# Parallel Performance Improvement

The Parallel Performance Improvement (PPI) effort is a collection of projects that are developing ways to improve the efficiency of parallel programs. We are investigating both performance analysis tools and system-level techniques for improving performance.

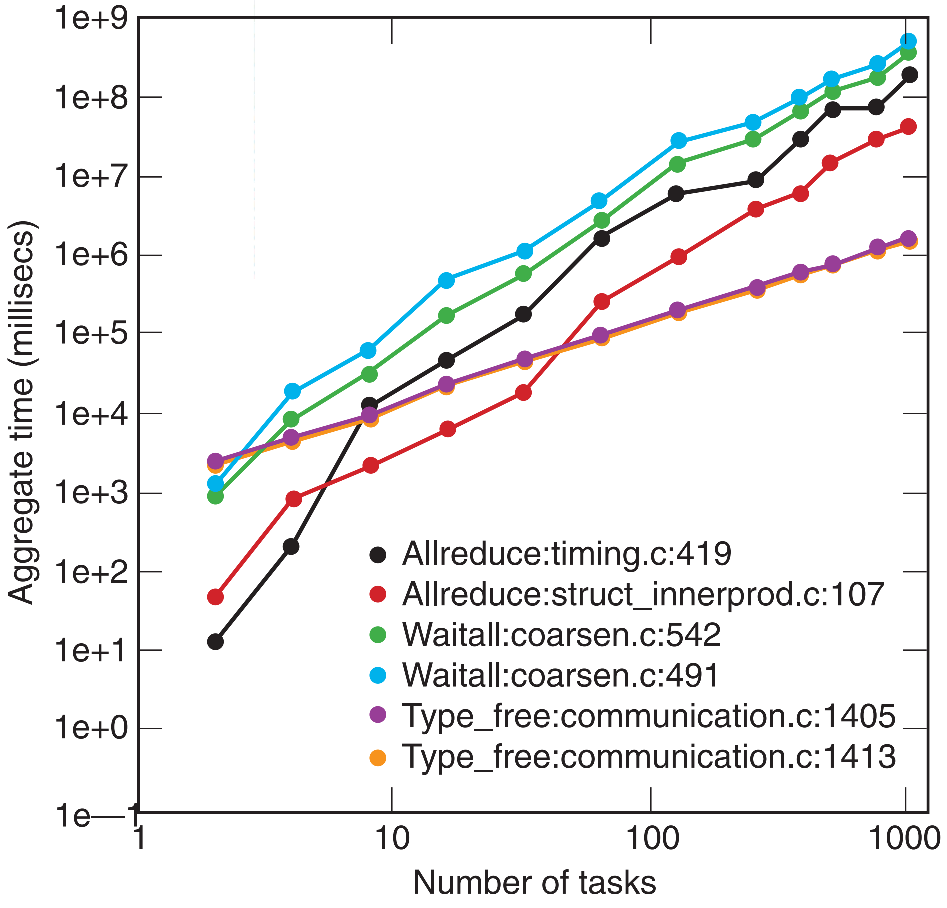


*Figure 1. The mpiP tool shows how much time various MPI calls take as the number of tasks increases. Steep increases indicate possible scaling problems.*

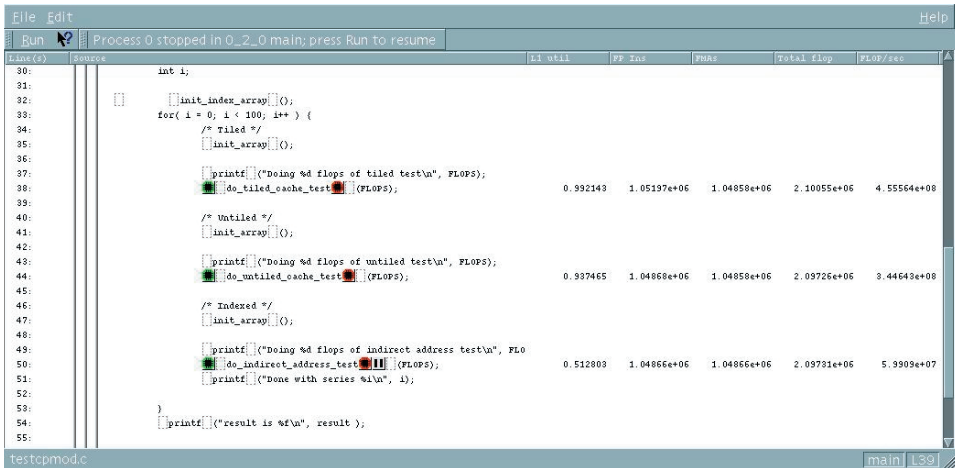
## Performance Analysis

Optimizing the performance of a parallel program requires attention to CPU utilization, memory subsystem utilization, communication efficiency, and other areas. Lawrence Livermore National Laboratory (LLNL) has developed tools to help developers investigate a variety of performance parameters.

The mpiP tool (Vetter and McCracken, 2001), for example, shows users how communication performance scales with increasing numbers of processors, and it offers information at varying levels of detail: categories of MPI functions (e.g., collective operations); specific MPI functions (e.g., all uses of MPI_Allreduce); and specific call sites (e.g., a call to MPI_Allreduce at a particular line of a particular file). These levels of detail enable users to see a general overview of communication performance and then zoom in to find specific problem areas. The mpiP tool gathers data from a whole program run without significantly slowing its execution, even when the program executes on 1000 processors or more (Figure 1).

Another tool, MPX (May, 2001), gathers statistics from CPU hardware performance counters (http://www.llnl.gov/CASC/mpx/). Performance data (such as requests to load data, cache misses, and floating point operations) can help programmers tune their codes to improve cache utilization or balance workload among processors. Unfortunately, hardware counters have somewhat limited functionality, since each counter is often designed to measure only a subset of the countable events on a CPU. Often, a combination of measurements is required to gain performance insights; for example, estimating cache utilization requires both the number of cache misses and the number of load requests. When hardware limitations prevent concurrent counting of hardware events, MPX uses time slicing, which directs the hardware registers to measure each event type in turn for a period of time. Using these measurements, MPX computes an accurate estimate of the frequency of each event over a specified measurement period. MPX technology has been integrated into the Performance API toolkit, developed at the University of Tennessee (PAPI, http://icl.cs.utk.edu/projects/papi/).

Tool Gear is software infrastructure that provides common services to tools like MPX and mpiP (http://www.llnl.gov/asci/projects/asde/toolgear.html). These services include source code navigation, dynamic instrumentation, data collection, and data display. Analysis tools focus on gathering data, and Tool Gear focuses on managing and presenting the data and also on user interactions (Figure 2). Tool Gear uses IBM's Dynamic Probe Class Library (DPCL), which lets tools insert instrumentation into a running program without the need for re-compilation or re-linking. Together, Tool Gear and DPCL offer a flexible combination



```
Process 0 stopped in 0_2_0 main; press Run to resume

Source                                                               L1 util    FP Ins       FMAs         Total flop   FLOP/sec
        int i;

        init_index_array();
        for( i = 0; i < 100; i++ ) {
                /* Tiled */
                init_array();

                printf("Doing %d flops of tiled test\n", FLOPS);
                do_tiled_cache_test(FLOPS);                          0.992143   1.05197e+06  1.04858e+06  2.10055e+06  4.55564e+08

                /* Untiled */
                init_array();

                printf("Doing %d flops of untiled test\n", FLOPS);
                do_untiled_cache_test(FLOPS);                        0.937465   1.04868e+06  1.04858e+06  2.09726e+06  3.44643e+08

                /* Indexed */
                init_array();

                printf("Doing %d flops of indirect address test\n", FLO
                do_indirect_address_test(FLOPS);                     0.512803   1.04866e+06  1.04866e+06  2.09731e+06  5.9909e+07
                printf("Done with series %i\n", i);

        }
        printf("result is %f\n", result );

testcpmod.c
```

***Figure 2. The Tool Gear infrastructure can be used to develop tools like this one, which gathers cache utilization and FLOP rate data from hardware performance counters. The user can select specific portions of the program to instrument while the program is running, with no need to compile or link special libraries into the application.***

of features that simplify the process of creating sophisticated performance analysis and debugging tools.

The ultimate goal is to develop tools like the one in Figure 2, which gathers cache utilization and FLOP rate data from hardware performance counters. The user can select specific portions of the program to instrument while the program is running, with no need to compile or link special libraries into the application.

LLNL has used these and other tools (such as the Sphinx parallel microbenchmark suite, http://www.llnl.gov/CASC/sphinx/sphinx.html) to gather a wealth of data on the performance characteristics of several scientific applications.

## Performance Tuning

The other goal of PPI is to develop system-level techniques for improving the performance of applications. We are examining patterns of memory accesses that programs generate and are developing techniques for handling these accesses more efficiently within memory subsystems. While standard caching methods work well when programs access the same data items repeatedly or in strict sequential order, many scientific codes access data in strided patterns or in other complex ways. We have developed tools to measure and characterize the memory access patterns of a program (Mohan, et al., 2001, and Parker, et al., 2001).

## Collaborations

LLNL works with many academic partners and vendors on performance analysis and tuning. We currently fund academic partners at the Universities of Maryland, Oregon, Tennessee, and Wisconsin, and at Portland State University and North Carolina State University. Vendor partners include IBM, MPI Software Technology, KAI, Pallas, and Etnus.

LLNL also has many ongoing collaborations with other major universities, vendors, and national laboratories. LLNL is a participant in the Performance Evaluation Research Center, funded by the Department of Energy's Scientific Discovery through Advanced Computing (SciDAC) initiative.

## Funding

Individual projects in PPI receive funding from a variety of sources, including ASCI Simulation Development Environments program (ASDE) and the Department of Energy's SciDAC program.

## References

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.

J.M. May, "MPX: Software for Multiplexing Hardware Performance Counters in Multithreaded Programs," Int. Parallel & Distr. Processing Sym. (IPDPS), 2001.

T. Mohan, B.R. de Supinski, S.A. McKee, F. Mueller, and A. Yoo, "Dynamic Detection of Streams in Memory References," SC 2001, Denver, CO, November 10-16, 2001.

E. Parker, B.R. de Supinski, and D.J. Quinlan, "Measuring the Regularity of Array References," SC 2001, Denver, CO, November 10-16, 2001.

*For more information:*

*John May, johnmay@llnl.gov;*
*Bronis de Supinski, bronis@llnl.gov; or*
*Jeffrey Vetter, jv@llnl.gov*

This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.