Enhancing Applications Performance on Intel Paragon through Dynamic Memory Allocation*

Subhash Saini and Horst Simon
NAS-NASA Ames Research Center, Mail Stop 258-6, Moffett Field, CA 94035-1000

Abstract

The Paragon operating system (OS) supports virtual memory (VM). The OS manages virtual memory by performing two services. Firstly, paging-in service pages the executable code from the service node to the compute nodes. This includes the paging-in of empty data corresponding to statically allocated arrays. Secondly, paging-out service is performed by paging the unused part of the OSF server to the boot node to make space available for the user's executable code. These paging-in and paging-out activities take place simultaneously and drastically degrade the performance of the user code. We have investigated this problem in detail, and found that the dynamic allocation of memory completely eliminates the unnecessary and undesirable effects of paging-in empty data arrays from the service node to the compute nodes and thereby increases the performance of the applications considered in the present work by 30% to 40%.

1: Introduction

The *Numerical Aerodynmical Simulation (NAS) Systems Division* received an *Intel Touchstone Sigma* prototype model *Paragon XP/S-15* in February 1993. It was found that performance of many applications including the assembly coded single node *BLAS 3* routine `DGEMM` [1] was lower than the performance on *Intel iPSC/860*. This finding was quite puzzling since the clock of the microprocessor *i860 XP* used in the *Paragon* is *25%* faster than the microprocessor *i860 XR* used in the *Intel iPSC/860* [2,3]. It was also found that the performance of the *NAS Parallel Benchmarks (NPB)* [4,5] is enhanced by about *30%* if they are run for second time in a `DO` loop. Furthermore, the performance of `DGEMM` was identical for the first run and the second run on a service node, but on a compute node the performance of the second run was about *40%* better than the first run. These anomalies in the performance on the *Paragon* led us to investigate the problem in more detail. This in turn led us to propose a method of dynamic allocation of memory that increases the performance of the applications by about *30%* to *40%*.

In Sec. 2 we give a brief overview of the *Paragon* system. Sec. 3 gives the description of the *BLAS 3* kernel and *NPB* used in the investigations. Section 4 discusses the allocation of memory per node for the *microkernel, OSF* server, system buffers and the amount of memory available for the user's application. Methodology for the investigations is given in Sec. 5. Based upon our numerical experiments, the formulated hypothesis is given in Sec. 7. The remedies for eliminating the paging-in of empty data arrays and thereby enhancing the performance of the applications are given in Sec. 8. Section 9 deals with the variation in performance when the application starts paging against itself. Our conclusions are drawn in Sec. 10.

2: Paragon Overview

2.1: The i860 XP microprocessor

The *Paragon* system is based on the *64* bit *i860 XP*^{TM} microprocessor [3] by Intel. The *i860 XP*^{TM} microprocessor has *2.5* million transistors in a single chip and runs at *50 MHz*. The theoretical speed is *100 MFLOPS* in *32* bit floating point and *75 MFLOPS* for *64* bit floating operations. The *i860 XP*^{TM} features *32* integer address registers with *32* bits each. It has *32* floating point registers with *32* bits each. The floating point registers can also be accessed as *16* floating point registers with *64* bits each or *8* floating point registers with *128* bits each. Each floating point register has two read ports, a write port and two-bidirectional ports. All these ports are *64* bits wide and can be used simultaneously. The floating point registers serve as input to the floating point adder and multiplier. In vector computations, these registers are used as buffers while the data cache serves as vector registers. The *i860 XP*^{TM} microprocessor has *16 KB* of instruction and *16 KB* of data caches. The data cache has a *32* bit path to the integer unit and *128* bit data path to the floating point unit. The *i860 XP*^{TM} has a number of advanced features to facilitate high execution rates. The *i860 XP*^{TM} microprocessor's floating point unit integrates single-cycle operation, *64* bit and *128* bit data paths on chip and a *128* bit data path to main memory for fast access to data and transfer of results. Floating point add, multiply and fetch from main memory are pipelined operations, and they take advantage of a three-stage pipeline to produce one result every clock for *32* bit add or multiply operations and *64* bit adds. The *64* bit multiplication takes two clocks.

2.2: NAS Intel Paragon XP/S-15

A single node of the *Paragon XP/S-15* [6] consists of two *i860 XP*^{TM} microprocessors: one for computation and the other for communication. The compute processor is for computation and the communication processor handles all message-protocol processing thus freeing the computation processor to do computations. Currently, the communication processor is *not* used in the *NAS Paragon*. Each compute processor has *16 MB* of local memory but at *NAS* only about *6 MB* is available for applications, the rest being used for the micro kernel, *OSF* server and system buffers.

The *NAS Paragon* has *256* slots for nodes. Slots are given physical node numbers from *0* through *255*. Slots are physically arranged in a rectangular grid of size *16* by *16*. There are *8* service nodes; four of them have *16 MB* of memory each and the other four have *32 MB* of memory each. Column *0* and column *14* have no physical nodes. The service partition contains *8* nodes in the last column. One of these service nodes is a boot node. This boot node has *32 MB* of memory and is connected to a *Redundant Array of Independent Disks-1 (RAID-1*). The compute partition has *208* nodes which occupy columns *1* through *13*. Compute processors are given logical numbers *0* through *207*. Compute processors are arranged in a *16* by *13* rectangular grid . The *227* nodes are arranged in a two-dimensional mesh using wormhole routing network technology. The four service nodes comprise the service partition and provide an interface to the outside world, serving as a *front end* to the *Paragon* system. Besides running jobs on the compute nodes, the service nodes run interactive jobs, such as *shells* and *editors*. They appear as one computer running *UNIX*.

Theoretical peak performance for *64* bit floating point arithmetic is *15.6 GFLOPS* for the *208* compute nodes. Hardware node-to-node bandwidth is *200 MB* per second in full duplex.

The nodes of the *NAS Paragon* are organized into groups called partitions [6]. Partitions are organized in a hierarchical structure similar to that of the *UNIX* file system. Each partition has a *pathname* in which successive levels of the tree are separated by a periods (“.”), analogous to “/” in the *UNIX* file system. A subpartition contains a subset of the nodes of the parent partition.

Currently, on the *NAS Paragon* there are no subpartitions of *.compute* or *.service*. The root partition (denoted by “.”) contains all *227* nodes of the *Paragon*. There are two subpartitions of the root partition: the compute partition, named *.compute*, contains *208* nodes to run parallel applications. The service partition, named *.service*, contains four nodes devoted to interactive jobs. The remaining eight nodes are not part of a subpartition and serve as disk controllers and are connected to the *RAID* for *I/O*. The four nodes of the service partition appear as one computer. In summary, the *NAS Paragon* system has *208* compute nodes, 3 *HiPPI* nodes, *1* boot node, *8* disk nodes, *4* service nodes of which *1* is a boot node and *4* nodes are not used at this time, for a total of *227* nodes. When a user logs onto the *Paragon*, the *shell* runs on one of the four service nodes. In the current release of the *Paragon OS*, processes do not move between service nodes to provide load balancing. However, the load leveler decides on which node a process should be started. In principle, partitions and subpartitons may overlap. For instance, there could be a subpartition called *.compute.part1* consisting of nodes *0-31* of *.compute*, and another subpartition called *.compute.part2* consisting of nodes *15-63* of *.compute*. However, in the current release of the operating system on the *NAS Paragon* , there are two problems which restrict the use of subpartitions. First, running more than one application on a node (either two jobs in the same partition or jobs in overlapping partitions) may cause the system to crash. Second, the existence of overlapping partitions sometimes causes jobs to wait when they need not. For these two reasons, there are currently no subpartitions of the *.comp*ute partition. All jobs run directly on the *.compute* partition.

2.3: Paragon Operating System

The *UNIX* operating system was originally designed for sequential computers and is not very well suited to the performance of massively parallel applications. The *Paragon* operating system is based upon two operating systems: the *Mach* system from *Carnegie Mellon University* and the *Open Software Foundation's OSF/1 AD* distributed system for multicomputers [7]. The *Paragon*'s operating system provides all the *UNIX* features including *virtual memory*; *shell*, *command*s and *utilities*; *I/O* services; and networking support for *ftp, rpc* and *NFS*. Each *Paragon* node has a small microkernel irrespective of the role of the node in the system. The *Paragon* operating system provides programming flexibility through virtual memory. In theory, virtual memory simplifies application development and porting by enabling code requiring large memory to run on a single compute node before being distributed across multiple nodes. The application runs in virtual memory which means that each process can access more memory than is physically available on each node.

3: Applications used

3.1: Basic Linear Algebra Subprograms

BLAS 1, 2 and *3* are the basic building blocks for many of scientific and engineering applications [1]. For example, the dot product is a basic kernel in *Intel's ProSolver Skyline Equation Solver (ProSolver-SES)* [8], a direct solver using skyline storage, useful for performing Finite Element Structural analysis in designing aerospace structures. *BLAS 3 (matrix-matrix)* kernels are basic kernels in *Intel's ProSolver Dense Equation Solver (ProSolver-DES)* [9], a direct solver that may be applied in solving computational electromagnetics *(CEM*) problems using *Method of Moments (MOM). BLAS 2* and *BLAS 3* are basic kernels in *LAPACK* [1]. In the present paper, we have used a *BLAS*

3 routine called `DGEMM` to compute *C = A*B*, where *A* and *B* are real general matrices. The `DGEMM` is a single node assembly coded routine and as such involves no interprocessor communication.

3.2: NAS Parallel Benchmarks

The *NPB* [4,5] were developed to evaluate the performance of highly parallel supercomputers. One of the main features of these benchmarks is their *pencil and paper* specification, which means that all details are specified algorithmically thereby avoiding many of the difficulties associated with traditional approaches to evaluating highly parallel supercomputers. The *NPB* consist of a set of eight problems each focusing on some important aspect of highly parallel supercomputing for computational aerosciences. The eight problems include five kernels and three simulated computational fluid dynamics *(CFD)* applications. The implementation of the kernels is relatively simple and straightforward and gives some insight into the general level of performance that can be expected for a given highly parallel machine. The other three simulated *CFD* applications need more effort to implement on highly parallel computers and are representative of the types of actual data movement and computation needed for computational aerosciences. In the present paper, we have used the block tridiagonal *(BT)* benchmark, which was ported from the *Intel iPSC/860* [10] to the *Paragon*. The *NPB* all involve significant interprocessor communication with the exception of the Embarrassingly Parallel *(EP)* benchmark which involves almost no interprocessor communication.

4: Distribution of memory on Paragon

The exact amount of memory available for the user's code is very hard to estimate as it depends upon several factors such as the history of the *Paragon* system since the last reboot, number of nodes, size of the system buffers set by the user at run time etc. The approximate breakdown of memory per node for the *NAS Paragon* is shown in Table 1. Memory taken by the microkernel per node on the *NAS Paragon* is bigger than claimed by *Intel* as the *NAS* is running a debugging version of the microkernel. The microkernel is the only system software component that is in the memory of each compute node at all times including its internal tables and buffers. The *OSF* server is in the memory of each compute node initially, but as pages are needed by the application unused parts of server is paged-out to the boot node. Across the whole machine the *Paragon OS* takes *2 GB* of memory out of total of *3.3 GB* of memory, thus leaving only *1.25 GB* for the user's application.

Table 1: Distribution of Memory on each NAS compute processor

Component of OS	Memory in MB
Microkernel	5
OSF Server	4
Message Buffer	1
Free Memory	6

5: Methodology

5.1: Operating System and Compiler

The *Paragon OS* used is version *R1.1*. and the *Fortran* compiler is *4.1* [11]. The compiler options used are the `f77 -O4 -Mvect -Knoieee abc.f -lkmath` [12] and the compilation was done on the service node. There is a compiler option by which one may set the size of the portion of the cache used by the vectorizer to *number* bytes. This *number* must be a multiple of *16*, and less than the cache size 16384 of the microprocessor *i860 XP*. In most cases the best results occur when *number* is set to *4096*, which is the default. In view of this we decided to choose the default size *4 KB* and the highest optimization level of *4* was used. This level of optimization generates a basic block for each *Fortran* statement and scheduling within the basic block is performed. It does perform aggressive register allocation for software pipelined loops. In addition, code for pipelined loops is scheduled several ways, with the best way selected for the assembly file. The option `-Knoieee` was used, which produces a program that flushes denormals to *0* on creation (which reduces underflow traps) and links in a math library that is not as accurate as the standard library, but offers greater performance. This library offers little or no support for exceptional data types such as *INF* and *NaN*, and will not trap on such values when encountered. If used while compiling, it tells the compiler to perform real and double precision divides using an inline divide algorithm that offers greater performance than the standard algorithm. This algorithm produces results that differ from the results specified by the *IEEE* standard by no more than three units in the last place *(ulp)*.

5.2: Procedure for 1st Run and 2nd Run

It was found that the performance of *NPB* codes is enhanced by about *30%* if they are run for second time in a `DO` loop. Furthermore, the performance of `DGEMM` was identical for the first run and second run on a service node but on a compute node the performance of the second run was about *40%* better than the first run. In our numerical results section we will present results for a first run and a second run of an application. The procedure to obtain first run and second run for a given application is illustrated in Table 2. In this table, a `DO` loop index `i` running from *1* to *2* is inserted just before the section of the code we want to

time for benchmark purposes. In this table the *first run* corresponds to *i=1* and the *second run* corresponds to *i=2* as shown in Table 2. The overhead in calling the function DCLOCK was estimated to be about $1.5x10^{-6}$second.

Table 2: Procedure for obtaining first run and second run

```
PROGRAM abc
...
DO i = 1, 2
t0 = DCLOCK()
t1 = DCLOCK
CALL DGEMM( ,..., ...)
t2 = DCLOCK()
time = t2 - (t1 - t0)
ENDDO
...
END
```

5.3: Compiling and linking on the partitions

The *Paragon* system has two types of partitions: (a) a *service partition* and (b) a *compute partition*. The partition where an application runs can be specified when you compile and execute it. The *-nx* switch defines the preprocessor symbol *_NODE* and links with the *nx* library *libnx.a* [11]. It also links with the start-up routine—the controlling process that runs in the service partition and starts up the application in the compute partition. Commands to run the application on service partition and compute partition are given in Table 3.

Table 3: Compiling and executing on Mesh

Service Partition	Compute Partition
> f77 prog.f > a.out	> f77 prog.f -nx > a.out -sz 1 > a.out -sz 64

5.4: Numerical experiments

The following two numerical experiments were performed:

(a) First experiment: Experiment in which no interprocessor communication is involved and only communication is due to the paging-in of executable code from the service node to the compute node and if memory requirement exceeds *6 MB* per node, then paging-out of the unused part of the *OSF* server from the compute nodes to the boot node. The single node *BLAS 3* routine DGEMM was used.

(b) Second experiment: Experiment in which interprocessor communication is involved in addition to the communication due to paging-in and paging-out. The *BT* application from the *NPB* was used.

6: Results

Fig 1(a) shows the results for the assembly coded *BLAS 3* routine DGEMM on a service node obtained for the first run and the second run. Notice that on the service node the results for first run and second run are identical. The routine DGEMM is a single node routine and as such involves no interprocessor communication.

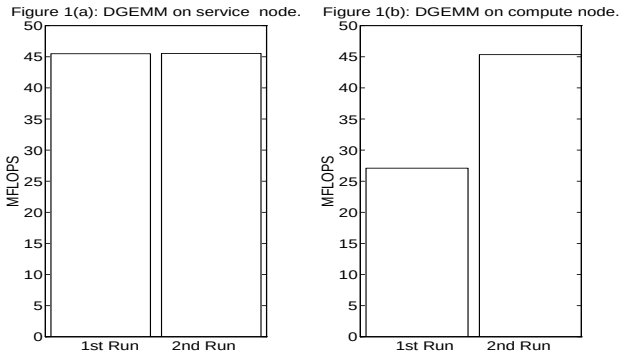
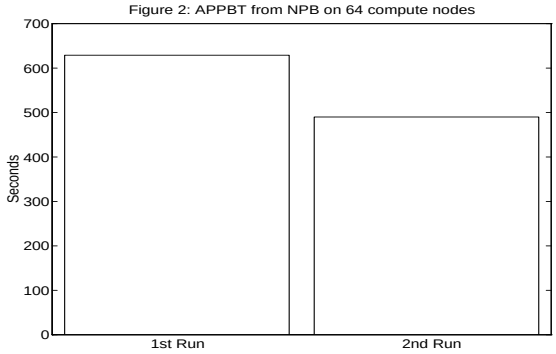


Fig 1(b) shows the results for the assembly coded *BLAS 3* on a compute node for the first run and second run. The performance of the DGEMM is *27 MFLOP*S for the first run and *45 MFLOPS* for the second run. The performance obtained by the second run is about *40%* better than the performance by the first run.

Figure 2 shows the performance of the *BT*. The *BT* code used is an *Intel iPSC/860* version which was ported to the *NAS Paragon*. Timings reported for the *BT* in Figure 2 are according to the *NPB* rules. The first run takes *629* seconds whereas the second runs takes *490* seconds. The performance of the second run is about *30%* better than the first run.



It is clear from the Figures 1-2 that the performance of the second run is about 30% to 40% higher than the first run. This degradation in the performance for the first run is not acceptable since users will always run their code once.

Figure 3 shows the performance of DGEMM on two compute nodes , i.e. on node *0* and *node 1*. The function gsync was used to synchronize all node processes. The function gsync [6] performs a global synchronization operation. When a node process calls the gsync() func-

tion, it waits until all other nodes in the application call `gsync()` before continuing. All nodes in the application must call `gsync` before any node in the application can continue. The *MFLOP*S rate shown in Figures 3-8 are for the *first* run. The performance has decreased from *27 MFLOPS* to *22 MFLOPS*.

Figure 4 shows the performance of `DGEMM` on four compute nodes. The performance has further degraded to an average of about *13 MFLOPS* except for node *1* on which it is about *18 MFLOPS*. The reason for relatively better performance on node *1* than on nodes *0, 2* and *3* is that node *1* happens to be the first node to receive the empty data arrays from the service node.

Figure 4: DGEMM on four compute nodes.

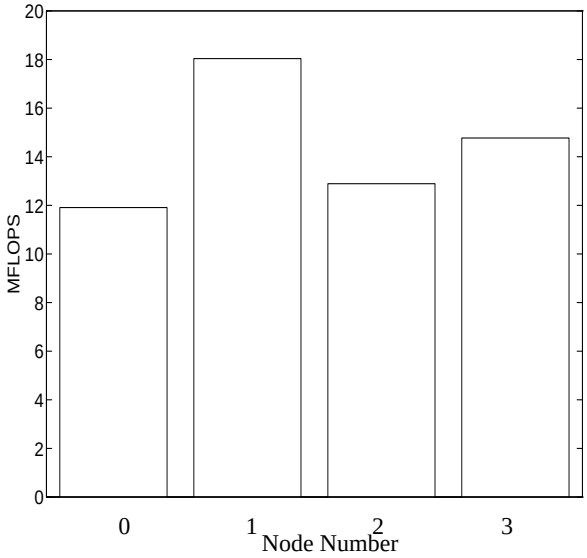


Figure 5 shows the performance of `DGEMM` on eight compute nodes. The performance has degraded further to an average of *7 MFLOPS*. The performance on node *0* is relatively better than the rest of the nodes.

Figure 6 shows the performance of `DGEMM` on sixteen nodes. Here the average performance has been further decreased to about *6 MFLOPS*. The performance on at least one node (node *0*) is *21 MFLOPS*, much better than the rest of the nodes.

Figure 7 shows the performance of `DGEMM` on *32* compute nodes. Here the average performance has further decreased to about *3 MFLOPS*. Unlike in Figures 4-6, here the performance on two compute nodes (node *25* and node *29*) is relatively better than on the rest of the nodes.

Figure 8 shows the performance of DGEMM on 64 compute nodes. Increasing the number of nodes from 32 to 64 has further degraded the performance. Here the performance on nodes *0*, *26* and *33* is much better compared to the rest of the nodes.

7: Hypothesis

It is clear from Figures 2-8 that as we increase the number of nodes, the performance decreases from *27 MFLOPS* to *1.5 MFLOPS* for the first run. The unused part of the *OSF* server must be paged out to the boot node, whenever the memory requirement is more than about *6 MB*, to provide space for the arrays *A, B* and *C* in the program on the compute node. While the paging-out of the unused part of the *OSF* server is going on, pages containing arrays *A, B* and *C* are being paged-in from the service node to each of the compute nodes. These paging-in and paging-out activities take place simultaneously at the first reference and use of the arrays *A, B* and *C* and *not* at the DIMENSION statement in the program. The net result is that the simultaneous paging-in and paging-out creates additional traffic in the network that increases with the increasing number of compute nodes.

8: Remedies for eliminating paging-in of empty data arrays

8.1: Locking the memory at run time

The activity of paging-in can be removed by using the run time option -plk [6] which causes all of the application's pages to be brought at the start of execution and to remain resident. The results of doing this are shown in Figure 9. The performance on each compute node is *45 MFLOP*S in the first run.

The run time option -plk was tried on codes of different sizes and we found that for a code that needs about *7 MB* per compute node, the time for loading the code from the service node on to the compute node became very large. In many cases load time became so large that we had to terminate the process of loading. From our experience we found that the run time option -plk is not a solution for codes which need more than *7 MB* of memory per node.

8.2: Dynamic Allocation of memory

The dynamic allocation of memory can be performed in a number of ways. The best method is to use the ALLOCATE statement [13]. The ALLOCATE statement allocates storage for each pointer-based variable and allocatable common block which appears in the statement. The DEALLOCATE statement causes the memory allocated for each pointer-based variable or allocatable COMMON block that appears in the statement to be deallocated. Fortunately both ALLOCATE and DEALLOCATE are available [13]. A dynamic or allocatable COMMON block is a common block whose storage is not allocated until an explicit ALLOCATE statement is executed. The syntax of statements ALLOCATE and DEALLOCATE is given in Table 4 and their use in *Appendix A*.

Table 4: Syntax for using ALLOCATE and DEALLOCATE statements

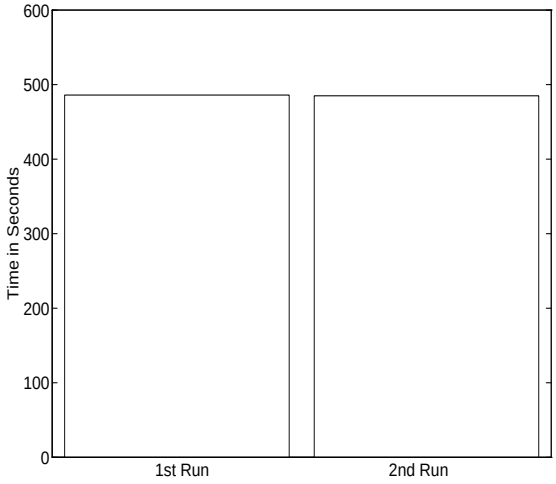
ALLOCATE (name[,name]...[,STAT= var])
DEALLOCATE (a1,[,a1]...[,STAT= var])
name is a pointer-based variable or the name of an allocatable COMMON block enclosed in slashes
a1 is a pointer-based variable or the name of an allocatable COMMON block enclosed in slashes.
var is an integer variable, integer array element, or integer member of a structure.

We have found that the most effective and elegant method of removing the undesirable and unnecessary paging-in of empty arrays provided by the *Paragon OS* is to use dynamic allocation of memory inside the application. The dynamic allocation of memory creates the arrays *A, B*

and *C* at run time on the compute processors rather than at compile time on the service node. The static allocation of memory creates the arrays *A*, *B* and *C* at compile time and at run time they are paged-in to the compute processors as and when they are first referenced and used. The performance of `DGEMM` using dynamic allocation of memory is shown in Figure 10. The dynamic memory allocation removes a serializing bottleneck and communication overhead.

The results for *BT* using dynamic allocation of memory are shown in Figure 11. We find that dynamic allocation of memory increases the performance of *BT* by *29%* and gives the correct performance in the first run.

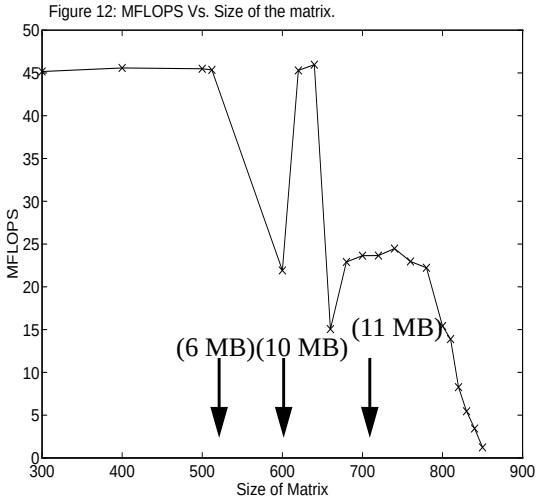
Figure 11: BT using dynamic allocation of memory



9: Paging of application against itself

The performance of `DGEMM` as a function of the size of the matrix is shown in Figure 12. When the size of the matrix is *512x512* it needs about *6.3 MB* of memory per node. As we increase the size of the matrix, initially performance goes down as the unused part of the *OSF* server is being paged-out from the compute node to the boot node to make space available for the application. As we further increase the size of the matrix, a limit is reached at about *10 MB* beyond which none of the *OSF* server is left to be paged-out and the application starts paging-out against itself. The effect of paging an application against itself is clearly seen at about *11 MB* when the performance of the `DGEMM` goes down to about *1 MFLOPS*.

Figure 12: MFLOPS Vs. Size of the matrix.



10: Conclusions

(1) Paging-in of data (empty arrays) during execution time degrades the performance of the application and should be avoided. This service performed by the *Paragon* operating system is unnecessary and is undesirable.

(2) One can use the run-time option `-plk` to lock the memory to resolve the problem. However, the use of the `-plk` option enormously increases the load time if the memory required by the application is about *7 MB* or higher per node. A genuine remedy for unnecessary effects of paging is to use dynamic allocation of memory using `ALLOCATE` and `DEALLOCATE` statements [13]. On *NAS Paragon* or any other *Paragon* system, irrespective of the memory requirement of the application, dynamic allocation of memory should **ALWAYS** be used to eliminate the service of paging-in of empty data arrays from the service node to the compute processor. The use of dynamic allocation of memory increases the performance of applications, considered in the present work, by 30% to 40%.

(3) The performance of the application starts decreasing when the application starts paging-out and ultimately it becomes unacceptable. On the *NAS Paragon*, after *10 MB*, the application starts paging against itself.

(4) The use of virtual memory by the *OSF/1* AD operating system has been a major drawback to the performance of the *Paragon*. The large amount of memory required by *OSF/1 AD* reduces available user memory to about *6 MB* per compute processor. This is a step back-

ward from the roughly *8 MB* per node memory available to the user on the *Intel iPSC/860*. Using the virtual memory system can lead to a significant drop in performance, and to other not very transparent performance variations, which make the machine less predictable for the user. These variations and the lack of memory could be tolerated as a price for increased system stability and ease of use. However, the promise of using *OSF/1 AD* for more reliable production operation has not yet materialized. This may change over time in favor of the *Paragon*.

(5) For any robust architecture and operating system, the performance of the applications should not change whether they are run with static allocation of memory or dynamic allocation of memory. On the *Paragon* system, the performance of the applications is considerably higher *(30%* to *40%* in the present paper) if dynamic allocation of memory rather than static allocation of memory is used.

In summary, there are still major challenges ahead for the *Paragon* compilers and systems software. *Intel* is aware of the problem but so far it has not been documented anywhere, including the latest *Release Notes 1.1* [15, 16].

Acknowledgment: One of the authors (SS) gratefully acknowledges many discussions with David McNab, Bernard Traversat, William J. Nitzberg, Thanh Phung, Art Lazanoff, and Todd F. Churchill.

***** The authors are employees of *Computer Sciences Corporation*. The work is supported through *NASA* contract *NAS2-12961*.

References

[1] E. Anderson et al., *LAPACK Users' Guide*, SIAM, Philadelphia, 1992.

[2] *Overview of the i860™ XR Supercomputing Microprocessor*, 1990, Intel Corporation.

[3] *Overview of the i860™ XP Supercomputing Microprocessor*, 1991, Intel Corporation.

[4] D. Bailey et al., eds, *The NAS Parallel Benchmarks*, Technical Report RNR-91-02, NAS Ames Research Center, Moffet Field, California, 1991.

[5] D. Bailey et al., *The NAS Parallel Benchmark Results*, IEEE Parallel & Distributed technology, 43-51, February 1993.

[6] *PARAGON™ OSF/1, User's Guide*, Intel Corporation, April 1993.

[7] *OSF/1™ Operating System User's Guide*, Revision 1.0, Prentice Hall Englewood, New Jersey, 1992.

[8] *iPSC/860 ProSolver™-SES Manual*, May, 1991, Intel Corporation.

[9] i*PSC/860 ProSolver™-DES Manual*, March 1992, Intel Corporation.

[10] *Intel iPSC/860 User's Guide*, April 1993

[11] *PARAGON™ OSF/1 Fortran Compiler User's Guide*, Intel Corporation, April 1993.

[12] *CLASSPACK, Basic Math Library User's Guide*, Kuck & Associates, Release 1.3, 1992.

[13] *PARAGON™ OSF/1, Fortran Language Reference Manual*, April 1993, Intel Corporation.

[14] *Proceedings of Intel Supercomputing User's Group Meeting*, Oct. 3-6, 1993, St. Louis, Missouri

[15] *Paragon System Software Release 1.1, Release Notes for the Paragon XP/S System*, October 1993, Intel Corporation.

[16] Thanh Phung, private communication, Nov. 1993.

APPENDIX A

Dynamic Allocation of Memory in Fortran

Figure 13 shows the Fortran program with static allocation of memory and Figure 14 shows a modified progam with dynamic allocation of memory.

Figure 13: Fortran program with static allocation of memory.

```
PROGRAM abc
...
REAL a(512), b(512), c(512), x(1024)
COMMON /block1/ x
...
call sub1(a,b,c)
...
END
```

Figure 14: Fortran program with dynamic allocation of memory.

```
PROGRAM abc
PARAMETER (n1=512, n2=1024)
...
REAL a(n1), b(n1), c(n1), x(n2)
POINTER (p1, a)
POINTER (p2, b)
POINTER (p3, c)
COMMON, ALLOCATABLE /block1/ x
...
ALLOCATE (a, STAT = isa)
ALLOCATE (b, STAT = isb)
ALLOCATE (c, STAT = isc)
ALLOCATE (/block1/, STAT = isblk1)
...
CALL sub1(a,b,c)
...
DEALLOCATE(a)
DEALLOCATE(b)
DEALLOCATE(c)
DEALLOCATE(/block1/)
END
```

Enhancing Applications Performance on Intel Paragon through Dynamic Memory Allocation*

Subhash Saini and Horst Simon
NAS-NASA Ames Research Center, Mail Stop 258-6, Moffett Field, CA 94035-1000

Abstract

The Paragon operating system (OS) supports virtual memory (VM). The OS manages virtual memory by performing two services. Firstly, paging-in service pages the executable code from the service node to the compute nodes. This includes the paging-in of empty data corresponding to statically allocated arrays. Secondly, paging-out service is performed by paging the unused part of the OSF server to the boot node to make space available for the user's executable code. These paging-in and paging-out activities take place simultaneously and drastically degrade the performance of the user code. We have investigated this problem in detail, and found that the dynamic allocation of memory completely eliminates the unnecessary and undesirable effects of paging-in empty data arrays from the service node to the compute nodes and thereby increases the performance of the applications considered in the present work by 30% to 40%.

1: Introduction

The *Numerical Aerodynmical Simulation (NAS) Systems Division* received an *Intel Touchstone Sigma* prototype model *Paragon XP/S-15* in February 1993. It was found that performance of many applications including the assembly coded single node *BLAS 3* routine `DGEMM` [1] was lower than the performance on *Intel iPSC/860*. This finding was quite puzzling since the clock of the microprocessor *i860 XP* used in the *Paragon* is *25%* faster than the microprocessor *i860 XR* used in the *Intel iPSC/860* [2,3]. It was also found that the performance of the *NAS Parallel Benchmarks (NPB)* [4,5] is enhanced by about *30%* if they are run for second time in a `DO` loop. Furthermore, the performance of `DGEMM` was identical for the first run and the second run on a service node, but on a compute node the performance of the second run was about *40%* better than the first run. These anomalies in the performance on the *Paragon* led us to investigate the problem in more detail. This in turn led us to propose a method of dynamic allocation of memory that increases the performance of the applications by about *30%* to *40%*.

In Sec. 2 we give a brief overview of the *Paragon* system. Sec. 3 gives the description of the *BLAS 3* kernel and *NPB* used in the investigations. Section 4 discusses the allocation of memory per node for the *microkernel, OSF* server, system buffers and the amount of memory available for the user's application. Methodology for the investigations is given in Sec. 5. Based upon our numerical experiments, the formulated hypothesis is given in Sec. 7. The remedies for eliminating the paging-in of empty data arrays and thereby enhancing the performance of the applications are given in Sec. 8. Section 9 deals with the variation in performance when the application starts paging against itself. Our conclusions are drawn in Sec. 10.

2: Paragon Overview

2.1: The i860 XP microprocessor

The *Paragon* system is based on the *64* bit *i860 XP*^{TM} microprocessor [3] by Intel. The *i860 XP*^{TM} microprocessor has *2.5* million transistors in a single chip and runs at *50 MHz*. The theoretical speed is *100 MFLOPS* in *32* bit floating point and *75 MFLOPS* for *64* bit floating operations. The *i860 XP*^{TM} features *32* integer address registers with *32* bits each. It has *32* floating point registers with *32* bits each. The floating point registers can also be accessed as *16* floating point registers with *64* bits each or *8* floating point registers with *128* bits each. Each floating point register has two read ports, a write port and two-bidirectional ports. All these ports are *64* bits wide and can be used simultaneously. The floating point registers serve as input to the floating point adder and multiplier. In vector computations, these registers are used as buffers while the data cache serves as vector registers. The *i860 XP*^{TM} microprocessor has *16 KB* of instruction and *16 KB* of data caches. The data cache has a *32* bit path to the integer unit and *128* bit data path to the floating point unit. The *i860 XP*^{TM} has a number of advanced features to facilitate high execution rates. The *i860 XP*^{TM} microprocessor's floating point unit integrates single-cycle operation, *64* bit and *128* bit data paths on chip and a *128* bit data path to main memory for fast access to data and transfer of results. Floating point add, multiply and fetch from main memory are pipelined operations, and they take advantage of a three-stage pipeline to produce one result every clock for *32* bit add or multiply operations and *64* bit adds. The *64* bit multiplication takes two clocks.

2.2: NAS Intel Paragon XP/S-15

A single node of the *Paragon XP/S-15* [6] consists of two *i860 XP*^{TM} microprocessors: one for computation and the other for communication. The compute processor is for computation and the communication processor handles all message-protocol processing thus freeing the computation processor to do computations. Currently, the communication processor is *not* used in the *NAS Paragon*. Each compute processor has *16 MB* of local memory but at *NAS* only about *6 MB* is available for applications, the rest being used for the micro kernel, *OSF* server and system buffers.

The *NAS Paragon* has *256* slots for nodes. Slots are given physical node numbers from *0* through *255*. Slots are physically arranged in a rectangular grid of size *16* by *16*. There are *8* service nodes; four of them have *16 MB* of memory each and the other four have *32 MB* of memory each. Column *0* and column *14* have no physical nodes. The service partition contains *8* nodes in the last column. One of these service nodes is a boot node. This boot node has *32 MB* of memory and is connected to a *Redundant Array of Independent Disks-1 (RAID-1*). The compute partition has *208* nodes which occupy columns *1* through *13*. Compute processors are given logical numbers *0* through *207*. Compute processors are arranged in a *16* by *13* rectangular grid . The *227* nodes are arranged in a two-dimensional mesh using wormhole routing network technology. The four service nodes comprise the service partition and provide an interface to the outside world, serving as a *front end* to the *Paragon* system. Besides running jobs on the compute nodes, the service nodes run interactive jobs, such as *shells* and *editors*. They appear as one computer running *UNIX*.

Theoretical peak performance for *64* bit floating point arithmetic is *15.6 GFLOPS* for the *208* compute nodes. Hardware node-to-node bandwidth is *200 MB* per second in full duplex.

The nodes of the *NAS Paragon* are organized into groups called partitions [6]. Partitions are organized in a hierarchical structure similar to that of the *UNIX* file system. Each partition has a *pathname* in which successive levels of the tree are separated by a periods (“.”), analogous to “/” in the *UNIX* file system. A subpartition contains a subset of the nodes of the parent partition.

Currently, on the *NAS Paragon* there are no subpartitions of *.compute* or *.service*. The root partition (denoted by “.”) contains all *227* nodes of the *Paragon*. There are two subpartitions of the root partition: the compute partition, named *.compute*, contains *208* nodes to run parallel applications. The service partition, named *.service*, contains four nodes devoted to interactive jobs. The remaining eight nodes are not part of a subpartition and serve as disk controllers and are connected to the *RAID* for *I/O*. The four nodes of the service partition appear as one computer. In summary, the *NAS Paragon* system has *208* compute nodes, 3 *HiPPI* nodes, *1* boot node, *8* disk nodes, *4* service nodes of which *1* is a boot node and *4* nodes are not used at this time, for a total of *227* nodes. When a user logs onto the *Paragon*, the *shell* runs on one of the four service nodes. In the current release of the *Paragon OS*, processes do not move between service nodes to provide load balancing. However, the load leveler decides on which node a process should be started. In principle, partitions and subpartitons may overlap. For instance, there could be a subpartition called *.compute.part1* consisting of nodes *0-31* of *.compute*, and another subpartition called *.compute.part2* consisting of nodes *15-63* of *.compute*. However, in the current release of the operating system on the *NAS Paragon* , there are two problems which restrict the use of subpartitions. First, running more than one application on a node (either two jobs in the same partition or jobs in overlapping partitions) may cause the system to crash. Second, the existence of overlapping partitions sometimes causes jobs to wait when they need not. For these two reasons, there are currently no subpartitions of the *.comp*ute partition. All jobs run directly on the *.compute* partition.

2.3: Paragon Operating System

The *UNIX* operating system was originally designed for sequential computers and is not very well suited to the performance of massively parallel applications. The *Paragon* operating system is based upon two operating systems: the *Mach* system from *Carnegie Mellon University* and the *Open Software Foundation's OSF/1 AD* distributed system for multicomputers [7]. The *Paragon*'s operating system provides all the *UNIX* features including *virtual memory*; *shell*, *command*s and *utilities*; *I/O* services; and networking support for *ftp, rpc* and *NFS*. Each *Paragon* node has a small microkernel irrespective of the role of the node in the system. The *Paragon* operating system provides programming flexibility through virtual memory. In theory, virtual memory simplifies application development and porting by enabling code requiring large memory to run on a single compute node before being distributed across multiple nodes. The application runs in virtual memory which means that each process can access more memory than is physically available on each node.

3: Applications used

3.1: Basic Linear Algebra Subprograms

BLAS 1, 2 and *3* are the basic building blocks for many of scientific and engineering applications [1]. For example, the dot product is a basic kernel in *Intel's ProSolver Skyline Equation Solver (ProSolver-SES)* [8], a direct solver using skyline storage, useful for performing Finite Element Structural analysis in designing aerospace structures. *BLAS 3 (matrix-matrix)* kernels are basic kernels in *Intel's ProSolver Dense Equation Solver (ProSolver-DES)* [9], a direct solver that may be applied in solving computational electromagnetics *(CEM*) problems using *Method of Moments (MOM). BLAS 2* and *BLAS 3* are basic kernels in *LAPACK* [1]. In the present paper, we have used a *BLAS*

3 routine called `DGEMM` to compute *C = A*B*, where *A* and *B* are real general matrices. The `DGEMM` is a single node assembly coded routine and as such involves no interprocessor communication.

3.2: NAS Parallel Benchmarks

The *NPB* [4,5] were developed to evaluate the performance of highly parallel supercomputers. One of the main features of these benchmarks is their *pencil and paper* specification, which means that all details are specified algorithmically thereby avoiding many of the difficulties associated with traditional approaches to evaluating highly parallel supercomputers. The *NPB* consist of a set of eight problems each focusing on some important aspect of highly parallel supercomputing for computational aerosciences. The eight problems include five kernels and three simulated computational fluid dynamics *(CFD)* applications. The implementation of the kernels is relatively simple and straightforward and gives some insight into the general level of performance that can be expected for a given highly parallel machine. The other three simulated *CFD* applications need more effort to implement on highly parallel computers and are representative of the types of actual data movement and computation needed for computational aerosciences. In the present paper, we have used the block tridiagonal *(BT)* benchmark, which was ported from the *Intel iPSC/860* [10] to the *Paragon*. The *NPB* all involve significant interprocessor communication with the exception of the Embarrassingly Parallel *(EP)* benchmark which involves almost no interprocessor communication.

4: Distribution of memory on Paragon

The exact amount of memory available for the user's code is very hard to estimate as it depends upon several factors such as the history of the *Paragon* system since the last reboot, number of nodes, size of the system buffers set by the user at run time etc. The approximate breakdown of memory per node for the *NAS Paragon* is shown in Table 1. Memory taken by the microkernel per node on the *NAS Paragon* is bigger than claimed by *Intel* as the *NAS* is running a debugging version of the microkernel. The microkernel is the only system software component that is in the memory of each compute node at all times including its internal tables and buffers. The *OSF* server is in the memory of each compute node initially, but as pages are needed by the application unused parts of server is paged-out to the boot node. Across the whole machine the *Paragon OS* takes *2 GB* of memory out of total of *3.3 GB* of memory, thus leaving only *1.25 GB* for the user's application.

Table 1: Distribution of Memory on each NAS compute processor

Component of OS	Memory in MB
Microkernel	5
OSF Server	4
Message Buffer	1
Free Memory	6

5: Methodology

5.1: Operating System and Compiler

The *Paragon OS* used is version *R1.1*. and the *Fortran* compiler is *4.1* [11]. The compiler options used are the `f77 -O4 -Mvect -Knoieee abc.f -lkmath` [12] and the compilation was done on the service node. There is a compiler option by which one may set the size of the portion of the cache used by the vectorizer to *number* bytes. This *number* must be a multiple of *16*, and less than the cache size 16384 of the microprocessor *i860 XP*. In most cases the best results occur when *number* is set to *4096*, which is the default. In view of this we decided to choose the default size *4 KB* and the highest optimization level of *4* was used. This level of optimization generates a basic block for each *Fortran* statement and scheduling within the basic block is performed. It does perform aggressive register allocation for software pipelined loops. In addition, code for pipelined loops is scheduled several ways, with the best way selected for the assembly file. The option `-Knoieee` was used, which produces a program that flushes denormals to *0* on creation (which reduces underflow traps) and links in a math library that is not as accurate as the standard library, but offers greater performance. This library offers little or no support for exceptional data types such as *INF* and *NaN*, and will not trap on such values when encountered. If used while compiling, it tells the compiler to perform real and double precision divides using an inline divide algorithm that offers greater performance than the standard algorithm. This algorithm produces results that differ from the results specified by the *IEEE* standard by no more than three units in the last place *(ulp)*.

5.2: Procedure for 1st Run and 2nd Run

It was found that the performance of *NPB* codes is enhanced by about *30%* if they are run for second time in a `DO` loop. Furthermore, the performance of `DGEMM` was identical for the first run and second run on a service node but on a compute node the performance of the second run was about *40%* better than the first run. In our numerical results section we will present results for a first run and a second run of an application. The procedure to obtain first run and second run for a given application is illustrated in Table 2. In this table, a `DO` loop index `i` running from *1* to *2* is inserted just before the section of the code we want to

time for benchmark purposes. In this table the *first run* corresponds to *i=1* and the *second run* corresponds to *i=2* as shown in Table 2. The overhead in calling the function DCLOCK was estimated to be about $1.5x10^{-6}$second.

Table 2: Procedure for obtaining first run and second run

```
PROGRAM abc
...
DO i = 1, 2
t0 = DCLOCK()
t1 = DCLOCK
CALL DGEMM( ,..., ...)
t2 = DCLOCK()
time = t2 - (t1 - t0)
ENDDO
...
END
```

5.3: Compiling and linking on the partitions

The *Paragon* system has two types of partitions: (a) a *service partition* and (b) a *compute partition*. The partition where an application runs can be specified when you compile and execute it. The *-nx* switch defines the preprocessor symbol *_NODE* and links with the *nx* library *libnx.a* [11]. It also links with the start-up routine—the controlling process that runs in the service partition and starts up the application in the compute partition. Commands to run the application on service partition and compute partition are given in Table 3.

Table 3: Compiling and executing on Mesh

Service Partition	Compute Partition
> f77 prog.f > a.out	> f77 prog.f -nx > a.out -sz 1 > a.out -sz 64

5.4: Numerical experiments

The following two numerical experiments were performed:

(a) First experiment: Experiment in which no interprocessor communication is involved and only communication is due to the paging-in of executable code from the service node to the compute node and if memory requirement exceeds *6 MB* per node, then paging-out of the unused part of the *OSF* server from the compute nodes to the boot node. The single node *BLAS 3* routine DGEMM was used.

(b) Second experiment: Experiment in which interprocessor communication is involved in addition to the communication due to paging-in and paging-out. The *BT* application from the *NPB* was used.

6: Results

Fig 1(a) shows the results for the assembly coded *BLAS 3* routine DGEMM on a service node obtained for the first run and the second run. Notice that on the service node the results for first run and second run are identical. The routine DGEMM is a single node routine and as such involves no interprocessor communication.

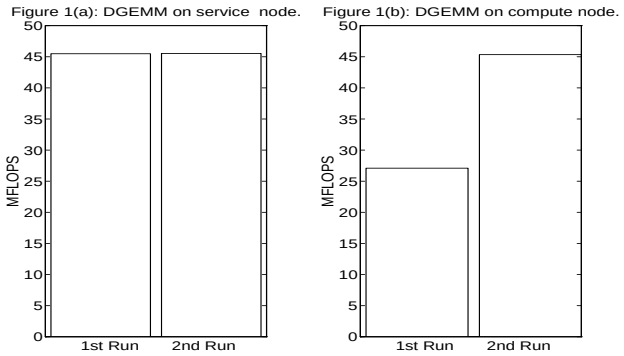
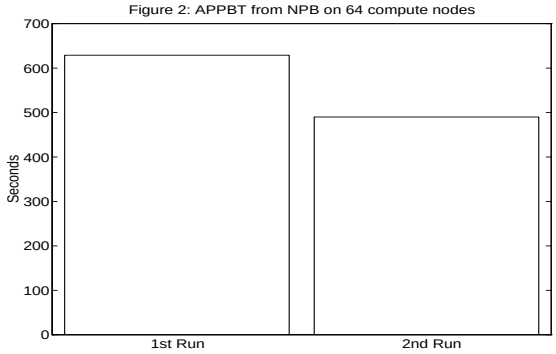


Fig 1(b) shows the results for the assembly coded *BLAS 3* on a compute node for the first run and second run. The performance of the DGEMM is *27 MFLOP*S for the first run and *45 MFLOPS* for the second run. The performance obtained by the second run is about *40%* better than the performance by the first run.

Figure 2 shows the performance of the *BT*. The *BT* code used is an *Intel iPSC/860* version which was ported to the *NAS Paragon*. Timings reported for the *BT* in Figure 2 are according to the *NPB* rules. The first run takes *629* seconds whereas the second runs takes *490* seconds. The performance of the second run is about *30%* better than the first run.



It is clear from the Figures 1-2 that the performance of the second run is about 30% to 40% higher than the first run. This degradation in the performance for the first run is not acceptable since users will always run their code once.

Figure 3 shows the performance of DGEMM on two compute nodes , i.e. on node *0* and *node 1*. The function gsync was used to synchronize all node processes. The function gsync [6] performs a global synchronization operation. When a node process calls the gsync() func-

tion, it waits until all other nodes in the application call `gsync()` before continuing. All nodes in the application must call `gsync` before any node in the application can continue. The *MFLOP*S rate shown in Figures 3-8 are for the *first* run. The performance has decreased from *27 MFLOPS* to *22 MFLOPS*.

Figure 4 shows the performance of `DGEMM` on four compute nodes. The performance has further degraded to an average of about *13 MFLOPS* except for node *1* on which it is about *18 MFLOPS*. The reason for relatively better performance on node *1* than on nodes *0, 2* and *3* is that node *1* happens to be the first node to receive the empty data arrays from the service node.

Figure 4: DGEMM on four compute nodes.

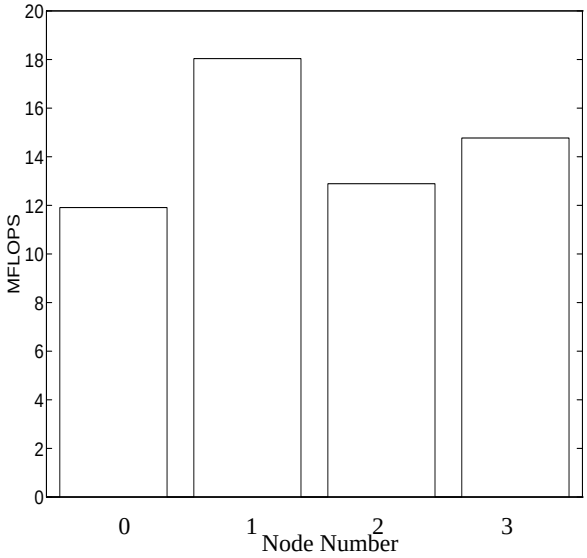


Figure 5 shows the performance of `DGEMM` on eight compute nodes. The performance has degraded further to an average of *7 MFLOPS*. The performance on node *0* is relatively better than the rest of the nodes.

Figure 6 shows the performance of `DGEMM` on sixteen nodes. Here the average performance has been further decreased to about *6 MFLOPS*. The performance on at least one node (node *0*) is *21 MFLOPS*, much better than the rest of the nodes.

Figure 7 shows the performance of `DGEMM` on *32* compute nodes. Here the average performance has further decreased to about *3 MFLOPS*. Unlike in Figures 4-6, here the performance on two compute nodes (node *25* and node *29*) is relatively better than on the rest of the nodes.

Figure 8 shows the performance of DGEMM on 64 compute nodes. Increasing the number of nodes from 32 to 64 has further degraded the performance. Here the performance on nodes *0*, *26* and *33* is much better compared to the rest of the nodes.

7: Hypothesis

It is clear from Figures 2-8 that as we increase the number of nodes, the performance decreases from *27 MFLOPS* to *1.5 MFLOPS* for the first run. The unused part of the *OSF* server must be paged out to the boot node, whenever the memory requirement is more than about *6 MB*, to provide space for the arrays *A, B* and *C* in the program on the compute node. While the paging-out of the unused part of the *OSF* server is going on, pages containing arrays *A, B* and *C* are being paged-in from the service node to each of the compute nodes. These paging-in and paging-out activities take place simultaneously at the first reference and use of the arrays *A, B* and *C* and *not* at the DIMENSION statement in the program. The net result is that the simultaneous paging-in and paging-out creates additional traffic in the network that increases with the increasing number of compute nodes.

8: Remedies for eliminating paging-in of empty data arrays

8.1: Locking the memory at run time

The activity of paging-in can be removed by using the run time option -plk [6] which causes all of the application's pages to be brought at the start of execution and to remain resident. The results of doing this are shown in Figure 9. The performance on each compute node is *45 MFLOP*S in the first run.

The run time option -plk was tried on codes of different sizes and we found that for a code that needs about *7 MB* per compute node, the time for loading the code from the service node on to the compute node became very large. In many cases load time became so large that we had to terminate the process of loading. From our experience we found that the run time option -plk is not a solution for codes which need more than *7 MB* of memory per node.

8.2: Dynamic Allocation of memory

The dynamic allocation of memory can be performed in a number of ways. The best method is to use the ALLOCATE statement [13]. The ALLOCATE statement allocates storage for each pointer-based variable and allocatable common block which appears in the statement. The DEALLOCATE statement causes the memory allocated for each pointer-based variable or allocatable COMMON block that appears in the statement to be deallocated. Fortunately both ALLOCATE and DEALLOCATE are available [13]. A dynamic or allocatable COMMON block is a common block whose storage is not allocated until an explicit ALLOCATE statement is executed. The syntax of statements ALLOCATE and DEALLOCATE is given in Table 4 and their use in *Appendix A*.

Table 4: Syntax for using ALLOCATE and DEALLOCATE statements

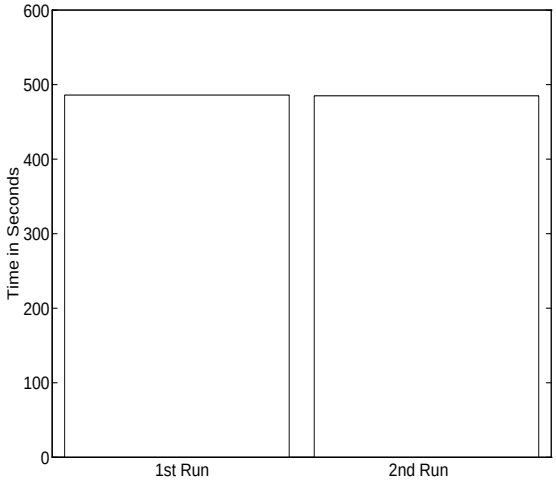
ALLOCATE (name[,name]...[,STAT= var])
DEALLOCATE (a1,[,a1]...[,STAT= var])
name is a pointer-based variable or the name of an allocatable COMMON block enclosed in slashes
a1 is a pointer-based variable or the name of an allocatable COMMON block enclosed in slashes.
var is an integer variable, integer array element, or integer member of a structure.

We have found that the most effective and elegant method of removing the undesirable and unnecessary paging-in of empty arrays provided by the *Paragon OS* is to use dynamic allocation of memory inside the application. The dynamic allocation of memory creates the arrays *A, B*

and *C* at run time on the compute processors rather than at compile time on the service node. The static allocation of memory creates the arrays *A*, *B* and *C* at compile time and at run time they are paged-in to the compute processors as and when they are first referenced and used. The performance of `DGEMM` using dynamic allocation of memory is shown in Figure 10. The dynamic memory allocation removes a serializing bottleneck and communication overhead.

The results for *BT* using dynamic allocation of memory are shown in Figure 11. We find that dynamic allocation of memory increases the performance of *BT* by *29%* and gives the correct performance in the first run.

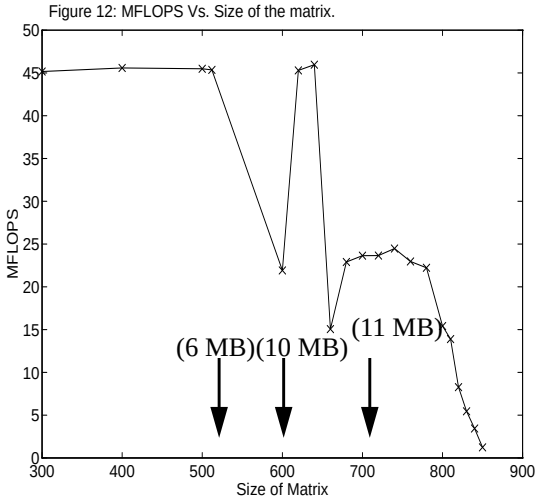
Figure 11: BT using dynamic allocation of memory



9: Paging of application against itself

The performance of `DGEMM` as a function of the size of the matrix is shown in Figure 12. When the size of the matrix is *512x512* it needs about *6.3 MB* of memory per node. As we increase the size of the matrix, initially performance goes down as the unused part of the *OSF* server is being paged-out from the compute node to the boot node to make space available for the application. As we further increase the size of the matrix, a limit is reached at about *10 MB* beyond which none of the *OSF* server is left to be paged-out and the application starts paging-out against itself. The effect of paging an application against itself is clearly seen at about *11 MB* when the performance of the `DGEMM` goes down to about *1 MFLOPS*.

Figure 12: MFLOPS Vs. Size of the matrix.



10: Conclusions

(1) Paging-in of data (empty arrays) during execution time degrades the performance of the application and should be avoided. This service performed by the *Paragon* operating system is unnecessary and is undesirable.

(2) One can use the run-time option `-plk` to lock the memory to resolve the problem. However, the use of the `-plk` option enormously increases the load time if the memory required by the application is about *7 MB* or higher per node. A genuine remedy for unnecessary effects of paging is to use dynamic allocation of memory using `ALLOCATE` and `DEALLOCATE` statements [13]. On *NAS Paragon* or any other *Paragon* system, irrespective of the memory requirement of the application, dynamic allocation of memory should **ALWAYS** be used to eliminate the service of paging-in of empty data arrays from the service node to the compute processor. The use of dynamic allocation of memory increases the performance of applications, considered in the present work, by 30% to 40%.

(3) The performance of the application starts decreasing when the application starts paging-out and ultimately it becomes unacceptable. On the *NAS Paragon*, after *10 MB*, the application starts paging against itself.

(4) The use of virtual memory by the *OSF/1* AD operating system has been a major drawback to the performance of the *Paragon*. The large amount of memory required by *OSF/1 AD* reduces available user memory to about *6 MB* per compute processor. This is a step back-

ward from the roughly *8 MB* per node memory available to the user on the *Intel iPSC/860*. Using the virtual memory system can lead to a significant drop in performance, and to other not very transparent performance variations, which make the machine less predictable for the user. These variations and the lack of memory could be tolerated as a price for increased system stability and ease of use. However, the promise of using *OSF/1 AD* for more reliable production operation has not yet materialized. This may change over time in favor of the *Paragon*.

(5) For any robust architecture and operating system, the performance of the applications should not change whether they are run with static allocation of memory or dynamic allocation of memory. On the *Paragon* system, the performance of the applications is considerably higher *(30%* to *40%* in the present paper) if dynamic allocation of memory rather than static allocation of memory is used.

In summary, there are still major challenges ahead for the *Paragon* compilers and systems software. *Intel* is aware of the problem but so far it has not been documented anywhere, including the latest *Release Notes 1.1* [15, 16].

Acknowledgment: One of the authors (SS) gratefully acknowledges many discussions with David McNab, Bernard Traversat, William J. Nitzberg, Thanh Phung, Art Lazanoff, and Todd F. Churchill.

***** The authors are employees of *Computer Sciences Corporation*. The work is supported through *NASA* contract *NAS2-12961*.

References

[1] E. Anderson et al., *LAPACK Users' Guide*, SIAM, Philadelphia, 1992.

[2] *Overview of the i860™ XR Supercomputing Microprocessor*, 1990, Intel Corporation.

[3] *Overview of the i860™ XP Supercomputing Microprocessor*, 1991, Intel Corporation.

[4] D. Bailey et al., eds, *The NAS Parallel Benchmarks*, Technical Report RNR-91-02, NAS Ames Research Center, Moffet Field, California, 1991.

[5] D. Bailey et al., *The NAS Parallel Benchmark Results*, IEEE Parallel & Distributed technology, 43-51, February 1993.

[6] *PARAGON™ OSF/1, User's Guide*, Intel Corporation, April 1993.

[7] *OSF/1™ Operating System User's Guide*, Revision 1.0, Prentice Hall Englewood, New Jersey, 1992.

[8] *iPSC/860 ProSolver™-SES Manual*, May, 1991, Intel Corporation.

[9] i*PSC/860 ProSolver™-DES Manual*, March 1992, Intel Corporation.

[10] *Intel iPSC/860 User's Guide*, April 1993

[11] *PARAGON™ OSF/1 Fortran Compiler User's Guide*, Intel Corporation, April 1993.

[12] *CLASSPACK, Basic Math Library User's Guide*, Kuck & Associates, Release 1.3, 1992.

[13] *PARAGON™ OSF/1, Fortran Language Reference Manual*, April 1993, Intel Corporation.

[14] *Proceedings of Intel Supercomputing User's Group Meeting*, Oct. 3-6, 1993, St. Louis, Missouri

[15] *Paragon System Software Release 1.1, Release Notes for the Paragon XP/S System*, October 1993, Intel Corporation.

[16] Thanh Phung, private communication, Nov. 1993.

APPENDIX A

Dynamic Allocation of Memory in Fortran

Figure 13 shows the Fortran program with static allocation of memory and Figure 14 shows a modified progam with dynamic allocation of memory.

Figure 13: Fortran program with static allocation of memory.

```
PROGRAM abc
...
REAL a(512), b(512), c(512), x(1024)
COMMON /block1/ x
...
call sub1(a,b,c)
...
END
```

Figure 14: Fortran program with dynamic allocation of memory.

```
PROGRAM abc
PARAMETER (n1=512, n2=1024)
...
REAL a(n1), b(n1), c(n1), x(n2)
POINTER (p1, a)
POINTER (p2, b)
POINTER (p3, c)
COMMON, ALLOCATABLE /block1/ x
...
ALLOCATE (a, STAT = isa)
ALLOCATE (b, STAT = isb)
ALLOCATE (c, STAT = isc)
ALLOCATE (/block1/, STAT = isblk1)
...
CALL sub1(a,b,c)
...
DEALLOCATE(a)
DEALLOCATE(b)
DEALLOCATE(c)
DEALLOCATE(/block1/)
END
```

Enhancing Applications Performance on Intel Paragon through Dynamic Memory Allocation*

Subhash Saini and Horst Simon
NAS-NASA Ames Research Center, Mail Stop 258-6, Moffett Field, CA 94035-1000

Abstract

The Paragon operating system (OS) supports virtual memory (VM). The OS manages virtual memory by performing two services. Firstly, paging-in service pages the executable code from the service node to the compute nodes. This includes the paging-in of empty data corresponding to statically allocated arrays. Secondly, paging-out service is performed by paging the unused part of the OSF server to the boot node to make space available for the user's executable code. These paging-in and paging-out activities take place simultaneously and drastically degrade the performance of the user code. We have investigated this problem in detail, and found that the dynamic allocation of memory completely eliminates the unnecessary and undesirable effects of paging-in empty data arrays from the service node to the compute nodes and thereby increases the performance of the applications considered in the present work by 30% to 40%.

1: Introduction

The *Numerical Aerodynmical Simulation (NAS) Systems Division* received an *Intel Touchstone Sigma* prototype model *Paragon XP/S-15* in February 1993. It was found that performance of many applications including the assembly coded single node *BLAS 3* routine `DGEMM` [1] was lower than the performance on *Intel iPSC/860*. This finding was quite puzzling since the clock of the microprocessor *i860 XP* used in the *Paragon* is *25%* faster than the microprocessor *i860 XR* used in the *Intel iPSC/860* [2,3]. It was also found that the performance of the *NAS Parallel Benchmarks (NPB)* [4,5] is enhanced by about *30%* if they are run for second time in a `DO` loop. Furthermore, the performance of `DGEMM` was identical for the first run and the second run on a service node, but on a compute node the performance of the second run was about *40%* better than the first run. These anomalies in the performance on the *Paragon* led us to investigate the problem in more detail. This in turn led us to propose a method of dynamic allocation of memory that increases the performance of the applications by about *30%* to *40%*.

In Sec. 2 we give a brief overview of the *Paragon* system. Sec. 3 gives the description of the *BLAS 3* kernel and *NPB* used in the investigations. Section 4 discusses the allocation of memory per node for the *microkernel, OSF* server, system buffers and the amount of memory available for the user's application. Methodology for the investigations is given in Sec. 5. Based upon our numerical experiments, the formulated hypothesis is given in Sec. 7. The remedies for eliminating the paging-in of empty data arrays and thereby enhancing the performance of the applications are given in Sec. 8. Section 9 deals with the variation in performance when the application starts paging against itself. Our conclusions are drawn in Sec. 10.

2: Paragon Overview

2.1: The i860 XP microprocessor

The *Paragon* system is based on the *64* bit *i860 XP*^{TM} microprocessor [3] by Intel. The *i860 XP*^{TM} microprocessor has *2.5* million transistors in a single chip and runs at *50 MHz*. The theoretical speed is *100 MFLOPS* in *32* bit floating point and *75 MFLOPS* for *64* bit floating operations. The *i860 XP*^{TM} features *32* integer address registers with *32* bits each. It has *32* floating point registers with *32* bits each. The floating point registers can also be accessed as *16* floating point registers with *64* bits each or *8* floating point registers with *128* bits each. Each floating point register has two read ports, a write port and two-bidirectional ports. All these ports are *64* bits wide and can be used simultaneously. The floating point registers serve as input to the floating point adder and multiplier. In vector computations, these registers are used as buffers while the data cache serves as vector registers. The *i860 XP*^{TM} microprocessor has *16 KB* of instruction and *16 KB* of data caches. The data cache has a *32* bit path to the integer unit and *128* bit data path to the floating point unit. The *i860 XP*^{TM} has a number of advanced features to facilitate high execution rates. The *i860 XP*^{TM} microprocessor's floating point unit integrates single-cycle operation, *64* bit and *128* bit data paths on chip and a *128* bit data path to main memory for fast access to data and transfer of results. Floating point add, multiply and fetch from main memory are pipelined operations, and they take advantage of a three-stage pipeline to produce one result every clock for *32* bit add or multiply operations and *64* bit adds. The *64* bit multiplication takes two clocks.

2.2: NAS Intel Paragon XP/S-15

A single node of the *Paragon XP/S-15* [6] consists of two *i860 XP*^{TM} microprocessors: one for computation and the other for communication. The compute processor is for computation and the communication processor handles all message-protocol processing thus freeing the computation processor to do computations. Currently, the communication processor is *not* used in the *NAS Paragon*. Each compute processor has *16 MB* of local memory but at *NAS* only about *6 MB* is available for applications, the rest being used for the micro kernel, *OSF* server and system buffers.

The *NAS Paragon* has *256* slots for nodes. Slots are given physical node numbers from *0* through *255*. Slots are physically arranged in a rectangular grid of size *16* by *16*. There are *8* service nodes; four of them have *16 MB* of memory each and the other four have *32 MB* of memory each. Column *0* and column *14* have no physical nodes. The service partition contains *8* nodes in the last column. One of these service nodes is a boot node. This boot node has *32 MB* of memory and is connected to a *Redundant Array of Independent Disks-1 (RAID-1*). The compute partition has *208* nodes which occupy columns *1* through *13*. Compute processors are given logical numbers *0* through *207*. Compute processors are arranged in a *16* by *13* rectangular grid . The *227* nodes are arranged in a two-dimensional mesh using wormhole routing network technology. The four service nodes comprise the service partition and provide an interface to the outside world, serving as a *front end* to the *Paragon* system. Besides running jobs on the compute nodes, the service nodes run interactive jobs, such as *shells* and *editors*. They appear as one computer running *UNIX*.

Theoretical peak performance for *64* bit floating point arithmetic is *15.6 GFLOPS* for the *208* compute nodes. Hardware node-to-node bandwidth is *200 MB* per second in full duplex.

The nodes of the *NAS Paragon* are organized into groups called partitions [6]. Partitions are organized in a hierarchical structure similar to that of the *UNIX* file system. Each partition has a *pathname* in which successive levels of the tree are separated by a periods (“.”), analogous to “/” in the *UNIX* file system. A subpartition contains a subset of the nodes of the parent partition.

Currently, on the *NAS Paragon* there are no subpartitions of *.compute* or *.service*. The root partition (denoted by “.”) contains all *227* nodes of the *Paragon*. There are two subpartitions of the root partition: the compute partition, named *.compute*, contains *208* nodes to run parallel applications. The service partition, named *.service*, contains four nodes devoted to interactive jobs. The remaining eight nodes are not part of a subpartition and serve as disk controllers and are connected to the *RAID* for *I/O*. The four nodes of the service partition appear as one computer. In summary, the *NAS Paragon* system has *208* compute nodes, 3 *HiPPI* nodes, *1* boot node, *8* disk nodes, *4* service nodes of which *1* is a boot node and *4* nodes are not used at this time, for a total of *227* nodes. When a user logs onto the *Paragon*, the *shell* runs on one of the four service nodes. In the current release of the *Paragon OS*, processes do not move between service nodes to provide load balancing. However, the load leveler decides on which node a process should be started. In principle, partitions and subpartitons may overlap. For instance, there could be a subpartition called *.compute.part1* consisting of nodes *0-31* of *.compute*, and another subpartition called *.compute.part2* consisting of nodes *15-63* of *.compute*. However, in the current release of the operating system on the *NAS Paragon* , there are two problems which restrict the use of subpartitions. First, running more than one application on a node (either two jobs in the same partition or jobs in overlapping partitions) may cause the system to crash. Second, the existence of overlapping partitions sometimes causes jobs to wait when they need not. For these two reasons, there are currently no subpartitions of the *.comp*ute partition. All jobs run directly on the *.compute* partition.

2.3: Paragon Operating System

The *UNIX* operating system was originally designed for sequential computers and is not very well suited to the performance of massively parallel applications. The *Paragon* operating system is based upon two operating systems: the *Mach* system from *Carnegie Mellon University* and the *Open Software Foundation's OSF/1 AD* distributed system for multicomputers [7]. The *Paragon*'s operating system provides all the *UNIX* features including *virtual memory*; *shell*, *command*s and *utilities*; *I/O* services; and networking support for *ftp, rpc* and *NFS*. Each *Paragon* node has a small microkernel irrespective of the role of the node in the system. The *Paragon* operating system provides programming flexibility through virtual memory. In theory, virtual memory simplifies application development and porting by enabling code requiring large memory to run on a single compute node before being distributed across multiple nodes. The application runs in virtual memory which means that each process can access more memory than is physically available on each node.

3: Applications used

3.1: Basic Linear Algebra Subprograms

BLAS 1, 2 and *3* are the basic building blocks for many of scientific and engineering applications [1]. For example, the dot product is a basic kernel in *Intel's ProSolver Skyline Equation Solver (ProSolver-SES)* [8], a direct solver using skyline storage, useful for performing Finite Element Structural analysis in designing aerospace structures. *BLAS 3 (matrix-matrix)* kernels are basic kernels in *Intel's ProSolver Dense Equation Solver (ProSolver-DES)* [9], a direct solver that may be applied in solving computational electromagnetics *(CEM*) problems using *Method of Moments (MOM). BLAS 2* and *BLAS 3* are basic kernels in *LAPACK* [1]. In the present paper, we have used a *BLAS*

3 routine called `DGEMM` to compute *C = A*B*, where *A* and *B* are real general matrices. The `DGEMM` is a single node assembly coded routine and as such involves no interprocessor communication.

3.2: NAS Parallel Benchmarks

The *NPB* [4,5] were developed to evaluate the performance of highly parallel supercomputers. One of the main features of these benchmarks is their *pencil and paper* specification, which means that all details are specified algorithmically thereby avoiding many of the difficulties associated with traditional approaches to evaluating highly parallel supercomputers. The *NPB* consist of a set of eight problems each focusing on some important aspect of highly parallel supercomputing for computational aerosciences. The eight problems include five kernels and three simulated computational fluid dynamics *(CFD)* applications. The implementation of the kernels is relatively simple and straightforward and gives some insight into the general level of performance that can be expected for a given highly parallel machine. The other three simulated *CFD* applications need more effort to implement on highly parallel computers and are representative of the types of actual data movement and computation needed for computational aerosciences. In the present paper, we have used the block tridiagonal *(BT)* benchmark, which was ported from the *Intel iPSC/860* [10] to the *Paragon*. The *NPB* all involve significant interprocessor communication with the exception of the Embarrassingly Parallel *(EP)* benchmark which involves almost no interprocessor communication.

4: Distribution of memory on Paragon

The exact amount of memory available for the user's code is very hard to estimate as it depends upon several factors such as the history of the *Paragon* system since the last reboot, number of nodes, size of the system buffers set by the user at run time etc. The approximate breakdown of memory per node for the *NAS Paragon* is shown in Table 1. Memory taken by the microkernel per node on the *NAS Paragon* is bigger than claimed by *Intel* as the *NAS* is running a debugging version of the microkernel. The microkernel is the only system software component that is in the memory of each compute node at all times including its internal tables and buffers. The *OSF* server is in the memory of each compute node initially, but as pages are needed by the application unused parts of server is paged-out to the boot node. Across the whole machine the *Paragon OS* takes *2 GB* of memory out of total of *3.3 GB* of memory, thus leaving only *1.25 GB* for the user's application.

Table 1: Distribution of Memory on each NAS compute processor

Component of OS	Memory in MB
Microkernel	5
OSF Server	4
Message Buffer	1
Free Memory	6

5: Methodology

5.1: Operating System and Compiler

The *Paragon OS* used is version *R1.1*. and the *Fortran* compiler is *4.1* [11]. The compiler options used are the `f77 -O4 -Mvect -Knoieee abc.f -lkmath` [12] and the compilation was done on the service node. There is a compiler option by which one may set the size of the portion of the cache used by the vectorizer to *number* bytes. This *number* must be a multiple of *16*, and less than the cache size 16384 of the microprocessor *i860 XP*. In most cases the best results occur when *number* is set to *4096*, which is the default. In view of this we decided to choose the default size *4 KB* and the highest optimization level of *4* was used. This level of optimization generates a basic block for each *Fortran* statement and scheduling within the basic block is performed. It does perform aggressive register allocation for software pipelined loops. In addition, code for pipelined loops is scheduled several ways, with the best way selected for the assembly file. The option `-Knoieee` was used, which produces a program that flushes denormals to *0* on creation (which reduces underflow traps) and links in a math library that is not as accurate as the standard library, but offers greater performance. This library offers little or no support for exceptional data types such as *INF* and *NaN*, and will not trap on such values when encountered. If used while compiling, it tells the compiler to perform real and double precision divides using an inline divide algorithm that offers greater performance than the standard algorithm. This algorithm produces results that differ from the results specified by the *IEEE* standard by no more than three units in the last place *(ulp)*.

5.2: Procedure for 1st Run and 2nd Run

It was found that the performance of *NPB* codes is enhanced by about *30%* if they are run for second time in a `DO` loop. Furthermore, the performance of `DGEMM` was identical for the first run and second run on a service node but on a compute node the performance of the second run was about *40%* better than the first run. In our numerical results section we will present results for a first run and a second run of an application. The procedure to obtain first run and second run for a given application is illustrated in Table 2. In this table, a `DO` loop index `i` running from *1* to *2* is inserted just before the section of the code we want to

time for benchmark purposes. In this table the *first run* corresponds to *i=1* and the *second run* corresponds to *i=2* as shown in Table 2. The overhead in calling the function DCLOCK was estimated to be about $1.5x10^{-6}$second.

Table 2: Procedure for obtaining first run and second run

```
PROGRAM abc
...
DO i = 1, 2
t0 = DCLOCK()
t1 = DCLOCK
CALL DGEMM( ,..., ...)
t2 = DCLOCK()
time = t2 - (t1 - t0)
ENDDO
...
END
```

5.3: Compiling and linking on the partitions

The *Paragon* system has two types of partitions: (a) a *service partition* and (b) a *compute partition*. The partition where an application runs can be specified when you compile and execute it. The *-nx* switch defines the preprocessor symbol *_NODE* and links with the *nx* library *libnx.a* [11]. It also links with the start-up routine—the controlling process that runs in the service partition and starts up the application in the compute partition. Commands to run the application on service partition and compute partition are given in Table 3.

Table 3: Compiling and executing on Mesh

Service Partition	Compute Partition
> f77 prog.f > a.out	> f77 prog.f -nx > a.out -sz 1 > a.out -sz 64

5.4: Numerical experiments

The following two numerical experiments were performed:

(a) First experiment: Experiment in which no interprocessor communication is involved and only communication is due to the paging-in of executable code from the service node to the compute node and if memory requirement exceeds *6 MB* per node, then paging-out of the unused part of the *OSF* server from the compute nodes to the boot node. The single node *BLAS 3* routine DGEMM was used.

(b) Second experiment: Experiment in which interprocessor communication is involved in addition to the communication due to paging-in and paging-out. The *BT* application from the *NPB* was used.

6: Results

Fig 1(a) shows the results for the assembly coded *BLAS 3* routine DGEMM on a service node obtained for the first run and the second run. Notice that on the service node the results for first run and second run are identical. The routine DGEMM is a single node routine and as such involves no interprocessor communication.

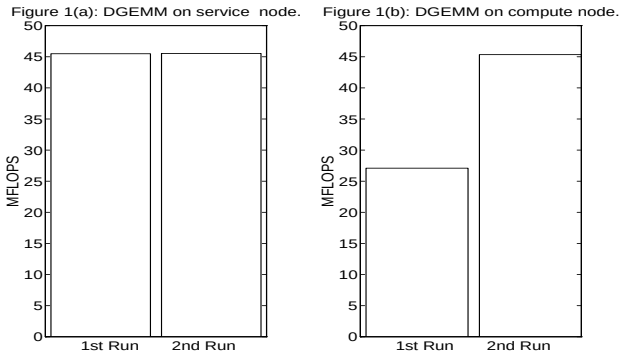
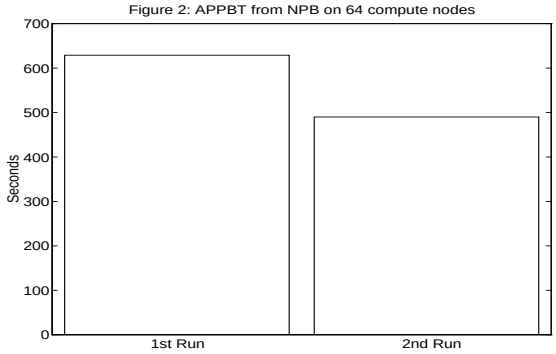


Fig 1(b) shows the results for the assembly coded *BLAS 3* on a compute node for the first run and second run. The performance of the DGEMM is *27 MFLOP*S for the first run and *45 MFLOPS* for the second run. The performance obtained by the second run is about *40%* better than the performance by the first run.

Figure 2 shows the performance of the *BT*. The *BT* code used is an *Intel iPSC/860* version which was ported to the *NAS Paragon*. Timings reported for the *BT* in Figure 2 are according to the *NPB* rules. The first run takes *629* seconds whereas the second runs takes *490* seconds. The performance of the second run is about *30%* better than the first run.



It is clear from the Figures 1-2 that the performance of the second run is about 30% to 40% higher than the first run. This degradation in the performance for the first run is not acceptable since users will always run their code once.

Figure 3 shows the performance of DGEMM on two compute nodes , i.e. on node *0* and *node 1*. The function gsync was used to synchronize all node processes. The function gsync [6] performs a global synchronization operation. When a node process calls the gsync() func-

tion, it waits until all other nodes in the application call `gsync()` before continuing. All nodes in the application must call `gsync` before any node in the application can continue. The *MFLOP*S rate shown in Figures 3-8 are for the *first* run. The performance has decreased from *27 MFLOPS* to *22 MFLOPS*.

Figure 4 shows the performance of `DGEMM` on four compute nodes. The performance has further degraded to an average of about *13 MFLOPS* except for node *1* on which it is about *18 MFLOPS*. The reason for relatively better performance on node *1* than on nodes *0, 2* and *3* is that node *1* happens to be the first node to receive the empty data arrays from the service node.

Figure 4: DGEMM on four compute nodes.

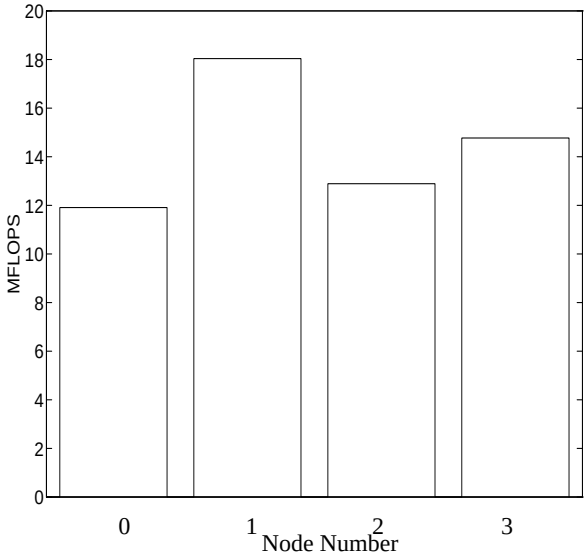


Figure 5 shows the performance of `DGEMM` on eight compute nodes. The performance has degraded further to an average of *7 MFLOPS*. The performance on node *0* is relatively better than the rest of the nodes.

Figure 6 shows the performance of `DGEMM` on sixteen nodes. Here the average performance has been further decreased to about *6 MFLOPS*. The performance on at least one node (node *0*) is *21 MFLOPS*, much better than the rest of the nodes.

Figure 7 shows the performance of `DGEMM` on *32* compute nodes. Here the average performance has further decreased to about *3 MFLOPS*. Unlike in Figures 4-6, here the performance on two compute nodes (node *25* and node *29*) is relatively better than on the rest of the nodes.

Figure 8 shows the performance of DGEMM on 64 compute nodes. Increasing the number of nodes from 32 to 64 has further degraded the performance. Here the performance on nodes *0*, *26* and *33* is much better compared to the rest of the nodes.

7: Hypothesis

It is clear from Figures 2-8 that as we increase the number of nodes, the performance decreases from *27 MFLOPS* to *1.5 MFLOPS* for the first run. The unused part of the *OSF* server must be paged out to the boot node, whenever the memory requirement is more than about *6 MB*, to provide space for the arrays *A, B* and *C* in the program on the compute node. While the paging-out of the unused part of the *OSF* server is going on, pages containing arrays *A, B* and *C* are being paged-in from the service node to each of the compute nodes. These paging-in and paging-out activities take place simultaneously at the first reference and use of the arrays *A, B* and *C* and *not* at the DIMENSION statement in the program. The net result is that the simultaneous paging-in and paging-out creates additional traffic in the network that increases with the increasing number of compute nodes.

8: Remedies for eliminating paging-in of empty data arrays

8.1: Locking the memory at run time

The activity of paging-in can be removed by using the run time option -plk [6] which causes all of the application's pages to be brought at the start of execution and to remain resident. The results of doing this are shown in Figure 9. The performance on each compute node is *45 MFLOP*S in the first run.

The run time option -plk was tried on codes of different sizes and we found that for a code that needs about *7 MB* per compute node, the time for loading the code from the service node on to the compute node became very large. In many cases load time became so large that we had to terminate the process of loading. From our experience we found that the run time option -plk is not a solution for codes which need more than *7 MB* of memory per node.

8.2: Dynamic Allocation of memory

The dynamic allocation of memory can be performed in a number of ways. The best method is to use the ALLOCATE statement [13]. The ALLOCATE statement allocates storage for each pointer-based variable and allocatable common block which appears in the statement. The DEALLOCATE statement causes the memory allocated for each pointer-based variable or allocatable COMMON block that appears in the statement to be deallocated. Fortunately both ALLOCATE and DEALLOCATE are available [13]. A dynamic or allocatable COMMON block is a common block whose storage is not allocated until an explicit ALLOCATE statement is executed. The syntax of statements ALLOCATE and DEALLOCATE is given in Table 4 and their use in *Appendix A*.

Table 4: Syntax for using ALLOCATE and DEALLOCATE statements

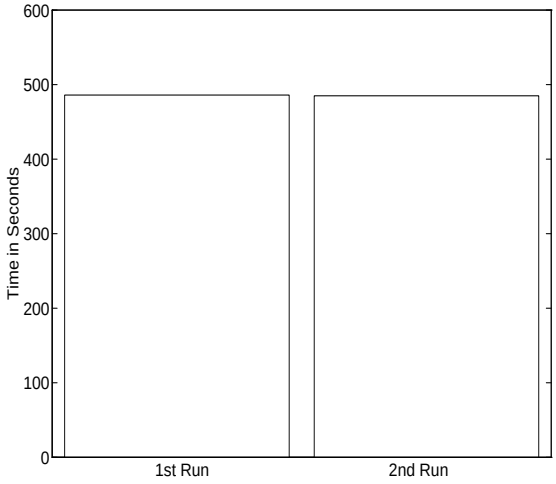
ALLOCATE (name[,name]...[,STAT= var])
DEALLOCATE (a1,[,a1]...[,STAT= var])
name is a pointer-based variable or the name of an allocatable COMMON block enclosed in slashes
a1 is a pointer-based variable or the name of an allocatable COMMON block enclosed in slashes.
var is an integer variable, integer array element, or integer member of a structure.

We have found that the most effective and elegant method of removing the undesirable and unnecessary paging-in of empty arrays provided by the *Paragon OS* is to use dynamic allocation of memory inside the application. The dynamic allocation of memory creates the arrays *A, B*

and *C* at run time on the compute processors rather than at compile time on the service node. The static allocation of memory creates the arrays *A*, *B* and *C* at compile time and at run time they are paged-in to the compute processors as and when they are first referenced and used. The performance of `DGEMM` using dynamic allocation of memory is shown in Figure 10. The dynamic memory allocation removes a serializing bottleneck and communication overhead.

The results for *BT* using dynamic allocation of memory are shown in Figure 11. We find that dynamic allocation of memory increases the performance of *BT* by *29%* and gives the correct performance in the first run.

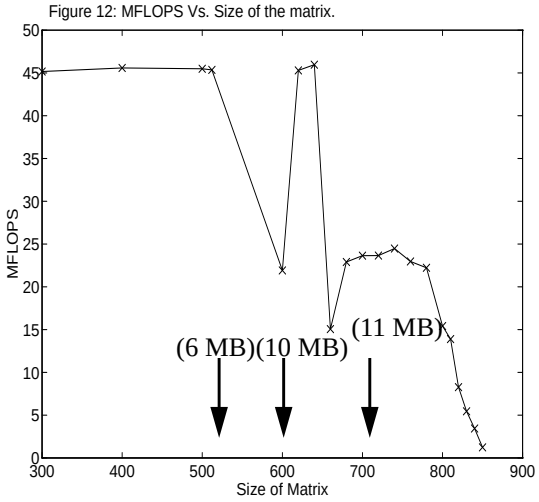
Figure 11: BT using dynamic allocation of memory



9: Paging of application against itself

The performance of `DGEMM` as a function of the size of the matrix is shown in Figure 12. When the size of the matrix is *512x512* it needs about *6.3 MB* of memory per node. As we increase the size of the matrix, initially performance goes down as the unused part of the *OSF* server is being paged-out from the compute node to the boot node to make space available for the application. As we further increase the size of the matrix, a limit is reached at about *10 MB* beyond which none of the *OSF* server is left to be paged-out and the application starts paging-out against itself. The effect of paging an application against itself is clearly seen at about *11 MB* when the performance of the `DGEMM` goes down to about *1 MFLOPS*.

Figure 12: MFLOPS Vs. Size of the matrix.



10: Conclusions

(1) Paging-in of data (empty arrays) during execution time degrades the performance of the application and should be avoided. This service performed by the *Paragon* operating system is unnecessary and is undesirable.

(2) One can use the run-time option `-plk` to lock the memory to resolve the problem. However, the use of the `-plk` option enormously increases the load time if the memory required by the application is about *7 MB* or higher per node. A genuine remedy for unnecessary effects of paging is to use dynamic allocation of memory using `ALLOCATE` and `DEALLOCATE` statements [13]. On *NAS Paragon* or any other *Paragon* system, irrespective of the memory requirement of the application, dynamic allocation of memory should **ALWAYS** be used to eliminate the service of paging-in of empty data arrays from the service node to the compute processor. The use of dynamic allocation of memory increases the performance of applications, considered in the present work, by 30% to 40%.

(3) The performance of the application starts decreasing when the application starts paging-out and ultimately it becomes unacceptable. On the *NAS Paragon*, after *10 MB*, the application starts paging against itself.

(4) The use of virtual memory by the *OSF/1* AD operating system has been a major drawback to the performance of the *Paragon*. The large amount of memory required by *OSF/1 AD* reduces available user memory to about *6 MB* per compute processor. This is a step back-

ward from the roughly *8 MB* per node memory available to the user on the *Intel iPSC/860*. Using the virtual memory system can lead to a significant drop in performance, and to other not very transparent performance variations, which make the machine less predictable for the user. These variations and the lack of memory could be tolerated as a price for increased system stability and ease of use. However, the promise of using *OSF/1 AD* for more reliable production operation has not yet materialized. This may change over time in favor of the *Paragon*.

(5) For any robust architecture and operating system, the performance of the applications should not change whether they are run with static allocation of memory or dynamic allocation of memory. On the *Paragon* system, the performance of the applications is considerably higher *(30%* to *40%* in the present paper) if dynamic allocation of memory rather than static allocation of memory is used.

In summary, there are still major challenges ahead for the *Paragon* compilers and systems software. *Intel* is aware of the problem but so far it has not been documented anywhere, including the latest *Release Notes 1.1* [15, 16].

Acknowledgment: One of the authors (SS) gratefully acknowledges many discussions with David McNab, Bernard Traversat, William J. Nitzberg, Thanh Phung, Art Lazanoff, and Todd F. Churchill.

***** The authors are employees of *Computer Sciences Corporation*. The work is supported through *NASA* contract *NAS2-12961*.

References

[1] E. Anderson et al., *LAPACK Users' Guide*, SIAM, Philadelphia, 1992.

[2] *Overview of the i860™ XR Supercomputing Microprocessor*, 1990, Intel Corporation.

[3] *Overview of the i860™ XP Supercomputing Microprocessor*, 1991, Intel Corporation.

[4] D. Bailey et al., eds, *The NAS Parallel Benchmarks*, Technical Report RNR-91-02, NAS Ames Research Center, Moffet Field, California, 1991.

[5] D. Bailey et al., *The NAS Parallel Benchmark Results*, IEEE Parallel & Distributed technology, 43-51, February 1993.

[6] *PARAGON™ OSF/1, User's Guide*, Intel Corporation, April 1993.

[7] *OSF/1™ Operating System User's Guide*, Revision 1.0, Prentice Hall Englewood, New Jersey, 1992.

[8] *iPSC/860 ProSolver™-SES Manual*, May, 1991, Intel Corporation.

[9] i*PSC/860 ProSolver™-DES Manual*, March 1992, Intel Corporation.

[10] *Intel iPSC/860 User's Guide*, April 1993

[11] *PARAGON™ OSF/1 Fortran Compiler User's Guide*, Intel Corporation, April 1993.

[12] *CLASSPACK, Basic Math Library User's Guide*, Kuck & Associates, Release 1.3, 1992.

[13] *PARAGON™ OSF/1, Fortran Language Reference Manual*, April 1993, Intel Corporation.

[14] *Proceedings of Intel Supercomputing User's Group Meeting*, Oct. 3-6, 1993, St. Louis, Missouri

[15] *Paragon System Software Release 1.1, Release Notes for the Paragon XP/S System*, October 1993, Intel Corporation.

[16] Thanh Phung, private communication, Nov. 1993.

APPENDIX A

Dynamic Allocation of Memory in Fortran

Figure 13 shows the Fortran program with static allocation of memory and Figure 14 shows a modified progam with dynamic allocation of memory.

Figure 13: Fortran program with static allocation of memory.

```
PROGRAM abc
...
REAL a(512), b(512), c(512), x(1024)
COMMON /block1/ x
...
call sub1(a,b,c)
...
END
```

Figure 14: Fortran program with dynamic allocation of memory.

```
PROGRAM abc
PARAMETER (n1=512, n2=1024)
...
REAL a(n1), b(n1), c(n1), x(n2)
POINTER (p1, a)
POINTER (p2, b)
POINTER (p3, c)
COMMON, ALLOCATABLE /block1/ x
...
ALLOCATE (a, STAT = isa)
ALLOCATE (b, STAT = isb)
ALLOCATE (c, STAT = isc)
ALLOCATE (/block1/, STAT = isblk1)
...
CALL sub1(a,b,c)
...
DEALLOCATE(a)
DEALLOCATE(b)
DEALLOCATE(c)
DEALLOCATE(/block1/)
END
```