

An Empirical Performance Evaluation of Scalable Scientific Applications

J.S. Vetter, A. Yoo

This article was submitted to
SC2002: High Performance Networking and Computing, Baltimore,
Maryland, November 16-22, 2002

July 24, 2002

U.S. Department of Energy

Lawrence
Livermore
National
Laboratory

DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

This is a preprint of a paper intended for publication in a journal or proceedings. Since changes may be made before publication, this preprint is made available with the understanding that it will not be cited or reproduced without the permission of the author.

This report has been reproduced directly from the best available copy.

Available electronically at <http://www.doe.gov/bridge>

Available for a processing fee to U.S. Department of Energy
and its contractors in paper from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831-0062
Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-mail: reports@adonis.osti.gov

Available for the sale to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Road
Springfield, VA 22161
Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-mail: orders@ntis.fedworld.gov
Online ordering: <http://www.ntis.gov/ordering.htm>

OR

Lawrence Livermore National Laboratory
Technical Information Department's Digital Library
<http://www.llnl.gov/tid/Library.html>

An Empirical Performance Evaluation of Scalable Scientific Applications

Jeffrey S. Vetter

Andy Yoo

Lawrence Livermore National Laboratory
Livermore, CA, USA 94551
{vetter3,yoo2}@llnl.gov

1 Introduction

Although programming models and languages appear to be converging, the computational workloads and communication patterns for scientific applications vary dramatically, depending, in part, on the nature of the problem the applications are solving. In this paper, we investigate the scalability, architectural requirements, and inherent behavioral characteristics of eight scalable scientific applications. We provide a comparative analysis of these applications and isolate their performance characteristics using empirical measurements. We refine our analysis into precise explanations of the factors that influence performance and scalability for each application; we distill these factors into common traits and overall recommendations.

Initially, we examine the overall scalability of each application. Then, based on these results, we iteratively investigate the primary factors that affect scalability and performance using a combination of measurement techniques, such as message tracing and monitoring hardware counters, until we can understand each application's primary performance properties and the root causes of those properties.

2 Applications

For our investigation, we targeted eight sophisticated scientific applications. Table 1 provides an overview of our applications. The *language* for the application represents the primary languages used in the source code, although most of these complex applications are mixed language. *Area* summarizes the specific domain the application targets. *Programming model* shows the significant programming model supported by the application. The respective references provide more detail on each application. In addition, the source code for each application is also available from the ASCI Purple Benchmark website (<http://www.llnl.gov/asci/platforms>) with the exception of Sweep3D, which is available from the ASCI Blue benchmark site (http://www.llnl.gov/asci_benchmarks).

2.1 Application Software Characteristics

As Table 1 illustrates, all of our scalable scientific applications use the Message Passing Interface (MPI) [12, 21] for coarse-grained distributed memory concurrency. Although MPI provides a common foundation for explicit communication, its wide range of functionality supports a diverse set of application communication characteristics due to variations in application domain, algorithm, software design, and problem size [24]. The MPI specification is implemented as a collection of library routines and a runtime system. In addition to MPI, several applications also exploit OpenMP [8] for shared memory

concurrency. OpenMP is implemented as a set of compiler directives or pragmas that instruct an OpenMP-compliant compiler to generate constructs for threads-based parallelism. (This paper does not address the OpenMP aspects of these applications.)

In practice, all of our scalable scientific applications are portable and they target only MPI and OpenMP; they do not require particular features from the underlying architecture, such as a specific processor or interconnect. This model is a good fit with current parallel computer architectures: clusters of shared memory compute nodes [9, 18].

Application developers focus on these two standards for three interrelated reasons: portability over architectures, compatibility with software tools, and performance. Although application performance is occasionally at odds with these first two reasons, many applications benefit from the community attention directed on making these standards efficient. Four of our applications use FORTRAN 77 or FORTRAN 90. Two applications use C. Two other applications use a combination of both languages. None of these applications use assembly instructions.

Application	Area	Language
SPPM	3-D gas dynamics problem on a uniform Cartesian mesh	F77
SMG2000	Parallel semicoarsening multigrid solver	C
SPHOT	2-D Photon transport code	F77
IRS	Implicit radiation solver	C
MDCASK	Molecular dynamics code to study radiation damage in metals	F77
UMT	3D, deterministic, multigroup, photon transport code for unstructured meshes	C/ F90
AZTEC	Parallel iterative library for solving linear systems	C/ F77
SWEEP3D	Solver for the 3-D, time-independent, particle transport equation	F77

Table 1: Summary of Applications

2.2 Descriptions

2.2.1 sPPM

sPPM [17] solves a 3-D gas dynamics problem on a uniform Cartesian mesh, using a simplified version of the Piecewise Parabolic Method. The algorithm makes use of a split scheme of X, Y, and Z Lagrangian and remap steps, which are computed as three separate sweeps through the mesh per timestep. Message passing provides updates to ghost cells from neighboring domains three times per timestep.

2.2.2 SMG2000

SMG2000 [5] is a parallel semicoarsening multigrid solver for the linear systems arising from finite difference, finite volume, or finite element discretizations of the diffusion equation $\nabla \cdot (D\nabla u) + \sigma u = f$ on logically rectangular grids. The code solves both 2-D and 3-D problems with discretization stencils of up to 9-point in 2-D and up to 27-point in 3-D. Applications where such a solver is needed include radiation diffusion and flow in porous media. Our examination includes both the setup of the linear system and the solve itself. Note that this setup phase can often be done just once, thus amortizing the cost of the setup phase over many timesteps. This trait is relatively common in implicit timestepping codes.

2.2.3 SPHOT

Sphot is a 2-D photon transport code. Photons are born in hot matter, and tracked through a spherical domain that is cylindrically symmetric on a logically rectilinear, 2-D mesh. Monte Carlo transport solves the Boltzmann transport equation by directly mimicking the behavior of photons as they are born in hot matter, move through and scatter in different materials, are absorbed or escape from the problem domain. Particles are born with an energy and direction that are determined by using random numbers to sample from appropriate distributions. This code tracks particles through a logically rectangular, 2-D mesh that is internally generated.

2.2.4 IRS

IRS [3] is an implicit radiation solver code and solves the radiation transport equation by the flux-limited diffusion approximation using an implicit matrix solution. IRS uses the preconditioned conjugate gradient method (PCCG) for inverting a matrix equation. In the algorithm, a planar radiation wave diffuses through a regular rectangular mesh from one end to another. The problems execute for longer than it takes to traverse the spatial problem. This forces the radiation iteration count to increase dramatically and is more stressful on certain aspects of parallel communications.

2.2.5 MDCASK

MDCASK [2] is a molecular dynamics code to study radiation damage in metals. The basic features of the code include an algorithm for integration of the equations of motion, an interatomic potential, and boundary conditions and constraints. The algorithm first defines the original position of the atoms in the lattice, calculates the energy and the forces on each atom using interatomic potential, and integrates the equations of motion to obtain the next values of positions and velocities. The equations of motion are integrated using a fourth-order predictor corrector algorithm [4]. The link cell method [1] is used to calculate the neighbors of each atom.

2.2.6 UMT

UMT is a 3D, deterministic, multigroup, photon transport code for unstructured meshes. The algorithm solves the first-order form of the steady-state Boltzmann transport equation. The equation's energy dependence is modeled using multiple photon energy groups. The angular dependence is modeled using a collocation of discrete directions. The spatial variable is modeled with an upstream corner balance finite volume differencing technique. The solution proceeds by tracking through the mesh in the direction of each ordinate. For each ordinate direction all energy groups are transported, accumulating the desired solution on each zone in the mesh. The code works on unstructured meshes, which it generates at run-time using a two-dimensional unstructured mesh and extruding it in the third dimension a user-specified amount.

2.2.7 Aztec

Aztec [22] is a parallel iterative library for solving linear systems. Simplicity is attained using the notion of a global distributed matrix. The global distributed matrix allows a user to specify of his application matrix exactly as he/she would in the serial setting. Issues such as local numbering, ghost variables, and messages are ignored by the user and are instead computed by an automated transformation function. Techniques such as standard distributed memory methods; locally numbered submatrices, ghost variables are used to make the code efficient. In addition, Aztec takes advantage of advanced partitioning techniques and utilizes efficient dense matrix algorithms when solving block sparse matrices.

2.2.8 Sweep3D

Sweep3D [14, 16] is a solver for the 3-D, time-independent, particle transport equation on an orthogonal mesh and it uses a multidimensional wavefront algorithm for "discrete ordinates" deterministic particle transport simulation. Sweep3D benefits from multiple wavefronts in multiple dimensions, which are partitioned and pipelined on a distributed memory system. The three dimensional space is decomposed onto a two-dimensional orthogonal mesh, where each processor is assigned one columnar domain. Sweep3D exchanges messages between processors as wavefronts propagate diagonally across this 3-D space in eight directions.

3 Methodology

Our methodology is to investigate iteratively over increasingly detailed empirical data for each application. At the highest level, we measure overall performance and scalability, and then, based on this analysis, we focus our efforts on those characteristics that limit scalability and performance. We do this by empirically measuring the application's communication and computation activity during execution.

For communication, we record MPI statistics and trace MPI operations, if necessary. For computation, we use subroutine profiling and hardware counters on the microprocessor to capture specific data about important blocks of computation. Most of these strategies must be used very carefully to allow us to collect relevant and accurate information yet limit both perturbation and data generation.

3.1 Measuring Computation Performance

Single node computation performance is relatively well-understood area provided that proper tools exist to capture interesting performance information. For instance, subroutine profilers must account for multiple threads of execution and attribute information appropriately. For this work, we use subroutine profiling and microprocessor performance counters to gather empirical data about application computation.

To identify subroutines that consume relatively large amounts of wall-clock time, we use traditional subroutine profiling [11], which has been extended to work properly with MPI applications. Tools, such as VGV [13], can also capture a trace of subroutines as the applications execute.

To capture information about processor instructions and memory activity, we rely on eight hardware counters in the IBM Power3 processor. These counters offer a wide assortment of metrics and we program them to count events of interest to our study [6]. Typically, we capture the number of cycles, number of completed instructions, number of floating-point operations, cache misses, and number of memory loads and stores. We are particularly interested in these metrics, because from this set of metrics, we can easily compute the instructions per cycle, computational intensity, cache hit ratios, and the number of floating point operations, which are typically less sensitive to compiler optimization than other instructions.

3.2 Measuring Communication Performance

All of our applications spend time communicating among their tasks using point-to-point and collective communication routines from MPI. The amount of time the applications spend in these routines is a good indicator of how much time the application spends communicating and not computing. A significant number of the MPI routines,

such as those used for managing derived types, are task local, and do not communicate or synchronize with other tasks.

MPI statistics provide a scalable, lightweight overview of the application's communication activity [23]. Our statistical tool, named MPIP (MPI Profiler), uses the MPI profiling layer to wrap significant MPI communication routines in timers. These timers record wall-clock time and record the elapsed time of every call to a hash table. The wrapper also records a call site stacktrace (of configurable depth) for each MPI call, and it uses this information to index the hash table. With this stacktrace, we can easily identify different phases of computation and different MPI call sites. Many task-local MPI routines are not profiled. MPIP also allows users to measure only phases of their application by enabling and disabling MPIP as their application executes. MPIP has the advantages that it has small perturbation and limited storage requirements, which, in turn, make it accurate and scalable.

3.2.1 MPI Tracing

Beyond MPI statistics, more complex performance phenomena, such as load imbalance and the interactions between communication and computation, often demand MPI tracing. MPI tracing provides a chronological event stream of the subroutine calls to the MPI library by each individual MPI task in the application. The events usually include the parameters passed to the subroutine, a timestamp, and the duration of the subroutine. During execution, the tracer records events to a local memory buffer. When this memory buffer is filled, the tracer writes this information to a file stored on the node's local disk. At the end of application execution, the tracer collects these events from each node and merges them into one trace file. We then analyzed the trace files offline. Most trace-based performance analysis systems, including PICL [10], Pablo [19], Tau [20], VGV [13], and Paraver [7] use this approach.

Two problems accompany the advantages of tracing's detailed information. First, tracing can easily create an intractable amount of information, even for relatively small experiments. Second, as the tracer captures and manages all this information during execution, the tracer can introduce perturbation onto the target application. With these issues in mind, we must use tracing carefully to focus on the phenomena under investigation. With our iterative approach, we first identify the performance bound areas in the code with a statistical overview, and then we restrict analysis to those areas to minimize perturbation due to instrumentation and data collection.

4 Evaluation

This evaluation focuses on the overall scalability and performance of each application. Then, based on these results, we iteratively investigate the factors limiting scalability and performance using a combination of our measurement techniques, making the appropriate tradeoffs of detail versus perturbation for each application.

4.1 Platform

We ran our tests on an IBM SP system, located at Lawrence Livermore National Laboratory. It is composed of 68 IBM RS/6000 NightHawk-2 16-way SMP nodes using 375 MHz IBM 64-bit POWER3-II CPUs [25] and has peak performance rating of 1.6 TeraOps, 1088 GB of global memory, and 20.6 TB of global disk. At the time of our tests, the batch partition had 63 nodes and the operating system was AIX 4.3.3. We compiled the various tests with the IBM XL and KAI Guide compilers using IBM's MPI

library in user-space mode. Our test jobs ran on dedicated nodes, although other jobs were concurrently using the network.

4.1.1 Computational Performance

The computational capability of the Power3 processor is peak execution rate of eight instructions per cycle and a sustained rate of four instructions per cycle. Its design allows for concurrent operation of load/store instructions, floating-point instructions, fixed-point instructions, and branch instructions. Each processor has three integer units, two floating-point units, and two load/store units. The processor can complete four floating-point operations per cycle by retiring a fused-multiply-add (2 floating-point operations) in each floating-point unit on every cycle. The 375 MHz IBM 64-bit POWER3-II, then, has a peak floating point rate of 1500MF. Each SMP nodes has a peak performance of 19,840 MF/s and a memory size of 16 GB. The L1 Data Cache is 64 KB; the L1 Instruction Cache is 32 KB, with a line size of 128 bytes. L2 cache is 8 MB per processor.

The BLAS routine *dgemm*, as provided by the highly-optimized IBM Engineering and Scientific Subroutine Library [15], can exploit this functionality to attain a sustained flop rate of 1260 MF/s (84% of the peak rate) on a matrix order 1000. This routine obtains a computational intensity of 3.9 and an IPC ratio of 2.7. The memory system performance of these systems obtain approximately 5,000 MB/s bandwidth when using data in cache as Figure 1 illustrates with the *cachebench* benchmark.

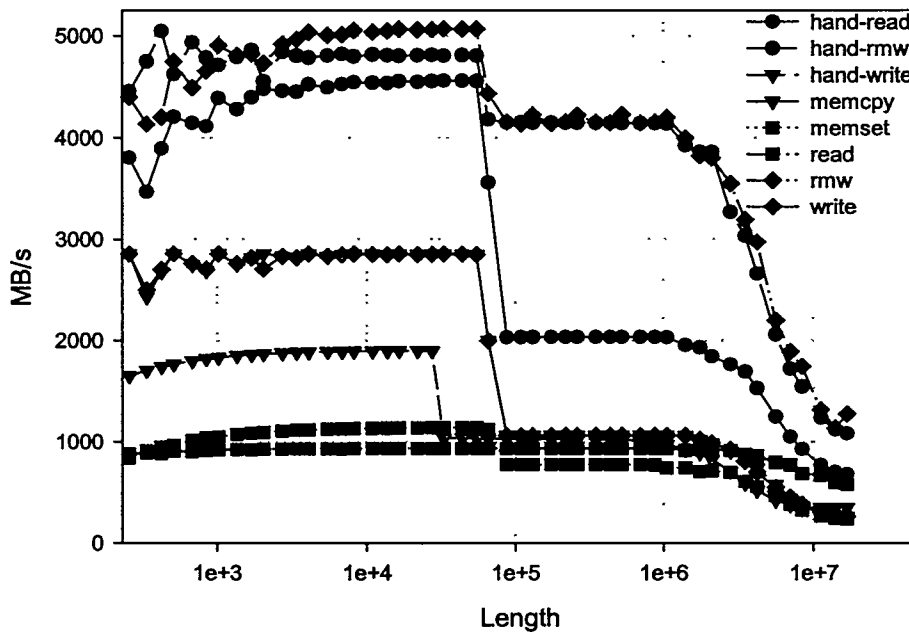


Figure 1: Memory System Performance on Cachebench.

4.1.2 Communication Performance

The system interconnect is IBM's SP Switch2 using the Colony double-single adapters, which provide a node-to-node bi-directional bandwidth of 2 GB/s. Figure 2 and Figure 3 illustrate both the internode and intranode unidirectional bandwidth and latency, respectively, as measured with the Pallas MPI PingPong Benchmark.

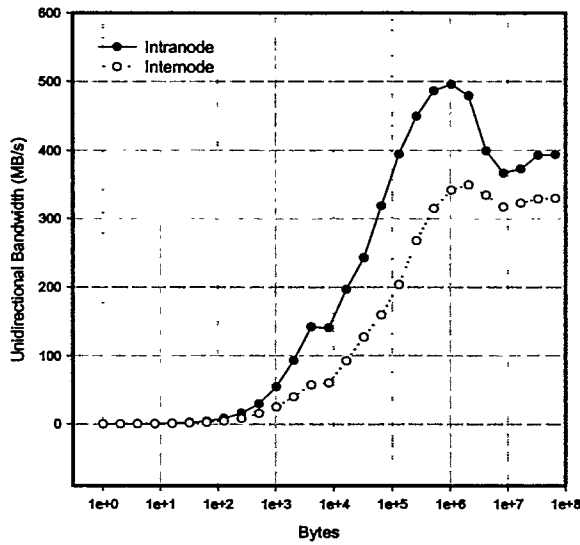


Figure 2: MPI Message Unidirectional Bandwidth.

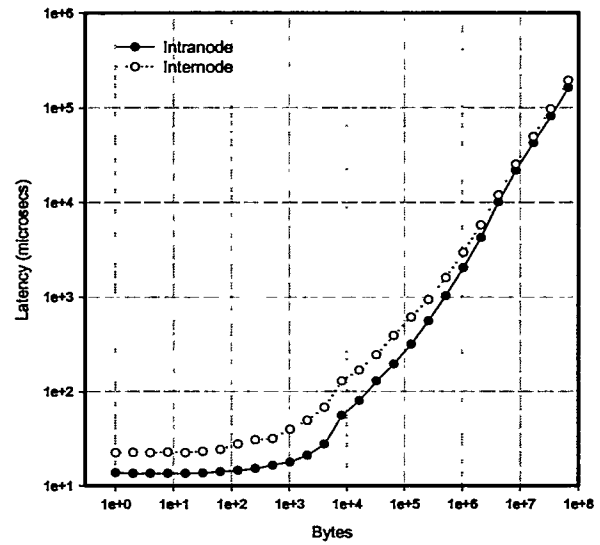


Figure 3: MPI Message Latency.

4.2 Overall Performance and Scaling

Figures 1-8 show the scaling behaviors for each application. Some application experiments are not easily scaled with the number of processors, so three application experiments use strong scaling (MDCASK, IRS, AZTEC) while the remainder use weak scaling.

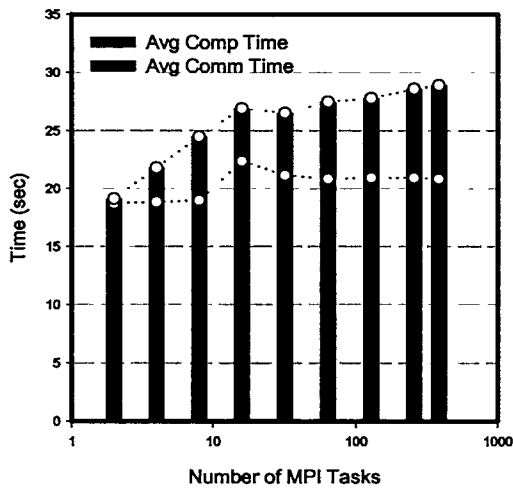


Figure 4: sPPM Scaling.

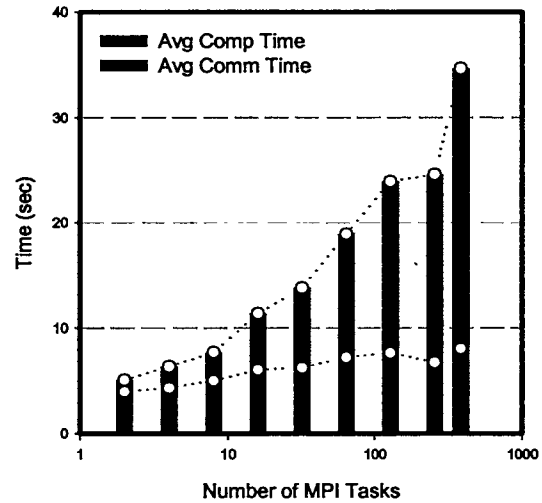


Figure 5: SMG2000 Scaling.

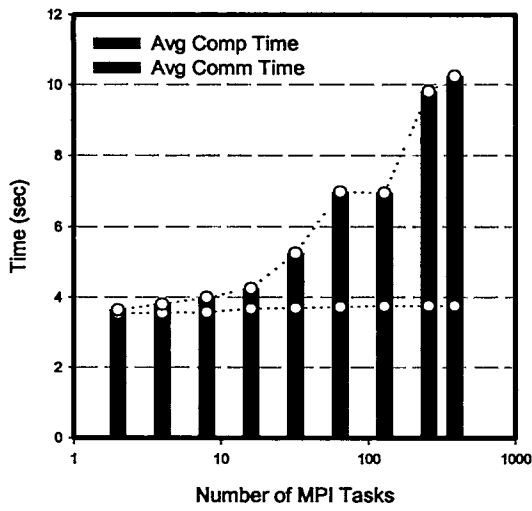


Figure 6: Sweep3D Scaling.

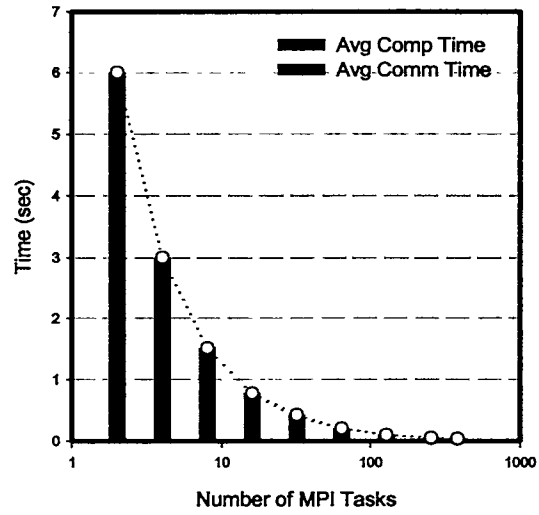


Figure 7: Sphot Scaling.

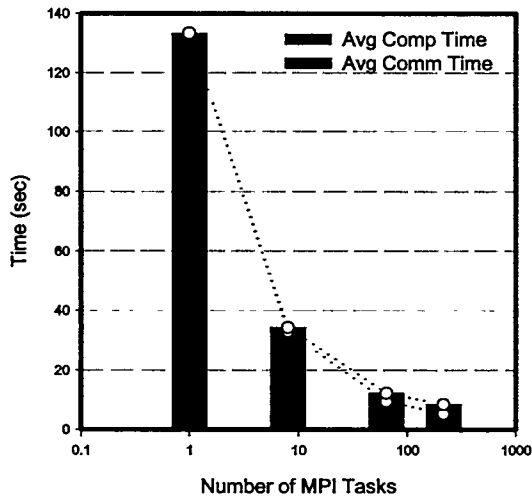


Figure 8: IRS Scaling.

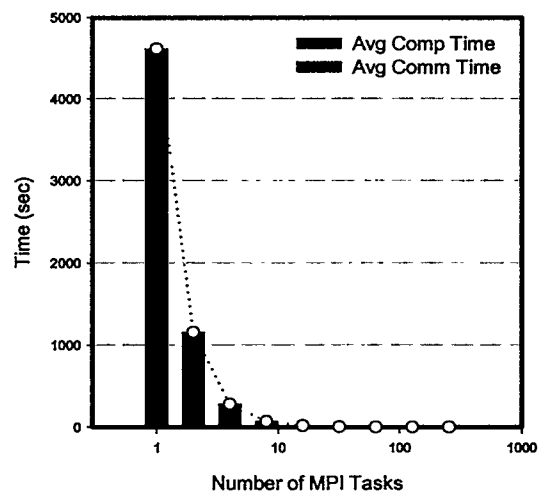


Figure 9: MDCASK Scaling.

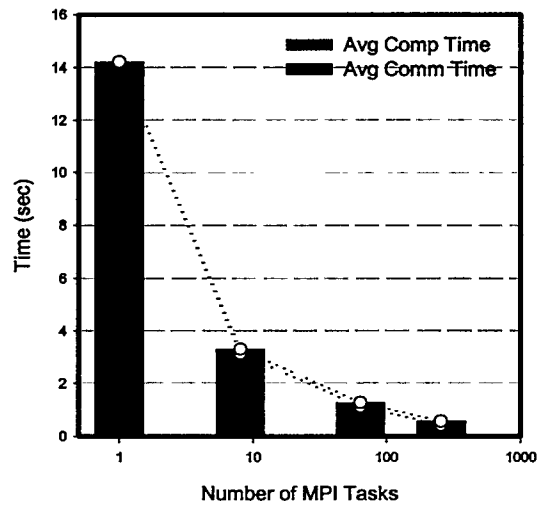
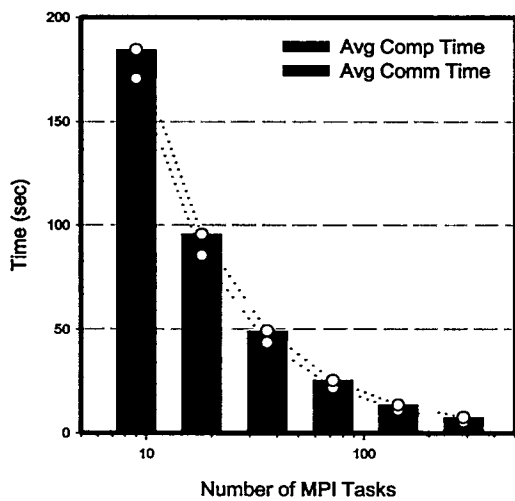


Figure 10: UMT Scaling.

Figure 11: AZTEC Scaling.

4.3 Applications

4.3.1 sPPM

As Figure 12 shows, each MPI task of SPPM has a regimented structure of computation followed by communication with six neighbors in the 3D domain; the trace shows all three waves of communication due to sweeps through the mesh for the double timestep. The scaling behavior of sPPM is quite good; it has demonstrated scalability to thousands of tasks [17]. However, as the application scales up, the MPI collective MPI_Allreduce begins consuming more time. The aggregate percentage of time consumed by calls to this routine grows from 0.1% to 4.2% when the number of tasks increased from 2 to 384. Also, all tasks of sPPM exchange their ghost cell update messages at approximately the same time, which, depending on the network topology, can lead to contention and bandwidth limitations in the network.

Each of three sweeps through each task's mesh calls the subroutine `sppm`, in turn, and `sppm` along with three major subroutines that it calls consume approximately 65% of computation time on each node. Using hardware counters, we found that the ratio of instructions per cycle (IPC) for this code region is approximately 0.87 across all tasks. The large majority of the array accesses are unit stride, so these accesses benefit from the Power 3's unit stride hardware prefetching as evidenced by high cache hit rate of 98% for L1 and 99.7% for L2. Also, this subroutine has a good computational intensity (ratio of floating point operations to number of memory accesses: sum of loads and stores) of 1.45.

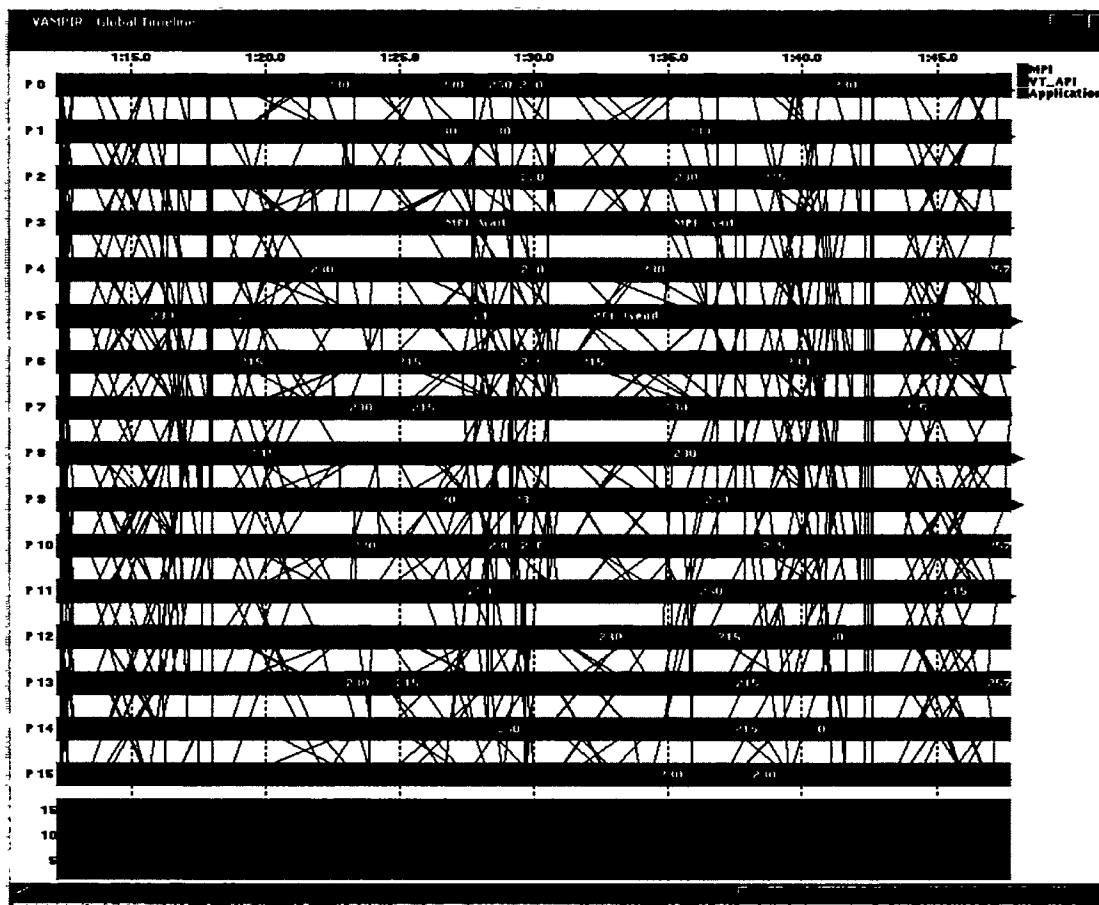


Figure 12: sPPM communication for one double timestep.

4.3.2 MDCASK

The domain decomposition used in this program is based on the link cell method [1]. The simulation box is divided into smaller cells, each of which has dimensions slightly bigger than the cut-off of the interatomic potential. Most of communication between nodes consists of updating the skin cells after each integration step and sending atoms across nodes. Most of the simulation time is spent in computing the forces and energies of each atom. MPI collective MPI_Bcast is called several times during this phase of computation. Other communication between nodes is performed by point-to-point MPI calls.

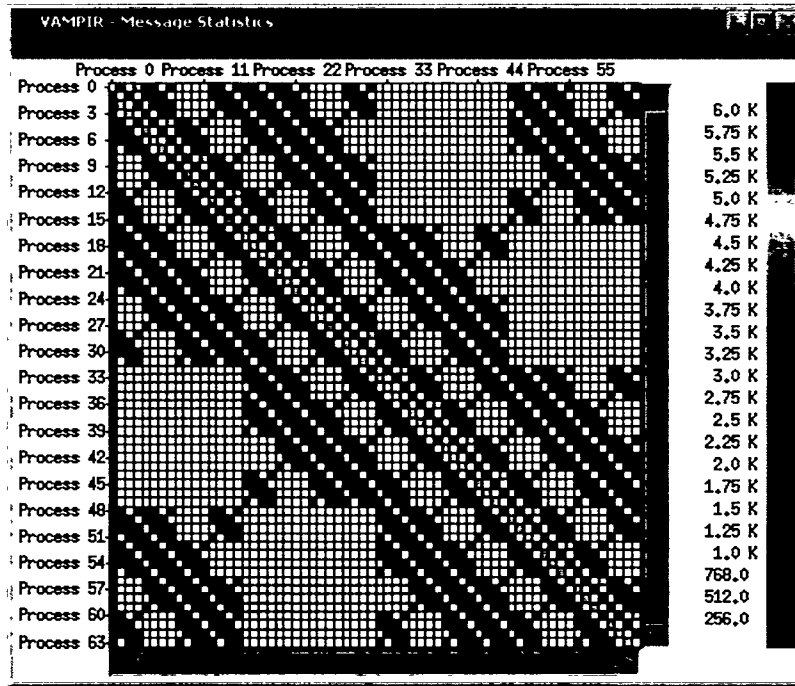


Figure 13: Message statistics for MDCASK (average message length).

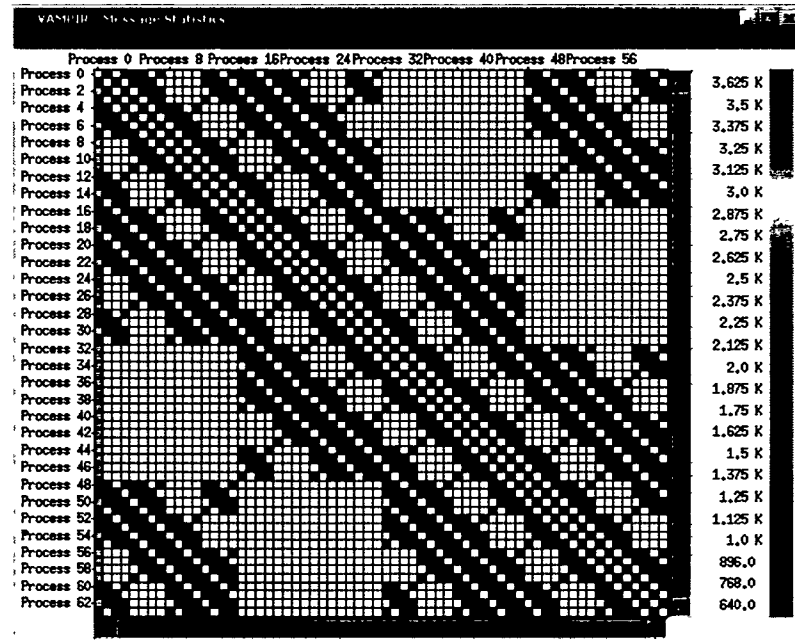


Figure 14: Message statistics for MDCASK (number of messages).

As Figure 6 shows, MDCASK scales well. When a small number of tasks (less than 16 tasks) are used, the total communication time is less than 4.15% of total execution time. However, as the number of tasks grows, this ratio increases. MDCASK spends 67.7% of its execution time for communication when 256 tasks are used. Further analysis on the communication behavior of MDCASK with tracing reveals that each task sends moderately sized messages (6 KB) to the next task in communication rank and a large

number of small messages (less than 256 bytes) to the root task (task 0) as shown in Figures 10 and 11. In addition, the code executes a fixed number of MPI_Bcast operations (3221 times) and the size of the messages exchanged by this collective is small (152 bytes on average). When a large number of tasks are used, the MPI_Bcast collective becomes a bottleneck as those tasks wait for the collective to finish broadcasting small messages.

4.3.3 SMG2000

SMG2000 has a very complex communication structure based, in part, on the recursive nature of the multigrid V-cycle. Most communication uses point-to-point messages with a limited number of calls to MPI_Allreduce. All messaging uses nonblocking MPI_Isends and MPI_Irecvs. For this short experiment, MPI_Irecv and MPI_Isend are invoked 40,830,584 times each: a tremendous number of messages relative to the other applications portrayed here. At 384 tasks, these two routines and the matching completion operations account for over 98% of the SMG2000's aggregate time in MPI, and almost 75% of the overall application aggregate time. MPI_Allreduce accounts for the remaining aggregate communication time of about 2%.

Although SMG2000 spends the majority of time in communication, compute performance is also important because the majority of memory access patterns for SMG2000 are not unit stride and they usually rely on indirection. On a 32 task experiment, our data shows that SMG2000 spends 34.3 seconds of a 54.3 second experiment (63%) in two routines: CyclicReduction and SMGResidual. The instructions per cycle for these routines are 0.811 and 0.923; however, the computational intensity is very low at 0.056 and 0.099, respectively. This low intensity indicates that the routines have a lot of memory traffic for each floating-point operation.

4.3.4 Sweep3D

As Sweep3D scales up in the number of tasks, so do the communication requirements of its wavefront algorithm as shown by Figure 6. In our experiments, at 384 tasks, Sweep3D spends approximately 63% of its aggregate time in MPI. Of this time in MPI, the MPI_Send and MPI_Recv routines that make up the core of the wavefront algorithm account for 50.5% and 42.5%, respectively. The computational component of Sweep3D is in a loop in the subroutine sweep.

4.3.5 UMT

As Figure 10 illustrates, the majority of UMT's time is spent in computation rather than communication. At 288 tasks, during one timestep, UMT spends about 21% of its aggregate time in MPI of which 17.3% is in the MPI_Barrier operation. The remaining 79% of time is spent in computation. On closer examination of the computation, we found that almost 71% of one timestep is attributable to one loop in procedure snswp3d. This loop updates the elements of the unstructured mesh using a series of mathematically intensive computations that compute the flux for the transport algorithm.

Further investigation revealed that the calculations in this loop have a large proportion of 64-bit floating-point instructions with a computational intensity of 1.5 and an IPC of about 0.74. In fact, our measurements indicate that for this loop, almost 6.1% of the instructions are divisions. In the Power3 architecture, most double-precision floating-point operations (including fmas) have a 3-cycle latency, 1-cycle throughput; however, several operations including division and square root have a latency of 18-25 cycles and they are not pipelined.

5 Observations and Recommendations

Table 2 summarizes our findings for factors limiting the performance and scalability of our workload. To be clear, we expect virtually all of our applications would benefit to some extent from improvements in any of these factors. However, our intent here is to identify those factors that appear to be the most critical to performance and scalability.

Six of eight applications use collective operations significantly, and our experiments show that as the applications scale up, these collective operations garner more execution time. Algorithms for operations such as broadcast and reduction are well known; however, these results may signal a need for more research into different algorithms and implementations that exploit hardware features of the interconnect, such as multicast.

APPLICATION	FACTORS					
	Collective Performance and scaling	Load imbalance	Point to Point message overhead and latency	Message Bandwidth	Memory Subsystem Performance	Instruction level parallelism
SPPM	X			X		X
SMG2000	X		X		X	
SPHOT	X					X
IRS	X				X	
MDCASK	X		X			
UMT			X	X	X	X
AZTEC		X				X
SWEEP3D	X		X			X

Table 2: Application Limiting Factors.

Five of eight applications could benefit from either exposing more instruction level parallelism to the compiler and hardware, or providing more instruction level parallelism in the hardware per se. Applications such as Sweep3D and sPPM have predictable memory access patterns at the loop level, so if the application properly exposes the parallelism to the compiler using software pipelining, for example, then the architecture can perform at near capacity.

Half of our applications use MPI's point-to-point communication operations extensively, so improvements in point-to-point message overhead and latency would enhance their performance. In fact, several of our applications send relatively small messages, usually less than 10K bytes, which boosts the requirements for low message overhead and appropriate message protocols.

6 Conclusions

We investigate the scalability, architectural requirements, and inherent behavioral characteristics of eight scalable scientific applications. We provide a comparative analysis of these applications by isolating their performance characteristics using a combination of empirical measurements. We gather performance data using a range of tools that include subroutine profiling, MPI tracing, and accessing hardware counters. We

iteratively refine our analysis into precise explanations of the factors that limit performance and scalability for each application. With this empirical data, we classify each application in terms of limiting factors and recommend architectural enhancements that would help to alleviate those major factors. We found that most applications would benefit from scalable collective operations, more instruction level parallelism, low overhead point-to-point communication, and improved techniques for managing latency in the memory hierarchy.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48. This paper is available as LLNL Technical Report UCRL-JC-148061.

References

- [1] M.P. Allen and D.J. Tildesley, *Computer simulation of liquids*. Oxford England, New York: Clarendon Press; Oxford University Press, 1989.
- [2] A. Almazouzi, M. Victoria, M.J. Caturla, and T.D.d.l. Rubia, "A Hierarchical Computer Simulation Model for the Evolution of the Microstructure Produced By Displacement Cascades In Metals," *EPFL Supercomputing Review*, 10(Nov), 1998.
- [3] W.K. Anderson, W.D. Gropp, D.K. Kaushik, D.E. Keyes, and B.F. Smith, "Achieving High Sustained Performance in an Unstructured Mesh CFD Application," *Proc. Supercomputing* (electronic publication), 1999.
- [4] H.J.C. Berendsen and W.F. Gunsteren, "Practical algorithms for dynamic simulations," *Proc. Molecular dynamics simulation of statistical mechanical systems*. Enrico Fermi Summer School, 1985, pp. 43-65.
- [5] P.N. Brown, R.D. Falgout, and J.E. Jones, "Semicoarsening multigrid on distributed memory machines," *SIAM Journal on Scientific Computing*, 21(5):1823-34, 2000.
- [6] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci, "A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters," *Proc. SC2000: High Performance Networking and Computing Conf.* (electronic publication), 2000.
- [7] J. Caubet, J. Gimenez, J. Labarta, L. DeRose, and J.S. Vetter, "A Dynamic Tracing Mechanism for Performance Analysis of OpenMP Applications," *Proc. Workshop on OpenMP Applications and Tools (WOMPAT)*, 2001.
- [8] R. Chandra, *Parallel programming in OpenMP*. San Francisco, CA: Morgan Kaufmann Publishers, 2001.
- [9] D.E. Culler, J.P. Singh, and A. Gupta, *Parallel computer architecture: a hardware software approach*. San Francisco: Morgan Kaufmann Publishers, 1999.
- [10] G.A. Geist, M.T. Heath, B.W. Peyton, and P.H. Worley, "A Users' Guide to PICL - A Portable Instrumented Communication Library," Oak Ridge National Laboratory, P.O.Box 2009, Bldg. 9207-A, Oak Ridge, TN 37831-8083 1991.
- [11] S.L. Graham, P.B. Kessler, and M.K. McKusick, "Gprof: A Call Graph Execution Profiler," *SIGPLAN Notices (SIGPLAN '82 Symp. Compiler Construction)*, 17(6):120-6, 1982.

- [12] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*, 2nd ed. Cambridge, MA: MIT Press, 1999.
- [13] J. Hoeflinger, B. Kuhn, P. Petersen, R. Hrabri, S. Shah, J.S. Vetter, M. Voss, and R. Woo, "An Integrated Performance Visualizer for OpenMP/MPI Programs," Proc. Workshop on OpenMP Applications and Tools (WOMPAT), 2001.
- [14] A. Hoisie, O. Lubeck, H. Wasserman, F. Petrini, and H. Alme, "A General Predictive Performance Model for Wavefront Algorithms on Clusters of SMPs," Proc. ICPP 2000, 2000.
- [15] IBM, *Engineering and Scientific Subroutine Library for AIX Guide and Reference*, 3.3 ed: IBM, 2001.
- [16] K.R. Koch, R.S. Baker, and R.E. Alcouffe, "Solution of the First-Order Form of the 3-D Discrete Ordinates Equation on a Massively Parallel Processor," *Trans. Amer. Nuc. Soc.*, 65(198), 1992.
- [17] A.A. Mirin, R.H. Cohen, B.C. Curtis, W.P. Dannevik, A.M. Dimits, M.A. Duchaineau, D.E. Eliason, D.R. Schikore, S.E. Anderson, D.H. Porter, P.R. Woodward, L.J. Shieh, and S.W. White, "Very High Resolution Simulation of Compressible Turbulence on the IBM-SP System," Proc. SC99: High Performance Networking and Computing Conf. (electronic publication), 1999.
- [18] G.F. Pfister, *In search of clusters : the coming battle in lowly parallel computing*. Upper Saddle River, N.J.: Prentice Hall, 1995.
- [19] D.A. Reed, R.A. Aydt, R.J. Noe, K.A. Shields, and B.W. Schwartz, "An Overview of the Pablo Performance Analysis Environment," Department of Computer Science, University of Illinois, 1304 West Springfield Avenue, Urbana, IL 61801 1992.
- [20] S. Shende, A.D. Malony, J. Cuny, P. Beckman, S. Karmesin, and K. Lindlan, "Portable profiling and tracing for parallel, scientific applications using C++," Proc. SIGMETRICS Symp. Parallel and Distributed Tools (SPDT), 1998, pp. 134-45.
- [21] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, Eds., *MPI--the complete reference*, 2nd ed. Cambridge, MA: MIT Press, 1998.
- [22] R.S. Tuminaro, S.A. Hutchinson, and J.N. Shadid, "The Aztec Iterative Package," Proc. International Linear Algebra Iterative Workshop, 1996.
- [23] J.S. Vetter and M.O. McCracken, "Statistical Scalability Analysis of Communication Operations in Distributed Applications," Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP), 2001.
- [24] J.S. Vetter and F. Mueller, "Communication Characteristics of Large-Scale Scientific Applications for Contemporary Cluster Architectures," Proc. International Parallel and Distributed Processing Symposium (IPDPS), 2002.
- [25] S. Vetter, S. Andersson, R. Bell, J. Hague, H. Holthoff, P. Mayes, J. Nakano, D. Shieh, and J. Tuccillo, *RS/6000 Scientific and Technical Computing: POWER3 Introduction and Tuning Guide*: IBM, 1998.