NAS Parallel Benchmarks I/O Version 2.4

Parkson Wong, Rob F. Van der Wijngaart
Computer Sciences Corporation
NASA Advanced Supercomputing (NAS) Division
NASA Ames Research Center, Moffett Field, CA 94035-1000
{parkson,wijngaar}@nas.nasa.gov

NAS Technical Report NAS-03-002

January 2003

Abstract

We describe a benchmark problem, based on the Block-Tridiagonal (BT) problem of the NAS Parallel Benchmarks (NPB), which is used to test the output capabilities of high-performance computing systems, especially parallel systems. We also present a source code implementation of the benchmark, called NPBIO2.4-MPI, based on the MPI implementation of NPB, using a variety of ways to write the computed solutions to file.

1 Introduction

The original NAS Parallel Benchmarks (NPB) [1], whose paper-and-pencil specification was released in 1992, and whose first complete parallel implementation [2] using MPI [6] (NPB, version 2.3) was released in 1997, aimed to provide an objective measure of the computational capabilities of modern high-performance computing systems. As improvements were made to parallel systems, and as scalability and absolute performance increased accordingly, a new bottleneck became evident in practical applications, namely the speed with which computed results were being written to and read from file. Limitations to I/O performance were due in part to the fact that application developers and users require data files whose structure does not depend on the number of processors participating in the generation of the files. This is important when the program that creates the data is not the same as that reading it, i.e. when post-processing takes place. We focus on such applications, and hence will be primarily concerned with the *output* capabilities of the system under consideration. In the case of scientific computations using domain decomposition as a workload distribution method, the multiple participating processors would typically each write a number of fragments of an output file. Depending on the number of processors and the specifics of the domain decomposition, the number of fragments contributed by each processor

could be very large, leading to many disk accesses and competition among processors, which would often degrade performance severely.

It was soon recognized that improvements could be otained by making use of the fact that aggregate inter-processor communication speeds were significantly higher than I/O speeds. Rearranging data in memory prior to writing to file could greatly reduce the number of disk accesses. This realization was behind much of the thinking that went into the definition of the MPI I/O Application Programmer Interface [3], which formally became a part of the MPI, version 2, specification in 1997 [5]. This API relied heavily on the powerful abstract data type definition functions already contained in MPI, version 1, which can be used to specify the relationship between the locations of data stored in (distributed) memory and data stored on disk. An MPI I/O library call can then use this relationship to optimize data traffic without further programmer interference, employing collective buffering [4].

2 BTIO

A good case for testing the speed of parallel I/O was provided by the Block-Tridiagonal (BT) NPB problem, whose MPI implementation employs a fairly complex domain decomposition called diagonal multi-partitioning [7]. Each processor is responsible for multiple Cartesian subsets of the entire data set, whose number increases as the square root of the number of processors participating in the computation. A naive implementation in which each processor writes directly into an output file those data elements for which it is responsible will typically exhibit abysmal performance due to a very high degree of fragmentation. Hence, this problem and its implementation could serve as good candidates for I/O performance improvements through collective buffering. A first such implementation—named BTIO—using MPI I/O was described by Fineberg *et al.* in [4].

In this report we formalize the specification of a benchmark derived from that effort, and describe the latest implementation, which is being released under the name NPBIO2.4-MPI. For specifics of the numerical algorithm we refer the reader to the NPB report [1]. I/O requirements and verification tests are as follows. After every five time steps the entire solution field, consisting of five double-precision words per mesh point, must be written to one or more files. After all time steps are finished, all data belonging to a single time step must be stored in the same file, and must be sorted by vector component, x-coordinate, y-coordinate, and z-coordinate, respectively. Any rearrangement of data in the file required to produce this layout must be included in the timing of the benchmark. It is not necessary, but allowed, to store different time steps in different files.

After the timing is stopped, the computed and stored solution fields are verified as follows. The computed residual field must pass the same verification test as specified in [1]. The solution field in memory is zeroed or otherwise voided, after which all stored solutions are read back from file. The norm of each solution error is computed in the way prescribed by [1], and is accumulated in a vector of double precision numbers of size five, upon which it is divided by the number of outputted solutions fields. Let s be the sequence number of the complete solution field written to file, out of a total of S

fields. Let $u^s_{i,j,k,m}$ signify the m^{th} component of that field at point (i, j, k) of a mesh of size $nx \times ny \times nz$ points, and let the corresponding "exact" (i.e. steady-state) solution at that point be $\overline{u}_{i,j,k,m}$. Then the following inequality must hold:

$$\left| \frac{1}{S} \sum_{s=1}^{S} \frac{1}{nx \cdot ny \cdot nz} \left(\sum_{i=1}^{nx} \sum_{j=1}^{ny} \sum_{k=1}^{nz} (u^s_{i,j,k,m} - \overline{u}_{i,j,k,m})^2 \right)^{1/2} - \delta u_m \right| / \delta u_m \leq \epsilon \,,$$

where δu_m is the verification value given in the Appendix, and ϵ is 10^{-8}.

3 Verification rationale

Due to the slowness of I/O devices compared to the CPUs, much effort has been spent, almost since the inception of digital computing, on asynchronous, buffered I/O. On many systems the semantics of write operations do not require data actually to have reached an external storage device upon completion of the operation. System buffers may hold large amounts of data to be written to disk, depending on the amount of memory available. Now imagine the following strategy for verifying correctness of our I/O benchmark. After a solution has been written to a file, void the solution array and read back the solution that has just been stored. This would simplify the procedure, since the original numerical verification test could only succeed if all read/write pairs would have succeeded. Moreover, since a solution is no longer needed once it has been written and retrieved, total space for the output file can be limited to a single time step. This is a significant benefit for the larger problem sizes. For example, Class D requires more than 135 GB of disk space to store all the required solutions, as opposed to 3 GB for only one solution. However, reading the output data piecemeal immediately after it has been written greatly increases the chances that much of it still resides in system buffers, so that not disk I/O but plain memory speed would be measured. While this is in principle impossible to avoid, we can reduce its likelihood by postponing the read operations to the end of the program.

4 MPI implementation

Like all other NPB this I/O benchmark is required to report completion time, including writing the data to file. In the NPB version 2.3 implementation we derive from the completion time the number of floating point operations executed per second (flops), since the total number of such operations is known. In this I/O benchmark the situation is less clear cut, however, since there is time spent on I/O as well as on computation. Since good systems will always provide buffered, asynchronous I/O, it is not possible to compute output bandwidth based on completion time and number of bytes written, not even if all I/O library calls were timed individually. This is because part of outputting the data may take place under control of a spawned process that cannot be timed directly and whose execution overlaps with the computation. Depending on what is considered the main result of the program, it is possible, though, to compute

effective flops or *effective output bandwidth*. They are defined as the total number of floating point operations and the total number of bytes written, respectively, each divided by the wall clock time spent between the beginning of the first time step and the verification of the solution. In the NPB 2.4 implementation we return effective flops. In the Appendix we provide the number of bytes written by each class of the benchmark, so that the effective output bandwidth can be determined as well. Finally, we point to the possibility of computing *output overhead* of the benchmark program by running it with and without I/O and subtracting the completion times. This is governed by the parameter `SUBTYPE`, which can assume the values `full`, `simple`, `fortran`, or `epio`, or be empty. If empty, no I/O of significance takes place, and the original BT benchmark obtains. The nonempty parameter values have the following effect:

1. `full`: MPI I/O *with* collective buffering [5]. This means that data scattered in memory among the processors is collected on a subset of the participating processors and rearranged before written to file in order to increase granularity.

2. `simple`: MPI I/O *without* collective buffering [5]. This means that no data rearrangement takes place, so that many seek operations are required to write the data to file.

3. `fortran`: Same as 2, but now plain Fortran 77 file operations are used instead of MPI I/O library calls.

4. `epio`: This option does not conform to the benchmark requirements, because each participating process writes the data belonging to its part of the domain to a separate file as a contiguous stream of data. It could be made to conform by merging the individual files into a single file afterwards and including the time spent in this operation in the benchmark completion time. However, we leave it as it is, since it gives a realistic measure of the maximum achievable I/O speed.

References

[1] D. Bailey, E. Barscz, J. Barton, D. Browning. R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, S. Weeratunga. *The NAS Parallel Benchmarks.* NAS Technical Report RNR-94-007, NASA Ames Research Center, Moffett Field, CA, 1994.

[2] D. Bailey, T. Harris, W. Saphir, R. Van der Wijngaart, A. Woo, M. Yarrow. *The NAS Parallel Benchmarks 2.0.* NAS Technical Report NAS-95-020, NASA Ames Research Center, Moffett Field, CA, 1995.

[3] P. Corbett, D. Feitelson, S. Fineberg, Y. Hsu, B. Nitzberg, J. Prost, M. Snir, B. Traversat, P. Wong Overview of the MPI-IO Parallel I/O Interface. in *Input/Output in Parallel and Distributed Computer Systems.* R. Jain, J. Werth, J. Browne (eds.), pp. 127-143, Kluwer Academic Publishers, 1996.

[4] S. Fineberg, P. Wong, B. Nitzberg, C. Kuszmaul. *PMPIO—A Portable Implementation of MPI-IO.* Proc. Sixth Symposium on the Frontiers of Massively Parallel Computation, pp. 188–195, IEEE Computer Society Press, October 1996.

[5] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, W. Nitzberg, W. Saphir, M. Snir. *MPI: The Complete Reference (Vol. 2).* MIT Press, 1998.

[6] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra. *MPI: The Complete Reference.* MIT Press, 1996.

[7] R. Van der Wijngaart. *Charon message-passing toolkit for Scientific computations.* 7^{th} Int'l Conf. High Performance Computing, Bangalore, India, December 17-20, 2000.

Appendix: Verification values

Here we list the values δu_m of the difference between steady-state and average intermediate values of the solution of the I/O benchmark. See the equation on page 3.

Class	m	δu_m	Mbytes written
S	1	$0.1149036328945 * 10^2$	0.83
	2	$0.9156788904727 * 10^0$	
	3	$0.2857899428614 * 10^1$	
	4	$0.2598273346734 * 10^1$	
	5	$0.2652795397547 * 10^2$	
W	1	$0.6729594398612 * 10^2$	22.12
	2	$0.5264523081690 * 10^1$	
	3	$0.1677107142637 * 10^2$	
	4	$0.1508721463436 * 10^2$	
	5	$0.1477018363393 * 10^3$	
A	1	$0.6482218724961 * 10^2$	419.43
	2	$0.5066461714527 * 10^1$	
	3	$0.1613931961359 * 10^2$	
	4	$0.1452010201481 * 10^2$	
	5	$0.1420099377681 * 10^3$	
B	1	$0.1477545106464 * 10^3$	1697.93
	2	$0.1108895555053 * 10^2$	
	3	$0.3698065590331 * 10^2$	
	4	$0.3310505581440 * 10^2$	
	5	$0.3157928282563 * 10^3$	
C	1	$0.2597156483475 * 10^3$	6802.44
	2	$0.1985384289495 * 10^2$	
	3	$0.6517950485788 * 10^2$	
	4	$0.5757235541520 * 10^2$	
	5	$0.5215668188726 * 10^3$	
D	1	$0.3813781566713 * 10^3$	135834.62
	2	$0.3160872966198 * 10^2$	
	3	$0.9593576357290 * 10^2$	
	4	$0.8363391989815 * 10^2$	
	5	$0.7063466087423 * 10^3$	