Purdue University

Purdue e-Pubs

Department of Computer Science Technical Reports

Department of Computer Science

1996

# On Gang Scheduling and Demand Paging

Dan C. Marinescu

Kuei Yu Wang

## Report Number:

96-016

Marinescu, Dan C. and Wang, Kuei Yu, "On Gang Scheduling and Demand Paging" (1996). *Department of Computer Science Technical Reports.* Paper 1272.
https://docs.lib.purdue.edu/cstech/1272

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries.
Please contact epubs@purdue.edu for additional information.

# ON GANG SCHEDULING AND DEMAND PAGING

Dan C. Marinescu
Kuei Yu Wang

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD TR-96-016
February 1996

To Appear in the Proceedings of the High Performance Computing Conference New Delhi, December 1995

# On Gang Scheduling and Demand Paging*

Dan C. Marinescu and Kuei Yu Wang
Computer Sciences Department
Purdue University
West Lafayette, IN 47907

**Abstract**

Uniprocessor system schedulers hide the high latency of a page fault, by performing a context switch from the process or thread experiencing the fault to another process ready to run. Gang scheduling attempts to schedule all the processes in a process group at the same time. Processes in a process group do not experience page faults at precisely the same time, and process group context switching is in general fairly expensive therefore one cannot hide the latency of a page fault by means of a group context switch.

This paper examines several scheduling strategies for systems which support both gang scheduling and demand paging and reports some measurements of the paging activity of SPMD programs on the Intel Paragon running OSF/1 under Mach.

*Work supported in part by NSF grants CCR-9119388, BIR-9301210 and MCR 9527131, by a grant from the Intel Corporation and by a grant from CNP$_q$ Brasil.

# Contents

1 Introduction 2

2 Gang Scheduling with Busy Waiting 4

3 Mixed Mode Scheduling 7

4 Correlation of the Paging Activity of Individual Node Programs in the SPMD Execution Mode 10

5 Conclusions 15

6 Literature 16

# 1 Introduction

Massively Parallel Processing systems, MPPs, and workstation clusters are distributed memory MIMD, Multiple Instruction Multiple Data systems used for high performance computing. Notable examples of MPPs are the Intel Paragon, the CM5 [6], and the Alliant FX/8 [11]. Clusters of workstations interconnected by medium or high speed networks like Ethernet, FDDI or ATM offer the most cost effective alternative to high performance computing.

Scalability of parallel systems and algorithms is a central issue in high performance computing. From an application point of view, it is highly desirable to run a small problem on one, or a few processors, and to be able to run a large problem on a cluster of tens or hundreds workstations or a large partition of an MPP, without any change of the code. To satisfy this aspect of scalability, the operating systems of the MPPs and workstations must provide similar functionality to the end user.

Demand paging is one of the common features of commodity Workstation Operating Systems, WOS, like Unix or OSF/1. Recently MPPs like the Intel Paragon or IBM, SP2 running commodity operating systems have emerged.

Gang scheduling is a concept introduced by Ousterhout [10]. He observed that a MIMD system performance degrades when a parallel application does not have all its interacting processes scheduled at the same time. Often, the number of active processes in a process group changes dynamically and the term *co-scheduling* relates to the concept of scheduling at the same time only the active processes in a process group.

Supporting both demand paging and gang scheduling is a rather difficult proposition. Demand paging has been studied for the past thirty years or so but gang scheduling is a relatively new concept and the interaction between the two are not understood at all. The latency of a page fault is hidden by the scheduler of a uniprocessor system by performing a context switch, the process or thread experiencing the fault is blocked and another process ready to run is scheduled. The problem is considerably more delicate when the operating system supports gang scheduling and demand paging. Hiding the latency of a page fault by a group context switch would only be feasible if (a) all processes in the group experience page faults at precisely the same time, and (b) the overhead associated with a group context switch would be much smaller than the page fault service time. Unfortunately, neither condition seems to be true. The measurements we report in §4 of this paper, in [13], and [14], indicate that even in the SPMD execution mode, when all PEs execute the same program but on different data, the actual paging rates and the time when the paging activities reaches its peaks are very different for different PEs.

There are open questions if operating systems should support the same functionality across a range of systems with widely different resources and usage. Should an WOS support gang scheduling. Should such support be provided by the scheduler or by a batch queuing system which can only guarantee some form of loose synchronization and a quasi-gang scheduling support. There are obvious advantages in using the idle cycles of workstations but is it possible to mix together an interactive load with a batch one and still guarantee the response time required by the interactive transactions and, in the same time, some form of synchronization required by gang scheduling.

Should the operating system of an MPP system, MPPOS, support demand paging and process group context switching. We hear often the argument that high performance computing should favor performance versus ease of use and/or unnecessary functionality. For example MPPOS kernels like SUNMOS provide a rather primitive functionality but levels of performance difficult to achieve under a commodity operating like OSF. There is no support for demand paging, node I/O, or process group context switching in SUNMOS.

We believe that the future parallel systems should be less dependent on intuition and more on measurements and understanding of the behavior of parallel programs. The questions raised above can only be answered by a careful analysis of existing parallel systems and parallel programs.

Only applications which use scalable algorithms can possibly take advantage of a scalable system, namely run efficiently on one PE for a small problem and on many PEs for a large problem. In addition, the application must be built around a self-scheduling algorithm, which discovers at run time the size of the system and of the problem and distributes the work accordingly.

The synchronization requirements of the application must be taken into account when deciding if gang scheduling is necessary [5]. In case of fine-grain interactions, gang scheduling and busy waiting lead to the best performance. For coarse-grain interactions, gang scheduling is not necessary. The synchronization requirements of an application are determined by the algorithms used and they are affected by the the computing and the communication speed ot the system used to run the application.

In this paper we present models of the paging activity of a parallel program and attempt a *qualitative* analysis to illustrate the benefits and the drawbacks of two gang scheduling algorithms, one suitable for MPPs and one for workstation clusters. The mixed mode algorithm we propose, is based upon pragmatic consideration and the need of the user community to use clusters of workstations as an alternative to expensive MPPs.

We are in the process of building a performance monitoring and analysis tool. The 4M system will support automatic instrumentation of parallel programs using a variety of communication and I/O primitives. It will support MPI, PVM and native communication and I/O libraries for the Paragon and SP2 systems. The analysis tools will allow to correlate paging and I/O activities of individual node programs and to study the overall load placed upon the I/O and paging systems. We believe that only a careful study of a parallel program can lead to the determination if it is truly scalable, if it can be run efficiently on a cluster of workstations and we hope that the 4M tool suite would provide such information to a user. In the same, we sense the need to gather data concerning the paging and I/O activity of parallel programs to support further developments in the area of parallel systems architecture.

Results that support gang scheduling were presented in [1], [5], [7], [12]. In our opinion a quantitative analysis based upon analytical models is unlikely to be successful at this time due to the complexity of the interactions involved in gang scheduling,

# 2 Gang Scheduling with Busy Waiting

Gang scheduling with busy waiting is a scheduling strategy supporting a static partition of a MIMD system. A job consisting of a process group is assigned a partition of the machine with a number of processors equal to the process group size and releases the partition upon the completion of all processes in the group. When a process becomes not ready, due to a page fault, an I/O operation, or a communication event, it does not release the control of the processor. This gang scheduling strategy is currently used by the PARAGON, CM5 [6], and Alliant FX/8 system [11]. When combined with support for demand paging, this strategy leads to wasted CPU cycles and longer execution time.

To formalize these concepts the following notations are used:

| | | |
|---|---|---|
| $A$ | = | is a process group $A = \{A_1, A_2, \ldots, A_{n_A}\}$. |
| $n_A$ | = | the size of the process group. |
| $r_i^A$ | = | the execution time (elapsed time) of process $A_i$ *without* any page faults. |
| $t_i^A$ | = | the execution time (elapsed time) of process $A_i$ *with* page faults. |
| $q_i^A$ | = | the count of synchronization events experienced by process $A_i$ during its lifetime. |
| $s_{ij}^A$ | = | the delay experienced by process $A_i$ at its $j$-th synchronization event due to paging activity. |
| $p_i^A$ | = | the average page fault rate of process $A_i$. |
| $w_i^A$ | = | the working set of process $A_i$. |
| $m_i^A$ | = | the size of the address space of process $A_i$. |
| $\varphi$ | = | the page fault service time. |
| $t^A$ | = | the execution time of process group $A$, the time the entire partition is allocated to $A$. |
| $p^A$ | = | the cumulative page fault rate due to process group $A$. |
| $m^A$ | = | the cumulative address space of process group $A$. |
| $w^A$ | = | the cumulative working set of process group $A$. |
| $M_i$ | = | the size of the main memory available on processor $\mathrm{PE}_i$ process $A_i$ is allocated to. |
| $\eta_i$ | = | the efficiency of processor $\mathrm{PE}_i$. |

The execution time in the presence of page faults is:

$$t_i^A = r_i^A(1 + p_i \times \varphi) + \sum_{j=1}^{q_i^A} s_{i,j}^A$$

and

$$t^A = \max_{(i)}(t_i^A) \quad 1 \leq i \leq n_A$$

In general it is difficult to estimate the synchronization delays $s_{i,j}^A$. The delay experienced by process $A_i$ at its $j$-th synchronization event, depends upon the communication pattern, the rank of the process in the group, the total delay due to page faults accumulated during the previous $j-1$ synchronization events, and possibly other factors. For example, in case of a circular synchronization pattern $A_1 \rightarrow A_2$, $A_2 \rightarrow A_3, \ldots, A_{n_A-2} \rightarrow A_{n_A-1}$, $A_{n_A-1} \rightarrow A_{n_A}$

$$s_{n_A,1} \geq \varphi \sum_{i=1}^{n_A-1} s_{i,1}^A p_i$$

for the first synchronization event.

The efficiency is

$$\eta_i \leq \frac{1}{1 + p_i^A \varphi}.$$

The cumulative average paging rate due to process group $A$ is:

$$p^A = \sum_{i=1}^{n_A} p_i^A.$$

For example, assume that the size of the process group is 1000, the page fault service time is $10^{-2}$ seconds, that all processes in the process group experience an average of 10 page faults/seconds, then the upper bound for efficiency is $\eta = 90\%$ and the cumulative page fault rate is 10,000 page faults/second.

Throughout this paper, the average page fault rate is obtained by dividing the number of page faults by the CPU time. The CPU time in case of gang scheduling with busy waiting is equal to the elapsed time.

Several strategies to improve the efficiency of gang scheduling with busy waiting may be considered. One is to reduce the page fault service time $\varphi$. Another is to determine apriori the working set $w_i^A(t)$ of each process in the process group and to ensure that all pages in $w_i^A(t)$ are fetched prior to time $t$.

The first strategy requires a storage hierarchy so that either the entire or some fraction of the cumulative address space $m^A$ is cached on some page server. In this case the page fault service time could be reduced by two or three orders of magnitude. This is a very costly solution and one may argue that a better approach is to distribute the memory of the page server to individual nodes and thus reduce the page fault rate.

The second strategy could be useful if a program is run several times and the dynamics of the working sets of individual node programs does not change drastically from run to run.

This strategy is used in a virtual shared memory implementation of a structural biology application and it is described in [4].

In spite of its drawbacks discussed above, gang scheduling with busy waiting is probably the only solution for solving large, memory constrained problems. In such cases, $w_i^A \simeq M_i$ for $i = 1, n_A$, the working set of each process of the process group barely fits into the local memory of each PE and any attempt to use the CPU cycles when process $A_i$ is waiting would fail because there is not enough memory to hold the working set of another process.

# 3 Mixed Mode Scheduling

This scheduling strategy is most suitable for clusters of workstations interconnected by a medium or high speed network (Ethernet, FDDI, ATM, etc.). Each workstation operates under the control of a local scheduler which mixes gang scheduling with scheduling of single processes. Then single processes may carry out interactive tasks generated locally while gang scheduling covers primarily compute intensive tasks as described by Atallah et al in [2].

The basic idea of the algorithm is to allow single processes to run whenever the process selected by the gang scheduling algorithm blocks, due to a page fault or due to an I/O operation. If the process stops for a synchronization event, the process remains active in a busy waiting state. Gang scheduling occurs relatively infrequently, once every major scheduling cycle, because of the relatively large overhead associated with a process group context switch. The following parameters characterize the mixed mode scheduling.

$\tau$ = the major scheduling cycle time.

$\tau_g$ = the time allocated during a major scheduling cycle to gangs.

$\tau_s$ = the time allocated during a major scheduling cycle to single processes.

$\alpha_g$ = the time required for a process group context switch.

$\alpha_s$ = the time required for a single process context switch.

$\sigma$ = the time slice.

$\varphi$ = the page fault service time.

$\eta$ = efficiency.

Typical values for the parameters above could be:

$$\tau = 10 \div 10^2 \text{ seconds},$$

$$\alpha_g = 10^{-2} \div 10^{-1} \text{ seconds,}$$
$$\alpha_s \simeq 10^{-4} \text{ seconds,}$$
$$\sigma \simeq 10^{-2} \text{ seconds,}$$
$$\varphi \simeq 10^{2} \text{ seconds.}$$

In general

$$\alpha_s << \sigma$$
$$\alpha_g >> \sigma$$
$$\varphi \simeq \sigma$$
$$\tau >> \sigma$$
$$\tau = \alpha_g + (\tau_g + \tau_s)$$

Assuming that every time when the process in the process group blocks, there is at least one single process ready to run, the efficiency of this scheduling algorithm is

$$\eta \simeq \frac{\tau - \alpha_g}{\tau} = 1 - \frac{\alpha_g}{\tau}.$$

If process group $A$ has the highest priority among all process groups then the running time of process $A_i$ in this mode is

$$T_i^A = t_i^A \left(1 + \frac{\alpha_g}{\tau}\right).$$

To implement this algorithm each scheduler performs an endless loop consisting of the following actions:

**Step 1.** Perform a gang context switch. Select a process group and synchronize the clocks of all processors assigned to the active processes of the process group. In case of SPMD (Same Program Multiple Data) execution mode, the working sets consist of the entire process group and the selection process may be based upon the priority of the process group.

Set $\tau'$, the count of remaining time slices in a major scheduling cycle, $\tau_g'$, the count of remaining time slices for the process group, and $\tau$'s, the count of remaining time slices for single processes as:

$$\tau' = \tau, \quad \tau_g' = \tau_g, \quad \text{and} \quad \tau_s' = \tau_s.$$

When a time slice expires, decrement $\tau'$ and either $\tau_g'$ or $\tau_s'$ if a group or a single process was active during that time slice. Go to Step 2.

**Step 2.** Run the process member of the selected process group. At the end of the time slice, or when a process blocks perform one of the following actions:

(a) If ($\tau' = 0$) go to Step 1.

(b) If ($\tau'_g = 0$) go to Step 4.

(c) If the process blocks due to a page fault or an I/O operation, go to Step 3.

(d) If the process waits for a synchronization event, it remains active in a busy waiting state.

(e) Else repeat Step 2.

**Step 3.** Run processes from the single process queue or block if this queue is empty, until the process member of the selected process group is ready again. Then go to Step 2.

**Step 4.** If $\tau'_s \geq 0$ run processes from the single process queue or block if this queue is empty. Repeat Step 4 until the major cycle expires, $\tau' = 0$, then go to Step 1.

The scheduling algorithm described above can be implemented using a multi-queue scheduler with a queue for process groups and one for individual processes. An alternative is to change dynamically process priorities. At the beginning of a major scheduling cycle the priority of the process belonging to the selected groups is raised above all other user processes and then it is lowered at the end of the gang scheduling allocation of a major scheduling cycle to allow single processes to run.

One may have different $\tau_g/\tau_s$ ratios to accommodate heterogeneity amongst different processors assigned to a process group. The faster the processor is the lower should this ratio be. Mixed mode scheduling works best when the time slice $\sigma$ is slightly larger than the page fault service time $\varphi$. In this case, when a page fault occurs, the context switch to an single process allows the utilization of the CPU cycles, but by the time of the next context switch the process belonging to a process group is ready to run again, and thus the elapsed execution time of a high priority process group is only slightly larger than the one experienced in case of gang scheduling with busy waiting.

Scheduling of compute intensive tasks on clusters of autonomous workstations could be done in a two phase approach. In the first phase, a bidding procedure like the one described in Atallah et al [2] is used to identify the workstations participating, the start-up time, the ratio of CPU cycles for gang scheduling, and for individual processes, and the priority of the process group. In phase 2 the brokers on each system add the process group to the ready queue at the agreed upon startup time and then the mixed mode strategy ensures that all processes in the process group run concurrently.

# 4 Correlation of the Paging Activity of Individual Node Programs in the SPMD Execution Mode

A detailed account of the methodology used for measuring the paging activity of parallel programs can be found in [14]. In this section we restrict our attention to the Same Program Multiple Data, SPMD, programming model and present some observations.

The conclusions drawn from our analysis of a few programs have to be confirmed by additional data before they can be used to improve the design of massively parallel systems and the mechanisms for resource sharing in such systems.

We have concentrated our attention on a few applications in the area of computational biology which we helped develop over the past few years. The programs we have studied are used for the 3-D atomic structure determination of large macromolecules such as viruses. These programs are: (a) The Envelope program used for real space electron density averaging. (b) The FFTsynth, a program used for transformation from reciprocal to real space by means of 3-D FFT. (c) The Recip program used to correlate calculated structure factors with observed ones. A brief outline of the computations and the algorithms used by the Envelope program follows.

The input for the Envelope program is a 3-D lattice with $nx \times ny \times nz$ points. Every grid point with coordinates $(x_0, y_0, z_0)$ has an electron density $\rho_{x_0,y_0,z_0}$. Symmetry operators: $\pi_1, \pi_2, \ldots, \pi_n$ allow us to associate to every grid point $(x_0, y_0, z_0)$ $n$ other points, $(x_1, y_1, z_1)$, $(x_2, y_2, z_2) \ldots (x_n, y_n, z_n)$, related to it by non-crystallographic symmetry. The electron density at every grid point is replaced by the average value of the electron density of all the points related by non-crystallographic symmetry.

The parallel algorithm used for averaging is based on a partition of the 3-D lattice into small volumes (bricks). Each PE is assigned a number of bricks to transform, but it needs access to the entire data space (the entire lattice) because points related by the non-crystallographic symmetry are scattered throughout the lattice. The program implements a shared virtual memory and operates in two modes, (a) the disk mode (DFS), and (b) the data cacheing in the nodes mode (DAN).

The memory maps of the two modes differ in the following way. In the DFS mode each processor fetches bricks on demand from the disk file into the dynamic brick memory area. In the DAN mode the bricks are cached in the static brick memory area of all nodes. When a processor needs a brick not available locally, it uses interrupt driven communication to fetch it from another processor's memory.

The measurements were performed on a Paragon XP/S system with 66 compute nodes, two I/O nodes and three service nodes with 32 Mbytes of memory per node. The system is

currently running Paragon OSF/1, Release 1.0.4, Patch R1.1.6.

We discuss briefly the intrusion due to our monitoring. Comparing the size of the original load module and of the load module of the instrumented programs, we noted that the instrumented code is 5-11% larger than the original code. The effects of instrumentation upon the execution are visible in the execution time of instrumented code of FFTsynth program which takes about 15% more time.

Yet another form of intrusion which affects the data obtained through monitoring, is caused by interactions of the program being monitored with programs running concurrently in other partitions of the system. This type of interaction is due to contention for the I/O and communication resources, and always leads to an increase of the execution time. Even when a program is not instrumented, its execution time may vary by a significant amount for the same input data depending upon the activities of its competitors. In our monitoring process, the correlation of the peaks of activity in time is affected by this type of intrusion.

While the first type of intrusion, the one due to our measurements *cannot* be eliminated, there are costly ways to eliminate the second type by using the machine in an exclusive fashion.

Our methodology for monitoring the paging activity of a parallel program is based upon detecting transitions from one state of the task to another and recording the paging activity data collected by the kernel at the time of the transition. A parallel program can be in one of the following states: compute, I/O and communication.

In this section we present synthetic data closely related to our model, namely (a) the total number of events recorded during the execution of the program, (b) the average number of events per node, (c) the total time spent by a program in each state, and (d) the average duration of an event for all the programs we have monitored. Figures 1 and 2 present these data for the Envelope program in the DFS and DAN mode.

The data we collect to study the paging activity of parallel programs are extremely useful for understanding the behavior of a parallel program, the way it uses the resources of the parallel system and for determining means for improving the performance of the parallel program.

For example, Figures 1a and 1b show that the Envelope program in DFS mode performs a large number of I/O operations and that there are virtually no communication events. Figure 1c shows that the I/O bandwidth of the system is insufficient. As a result there is no gain in using 64 PEs; the execution time with 64 PEs is essentially the same as the one with 32 PEs, about 1,300 seconds. While in the case of 16 nodes, the time spent in the I/O state is less than 10% of the total execution time (200 seconds for a 2200 seconds execution time), it represents more than 60% when 64 PEs are used (about 1,000 seconds out of 1,300 seconds total execution time). Figure 1d provides additional arguments that the I/O is the

bottleneck: the average duration of an I/O event increases from about 0.08 seconds for 16 PEs to about 0.2 seconds for 32 PEs and about 0.5 seconds for the 64 PEs.

This analysis justifies our approach used in the DAN execution mode of the Envelope program to cache data across nodes. Figures 2a-d present this mode of execution using 16 PEs in two different experiments called A and B, and 32 PEs. As expected, the number of I/O events is considerably lower, but there are many more communication events compared with the DFS mode. The total execution time is reduced, about 1,600 seconds (versus 2,200 seconds) with 16 PEs, and about 800 seconds (versus 1,300 seconds) with 32 PEs. Figure 2c also shows that the total time spent in the compute state is a fairly large fraction of the total time (more than 90% for the 32 PE case) and that the fraction of time spent in the I/O state is insignificant. Figure 2d illustrates that the execution time of a program is influenced by the other program running concurrently in other partitions of the system. In the case of experiment A the average duration of an I/O event is larger than that of experiment B, because for A, other programs requesting I/O were running concurrently.

The methodology used to study the correlation of the paging activities of individual PEs is to isolate the peaks of activity and to study how their amplitude and time of occurrence relate to each other. For example, Figure 3 shows the page fault data obtained for the Envelope program running in 16 nodes in DFS mode. Figures 3a, 3b, and 3c show the amplitude correlation in the three states, compute, I/O and communication, while Figures 3d, 3e, and 3f show the same data for time correlation. show the same data for time correlation. From Figure 3a, we see that in the compute state, we have isolated 16 peaks. Peaks 1, 2 and 5 exhibit the most dissimilar behavior. For example, among the 16 PEs, the lowest rate of page fault for the first peak of activity is slightly lower than 250 faults/sec and the highest rate observed is slightly lower than 950 page faults/second. Figure 3d shows that the first seven peaks of the page-fault activity occurred within the first 10-15 seconds and the time elapsed between the first and the last occurrence of a certain peak is less than 50 seconds. We see that the time elapsed between the first and the last occurrence of a certain peak increases as the peak id increases. For example, the 15-th peak occurred first after about 220 seconds and the last PE experienced this peak some 1700 seconds later.

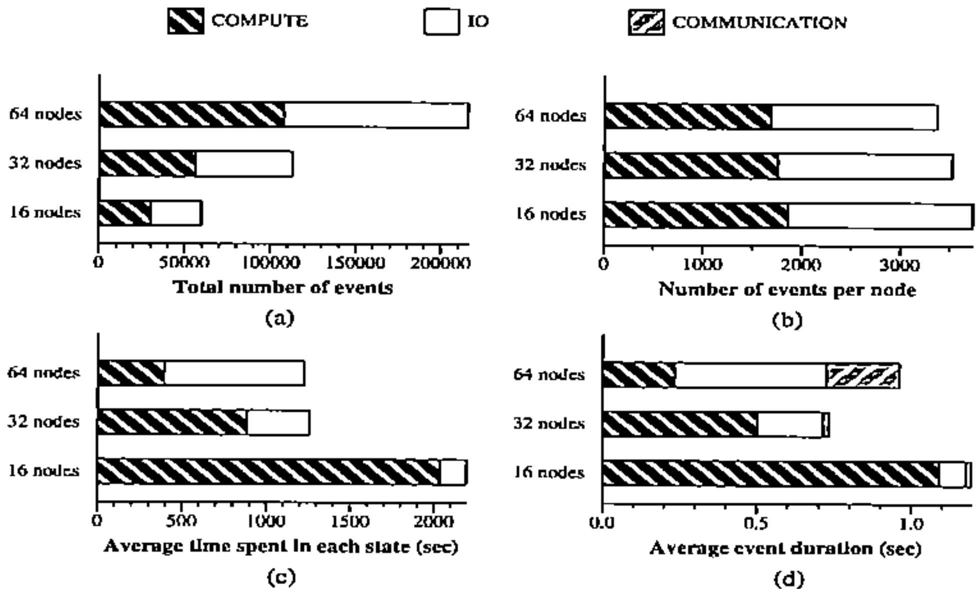



Figure 1: Summary of information concerning the number of events, the average lifetime of a state and the average lifetime of an event for the Envelope program running in DFS mode.

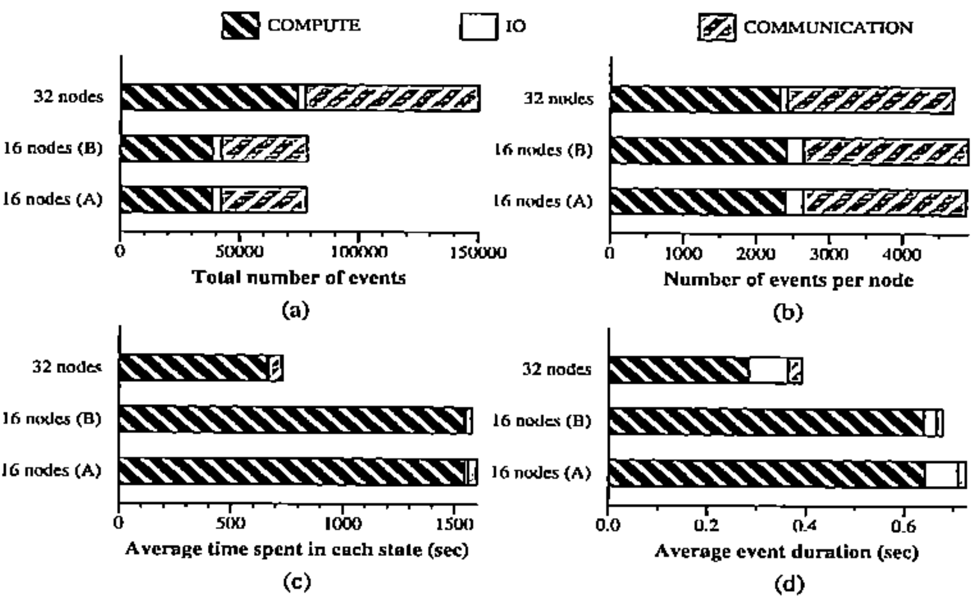
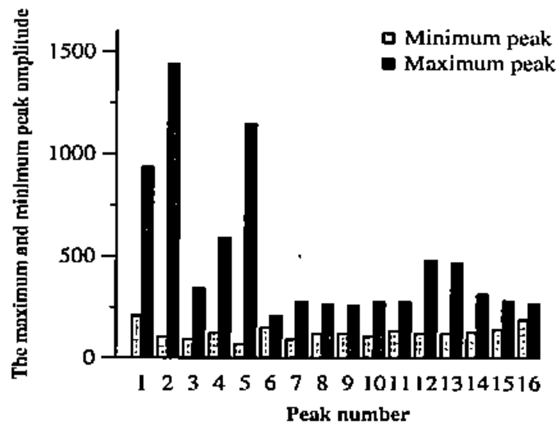
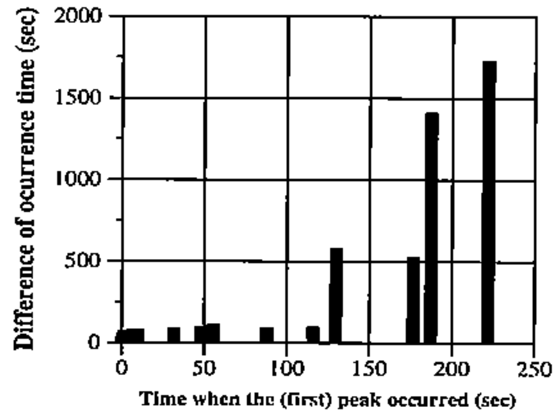


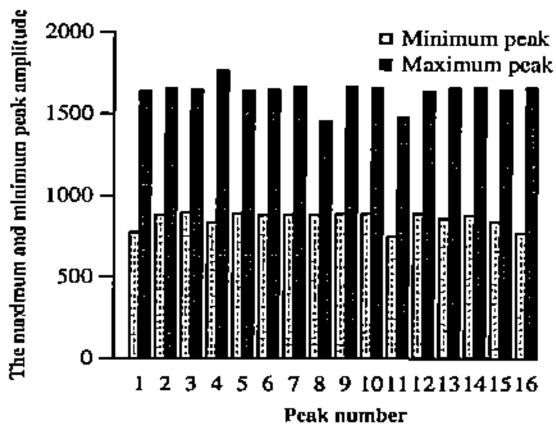

Figure 2: Summary of information concerning the number of events, the average lifetime of a state and the average lifetime of an event for the Envelope program running in DAN mode.



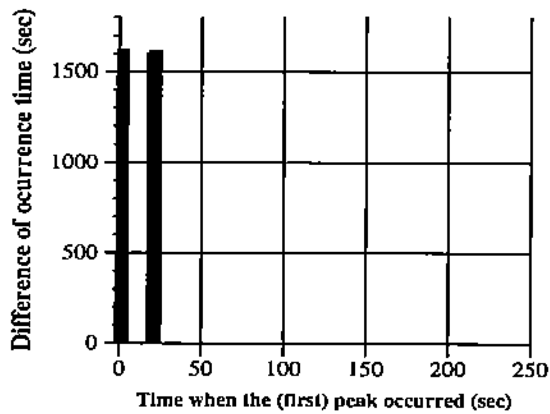

(a) COMPUTE state: amplitude correlation



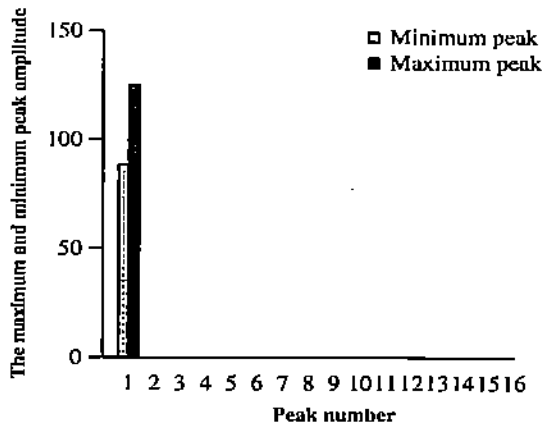

(d) COMPUTE state: time correlation



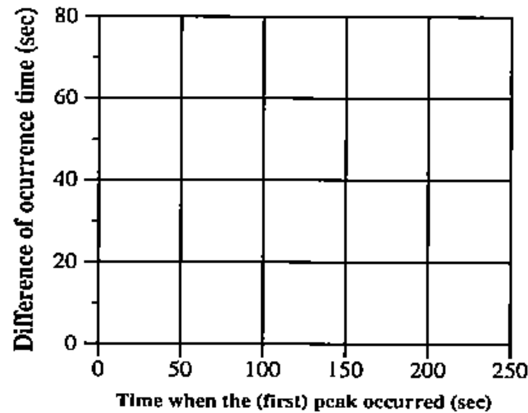

(b) I/O state: amplitude correlation




(e) I/O state: time correlation




(c) COMMUN. state: amplitude correlation




(f) COMMUN. state: time correlation

Figure 3: The page fault analysis. The correlation of amplitude and time of occurrence for the peaks of page fault activity for the Envelope (DFS mode) program running in 16 nodes.

# 5 Conclusions

At this time there are no solutions to support efficiently demand paging and gang scheduling on MPPs. Gang scheduling with busy waiting seems the only strategy suitable for executing parallel programs with fine grain synchronization on MPPs but paging and I/O activity increase dramatically the execution time and lower the processor utilization. Increasing the amount of PE memory makes the problem less severe but it is no substitute for demand paging.

Workstation clusters have a rather high communication latency and relatively low bandwidth though improvements in both areas are expected due to more efficient communication protocols and high speed networks like ATM. Yet there are parallel programs that can run efficiently on a workstation cluster and the mixed mode scheduling algorithm proposed in this paper may provide a solution to the problem of concurrent execution of communicating tasks on a cluster of workstations. Further experiments in this area are needed.

We report measurements of the paging activity of SPMD programs. Our measurements suggest that even in the SPMD case, the paging activities of different PEs are rather uncorrelated in time and the amplitude of the peaks of activity varies widely.

More measurements of the paging, I/O activity, and synchronization requirements of parallel programs are needed to provide a foundation for the design of future parallel systems. We may also need to revise our idyllic concept of scalability and recognize that it can only be achieved for some range of the number of processors and some parallel programs.

# 6 Literature

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

[2] Atallah, M.J., Lock, C., Marinescu, D.C., Siegel, H.J., and Casavant, T.L., Models and algorithms for co-scheduling compute-intensive tasks on a network of workstations. *Journal of Parallel and Distributed Computing* vol **16**, (1992), pp. 319–327.

[3] Burger, D.C., Hyder, R.S., Miller, B.P., and Wood, D.A., Paging tradeoffs in distributed-shared-memory multiprocessors. *Supercomputing'94*, Nov. 1994.

[4] Cornea, M., Marinescu, D.C., and Zhang, Z., Data management for a class of iterative computations on distributed memory MIMD systems. *Concurrency: Practice and Experience* vol **6(3)**, (1994), pp. 205–229. *Computer* **23**, 5 (May 1990), 65–77.

[5] Feitelson, D.G., and Rudolph, L., Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing* **16** (1992), pp. 306–318.

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

[7] Leutenegger, S.T., and Vernon, M.K., The performance of multi-programmed multiprocessor scheduling policies. *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.* (May 1990), pp. 226–236.

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

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

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

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

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

[13] Wang, K.Y., and Marinescu, D.C., Correlation of the paging activity of individual node programs in the SPMD execution mode. *Proceedings of HICSS'28, IEEE Press* vol 1, (1995), pp. 61–71.

[14] Wang, K.Y., and Marinescu, D.C., Characterization of the Paging Activity of NAS Benchmark Programs on the Intel Paragon, *Computer Sciences Department, Purdue University, CSD-TR-95-015*