



# **Combining Linux\* Message Passing and Threading in High Performance Computing**

by Andrew Binstock

September 15, 2004

The information contained in this document is provided for informational purposes only and represents the current view of Intel Corporation Intel on the date of publication. Intel makes no commitment to update the information contained in this document, and Intel reserves the right to make changes at any time, without notice.

DISCLAIMER. THIS DOCUMENT AND ALL INFORMATION CONTAINED HEREIN IS PROVIDED AS IS. INTEL MAKES NO REPRESENTATIONS OF ANY KIND WITH RESPECT TO PRODUCTS REFERENCED HEREIN, WHETHER SUCH PRODUCTS ARE THOSE OF INTEL OR THIRD PARTIES. INTEL EXPRESSLY DISCLAIMS ANY AND ALL WARRANTIES, IMPLIED OR EXPRESS, INCLUDING WITHOUT LIMITATION, ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, AND ANY WARRANTY ARISING OUT OF THE INFORMATION CONTAINED HEREIN, INCLUDING WITHOUT LIMITATION, ANY PRODUCTS, SPECIFICATIONS, OR OTHER MATERIALS REFERENCED HEREIN. INTEL DOES NOT WARRANT THAT THIS DOCUMENT OR THE INFORMATION CONTAINED HEREIN IS FREE FROM ERRORS, OR THAT ANY PRODUCTS OR OTHER TECHNOLOGY DEVELOPED IN CONFORMANCE WITH THIS DOCUMENT WILL PERFORM IN THE INTENDED MANNER, OR WILL BE FREE FROM INFRINGEMENT OF THIRD PARTY PROPRIETARY RIGHTS, AND INTEL DISCLAIMS ALL LIABILITY THEREFOR.

INTEL DOES NOT WARRANT THAT ANY PRODUCT REFERENCED HEREIN OR ANY PRODUCT OR TECHNOLOGY DEVELOPED IN RELIANCE UPON THIS DOCUMENT, IN WHOLE OR IN PART, WILL BE SUFFICIENT, ACCURATE, RELIABLE, COMPLETE, FREE FROM DEFECTS OR SAFE FOR ITS INTENDED PURPOSE, AND HEREBY DISCLAIMS ALL LIABILITIES THEREFOR. ANY PERSON MAKING, USING OR SELLING SUCH PRODUCT OR TECHNOLOGY DOES SO AT HIS OR HER OWN RISK.

Licenses may be required. Intel and others may have patents or pending patent applications, trademarks, copyrights or other intellectual proprietary rights covering subject matter contained or described in this document. No license, express, implied, by estoppels or otherwise, to any intellectual property rights of Intel or any other party is granted herein. It is your responsibility to seek licenses for such intellectual property rights from Intel and others where appropriate.

Limited License Grant. Intel hereby grants you a limited copyright license to copy this document for your use and internal distribution only. You may not distribute this document externally, in whole or in part, to any other person or entity.

LIMITED LIABILITY. IN NO EVENT SHALL INTEL HAVE ANY LIABILITY TO YOU OR TO ANY OTHER THIRD PARTY, FOR ANY LOST PROFITS, LOST DATA, LOSS OF USE OR COSTS OF PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES, OR FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF YOUR USE OF THIS DOCUMENT OR RELIANCE UPON THE INFORMATION CONTAINED HEREIN, UNDER ANY CAUSE OF ACTION OR THEORY OF LIABILITY, AND IRRESPECTIVE OF WHETHER INTEL HAS ADVANCE NOTICE OF THE POSSIBILITY OF SUCH DAMAGES. THESE LIMITATIONS SHALL APPLY NOTWITHSTANDING THE FAILURE OF THE ESSENTIAL PURPOSE OF ANY LIMITED REMEDY.

Intel, the Intel logo, Pentium, Intel Xeon, and VTune are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

\*Other names and brands may be claimed as the property of others.

Copyright © 2005 Intel Corporation

## Introduction

Message passing is an important tool for sites relying on high-performance computing (HPC) on Linux\* systems. It enables large data sets to be tackled with ease and often represents an important adjunct or alternative to thread-based solution design. Integrating message passing and threading has its challenges, however.

High-performance computing is characterized by running multiple tasks in parallel. These tasks tend to be similar in nature and can be run on different systems (as in a cluster or a grid), different processors (on a single system) or even different execution pipelines (as on processors with Hyper-Threading Technology). HPC systems add computing power by the addition of discrete computing engines (be they blades, nodes, or processors) rather than by upgrading existing processors. This aspect underscores one of the salient aspects of HPC: the number of executing processes can vary, and so software must be architected to scale dynamically across the number of available execution units.

One way to write software that dynamically takes advantage of a variable number of execution resources is to use OpenMP\*, a set of threading interfaces and tools that greatly simplifies threaded programming. Using OpenMP pragmas (in C/C++, or directives in Fortran), you can generate an executable that determines at run time the appropriate number of threads to create and then distribute a workload over these threads dynamically. OpenMP's other advantages, such as portability across a wide set of platforms, are detailed in a series of articles on this Web site. These articles also explain how to get started with OpenMP. For additional information, go to the OpenMP home page at [www.openmp.org](http://www.openmp.org).

Another common approach to dynamic use of threading resources is to use the native threading APIs (Win32\* on Windows\* and Pthreads on Linux) in conjunction with operating system-specific calls to determine the number of available execution pipelines. Work is then distributed across these threads. Using native threading APIs confers low-level control at the cost of additional complexity of the code. But this approach is certainly workable. However, scientific computing often adds a dimension to HPC that prevents either OpenMP or native APIs from being a complete solution: the need to share data between processes at high speeds.

HPC often involves breaking large amounts of data into smaller blocks and then performing calculations in parallel on these data blocks. Once these calculations are complete, the results are frequently funneled to other processes that use them as input. In many cases, data must be moved back and forth between executing processes; and this data passing must be done at high speed, so as to not overly delay the work of the two processes.

## Data Passing Between Processes

The problem of passing data between processes has seen a wide variety of attempted solutions over the years, starting with UNIX's use of pipes and later sockets. In modern multiprocessor architecture, a common solution is shared memory, in which all processors can access the primary system memory. Processes then create queues in memory into which data items are placed in the form of messages.

Memory sharing, however, is not well suited to systems with more than 32 processors, because contention for access to the memory bus begins to choke performance. So, to develop scalable machines that might include hundreds or thousands of processors, system architects began looking at message passing protocols that would enable a process to send data to another process without using the memory bus. We will examine them shortly.

On systems such as clusters, data passing between nodes is still done awkwardly today. Many clusters running JVMs still use databases to share state information. This approach is somewhat workable on small clusters where not much data has to be shared, but it immediately leads to database thrash and excessive latency if volume spikes. Where performance is the entire story, such as in HPC, sites have chosen special message-passing solutions. These products are custom built for speed and today they are a foundational piece of HPC software architecture.

## Message Passing

In the early days of HPC, system vendors each had their own message-passing interface, which made porting HPC code very difficult. Researchers at many institutions developed numerous portable APIs to solve this problem. One API, called Parallel Virtual Machine (PVM) and designed at the Oak Ridge National Laboratories in Tennessee, USA. (see <http://www.csm.ornl.gov/pvm/>) is still very much in use today. Its primary orientation is towards running tasks in parallel on separate machines—such as workstations—and providing a robust message-passing mechanism between them.

When PVM first appeared, it was one of several competing solutions that provided a portable, message-passing API. To settle on a single, standard API, a group of vendors and researchers came together in the early 1990s to design a standard for message passing. The Forum eventually adopted the moniker of the Message Passing Interface Forum, or MPI Forum. The group's work is hosted today under the auspices of the Argonne National Laboratories at <http://www-unix.mcs.anl.gov/mpi/>. Its first standard was called Message Passing Interface (or just MPI 1). It underwent subsequent revisions during the 1990s, and it currently stands at MPI 2.

There are several open-source implementations of most of the MPI 2 spec available. The first, is MPI Chameleon (written MPICH and pronounced "Em-pitch"), which was written by the Argonne National Labs. It can be downloaded as source code from <http://www-unix.mcs.anl.gov/mpi/mpich2/> This implementation has been tested on Windows and Linux running on IA-32, IA-64, Alpha, and Opteron processors, as well as on Solaris. The download site also includes a Windows binary with the MSI installer file.

A competing open-source version, called Local Area Multicomputer (LAM) is available at <http://www.lam-mpi.org/>. LAM is somewhat more UNIX-oriented (it ships no binaries for Windows). Like MPICH, LAM is not a complete implementation of MPI 2, although it delivers most of the functionality of this revision. The LAM website offers numerous tools to use with the product as well as tutorials on the use of MPI. Both LAM and MPICH are widely used today.

## Code-Level MPI

Coding for MPI is not difficult. It consists of a predictable series of steps: initialize the environment, perform computing while sending and receiving messages via calls to MPI, close the environment. Figure 1a. shows a minimal program skeleton. Figure 1b. shows a possible output from running the code on a four-process system.

```
#include <stdio.h>
#include <mpi.h>

void main (int argc, char *argv[]) {

    int myrank, size;

    MPI_Init(&argc, &argv);          /* Initialize MPI    */

    MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* Get my rank    */
    MPI_Comm_size(MPI_COMM_WORLD, &size); /* Get the total
                                         number of processors */
    printf("Process %d of %d: Hello World!\n", myrank, size);

    MPI_Finalize();                 /* Terminate MPI   */

}
```

Figure 1a. A minimal MPI program that prints out 'Hello World!' from all the processors. (Courtesy University of Illinois)

If this program were run on each of four processes, the result would be (The order of the lines might differ.):

Process 2 of 4: Hello World!

Process 1 of 4: Hello World!

Process 3 of 4: Hello World!

Process 0 of 4: Hello World!

Figure 1b. Possible output from the code in Figure 1a, if it were run on each of four processes. (Courtesy University of Illinois)

Sending (and similarly receiving) messages can be done on a blocking or non-blocking basis. In the former case, blocking sends pause execution until it is safe to proceed while non-blocking sends return immediately. The blocking send API in C looks like this.

```
int MPI_Send(void *buf, int count, MPI_Datatype dtype,
             int dest, int tag, MPI_Comm comm);
```

Figure 2. Sending a message in MPI. `buf` is the address of the data location, `count` is the number of items of type `dtype` (These are MPI data types that include bytes and most integral types.). The `dest` and `tag` fields identify the destination process, and the `comm` parameter identifies the type of send to use. (Courtesy University of Illinois)

There are numerous other APIs, such as a broadcast call, which can send a starting value to all processes using MPI.

In MPI, there are two principal ways by which individual processes become part of MPI group. In the LAM implementation, a daemon is started on every machine, which then builds up the MPI environment. On MPICH, a configuration file similar to a HOSTS file is created and serves as a directory for the messaging operations. The documentation and tutorials available on the respective MPI sites provide considerable instruction on configuring and running programs with message passing.

It should be clear from this quick overview MPI provides a comparatively easy way to share data at high speed between multiple processes.

## Threading And MPI

Nothing in MPI prevents a program from using the standard threading APIs. Hence, MPI can be combined with OpenMP or Pthreads on Linux to exploit fully the processor's power while maintaining topflight inter-process communication. One workable way of doing this is to use MPI to distribute the tasks and OpenMP to actually execute them. To do this, certain safeguards must be in place:

The MPI tasks must be thread safe, otherwise the OpenMP implementation is likely to incorrectly execute the code.

MPI initialization code cannot be called from within OpenMP.

Message passing OpenMP threads needs to be done carefully. Of particular importance is the volume of messaging activity that goes on. If the use of OpenMP means that every thread will be sending and receiving messages, the messaging fabric is likely to become swamped with the traffic, which will deleteriously affect performance. Hence, it's important that threading not increase message loads significantly.

Finally, as in all threaded programming, it's important that thread synchronization not affect performance. On an HPC system, delay caused by synchronization might have no real effect or its impact could be dramatic if the entire system is waiting for a single task to finish.

To assist in extracting the maximum performance from this combination of resources, Intel provides several important developer tools. For performance analysis, developers can make use of the Intel® VTune™ Performance Analyzer. This tool collects performance data on a Linux system and displays its analysis on a Windows workstation. VTune Analyzer can show what is happening at the lowest levels of code.

Intel offers a threading add-on to the VTune Analyzer that enables Hyper-Threading execution profiles to be lifted from the Linux run-time, performance data. An additional plugin can analyze this data and diagnose common threading problems such as data races and threading conflicts.

Two tools recently acquired by Intel from German firm Pallas Software help developers track MPI events as they occur in a cluster. The Intel® Trace Collector gathers up the data, while the Intel® Trace Analyzer diagnoses what is happening between processes. A primary benefit of this analysis is that it enables developers to see the sequence of actions as they occur within the node. The Intel Trace Collector is designed to be a very low-overhead product so that the data collection process does not interfere with the normal operation and interaction of cluster nodes and of individual threads executing on those nodes. For more information on these tools, consult <http://www.intel.com/products/software/>

## Conclusion

Finally, many articles on this website describe the ways to use threading to maximize program performance. Many of them rightly suggest using OpenMP to do this in a portable, simple manner. However, HPC is not characterized only by the use of many threads and many processes; it is also marked by a need—especially evident in scientific computing—to share data between processes running in parallel. MPI is a powerful, portable way to implement this data sharing. And now, for the first time, Intel provides all the tools a developer might need to use threading and MPI effectively on the Linux platform.

## About the Author

Andrew Binstock is the principal analyst at Pacific Data Works LLC. Previously he was the director of PricewaterhouseCoopers's Global Technology Forecasts. He writes the business integration column for *SD Times*. Binstock co-authored "Programming with Hyper-Threading Technology," which is now available from Intel Press. He can be reached at [abinstock@pacificdataworks.com](mailto:abinstock@pacificdataworks.com)

## Additional Resources

A project funded by the US Department of Defense employed techniques discussed in this article. To read a detailed technical description of how MPI and OpenMP were combined, see the project report at:

[https://okcacs.wes.army.mil/ContentServer/ERDCDepot/OKC/Archive/PreviousTechReports/pdf/tr\\_9918.pdf](https://okcacs.wes.army.mil/ContentServer/ERDCDepot/OKC/Archive/PreviousTechReports/pdf/tr_9918.pdf)

[Intel® Software Products](#)

[Intel® HPC Developer Center](#)

[Intel Developer Services HPC Discussion Forum](#)

[Intel Linux Developer Center](#)

