UCRL-TR-209658

LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

MPI Profiling

D.K. Han, T.R. Jones

August 20, 2004

This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

MPI Profiling

Daniel Han
University of Southern California
E-mail: `handanie@usc.edu`

Terry Jones
Lawrence Livermore National Laboratory
E-mail: `trj@llnl.gov`

August 20, 2004

Abstract

The Message Passing Interface (MPI) is the de facto message-passing standard for massively parallel programs. It is often the case that application performance is a crucial factor, especially for solving grand challenge problems. While there have been many studies on the scalability of applications, there have not been many focusing on the specific types of MPI calls being made and their impact on application performance. Using a profiling tool called mpiP, a large spectrum of parallel scientific applications were surveyed and their performance results analyzed.

1 Introduction

Inherent in any type of parallel program is the need to communicate between processors, whether it be decomposing the problem set, synchronization, or computation of the final results. One way to accomplish this is with the Message Passing Interface (MPI) [1]. MPI allows the user programmer to explicitly parallelize the code and, as a result, allows for a high level of flexibility. This flexibility, however, may hinder performance and scalability. Thus, it is imperative on the part of the MPI implementer to ensure that message passing does not negatively affect performance. Using mpiP [2], a variety of statistics were compiled that may help both user programmers and MPI vendors to identify areas that could lend themselves to optimization.

2 Methodology

2.1 mpiP

Applications were profiled using mpiP, a lightweight profiling library for MPI applications. All the information captured by mpiP is task-local. It only uses communication during report generation, typically at the end of the experiment to merge results from all of the tasks into one output file. All applications that were available to download were compiled and profiled locally; otherwise, the maintainers of the code were given instructions on how to profile their code. The output file provides a variety of information in the areas of timing and communication statistics for MPI function calls.

2.2 Platform

Most of the profiles were done on the IBM SP or Linux clusters at Lawrence Livermore National Laboratory (LLNL). The IBM machine, UV, consists of 128 8-CPU SMP nodes with 16 GB of memory per node and a 1.5-GHz Power4 p655 core. The operating system running on the IBM is AIX 5.2. The Linux cluster, MCR, consists of 1154 nodes with each node having a dual 2.4-GHz Intel Xeon processor and 4 GB of memory. The Linux cluster uses an operating system derived from Red Hat Linux called CHAOS (Clustered High Availability Operating System) [3].

3 Applications

The applications chosen for profiling, most of which were developed at LLNL, utilized MPI significantly.

- AMTRAN: Solves 2D/3D deterministic neutron transport problem using adaptive mesh refinement technology [4].
- Ardra: Offers robust scalable solution methods for neutron and radiation transport problems in complex 3D geometries. High resolution in space, energy, and direction are supported [5].
- Ares: An instability 3D simulation in massive star nova envelopes [6].
- GEODYN: Hydrodynamics simulation used in shock physics involving material strength and response [7].
- GP: Implements first-principles molecular dynamics within density functional plane-wave pseudopotential formalism. GP has been used to simulate the properties of molecular liquids, solids, and semiconductor surfaces [8].
- IRS: The implicit radiation solver code solves the radiation transport equation by the flux-limited diffusion approximation using an implicit matrix solution. In fact, IRS is a general diffusion equation solver, but its flux limiter imposes the speed of light as the maximum signal speed, thus making it a radiation solver [9].
- Linpack: The HPL version of Linpack solves a (random) dense linear system in double precision arithmetic on distributed-memory computers. It can thus be regarded as a portable as well as freely available implementation of the High Performance Computing Linpack Benchmark [10].
- MDCASK: A molecular dynamics code to study radiation damage in metals [11].

- Miranda: A hydrodynamics code ideal for simulating Rayleigh–Taylor and Richtmyer–Meshkov instability growth. The code uses 10th-order compact (spectral-like) or spectral schemes in all directions to compute global derivative and filtering operations [12].
- SMG2000: A parallel semicoarsening multigrid solver for the linear systems arising from finite difference, finite volume, or finite element discretizations of the diffusion equation [13].
- Spheral: Spheral++ is a numerical tool for simulating the evolution of a set of fluid or solid materials subject to hydrodynamic, gravitational, and radiative effects [14].
- sPPM: Solves a 3D gas dynamics problem on a uniform Cartesian mesh using a simplified version of the PPM (Piecewise Parabolic Method) code [15].
- SWEEP3D: Solves a 1-group time-independent discrete ordinates (Sn) 3D Cartesian (XYZ) geometry neutron transport problem [16].
- UMT2K: A 3D deterministic, multigroup photon transport code for unstructured meshes [17].

4 Results

4.1 Time in MPI

Figure 1 shows the percentage of time each application spends in MPI versus time spent in the application (wall-clock time for all MPI calls versus wall-clock time between MPI_Init and MPI_Finalize). Data were averaged across several runs, anywhere from 32 processes to 640 processes, depending on the application. MPI makes up more than half of the time in most applications. The user programmer would like to minimize time in MPI, because time spent there is overhead and may greatly affect the ability of an application to scale.

Figures 2 and 3 give an overview of the MPI functions that dominate the applications with respect to time. The data were again averaged across several runs, both large and small. To simplify analysis, variations of similar functions were grouped together. For example, calls such as MPI_Gather, MPI_Gatherv, and MPI_Allgather were grouped under the heading of Gather in Figure 2. Although the differences between collectives such as Gather and AllGather[1] may be considerable with respect to performance, the focus here is more on the overall behavior of the application than on a strict quantitative analysis.[2] Collectives tend to be of interest because they are usually the dominating MPI calls in terms of communication and time. We account here for only those calls that took up a significant percentage of MPI time. Thus, there were other calls (less than 5% of MPI) that are not accounted for in these figures.

From Figure 2, we can see that most applications usually have one category of collectives that account for most of the time. Reduces take up a majority of several

[1] All processes receive data in the buffer rather than just one root process.

[2] For a performance analysis of many of the applications found here, see Vetter [18].

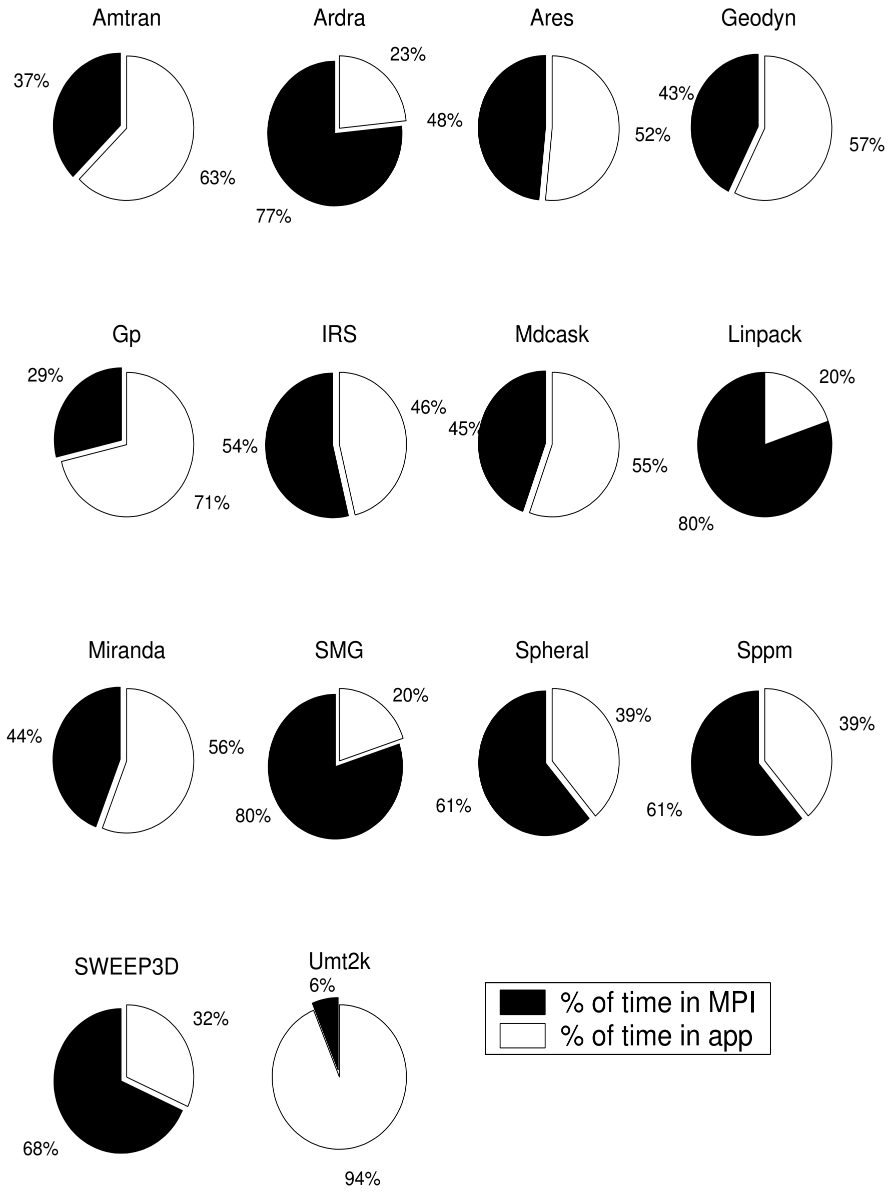

Figure 1: Time in MPI vs time in application.

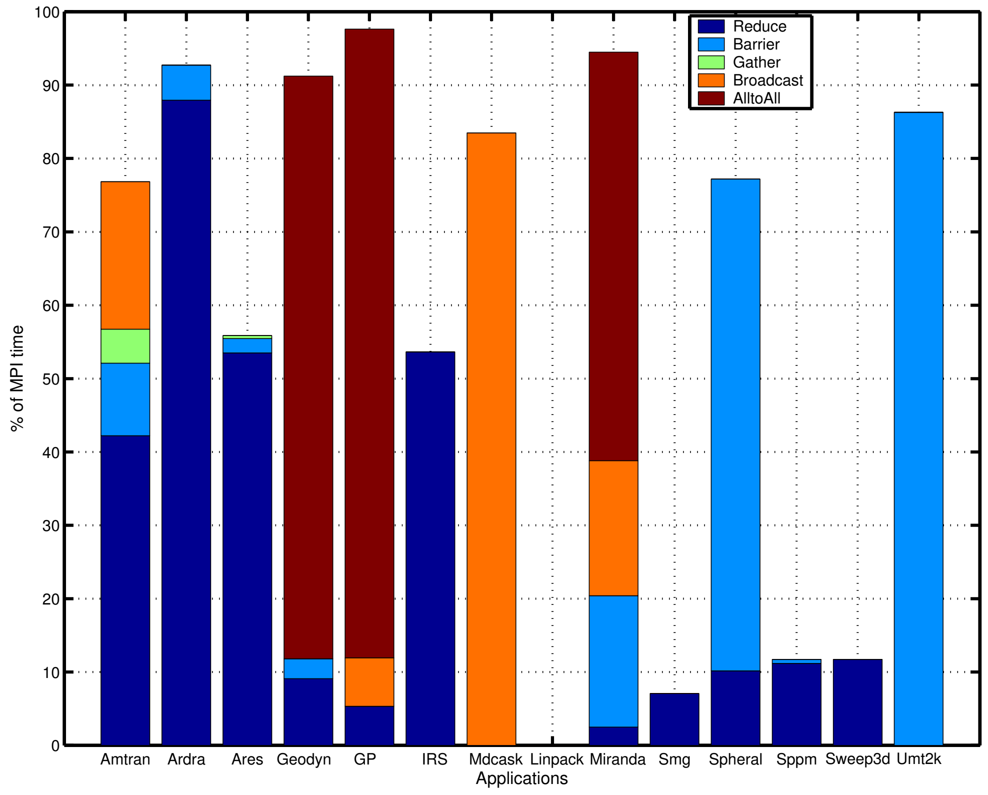

Figure 2: Top collective MPI calls.

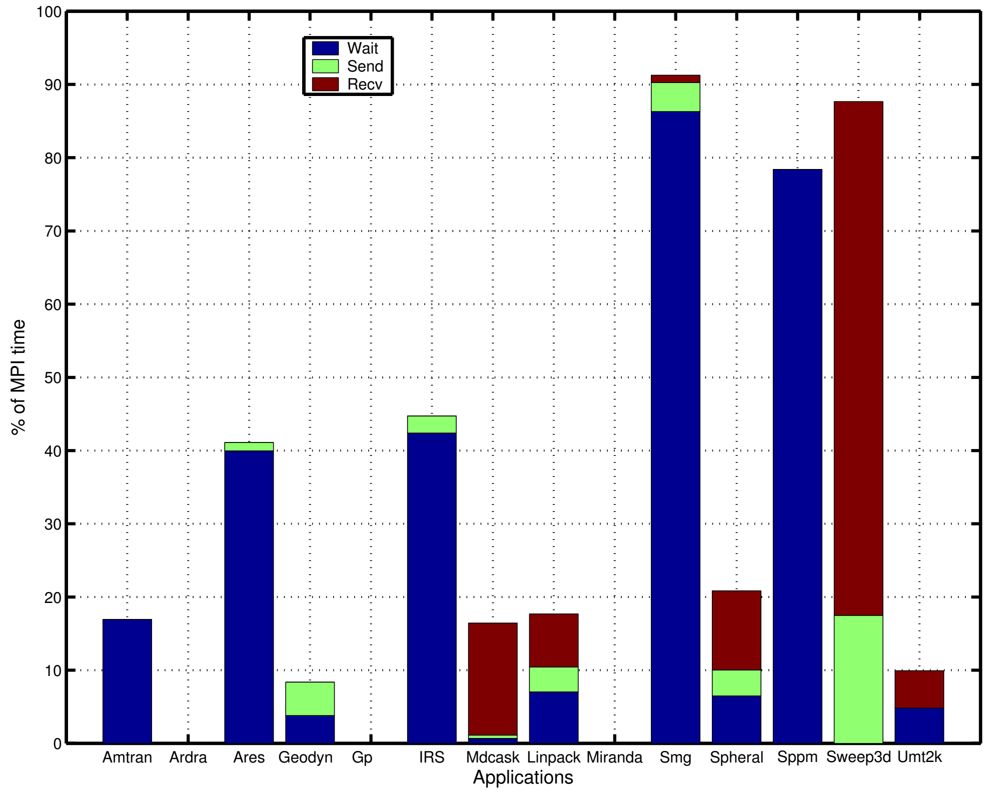

Figure 3: Top point-to-point MPI calls.

applications, while Gathers have a minimal effect. Figure 3 shows the same analysis with the point-to-point calls. Among the point-to-point calls, the MPI_Wait function is dominant. This is expected because many applications use nonblocking sends/receives to hide latency. Comparing the two figures, we see that collectives rather than point-to-point calls take up a larger percentage of the MPI time for an application. One notable exception is Linpack, in which neither collectives nor point-to-point calls have a great impact.[3]

4.2 Communication Patterns

While timing statistics are usually quite important, communication patterns are also of interest, especially to MPI implementors and vendors. Identifying the predominant message sizes could help in optimizing low-level protocols. Table 1 shows the message sizes that took up the majority of all messages sent. The smaller runs were usually fewer than 64 processors, while the larger ones were at least 256 processors. It is clear that within a given application, one type of message size generally predominates.

[3]Interestingly, Linpack spends most of the time in a communicator function, MPI_Comm_split, which partitions a given communicator into a new set of communicators.

Table 1: Predominant message sizes.

Application	Smaller Runs		Larger Runs	
	Kbytes	% of MPI	Kbytes	% of MPI
AMTRAN	23.63	94.79	784.18	99.24
Ardra	996.09	95.17	146.48	81.35
Ares	9.16	99.17	17.87	97.33
GEODYN	550.70	99.06	639.64	97.22
IRS	2.92	99.76	2.23	98.49
Linpack	1.50	91.05	Less than 0.5	91.45
MDCASK	4.62	99.76	2.68	99.62
Miranda	Less than 2	90.30	Less than 0.5	95.84
SMG2000	Less than 0.1	99.65	1	99.85
Spheral	Less than 3.6	100.00	0.1	100.0
sPPM	719.0	43.90	719.0	40.02
SWEEP3D	45.0	100.00	Less than 0.1	100.0

sPPM is the one exception; fewer than half the messages were 719 Kbytes.[4] Based on this data, it is difficult to pinpoint one common message size across all applications for MPI implementors to use. The general trend, however, seems to be that an application can be categorized as a small message passing (less than 5 Kbytes) application versus a large message passing (larger than 500 Kbytes) application.

4.3 Calling Patterns

Table 2 shows the average number of times a certain MPI function was called in larger runs. The functions chosen here were ones that were the most prevalent across all applications. Note that these values are normalized to take into account the number of processes that were executed. Essentially, it is the average number of calls made per process. It was surprising to see such a high number of calls being made to Waitany and the immediate sends/receives.

Figure 4 displays the average number of callsites broken down by function. This is the number of times a certain MPI function is found in the source code, which gives an idea of how MPI programmers are coding their parallel applications. The number of broadcast callsites seems high with over 10 callsites, while Allgather has an average of about 1 callsite.

4.4 Other Statistics

Figure 5 shows scalability limitations in terms of number of processors. The values for this figure were actual runs. All applications are capable of running in a single processor mode in order to debug the application and verify that the serial execution provided the correct results. Normal runs were considered to be runs with a moderately sized problem set or daily runs. The aggressive runs were those that pushed the limits

[4]The remaining messages were greater than 1,000 Kbytes for both large and small runs.

Table 2: Frequency of MPI calls.

MPI Function	Average No. Calls per Process
Allgather	68.5
Allreduce	10616.4
Alltoall	1057.0
Barrier	56.9
Bcast	2067.3
Gather	134.0
Gatherv	284.0
Irecv	246530.9
Isend	222527.9
Recv	53648.3
Reduce	250.9
Send	80337.4
Wait	65881.4
Waitall	31983.5
Waitany	562436.5

for that application. Many of these applications may scale to higher numbers because these values represent only documented runs. Table 3 lists the number of lines of code for each application. These values were obtained using the UNIX utility, wc, on the source code.

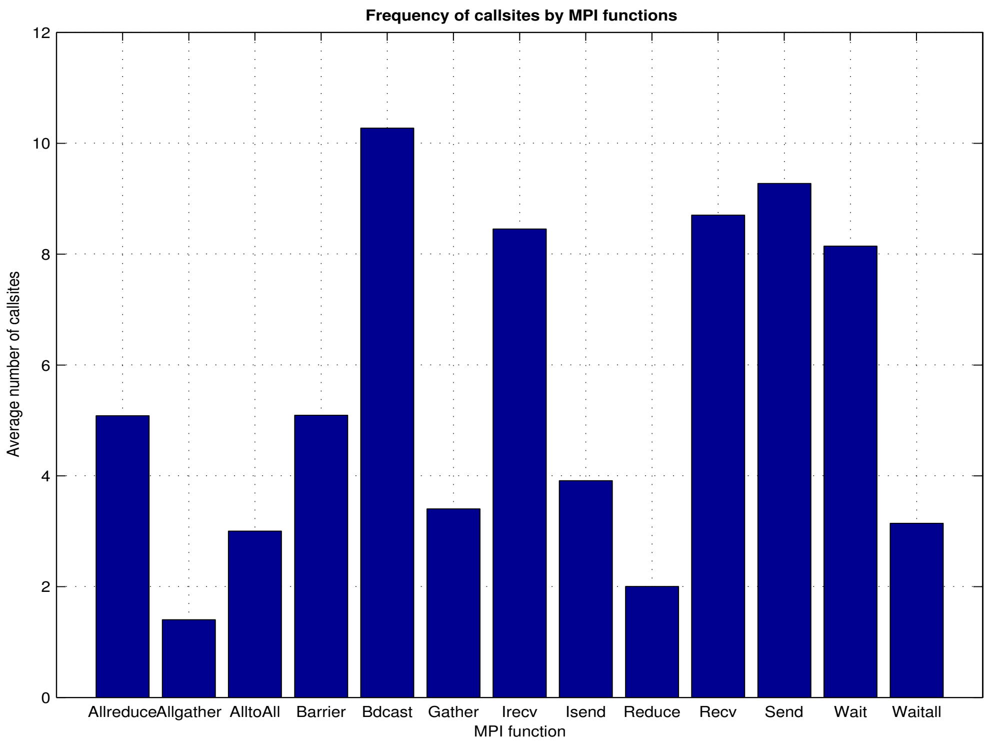

Figure 4: Average number of callsites.

Table 3: Lines of source code.

Application	Lines of Code
AMTRAN	89000
Ardra	75000
Ares	509800
GEODYN	100000
GP	NA
IRS	67400
MDCASK	21000
Linpack	26800
Miranda	NA
SMG2000	22700
Spheral	54200
sPPM	29700
SWEEP3D	3900
UMT2K	17000

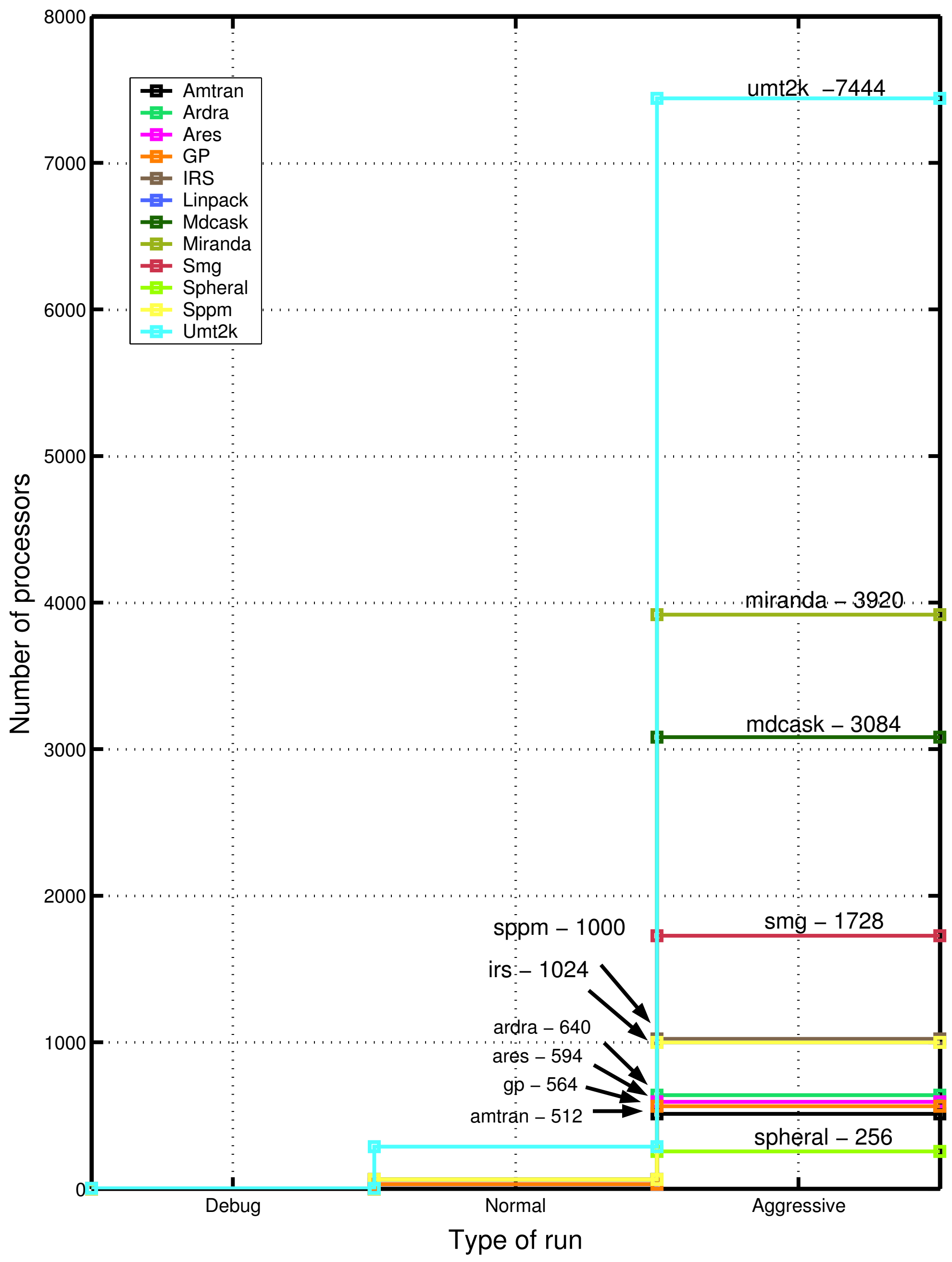

Figure 5: Scalability of applications.

5 Conclusion

Our profiling of various scientific parallel applications has given us insight into how they use MPI. Many of the applications spend significant time in MPI and are thus not fully optimized. Application programmers may be able to optimize these applications by reevaluating the problem specification in terms of data decomposition to see whether any parallelism can be exploited. Within MPI, MPI_Reduce and MPI_Barrier are most often called. MPI implementors and vendors can focus on these calls in future implementations to reduce time taken there, and application programmers may want to consider how they might change their implementation. An indepth profile of a specific application would clearly be beneficial in deciding which specific calls are causing bottlenecks.

Our profiles did not indicate the existence of an optimal message size. Smaller messages dominate, which is reasonable because of performance hits when scaling up to many processors. Examining MPI message sizes would be an interesting area for future research.

As these applications begin to run on hundreds and thousands of processors, performance will become increasingly important. By focusing on certain aspects such as time in calls, the specific MPI functions, and communication patterns, future implementations of MPI may ameliorate current performance issues.

References

[1] Message Passing Interface Forum. URL: http://www.mpi-forum.org/

[2] Vetter, J.S., and C.M. Chambreau. mpiP: Lightweight Scalable MPI Profiling. URL: http://www.llnl.gov/CASC/mpip/

[3] Garlick, J.E., and C.M. Dunlap. Clustered High Availability Operating System. URL: http://www.llnl.gov/linux/chaos/chaos.html

[4] Compton, J.C., and C.J. Clouse. "Domain Decomposition and Load Balancing in the AMTRAN Neutron Transport Code," in *Proc. 15th Intl. Domain Decomposition Conference*, Berlin, Germany (July 21–25, 2003). Also available as UCRL-JC-152231.

[5] Brown, P.N. Ardra: Scalable Parallel Code System to Perform Neutron and Radiation Transport Calculations. URL: http://www.llnl.gov/casc/ardra/

[6] Ares. Code developed by B.S. Pudliner, Lawrence Livermore National Laboratory.

[7] GEODYN. Code developed by I.N. Lomov and B.T. Liu, Lawrence Livermore National Laboratory.

[8] GP (formerly JEEP). Code developed by François Gygi, Lawrence Livermore National Laboratory. See "Quantum Simulations Tell the Atomic-Level Story," Science & Technology Review (April 2002).
URL: http://www.llnl.gov/str/April02/pdfs/04_02.01.pdf

[9] Dawson, S.A. IRS: An Implicit Radiation Solver.
URL: http://www.llnl.gov/asci/purple/benchmarks/limited/irs/

[10] Petitet, A., R.C. Whaley, J. Dongarra, and A. Cleary. HPL - A Portable Implementation of the High Performance Linpack Benchmark for Distributed-Memory Computers. URL: http://www.netlib.org/benchmark/hpl/

[11] Caturla, M.J. The MDCASK Benchmark Code.
URL: http://www.llnl.gov/asci/purple/benchmarks/limited/mdcask/

[12] Miranda. Code developed by A.W. Cook and W.H. Cabot, Lawrence Livermore National Laboratory. See Cabot, W.H., and A.W. Cook. "Large-Scale Simulations with Miranda on BlueGene/L," UCRL-PRES-200327.
URL: http://www.llnl.gov/asci/platforms/bluegene/papers/21cabot.pdf

[13] Brown, P.N, R.D. Falgout, and J.E. Jones. "Semicoarsening Multigrid on Distributed Memory Machines," *SIAM Journal on Scientific Computing* 21(5): 1824-34. Also available as UCRL-JC-130720.
URL: http://www.llnl.gov/asci/purple/benchmarks/limited/smg/

[14] Owen, M., P. Miller, M. Casado, J. Johnson, and N. Keen. Spheral++.
URL: http://spheral.sourceforge.net/

[15] Woodward, P.R., and Anderson, S.E. The sPPM Benchmark Code.
URL: http://www.lcse.umn.edu/research/sppm/README.html

[16] Owens, J.L. The ASCI SWEEP3D Benchmark Code.
URL: http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/

[17] Chan, B. The UMT Benchmark Code.
URL: http://www.llnl.gov/asci/purple/benchmarks/limited/umt/

[18] Vetter, J.S., and A. Yoo. "An Empirical Performance Evaluation of Scalable Scientific Applications," *SC2002: High Performance Networking and Computing*, Baltimore, MD (November 16–22, 2002). Also available as UCRL-JC-148061.
URL: http://sc-2002.org/paperpdfs/pap.pap222.pdf