Implementing the NHT-1 Application I/O Benchmark

Samuel A. Fineberg

Report RND-93-007 May 1993

Computer Sciences Corporation[1]
Numerical Aerodynamic Simulation
NASA Ames Research Center, M/S 258-6
Moffett Field, CA 94035-1000
(415)604-4319
e-mail: fineberg@nas.nasa.gov

Abstract

The NHT-1 I/O (Input/Output) benchmarks are a benchmark suite developed at the Numerical Aerodynamic Simulation Facility (NAS) located at NASA Ames Research Center. These benchmarks are designed to test various aspects of the I/O performance of parallel supercomputers. One of these benchmarks, the Application I/O Benchmark, is designed to test the I/O performance of a system while executing a typical computational fluid dynamics application. In this paper, the implementation of this benchmark on three parallel systems located at NAS and the results obtained from these implementations are reported. The machines used were an 8 processor Cray Y-MP, a 32768 processor CM-2, and a 128 processor iPSC/860. The results show that the Y-MP is the fastest machine and has relatively well balanced I/O performance. I/O adds 2-40% overhead, depending on the number of processors utilized. The CM-2 is the slowest machine, but it has I/O that is fast relative to its computational performance. This resulted in typical I/O overheads on the CM-2 of less than 4%. Finally, the iPSC/860, while not as computationally fast as the Y-MP, is considerably faster than the CM-2. However, the iPSC/860's I/O performance is quite poor and can add overhead of more than 70%.

1. This work was supported through NASA contract NAS 2-12961.

1.0 Introduction

The NHT-1 I/O (Input/Output) benchmarks are a new benchmark suite being developed at the Numerical Aerodynamic Simulation Facility (NAS), located at NASA Ames Research Center. These benchmarks are designed to test the performance of parallel I/O subsystems under typical workloads encountered at NAS. The benchmarks are broken into three main categories, application disk I/O, peak (or system) disk I/O, and network I/O. In this report, the experiences encountered when implementing the application disk I/O benchmark on systems located at NAS will be reported. Further, the results of the benchmark on these systems are presented.

2.0 The Application I/O Benchmark[1] [CaC92]

2.1 Background

Computational Fluid Dynamics (CFD) is one of the primary fields of research that has driven modern supercomputers. This technique is used for aerodynamic simulation, weather modeling, as well as other applications where it is necessary to model fluid flows. CFD applications involve the numerical solution of non-linear partial differential equations in two or three spatial dimensions. The governing differential equations representing the physical laws governing fluids in motion are referred to as the Navier-Stokes equations. The NAS Parallel Benchmarks [BaB91] consist of a set of five kernels, less complex problems intended to highlight specific areas of machine performance, and three application benchmarks. The application benchmarks are iterative partial differential equation solvers that are typical of CFD codes. While the NAS Parallel Benchmarks are a good measure of computational performance, I/O is also a necessary component of numerical simulation. Typically, CFD codes iterate for a predetermined number of steps. Due to the large amount of data in a solution set at each step, the solution files are written intermittently to reduce I/O bandwidth requirements for the initial storage as well as for future post-processing. The *Application I/O Benchmark* [CaC92] simulates the I/O required by a pseudo-time stepping flow solver that periodically writes its solution matrix for post-processing (e.g., visualization). This is accomplished by implementing the *Approximate Factorization Benchmark* (called BT because it involves finding the solution to a block tridiagonal system of equations) precisely as described in Section 4.7.1 of The NAS Parallel Benchmarks [BaB91], with additions described below. In an absolute sense, this benchmark only measures the performance of a system on this particular class of CFD applications and only a single type of application I/O. However, the results from this benchmark should also be useful for predicting the performance of other applications that exhibit similar behavior. The specification is intended to conform to the "paper and pencil" format promulgated in the NAS Parallel Benchmarks, and in particular the Benchmark Rules as described in Section 1.2 of [BaB91].

---

1. To obtain a copy of the NAS Parallel Benchmarks or the NHT-1 I/O Benchmarks report as well as sample implementations of the benchmarks, send e-mail to `bm-codes@nas.nasa.gov` or send US-mail to NAS Systems Development Branch, M/S 258-5, NASA Ames Research Center, Moffett Field, CA 94035.

2.2 Benchmark Instructions

The BT benchmark consists of a set of N_S iterations performed on a solution vector U. For the Application I/O Benchmark, BT is to be performed with precisely the same specifications as in the NAS parallel benchmarks, with the additional requirement that every I_W iterations, the solution vector, U, must be written to disk file(s) in a serial format. The serial format restriction is imposed because most post-processing is currently performed on serial machines (e.g., workstations) or other parallel systems. Therefore, the data must be in a format that is interchangeable with other systems without significant modification. I/O may be performed either synchronously or asynchronously with the computations. Performance on the Application I/O Benchmark is to be reported as three quantities: The elapsed time T_T, the computed I/O transfer rate R_{IO}, and the I/O overhead ζ. These quantities are described in detail below.

The specification of the Application I/O Benchmark is intended to facilitate the evaluation of the I/O subsystems as integrated with the processing elements. Hence no requirement is made of initial data layout, or method or order of transfer. In particular, it is permissible to sacrifice floating point performance for I/O performance. It is important to note, however, that the computation-only performance will be taken to be the best verified time of the BT benchmark.

For this paper, the matrix dimensions, N_ξ, N_η, and N_ζ are assumed to be equal and are lumped in to a single parameter called N. The benchmark is to be run with the input parameters shown for the largest problem size in Table 1. In addition, for this paper, the two smaller sizes were also measured to facilitate comparison with slower machines.

TABLE 1. Benchmark input parameters

N	N_S	I_W
12	60	10
64	200	5
102	200	5

2.3 Reported Quantities

2.3.1 Elapsed Time

The elapsed time T_T is to be measured from the identical timing start point as specified for the BT benchmark, to the larger of the time required to complete the file transfers, or the time to complete the computations. The time required to verify the accuracy of the output files generated is not to be included in the time reported for the Application I/O Benchmark.

2.3.2 Computed I/O Transfer Rate

The computed I/O transfer rate R_{IO} is an indication of total application performance, not just I/O performance. It is to be calculated from the following formula:

$$R_{IO} = \frac{(5w) \times (N^3) \times N_S}{I_W T_T}$$

Here, N^3 is the grid size dimension, N_S is the total number of iterations, I_W is the number of iterations between write operations, w is the word size of a data element in bytes, e.g., 4 or 8, and T_T is the total elapsed time for the BT benchmark with the added write operations. Note that the 5 in the numerator of the equation for R_{IO} reflects the fact that U is a 5xNxNxN matrix. The units of R_{IO} are bytes per second.

2.3.3 I/O Overhead

I/O overhead, ζ, is used to measure system "balance." It is computed as follows:

$$\zeta = \frac{T_T}{T_C} - 1$$

The quantity T_C is the best verified run time in seconds for the BT benchmark, for an identically sized benchmark run on an identically configured system. This is vital to insure that any algorithm changes needed to implement fast I/O do not skew the overhead calculation by generating a T_C that is too large. In this paper this constraint was not strictly followed. Instead, T_C was assumed to be the run-time of the particular BT application without any I/O code added and no algorithm modifications were made to improve I/O performance. This was because there were no published results for the largest N=102 size of the BT benchmark and not all of the codes used to generate the published N=64 results were available. The effects of variations in T_C will be discussed further in Section 4.

2.4 Verification

Another aspect of the Application I/O Benchmark is that the integrity of the data stored on disk must verified by the execution of a post-processing program on a uniprocessor system that sequentially reads the file(s) and prints elements on the solution vector's diagonal. Output from this program is compared to output from an implementation that is known to be correct in order to verify the file's integrity.

3.0 Implementation

Implementing the application I/O benchmark on a given machine involves two distinct tasks. First, one must develop or obtain a version of the BT benchmark for the target system. Second, one has to add the I/O operations to the existing code and make any possible optimizations. Of these tasks, however, the first is the most critical. This is because the quality of implementation of the BT benchmark code can greatly effect both R_{IO} and ζ. Further, if the value of T_C used for the overhead calculation is not the best one available, the amount of overhead may be greatly distorted. Examples of this will be discussed in the following sections.

3.1 Cray Y-MP

The Cray Y-MP 8/256 on which the experiments were run has eight processors and 256MW (2GBytes) of 64-bit 15-ns memory. Its clock cycle time is 6-ns and its peak speed is 2.7 GFLOPS. I/O is performed on a set of 48 disk drives making up 90GBytes of storage. For this benchmark, the Session Reservable File System (SRFS) [Cio92] was used. This is a large area of temporary disk space that can be reserved by applications while they are running. This assures that a sufficient amount of relatively fast (8MByte/sec) disk space will be available while an application is running. While additional performance might be gained by manually striping files across multiple file systems, there is no support for automatic striping of files.

The BT benchmark for the Cray was obtained from Cray Research. This code will be referred to as *CrayBT*. CrayBT was written in FORTRAN 77 using standard Cray directives and multitasking. It was designed for the N=64 benchmark and had to be modified to run for the N=102 case. This modification was completed, the I/O portion of the benchmark code was added, and a verification program was written. Measurements were made in dedicated mode to eliminate any performance variations due to system load.

The I/O code was implemented on the Cray using FORTRAN unformatted writes. U was laid out as a simple NxNxNx5 matrix and could be written with a simple `write` statement. The code used for opening the file is shown below:

```
      open (unit=ibin,file='btout.bin',status='unknown',
     *access='sequential',form='unformatted')
      rewind ibin
```

where `ibin` is the unit number to which the output file will be assigned and the file name is `btout.bin`. Then, the actual `writes` are performed as follows:

```
      do l=1,nz
         write(ibin) (((u(j,k,l,i),j=1,nx),k=1,ny),i=1,5)
      enddo
```

Here, u is the solution vector, and nx, ny, and nz are equivalent to N_ξ, N_η, and N_ζ. During each write step where step mod $I_W = 0$, nz `writes` occur, each writing nx*ny*5 words for a total of nz*nx*ny*5*8 bytes (69120 for N=12, 10485760 for N=64, and 42448320 for N=102) per write of U. The advantage of this format is that when verifying the code, the solution vector may be read back nx*ny*5*8 bytes (5760 for N=12, 163840 for N=64, and 416160 for N=102) at a time, significantly reducing the amount of memory needed for the verification program.

Finally, the verification can be accomplished easily with the following short program:

```
      program btioverify
      integer bsize,isize,ns,iw,nwrite
c defines isize, bsize, etc.
      include 'btioverify.incl'
      real*8 u(isize,isize,bsize)
      integer tstep,i,j
      open (unit=8,file='btout.bin',status='unknown',
     *access='sequential',form='unformatted')
      do tstep=1,nwrite
        do i=1,isize
           read(8) u
           do j=1,bsize
              write(6,'(F15.10)') u(i,i,j)
           enddo
        enddo
      enddo
      stop
      end
```

3.2 Thinking Machines CM-2

The CM-2 is a SIMD parallel system consisting of up to 65536 1-bit processors and up to 2048 floating point units [Hil87, ZeL88]. The configuration used for these experiments had 32768 1-bit processors, 1024 floating point units, and 4GBytes of memory. The system is controlled by a Sun 4/490 front end. The primary I/O device is the DataVault. The DataVault is a striped, parity checked disk array capable of memory to disk transfer rates of up to 25MBytes/sec. However, to achieve this speed, it uses a parallel file format that is unusable by any other machine or even a different CM-2 configuration. To satisfy the file format constraints of the benchmark, it was necessary to use the DataVault in "serial" mode. This was done with the `cm_array_to_file_so` subroutine call provided by Thinking Machines. The result of this call is a file in serial FORTRAN order containing the array with no record markers. This call was measured to operate at approximately 4.3 MBytes/sec.

Verification of data for the CM-2, however, required a different approach. Unlike the Cray or iPSC/860, it is not feasible to verify the results on a single node. Therefore, one must transfer the file to another machine to verify the data. Due to the large size of the output file (1.6GBytes for the full size benchmark), ethernet transfers were impractical. Further, problems with the network interface slowed down the high speed network link provided and limited the choice of verification machines. The most practical machine to verify the benchmark was the Cray Y-MP due to its high speed network links and its large available disk space. Therefore, the files were transferred to the Cray through the CM-HiPPI and UltraNet hub. One difficulty in verifying the data was the different floating point formats of the

Cray and the CM-2. This was alleviated using the `ieg2cray` function to convert the 64-bit IEEE floating point numbers generated on the CM-2 to 128-bit Cray floating point numbers. The 128-bit Cray format was chosen so that this conversion could be done with no loss of precision.

Initially, the I/O benchmark was implemented using the publicly available sample implementation of the BT benchmark written in CM-FORTRAN. This code will be referred to as *SampleBT/CMF*. The computation rate of SampleBT/CMF was very slow. A faster version was obtained from NAS's Applied Research Department (RNR). This version, RNRBT/CMF, was also written in CM-FORTRAN. It was faster but was not as fast as the codes cited in [BaB92]. It is the fastest version that can run for N=12 and N=102 and does not use the TMC supplied library block tridiagonal solver. The fastest BT code for N=64 used an algorithm that was dependent on N being evenly divisible by 16 and was therefore unsuitable for the I/O benchmark (i.e., it would not run for the official benchmark size of N=102). While the RNRBT/CMF code was about 32% faster than SampleBT/CMF, it still could not complete the large size (N=102) benchmark in less than about 18 hours.

The actual I/O code was quite simple to implement. The file was opened as follows:

```
 call cmf_file_open(ibin,
$      'datavault:/fineberg/btout.bin',istat)
 call cmf_file_rewind(ibin, istat)
```

where `ibin` is the unit number, `istat` is a variable in which the status of the operation will be stored, and the file to be stored on the datavault is called `/fineberg/btout.bin`. For SampleBT/CMF, U was stored in a 4-dimensional matrix spread across the processors. The code for writing this matrix was as follows:

```
if (mod(istep,ibinw) .eq. 0) then
   call cmf_cm_array_to_file_so(ibin, u, istat)
endif
```

where `istep` is the current step number, and `ibinw` is the write interval I_W. For RNRBT/CMF, U was broken up into five 3-dimensional matrices. These were written consecutively as follows:

```
if (mod(istep, ibinw) .eq. 0) then
   call cmf_cm_array_to_file_so(ibin, u1, istat)
   call cmf_cm_array_to_file_so(ibin, u2, istat)
   call cmf_cm_array_to_file_so(ibin, u3, istat)
   call cmf_cm_array_to_file_so(ibin, u4, istat)
   call cmf_cm_array_to_file_so(ibin, u5, istat)
endif
```

Verification was performed on the Cray Y-MP with the following C program:

```
include <stdio.h>
```

```
#include “btioverify.incl”
main()
{
   FILE *fp;
   long i,j,k,n;
   char foreign[8];
   double data;
   fp = fopen(“btout.bin”, “r”);
   for (k=0; k<RPT; k++){
      for (i=0; i<DIM; i++){
         fseek(fp, 5*8*(DIM*DIM*DIM)*k + 8*(i + DIM*i +
            DIM*DIM*i), 0);
         for (j=0; j<5; j++){
            fread(foreign, 8, 1, fp);
            CONVERT(foreign, &data);
            printf(“%15.10lf\n”, data);
            fseek(fp, 8*(DIM*DIM*DIM)-8, 1);
            }
         }
      }
 fclose(fp);
}
```

where `DIM`=N and `RPT`=N_S/I_W (these are defined in `btioverify.incl`). `CONVERT` is a small FORTRAN program that calls Cray’s ieg2cray function to convert from IEEE 64-bit floating point numbers to Cray 128-bit floating point numbers. The text to convert is as follows:

```
function CONVERT(a, b)
real a
double precision b
ierr = ieg2cray(3, 1, a, 0, b, 1, x)
convert=1
return
end
```

Note this program assumes that the matrix is written as specified for RNRBT/CMF. For SampleBT/CMF, U was a 5xNxNxN matrix, so the `fseek`’s would be different (RNRBT/CMF’s output is written in FORTRAN order for a NxNxNx5 matrix). Both file formats are legal for the benchmark and should result in the same verification output file.

3.3 Intel iPSC/860

The iPSC/860 is a hypercube interconnected multicomputer consisting of up to 128 i860 computational nodes and an i386 based host processor [Int91]. The nodes each have 8MB of memory (1GByte total) and run a small run-time kernel based OS called NX. The host system runs UNIX. The hypercube network uses a form of circuit switching [Nug88] and links between nodes operate at about 2.8MB/sec. I/O on the iPSC/860 is handled by its *Concurrent File System* (CFS), a set of i386 based processors controlling individual SCSI disks. The I/O nodes are each connected to a node on the hypercube network via an added link at each node, i.e., each node has 8 links, 7 for communicating with other nodes and one that can be used to connect to an I/O node. The CFS used for these experiments had 10 I/O processors, attached to each was a disk with a theoretical transfer rate of 1MByte/-sec. CFS files can be striped across disks for a theoretical peak throughput of 10MBytes/sec, although actual obtained performance is considerably lower [Nit92] and is dependent on block size and data layout. I/O was implemented using the `cwrite()` synchronous write calls provided by Intel. The use of asynchronous I/O was avoided because the amount of data generated was larger than the available buffer space and it slowed down I/O for the larger problem sizes.

Initial experiments used a version of the BT benchmark (SampleBT/iPSC) with a 1D data decomposition where only N processors could be used for a size N^3 problem. This code was written in FORTRAN 77 using Intel's standard message passing library. As expected, the results were disappointing, and though the measured overhead was low, the execution time was high and R_{IO} was low. For more on these results, see Section 4.3. Implementation of the I/O code for the 1D partitioning was more difficult than for both the Cray and CM-FORTRAN implementations because it was necessary to construct a sequential ordering on the CFS from independent regions of memory on each node. Unlike the CM-2, there is no library support for this. This code is implemented as follows. First, node 0 makes sure that no pre-existing file is resident on the CFS by opening it and closing it with the option "`status='delete'`." Next, node 0 opens the file and pre-allocates the space using `lsize`. It then closes the file and broadcasts a message to all other nodes indicating whether the pre-allocation was successful or not. If the pre-allocation fails, the program terminates. If not, all nodes open the file. The code for this is shown below:

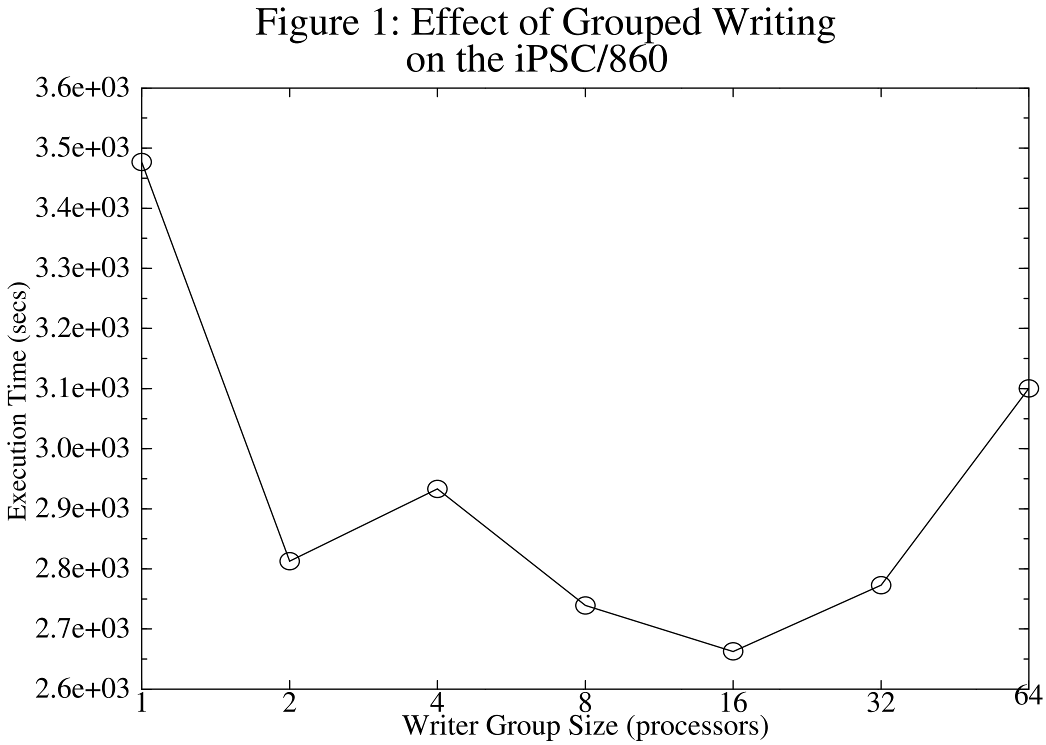
```
c iam is equal to the processor's node number
      if (iam .eq. 0) then
        open ( unit=ibin, file='/cfs/fineberg/btout.bin',
     $  status='unknown',form='unformatted')
c delete any pre-existing file
        close (unit=ibin, status='delete')
        open ( unit=ibin, file='/cfs/fineberg/btout.bin',
     $  status='new',form='unformatted')
c Calculate length then pre-allocate file space
        length1 = nx*ny*nz*8*5*(itmax/ibinw)
        length = lsize(ibin, length1, 0)
        close(unit=ibin)
c check if lsize worked
        if (length1 .ne. length) then
c lsize didn't work
           call csend (12345, 0, 4, -1, 0)
```

```
         stop 'Inadequate CFS file space'
      else
c lsize worked
         call csend (12345, 1, 4, -1, 0)
      endif
    else
      call crecv(12345, iok, 4)
      if (iok .eq. 0) then
c lsize didn't work
         stop
      endif
    endif

c everyone opens file
    open ( unit=ibin, file='/cfs/fineberg/btout.bin',
   $status='old',form='unformatted')
    rewind ibin
```

Next, the program begins to iterate, and every `ibinw` iterations the data is written as follows:

```
    if (mod(istep, ibinw) .eq. 0) then
      if (iam .lt. nx) then
         offset = ((istep/ibinw)-1)*nx*ny*nz*5*8 + 5*8*iam
         istat = lseek(8, offset, 0)
         do 992 ia=1,nz
            do 991 ja=1,ny
               call cwrite(8, u(1, 1, ja, ia), 40)
               offset = (nx-1)*5*8
               istat = lseek(8, offset, 1)
991         continue
992      continue

      endif
    endif
```

As demonstrated with the other machines, results utilizing a less than optimal coding of the computational part of the benchmark were not useful. Therefore, a better implementation of the BT benchmark that decomposed data along three dimensions (RNRBT_3D/iPSC) was obtained from Sisira Weeratunga [Wee92] of NAS's Applied Research Department. This code, also written in FORTRAN 77 with message passing, was considerably faster than the 1D code and more efficiently used the available processors. However, for the RNRBT_3D/iPSC implementation described later, additional synchronization had to be added to the I/O benchmark to correct a timing problem that appeared after adding the I/O code. The problem was most evident when using 128 processors. To correct the problem, when 128 processors were used, the processors were grouped such that only a fixed number of processors (<128) could write at once. Fortunately, this additional synchronization not only corrected the problem but also decreased execution time. The effect of this grouping of processors on the N=102 application I/O benchmark's execution time is plotted in Figure 1. As can be seen from this graph, the



lowest execution time was achieved for a write group size of 16. Therefore, for the experiments shown later in this paper, only groups of 16 processors were allowed to write at a time for all 128 node experiments on the iPSC/860. Note that these results corroborate similar CFS performance anomalies described in [Nit92].

The complicated data layout and the required synchronization caused the implementation of the I/O code for RNRBT/iPSC to be more difficult than for the other systems. The sequence for opening the file was the same as was shown for the SampleBT/iPSC example. However, the code for each of the writes of the solution vector `u` was much more complicated. This code is shown below:

```
    if (mod(istep, ibinw) .eq. 0) then
      if (iam .ge. group) then
c wait until node iam-group has finished
         call crecv(9999, itemp, 1)
      endif
      offset1 = (((istep/ibinw)-1)*nx*ny*nz +
   $        mod(iam,nodex)*ldx)
      do na3 = 0, ldz-1
         offset3 = ((iam/(nodex*nodey))*ldz + na3)*nx*ny
         do na2 = 0, ldy-1
            offset2 = ((mod(iam,nodey*nodex)/nodex)*ldy
   $              + na2)*nx
```

```
            offset = 40*(offset1+offset2+offset3)
c find correct file position
            istat = lseek(ibin, offset, 0)
c write data
            call cwrite(ibin, u(1,1,na2+1,na3+1), 40*ldx)
         enddo
      enddo
      if (iam .lt. nodes-group) then
c start node iam+group
         call csend (9999, itemp, 1, iam+group, 0)
      endif
      call gsync()
    endif
```

The writes for RNRBT/iPSC went as follows. The first `group` processors start writing data while the other processors wait for a message. The writing consists of a series of `lseeks` to the proper file positions followed by `cwrites`. When a node finishes writing its data, it sends a message to the node `iam+group`, causing that node to start writing data and waits for all nodes to complete by executing a `gsync()` (global synchronization) call. This prevented more than `group` nodes from writing at a time. Nodes in the last group do not send the final message, and when all nodes complete their writes, the nodes proceed past the `gsync()` and resume the computation portion of the benchmark.

Another problem encountered on the iPSC/860 was with verification. Because the system's nodes were each workstation CPUs, i.e., they were not 1-bit processors, the data could be verified by a single node. This worked well for N=12 and N=64, however, for N=102, the amount of time required for verification was longer than the average time between system reboots. Additionally, because the only method for removing data from the CFS was through an ethernet connection, the amount of time required to move the large data file off of the system was also greater than the average time between reboots. Therefore, the largest problem size was not verified at the time this paper was written.

The verification code was written in C and ran on a single i860 node. This code is shown below:

```
#include <stdio.h>
/* “btioverify.incl” defines the matrix dimension (N) as DIM,
and the number of writes (N_S/I_W) as RPT*/
#include “btioverify.incl”
main()
{
 FILE *fp;
 long i,j,k,n;
 double data;
```

```
 fp = fopen("/cfs/fineberg/btout.bin", "r");
 fp = fopen("btout.bin", "r");

 for (k=0; k<RPT; k++){
   fseek(fp, 5L*8L*(DIM*DIM*DIM)*k, 0);
   for (i=0; i<DIM; i++){
      for (j=0; j<5; j++){
         fread(&data, 8, 1, fp);
         printf("%15.10lf\n", data);
         }
      fseek(fp, (5L*8L*(DIM*DIM) + DIM*5L*8L), 1);
      }
   }
 fclose(fp);
}
```

4.0 Results

4.1 Cray Y-MP

Results for an 8 processor Cray Y-MP in dedicated time are shown in Table 2 and

TABLE 2. Results for CrayBT based benchmark

N	*No. Proc.*	T_C *(secs)*	T_T *(secs)*	R_{IO} *(bytes/sec)*	ζ
64	8	117.9	141.7	2442810	0.202
102	1	3554	3740	453993	0.052
102	2	1816	1857	914342	0.023
102	4	930.6	975.3	1740934	0.048
102	6	645.0	774.5	2192295	0.201
102	8	506.4	694.8	2443772	0.372

are plotted in Figure 2. This benchmark implementation was the only one that obtained true "supercomputer" performance. With 8 processors, the Cray ran at 1.38 GFLOPS without I/O and 1.08 GFLOPS with I/O. Note that the fast computation speed exaggerates the overhead of I/O. The overhead measurement here (0.372) was the largest of any of those measured in this study. However, R_{IO} was the highest measured and the execution times were the lowest.

4.2 Thinking Machines CM-2

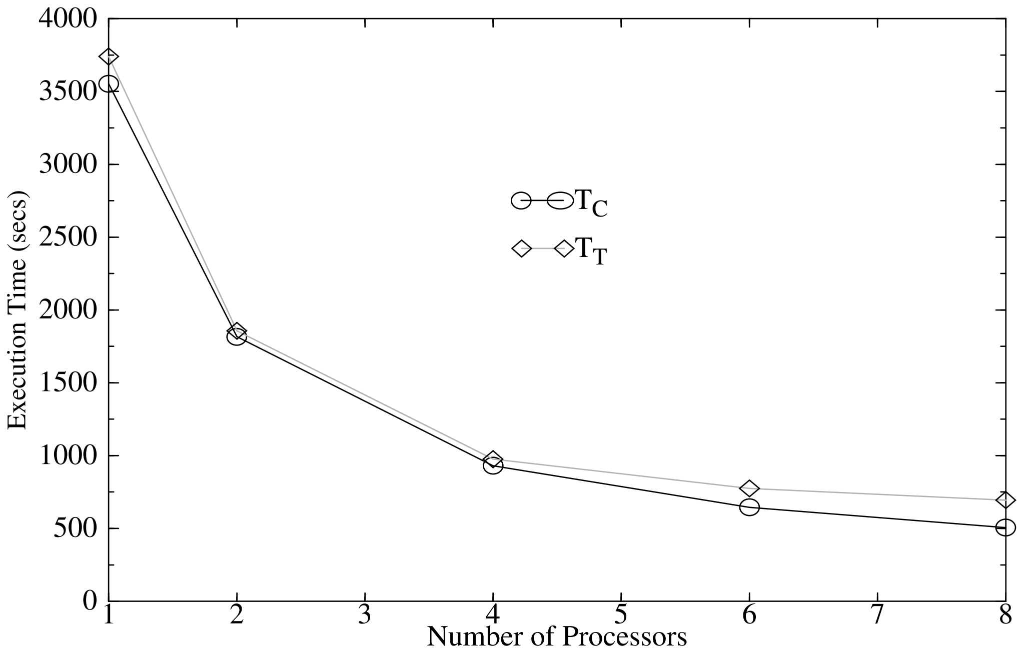
In Table 3, the data collected for two values of *N*, 12 and 64, is listed using the execution time of SampleBT (without the addition of I/O code) as T_C.

TABLE 3. Results for SampleBT/CMF based benchmark

N	T_C *(secs)*	T_T *(secs)*	R_{IO} *(bytes/sec)*	ζ
12	60.25	81.85	5067	0.359
64	5526	5621	74618	0.017

As can be seen from these results, the runtime of SampleBT/CMF was quite large. This generally distorts results by making R_{IO} and ζ too low (note that while ζ was

Figure 2: Cray Y-MP/8 Performance



not too low for N=12, this does not occur for any other problem sizes on the CM-2 or for the other machines). A better BT implementation was RNRBT/CMF. Its results are shown in Table 4 for N=64 and N=102.Note that the N=102 execution time is an approximation determined from running the benchmark for 1/8 of the total number of iterations and multiplying the result by 8. This was necessary because the amount of run-time needed for the large benchmark on the CM-2 was prohibitive.

TABLE 4. Results for RNRBT/CMF based benchmark

N	T_C *(secs)*	T_T *(secs)*	R_{IO} *(bytes/sec)*	ζ
64	3754	3901	107519	0.039
102[a]	63426	64769	26215	0.021

a. approximated by running for 25 iterations and multiplying elapsed time by eight.

4.3 Intel iPSC/860

Experiments were performed on the iPSC using the SampleBT/iPSC code for N=12 and 64 with 16 and 64 processors respectively. These results are summarized in Table 5. As with the CM-2, the results with this inefficient BT implementation

TABLE 5. Results for SampleBT/iPSC based benchmark

N	T_C *(secs)*	T_T *(secs)*	R_{IO} *(bytes/sec)*	ζ
12	37.96	39.82	10415	0.049
64	6663	9262	45285	0.390

yield low results for both R_{IO} and ζ.

In addition, the faster RNRBT_3D/iPSC code was run for N=12, 64, and 102 with a varying number of processors. These results are shown in Table 6. Note that for

TABLE 6. Results for RNRBT_3D /iPSC based benchmark

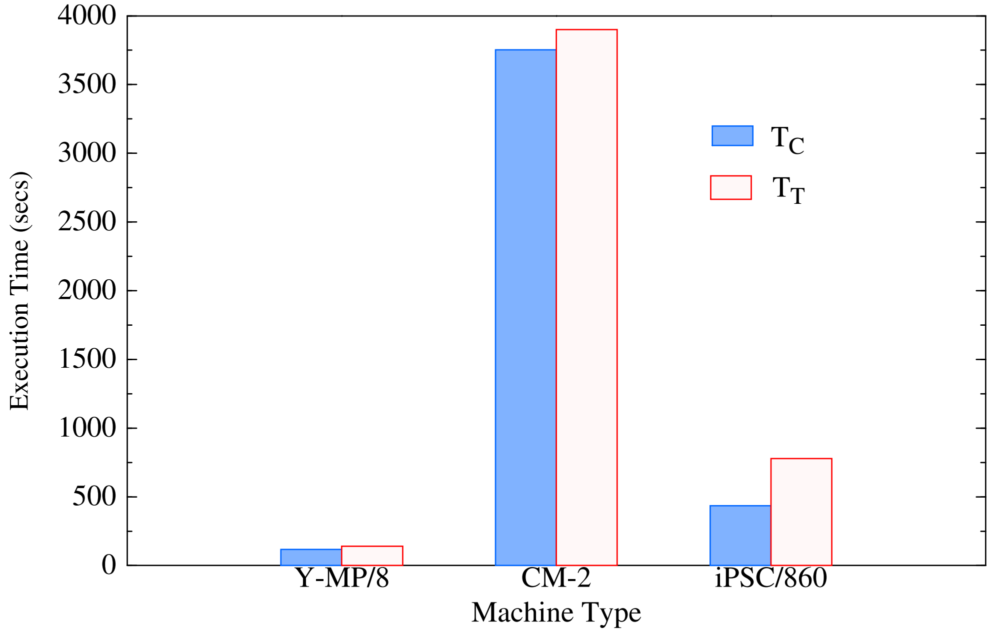
N	*No. Proc.*	T_C *(secs)*	T_T *(secs)*	R_{IO}*(bytes/sec)*	ζ
12	8	9.694	10.20	40659	0.052
64	32	1381	1567	267665	0.135
64	64	753.0	1036	404856	0.376
64	128	435.6	779.9	537800	0.719
102	128	1566	2663	637602	0.701

N=12 and 102, only a single machine size was possible. RNRBT_3D/iPSC requires the processors to be laid out as a 3 dimensional grid with sides that are powers of two. For N=12, the data distribution was too sparse for more than 8 processors (a 2x2x2 grid). For N=102, all 128 processors had to be used to have enough memory to store the programs data set. The 128 processors were laid out as a 4x4x8 grid. For N=64, three machine sizes were possible including 32 (2x4x4 grid), 64 (4x4x4 grid), and 128 (4x4x8 grid) processors.

5.0 Analysis

In high performance computer systems, balance can be defined as the property a system exhibits when all of its components are well matched in performance. Therefore, a “well-balanced” I/O system should be capable of performing its required tasks at a rate commensurate with the system’s computational performance. Unfortunately, balance is not only an application dependant characteristic, but is also subjective. However, as a goal, scientists at NAS generally consider an overhead of 10% (0.1) to be acceptable [Wee92]. In a balanced system, one would hope to keep ζ low, though a ζ that is “too” low may indicate I/O hardware that has a higher capacity than is required for a system’s computational capacity. One also wants to decrease execution time as much as possible, thus keeping R_{IO} high. The goals of increasing R_{IO} and decreasing ζ may be contradictory if I/O performance does not scale with computation performance. Consider Figure 3. In this graph, the performance of the three machines is compared. First, note the difference in aggregate system performance, R_{IO}. Clearly, the Y-MP is the fastest machine, followed by the iPSC/860, with the CM-2 being the slowest. These are reflected in the val-

Figure 3: System Comparison for N=64

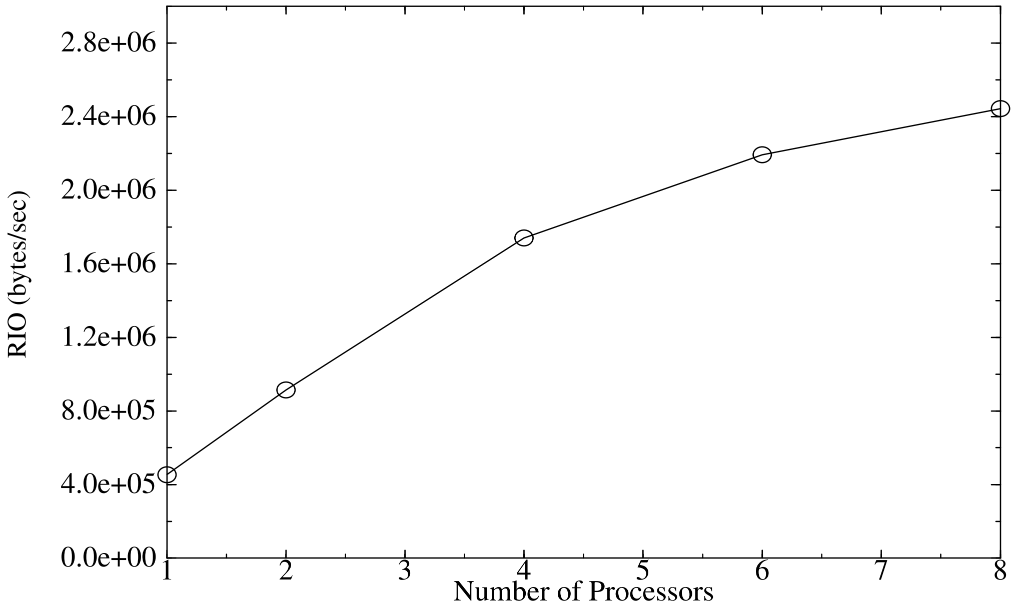


ues of R_{IO} shown in Tables 4, 6, and 8. These are 2.3 MBytes/sec for the Y-MP, 0.61 MBytes/sec for the iPSC/860, and 0.025 MBytes/sec for the CM-2. (Note, the iPSC/860's performance is still repeatable given the approximately 10 times price difference between the iPSC/860 and the Y-MP.)

What is of more interest, however, is the added execution time due to I/O. For the CM-2, the T_T is only slightly greater than T_C. However, for the iPSC/860, T_T is almost twice T_C. This indicates that while the iPSC/860's overall performance is significantly better than the CM-2, its I/O performance is much worse relative to its computation performance. In terms of system balance, neither of these machines is well balanced. The iPSC/860 lacks adequate I/O performance, and the CM-2 has I/O performance greater than that needed for its computational power. T_C and T_T for the Y-MP seem to be equal, however, the relative difference between T_T and T_C falls between that of the other two machines. Thus, T_T is significantly greater than T_C (by about 20%, see Table 2), but the difference is not nearly as great as the relative difference between T_C and T_T for the iPSC/860. This indicates that the Y-MP is balanced such that I/O only adds 20% to execution time. This is better than the iPSC/860 and indicates a better balance between computation and I/O performance than the CM-2. However, it still does not reach the goal of 10% overhead.

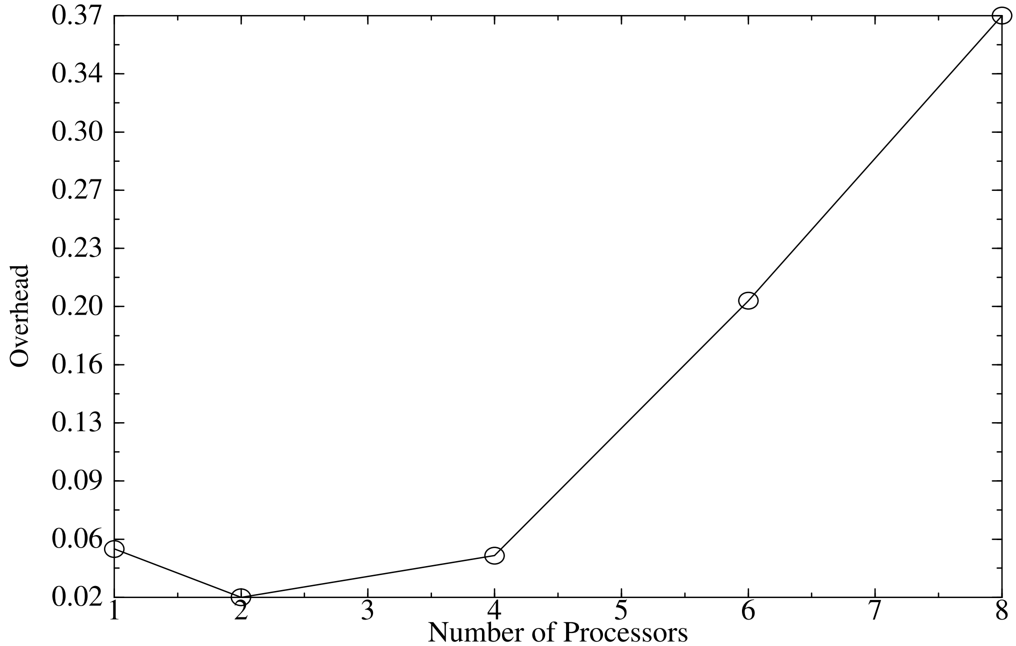
Consider what happens if we change the computational performance of the Cray Y-MP (by varying the number of CPUs). As the number of processors is increased, T_C and T_T decrease (see Figure 2). The change in computation rate can be observed using R_{IO} which is a measure of overall system performance (see Figure 4). Therefore, as the number of processors, and therefore the computational performance is increased, R_{IO} increases proportionately. However, note that in Figure 2

Figure 4: Cray Y-MP/8 I/O Rate



the difference between T_C and T_T increases as the number of processors is increased. To better illustrate this, Figure 5 plots ζ (overhead) vs. number of processors used on the Y-MP. As can be seen, the I/O overhead increases steadily as

Figure 5: Cray Y-MP/8 Overhead



the number of processors is increased. This is because the I/O rate is not being scaled with the computational performance (i.e., the number of processors). Rather, the I/O transfer rate remains approximately constant with two or more processors. Because computation time is decreasing as more processors are added and

I/O time is remaining relatively constant, the added overhead due to I/O increases. Thus, balance is dependent on both computation and I/O performance and neither alone is a good system metric. Further, we see that the Cray Y-MP could use slightly better I/O performance to achieve the 10% goal for 8 CPUs and reaches this goal for balance with approximately 5 CPUs. While this may be unimportant for an existing machine (i.e., an application running on an existing machine will generally use all available processors), it indicates that for a more powerful system (e.g., a Y-MP C90) the I/O performance must be scaled with computational performance.

Now, consider what happens when a sytem is out of balance. First, there is the CM-2, which is an example of a machine with a low R_{IO}, and I/O performance that is relatively high. Consider the data for RNR_BT/CMF with N=64. R_{IO} is low, 107519 bytes/sec, but overhead is also low, 3.9% (0.039). This indicates a system that is out of balance relative to the 10% goal in that the computation rate is too low for the I/O performance. Next, there is the iPSC/860. For N=64, R_{IO} is 537800 bytes/sec, five times that of the CM-2, though not as good as the Y-MP. However, the iPSC/860's I/O performance is less than both the Y-MP and the CM-2. This results in an overhead of 71.9% (0.719), i.e., 71.9% of the time required to run the I/O benchmark is spent writing data. This indicates that the I/O system is slower than is needed to achieve the 10% goal with the available computation performance.

6.0 Conclusions

In this paper it has been shown that the NHT-1 application I/O benchmark is a measure of both absolute system performance and balance for a given type of computational and I/O workload. System performance is indicated by the R_{IO} metric by measuring the total run time of an application that includes significant quantities of I/O and dividing that time by the total amount of I/O performed. This allows R_{IO} to reflect both the system's computational and its I/O performance. Therefore a system with a high R_{IO} must achieve high performance for applications with significant amounts of I/O. However, what R_{IO} does not indicate is the relative balance of I/O and computational performance. Balance is then indicated by ζ. A machine with a high R_{IO} and a ζ around 0.1 is a fast, well-balanced machine. A machine with a high ζ, is unbalanced in that its I/O is too slow in proportion to its computational power. If ζ is low, however, the machine is still unbalanced, in that the I/O performance is higher than is warranted for the system's computational performance.

In this paper we have seen results from three machines, none of which were perfectly balanced. The CM-2 is quite slow in absolute performance, but has I/O that is relatively fast (particularly given its lack of computational power). The iPSC/860 is relatively fast, but has I/O that is much too slow for its computational power. The Cray Y-MP is relatively well balanced but could use slightly faster I/O for applications that use all 8 processors.

These results indicate that no single benchmark metric is appropriate for measuring application I/O performance. Rather, by measuring both absolute performance and balance, it is possible to get a good sense for how a machine will perform under loads that include significant amounts of disk I/O. These two aspects are measured by the R_{IO} and ζ metrics that are generated by the NHT-1 Application I/O benchmark. Thus, the benchmark allows one to determine if a system configuration meets absolute performance goals and has an appropriate amount of I/O performance relative to its computational power.

7.0 Acknowledgments

The author of this paper would like to acknowledge Russell Carter, Bill Nitzberg, and Bernard Traversat for their help in preparing this paper, as well as the members of the High Speed Processor (Cray support) and Parallel Systems support staff at NAS for putting up with the abuse applied to their systems while collecting the data for this paper.

8.0 References[2]

[BaB91] D. Bailey, J. Barton, T. Lasinski, and H. Simon, eds, *The NAS Parallel Benchmarks, Revision 2*, Technical Report RNR-91-002, NASA Ames Research Center, Moffett Field, CA, July 1991.

[BaB92] D. H. Bailey, E. Barszcz, L. Dagum, and H. D. Simon, NAS Parallel Benchmark Results, Technical Report RNR-92-002, NASA Ames Research Center, Moffett Field, CA, August 1992.

[CaC92] R. Carter, B. Ciotti, S. Fineberg, and B. Nitzberg, *NHT-1 I/O Benchmarks*, Technical Report RND-92-016, NASA Ames Research Center, Moffett Field, CA, November 1992.

[Cio92] B. Ciotti, "Session Reservable File Systems (SRFS)," *Spring 1992 CRAY User Group Proceedings*, Berlin, Germany, 1992.

[Hil87] W. D. Hillis, "The connection machine," *Scientific American*, vol. 256, June 1987, pp. 108-115.

[Int91] Intel Supercomputing Systems Division, *iPSC/2 and iPSC/860 User's Guide,* Intel Corporation, Beaverton, OR, 1991.

[Nit92] B. Nitzberg, *Performance of the iPSC/860 Concurrent File System*, Technical Report RND-92-020, NASA Ames Research Center, Moffett Field, CA,

2. NAS technical reports may be obtained by sending e-mail to `doc-center@nas.nasa.gov`.

December 1992.

[Nug88] S. F. Nugent, “The iPSC/2 direct connect communications technology,” *Third Conference on Hypercube Concurrent Computers and Applications*, January 1988, pp. 51-60.

[Wee92] S. Weeretunga, NAS Applied Research Department, personal communication, October 1992.

[ZeL88] S. A. Zenios, R. A. Lasken, “The connection machines CM-1 and CM-2: solving nonlinear network problems,” 1988 International Conference on Supercomputing, July 1988, pp. 648-658.