1996

Dynamic Scheduling of Process Groups

Kuei Yu Wang

Dan C. Marinescu

Octavian F. Carbunar

Report Number:
96-030

DYNAMIC SCHEDULING OF PROCESS GROUPS

Kuei Yu Wang
Dan C. Marinescu
Octavian F. Carbunar

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD TR-96-030
May 1996
(Revised 6/96)

Dynamic Scheduling of Process Groups*

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

May 30, 1996

Abstract

In this paper we introduce the concept of temporal locality of communication for process groups. Empirical evidence suggests that, once a member of a process group starts to communicate with other processes in the group, it will continue to do so, while an independent process will maintain its state of isolation for some time. Other instances of inertial behavior of programs are known. Temporal and spatial locality of reference are example of inertial behavior of programs, exploited by hierarchical storage systems; once a block of information (program or data) is brought into faster storage, it is very likely that it will be referenced again within a short time frame. The temporal locality of communication can be used to schedule concurrently multiple process groups. When process groups exhibit temporal locality of communication, this information can be used to hide the latency of paging and I/O operations, to perform dynamic scheduling to reduce processor fragmentation, and to identify optimal instances of time for checkpointing of process groups.

*Work supported in part by NSF grants BIR-9301210 and MCR-9527131, by the Scalable I/O Initiative, by a grant from Intel Corporation and by CNPq Brazil

1 Overview

The availability and usage of multiprocessor systems have grown rapidly. Multiprogramming parallel machines, with several parallel applications being processed concurrently, has been proposed as a method to improve utilization of multiprocessor systems.

Multiprogramming can be achieved by time sharing the multiprocessor among several applications, as in the uniprocessing systems, or by space sharing where subsets of processors are assigned to various applications. Thus, although multiprogramming allows better service to be provided to the users, it also complicates the processor allocation issues in multiprocessor systems.

Processor scheduling algorithms for general purpose multiprogrammed multiprocessors are not yet well understood [9]. Two classes of processor scheduling algorithms for multiprogrammed parallel systems are the static and the dynamic scheduling algorithms [8], [3], [11], [12]. Static scheduling algorithms are non-preemptive scheduling algorithms in which each application runs to completion without interruption on the set of processors initially allocated for it. All others belong to the class of dynamic scheduling. The static scheduling algorithms are simpler to implement and have lower overhead than dynamic algorithms. On the other hand, dynamic algorithms may adjust the spatial and/or temporal allocations of processing elements to execute a parallel application according to its need for resources and to system's requirements for load balancing and load sharing.

Dynamic scheduling algorithms are rarely used because it is very difficult to characterize analytically the behavior of interacting processes. The interaction between the computation time of the processes, the time spent while waiting for messages from other processes, and the time spent while waiting for processor to be available are too complex to be predicted accurately and studied in advance.

A commonly implemented processor allocation strategy, available on most existing system, is the gang scheduling with busy waiting. Gang scheduling with busy waiting is a scheduling strategy supporting a static partition of a multiprocessor 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 blocks due to a page fault, an I/O operation, or a communication event, it does not release the control of the processor. When combined with support for demand paging, this strategy leads to wasted CPU cycles and longer execution time [2], [13].

The paper is organized as follows. Section 2 discusses the concept of temporal locality

of communication and describes a simple mechanism to hide latency of page faults and I/O operations for temporarily independent processes. Section 3 presents a simple state transition diagram used to detect at run time when a member of a process group is temporarily independent and then shows how dynamic scheduling can be used to minimize the effects of processor fragmentation. Section 4 presents the empirical evidence suggesting that process groups exhibit temporal locality of communication. We describe the programs we have instrumented and the clustering algorithms used to identify the clusters of communicating processes in a process group.

2 Interleaved scheduling of multiple process groups

A *process group*, $P = \{P_1, P_2, \ldots, P_i, \ldots, P_n\}$, consists of n processes, P_i, $i = 1, n$, that need to be scheduled concurrently on the Processing Elements, PEs, of a parallel system because they communicate with one another.

The corresponding scheduling mechanism called *gang scheduling* requires concurrent scheduling of all members of a process group regardless of the dynamics of their synchronization. When a member P_i of a process group P encounters a page fault, in case of demand paging, or waits for the completion of an I/O operation, the process blocks but the local scheduler does not perform a context switch. Gang scheduling, coupled with busy waiting, is used extensively by existing MIMD systems, e.g., Paragon, CM5, SP2, Alliant FX/8 for parallel applications which exhibit fine-grain interactions.

Co-scheduling is an alternative to gang scheduling [1], [10]. In case of co-scheduling, the dynamics of synchronization requirements of the application is taken into account; only those members of the process group which need to communicate with one another are scheduled concurrently. Determining the working set of the process group – the set of members which need to communicate with one another at a given moment of time – is a difficult proposition, and we are not aware of any system which implements such a co-scheduling policy.

There is another practical motivation for gang scheduling with busy waiting, namely, memory constraints. Often, since the amount of physical memory available to individual PEs is insufficient to accommodate multiple process groups, performing a partial context-switch is impractical. Yet, gang scheduling with busy waiting is likely to incur increasingly higher costs because the processor speed experiences a considerable higher improvement rate than the I/O speed. Moreover, one way to increase the I/O bandwidth is to coalesce I/O operations based on temporal and spatial locality of reference, to transfer larger blocks of data.

The question on how to hide the high latency of I/O operations in case of gang scheduling is still unresolved. In this section we discuss a possible solution to this problem, based upon our studies of the dynamics of synchronization for several applications we have examined. For simplicity, assume that time is slotted and the duration of a slot, Δ, is much larger than the time required to perform a local context switch of process P_i running on PE_k. We consider Δ to be approximately equal to the time to perform an I/O operation. Let us for the moment assume that we can predict with some level of confidence that a member of a process group is unlikely to experience a communication event during the next few slots. Call such a process a *temporarily independent* process, or TI-process. If at time t, process $P_i \in P$ experiences an event leading to blocking, then one could consider the following alternative to busy waiting:

(a) Determine if P_i is a TI-process and, if so, find if there is another TI-process $Q_j \in Q$ ready to run on PE_k, where P_i is the currently running process. This is a local decision of the scheduler of PE_k.

(b) If conditions in (a) are satisfied then perform a local context switch and let Q_j run until P_i becomes dispatchable again. Otherwise, block P_i and wait until it is ready.

The strategy described above is practical if one can determine with relative ease when a member of a process group belongs to the TI-class.

To explain the intuition behind the heuristics used to determine if a process belongs to the TI-class, we review briefly the concept of locality of reference, another example of inertial behavior of a program. Modern computer systems have a hierarchy of storage, primary and L2 caches, main memory, virtual memory based upon demand paging. As one traverses this hierarchy, the latency increases, but the cost of storage decreases, and the size of the available storage increases. Hierarchical storage systems work because programs tend to exhibit both temporal and spatial locality of reference. This implies that once a block of information (program or data) is brought into faster storage, it is very likely that it will be referenced again within a short time frame.

The question we address is if it is reasonable to assume that the synchronization requirements of a member of a process group exhibit the same type of inertial behavior; namely, if a process which has been communicating during the immediate past will continue to do so during the next slots, and conversely, if it has not been communicating, it will continue to work in isolation in the immediate future. If we can support a positive answer to this question, then we can design very simple algorithms to determine if at time t process $P_i \in P$

belongs to the TI-class or not. The analogy with demand paging can be extended, and we can use as models the page replacement algorithms.

3 Exploiting temporal locality of communication for dynamic scheduling and checkpointing

Given a process group P of size n_P, let us define two n_P bit vectors: PGC_P, Process Group Communication status vector, and PGTI_P, Process Group Temporarily Independent status vector. If $P_i \in P$ belongs to the TI-class, then $\mathrm{PGC}_P(i) = 0$ and $\mathrm{PGTI}_P(i) = 1$; if P_i does not belong to the TI-class, $\mathrm{PGC}_P(i) = 1$ and $\mathrm{PGTI}_P(i) = 0$.

Call $n_P^C(t_k)$ the number of processes $P_i \in P$ with the PGC bit on at time $t_k = k \cdot \Delta$ and $n_P^{TI}(t_k)$ the number of processes in P belonging to the TI-class. In general, $n_P^C(t_k) + n_P^{TI}(t_k) \leq n_P$. Indeed, if the state of $P_i \in P$ is given by the tuple $(\mathrm{PGC}_P(i), \mathrm{PGTI}_P(i))$ a process may be in an transient state (0,0). The full state transition diagram of process P_i is given in Figure 1.

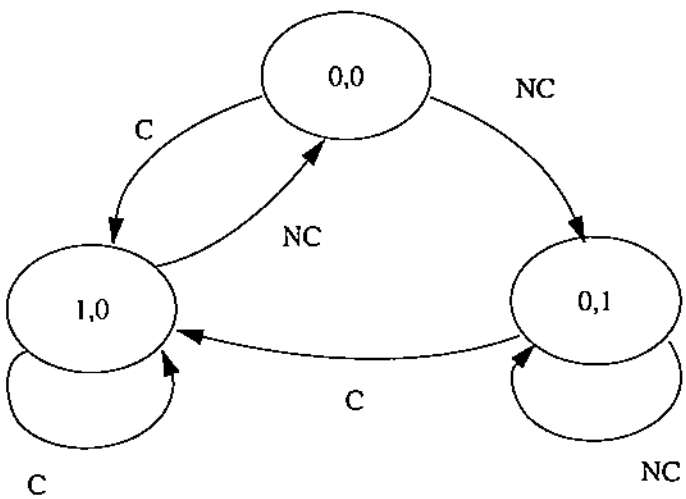


Figure 1. The state transition diagram of process $P_i \in P$ at time t_k. The time is slotted $t_k = k \cdot \Delta$. A transition in slot k is determined by the occurrence of a communication event (C) or the absence of a communication event (NC). Initially, processes start in state (0,1).

We conjecture that knowing the pair of vectors PGC and PGTI for a set of process groups $P, Q, R \ldots$, a global system scheduler could make scheduling decisions leading to a

better utilization of the resources of a system with N processing elements. One of the main drawbacks of static scheduling is *processor fragmentation.* If we call n_P, n_Q, n_R, etc., the number of processes in each of the process groups P, Q, R and so on, a static scheduler attempts to partition the system into disjoint partitions and allocate each process group to one partition, such that $n_t = n_P + n_Q + n_R + \ldots$ be as close as possible to N. But in this scheme, $N - n_t$ processing elements may end up not assigned to any partition, leading to the so-called processor fragmentation. Knowing the dynamics of synchronization for each process group provides some flexibility. Instead of being constrained to schedule all n_P processes belonging to process group P, the global scheduler may elect to schedule any number n_P^a of members of P such that $n_P^C \leq n_P^a \leq n_P$. Clearly $n_t^a = n_P^a + n_Q^a + n_R^a + \ldots$ may cover N better than n_t previously defined.

Whenever n_P^C increases because one or more processes leave their state of temporal isolation, the global scheduler needs only to discover which process group has more active processes than its current communication vector and order few local context switches.

Checkpointing is a problem of concern for massively parallel systems, yet few systems support automatic checkpointing. We contend that, knowing the two state vectors defined earlier in this section, the system scheduler could make intelligent decisions relative to checkpointing of a process group.

Clearly, checkpointing a process group at a time of intense communication, when n_P^C is close to n_P, involves a considerable overhead; there are $n_P^a (\simeq n_P^C)$ active processes' contexts to be saved and all the active pages of all n_P^a processes to be backed up. An alternative strategy is doing the checkpointing when $n_P^a \simeq n_P^C << n_P$; that is, when the only active processes are those that communicate with one another. The strategy can be achieved by following the TI-status vector and, when n_P^{TI} exceeds a certain threshold, the local schedulers force context switching for all processes $Pi \in P$ which belong to the TI-class, at the same time, checkpointing processes being released. When only communicating processes in the process group are active, the entire group is propitious for checkpointing.

4 Empirical evidence

To gather experimental evidence to confirm temporal locality of communication among members of a process group, we have monitored several parallel applications. We have instrumented the communication statements and examined the collected data. One possible approach to represent the results is to display the time elapsed between successive communication events for every thread of control. Such a representation has obvious disadvantages

since it is dependent upon the number of threads of control. We have opted to represent the dynamics of the size of the working set of the process group. If we consider a window of size Δ, for each Δ the synchronization dependencies partition the entire process group P into disjoint working sets. We represent the size of the largest working set as a function of time.

4.1 Correlation of Communication Events

For the characterization of communication patterns, we want to identify sub-groups of processes within a process group which are related to each other through communication events. To clarify the definitions, we use the following notation:

A	=	a process group; $A = \{A_1, A_2, \ldots, A_{n_A}\}$.
		a collection of communicating sub-groups; $A = \{G_1 \cup G_2 \cup \ldots \cup G_{n_G}\}$,
		for $i \neq j, G_i \cap G_j = \emptyset$
n_A	=	the size of the process group.
n_G	=	the number of clusters (sub-groups).
G_i	=	cluster i.
g_i	=	the number of elements of cluster i.

Following the definition of working set model introduced by Denning [7], we define the communication working set of process A_k at time t as:

$C_{A_k}(t, \Delta)$	=	collection of processes that communicate with process A_k during the time interval $(t - \Delta, t)$
Δ	=	working set parameter

For a given time window $[t - \Delta, t]$, $C_{A_k}(t, \Delta)$ can be obtained by looking at the communication events between process A_k and others of the same application during the time window. For example, $C_{A_k}(t, \Delta)$ can be calculated from the set of trace files containing the communication events of all processes of a parallel application.

Based on the assumption that non-overlapping communicating groups may be scheduled independently, our objective is to identify sub-groups $\{G_1, G_2, \ldots, G_{n_G}\}$ based on $C(t, \Delta)$ of each process, obtained from the trace data of program execution.

Given a set of trace files of a parallel program A executed on n_A PEs with the overall

```
t = T_0
while t < T_p {
    for each trace file k {
        select trace records belonging to [t − Δ, t] ;
        construct C_{A_k} (communication relationship map, fill M[k, ]) ;
    }
    use M to find the largest cluster of communicating processes ;
    output the elements and the size of the largest cluster ;
    t = t + Δ ;
}
```

Figure 2. Algorithm for calculating the largest communicating cluster (sub-group).

execution time T_A, we are interested in the size and the elements of the largest communicating sub-group throughout the execution from T_0 to T_A, at each interval Δ.

Let M be an $n_A \times n_A$ matrix, a communication relationship map, where each element $M[i,j]$ $(0 \leq i,j < n_A)$ can be either one or zero defined as follows:

$$M[i,j] = \begin{cases} 1 & A_i \text{ sends messages to } A_j \text{ during the time window } \Delta \\ 0 & \texttt{if} \text{ there is no communication between } A_i \text{ and } A_j \end{cases}$$

The pseudo code for finding the largest communication working set is illustrated in Figure 2. The algorithm for finding communicating sub-groups (or clusters) is similar to the algorithm in [4] for determining the connected components of an undirected graph using the disjoint-set data structure. Figure 3 adapted from [4], illustrates the pseudo code for finding connected components (clusters).

```
for each A_i ∈ A {
    do
        Make_set(A_i)                    // create a set for each A_i
}
for each M[i, j] = 1 {
    do
        if Find_set[i] ≠ Find_set[j] then
            Union(i,j)                   // unite two disjoint sets
}
```

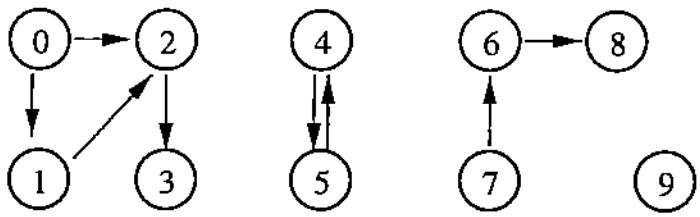
Figure 3. Procedure that uses disjoint set operations for finding connected components (clusters).

Using the *union by rank* and *path compression* heuristics presented in [4], the running time for finding connected components is almost linear in the total number of operations. (Make_set, Find_set and Union operations.) A snapshot of the computation is illustrated in Figure 4. In Figure 4(a) a directed graph with four connected components illustrates the *send* communication relationship among components. Note that, although what we have is a directed graph, the edges can be treated as undirected edges without altering the clustering algorithm. And Figure 4(b) illustrates the communication matrix $M[i,j]$ and the collection of clusters $\{G_1, G_2, \ldots, G_4\}$ after processing C_{A_k} of each PE A_k.

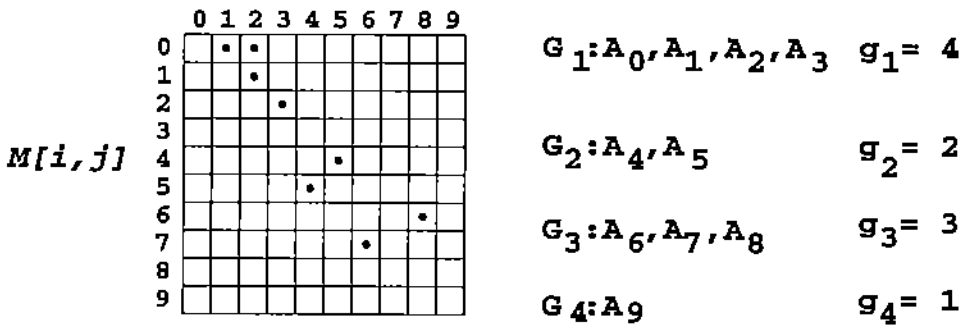
4.2 The applications

We discuss below two programs in the Molecular Replacement suite we have examined, `envelope` and `fftsynth`, and results gathered during their execution on a Paragon system [6].

The `envelope` program computes the molecular envelope of a virus. It needs as input a 3-D lattice with up to 10^9 grid points and produces a lattice of equal size as output. For every grid point, information about the electron density and a mask describing if the grid point is located in the protein, nucleic acid or solvent is provided. A spherical virus has icosahedral symmetry, and the program exploits this symmetry to get better estimates of the electron density at every grid point by calculating the average of the electron density of all points related by non-crystallographic symmetry. The program implements a shared



(a)



(b)

Figure 4. Snapshot of communication working set at $t_k = k \cdot \Delta$. There are four clusters, G_1, G_2, G_3 and G_4. The size of the largest cluster is 4. The algorithm indicates that 9 processes (0 to 8) are communicating and one is temporary independent.

virtual memory and operates in two modes, the DD mode, where the shared virtual memory resides on the external storage device, and the DC mode, where the input data is distributed over the set of compute nodes. Reference [5] describes different data management strategies for implementing a shared virtual memory. The entire data set is partitioned into *Data Allocation Units*, DAUs. The *working set* of DAU^j consists of all DAUs needed to carry out the computations associated with DAU^j. DAU *faults* occur when a compute node needs to access a DAU stored elsewhere; the penalty for a DAU fault can be significant. In the DD mode a DAU fault requires a disk access, and in the DC mode it requires access to data stored on a different compute node or on a data server node. A load balancing algorithm distributes the DAUs based upon an estimate of the amount of work associated with each of them. Both modes exploit the locality of reference and attempt to minimize the number of DAU faults by processing the DAUs assigned to each compute node to maximize the intersection of the working sets of DAUs processed in sequence.

The second program, `fftsynth`, carries out a 3-D FFT. It reads in a set of complex valued structure factors (discrete Fourier coefficients), computes the FFT and writes out the calculated electron density. A 3-D FFT is obtained by a 2-D FFT followed by a 1-D FFT in the third dimension. The algorithm requires a global exchange between phase one and phase two of the algorithm. If the amount of the combined local memory of the PEs is large enough to hold all the data, then intermediate results are exchanged through message passing among PEs, otherwise the global exchange is done using an external file.

4.3 Experimental results

We have instrumented all the communication statements of the `envelope` and `fftsynth` program. The analysis package processes the trace records and calculates the sets of communicating sub-groups for each time window of size Δ. The output of the analysis is the working set profile of a parallel application showing the largest process working set size for each time window Δ.

For example, Figures 5, 6 and 7 show the communication working set of the `envelope` program running on 64 nodes, with Δ equal to 0.2, 0.5 and 1 second, respectively. Note that these graphs derive from the same set of trace record files, but are processed with different Δ.

For an execution of a parallel application using n_A processes, the maximum communication working set of size n_A indicates that each of the n_A processes is directly or indirectly related to all others during the interval $[t-\Delta, t]$. On the other hand, a working set size equal

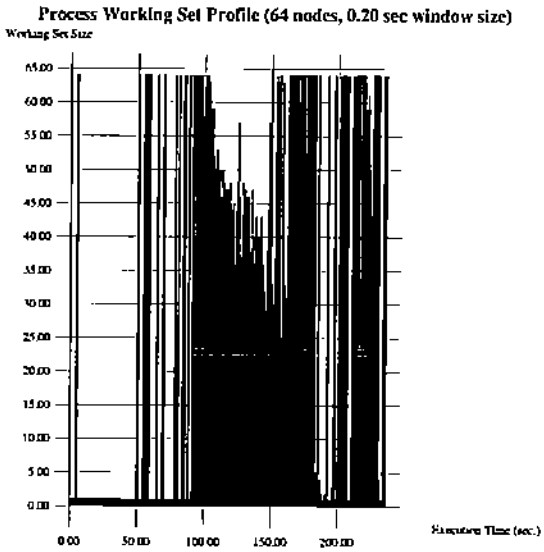


Figure 5. Envelope working set profile, $\Delta = 0.2$ seconds, for a parallel execution using 64 nodes.

to one indicates that none of the processes communicates with others during the period.

The parameter Δ plays an important role in the analysis of the processor working set. The characterization of the communication groups could provide information for a dynamic processor scheduling algorithm to make better use of computing resources. For example, by using Δ in the order of magnitude of the time necessary for a group context switch, we could observe the communication dependency among processes of a process group within the granularity of context switch time.

Figures 8, 9, and 10 show the processor working set of the `envelope` program running on 32 nodes for Δ equal to 0.2, 0.5 and 1 second respectively. From both sets of graphs, we note that the `envelope` program presents a strong communication relationship during most of its execution time (for example, the period between 80 seconds and about 170 seconds in the 64 node execution and the period between 90 seconds and 240 seconds in the 32 node execution).

The `fftsynth` program presents a communication characterization completely different from the `envelope` program. Even when the global exchange is carried out "in-place", as shown in Figures 11, 12, and 13 there is little communication dependency during the execution of the `fftsynth`. Figures 14, 15 and 16 show the working set profile of a `fftsynth` execution using external storage for the global exchange.

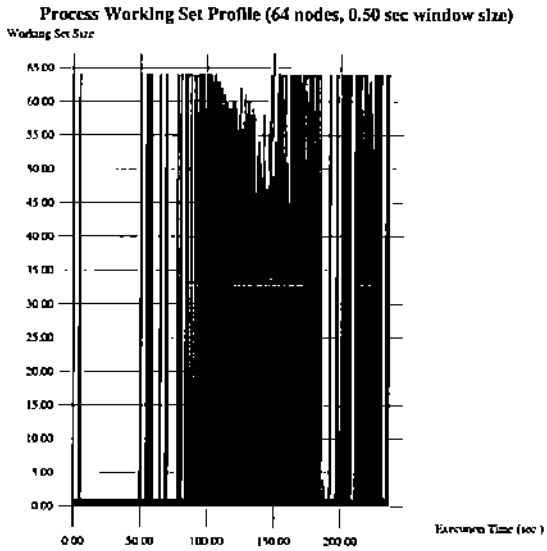


Figure 6. Envelope working set profile, $\Delta = 0.5$ seconds, for a parallel execution using 64 nodes.

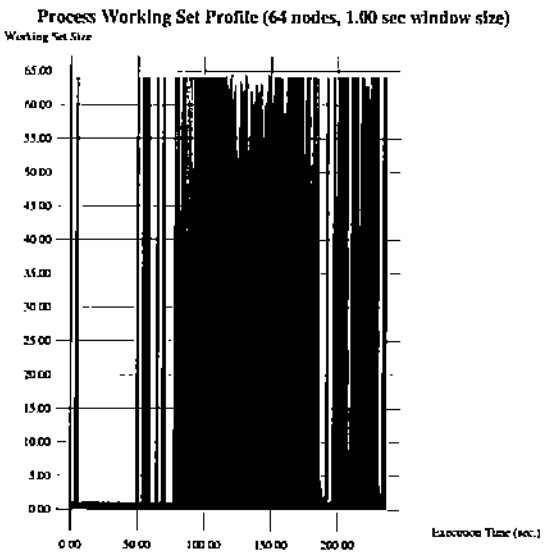


Figure 7. Envelope working set profile, $\Delta = 1$ second, for a parallel execution using 64 nodes.

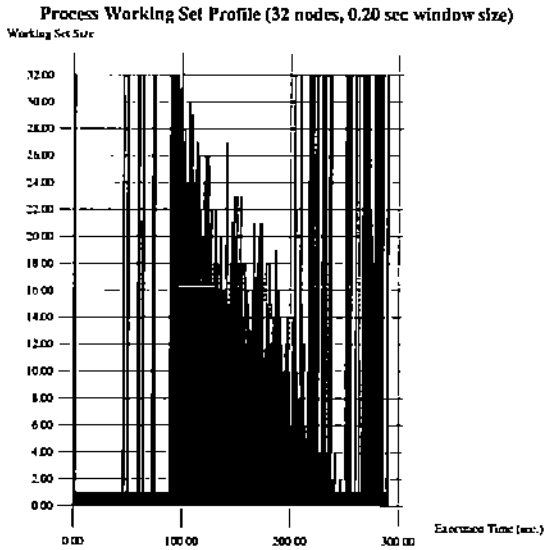


Figure 8. Envelope working set profile, $\Delta = 0.2$ seconds, for a parallel execution using 32 nodes.

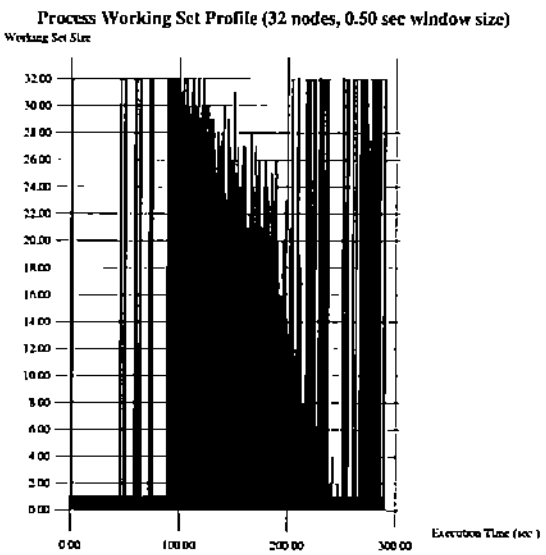


Figure 9. Envelope working set profile, $\Delta = 0.5$ seconds, for a parallel execution using 32 nodes.

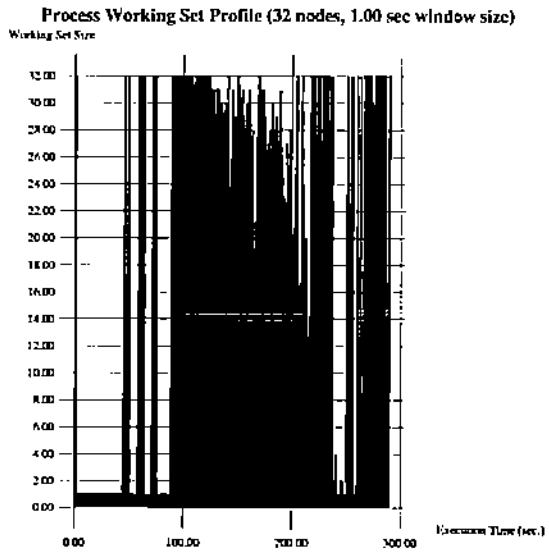


Figure 10. Envelope working set profile, $\Delta = 1$ second, for a parallel execution using 32 nodes.

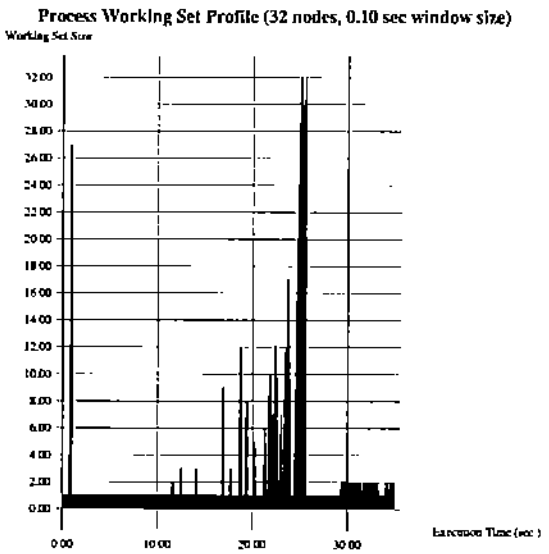


Figure 11. Fftsynth working set profile, $\Delta = 0.1$ seconds, for a parallel execution using 32 nodes and 11MB of input data.

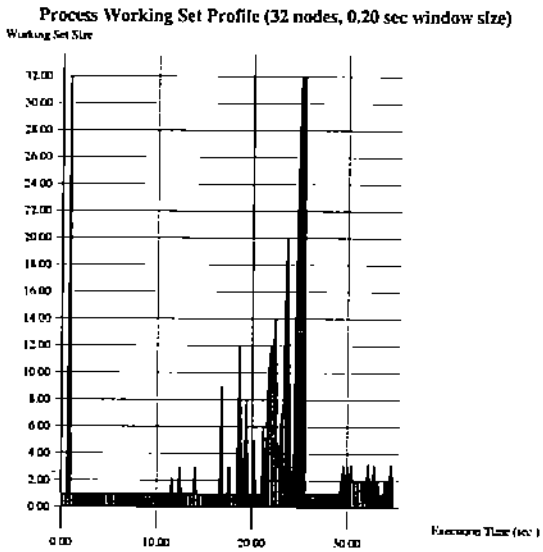


Figure 12. Fftsynth working set profile, $\Delta = 0.2$ seconds, for a parallel execution using 32 nodes and 11MB of input data.

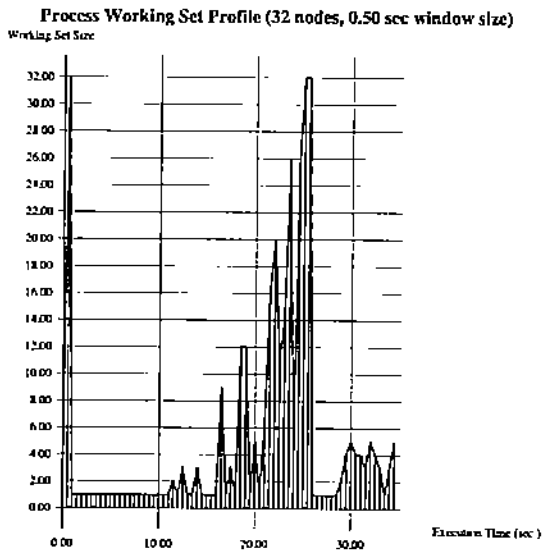


Figure 13. Fftsynth working set profile, $\Delta = 0.5$ seconds, for a parallel execution using 32 nodes and 11MB of input data.

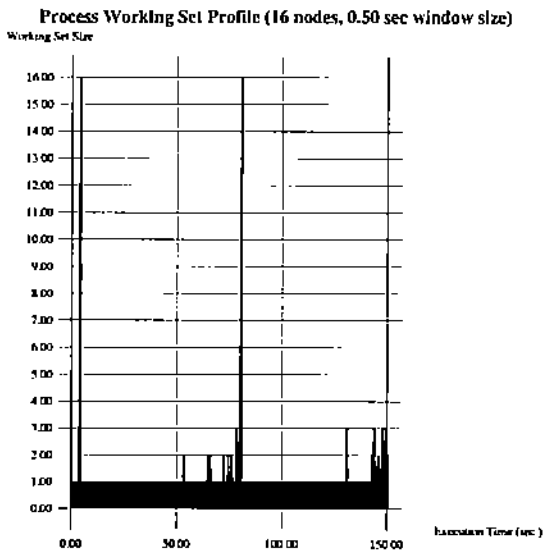


Figure 14. Fftsynth working set profile, $\Delta = 0.5$ seconds, for a parallel execution using 16 nodes and 40MB of input data.

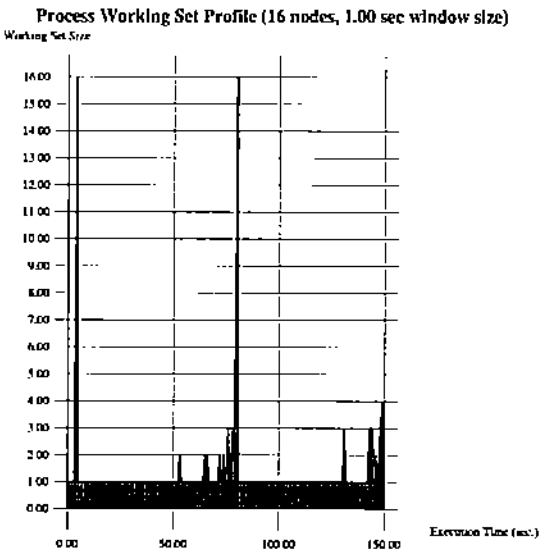


Figure 15. Fftsynth working set profile, $\Delta = 1$ second for a parallel execution using 16 nodes and 40MB of input data.

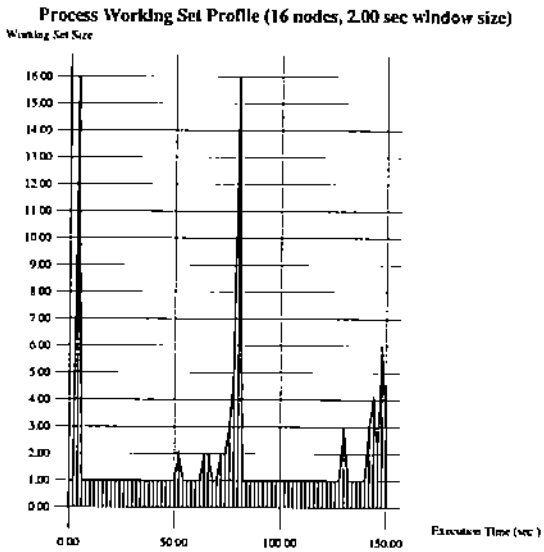


Figure 16. Fftsynth working set profile, $\Delta = 2$ seconds, for a parallel execution using 16 nodes and 40MB of input data.

4.4 Discussion of the results

The two programs we have examined exhibit very different communication patterns. The `envelope` program starts with a global communication stage followed by a relatively long period of isolation when different threads of control perform initialization tasks, and caching the input lattice across nodes. This phase ends after about 80 seconds and a period of intense communication follows until the execution completes. As expected, during this period the working set size is close to the size of the process group; different PEs use the active message facilities available on the Paragon to retrieve data stored elsewhere, as needed by the local computations.

Even for a relatively small window size (0.2 seconds) we observe that a process has a high probability of communicating with virtually all other processes in the group. This trend becomes dominant as the window size increases to 0.5 and 1.0 seconds. The same trend is observed whether 64 or 32 PEs are used.

The fftsynth exhibits quite an opposite behavior. The threads of control work in isolation until the time for the global exchange course. Again this behavior is independent upon the number of PEs.

5 Conclusions

We conjecture that sometimes processes belonging to a process group exhibit the following communication pattern: once they start communicating, they do so for some time; when they are working in isolation, they tend to maintain the state. Clearly, not all applications are expected to exhibit such a behavior at all times. Similarly, there are programs which do not exhibit locality of reference. But when process groups do exhibit locality of communication, the state of the process group provides information useful to hide the latency of I/O and paging for the individual processes in the group, and to avoid processor fragmentation. In the first case, when the process experiencing a high latency operation is temporarily independent, a local context switch to another temporarily independent process is better than busy waiting.

Processor fragmentation can be avoided, or its negative effects diminished through dynamic scheduling. Indeed, instead of being constrained to schedule all the members of a group, the system scheduler has the flexibility to schedule any number of processes larger than the kernel of n_p^C communicating processes at that time. Dynamic scheduling requires global decisions. The scheduler needs to maintain the state vectors for all process groups.

The analysis presented in Section 4 was done off line using data collected at run time. The clustering algorithm allows us to identify several clusters of processes in a process group communicating to each other. Yet, at run time, it is very expensive to identify such clusters. Using status vectors, all we know is the set of processes which are in a communication period, n_p^C. If we had the knowledge of the clusters of communicating processes, then the system (global) scheduler might be able to schedule one cluster of a process group at a time.

Last, but not least, we argue that checkpointing of a parallel program may be done more effectively using the information provided by the state vectors introduced in §3.1.

Finally, we need to answer the question of what happens if a process group does not exhibit locality of communication and the system attempts to either hide the latency by local context switches or to reduce the effects of processor fragmentation by dynamic scheduling. The answer is that the overall effect will be loss of performance through ill advised local or global context switches. But this is expected, as we already know that hierarchical storage systems have poor performance when programs do not exhibit locality of reference.

6 Acknowledgments

The authors express their thanks to Victor Abell for pointing out the importance of memory constraints for process group context switching.

References

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

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

[3] S.-H. Chiang, R. K. Mansharamani, and M. K. Vernon. Use of application characteristics and limited preemption for run-to-completion parallel processor scheduling policies. *Performance Evaluation Review*, 22(1):33–44, May 1994.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. McGraw-Hill Book Company, 1991.

[5] M. A. Cornea-Hasegan, D. C. Marinescu, and Z. Zhang. Data management for a class of iterative computations on distributed memory MIMD systems. *Concurrency: Practice and Experience*, 6(3):205–229, 1994.

[6] M. A. Cornea-Hasegan, Z. Zhang, R. E. Lynch, D. C. Marinescu, A. Hadfield, J. K. Muckelbauer, S. Munshi, L. Tong, and M. G. Rossmann. Phase refinement and extension by means of non-crystallographic symmetry averaging using parallel computers. *Acta Crystallographica*, D51:749–759, 1995.

[7] P. J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, May 1968.

[8] A. Ieumwananonthachai, A. N. Aizawa, S. R. Schwartz, B. W. Wah, and J. C. Yan. Intelligent mapping of communicating processes in distributed computing systems. In *Supercomputing '91*, pages 512–521, November 1991.

[9] S. Majumdar, D. L. Eager, and R. B. Bunt. Characterization of programs for scheduling in multiprogrammed parallel systems. *Performance Evaluation* 13(2):109–130, 1991.

[10] J. K. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of the 3rd Intl. Conf. Distributed Computing Systems*, pages 22–30, October 1982.

[11] K. C. Sevcik. Characterization of parallelism in applications and their use in scheduling. *Performance Evaluation Review*, 17:171–180, May 1989.

[12] K. C. Sevcik. Application scheduling and processor allocation in multiprogrammed parallel processing systems. *Performance Evaluation*, 19(2–3):107–140, March 1994.

[13] K. Y. Wang and D. C. Marinescu. Correlation of the paging activity of individual node programs in the SPMD execution mode. In *Proceedings of the 28th Hawaii International Conference on System Sciences, HICSS'28*, pages 61–71. IEEE Press, January 1995.