CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2012; **24**:155–166
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.1888

SPECIAL ISSUE PAPER

Performance analysis of Intel multiprocessors using astrophysics simulations

Tyler A. Simon[1,*,†], William A. Ward Jr.[2] and Alan P. Boss[3]

[1]*NASA Center for Climate Simulation, Goddard Space Flight Center, Greenbelt, MD, USA*
[2]*DoD High Performance Computing Modernization Program, Lorton, VA, USA*
[3]*Department of Terrestrial Magnetism, Carnegie Institute of Washington, WA, USA*

SUMMARY

This paper provides a performance evaluation and investigation of the astrophysics code FLASH for a variety of Intel multiprocessors. This work was performed at the NASA Center for Computational Sciences (NCCS) on behalf of the Carnegie Institution of Washington (CIW) as a study preliminary to the acquisition of a high-performance computing (HPC) system at the CIW and for the NCCS itself to measure the relative performance of a recently acquired Intel Nehalem-based system against previously installed multicore HPC resources. A brief overview of computer performance evaluation is provided, followed by a description of the systems under test, a description of the FLASH test problem, and the test results. Additionally, the paper characterizes some of the effects of load imbalance imposed by adaptive mesh refinement. Copyright © 2011 John Wiley & Sons, Ltd.

Received 11 February 2011; Revised 10 July 2011; Accepted 22 August 2011

KEY WORDS: FLASH; performance analysis; multicore

1. BACKGROUND

Computer performance evaluation is of primary interest when computer systems are designed, selected, or improved (tuned) [1]. This perspective may be expanded to include not just vendor-provided hardware and system software but also application software. Computer scientists use one or more approaches when addressing these situations, including mathematical analysis, simulation, and benchmarking. See [2, Chapter 3] for an excellent discussion of the selection process. Because of the advantages in timeliness and accuracy, benchmarking, defined as the use of one or more software components to measure computer system performance, has been adopted for this study. Given that benchmarking is the appropriate approach, further refinement of the process may be accomplished on the basis of how well the approach meets certain test goals. In a system procurement context such as the one under consideration here, among the most important are minimization of time and cost and maximization of benchmark representativeness [3]. The importance of the former goal is self-evident. Achieving the latter goal is critical because it relates to how well the benchmark test mirrors the way in which the system will be used in practice. Finally, because of the high-performance computing (HPC) context of this study, we may add maximization of test component scalability, in the sense that the software test components should measure whether performance improves (run time decreases) as more CPUs are applied to the problem.

*Correspondence to: Tyler A. Simon, NASA Center for Climate Simulation, Goddard Space Flight Center, Greenbelt, MD 20706, USA.
†E-mail: tyler.simon@nasa.gov

These goals guide further decisions regarding the conduct of the test. The software test components may be (i) synthetic programs developed for the sole purpose of measuring system performance but which otherwise do no useful work [4–6], (ii) software kernels extracted from larger programs [7–10], or (iii) actual applications [11]. The benchmark test mode is also important; the test components may be run on a 'dedicated' system without interference from other programs, on a loaded system competing with otherwise unrelated programs, or as part of a specially constructed workload imposed on the system under test [3]. Various figures of merit may be gathered as part of a test, such as instructions per second, floating-point execution rate, memory access rate, communication rate over an interconnect, I/O rate, and elapsed wall clock time for a run. The purpose of this study is to estimate the performance of a system with performance characteristics similar to what might be acquired in the near future; the current system is a small Intel Xeon-based cluster, which is several years old. Normally, a single code, even if it is an application, is insufficient to thoroughly test a system. However, because FLASH will be the primary application on the new system, because example test input data has been provided, and because the code itself is highly scalable, it met the goals noted previously and was used as the sole benchmark software test component for this study. Further information and guidance on computer performance evaluation in general and benchmarking in particular may be found, for example, in [12–16].

1.1. Systems under test

Most modern high-performance computers are commodity clusters; that is, they are built using components produced in volume for a variety of purposes. These systems are typically constructed of rack-mounted computed units that implement one or more complete, though diskless, computer systems. Each system typically contains one or more Intel or Advanced Micro Devices (AMD) microprocessors, a shared memory, and Ethernet or Infiniband interfaces to connect to an interconnection network that enables communication between the various systems. Floating-point accelerators, such as graphics processing units, are sometimes added to improve performance, but these often require special programming effort to be effective. Finally, the overall hardware system includes an interconnect and an I/O subsystem. The predominant operating system installed on clusters is a version of Linux; for this reason, HPC systems of this type are called 'Linux clusters'. As a minimum, compilers, libraries of mathematical functions, and a library implementing the message passing interface (MPI) are required to complete the software environment. Applications may then be custom written, or obtained from other sources, often from some particular community of researchers.

The system under test used for this study was the Linux cluster, *discover*, at the NASA Center for Computational Sciences (NCCS). It has been constructed beginning with a 'base unit' and expanded with five 'scalable units' (SCUs); each of these consists of multiple racks of computed nodes purchased over time sharing a common interconnect and I/O subsystem. The base unit, purchased from Linux Networx, Inc. (LNXI), contains 126 nodes; each node consists of two sockets, each of which is populated by a dual-core Intel 'Dempsey' processor running at 3.2 GHz.[‡] SCU1 and SCU2, also purchased from LNXI, each contain 258 nodes, each node consisting of two dual-core Intel 'Woodcrest' processors running at 2.66 GHz. SCU3 and SCU4 are International Business Machine (IBM) iDataplex SCUs each with 256 nodes, each node having two quad-core Intel 'Harpertown' processors running at 2.5 GHz.[§] Recently installed SCU5 is also an IBM iDataplex SCU with 516 nodes, each having two native quad-core Intel 'Nehalem' processors running at 2.8 GHz. The Dempsey cores have a peak rate of two double-precision floating-point operations per cycle, whereas the others have a peak rate of four.[¶] This gives *discover* an aggregate theoretical peak performance of 112.8 Tflop/s. Table I summarizes these and other information about the configuration of *discover*. Further information on the architecture of the Intel processors may be found in [17]. The network topology for *discover* consists of a fat tree connected over a double data rate InfiniBand

[‡]This and subsequent clock rates are as reported by the Linux utility *hwinfo*.
[§]Intel refers to the Harpertown as a 'quad-core, dual-die' processor. In point of fact, two dual-core dies sharing a single ceramic package, not a single *native* quad-core chip as in the AMD 'Barcelona' or the Intel 'Nehalem'.
[¶]All floating-point operation rates noted in this study are rates for operations in double precision.

Table I. Base and scalable units on *discover* as of June 30, 2009.

Unit	Base unit	SCUs 1 and 2	SCUs 3 and 4	SCU 5
Vendor	LNXI	LNXI	IBM	IBM
Model	Custom	Custom	iDataplex	iDataplex
Nodes/unit	126	258	258	516
Processors/node	2	2	2	2
Cores/processor	2	2	4	4
Cores/unit	504	1032	2064	4128
Peak Tflop/s/unit	3.33	10.98	20.64	42.93
Memory/node	4 GB	4 GB	16 GB	24 GB
Processor	Intel Dempsey	Intel Woodcrest	Intel Harpertown	Intel Nehalem
Lithography	65 nm	65 nm	45 nm	45 nm
Clock rate	3.2 GHz	2.667 GHz	2.5 GHz	2.8 GHz
Flop/clock/core	2	4	4	4
Level 1 I-cache	2×12K	2×32K	4×32K	4×32K
Level 1 D-cache	2×16K	2×32K	4×32K	4×32K
Level 2 cache	2×2M	4M	2×6M	4×256K
Level 3 cache	None	None	None	8M

SCU, scalable unit; LNXI, Linux Networx, Inc.; IBM, International Business Machine.

interconnect, which has a measured peak bidirectional bandwidth of 2.5 GB/s. Finally, all of the experiments were run under the following software environment:

- Operating system: SUSE Linux Enterprise Server Version 10 Patchlevel 1
- File system: IBM General Parallel File System
- Batch scheduler: Portable Batch System Pro Version 8.0.0.63106
- Compiler: Intel Fortran Compiler Version 10.1.017
- Math library: Intel Math Kernel Library Version 10.0.5.025
- MPI library: Intel MPI Version 3.2.011

1.2. Application used for testing

FLASH, the code used in this performance study, is a research code developed at the Center for Astrophysical Thermonuclear Flashes at the University of Chicago [18]. Version 1, developed in the late 1990s, was designed to study various types of runaway stellar thermonuclear burning events, including X-ray bursts, novae, and Type 1a supernovae. Accretion of material onto the surface of white dwarfs, thermonuclear ignition resulting in convection, and a nova expanding around a sibling binary star are examples of events that FLASH models. Physics featured in the code include compressible hydrodynamics implemented using an explicit high-order Godunov solver, user-selectable equations of state, an optionally enabled nuclear-reaction network, and external gravity. Interstellar models, electron degeneracy and radiation pressure is allowed [19]. Since its initial release, FLASH has been continuously enhanced, and new features have been incorporated; version 2.5 of the code was used in this study.

FLASH, written in Fortran 90, has three important characteristics [20]. First, it is modular in the sense that it has been designed to allow users to easily specify different initial and boundary conditions, supply different algorithms, and add their own new physical properties calculations. Secondly, it uses adaptive mesh refinement (AMR) to overcome the limitations that would be imposed by a uniformly high-resolution equispaced grid; the AMR package used here is PARAMESH [21], developed at the NASA Goddard Space Flight Center. Finally, FLASH was designed from the outset to be highly scalable so as to run efficiently on thousands of processors; MPI is used to implement parallelism in a portable fashion. The source code itself is broken down into subdirectories for the driver, AMR, hydrodynamics, equations of state, thermonuclear burning, gravitational effects, and

I/O. Further information on FLASH and its performance on high-performance computers may be found in [22–24]. Prospective users may consult the user's guide [25].

1.3. Data gathering instrumentation

The MPI standard requires that a compliant MPI library be implemented so that each MPI routine be implemented by calling a corresponding profiling interface routine; for example, routine MPI_Send calls PMPI_Send so that MPI_Send essentially serves as a wrapper. Integrated Performance Monitoring (IPM) [26], the software instrumentation tool used in this study, takes advantage of this aspect of MPI by using code in the wrapper layer to gather data about the message passing behavior of a code. Because the code that performs the instrumentation is completely contained inside MPI routines, IPM may be enabled by merely linking with the IPM version of the MPI library; those routines, in turn, call the corresponding PMPI routines to do the actual message passing work. Although not measured as part of this study, anecdotal information indicates that IPM overhead is less than about 5% of job walltime.

2. PERFORMANCE TESTING

2.1. Problem definition

An important study to be performed with FLASH on the to-be-acquired system will investigate the mechanisms of planetary formation. Cosmochemical evidence for the existence of short-lived radioisotopes (SLRI) such as ^{26}Al and ^{60}Fe at the time of the formation of primitive meteorites requires that these isotopes were synthesized in a massive star and then incorporated into chondrites within about 1 Myr. A supernova shock wave has long been hypothesized to have transported the SLRI to the presolar dense cloud core, triggered cloud collapse, and injected the isotopes. Previous numerical calculations have shown that this scenario is plausible when the shock wave and dense cloud core are assumed to be isothermal at about 10 K but not when compressional heating to about 1000 K is assumed. We have shown for the first time that when calculated with FLASH, a 20 km/s shock wave can indeed trigger the collapse of one solar mass cloud while simultaneously injecting shock wave isotopes into the collapsing cloud, provided that cooling by molecular species such as H_2O, CO_2, and H_2 is included [27, 28]. These calculations imply that the supernova trigger hypothesis is the most likely mechanism for delivering the SLRI present during the formation of the solar system.

2.2. Computational methods

FLASH employs a block-structured adaptive grid approach. Advection is handled by the piecewise parabolic method, featuring a Riemann solver at cell boundaries that handles shock fronts exceptionally well. In previous work [27, 28], Boss used the two-dimensional (2D), cylindrical coordinate (R, Z) version of FLASH, with axisymmetry about the rotational axis (z). Multipole gravity was used, including Legendre polynomials up to $l = 10$. However, there is a clear need to extend the models to be fully three dimensional (3D), and with a current cluster of 48 Xeon processors (purchased in 2003), only very-low-resolution models in 3D may be run, much lower than the resolution found to be necessary to achieve a good result in 2D (i.e., one where the results are beginning to converge to the continuum limit). Prior to purchasing a new cluster that would be capable of handling the 3D problem, it was desirable to run a number of FLASH code benchmarks on the *discover* cluster at the NCCS. For this 3D benchmark testing, we set the number of blocks in x and z to be five and in y (the long axis of the calculational volume) to be 15, with each block consisting of 8×8 grid points, equivalent to an initial grid of $40 \times 40 \times 120$. With four levels of refinement allowed, FLASH is then able to follow small-scale structures with the effective resolution of a grid eight times finer in scale, or effectively $320 \times 320 \times 960$, comparable with the resolutions run in some of our 2D models of shocked cloud cores [27, 28]. The three different versions of the test case differ only in the setting of the input parameter $nend$ that specifies the number of time steps in the simulation run; values of 10, 100, and 1000 were considered to systematically increase the duration of the run.

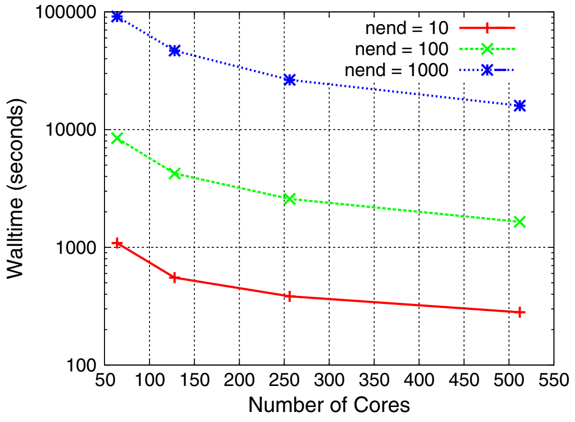

Figure 1. FLASH walltimes in seconds on *Harpertown* nodes.

2.3. *Preliminary tests*

Initially, the code was compiled with -O2 optimization using the Intel Fortran compiler (ifort). According to the ifort manual page, this level enables constant propagation, copy propagation, dead-code elimination, global code scheduling, global control speculation, global instruction scheduling, global register allocation, induction variable simplification, inlining of intrinsics, interprocedural optimization, loop unrolling, operator strength reduction, peephole optimization, predication, software pipelining, speculation, and tail recursion among other code optimizations. We also ran tests compiled at -O3 in order to perform the more aggressive optimizations, such as data prefetching, loop transformations, memory access transformations, and scalar replacement. Refer to [29] for a thorough explanation of compiler optimizations. We did not see any runtime performance improvement when run at -O3; thus, the tests were carried out with -O2. The same binary executable was used for all of the test runs.

The first series of runs investigated run-to-run variation. Figure 1 shows the results of these tests on Harpertown nodes of *discover*. Five FLASH runs were performed at 32, 64, 128, 256, and 512 cores for $nend = 10$, $nend = 100$, and $nend = 1000$; these runs used fully populated nodes.[‖] For the former case, standard deviation seemed relatively constant, until a percent of the mean approached 10% for 512 cores. For the longer run, the percent standard deviation was uniformly less than 1%. A job that is subject to high variation in walltime over repeated runs is unsuitable as a test case; consequently, runs with $nend = 10$ were not used as the basis for performance comparisons. Another concern was that a too small number of time steps would not demonstrate the steady state behavior of FLASH. Consequently, additional runs with $nend = 1000$ were performed for each of the same five core counts noted previously and the times compared with the results for $nend = 10$ and $nend = 100$. They show that FLASH continues to demonstrate good scalability for increasing numbers of time steps. More specifically, the nearly identical slopes of the three performance profiles in Figure 1 indicate nearly identical scaling behavior for this test case regardless of the number of time steps. This behavior implies that the work within a single time step has been effectively parallelized and that runs at large numbers of time steps would not be required for this study.

[‖]It is possible to distribute the same number of MPI processes across nodes in various ways. For instance, a 128-process job may be run on a system with four cores per node by allocating 32 nodes with four processes per node (fully populated nodes), 64 nodes with two processes per node (two idle cores per node), or 128 nodes with one process per node (three idle cores per node). Spreading the processes out as in the latter two cases often results in faster interprocess communication and better memory performance because there are fewer processes on a node using the same off-node bandwidth. However, because this approach prevents the additional nodes from being used by other jobs and lowers overall system throughput, its use is not encouraged on NCCS systems, and jobs are charged for use of the entire node even if some cores are idle.

As a final confirmation of this observation, a FLASH job was submitted to run for 10,000 time steps. Unfortunately, this job stopped progressing at time step 4322 (although it did not abort). Further analysis of this run revealed that times for individual steps were relatively consistent and under 20 s per step up to the vicinity of time step 2400. Then, there would be occasional time steps that took 40–60 s or more; these occurred with increasing frequency until the job hung, as shown in Figure 2. It is believed that increasingly frequent episodes of adaptive mesh refinement caused the increase in time per step, but confirmation of this hypothesis was beyond the scope of this study. Although it is obvious that this episodic behavior will negatively impact code scalability if it increases in frequency, it is nevertheless possible to claim that within the time step window under observation, average time per step is not severely impacted. Therefore, with the preliminary data, the performance study continued using the $nend = 100$ input cases, but without repeated runs.

2.4. *Processor performance comparisons*

Runs were performed for 32, 64, 128, and 256 cores for all four processor types available on *discover*. Insufficient Dempsey-based nodes were available for a 512-core job, so those runs were performed only on the other three types. Figure 3 shows the walltimes in seconds for these runs (smaller is better). Relative performance will be discussed subsequently, but it is clear from these data that the Nehalem processor typically outperforms the others by a factor of two or more at all processor counts. Figure 4 compares SCUs using performance relative to the oldest SCU on *discover*. Specifically, the reciprocal of each walltime (the performance) from the data in Figure 1 was

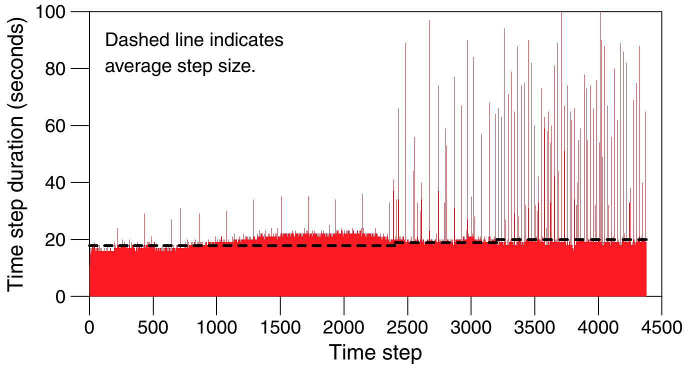

Figure 2. Time step variation in a 512-core FLASH job.

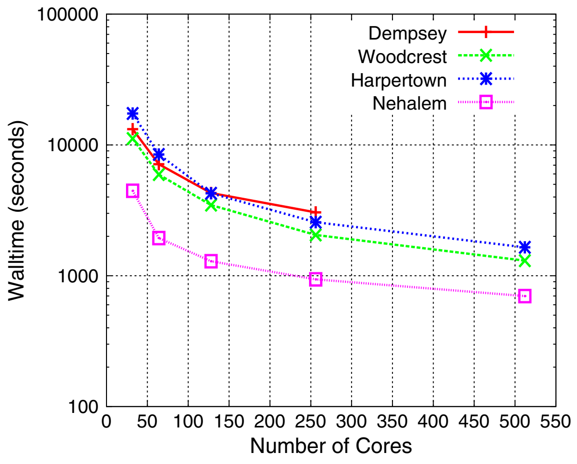

Figure 3. FLASH walltimes on *discover* for $nend = 100$.

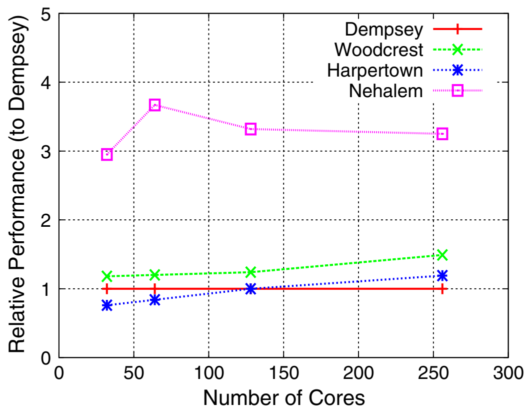

Figure 4. FLASH relative performance on *discover* for $nend = 100$.

scaled by multiplying by the Dempsey walltime for the corresponding number of Dempsey cores (larger is better).

Figures 3 and 4 show performance improvement due to processor improvements, including faster clock rate (if any), processor architecture enhancements, and improvements to the memory interface (as on Nehalem). Ideally, the improvement factors in a given column should be the same, but undoubtedly, features that affect the interconnect performance contribute to the variation. An example would be an increasing ratio of computation to communication in the test case as the number of cores increases coupled with better processor performance relative to Dempsey. Other possible explanations include varying I/O and communication traffics from other jobs while these data were gathered, and differences in process-on-node placement. However, positing either of these last two reasons would contradict data gathered in the initial phase of this study that indicated relatively low run-to-run variation for this test case (at least for Harpertown). Figure 5 compares processors by comparing their scalability. Specifically, the reciprocal of each walltime (the performance) from Figure 3 was scaled by multiplying by the walltimes for the 32-core run of the same processor type (again, larger is better). The Harpertown data show that it has better scalability than Nehalem and Woodcrest even though their absolute performance is superior, indicating that the Harpertown SCUs are architecturally more balanced systems. In a perfectly scalable system, the data in these columns would grow by factors of two. That they do not indicates the extent to which both hardware and software factors prevent perfect subdivision of computational work and, at the same time, introduce increasing communications overhead as more cores are applied to the problem.

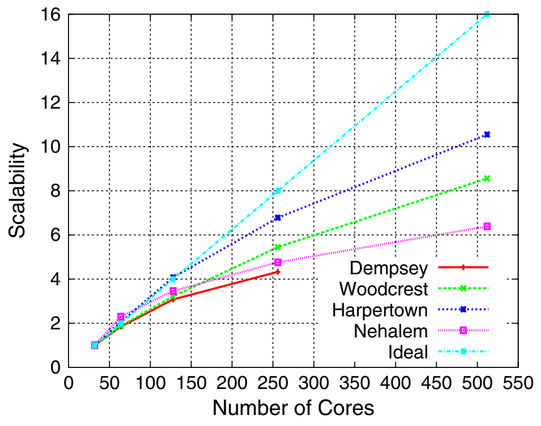

Figure 5. FLASH scalability on *discover* for $nend = 100$.

This decreasing efficiency as the number of cores increases is explicitly shown in Figure 6. Each datum in this graph is obtained from the corresponding datum in Figure 5 and is divided by the ideal speedup relative to the 32-core run on that SCU, thus giving the fraction of perfect scalability (relative to 32-core performance) available at this core count. Again, the Harpertown and Nehalem data illustrate that high parallel efficiency does not necessarily imply high absolute performance (and vice versa).

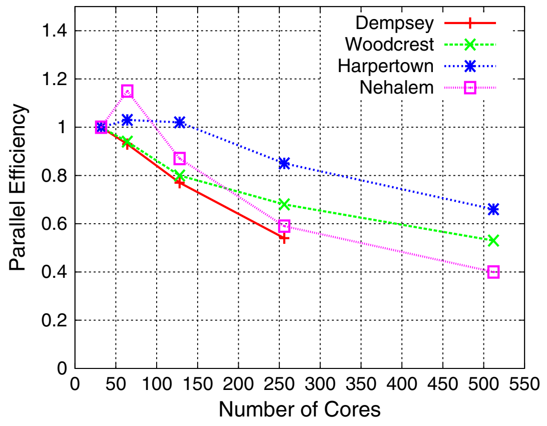

Figure 6. FLASH parallel efficiency for $nend = 100$.

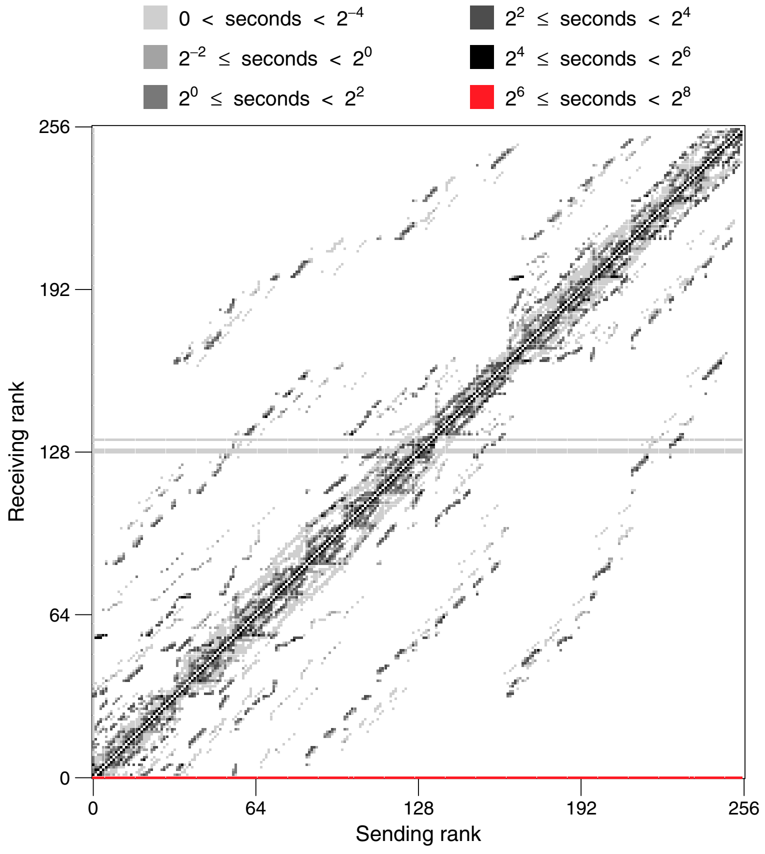

Figure 7. Time spent in message passing interface by message passing interface rank in 256-core FLASH job.

2.5. *Communication performance*

As mentioned earlier, IPM was the tool used to gather more detailed performance data regarding the communication behaviors of FLASH on discover. A 256-core FLASH job was instrumented to provide information about MPI communication, focusing on the number of MPI calls, data volume, and time spent in the MPI library. Two types of figures were used for each of these metrics. The first type, for example, Figure 7, uses an xy coordinate system with the x coordinate being the sending rank and the y coordinate being the receiving rank. The color of the square at each (x, y) point indicates the z-value and the bin size for each z-value/color increases as a power of 4. The result is a performance profile surface made up of colored squares. Figure 7 is a 256×256 plot displaying values for every sender–receiver pair in the job. The second type of figure, Figures 8 and 9, contain two stacked bar charts with one bar per sending rank and each bar made up of multiple colors to show the percentage of a particular category. Both bar charts display the same data, the difference being that the first such bar chart in the figure is ordered by sending rank and that the second is ordered by the metric in question (bar height).

For this study, communication data was generated for three metrics, number of MPI calls, data volume between MPI ranks, and time spent inside the MPI library; for brevity, we only present figures of the time data. By visually inspecting Figure 7, the pattern shown is generally symmetric but with four major exceptions. One horizontal bar at the bottom of the chart and two in the middle indicate that all of the ranks (0–255) are sending data to receiving ranks 0, 128, 129, and 133. The number of calls involved in the latter three is relatively small, but the numbers of calls to send data to rank 0 is quite large and roughly the same as the numbers of calls to send data to the nearest neighbors (indicated by black squares immediately above and below the diagonal). The concentration of communication along the diagonal results from logically contiguous MPI ranks simulating

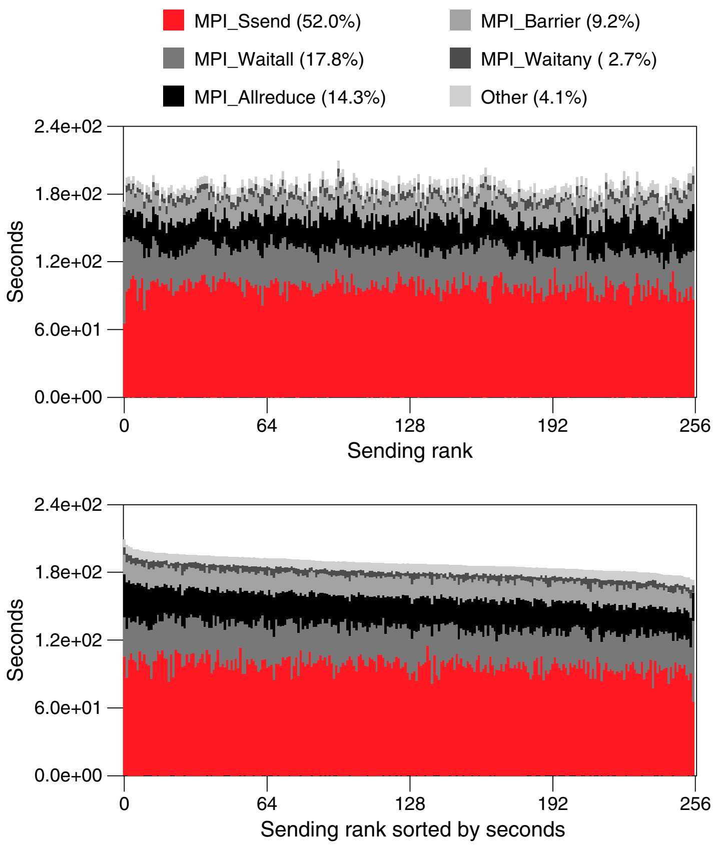

Figure 8. Time spent in message passing interface by message passing interface (MPI) call in 256-core FLASH job.

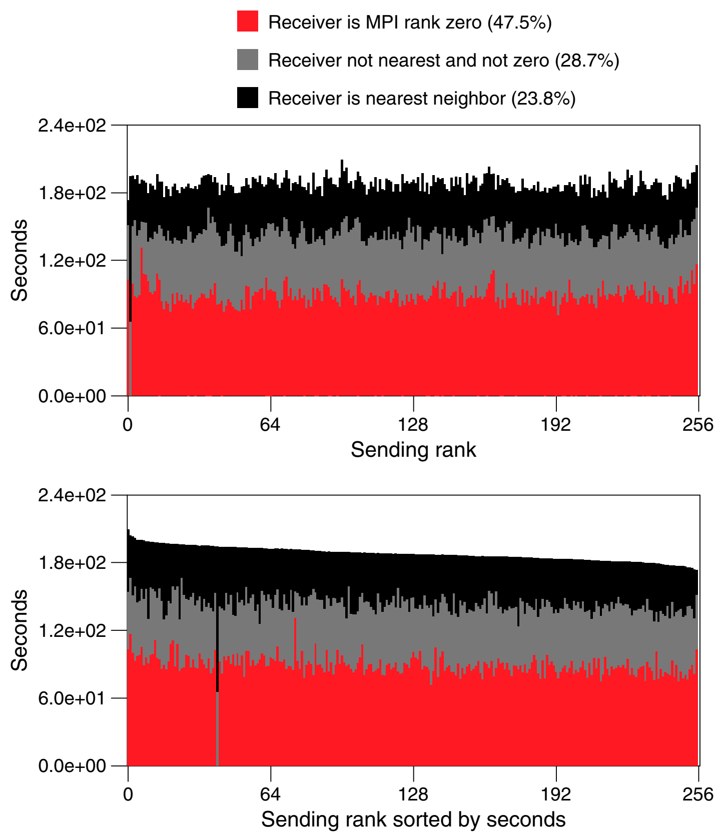

Figure 9. Time spent in message passing interface by neighbor in a 256-core FLASH job.

physically contiguous data subdomains and then communicating subdomain edge data to neighboring ranks. Rank 0 being responsible for writing output data received from other ranks causes the concentration of calls at the bottom row. Figure 8 provides the additional information that most of the MPI calls are either MPI_Ssend or MPI_Irecv, the latter being used to receive data sent by the former and accounting for the equal distribution of calls between them.

Data traffic resembles the number of MPI calls except that the increased concentration of high values indicates that most data traffic takes place either between the nearest neighbors (presumably during the compute phase) or is sent to rank 0 (presumably to accomplish output). We have confirmed earlier observations regarding destination ranks for data traffic by showing in Figure 9 that time spent in nearest neighbor communication is nearly 45% of the total data volume, data sent to rank 0 over 30%, and data sent to all other ranks only about 24%. Also, worthy of note is the rather wide variation in data volume between ranks sometimes as high as 100%.

Figures 7–9 show communication performances by the time spent inside the MPI library. The aspect that immediately stands out is that nearly half of the time spent in MPI is spent sending data to rank 0. The graphs showing time spent in Figures 8 and 9 are significantly flatter than the measurements for data volume and number of messages. This improved balance in time over data transfer volume must come at the price of the time waited by some ranks. Figure 8 shows that most MPI time is spent in MPI_Ssend; interestingly, however, nearly all of the remaining time is spent doing MPI collective operations and barriers.

3. CONCLUSIONS

Several important conclusions may be drawn from the following studies:

- There is still room for improving the balance of MPI communications for this test case. This would probably require tightening the threshold at which additional mesh refinement is performed as the run progresses. Certainly, if the threshold is set too low, too frequent mesh refinement may increase the walltime.

- The Intel Nehalem processor affords a significant step forward in performance relative to the previous Intel processors, specifically the Dempsey, Woodcrest, and most particularly Harpertown processors. The improvement seems primarily due to the on-chip memory controller (following the lead of the AMD Opteron's Hypertransport technology) and not to a faster clock rate. Data in this study show Nehalem to be three times faster than Dempsey for a range of core counts even though it has a clock rate of 12% slower.
- Additional data should be gathered on the relative floating-point performance of the FLASH test case. Future investigations will address application performance when utilizing hybrid programming schemes for increased load balancing of the mesh refinement.

REFERENCES

1. Ferrari D. Workload characterization and selection in computer performance measurement. *Computer* 1972; **5**(4):18–24.
2. Jain R. *The Art of Computer Systems Performance Analysis*. John Wiley and Sons: New York, 1991.
3. Federal Computer Performance Evaluation and Simulation Center. *Use and Specifications of Remote Terminal Emulation in Adp System Acquisitions*, FPR 1-4.11. General Services Administration Automated Data and Telecommunications Services: Washington, DC, 1979.
4. Curnow HJ, Wichman BA. A synthetic benchmark. *The Computer Journal* 1976; **19**(1):43–49.
5. McCalpin JD. Memory bandwidth and machine balance in high performance computers. *Technical Committee on Computer Architecture Newsletter*, 1995 :19–25.
6. Weicker RP. Dhrystone: a synthetic systems programming benchmark. *Communications of the ACM* 1984; **27**(10):1013–1030.
7. Dongarra JJ. Performance of various computers using standard linear equations software. *CS-89-85*, Computer Science Department, University of Tennessee, Knoxville, Tennessee, 2004.
8. Bailey DH, Barton JT. The NAS kernel benchmark program. *NASA Technical Memorandum 86711*, National Aeronautics and Space Administration Ames Research Center, Moffett Field, California, 1985.
9. Van der Wijngaart RF. NAS parallel benchmarks version 2.4. *NAS Technical Report NAS-02-007*, NASA Advanced Supercomputing Division, NASA Ames Research Center, Moffett Field, CA 94035-1000, 2002.
10. Wong P, Van der Wijngaart RF. NAS parallel benchmarks I/O version 2.4. *NAS Technical Report NAS-03-002*, NASA Advanced Supercomputing Division, NASA Ames Research Center, Moffett Field, CA 94035-1000, 2003.
11. Saini S, Talcott D, Jespersen D, Djomehri J, Jin H, Biswas R. Scientific application-based performance comparison of SGI altix 4700, IBM power5+, and SGI ICE 8200 supercomputers. *NAS Technical Report NAS-09-001*, NASA Advanced Supercomputing Division, NASA Ames Research Center, Moffett Field, CA 94035-1000, 2009.
12. National Bureau of Standards. Guidelines for benchmarking ADP systems in the competitive procurement environment. *FIPS PUB 42-1*, National Bureau of Standards, Washington, DC, 1977.
13. National Bureau of Standards. Guidelines for constructing benchmarks for ADP system acquisition. *FIPS PUB 75*, National Bureau of Standards, Washington, DC, 1980.
14. Dongarra J, Martin J, Worlton J. Computer benchmarking: paths and pitfalls. *IEEE Spectrum* 1987; **24**(7):38–43.
15. Ferrari D. *Computer Systems Performance Evaluation*. Prentice-Hall: Englewood Cliffs, NJ, 1978.
16. Lilja DJ. *Measuring Computer Performance: a Practioner's Guide*. Cambridge University Press: Cambridge, 2000.
17. Intel Corporation. Intel 64 and IA-32 architectures optimization reference manual. *Order Number 248966-018*, Intel Corporation, Santa Clara, California, 2009.
18. ASC Center for Astrophysical Thermonuclear Flashes. website January 2010. URL http://flash.uchicago.edu.
19. Rosner R, Calder AC, Dursi LJ, Fryxell B, Lamb DQ, Niemeyer JC, Olson K, Ricker P, Timmes FX, Truran JW, Tufo HM, Young Y-N, Zingale M, Lusk E, Stevens R. FLASH code: studying astrophysical thermonuclear flashes. *Computing in Science and Engineering* 2000; **2**:33–41.
20. Fryxell B, Olson K, Ricker P, Timmes FX, Zingale M, Lamb DQ, MacNeice P, Rosner R, Truran JW, Tufo H. FLASH: an adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. *The Astrophysical Journal Supplement Series* 2000; **131**:273–334.
21. MacNeice P, Olson KM, Mobarry C, deFainchtein R, Packer C. PARAMESH: a parallel adaptive mesh refinement community toolkit. *Computer Physics Communications* 2000; **126**:330–354.
22. Calder AC, Curts BC, Dursi LJ, Fryxell B, Henry G, MacNece P, Olson K, Ricker P, Rosner R, Timmes FX, Tufo HM, Truran JW, Zingale M. High-performance reactive fluid flow simulations using adaptive mesh refinement on thousands of processors. *ACM/IEEE Supercomputing 2000 Conference*, IEEE Computer Society: Los Alamitos, CA, 2000.
23. Dursi LJ, Zingale M. Efficiency gains from time refinement on AMR meshes and explicit timestepping. In *Adaptive mesh refinement—theory and applications*, Vol. 41, Plewa T, Linde T, Weirs VG (eds), Lecture Notes in Computational Science and Engineering. Springer: Berlin, 2005; 103–113.

24. Ross R, Nurmi D, Cheng A, Zingale M. A case study in application I/O on linux clusters. *Proceedings of Supercomputing 2001*, Association for Computing Machinery: New York, 2001.
25. ASC FLASH Center. *FLASH User'S Guide*. University of Chicago: Chicago, 2005.
26. National Energy Research Scientific Computing Center. Integrated Performance Monitoring (IPM). *http://www.nersc.gov/nusers/resources/software/tools/ipm.php*, Lawrence Berkeley National Laboratory, DOE Office of Science, Berkeley, CA, 2009.
27. Boss AP, Ipatov SI, Keiser SA, Myhill EA, Vanhala HAT. Simultaneous triggered collapse of the presolar dense cloud core and injection of short-lived radioisotopes by a supernova shock wave. *Astrophysical Journal (Letters)* 2008; **686**:L119–L122.
28. Boss AP, Ipatov SI, Keiser SA, Myhill EA, Vanhala HAT. Triggering collapse of the presolar dense cloud core and injecting short-lived radioisotopes with a supernova shock wave. *Astrophysical Journal (Letters)* 2009:to appear.
29. Allen R, Kennedy K. *Optimizing Compilers for Modern Architectures: a Dependence-based Approach*. Morgan Kaufmann: San Francisco, 2001.