single machine, and synchronizing distributed scheduling servers.

### 3.1 Controlling CPU share

The prototype was developed for the Linux operating system, version 2.0. Newer Linux kernels provide real-time scheduling classes (similar to the schedulers of Solaris or NeXTSTEP) in the form of a system call `set_scheduling_priority()`. Therefore, the basic structure is the same as in [13] or [8]: a high-priority server process acts cyclically upon controlled processes, which run at a medium real-time priority. These real-time scheduling classes have strictly higher priority than time-shared processes, and are therefore not impeded by background load. The server process sleeps almost all the time, and only wakes up periodically to act out its scheduling decisions: activating or deactivating controlled processes. The operating system's time-sharing scheduler remains in control of process that are not under the control of the scheduling server. The amount of time given to each process is an input parameter to the scheduling server. This amount can, e.g., be computed by well-known algorithms like rate-monotonic scheduling, or can be set according to user input or requirements.

There are two basic possibilities for acting out scheduling decisions: manipulating priorities, or explicitly suspending/resuming the controlled process. Manipulating priorities can be done by setting a process' scheduling class to either real-time or standard time-sharing. The natural implementation for the second alternative under Linux is to send SIGSTOP or SIGCONT signals. In the first version, a controlled process may run even if it is not activated by the scheduling server (by competing with normal processes for the CPU via standard time-sharing mechanisms); in the second version, this is not possible.

This prototype uses the signal-based approach since it showed better stability under background load compared to an implementation that manipulates process priorities. Also, it provides cleaner semantics as far as maximum CPU share is concerned. Due to the use of standard signals, this prototype should be easily portable to any operating system that supports fixed-priority scheduling classes.

It should be noted that even though signals are reputed for their long (and hard to predict) delivery latencies, this is generally not true for SIGSTOP or SIGCONT signals. These signals are not actually delivered to the receiving processes but rather are processed by the operating system kernel directly and merely change internal tables of the scheduler.

### 3.2 Synchronizing distributed schedulers

Work on gang scheduling and coscheduling has conclusively shown the need to coordinate the execution of distributed threads of a parallel application in time to obtain acceptable performance. Since the scheduling server controls the times of execution of these processes, it follows that the distributed servers need to synchronize their scheduling activities.

Ideally, synchronizing the scheduling servers should be done in a decentralized fashion, with only a few and rare control messages. An obvious way to achieve this is to use some of the well-known clock synchronization protocols — one server sending its remaining sleep time to its peers, which adjust to it. But this

approach is problematic: it requires sleep times of well under a millisecond to achieve acceptable synchronization. However, Linux on PCs only provides timer interrupts with 10 milliseconds granularity.

This raises the need for generating interrupts that start execution of the operating system scheduler at more or less arbitrary points of time. Message arrivals do generate interrupts. Hence, messages to the scheduling servers can be used to start their execution. In the initial prototype, this fact is exploited by designating one of the scheduling servers as master which sends a short control message to the other slave scheduling servers. If the communication medium is capable of broad- or multicast (e.g. like Ethernet) this control message is no scalability bottleneck. This control message only contains the current slot number in the schedule. The schedule itself has to be communicated between machines only when a parallel application starts or finishes. The complete schedule contains the name of the length of a slot, the number of slots in the schedule, and for each slot, the name of the distributed application that occupies it (if any). The slave servers receive a copy of the master server's schedule, which in turn is can be computed according to various algorithms, e.g. rate-monotonic scheduling or simple round robin. Free slots can be used for local applications, and a number of slots has to be left free to allow the rest of the system to make progress.

Since the slave servers run at highest real-time priority, but are usually blocked in the receive call for this control message, they will be scheduled immediately after message arrival. In this design, only the master server sleeps and wakes up in a time-triggered way; the scheduling activities of the slave servers are driven by message arrival from the master server.

Note that this does not mean that messages between the distributed parts of the application have to pass through the scheduling servers. Indeed, the application itself can completely ignore the fact that it runs under control of such a distributed scheduling server; there is no need to modify it in any way.

Comparing this approach with [14] shows that for the price of modifying the network driver, fast scheduling events can be achieved. Also, no explicit control messages are used. However, no share-control is possible. Indeed, both techniques could be fruitfully combined.

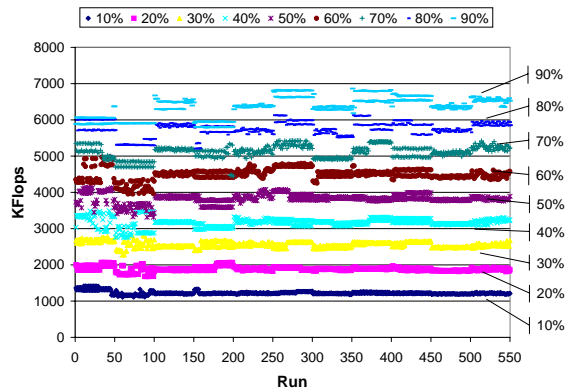
## 4 Experiments

For an experimental evaluation of the proposed prototype, four Pentium 90-based PCs, running Linux version 2.0.31 (completely unmodified) and connected via standard 10 MBit/s Ethernet, were used. The machines were used exclusively, but some external network traffic during the experiments could not be completely ruled out. Section 4.1 shows results about the stability of a scheduling server-controlled program under background load on a stand-alone machine, Section 4.2 briefly introduces the programming model used for the experiments with distributed programs, Section 4.3 discusses the behavior of a single barrier synchronization and Section 4.4 shows execution time for a parallel application with many synchronization operations. More detailed results can be found in [9].

## 4.1 Stability

The main purpose of the proposed scheduling server is to allocate a predetermined share of the CPU to a controlled program and also to make sure that this program does not exceed its share. To investigate the stability of this allocation under increasing load on a stand-alone machine, we used a slightly modified version of Linpack as controlled program. This Linpack version computes the actual floating point performance received during its runtime — this number can be interpreted as the relative share of CPU time.

In Figure 1, Linpack’s performance under scheduling server control is shown, with 50 runs done for 0, 1, 2, . . . simultaneously active background processes each, and CPU share ranging from 10% to 90%. While the behavior is not perfect, the CPU share allocated to Linpack is reasonably stable.



**Fig. 1.** Linpack under Scheduling Server control with different levels of CPU share and increasing number of background jobs

The reserved CPU share should not exceed about 50% or 60%, since at least the operating system itself needs time for its own operation. Stability degrades considerably beyond this point.

## 4.2 BSP programming model

A BSP program consists of parallel threads that run independently and synchronize using barrier synchronization. For the following experiments, each machine is assigned one parallel thread, which runs as an individual process. One such thread is characterized by its execution time. This time is drawn from a uniform random distribution  $\mathcal{U}(g - (g * v)/2, g + (g * v)/2)$ , where  $g$  is the granularity (average thread length) of the application, and  $v$  represents the load imbalance between parallel threads, expressed in percent of the granularity. Here, granularities of 100  $\mu$ s, 500  $\mu$ s, 1 ms, 5 ms, 10 ms and 100 ms are considered, in

combination with load imbalances of 0%, 25%, 50%, 75% and 100%. This barrier is repeated  $n$  times, the total run time of  $n$  cycles forms one measurement, and  $i$  of these  $n$  iterations are performed for statistical relevance.

To achieve implicit coscheduling, two-phase spin-blocking is typically used: a process spins, constantly checking for message arrivals. If, after a certain spin-time, no message has arrived, the process blocks. This spin-time  $t$  is the last parameter for our programming model.<sup>1</sup>

Parallel threads are only allowed to communicate immediately after barrier synchronizations. Here we show results for the following communication patterns:

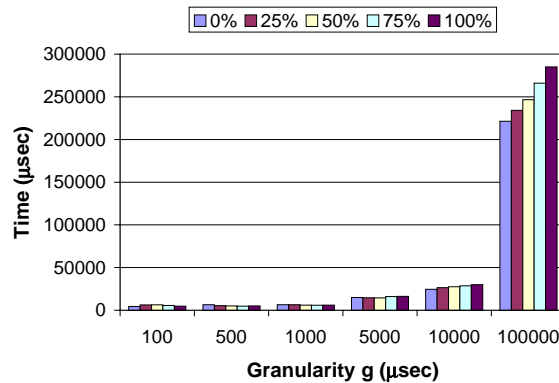
**barrier:** no communication between threads, only the barrier synchronization,  
**complete:** each thread sends a (short, fixed-size) message to every other thread.

Additionally, after receiving a message, a process has to process this message, which takes 10 ms each.

### 4.3 Barrier synchronization

Section 4.1 has established acceptable stability for programs running under scheduling server control on a single machine. This section investigates how a BSP-style program performs when run under scheduling server control and distributed over several machines.

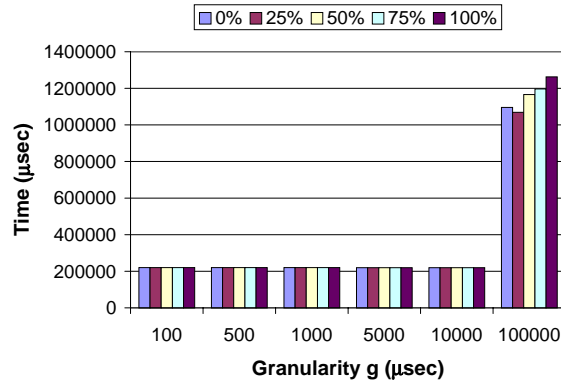
As a basis for comparison, Figure 2 gives the average times in microseconds for a single barrier synchronization, i.e.  $n = 1$ . There is no scheduling server control, blocking communication primitives are used, and the granularities and load imbalances discussed in the previous Section 4.3 serve as parameters.



**Fig. 2.** Performance of a single barrier synchronization without scheduling server, blocking communication

<sup>1</sup> This model is similar to the one used in [6], which additionally suggests an adaptive spin-time. Currently, the experiments reported here are limited to fixed spin-times only.

Using unsynchronized scheduling servers on all machines results in the numbers shown in Figure 3. Here as in all following experiments, the scheduling server allocated 2 out of 10 time slices to the controlled program, where one time slice is 20 milliseconds. Evidently, with unsynchronized scheduling servers, the slowdown is much larger than the expected factor of five — an entire scheduling round is necessary to complete a single synchronization. This clearly indicates that unsynchronized scheduling servers are not acceptable for distributed programs.

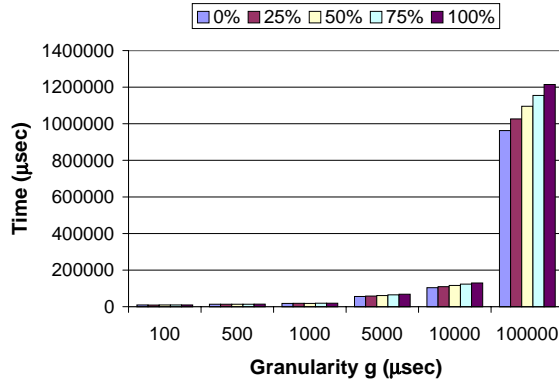


**Fig. 3.** Performance of a single barrier synchronization with unsynchronized scheduling servers, blocking communication

The effects of the synchronization mechanism among scheduling servers proposed in this paper are shown in Figure 4. Again, these times should be about five times as long as in Figure 2, but they are actually better than this. For coarse-grained programs, this slowdown approaches roughly 4.5. For fine-grained program, it is only about 1.5 to 2; since the program is communication-bound and not computation-bound, the CPU limitation is not as severe. This smaller than expected slowdown can be attributed to both coscheduling effects and to the higher priority used by the processes. Due to their vastly superior performance, only synchronized scheduling servers will be considered henceforth.

#### 4.4 Total execution time

While the previous Section 4.3 has investigated the times needed to perform a *single* barrier synchronization, this section looks at the total execution time of a program that consists of *many* such synchronization operations. The behavior differs somewhat between the **barrier** and the **complete** communication pattern, which are discussed in Section 4.4 and Section 4.4, respectively. Blocking and spin-blocking communication is used, as well as programs running without and with scheduling server control (only synchronized scheduling servers are considered). The scheduling servers were again set to give 20% of a machine's



**Fig. 4.** Performance of a single barrier synchronization with synchronized scheduling server, blocking communication

CPU time to the controlled program. The number of synchronization calls  $n$  was again to give reasonable total execution time, but always at least  $n = 50$  iterations were performed for one measurement.  $i = 50$  measurements were taken to compute average and standard deviation of the execution time.

One of the main purposes of using scheduling servers place is to make program behavior more predictable. A simple measure of predictability is standard deviation of runtime. Since the standard deviation depends on the actual scale and is not unit-less, a slightly more convenient parameter is the variation coefficient, which is defined as the standard deviation divided by the average. The variation coefficient will be discussed for the `complete` communication pattern in Section 4.4.

**barrier** To abbreviate the exposition, actual runtime is only shown for a base case. For other cases, only the relative numbers are given with respect to this. The base case is blocking communication without scheduling server support; runtimes for this are shown in Figure 5.

Adding scheduling server control to all four distributed program parts should incur a slowdown of at most five. Figure 6 shows the ratio of runtimes between scheduling server support and uncontrolled program. It is notable that for fine-grained programs, the synchronized scheduling servers perform particularly well and stay well below their maximum acceptable ratio of five. With increasing grain size, the ratio approaches this limit.

The comparison between controlled and uncontrolled execution of spin-blocking programs (with spin time 100 microseconds) is shown in Figure 7. Similar to blocking communication, the scheduling server has a smaller slowdown for fine-grained programs and approaches its natural limit for coarse-grained programs. Since there is no communication after a barrier synchronization, using spin-blocking while under scheduling server control does not significantly increase performance.

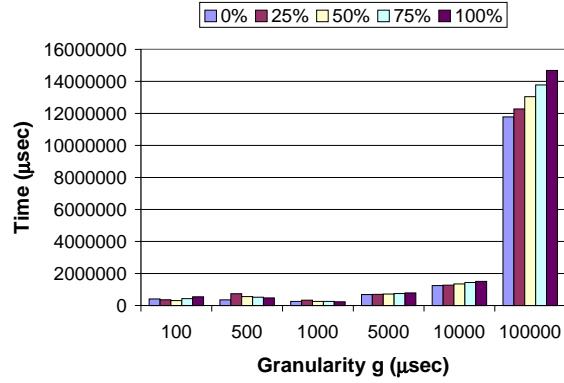


Fig. 5. Runtime of barrier, blocking communication, no scheduling server

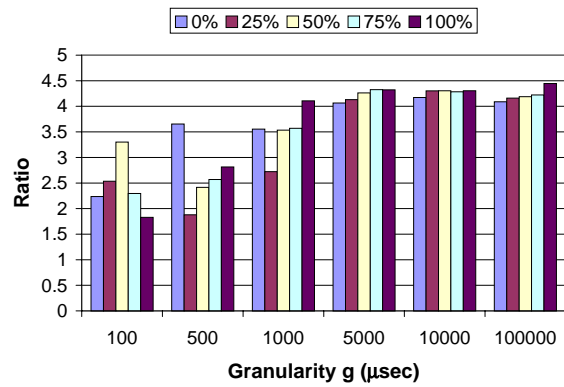


Fig. 6. Ratio of runtime for barrier between scheduling server and uncontrolled program, blocking communication

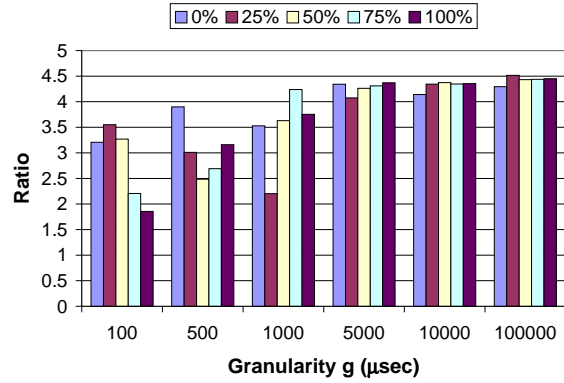
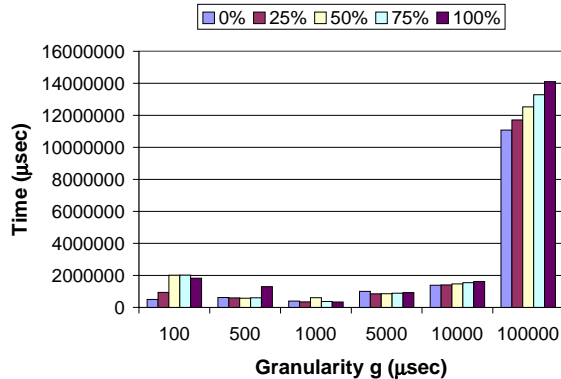


Fig. 7. Ratio of runtime for barrier between scheduling server and uncontrolled program, spin-blocking communication (100 ms)

**complete** This section consider programs with the **complete** communication pattern: after a barrier synchronization has been performed, each process exchanges messages with every other. This puts stronger requirements on coscheduling since messages follow each other immediately. For this pattern, both average runtime and the variation coefficient will be discussed.

*Average runtime* Figure 8 gives the base case: blocking communication, no scheduling server. It is remarkable that the fine-grained, unbalanced cases are slower than those with larger granularity; this is conjectured to be an artificial byproduct of the experimental environment and needs further verification.

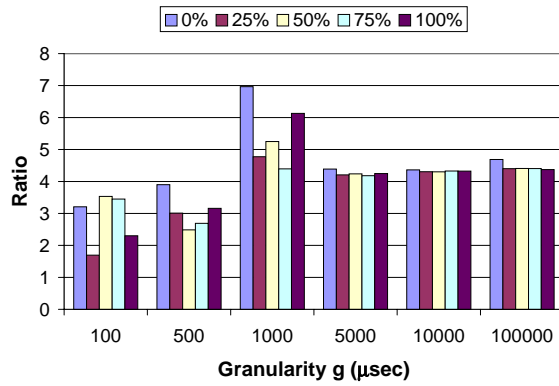


**Fig. 8.** Runtime of **complete**, blocking communication, no scheduling server

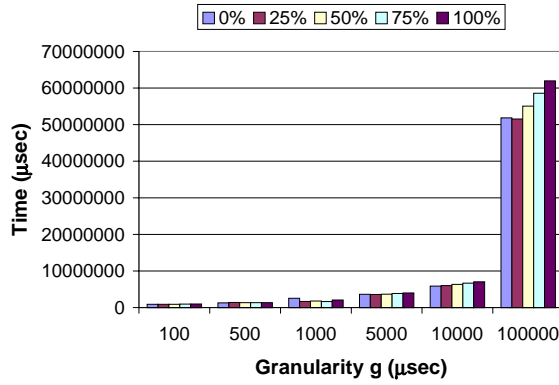
If only blocking communication is used, programs with the **complete** pattern perform less favorably under scheduling server control than programs that only perform barrier synchronizations. For some combinations of  $g$  and  $v$ , the controlled execution takes up to 6.5 times longer than the uncontrolled one. N.B. that in Figure 6 this was not the case.

This changes, however, if the controlled program uses spin-blocking. As can be seen in Figure 9, now the slowdown is on the same order as what should be expected under server control; indeed, it is slightly better due to the higher priority. This comes as no surprise: a program under server control that uses blocking communication gives up its time slice when blocking on a receive. It does maintain the permission to run, since the scheduling server has not (in general) send the SIGSTOP signal. But it has to be rescheduled by the normal operating system scheduler after a network interrupt occurs. Therefore spinning is clearly beneficial for heavily communicating programs even under scheduling server control. This is substantiated by Figure 10 (compared to Figure 8) — the slowdown is again well under the acceptable maximum of five.

*Variation coefficient* Since one of the objectives of using scheduling servers is predictable execution of parallel programs, the variation coefficient is an important metric. Naturally, the runtime variation of programs with a complete



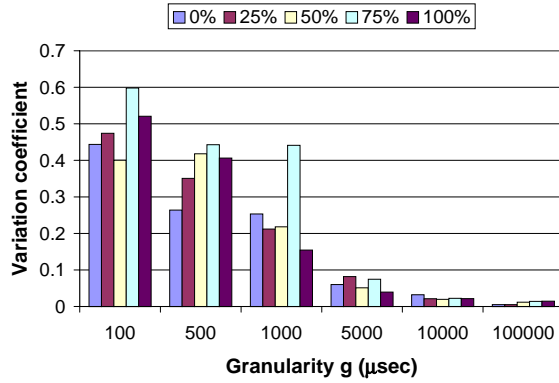
**Fig. 9.** Ratio of runtime for `complete` between scheduling server and uncontrolled program, spin-blocking communication (100 microseconds)



**Fig. 10.** Runtime of `complete`, spin-blocking communication (100 ms), with scheduling server

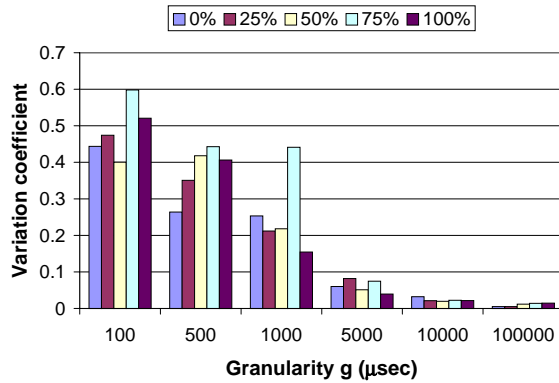
communication pattern is subject to a large number of external influences (e.g. collisions on a shared medium).

The discussion on average execution time has shown that spin-blocking communication is recommendable for programs with the `complete` communication pattern. Hence, Figure 11 shows the variation coefficient for spin-blocking programs with `complete` communication.



**Fig. 11.** Variation coefficient of `complete`'s runtime, spin-blocking communication (100 ms), no scheduling server

This should be compared to the variation coefficient of the same programs under scheduling server control: Figure 12 shows these numbers. Under scheduling server control, the variation coefficient is smaller for fine-grained programs and are more or less unchanged for coarse-grained programs. This shows that scheduling servers improve the predictability of program execution.



**Fig. 12.** Variation coefficient of `complete`'s runtime, spin-blocking communication (100 ms), with scheduling server

## 5 Future work

There are a number of possible extensions. Experiments on a larger cluster to address questions of scalability, multiple distributed programs running under scheduling server control, or with stochastic background load should be performed.

To give guarantees on the execution time, not only the CPU, but other resources like memory (page locking) or network bandwidth should be controlled by the scheduling servers as well.

The coarse timer resolution is a major obstacle. We are currently investigating an extension to the Linux kernel, *utime* [3], that promises to support micro-second timer resolution. This would enable time-driven synchronization as opposed to the message-based synchronization described in this paper. Other operating systems (e.g. QNX) and/or hardware support (similar to [14]) are other interesting possibilities. Finally, the integration into a resource management scheme like [4] is of practical importance.

## 6 Conclusions

This paper addressed the need for guaranteed resources for parallel programs on NOWs. It presented a prototypical implementation of a synchronized scheduling server for the Linux operating system. Among several design choices, a signal-based implementation was used due to its stability and portability.

While a straightforward implementation works well for stand-alone programs, the performance impact on parallel programs proved to be disastrous. A synchronization mechanism was suggested that copes with the limited clock resolution of commodity systems and still achieves reasonable performance by coscheduling distributed programs.

Using extensive measurements, the influence of various parameters like granularity, load imbalance, or communication paradigm was investigated. For fine-grained, moderately communicating programs synchronized scheduling servers provide a reasonable means of achieving coscheduling; for heavily communication programs, spin-blocking has to be added. Synchronized scheduling servers reduce the variation coefficient of program execution time, guarantee a predetermined share of CPU time, and therefore improve the predictability of program execution time.

This prototype gives a practical mechanism to execute even distributed programs in a predictable fashion even on standard operating systems, without having to modify the operating system itself. Combined with consensus-based fault tolerance, this is a step towards predictable network computing.

## References

1. T. E. Anderson, D. E. Culler, David A. Patterson, et al. A Case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

2. A. C. Arpaci-Dusseau, D. E. Culler, and A. M. Mainwaring. Scheduling with Implicit Information in Distributed Systems. In *Proc. of SIGMETRICS '98/PERFORMANCE '98 Joint Conf. on Measurement and Modeling of Computer Systems*, pages 233–243. ACM, June 1998.
3. S. Balji, R. Menon, F. Ansari, J. Keinig, and A. Sheth. UTIME — Micro-Second Resolution Timers for Linux. Project homepage at <http://hegel.ittc.ukans.edu/projects/utime/index.html>, April 1998.
4. A. Baratloo, A. Itzkovitch, Z. Kedem, and Y. Zhao. Just-in-time Transparent Resource Management in Distributed Systems. Technical Report 1998-762, Courante Institute of Mathematical Sciences, New York University, March 1998.
5. H.-H. Chu and K. Nahrstedt. A Soft Real Time Scheduling Server in UNIX Operating System. In R. Steinmetz and L. C. Wolf, editors, *Proc. of Interactive Distributed Multimedia Systems and Telecommunication Services*, pages 153–162, Darmstadt, Germany, September 1997.
6. A. C. Dusseau, R. H. Arpaci, and D. E. Culler. Effective Distributed Scheduling of Parallel Workloads. In *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 25–36, Philadelphia, PA, May 1996.
7. A. Hori, F. B. O'Carroll, H. Tezuka, and Y. Ishikawa. Gang Scheduling vs. Coscheduling: A Comparison with Data-Parallel Workloads. In H. R. Arabnia, editor, *Proc. Intl. Conf. on Parallel and Distributed Processing Techniques and Applications*, volume 2, pages 1050–1057, Las Vegas, NV, July 1998. CSREA.
8. J. Kamada, M. Yuharo, and E. Ono. User-level Realtime Scheduler Exploiting Kernel-level Fixed Priority Scheduler. In *International Symposium on Multimedia Systems*, Yokohama, Japan, March 1996.
9. H. Karl. A Prototype for Controlled Gang-Scheduling. Technical Report Informatik Bericht 112, Institut für Informatik, Humboldt-Universität, Berlin, Germany, August 1998.
10. C. Lee, R. Rajkumar, and C. Mercer. Experiences with Processor Reservation and Dynamic QOS in Real-Time Mach. In *Proc. of Multimedia Japan*, April 1996.
11. M. Malek. A Consensus-Based Framework and Model for the Design of Responsive Computing Systems. In G. M. Koob and C.G. Lau, editors, *Foundations of Dependable Computing: Models and Frameworks for Dependable Systems*, chapter 1.1, pages 3–21. Kluwer Academic Publishers, Boston, MA, 1994.
12. J. K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proc. 3rd Intl. Conf. Distributed Computing Systems*, pages 22–30, October 1982.
13. A. Polze. How to Partition a Workstation. In *Proc. Eighth IASTED/ISMM Intl. Conf. on Parallel and Distributed Computing and Systems*, pages 184–187, Chicago, IL, October 1996.
14. P. G. Sobalvarro, S. Pakin, W. E. Weihl, and A. A. Chien. Dynamic Coscheduling on Workstation Clusters. Technical Report 1997-017, DEC Systems Research Center, Palo Alto, CA, March 1997.
15. L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, August 1990.