

x
Loading

 Search

[Take our survey](#) for a chance to win an HP ProLiant BL465c Server Blade

Hi, [ozturan](#) | [Logout](#) | [Preferences](#)



newegg.com
ONCE YOU KNOW, YOU NEWEGG.®

SUPERMICRO
HIGH DENSITY COMPUTING
The Dawn Of A New Era in Performance & Efficiency
99% OF ORDERS SHIP WITHIN 1 BUSINESS DAY

LEARN MORE

-
-
-
-
-
-
-
-
- [Analysis](#)
- [Interviews](#)
- [Webinars](#)
- [White Papers](#)
- [Case Studies](#)

MPI on Multicore, an OpenMP Alternative?

No matter how you cut it, coding for multicore is really just parallel programming. Doug Eadline explains the differences between OpenMP and MPI, when it's smart to use existing code and when it's time to rewrite an application to scale better on multicore systems.

[Douglas Eadline, Ph.D.](#)
Tuesday, December 11th, 2007

No matter how you cut it, coding for multicore is really just parallel programming. Once you've realized that, it's time to look at the options, whether your existing codebase will scale, or if you need to rewrite your code and how.

As stated in [The Multicore Programming Challenge](#), parallel programming can be difficult. It moves the programmer closer to the hardware and further from their application space or problem. Fortunately, people like rocket scientists have been writing parallel software for quite some time in the HPC (High Performance Computing) sector.

As any good programmer knows, an existing code base can be valuable to current programming projects. First, the possibility of re-using existing code is a major incentive. Also, learning how someone else attacked a similar problem is very valuable.

In the HPC sector, most parallel programs are written using Message Passing Interface (MPI). While MPI is normally used on large computing systems (clusters) it can be also be used on a multicore processor. The "MPI proposition" may seem counter to conventional wisdom as MPI was designed for distributed memory (i.e. each core/processor has its own private memory), whereas OpenMP was designed for shared memory.

The lazy assumption suggests that OpenMP is a better solution because it was designed for shared memory. However, the possibility of re-using an existing MPI code base is worth considering before you spend a month(s) re-inventing the software wheel. Ultimately, the question is really about efficiency. Namely, *How does the performance of MPI compare to OpenMP on a multicore system?*

The answer to this question is important. If I can re-use MPI codes that work *well enough* on multicore, then there is no need to (re)write my application using OpenMP. If, on the other hand, OpenMP or threads provide scaling benefits sufficient enough to justify re-writing the code, then investing the time in re-coding might be in order.

Although your application(s) are always the ultimate test of hardware, a comparison of the same program written in MPI and OpenMP would be interesting. Fortunately for us, the people at NASA (the rocket science guys) have an interest in such things as well. The venerable [NAS Parallel Suite](#) is now available in MPI, OpenMP, Java, and HPF.

This enhancement means a head to head comparison of MPI and OpenMP is possible. (I'll leave the Java and HPF runs as an exercise for the reader). Before we get to the main event however, some background on how OpenMP and MPI differ may be helpful.

OpenMP and MPI Primer

Community Tools

[Share This](#)

Recommend This
☆☆☆☆ (No Ratings Yet)
Loading ...

Tags:
 2 Comments ([view all](#))

intel
Xeon
PROCESSOR

THE IBM
BLADECENTER
▶ TAKE THE BLADES ASSESSMENT

IBM BladeCenter features Quad-core Intel® Xeon® Processors.

- Video
- Podcasts
- White Papers



[Interview: Intel's Richard Dracott \(Part One\)](#)

Doug Eadline talks with Intel's Richard Dracott, General Manager of the High Performance Computing Organization.



[IBM Blue Gene is the World's Fastest Supercomputer](#)

Doug Eadline visits the IBM booth at SC'07 to get a look at IBM's Blue Gene.

Jumpstart Your Servers



[Enter to Win an HP BladeSystem for Your IT Infrastructure](#)

[Linux Simply Runs Better on ProLiant Servers](#)

[Harness the Power of Virtualization for Server Consolidation](#)

[SUSE Linux Enterprise Server: The Solution for Mission-critical Computing](#)

Because native Pthread programming can be cumbersome, a higher level of abstraction has been developed called OpenMP. As with all higher level approaches, OpenMP sacrifices flexibility for the ease of writing code. At its core, OpenMP uses threads, but the details are hidden from the programmer.

OpenMP is implemented as compiler directives in program comments. Typically, computationally heavy loops are augmented with OpenMP directives that the compiler uses to automatically “thread the loop”. This type of approach has the distinct advantage that it may be possible to leave the original program “untouched” (except for comment-directives) and provide simple recompilation for a sequential (non-threaded) version where the OpenMP directives are ignored. (Read the [OpenMP Web site](#) to get the complete picture.)

For those who don’t follow software trends, but instead rely on the crack *Linux Magazine* columnists to provide them with all the important advances, GCC 4.2 (and later) has support for OpenMP. This is important for the open source crowd, because OpenMP was only available in commercial compilers before GCC 4.2 was released.

GCC 4.2 has not found its way into all distributions, so you may need to [download](#) and build it from source if you want to play along with this article. Of course if you have a commercial compiler, it probably already has OpenMP support.

For gcc and gfortran, OpenMP programs can be compiled by including the `-fopenmp` option. In order to test this new capability, I found an OpenMP version of the ubiquitous matrix multiplication [program](#). I built two versions of the program, one with OpenMP enabled and one without:

```
$ gfortran -fopenmp -o matmult_omp matmult.f
$ gfortran -o matmult matmult.f
```

Then I ran the sequential version on an Intel Core 2 Duo system (two cores):

```
$time ./matmult

real    0m9.079s
user    0m8.988s
sys     0m0.012s
```

The OpenMP version was run as well. Note that there is an environment variable called `OMP_NUM_THREADS` that will tell OpenMP binaries how many threads to use. If this is not defined, one thread per CPU (core) is used. Ultimately however, the maximum number of threads may be defined by the program. The OpenMP results for two cores is shown below.)

```
$ time ./matmult_omp

real    0m4.967s
user    0m9.783s
sys     0m0.018s
```

The OpenMP version reduced the wall clock time by forty five percent. Astute readers may be wondering, why the *user* time is almost double the *real* time. This effect is due to using two cores, i.e. your total CPU time is a sum of the cores your application is uses. As we will see below, the *user* time can be quite a bit higher than the *real* time for eight cores.

In contrast to OpenMP, MPI uses a software library to send data from one process to another. Each process has its own memory space and thus MPI is basically a message copying methodology. In addition, MPI makes no distinction where a process runs. It can run on the same machine or on another machine. If one were to *time* an 8-way OpenMP and MPI program, the following would result (OpenMP is run first.):

```
$time bin/cg.B
real    1m11.735s
user    9m23.287s
sys     0m2.012s

$time mpirun -np 8 bin/cg.B.8
real    1m16.138s
user    0m0.000s
sys     0m0.004s
```

In the first case, OpenMP shows a *real* time of about one minute with *user* time of almost 9 and a half minutes indicating a good speed up. In the second case, the MPI run shows a comparable *real* time, but zero *user* time. This result is easily understood in terms of how MPI jobs are run. The *mpirun* command starts each separate MPI process and then waits until they are finished, thus no *user* time. OpenMP jobs, however, share a process space which makes them tractable to the OS.

The Process View

While we are talking about OpenMP and MPI, there’s one big difference between these programming methods in terms of the OS process space. OpenMP programs run as a single

[HP](#)

Free Email Newsletters

Blade & Virtualization Linux Magazine Case Study Update Linux Magazine Webinar Update Linux Magazine White Paper Update Linux Magazine PR Daily

SPONSORED LINKS

Simplify deployment of clusters using reference architecture and tools. **Intel Cluster Ready:** <http://www.intel.com/go/cluster>

The HP BladeSystem C3000 [Same capability. Less space. No compromise.](#)

process and the parallelism is expressed as threads. This behavior can be viewed quite clearly when using an eight core server (two quad-core processors). For instance, examining a running OpenMP program using *top* shows only a single process running. (See *Figure One*)

```

deadline@clover:~/NPB3.2.1/NPB3.2-MPI
File Edit View Terminal Tabs Help
top - 17:05:55 up 7:52, 3 users, load average: 2.40, 2.34, 1.72
Tasks: 193 total, 1 running, 192 sleeping, 0 stopped, 0 zombie
Cpu0 : 98.0%us, 0.0%sy, 0.0%ni, 2.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu1 : 97.0%us, 0.7%sy, 0.0%ni, 2.3%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu2 : 99.0%us, 0.3%sy, 0.0%ni, 0.7%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu3 : 99.0%us, 0.0%sy, 0.0%ni, 1.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu4 : 98.3%us, 0.0%sy, 0.0%ni, 1.7%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu5 : 98.0%us, 0.3%sy, 0.0%ni, 1.7%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu6 : 98.0%us, 1.0%sy, 0.0%ni, 1.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu7 : 99.0%us, 0.7%sy, 0.0%ni, 0.3%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 8143108k total, 5452124k used, 2690984k free, 238552k buffers
Swap: 2031608k total, 0k used, 2031608k free, 4302152k cached

  PID USER      PR  NI  VIRT  RES  SHR  S  %CPU  %MEM    TIME+  COMMAND
 17701 deadline  18   0  425m 341m  836  S   788   4.3   2:05.99 cg.B
  3615 root       18   0  5848  420  320  S    0.0   0.0   0:07.87 wulfd
 17695 deadline  15   0 12708 1140  800  R    0.0   0.0   0:00.37 top
    1 root       15   0 10304  660  548  S    0.0   0.0   0:02.09 init
    2 root       RT    0    0    0    0  S    0.0   0.0   0:00.01 migration/0
    3 root       34  19    0    0    0  S    0.0   0.0   0:00.00 ksoftirqd/0
    4 root       RT    0    0    0    0  S    0.0   0.0   0:00.00 watchdog/0
    5 root       RT    0    0    0    0  S    0.0   0.0   0:00.00 migration/1
    6 root       34  19    0    0    0  S    0.0   0.0   0:00.00 ksoftirqd/1
    7 root       RT    0    0    0    0  S    0.0   0.0   0:00.00 watchdog/1
    
```

Figure One: OpenMP program (cg.B) running on eight cores.

In contrast to the OpenMP, MPI actually starts one process per core using the `mpirun -np 8 ...` command. This situation is shown in *Figure Two* where an MPI version of the same program is now running. Note the number of processes is now eight. The processor (core) loads are about the same for both, however.

```

deadline@clover:~/NPB3.2.1/NPB3.2-OMP
File Edit View Terminal Tabs Help
top - 17:08:37 up 7:55, 3 users, load average: 4.17, 3.10, 2.12
Tasks: 201 total, 9 running, 192 sleeping, 0 stopped, 0 zombie
Cpu0 : 99.0%us, 1.0%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu1 : 99.7%us, 0.3%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu2 : 99.3%us, 0.7%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu3 : 99.3%us, 0.7%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu4 : 99.7%us, 0.3%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu5 : 99.0%us, 1.0%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu6 : 99.3%us, 0.7%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu7 :100.0%us, 0.0%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 8143108k total, 5540544k used, 2602564k free, 238800k buffers
Swap: 2031608k total, 0k used, 2031608k free, 4303928k cached

  PID USER      PR  NI  VIRT  RES  SHR  S  %CPU  %MEM    TIME+  COMMAND
 17719 deadline  18   0  112m  51m 2676  R   100   0.7   0:29.23 cg.B.8
 17715 deadline  18   0  112m  59m 3108  R   100   0.8   0:29.25 cg.B.8
 17716 deadline  24   0  112m  58m 2520  R   100   0.7   0:29.26 cg.B.8
 17717 deadline  19   0  112m  51m 2536  R   100   0.6   0:29.25 cg.B.8
 17718 deadline  19   0  112m  51m 2676  R   100   0.7   0:29.25 cg.B.8
 17722 deadline  18   0  112m  59m 2516  R   100   0.7   0:29.23 cg.B.8
 17720 deadline  18   0  112m  51m 2676  R   100   0.6   0:29.17 cg.B.8
 17721 deadline  18   0  112m  59m 2528  R   100   0.7   0:29.20 cg.B.8
  3615 root       18   0  5848  420  320  S    0.0   0.0   0:08.00 wulfd
 15252 root       15   0  69732 2816 2176  S    0.0   0.0   0:00.29 sshd
    1 root       15   0 10304  660  548  S    0.0   0.0   0:02.09 init
    
```

Figure Two: MPI program (cg.B.8) running on eight cores.

One final and subtle point. In OpenMP communication is through shared memory, which means *threads share access* to a memory location. With MPI programs on SMP systems communication is also through shared memory, but *processes send messages by writing from*

private to shared memory.

Obviously, sharing memory locations seems more efficient than sending copies of memory locations to other processes, but it all depends. In the MPI process model, single processes have exclusive access to all their process memory. For some programs this situation may be more efficient because it is better to copy data (send a message) than to wait for shared memory access. On the other hand, in the OpenMP model, threads can share access to all memory in the process space. In this case, some programs may be much more efficient as the large overhead of copying memory is not needed.

Looking at the Numbers

An eight-core Intel server (two four core Clovertown processors) was used to run the tests. The OpenMP tests used gcc/gfortran version 4.2. The MPI tests used LAM version 7.1.2. The OpenMP and MPI suites have six programs in common and each of these was run five times and averaged (Class B problem sizes were used). The results are given in Mops (million operations per second) in *Table One*. The percent difference is also shown.

Test	OpenMP gcc/gfortran 4.2	MPI LAM 7.1.2	Percent Difference
CG	790.6	739.1	7%
EP	166.5	162.8	2%
FT	3535.9	2090.8	69%
IS	51.1	122.5	139%
LU	5620.5	5168.8	9%
MG	1616.0	2046.2	27%

Table One: Results for the OpenMP/MPI benchmarks. (winning test is in bold)

Tests CG and EP are about the same. Indeed, EP is a good check as both methods should produce a similar result because there is very little communication. OpenMP is the clear winner with FT performance, but MPI does surprisingly better with the latency sensitive IS benchmark. In the fifth test, OpenMP does best with the LU benchmark, while MPI does best with MG. Overall the comparison is a bit of draw.

The results are clear on one point, there is not a definitive winner in this match-up. This result may come as a surprise to those who would assume, OpenMP would easily beat MPI on an multicore machine. (Or any SMP machine for that matter.) Maybe MPI is *good enough* to stand toe-to-toe with OpenMP for many applications.

In only one case (FT), did OpenMP run away from MPI. In other cases, MPI was a clear winner, and taking the time to convert your code to OpenMP would actually result in a performance loss. The story is far from over, more benchmarks are in order using other hardware and commercial compilers.

Other Things to Consider

Getting back to our question, “*do I need to re-code my MPI programs for these multicore things?*,” the answer is a resounding *maybe not*. MPI may just be good enough in many cases. Again, more data, and results for your application are needed for more solid recommendations.

Another important question to ask is how scalable your application is. As more processors are added, parallel execution will always hit a point of diminishing returns. This situation means that creating more threads or processes will not improve performance and it may actually hurt performance. The size of your data set may also come into play. One of the advantages of distributed MPI programs is the ability to distribute large data sets over many processors thereby solving problems that would never fit in an SMP memory space.

If you’re considering a writing a new application from scratch, the choice of OpenMP or MPI includes other considerations. OpenMP is designed for shared memory (SMP) machines. As multicore continues to grow the number of processors on an SMP will continue to grow, but OpenMP is not designed to run across multiple machines like MPI.

If you want your application to be portable on clusters and SMP machines, MPI might be the best solution. If, however, you do not envision using more than eight or sixteen cores, then OpenMP is probably one of your best choices if the benchmarks point in that direction. From a conceptual standpoint, those with experience in both paradigms state that using OpenMP and MPI provide a similar learning curve and *nuance level*. There are no shortcuts or free lunches with OpenMP, or MPI for that matter.

Douglas Eadline is the Senior HPC Editor for Linux Magazine.

2 Comments on MPI on Multicore, an OpenMP

Alternative? »

 [Comments via RSS](#)

hzmonte said:

+0  



Could you post (the link to) the OpenMP and MPI source code for OpenMP/MPI benchmark apps on the web?

April 8th, 2008 11:00 PM ([permalink](#))

[Reply to this comment](#)

deadline said:

+0  



There is a link in the article. Here it is again:

NAS Parallel Suite:

<http://www.nas.nasa.gov/Resources/Software/npb.html>

April 17th, 2008 12:57 PM ([permalink](#))

[Reply to this comment](#)

 [Comments via RSS](#)

You are logged in as **ozturan**. [Logout »](#)

Have a [Gravatar](#)? Your Gravatar pic will appear next to your comments. [?]

Your Comment

You may use `` `<abbr title="">` `<acronym title="">` `` `<blockquote cite="">` `<code>` `` `<i>` `<strike>` `` in your comment.



[About](#) | [Contact Us](#) | [Privacy Policy](#) | [FAQ](#) | [RSS](#)

[Back Issues](#) | [Subscribe](#) | [Give a Gift](#) | [Renew](#) | [Customer Service](#) | [Change Address](#) | [Advertise](#)

© Linux Magazine 1999-2008 All rights reserved.

LinuxMagazine.com v4.0