CHAPTER 5

Addressing Application Bottlenecks: Distributed Memory

The first application optimization level accessible to the ever-busy performance analyst is the distributed memory one, normally expressed in terms of the Message Passing Interface (MPI).[1] By its very nature, the distributed memory paradigm is concerned with communication. Some people consider all communication as *overhead*—that is, something intrinsically harmful that needs to be eliminated. We tend to call it "investment." Indeed, by moving data around in the right manner, you hope to get more computational power in return. The main point, then, is to optimize this investment so that your returns are maximized.

The time spent on the problem analysis and solution is an integral part of the overall investment. Hence, it is important to detect quickly what direction may be successful and what is going to be a waste of time, and to focus on the most promising leads. Following this pragmatic approach, in this chapter we will show how to detect and exploit optimization opportunities in the realm of communication patterns. Further chapters will step deeper into the increasingly local optimization levels. "And where are the algorithms?" you may ask. Well, we will deal with them as we go along, because algorithms will cross our path at every possible level. If you have ever tried to optimize bubble sort and then compared the result with the quick sort, you will easily appreciate the importance of algorithmic optimization.

Algorithm for Optimizing MPI Performance

Here is the algorithm we will use to optimize MPI performance, inspired in part by the work done by our friends and colleagues:[2]

1. Comprehend the underlying MPI performance.
2. Do an initial performance investigation of the application.

3. If the initial investigation indicates that performance may be improved, do an in-depth MPI analysis and optimization using the closed-loop approach, as follows:
    a. Get an overview of the application scalability and performance.
    b. If a load imbalance exceeds the cost of communication, address the load imbalance first or else perform MPI optimization.
    c. Repeat while performance improves and you still have time left.
4. Proceed to the node-level MPI optimization.

Let's go through these steps in detail.

Comprehending the Underlying MPI Performance

About the only sure way to grasp what is happening with application performance is to do benchmarking. Occasionally, you can deduce a performance estimate by plugging numbers into an analytical model that links, say, the estimated execution time to certain factors like the number of processes and their layout. However, this is more often the exception than the rule.

Recalling Some Benchmarking Basics

The first rule in benchmarking is to have a clean system setup. You have learned how to achieve that in Chapter 4. It may not always be possible to get to the ideal, no-interference situation, especially if you are doing your measurements on a system that is being utilized by many users at the same time, as they normally are. In this case, you will have to do several runs per parameter combination, possibly at different times of the day and week, and then apply statistical methods—or at least common sense—to estimate how reliable your data is.

To estimate the system variability, as well as to learn more about the underlying MPI performance, you may want to run Intel MPI Benchmarks (IMB).[3] Once started on a number of processes, this handy MPI benchmark will output timings, bandwidths, and other relevant information for several popular point-to-point and collective exchange patterns. You can also use any other benchmark you trust, but for now we will concentrate on the IMB, which was developed with the specific goal of representing typical application-level MPI use cases.

Gauging Default Intranode Communication Performance

Let us look first into the intranode communication—that is, data transfers done within one node. It is fairly easy to get started on IMB. The binary executable file `IMB-MPI1` is provided as part of the Intel MPI library distribution. Having set up the Intel MPI environment as described in Chapter 4, you can run this MPI program now in the way you probably know better than we do. On a typical system with the Intel MPI library installed, this would look as follows:

```
$ mpirun -np 2 ./IMB-MPI1 PingPong
```

By default, the Intel MPI library will try to select the fastest possible communication path for any particular runtime configuration. Here, the most likely candidate is the shared memory channel. On our workstation, this leads to the output (skipping unessential parts) shown in Listing 5-1:

Listing 5-1. Example IMB-MPI1 Output (Workstation, Intranode)

```
#---------------------------------------------------
# Benchmarking PingPong
# #processes = 2
#---------------------------------------------------
       #bytes #repetitions      t[usec]   Mbytes/sec
            0         1000         1.16         0.00
            1         1000         0.78         1.22
            2         1000         0.75         2.53
            4         1000         0.78         4.89
            8         1000         0.78         9.77
           16         1000         0.78        19.55
           32         1000         0.88        34.50
           64         1000         0.89        68.65
          128         1000         0.99       123.30
          256         1000         1.04       234.54
          512         1000         1.16       420.02
         1024         1000         1.38       706.15
         2048         1000         1.63      1199.68
         4096         1000         2.48      1574.10
         8192         1000         3.74      2090.00
        16384         1000         7.05      2214.91
        32768         1000        12.95      2412.56
        65536          640        14.93      4184.94
       131072          320        25.40      4921.88
       262144          160        44.55      5611.30
       524288           80        91.16      5485.08
      1048576           40       208.15      4804.20
      2097152           20       444.45      4499.96
      4194304           10       916.46      4364.63
```

The PingPong test is an elementary point-to-point exchange pattern, in which one MPI process sends a message to another and expects a matching response in return. Half of the turnaround time measured is dubbed "latency" in this case, and the message size divided by latency is called "bandwidth." These two numbers constitute the two most important characteristics of a message-passing communication path for a particular message size. If you want to reduce this to just two numbers for the whole message range, take the zero-byte message latency and the peak bandwidth at whatever message size it is achieved. Note, however, that IMB performance may differ from what you see in a real application.

From the output shown here we can deduce that zero-byte message latency is equal to 1.16 microseconds, while the maximum bandwidth of 5.6 GB/s is achieved on messages of 256 KiB. This is what the shared memory communication channel, possibly with some extra help from the networking card and other MPI implementor tricks, is capable of achieving in the default Intel MPI configuration. Note that the default intranode MPI latency in particular is 7 to 20 times the memory access latency, depending on the exact communication path taken (compare Listing 5-4). This is the price you pay for the MPI flexibility, and this is why people call all communication “overhead.” This overhead is what may make threading a viable option in some cases.

■ **Note** The Intel MPI Library is tuned by default for better bandwidth rather than for lower latency, so that the latency can easily be improved by playing a bit with the process pinning. We will look into this in due time.

The general picture of the bandwidth values (the last column in Listing 5-1) is almost normal: they start small, grow to the L2 cache peak, and then go down stepwise, basically reaching the main memory bandwidth on very long messages (most likely, well beyond the 4 MiB cutoff selected by default).

However, looking a little closer at the latency numbers (third column), we notice an interesting anomaly: zero-byte latency is substantially larger than that for 1-byte messages. Something is fishy here. After a couple of extra runs we can be sure of this (anomalous values are highlighted in italic; see Table 5-1):

Table 5-1. *Small Message Latency Anomaly (Microseconds, Workstation)*

#bytes	Run 1	Run 2	Run 3	Min
0	*1.16*	*1.31*	*1.28*	*1.16*
1	0.78	*1.03*	*1.27*	0.78
2	0.75	0.77	*1.04*	0.75
4	0.78	0.79	0.71	0.71

This may be a measurement artifact, but it may as well be something worth keeping in mind if your application is strongly latency bound. Note that doing at least three runs is a good idea, even though your Statistics 101 course told you that this is not enough to get anywhere close to certainty. Practically speaking, if you indeed have to deal with outliers, you will be extremely unlucky to get two or all three of them in a row. And if just one outlier is there, you will easily detect its presence and eliminate it by comparison to other two results. If you still feel unsafe after this rather unscientific passage, do the necessary calculations and increase the number of trials accordingly.

Let us try to eliminate the artifact as a factor by increasing tenfold the number of iterations done per message size from its default value of 1000:

```
$ mpirun -np 2 ./IMB-MPI1 -iter 10000 PingPong
```

The option `-iter 10000` requests 10,000 iterations to be done for each message size. This is what we get this time (again, skipping unessential output); see Listing 5-2.

Listing 5-2. Modified IMB-MPI1 Output (Workstation, Intranode, with 10,000 Iterations)

```
#---------------------------------------------------
# Benchmarking PingPong
# #processes = 2
#---------------------------------------------------
       #bytes #repetitions      t[usec]   Mbytes/sec
            0        10000         0.97         0.00
            1        10000         0.80         1.20
            2        10000         0.80         2.39
            4        10000         0.78         4.87
            8        10000         0.79         9.69
           16        10000         0.79        19.33
           32        10000         0.93        32.99
           64        10000         0.95        64.06
          128        10000         1.06       115.61
          256        10000         1.05       232.74
          512        10000         1.19       412.04
         1024        10000         1.40       697.15
         2048        10000         1.55      1261.09
         4096        10000         1.98      1967.93
         8192         5120         3.21      2437.08
        16384         2560         6.27      2493.14
        32768         1280        11.38      2747.05
        65536          640        13.35      4680.56
       131072          320        24.89      5021.92
       262144          160        44.77      5584.68
       524288           80        91.44      5467.92
      1048576           40       208.23      4802.48
      2097152           20       445.75      4486.85
      4194304           10       917.90      4357.78
```

From this, it does look like we get a measurement artifact at the lower message sizes, just because the machine is lightning fast. We can increase the iteration count even more and check that out.

EXERCISE 5-1

Verify the existence of the IMB short message anomaly on your favorite platform. If it is observable, file an issue report via Intel Premier Support.[4]

As before, the peak intranode bandwidth of 5.6 GiB/s at 256 KiB is confirmed, and we can deduce that the intranode bandwidth stabilizes at about 4.4 GB/s for large messages. These are quite reasonable numbers, and now we can proceed to investigate other aspects of the baseline MPI performance.

Before we do this, just to be sure, we will do two extra runs (oh, how important it is to be diligent during benchmarking!) and drive the new data into a new table (anomalous values are highlighted in italic again); see Table 5-2:

Table 5-2. *Small Message Latency Anomaly Almost Disappears (Microseconds, Workstation, with 10,000 Iterations)*

#bytes	Run 1	Run 2	Run 3	Min
0	*0.90*	*0.86*	*0.85*	*0.85*
1	0.69	0.72	0.72	0.69
2	0.70	0.71	0.73	0.70
4	0.70	0.71	0.72	0.70

Alternatively, if the observed anomaly can be attributed to the warm-up effects (say, connection establishment on the fly, buffer allocation, and so on), running another benchmark before the PingPong in the same invocation may eliminate this. The command would look as follows:

```
$ mpirun -np 2 ./IMB-MPI1 -iter 10000 PingPing PingPong
```

Listing 5-3 shows the effect we see on our workstation:

Listing 5-3. Modified IMB-MPI1 Output: PingPong after PingPing (Workstation, Intranode, with 10,000 Iterations)

```
#---------------------------------------------------
# Benchmarking PingPong
# #processes = 2
#---------------------------------------------------
       #bytes #repetitions      t[usec]   Mbytes/sec
            0        10000         0.56         0.00
            1        10000         0.56         1.69
            2        10000         0.57         3.37
            4        10000         0.57         6.73
```

```
           8        10000         0.58        13.27
          16        10000         0.58        26.49
          32        10000         0.69        44.49
          64        10000         0.69        88.48
         128        10000         0.78       155.68
         256        10000         0.81       300.65
         512        10000         0.93       527.47
        1024        10000         1.13       861.66
        2048        10000         1.50      1305.38
        4096        10000         2.14      1824.66
        8192         5120         3.73      2094.46
       16384         2560         6.48      2412.18
       32768         1280        11.83      2642.52
       65536          640        11.72      5334.40
      131072          320        22.33      5598.75
      262144          160        39.44      6338.08
      524288           80        76.32      6551.55
     1048576           40       183.25      5456.98
     2097152           20       402.50      4968.89
     4194304           10       783.05      5108.23
     8388608            5      1588.30      5036.84
    16777216            2      3417.25      4682.12
```

You can see not only that now the anomaly is gone but also that the numbers have changed quite substantially. This is in part why an application may behave differently from the most carefully designed benchmark. It is arguable whether doing special preconditioning of the benchmark like the one described earlier is valid all the time, so we will refrain from this approach further on.

Of course, we will keep all the log files, clearly named, safe and sound for future reference. The names like `IMB-MPI1-n1p2-PingPong.logN`, where `N` stands for the run number, will do just fine in this case. The notation `n1p2` tells us that the results have been obtained on one node using two MPI processes.

Gauging Default Internode Communication Performance

If you are addressing a cluster rather than a single node or a workstation, you will want to perform a comparable investigation of the internode performance. The principle is similar to the one explained in the previous section. Let's start again with the two-process IMB PingPong benchmark.

Since in this case we are going to use more than one node, MPI startup will of necessity be a bit more complicated, because the MPI library should be made aware of the identity of the nodes we intend to run on. By far the easiest way that also leaves a clear log trace of what exactly was done is to specify those nodes explicitly in the IMB invocation command. For instance, on our example system:

```
$ mpirun -host esg054 -np 1 ./IMB-MPI1 PingPong : -host esg055 -np
1 ./IMB-MPI1 PingPong
```

Here, `esg054` and `esg055` stand for the respective node hostnames. They are very likely to be rather different in your installation. If you're in doubt, ask your friendly systems administrator.

■ **Note** There are certainly more elegant and powerful ways of selecting the target nodes for an Intel MPI run. Do not worry; we will learn them one by one in due time. This precise inline method is just what we need right now.

Of course, your cluster may be controlled by a job-management system like PBS Pro, LSF, Torque, or one of half a dozen other alternative products. The chances are that `mpirun` will recognize any of them and allow a job to be started anyway, but this is a topic we would need to devote a whole chapter to. Just ask one of the local experts you know, and he or she will tell you what is needed to submit multiple node jobs.

Another conceptual complication that we will deal with is the way in which both nodes will communicate with each other. Normally, as in the intranode case, Intel MPI library will automatically try to select the fastest available communication path. Most likely, this will be InfiniBand on a dedicated HPC cluster and some Gigabit Ethernet on a general purpose cluster. In the case of InfiniBand, we get the following output on our test cluster introduced in Chapter 4; see Listings 5-4 and 5-5:

Listing 5-4. IMB-MPI1 Output (Cluster, Intranode)

```
#---------------------------------------------------
# Benchmarking PingPong
# #processes = 2
#---------------------------------------------------
       #bytes #repetitions      t[usec]   Mbytes/sec
            0         1000         0.67         0.00
            1         1000         0.67         1.42
            2         1000         0.68         2.82
            4         1000         0.68         5.62
            8         1000         0.70        10.85
           16         1000         0.71        21.54
           32         1000         0.86        35.63
           64         1000         0.88        69.40
          128         1000         0.98       124.95
          256         1000         0.99       246.72
          512         1000         1.15       426.27
         1024         1000         1.42       685.35
         2048         1000         1.78      1095.41
         4096         1000         2.79      1400.88
         8192         1000         4.64      1685.16
        16384         1000         8.20      1904.89
        32768         1000        15.10      2069.54
        65536          640        16.79      3721.45
       131072          320        31.61      3954.93
       262144          160        57.92      4316.18
       524288           80       107.18      4665.26
      1048576           40       238.57      4191.58
      2097152           20       503.15      3974.94
      4194304           10      1036.91      3857.63
```

Listing 5-5. IMB-MPI1 Output (Cluster, Internode)

```
#---------------------------------------------------
# Benchmarking PingPong
# #processes = 2
#---------------------------------------------------
       #bytes #repetitions      t[usec]   Mbytes/sec
            0         1000         1.09         0.00
            1         1000         1.09         0.88
            2         1000         1.09         1.75
            4         1000         1.10         3.47
            8         1000         1.10         6.91
           16         1000         1.11        13.74
           32         1000         1.15        26.44
           64         1000         1.16        52.71
          128         1000         1.23        98.97
          256         1000         1.87       130.55
          512         1000         1.98       246.30
         1024         1000         2.30       425.25
         2048         1000         2.85       685.90
         4096         1000         3.42      1140.67
         8192         1000         4.77      1639.06
        16384         1000         7.28      2145.56
        32768         1000        10.34      3021.38
        65536         1000        16.76      3728.35
       131072         1000        28.36      4407.30
       262144          800        45.51      5493.00
       524288          400        89.05      5614.98
      1048576          200       171.75      5822.49
      2097152          100       338.53      5907.97
      4194304           50       671.06      5960.72
```

Several interesting differences between the shared memory and the InfiniBand paths are worth contemplating. Let's compare these results graphically; see Figures 5-1 and 5-2.

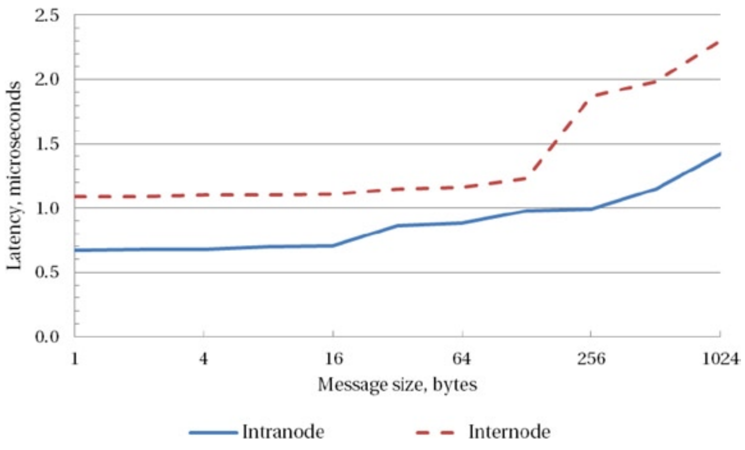


Figure 5-1. *IMB-MPI1 PingPong latency comparison: cluster, intranode vs internode (lower is better)*

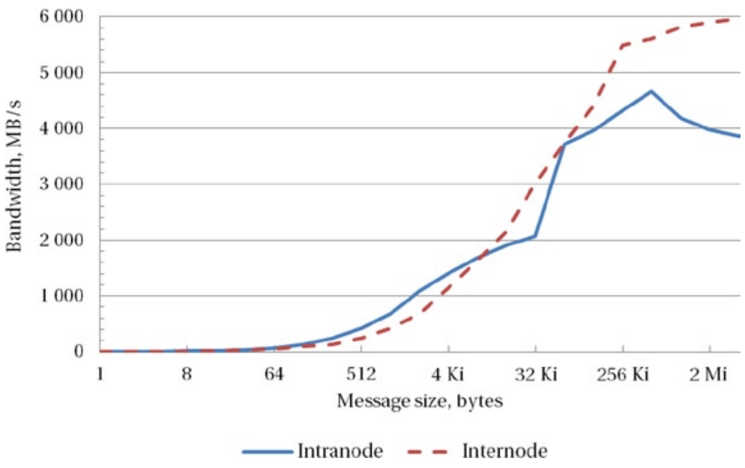


Figure 5-2. *IMB-MPI1 PingPong bandwidth comparison: cluster, intranode vs internode (higher is better)*

Now, let's enumerate the differences that may be important when we later start optimizing our application on the target cluster:

1. Intranode latency is substantially better than internode latency on smaller message sizes, with the crossover occurring at around 8 KiB. Hence, we should try to put onto the same node as many processes that send smaller messages to each other as possible.
2. Internode bandwidth is considerably higher than intranode bandwidth on larger messages above 8 KiB, with the exception of roughly 64 KiB, where the curves touch again. Hence, we may want to put onto different nodes those MPI ranks that send messages larger than 8 KiB, and surely larger than 64 KiB, to each other.
3. It is just possible that InfiniBand might be beating the shared memory path on the intranode bandwidth, as well. Since Intel MPI is capable of exploiting this after a minor adjustment, another small investigation is warranted to ascertain whether there is any potential for performance improvement in using InfiniBand for larger message intranode transfers.

Discovering Default Process Layout and Pinning Details

Is there an opportunity to further improve the underlying MPI performance? Certainly there are quite a few, starting with improving process pinning. Let's look at the output of the `cpuinfo` utility that is provided with the Intel MPI library; see Listing 5-6:

Listing 5-6. Cpuinfo Utility Output (Workstation)

```
Intel(R) processor family information utility, Version 5.0 Update 1 Build
20140709
Copyright (C) 2005-2014 Intel Corporation. All rights reserved.

=====  Processor composition  =====
Processor name    : Genuine Intel(R)  E2697V
Packages(sockets) : 2
Cores             : 24
Processors(CPUs)  : 48
Cores per package : 12
Threads per core  : 2
```

```
=====  Processor identification  =====
Processor         Thread Id.        Core Id.          Package Id.
0                 0                 0                 0
1                 0                 1                 0
2                 0                 2                 0
3                 0                 3                 0
4                 0                 4                 0
5                 0                 5                 0
6                 0                 8                 0
7                 0                 9                 0
8                 0                 10                0
9                 0                 11                0
10                0                 12                0
11                0                 13                0
12                0                 0                 1
13                0                 1                 1
14                0                 2                 1
15                0                 3                 1
16                0                 4                 1
17                0                 5                 1
18                0                 8                 1
19                0                 9                 1
20                0                 10                1
21                0                 11                1
22                0                 12                1
23                0                 13                1
24                1                 0                 0
25                1                 1                 0
26                1                 2                 0
27                1                 3                 0
28                1                 4                 0
29                1                 5                 0
30                1                 8                 0
31                1                 9                 0
32                1                 10                0
33                1                 11                0
34                1                 12                0
35                1                 13                0
36                1                 0                 1
37                1                 1                 1
38                1                 2                 1
39                1                 3                 1
40                1                 4                 1
41                1                 5                 1
42                1                 8                 1
43                1                 9                 1
44                1                 10                1
```

```
45              1               11              1
46              1               12              1
47              1               13              1
=====  Placement on packages  =====
Package Id.     Core Id.        Processors
0                               0,1,2,3,4,5,8,9,10,11,12,13
                                (0,24)(1,25)(2,26)(3,27)(4,28)(5,29)(6,30)
                                (7,31)(8,32)(9,33)(10,34)(11,35)
1                               0,1,2,3,4,5,8,9,10,11,12,13
                                (12,36)(13,37)(14,38)(15,39)(16,40)(17,41)
                                (18,42)(19,43)(20,44)(21,45)(22,46)(23,47)

=====  Cache sharing  =====
Cache   Size            Processors
L1      32  KB          (0,24)(1,25)(2,26)(3,27)(4,28)(5,29)(6,30)(7,31)
                        (8,32)(9,33)(10,34)(11,35)(12,36)(13,37)(14,38)
                        (15,39)(16,40)(17,41)(18,42)(19,43)(20,44)(21,45)
                        (22,46)(23,47)
L2      256 KB          (0,24)(1,25)(2,26)(3,27)(4,28)(5,29)(6,30)(7,31)
                        (8,32)(9,33)(10,34)(11,35)(12,36)(13,37)(14,38)
                        (15,39)(16,40)(17,41)(18,42)(19,43)(20,44)(21,45)
                        (22,46)(23,47)
L3      30  MB          (0,1,2,3,4,5,6,7,8,9,10,11,24,25,26,27,28,29,30,31,
                        32,33,34,35)(12,13,14,15,16,17,18,19,20,21,22,23,
                        36,37,38,39,40,41,42,43,44,45,46,47)
```

This utility outputs detailed information about the Intel processors involved. On our example workstation we have two processor packages (sockets) of 12 physical cores apiece, each of them in turn running two hardware threads, for the total of 48 hardware threads for the whole machine. Disregarding gaps in the core numbering, they look well organized. It is important to notice that both sockets share the 30 MB L3 cache, while the much smaller L1 and L2 caches are shared only by the virtual cores (OS processors) that are closest to each other in the processor hierarchy. This may have interesting performance implications.

Now, let's see how Intel MPI puts processes onto the cores by default. Recalling Chapter 1, for this we can use any MPI program, setting the environment variable `I_MPI_DEBUG` to 4 in order to get the process mapping output. If you use a simple start/stop program containing only calls to the `MPI_Init` and `MPI_Finalize`, you will get output comparable to Listing 5-7, once unnecessary data is culled from it:

Listing 5-7. Default Process Pinning (Workstation, 16 MPI Processes)

```
[0] MPI startup(): Rank    Pid      Node name  Pin cpu
[0] MPI startup(): 0       210515   book       {0,1,24}
[0] MPI startup(): 1       210516   book       {2,25,26}
[0] MPI startup(): 2       210517   book       {3,4,27}
[0] MPI startup(): 3       210518   book       {5,28,29}
[0] MPI startup(): 4       210519   book       {6,7,30}
```

```
[0] MPI startup(): 5       210520   book        {8,31,32}
[0] MPI startup(): 6       210521   book        {9,10,33}
[0] MPI startup(): 7       210522   book        {11,34,35}
[0] MPI startup(): 8       210523   book        {12,13,36}
[0] MPI startup(): 9       210524   book        {14,37,38}
[0] MPI startup(): 10      210525   book        {15,16,39}
[0] MPI startup(): 11      210526   book        {17,40,41}
[0] MPI startup(): 12      210527   book        {18,19,42}
[0] MPI startup(): 13      210528   book        {20,43,44}
[0] MPI startup(): 14      210529   book        {21,22,45}
[0] MPI startup(): 15      210530   book        {23,46,47}
```

Comparing Listings 5-6 and 5-7, we can see that the first eight MPI processes occupy the first processor package, while the remaining eight MPI processes occupy the other package. This is good if we require as much bandwidth as we can get, for two parts of the job will be using separate memory paths. This may be bad, however, if the relatively slower intersocket link is crossed by very short messages that clamor for the lowest possibly latency. That situation would normally favor co-location of the intensively interacting processes on the cores that share the highest possible cache level, up to and including L1.

Gauging Physical Core Performance

What remains to be investigated is how much the virtual cores we have been using so far influence pure MPI performance. To look into this, we have to make Intel MPI use only the physical cores. The easiest way to do this is as follows:

```
$ export I_MPI_PIN_PROCESSOR_LIST=allcores
```

If you wonder what effect this will have upon performance, compare Listing 5-1 with Listing 5-8:

Listing 5-8. Example IMB-MPI1 Output (Workstation, Intranode, Physical Cores Only)

```
#-----------------------------------------------------
# Benchmarking PingPong
# #processes = 2
#-----------------------------------------------------
       #bytes #repetitions      t[usec]   Mbytes/sec
            0         1000         0.58         0.00
            1         1000         0.61         1.56
            2         1000         0.62         3.08
            4         1000         0.27        14.21
            8         1000         0.28        27.65
           16         1000         0.32        48.05
           32         1000         0.37        81.48
           64         1000         0.38       161.67
```

```
         128         1000         0.42       293.83
         256         1000         0.44       556.07
         512         1000         0.50       975.70
        1024         1000         0.59      1659.31
        2048         1000         0.79      2470.82
        4096         1000         1.21      3229.65
        8192         1000         2.06      3799.85
       16384         1000         3.77      4145.09
       32768         1000         6.79      4605.72
       65536          640        10.30      6066.17
      131072          320        18.66      6699.50
      262144          160        35.94      6956.02
      524288           80        65.84      7593.73
     1048576           40       125.46      7970.55
     2097152           20       245.08      8160.72
     4194304           10       482.80      8285.04
```

Note that we can still observe the small message latency anomaly in some form. This becomes outright intriguing. For the rest of it, latency is down by up to three times and bandwidth is up by 40 to 50 percent, with bandwidth in particular still going up, whereas it would sharply drop in prior tests. This is natural: in the absence of necessity to share both the core internals and the off-core resources typical of the virtual cores, MPI performance will normally go up. This is why pure MPI programs may experience a substantial performance hike when run on the physical cores.

Note also that the performance hike observed here has to do as well with the change in the process layout with respect to the processor sockets. If you investigate the process layout and pinning in both cases (not shown), you will see that in the default configuration, MPI ranks 0 and 1 occupy different processor sockets, while in the configuration illustrated by Listing 5-8, these ranks sit on adjacent physical cores of the same processor socket. That is, the observed difference is also the difference between the intersocket and intrasocket performance, respectively.

At this point we have discovered about 90 percent of what needs to be known about the underlying MPI performance. You might want to run more complicated IMB sessions and see how particular collective operations behave on more than two processes and so on. Resist this temptation. Before we go there, we need to learn a bit more about the target application.

EXERCISE 5-2

Compare the virtual and physical core performance of your favorite platform using the procedure described here. Try the `-cache_off` option of the IMB to assess the influence of the cache vs. memory performance at the MPI level. Consider how relevant these results may be to your favorite application.

Doing Initial Performance Analysis

Let us proceed to the next step of the performance investigation algorithm. When you optimize an application at the MPI level, it is not so interesting at first what is happening inside any particular process. What is more pertinent is how these processes interact, how much time is spent doing this, and whether this interaction can be improved to a noticeable degree. Thus, performance investigation of an MPI application starts with the initial benchmarking and a couple of estimates.

Is It Worth the Trouble?

This is the first question to answer, and this is not a trivial matter. One measurement is not likely to give the final answer here, since application behavior may depend on the run configuration (number of nodes, kind of the fabrics selected, MPI settings), as well as on the workload used and other, sometimes outright mysterious, factors.

Following the typical engineering practice of estimating upfront the problem by the order of magnitude, we recommend you do the following first:

1. Select one or two representative workloads.
2. Use the default Intel MPI settings and activate the built-in statistics gathering to collect vital profile information (`export I_MPI_STATS=ipm`).
3. Do several benchmarking runs at a low, medium, and high (but still practicable) number of processes, for any curve can connect two points, as they say.[5]
4. Analyze the statistics output files to find out whether it is worth bothering about the application's distributed memory performance, in particular.

By following this routine, you will not only understand whether there is a noticeable optimization potential at the distributed memory level but also learn how your application scales with the number of nodes and what MPI operations it uses most extensively. Moreover, you will establish a performance baseline that you will compare your results against every time you introduce a purported improvement into the application or the platform. All this information will flow directly into the further optimization process, and none of your time will be wasted.

Example 1: Initial HPL Performance Investigation

Let us revisit the High Performance Linpack Benchmark that we mentioned in Chapter 1, and practice a little on it.[6] To save time in configuring and building an executable with all the necessary optimizations and libraries inside, we will fetch Intel's pre-cooked HPL that we quietly used in Chapter 4.[7]

We do not have to select the workload because HPL generates it automatically during startup. What we need to change are a few workload parameters; see Listing 5-9:

Listing 5-9. HPL Input File (HPL.dat) with the Most Important Parameters Highlighted

```
HPLinpack benchmark input file
Innovative Computing Laboratory, University of Tennessee
HPL.out      output file name (if any)
6            device out (6=stdout,7=stderr,file)
1            # of problems sizes (N)
235520       Ns
1            # of NBs
256          NBs
1            PMAP process mapping (0=Row-,1=Column-major)
1            # of process grids (P x Q)
4            Ps
4            Qs
16.0         threshold
1            # of panel fact
2            PFACTs (0=left, 1=Crout, 2=Right)
1            # of recursive stopping criterium
4            NBMINs (>=1)
1            # of panels in recursion
2            NDIVs
1            # of recursive panel fact.
1            RFACTs (0=left, 1=Crout, 2=Right)
1            # of broadcast
0            BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
1            # of lookahead depth
1            DEPTHs (>=0)
0            SWAP (0=bin-exch,1=long,2=mix)
1            swapping threshold
1            L1 in (0=transposed,1=no-transposed) form
1            U  in (0=transposed,1=no-transposed) form
0            Equilibration (0=no,1=yes)
8            memory alignment in double (>0)
```

Some of the points to note from the script in Listing 5-8 are:

- *Problem size* (N) is normally chosen to take about 80 percent of the available physical memory by the formula *memory* = $8N^2$ for double precision calculations.
- *Number of blocks* (NB) usually ranges between 32 and 256, with the higher numbers promoting higher computational efficiency while creating more communication.
- *Process grid dimensions* (P and Q), where both P and Q are typically greater than 1, P is equal to or slightly smaller than Q, and the product of P and Q is the total number of processes involved in the computation.

This and further details are well explained in the *HPL FAQ*.[8] As can be seen, Listing 5-9 was generated when the matrix size was set to 235,520, yielding total occupied memory of about 413 GiB. We used 256 blocks and the process grid dimensions 4 x 4. A quick look into the built-in statistics output given in Listing 1-1 that was obtained for this input data shows that MPI communication occupied between 5.3 and 11.3 percent of the total run time, and that the `MPI_Send`, `MPI_Recv`, and `MPI_Wait` operations took about 81, 12, and 7 percent of the total MPI time, respectively. The truncated HPL output file (see Listing 5-10) reveals that the run completed correctly, took about 40 minutes, and achieved about 3.7 TFLOPS.

Listing 5-10. HPL Report with the Most Important Data Highlighted (Cluster, 16 MPI Processes)

```
================================================================================
HPLinpack 2.1 -- High-Performance Linpack benchmark -- October 26, 2012
Written by A. Petitet and R. Clint Whaley,  Innovative Computing Laboratory, UTK
Modified by Piotr Luszczek, Innovative Computing Laboratory, UTK
Modified by Julien Langou, University of Colorado Denver
================================================================================

An explanation of the input/output parameters follows:

T/V    : Wall time / encoded variant.
N      : The order of the coefficient matrix A.
NB     : The partitioning blocking factor.
P      : The number of process rows.
Q      : The number of process columns.
Time   : Time in seconds to solve the linear system.
Gflops : Rate of execution for solving the linear system.

The following parameter values will be used:

N        :  235520
NB       :     256
PMAP     : Column-major process mapping
P        :       4
Q        :       4
PFACT    : Right
NBMIN    :       4
NDIV     :       2
RFACT    : Crout
BCAST    : 1ring
DEPTH    :       1
SWAP     : Binary-exchange
L1       : no-transposed form
U        : no-transposed form
EQUIL    : no
ALIGN    :    8 double precision words
```

```
--------------------------------------------------------------------------------
- The matrix A is randomly generated for each test.
- The following scaled residual check will be computed:
      ||Ax-b||_oo / ( eps * ( || x ||_oo * || A ||_oo + || b ||_oo ) * N )
- The relative machine precision (eps) is taken to be 1.110223e-16
- Computational tests pass if scaled residuals are less than   16.0
Column=001280 Fraction=0.005 Mflops=4809238.67
Column=002560 Fraction=0.010 Mflops=4314045.98
...
Column=210944 Fraction=0.895 Mflops=3710381.21
Column=234496 Fraction=0.995 Mflops=3706630.12
================================================================================
T/V                N    NB     P     Q               Time                 Gflops
--------------------------------------------------------------------------------
WC10C2R4      235520   256     4     4            2350.76            3.70500e+03
HPL_pdgesv() start time Fri Feb 14 05:44:48 2014

HPL_pdgesv() end time   Fri Feb 14 06:23:59 2014
--------------------------------------------------------------------------------
||Ax-b||_oo/(eps*(||A||_oo*||x||_oo+||b||_oo)*N)=    0.0028696 ...... PASSED
================================================================================
Finished      1 tests with the following results:
              1 tests completed and passed residual checks,
              0 tests completed and failed residual checks,
              0 tests skipped because of illegal input values.
--------------------------------------------------------------------------------
End of Tests.
================================================================================
```

Now, let’s document what we have found. The input and output files form the basis of this dataset that needs to be securely stored. In addition to this, we should note that this run was done on eight nodes with two Ivy Bridge processors, with 12 physical cores in turbo mode per processor and 64 GiB of memory per node.

The following tools were used for this run:

- Intel MPI 5.0.1
- Intel MKL 11.2.0 (including MP_LINPACK binary precompiled by Intel Corporation)
- Intel Composer XE 2015

The environment variables for this test were as follows:

```
export I_MPI_DAPL_PROVIDER=ofa-v2-mlx4_0-1
export I_MPI_PIN=enable
export I_MPI_PIN_DOMAIN=socket
export OMP_NUM_THREADS=12
export KMP_AFFINITY=verbose,granularity=fine,physical
export I_MPI_STATS=ipm
```

Some of these variables are set by default. However, setting them explicitly increases the chances that we truly know what is being done by the library. The first line indicates a particular communication fabric to be used by Intel MPI. The next four lines control the Intel MPI and OpenMP process and thread pinning. (We will look into why and how here, and in Chapter 6.) The last line requests the built-in, IPM-style statistics output to be produced by the Intel MPI Library.

This dataset complements the lower-level data about the platform involved that we collected and documented in Chapter 4. Taken together, they allow us to reproduce this result if necessary, or to root-cause any deviation that may be observed in the future (or in the past).

Since this program has not been designed to run on small problem sizes or small numbers of processes, it does not make much sense to continue the runs before we come to the preliminary conclusion. One data point will be sufficient, and we can decide what to do next. If we compute the efficiency achieved during this run, we see it comes to about 90 percent. This is not far from the expected top efficiency of about 95 percent. From this observation, as well as the MPI communication percentages shown here and Amdahl's Law explained earlier, we can deduce that there is possibly 2—at most 3—percent overall performance upside in tuning MPI. In other words, it makes sense to spend more time tuning MPI for this particular application only if you are after those last few extra drops of efficiency. This may very well be the case if you want to break a record, by the way.

Just for comparison, we took the stock HPL off the `netlib.org` and compared it to the optimized version presented here. The only tool in common was the Intel MPI library. We used the GNU compiler, BLAS library off the `netlib.org`, and whatever default settings were included in the provided `Makefile`.[9] First, we were not able to run the full-size problem owing to a segmentation fault. Second, the matrix size of 100,000 was taking so much time it was impractical to wait for its completion. Third, on the very modest matrix size of 10,000, with the rest of the aforementioned `HPL.dat` file unchanged (see Listing 5-9), we got 35.66 GFLOPS for the stock HPL vs. 152.07 GFLOPS for the optimized HPL, or a factor of more than *4.5 times* in favor of the optimized HPL. As we know from the estimates given, and a comparison of the communication statistics (not shown), most of this improvement does not seem to be coming from the MPI side of the equation. We will revisit this example in the coming chapters dedicated to other levels of optimization to find out how to get this fabulous acceleration.

All this may look to you like a very costly exercise in finding out the painfully obvious. Of course, we know that Intel teams involved in the development of the respective tools have done a good job optimizing them for one of the most influential HPC benchmarks. We also know what parameters to set and how, both for the application and for the Intel MPI Library. However, all this misses the point. Even if you had a different application at hand, you would be well advised to follow this simple and efficient routine before going any further. By the way, if you miss beautiful graphs here, you will do well to get used to this right away. Normally you will have no time to produce any pictures, unless you want to unduly impress your clients or managers. Well-organized textual information will often be required if you take part in the formal benchmarking efforts. If you would rather analyze data visually, you will have to find something better than the plain text tables and Excel graphing capabilities we have gotten used to.

EXERCISE 5-3

Do an initial performance investigation of the HPCG benchmark,[10] and determine whether it is desirable and indeed feasible to improve its distributed memory performance. Repeat this exercise with your favorite application.

Getting an Overview of Scalability and Performance

If an initial investigation suggests that there may be some improvement potential in the area of distributed memory performance, and if a couple of the simple tricks described in Chapter 1 do not yield a quick relief, it is time to start an orderly siege. The primary goal at this point is to understand whether the application is memory-bound or compute-bound, whether it scales as expected (if scaling is indeed a goal), and how the observed performance relates to the expected peak performance of the underlying platform.

Learning Application Behavior

Now, you are in for a lot of benchmarking. Proper selection of the representative workloads, application parameters, MPI process and OpenMP thread counts, and other relevant settings are paramount. Also desirable are scripting skills or a special tool that will help you run benchmarks and organize the pile of resulting data.

There is also a distinct temptation at this stage to follow the white rabbit down the hole. Try hard to resist this temptation because the first performance issue you observe may or may not be the primary one, both in its causal importance and in its relative magnitude. Only when you have a complete overview of the application behavior and its quirks will you be able to chart the most effective way of addressing the real performance issues, if indeed there are any. Let us look at a very good example of this view that we will keep revisiting as we go.

Example 2: MiniFE Performance Investigation

The miniFE application from the Mantevo suite represents a typical finite element method of solving implicit unstructured partial differential equations.[11] It includes all important solution stages, like the sparse matrix assembly and the solution of the resulting system of linear equations. Thus, whatever we learn here may be directly transferrable to the more involved packages that provide general finite element capabilities.

It is relatively easy to set up and run this application. Upon downloading the application archive and unpacking it recursively, you will end up with a number of directories, one of which is named `miniFE-2.0_mkl`. Note that we quite intentionally go for the Intel MPI and Intel MKL-based executable here, for we have learned earlier in the example of HPL that this gives us a head start on performance. In other words, by now we are almost past the recommendations of Chapter 1 and into the realm of the unknown.

First, you need to fetch and build the program. Upon unpacking, go to the directory `miniFE-2.0-mkl/src`, copy the `Makefile.intel.mpi` into the `Makefile`, change the `-fopenmp` flag there to `-qopenmp` so that the multithreaded Intel MPI Library is picked up by default, then type `make` and enjoy.

Next you need to find a proper workload. After a couple of attempts, with the system sizes of 10 and 100 being apparently too small, and the system size of 1000 leading to the operating system killing the job (results not shown), the following launch string looks adequate:

```
$ mpirun -np 16 ./miniFE.x nx=500
```

This command produces the output seen in Listing 5-11:

Listing 5-11. MiniFE Output (Workstation, Size 500, 16 MPI Processes)

```
      creating/filling mesh...0.377832s, total time: 0.377833
generating matrix structure...17.6959s, total time: 18.0737
         assembling FE data...13.5461s, total time: 31.6199
      imposing Dirichlet BC...11.9997s, total time: 43.6195
      imposing Dirichlet BC...0.47753s, total time: 44.0971
making matrix indices local...14.6372s, total time: 58.7342
Starting CG solver ...
Initial Residual = 501.001
Iteration = 20 Residual = 0.0599256
Iteration = 40 Residual = 0.0287661
Iteration = 60 Residual = 0.0185888
Iteration = 80 Residual = 0.121056
Iteration = 100 Residual = 0.0440518
Iteration = 120 Residual = 0.00938303
Iteration = 140 Residual = 0.00666799
Iteration = 160 Residual = 0.00556699
Iteration = 180 Residual = 0.00472206
Iteration = 200 Residual = 0.00404725
Final Resid Norm: 0.00404725
```

There is also a corresponding report file with the file extension `.yaml`, shown in Listing 5-12:

Listing 5-12. MiniFE Report (Workstation, Size 500, 16 MPI Processes)

```
Mini-Application Name: miniFE
Mini-Application Version: 2.0
Global Run Parameters:
  dimensions:
    nx: 500
    ny: 500
    nz: 500
  load_imbalance: 0
  mv_overlap_comm_comp: 0 (no)
  number of processors: 16
  ScalarType: double
  GlobalOrdinalType: int
  LocalOrdinalType: int
Platform:
  hostname: book
  kernel name: 'Linux'
  kernel release: '2.6.32-431.17.1.el6.x86_64'
  processor: 'x86_64'
Build:
  CXX: '/opt/intel/impi_latest/intel64/bin/mpiicpc'
  compiler version: 'icpc (ICC) 15.0.0 20140723'
  CXXFLAGS: '-O3 -mkl -DMINIFE_MKL_DOUBLE -qopenmp -DUSE_MKL_DAXPBY -mavx'
  using MPI: yes
Run Date/Time: 2014-05-27, 19-21-30
Rows-per-proc Load Imbalance:
  Largest (from avg, %): 0
  Std Dev (%): 0
Matrix structure generation:
  Mat-struc-gen Time: 17.6959
FE assembly:
  FE assembly Time: 13.5461
Matrix attributes:
  Global Nrows: 125751501
  Global NNZ: 3381754501
  Global Memory (GB): 38.731
  Pll Memory Overhead (MB): 28.8872
  Rows per proc MIN: 7812500
  Rows per proc MAX: 7938126
  Rows per proc AVG: 7.85947e+06
  NNZ per proc MIN: 209814374
  NNZ per proc MAX: 213195008
  NNZ per proc AVG: 2.1136e+08
```

```
CG solve:
  Iterations: 200
  Final Resid Norm: 0.00404725
  WAXPY Time: 21.2859
  WAXPY Flops: 2.2575e+11
  WAXPY Mflops: 10605.6
  DOT Time: 6.72744
  DOT Flops: 1e+11
  DOT Mflops: 14864.5
  MATVEC Time: 98.8167
  MATVEC Flops: 1.35947e+12
  MATVEC Mflops: 13757.4
  Total:
    Total CG Time: 126.929
    Total CG Flops: 1.68522e+12
    Total CG Mflops: 13276.9
  Time per iteration: 0.634643
Total Program Time: 185.796
```

From the last few lines of Listing 5-12, you can see that we achieve about 13.3 GFLOPS during the conjugate gradient (CG) solution stage, taking 185.8 seconds for the whole job. Now we will look into whether this is the optimum we are after with respect to the problem size, the number of the MPI processes, and the number of OpenMP threads that are used implicitly by the Intel MKL. For comparison, we achieved only 10.72 MFLOPS for the problem size of 10 and 12.72 GFLOPS for the problem size of 100, so that there is some dependency here.

For now, let’s do a quick investigation of the MPI usage along the lines mentioned. If we collect the Intel MPI built-in statistics, we get the output seen in Listing 5-13:

Listing 5-13. MiniFE Statistics (Workstation, Size 500, 16 MPI Processes)

```
################################################################################
#
# command : ./miniFE.x (completed)
# host    : book/x86_64_Linux               mpi_tasks : 16 on 1 nodes
# start   : 05/27/14/17:21:30               wallclock : 185.912397 sec
# stop    : 05/27/14/17:24:35               %comm     : 7.34
# gbytes  : 0.00000e+00 total               gflop/sec : NA
#
```

```
###############################################################################
# region  : * [ntasks] = 16
#
#                          [total]       <avg>         min           max
# entries                  16            1             1             1
# wallclock                2974.58       185.911       185.91        185.912
# user                     3402.7        212.668       211.283       213.969
# system                   20.6389       1.28993       0.977852      1.56376
# mpi                      218.361       13.6475       4.97802       20.179
# %comm                                  7.3409        2.67765       10.8541
# gflop/sec                NA            NA            NA            NA
# gbytes                   0             0             0             0
#
#
#                            [time]        [calls]       <%mpi>
<%wall>
# MPI_Allreduce              212.649       6512          97.38         7.15
# MPI_Send                   2.89075       29376         1.32          0.10
# MPI_Init                   1.81538       16            0.83          0.06
# MPI_Wait                   0.686448      29376         0.31          0.02
# MPI_Allgather              0.269436      48            0.12          0.01
# MPI_Irecv                  0.0444376     29376         0.02          0.00
# MPI_Comm_size              0.00278258    3360          0.00          0.00
# MPI_Bcast                  0.00242162    32            0.00          0.00
# MPI_Comm_rank              1.62125e-05   176           0.00          0.00
# MPI_Finalize               5.24521e-06   16            0.00          0.00
# MPI_TOTAL                  218.361       98288         100.00        7.34
###############################################################################
```

This is positively interesting. One MPI operation—MPI_Allreduce–is taking almost 97.5 percent of the total MPI time that in turn accounts for from 2.67 to 10.85 percent of the overall application time. Do you feel the almost irresistible temptation to start playing with the MPI_Allreduce tuning settings right away? Be cool. We will show soon enough how wrong it would be to succumb to the tempation (to be continued).

Choosing Representative Workload(s)

Workload size—or more generally, the computational and memory load created by the workload—may dramatically affect application characteristics. To understand this, you have only to recall the memory hierarchy and the attending latencies mentioned at the beginning of Chapter 3.

You can imagine that at the very beginning, at a very low memory load, the application will be basically starving for data; the computations will consume all the data in the highest level of cache where it fits, and they will stop before it can achieve full computational performance. Moreover, interprocess and interthread synchronization will probably play a more noticeable role here than at higher memory loads. Finally, the program may even break if each of its computational units, be they processes or threads, receives less than the minimum amount of data that it can sensibly handle.

As the memory load continues to grow, the workload will start occupying the LLC pretty regularly. This is where you are likely to observe the maximum possible computational performance of a particular computational node. This point in the performance curve is very important because it may show to what degree the overall problem needs to be split into smaller parts, so that those parts can be computed with maximum efficiency by separate cluster nodes.

Further growth of the memory load will lead to part of the data spilling over into the main system memory. At this point the application may become memory bound, unless clever techniques or the built-in facilities of the platform, like prefetching, are able to alleviate the detrimental effects of the spilling.

Eventually, when the size of the workload exceeds the size of the physical memory available to the current process's working set, the virtual memory mechanism of the operating system will kick in and, depending on its quality and the speed of the offline storage (like hard disk drives [HDD] or solid state disks [SSD]), this may depress performance further.

Finally, the growing memory load will cause a job to exceed the limits of the virtual memory subsystem, and the job will start extensively swapping data in and out of the main memory. This effect is called *thrashing*. The program will effectively become strongly I/O bound. At this point, unless the program was designed to handle offload data gracefully (like so many off-core solvers of yore), all bets are off.

Another, no less important aspect of the workload selection is the choice of the typical target problem class that the benchmarking will address. For example, if the target application is intended for computing—as in car-to-car collisions—it may not make much sense to benchmark it on a test case that leads to no contact and deformation of the objects involved.

Benchmarking and a bit of back-of-the-envelope calculations can help in choosing the right *workload size*. Only your experience and knowledge of the standards, traditions, and expectations of the target area are going to help you to choose the right *workload class*. Fortunately, both selections are more often than not resolved by the clients, who tell you upfront what they are interested in.

Example 2 (cont.): MiniFE Performance Investigation

We ruffled through the selection of the workload in this example earlier and settled on the workload size of 500 after very few simple runs. Let's revisit this choice now that we know the program may have noticeable MPI optimization potential.

We know from the earlier attempts that the size of 10 is so low as to lead to some 10 MFLOPS. This is a clear indication of the problem size leading to the data starvation mentioned earlier. The size of 100 achieves some 12 GFLOPS, which is not so far from the 13 GFLOPS we can observe on the size of 500. Unfortunately, the size of 1000 is apparently too high, and the system protects itself by killing off the offending job.

What we should try to gauge now is how low we can go before we see data starvation, and how high we can go before we exhaust the system memory to the point of activating its self-protection instincts. Given the points of 100 and 500, as well as the desire to do as few experiments as possible, we find that four extra data points appear warranted, namely 50, 250, 375, and 750. If we do these extra measurements in the 16 MPI processes, three thread configurations used so far, we can add the new data to the data already obtained, and thus save a bit of time.

Table 5-3 shows what we get once we drive all the data together:

Table 5-3. *MiniFE Dependency on Problem Size (Workstation, 16 MPI Processes)*

Size	CG (GLOPFS)	Total Time (seconds)	Memory (GB)
10	36.052	0.145	0.00034
50	6578.06	0.35	0.039
100	12738.8	1.5	0.31
250	13213.7	22.9	4.9
375	13225.6	78.1	16.4
500	13335.9	187.6	38.7

Recalling the characteristics of the workstation at hand, we can deduce that the problem size of 250 is probably the last one to fit into the physical memory, although the virtual memory mechanism will kick in anyway long before that. It does not look as if the size of 500 was overloading the system unduly, so we can safely keep using it.

Being a proper benchmark, this program outputs a lot of useful data that can be analyzed graphically with relative ease. For example, Figure 5-3 illustrates the absolute contribution of various stages of the computation to the total execution time.

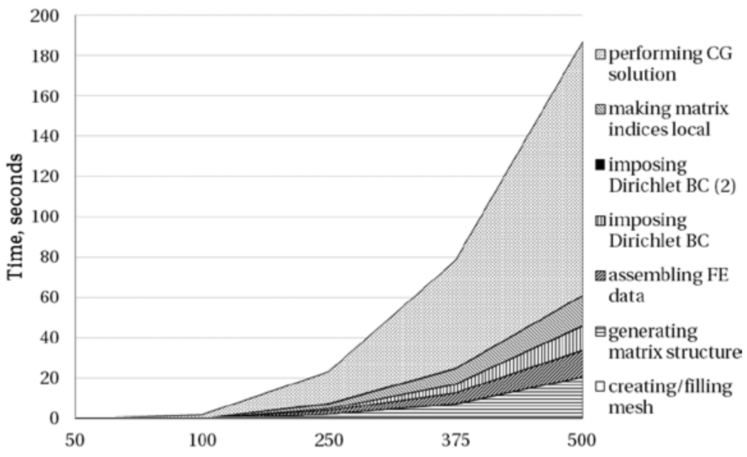


Figure 5-3. *MiniFE stage cummulative timing dependency on the problem size (16 MPI processes)*

These curves look like some power dependency to the trained eye, and this is what they should look like, given that the total number of mesh nodes grows as a cube of the problem size, while the number of the nonzero matrix elements grows as a quadrat of the problem size owing to the two-dimensional nature of the finite element interaction. This, however, is only a speculation until you can prove it (to be continued).

Balancing Process and Thread Parallelism

We started with the process/thread combination that looked reasonable for the earlier benchmarks, namely 16 MPI processes, each of them running three OpenMP threads if the application so desires. It is not clear, however, whether this is the optimum we are after. Threads have a lower context switch overhead and can use shared memory and synchronization primitives over it directly. It is not impossible that they may enjoy a slight performance advantage over the MPI processes owing to these features, at least as long as the number of threads per process is relatively low. On the other hand, the complexity of a hybrid program may actually lead to the threading adding extra overhead that detrimentally affects the overall program performance. About the only way to find out what is happening to a particular application is—you have guessed it—benchmarking.

Example 2 (cont.): MiniFE Performance Investigation

Let's do a couple of experiments to make sure that we strike the right balance between the processes and the threads. Given the total of 48 virtual cores per node, we can reasonably start not only 16 MPI processes of three OpenMP threads each but also 24 MPI processes with two OpenMP threads, and 12 MPI processes with four OpenMP threads each, and so on, up to the extreme combinations of 48 MPI processes or 48 threads that will still occupy all available computational units. Here is the required run string that needs to be changed according to the derivation for other process/thread ratios:

```
$ mpirun -genv OMP_NUM_THREADS 3 -np 16 ./miniFE.x nx=500
```

This method of inline definition of the environment variables is normally preferable because you cannot accidentally leave any of them behind, which, if that happened, could inadvertently spoil the future measurement series.

Doing our usual three attempts each time, we get the results shown in Tables 5-4 and 5-5:

Table 5-4. *MiniFE CG Performance Dependency on the Process to Thread Ratio (GFLOPS, Size 500, Workstation)*

MPI proc.	OpenMP thr.	Run 1	Run 2	Run 3	Mean	Std. dev, %
12	4	13.24	13.24	13.21	13.23	0.13
16	3	13.27	13.26	13.26	13.26	0.02
24	2	13.26	13.25	13.26	13.26	0.04

Table 5-5. *MiniFE Total Time Dependency on the Process to Thread Ratio (Seconds, Size 500, Workstation)*

MPI proc.	OpenMP thr.	Run 1	Run 2	Run 3	Mean	Std. dev, %
12	4	210.39	210.93	210.58	210.64	0.13
16	3	187.77	186.39	185.94	186.70	0.51
24	2	174.82	175.19	174.55	174.85	0.18

Although computational performance of the main CG block is equal for 16 and 24 processes, the overall time for 24 processes is lower. This is significant because the benchmark tries to approximate the behavior of a complete finite element application, and the total execution time is a more pertinent metrics here. We will focus on this wall-clock metric while keeping in mind that we do want to use the processor as efficiently as possible during the main computational step (to be continued).

Doing a Scalability Review

If you recall the treatise in Chapter 2, there are two major types of scalability: weak and strong. Weak scalability series increases the load proportionally to the number of processes involved. In other words, it keeps the per-node load constant and seeks to investigate the effect of the growing number of connections between the nodes. Strong scalability keeps the problem size constant while increasing the number of processes involved. This is what we are interested in primarily now. Just as in the case of intranode communication, here we want to see where the problem starts loading the machine so much as to make further increase in the computational resources allocated pointless or even counterproductive.

Example 2 (cont.): MiniFE Performance Investigation

First, let's look into strong scalability of the miniFE. We will put up to 48 MPI processes on one node and let the runtime decide how many threads to start. Further, we will try to load the nodes so that we get into the memory-bound state from the very beginning, and gradually move toward the compute-bound situation, looking for the knee of the graph. We will also incidentally check whether explicit setting of the OpenMP thread number is indeed helping instead of hurting performance. After a series of respective runs without the `OMP_NUM_THREADS` variable set, we get the data shown in Tables 5-6 and 5-7:

Table 5-6. *MiniFE CG Performance Dependency on the Process Number (GFLOPS, Size 500, Workstation, No OpenMP Threads)*

MPI proc.	OpenMP thr.	Run 1	Run 2	Run 3	Mean	Std. dev, %
8	undefined	11.95	11.83	12.04	11.94	0.90
12	undefined	13.02	13.00	13.00	13.00	0.11
16	undefined	13.32	13.32	13.32	13.32	0.01
24	undefined	13.36	13.36	13.36	13.36	0.03
48	undefined	13.19	13.20	13.20	13.19	0.04

Table 5-7. *MiniFE CG Total Time Dependency on the Process Number (Seconds, Size 500, Workstation, No OpenMP Threads)*

MPI proc.	OpenMP thr.	Run 1	Run 2	Run 3	Mean	Std. dev, %
8	undefined	257.41	254.92	257.63	256.65	0.59
12	undefined	212.77	212.80	212.28	212.61	0.14
16	undefined	185.80	185.72	185.43	185.76	0.03
24	undefined	173.37	173.71	173.65	173.58	0.11
48	undefined	160.92	160.91	160.69	160.84	0.08

By setting the environment variable `KMF_AFFINITY` to `verbose`, you can verify that more than one OpenMP thread is started even if its number is not specified.
It is interesting that we get about 100 MFLOPS extra by not setting the OpenMP thread number explicitly, and that the total time drops still further if all 48 cores are each running MPI process Moreover, it drops between the process counts by as much as 16 and 24. This indicates that the application has substantial scaling potential in this strong scaling scenario.

The tendency toward performance growth with the number of MPI processes suggests that it might be interesting to see what happens if we use only the physical cores. Employing the recipe described earlier, we get 13.38 GFLOPS on 24 MPI processes put on the physical cores, taking 176.47 seconds for the whole job versus 173.58 for 24 MPI processes placed by default. So, there is no big and apparent benefit in using the physical cores explicitly.

Thus, we are faced with the question of what configuration is most appropriate for the following investigation. From Tables 5-4 through 5-7, it looks like 16 MPI processes running three threads each combine reasonable overall runtime, high CG block performance, and potential for further tweaks at the OpenMP level. One possible issue in the 16 MPI process, three OpenMP thread configuration is that every second core will contain parts of two different MPI processes, which may detrimentally affect the caching. Keeping this in mind, we will focus on this configuration from now on, and count on the 24 process, two thread configuration as plan B.

We could continue the scalability review by proceeding from the workstation to the cluster. In particular, the speedup *S(p)* and efficiency *E(p)* graphs can be used to track the expected and observed performance at different MPI process and OpenMP thread counts. For an ideal scaling program, *S(p) = p* and *E(p) = 1*, so it will be easy to detect any deviation visually. This investigation can be done for the overall program execution time, which can be measured directly. It can also be done for the time used up by its components, be that computation versus communication, or particular function calls, or even code fragments. This more detailed information can be discovered by directly embedding the timing calls, like `MPI_Wtime`, into the program code; looking into the statistics output we have seen before; or using one of the advanced analysis tools described later in this book. However, the limited scope of this example does not make this investigation necessary. In any case, we have settled both the representative workload and the most promising run configurations, and this is good enough for now.

Note that there may be a certain interaction between the process: thread ratio, on one hand, and the workload size, on the other hand. So far, we have been basically ignoring this effect, hoping that we can change the respective coordinates independently or that at least this effect will be of the second order. This may or may not be true in the general case: it is conceivable that smaller workloads will lend themselves better to the higher thread counts. However, to gauge this effect, we would have to perform a full series of the measurements over all the MPI process and thread counts, as well as the problem sizes. That is, instead of probing this three-dimensional Cartesian space along two lines (the process:thread ratio at problem size of 500, and then the problem size at the process:thread ratio of 16:3), we would have to do a full search. The time required for this, as well as the amount of data produced, would probably be prohibitive for the scope of this book (to be continued).

EXERCISE 5-4

Perform a focused sampling around the point that we consider as the optimum for miniFE, to verify it is indeed at least the local maximum of performance we are after. Replace miniFE by your favorite application and repeat the investigation. If intranode scalability results warrant this, go beyond one node.

Analyzing the Details of the Application Behavior

There are many ways to analyze the behavior of parallel applications. That is, they differ in the way in which the data is collected. Three of them are most frequently used:

- *Printing* uses timestamp collection and output statements built into the program during its development or added specifically for debugging. Surprisingly enough, this is probably still the best way to understand the overall program behavior unless you are after an issue that disappears when observed (so-called Heisenbug).
- *Sampling* takes snapshots of the system, either at fixed time intervals or at certain points of the program or system lifecycle. Information collected this way comes in the form of hardware and software counters, register values, and so on, and it normally requires a tool to make sense of it.
- *Tracing* follows program execution and tracks all important events as they occur by creating a so-called application trace. Again, tools are nearly unavoidable if a nontrivial program is to be analyzed.

People will also just go through the application in an interactive debugger, but this mode is more suitable for debugging than for performance analysis per se. In any case, there are arguments in favor of each of these methods, as well as interesting cases when they may usefully complement each other. We will see some of them later.

The Intel Trace Analyzer and Collector (ITAC) we are going to use for the distributed memory performance analysis in this book is a tracing tool, one of many that can produce and visualize application trace files in various forms. Instead of trying to describe this very powerful program in general terms, we propose to simply use it for the example at hand.

Example 2 (cont.): MiniFE Performance Investigation

You can use ITAC to generate an application trace file and to inspect it visually. You enter the following commands to get the trace file `miniFE.x.stf`:

```
$ source /opt/intel/itac_latest/bin/itacvars.sh
$ mpirun -trace -np 16 ./miniFE.x -nx=500
```

The first command establishes the necessary environment. As usual, we added this command to the script `0env.sh` included in the respective example archive, so if you have sourced that file already, you do not need to source the specific ITAC environment script. The `-trace` option in the `mpirun` invocation instructs the Intel MPI library link the executable at runtime against the ITAC dynamic library, which in turn creates the requested trace file. Application rebuilding is not required.

If you work on a cluster or another remote computer, you will have to ship all the files associated with the main trace file `miniFE.x.stf` (most of them are covered by the file mask `miniFE.x.stf*`) to a computer where you have the ITAC installed. To make this process a little easier, you can ask ITAC to produce a single trace file if you use the following command instead of the earlier `mpirun` invocation:

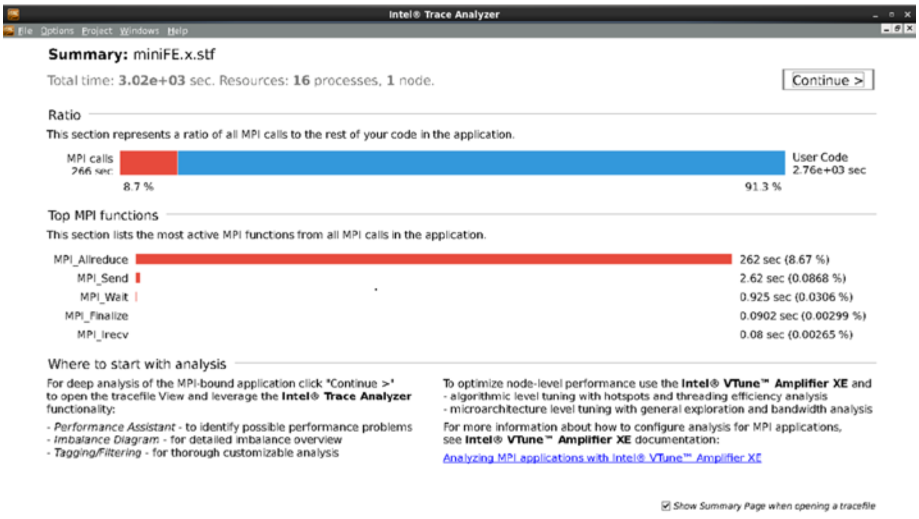
```
$ mpirun -trace -np 16 ./miniFE.x -nx=500 --itc-args --logfile-format
SINGLESTF --itc-args-end
```

You can learn more about the ways to control ITAC runtime configuration in the product online documentation.[12]

Now you can run the ITAC:

```
$ traceanalyzer miniFE.x.stf
```

This way or another, after a few splash screens, the ITAC summary chart shows up (see Figure 5-4; note that we maximized the respective view inside the ITAC window).



Summary: miniFE.x.stf

Total time: **3.02e+03** sec. Resources: **16** processes, **1** node.

Ratio

This section represents a ratio of all MPI calls to the rest of your code in the application.

MPI calls 266 sec 8.7 %
User Code 2.76e+03 sec 91.3 %

Top MPI functions

This section lists the most active MPI functions from all MPI calls in the application.

MPI_Allreduce	262 sec (8.67 %)
MPI_Send	2.62 sec (0.0868 %)
MPI_Wait	0.925 sec (0.0306 %)
MPI_Finalize	0.0902 sec (0.00299 %)
MPI_Irecv	0.08 sec (0.00265 %)

Where to start with analysis

For deep analysis of the MPI-bound application click 'Continue >' to open the tracefile View and leverage the **Intel® Trace Analyzer** functionality:

- *Performance Assistant* - to identify possible performance problems
- *Imbalance Diagram* - for detailed imbalance overview
- *Tagging/Filtering* - for thorough customizable analysis

To optimize node-level performance use the **Intel® VTune™ Amplifier XE** and
- algorithmic level tuning with hotspots and threading efficiency analysis
- microarchitecture level tuning with general exploration and bandwidth analysis

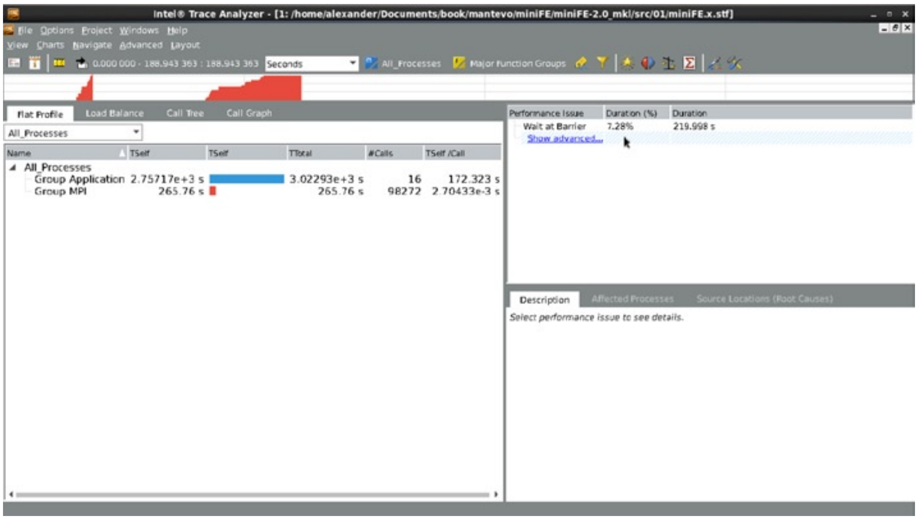
For more information about how to configure analysis for MPI applications, see **Intel® VTune™ Amplifier XE** documentation:

Analyzing MPI applications with Intel® VTune™ Amplifier XE

✓ *Show Summary Page when opening a tracefile*

Figure 5-4. *MiniFE trace file in ITAC summary chart (Workstation, 16 MPI processes)*

This view basically confirms what we already know from the built-in statistics output. Press the *Continue* button at the upper right corner, and you will see the default ITAC screen that includes the function profile (left) and the performance assistant (right), with the rest of the screen occupied by the main program and view menus (very top), as well as the handy icons and the schematic timeline (below the top); see Figure 5-5, and note that we maximized the respective window once again.

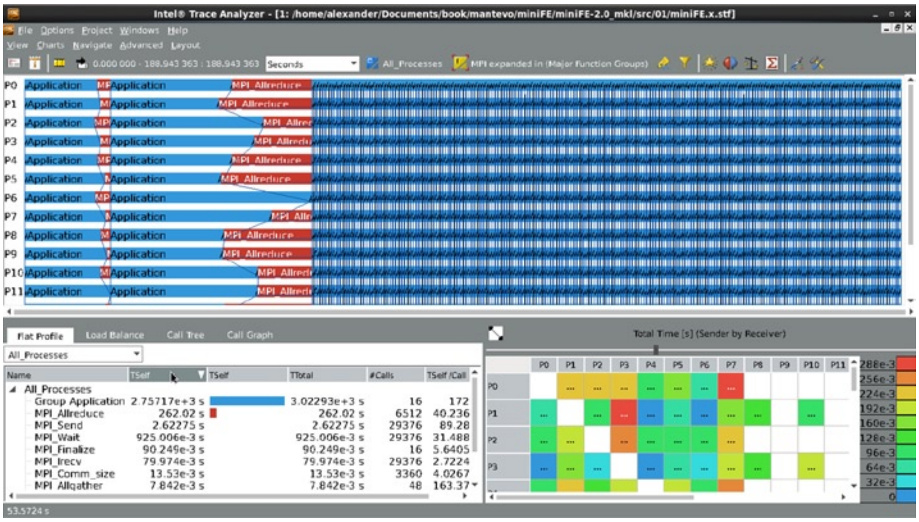

Intel® Trace Analyzer - [1: /home/alexander/Documents/book/mantevo/miniFE/miniFE-2.0_mkl/src/01/miniFE.x.stf]
File Options Project Windows Help
View Charts Navigate Advanced Layout
0.000 000 - 188.943 363 : 188.943 363
Seconds
All_Processes
Major Function Groups
Flat Profile
Load Balance
Call Tree
Call Graph
All_Processes
Description
Affected Processes
Source Locations (Root Causes)
Select performance issue to see details.

Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
All_Processes					
Group Application	2.75717e+3 s		3.02293e+3 s	16	172.323 s
Group MPI	265.76 s		265.76 s	98272	2.70433e-3 s

Performance Issue	Duration (%)	Duration
Wait at Barrier	7.28%	219.998 s
Show advanced...		

Figure 5-5. *MiniFE trace file in ITAC default view (Workstation, 16 MPI processes)*

The function profile is basically a reiteration of the statistics output at the moment, while the performance assistant is pointing out an issue we may want to deal with when we have performed the initial trace file review. To that end, let us restore the historical ITAC trace file view. Go to the *Charts* item in the main chart menu and select the *Event Timeline* item there. This chart will occupy the top of the screen. Again in the main view menu, deselect the *Performance Assistant* item, then select the *Message Profile* item. Also, hide the schematic timeline by right-clicking it and selecting the *Hide* item in the popup menu. This will display the historical ITAC analysis view; see Figure 5-6.

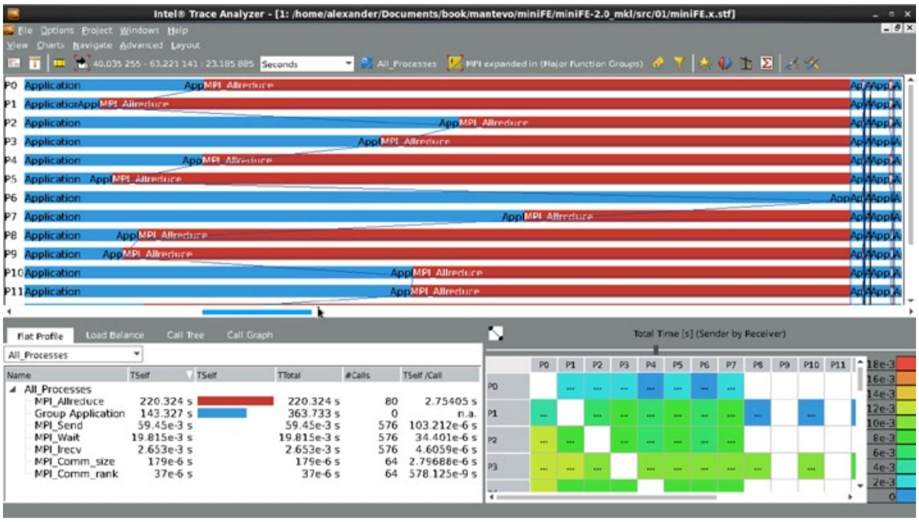

Intel® Trace Analyzer - [1: /home/alexander/Documents/book/mantevo/miniFE/miniFE-2.0_mkl/src/01/miniFE.x.stf]
File Options Project Windows Help
View Charts Navigate Advanced Layout
0.000 000 - 188.943 363 : 188.943 363
Seconds
All_Processes
MPI expanded in (Major Function Groups)
P0
P1
P2
P3
P4
P5
P6
P7
P8
P9
P10
P11
Application
MPI_Allreduce
Flat Profile
Load Balance
Call Tree
Call Graph
All_Processes
Total Time [s] (Sender by Receiver)
288e-3
256e-3
224e-3
192e-3
160e-3
128e-3
96e-3
64e-3
32e-3
0
53.5724 s

Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
All_Processes					
Group Application	2.75717e+3 s		3.02293e+3 s	16	172
MPI_Allreduce	262.02 s		262.02 s	6512	40.236
MPI_Send	2.62275 s		2.62275 s	29376	89.28
MPI_Wait	925.006e-3 s		925.006e-3 s	29376	31.488
MPI_Finalize	90.249e-3 s		90.249e-3 s	16	5.6405
MPI_Irecv	79.974e-3 s		79.974e-3 s	29376	2.7224
MPI_Comm_size	13.53e-3 s		13.53e-3 s	3360	4.0267
MPI_Allgather	7.842e-3 s		7.842e-3 s	48	163.37

Figure 5-6. *MiniFE trace file in ITAC historical view (Workstation, 16 MPI processes)*

Nothing can beat this view in the eyes of an experienced analyst. The event timeline shows pretty clearly that the program is busy computing and communicating most of the time after setup. However, during setup there are substantial issues concerning one of the `MPI_Allreduce` calls that may need to be addressed. The message profile illustrates the neighbor exchanges between the adjacent processes that possess the respective adjacent slabs of the overall computation domain. These relatively short exchanges still differ in duration by approximately four times. To make sure this is indeed the case, you can scroll this view up and down using the scrollbar on the right. If you right-click on the *Group MPI* in the function profile, and select *Ungroup MPI* in the popup menu, this will show how MPI time is split between the calls. Again, this information is known to you from the built-in statistics. Some scrolling may be required here as well, depending on the size of your display. Alternatively, click on any column header (like *TSelf*) to sort the list.

Now, zoom in on a piece of the event timeline around the offending `MPI_Allreduce;` move the mouse cursor where you see fit, hold and drag to highlight the selected rectangle, and release to see the result. All charts will automatically adjust themselves to the selected time range (see Figure 5-7).


Intel® Trace Analyzer - [1: /home/alexander/Documents/book/mantevo/miniFE/miniFE-2.0_mkl/src/01/miniFE.x.stf]
File Options Project Windows Help
View Charts Navigate Advanced Layout
40.035 255 - 63.221 141 : 23.185 885
Seconds
All_Processes
MPI expanded in (Major Function Groups)
Application
MPI_Allreduce
Flat Profile
Load Balance
Call Tree
Call Graph
Total Time [s] (Sender by Receiver)

All_Processes

Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
All_Processes					
MPI_Allreduce	220.324 s		220.324 s	80	2.75405 s
Group Application	143.327 s		363.733 s	0	n.a.
MPI_Send	59.45e-3 s		59.45e-3 s	576	103.212e-6 s
MPI_Wait	19.815e-3 s		19.815e-3 s	576	34.401e-6 s
MPI_Irecv	2.653e-3 s		2.653e-3 s	576	4.6059e-6 s
MPI_Comm_size	179e-6 s		179e-6 s	64	2.79688e-6 s
MPI_Comm_rank	37e-6 s		37e-6 s	64	578.125e-9 s

Figure 5-7. *MiniFE trace file in ITAC zoomed in upon the offending MPI_Allreduce operation (Workstation, 16 MPI processes)*

Well, this is exactly what we need to see if we want to understand the ostensibly main MPI-related performance issue in this program. The updated Functional Profile chart confirms that it is indeed this `MPI_Allreduce` operation that takes the lion's share of MPI communication time. On the other hand, the time spent for the actual data exchange is very low, as can be seen in the Message Profile chart, so the volume of communication cannot be the reason for the huge overhead observed. Therefore, we must assume this is load imbalance. Let us take this as a working hypothesis (to be continued).

EXERCISE 5-5

Analyze the behavior of your favorite application using the process described here. What operations consume most of the MPI communication time? Is this really communication time or load imbalance?

Choosing the Optimization Objective

If you recall, at the very beginning of this chapter we faced the decision as to what to address: load imbalance or MPI performance. The criteria for selecting one over the other are relatively soft, for the situation in real-life programs is rarely black and white. Normally there is some degree of load imbalance and some degree of MPI sloppiness. If one of them clearly dominates the other, the choice may seem obvious. However, you need to keep in mind that even a relatively small load imbalance may jog some collective operations off tune; alternatively, suboptimal performance of an MPI operation may lead to something that looks like load imbalance. How does one lighten this gray area?

Detecting Load Imbalance

Fortunately, there is a sure way to detect load imbalance in a distributed memory program. Imagine that you take out all the communication costs, essentially presuming that you run over an ideal communication fabric that has zero latency and infinite bandwidth. It is clear that, in this case, you cannot blame the network for any undue delays left in the program. Whatever is left behind is, then, the program's own fault rather than the network's or MPI's. This is why an advanced analysis tool like ITAC offers both an Ideal Interconnect Simulator (IIS) and a Load Imbalance Diagram that are broadly based on this idea and therefore help to pinpoint the load imbalance and its main victims.

Example 2 (cont.): MiniFE Performance Investigation

Let us get back to the miniFE application example, in which we came to the preliminary suspicion that load imbalance might be to blame for the extraordinarily bad observed behavior of one particular `MPI_Allreduce` operation.

There is a very good way to check out our working hypothesis. Go to the *Advanced* tab in the main menu, select the *Idealize* command and click OK in the respective popup to generate the ideal trace file. Now, open the ideal trace file using the *File* control in the main ITAC window, click on the *Advanced* tab, and select the *Imbalance Diagram* item. Then click *OK* in the resulting popup. Upon some meditation reflected by a progress bar, the program will visually confirm the initial suspicion: all of the MPI communication seems to be covered by load imbalance. Figure 5-8 shows that (note that we changed the default colors to make the difference more visible).

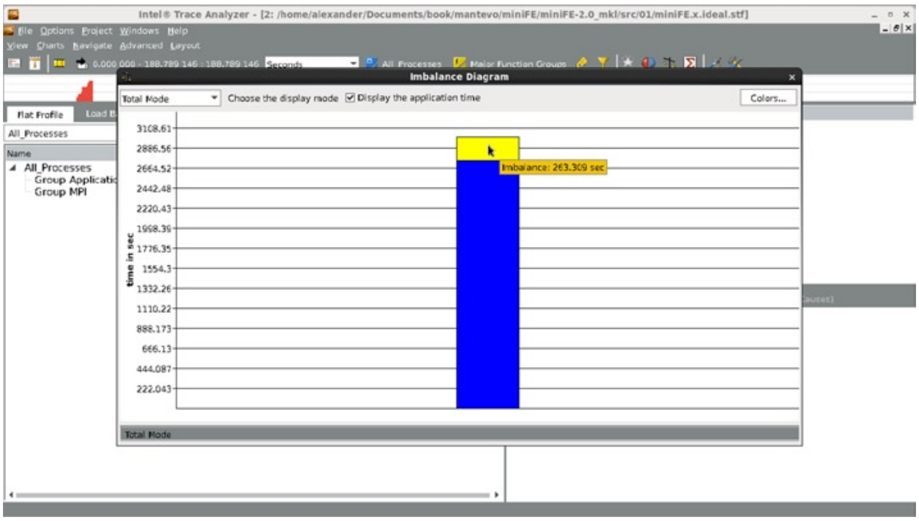


Figure 5-8. *MiniFE trace file in ITAC imbalance diagram (C version, Workstation, 16 MPI processes, 3 OpenMP threads)*

Of course, this may be an artifact of the model used to compute the load imbalance. However, this certainly indicates we should look into the load imbalance first, and only then look into further MPI communication details. Depending on this, we will decide where to go.

Referring back to the Performance Assistant chart (see Figure 5-5), we can conclude that the `MPI_Wait` issue most likely related to the internal workings of the `MPI_Allreduce` operation that might issue a call to the `MPI_Wait` behind the curtain. However, taken at face value, this indication itself is somewhat misleading until we understand what stands behind the reported issue. Indeed, if you switch to the *Breakdown Mode* in the view of Figure 5-8 (not shown), you will see that small message performance of the `MPI_Wait` call is the sole major contributor of the load imbalance observed (to be continued).

EXERCISE 5-6

Choose the primary optimization objective for your favorite application using the method described in this section. Is this load imbalance or MPI tuning? How can you justify your choice?

Dealing with Load Imbalance

Once the decision is made to address the load imbalance, it is necessary to understand what causes it and then devise an appropriate cure.

Classifying Load Imbalance

Load imbalance can come from different quarters. The first is from the *application itself*: its data layout, algorithms, and implementation quality. If the application developer did not think hard about dividing the data among the job components—be they processes or threads or tasks—fairly and according to their capabilities, there will be no other way to attack load imbalance than to fix the respective data layout, algorithmic, and implementation issues of the application.

This is where the second major source of load imbalance pops up: *platform heterogeneity*, especially heterogeneity that was not taken into account when the application was conceived. A typical example is the use of different processors across the machine, be they different CPUs or various accelerators. Another example of heterogeneity is the difference in communication characteristics of the underlying platform. Even the difference between shared memory, on the one hand, and fast network, on the other hand, unless properly accounted for, may lead to part of the job's lagging behind, waiting for the necessary data to come over the slower link.

These dependencies may or may not be explicit. It is relatively clear what is happening when two MPI processes send data to each other in the point-to-point fashion. As soon as any collective operation is involved, the choice of communication pattern is delegated to the MPI library, and ultimately, to the MPI implementor who created this library. In that case, it may be necessary to understand exactly what algorithm is being used, especially if there are more than two processes involved.

However, the situation may be substantially less transparent. Many libraries and language extensions (like offload) try to hide the actual data movement from the application programmer. In that case, it may be necessary to understand what exactly is happening beneath the hood, up to and including monitoring the activities of the underlying software and hardware components, or at least talking to someone in the know.

Addressing Load Imbalance

The treatment for load imbalances is basically determined by their source of the issue and the amount of time available.

Data partitioning and algorithmic issues may be the hardest to address, unless the program already possesses mechanisms that provide relatively easy control over these parameters. In some cases, the amount of data apportioned to each computational unit can be defined by the program input file. In other cases, the amount of work (rather than only data) apportioned to a program component may depend on its role. For example, if boundary conditions are involved, corner segments will have only two neighbors in the two-dimensional case, while internal segments will have four, and so on. If data or work partitioning is implicated in the load imbalance, a deep dive into the program may be required, up to and including reformulation of the data layout or work-partitioning strategy; replacement of the data-partitioning algorithm or component; selection of a more advanced, easier to parallelize algorithm; and so on. There is little by the way of general advice that can be given here.

Platform heterogeneity may pop up everywhere. In a modern heterogeneous cluster that uses Intel Xeon CPUs and Intel Xeon Phi coprocessors connected to the main processors by the PCI Express bus, with a fast network like InfiniBand connecting the nodes, there are so many way to get things wrong that, most likely, only proper application design upfront can “guarantee” success of the undertaking. Indeed, in such a cluster, you will have several effects uniting their forces to cause trouble:

1. Differences in the clock rate and functionality of the processors involved. These differences may go up to several times, especially as far as the clock rate is concerned. You will have to allocate proportionally less data to the weaker components.
2. Differences between the intranode communication over the shared memory and over the PCI Express bus. Again, the latency and bandwidth will vary greatly, and sometimes the relationship will not be linear. For example, PCI Express will normally lose to the shared memory on latency but may overtake it on bandwidth on certain message sizes, depending on the way in which the bus is programmed.
3. Differences between the intranode communication of any kind, on one hand, and internode communication over the fast network, on the other. In addition to this normal situation typical of any cluster, in a heterogeneous cluster with accelerated nodes, there may be the need to tunnel data from accelerator to accelerator via the PCI Express bus, over the network, and then over the PCI Express bus on the other side.

Of course, a properly implemented MPI library will handle all of this transfer transparently to your benefit, but you may still see big differences in the performance of the various communication links involved, and you will have to take this into account when partitioning the data.

On top of this, there is an interesting interaction between the component’s computing capacity and its ability to push data to other components. Because of this interaction, in an ideal situation, it is possible that a relatively slower component sitting on a relatively slower interface may be loaded 100 percent of the time and cause no trouble across the job, provided the relatively faster components get larger pieces of data to deal with and direct the bulk data exchanges to the fastest available communication paths. However, it may be difficult to arrive at this ideal situation. This consideration applies, of course, to both explicit and implicit data-movement mechanisms.

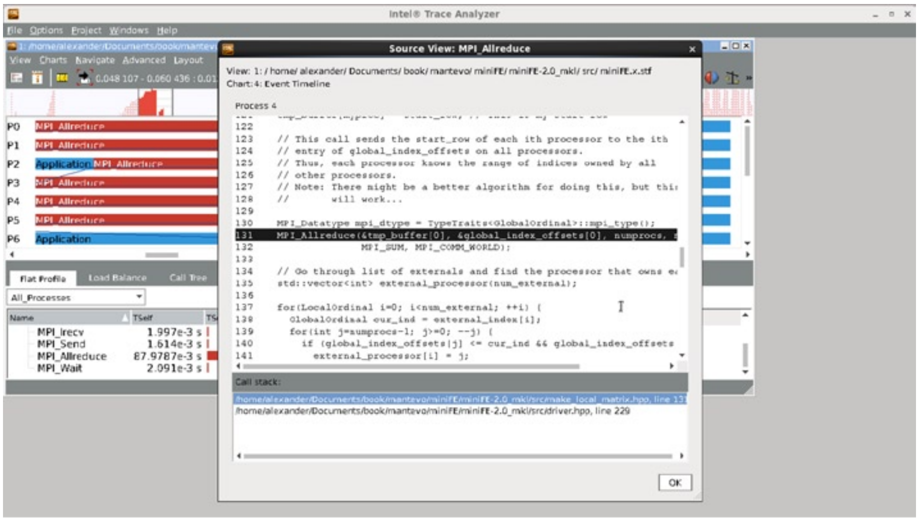
Example 2 (cont.): MiniFE Performance Investigation

To complete the miniFE investigation at the MPI level, we need to understand what is causing the load imbalance detected earlier. Once again, ITAC is going to be of great help in finding that out.

First, rebuild your application so that the compiler adds debugging information to the files it produces. Adding the `-g` flag to the `CFLAGS` variable in the miniFE `src/Makefile`, doing `make clean` there, and then `make` does the trick.

Now, set the environment variable VT_PCTRACE to 5 and rerun the miniFE, asking for the trace file to be produced. (You know how to do this.) Note, however, that call stack tracing requested this time is a relatively expensive procedure that will slow the execution, so it may make sense to take a rather low problem size, hoping that the program execution path does not depend on it. We used the size of 50.

Open the resulting file miniFE.x.stf in the ITAC, go to the offending MPI_Allreduce operation in the event timeline, right-click on it, and ask for details. When you click on the *View Source Code* item in the resulting popup window, you will see where the offending MPI_Allreduce was called from (see Figure 5-9).



Name	TSelf
MPI_Irecv	1.997e-3 s
MPI_Send	1.614e-3 s
MPI_Allreduce	87.9787e-3 s
MPI_Wait	2.091e-3 s

View: 1: / home/ alexander/ Documents/ book/ mantevo/ miniFE/ miniFE-2.0_mkl/ src/ miniFE.x.stf
Chart: 4: Event Timeline

Process 4

```

  // This call sends the start_row of each ith processor to the ith
  // entry of global_index_offsets on all processors.
  // Thus, each processor knows the range of indices owned by all
  // other processors.
  // Note: There might be a better algorithm for doing this, but thi:
  //       will work...

  MPI_Datatype mpi_dtype = TypeTraits<GlobalOrdinal>::mpi_type();
  MPI_Allreduce(&tmp_buffer[0], &global_index_offsets[0], numprocs,
                MPI_SUM, MPI_COMM_WORLD);

  // Go through list of externals and find the processor that owns e
  std::vector<int> external_processor(num_external);

  for(LocalOrdinal i=0; i<num_external; ++i) {
    GlobalOrdinal cur_ind = external_index[i];
    for(int j=numprocs-1; j>=0; --j) {
      if (global_index_offsets[j] <= cur_ind && global_index_offsets
        external_processor[i] = j;
```

Call stack:

/home/alexander/Documents/book/mantevo/miniFE/miniFE-2.0_mkl/src/make_local_matrix.hpp, line 131
/home/alexander/Documents/book/mantevo/miniFE/miniFE-2.0_mkl/src/driver.hpp, line 229

Figure 5-9. *Finding MPI_Allreduce source code location in miniFE (Workstation)*

If you browse the source code in this window, you will see that immediately prior to this MPI_Allreduce call, the program imposes Dirichlet boundary conditions. Very likely, the imbalance is coming from that piece of code. This is only a guess for now, so you will have to do more work before you can be sure of having found the culprit. However, if this guess is correct, and given that the program itself reports very low data imbalance as far as the distribution of nonzero matrix elements across the processes is concerned, it looks like an algorithmic issue. If you want to address it now, you know where to start. Later on, we will cover advanced techniques that will allow you to pinpoint the exact problematic code location in any situation, not only in presence of the conveniently placed and easily identifiable MPI operational brackets (to be continued in Chapter 6).

EXERCISE 5-7

Narrow down the search area by recalling from Figure 5-6 that, prior to the problematic `MPI_Allreduce` operation, there was another `MPI_Allreduce` operation that also synchronized the processes that were almost perfectly aligned at that moment. What remains to be done is to repeat this procedure and find out the other code location. Did you find the culprit?

Example 3: MiniMD Performance Investigation

Having looked into an algorithmically induced load imbalance in the case of miniFE, we can now take time to investigate another application from the same Mantevo suite, namely miniMD. This application is reported as representing, in a very lightweight form, the core of the typical molecular dynamics application LAMMPS.[13] Another useful feature is that this application (at least in its `miniMD-Intel` reincarnation) has been ported to the Intel Xeon Phi coprocessor, which may allow us to investigate heterogeneity-induced load imbalance issues without investing any time in the porting effort.

If you repeat all the steps mentioned here with the miniMD application, you will learn that this application shows admirable scalability intranode (see Table 5-8).

Table 5-8. *MiniMD Execution Time Dependency on the Process Number (Seconds, Workstation)*

MPI proc.	Run 1	Run 2	Run 3	Mean	Std. dev, %
1	6.402392	6.412376	6.401814	6.405527	0.075692
2	4.146758	3.884414	3.623191	3.884788	5.502115
4	1.739194	1.839692	1.683867	1.754251	3.676788
8	0.944237	0.951552	0.91314	0.93631	1.778618
16	0.518546	0.523697	0.504854	0.515699	1.541922
24	0.367219	0.365578	0.365644	0.366147	0.207156
32	0.409625	0.407031	0.397341	0.404666	1.306382
48	0.287009	0.28772	0.277317	0.284015	1.670798

The only hitch happens around 32 processes. It is probably caused by half of the MPI processes running on eight physical cores, with the other half occupying the other 16 cores for themselves. This is effectively a heterogeneous situation. Indeed, Listing 5-14 shows what the process pinning looks like:

Listing 5-14. Default Pinning for 32 MPI Processes (Workstation)

```
[0] MPI startup(): Rank    Pid      Node name  Pin cpu
[0] MPI startup(): 0       225142   book       0
[0] MPI startup(): 1       225143   book       24
[0] MPI startup(): 2       225144   book       1
[0] MPI startup(): 3       225145   book       25
[0] MPI startup(): 4       225146   book       2
[0] MPI startup(): 5       225147   book       26
[0] MPI startup(): 6       225148   book       3
[0] MPI startup(): 7       225149   book       27
[0] MPI startup(): 8       225150   book       4
[0] MPI startup(): 9       225151   book       5
[0] MPI startup(): 10      225152   book       6
[0] MPI startup(): 11      225153   book       7
[0] MPI startup(): 12      225154   book       8
[0] MPI startup(): 13      225155   book       9
[0] MPI startup(): 14      225156   book       10
[0] MPI startup(): 15      225157   book       11
[0] MPI startup(): 16      225158   book       12
[0] MPI startup(): 17      225159   book       36
[0] MPI startup(): 18      225160   book       13
[0] MPI startup(): 19      225161   book       37
[0] MPI startup(): 20      225162   book       14
[0] MPI startup(): 21      225163   book       38
[0] MPI startup(): 22      225164   book       15
[0] MPI startup(): 23      225165   book       39
[0] MPI startup(): 24      225166   book       16
[0] MPI startup(): 25      225167   book       17
[0] MPI startup(): 26      225168   book       18
[0] MPI startup(): 27      225169   book       19
[0] MPI startup(): 28      225170   book       20
[0] MPI startup(): 29      225171   book       21
[0] MPI startup(): 30      225172   book       22
[0] MPI startup(): 31      225173   book       23
```

Comparing this to Listing 5-5, we get the distribution of the MPI processes among the virtual processors, as shown in Figure 5-10.

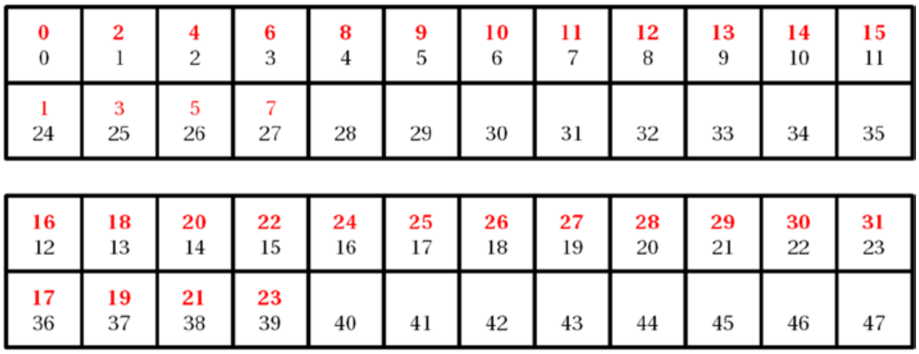


Figure 5-10. *Default process pinning (workstation, 32 MPI processes): MPI ranks (gray upper numbers) mapped upon processor identifiers (black lower numbers)*

You could argue that this may not be the fairest mapping of all, but whatever you do, you will end up with some MPI processes out of 32 running two apiece on some physical cores. This probably explains the hitch we observed in Table 5-9.

Table 5-9. *Intel MPI Library Communication Fabric Selection*

`I_MPI_DEVICE`	`I_MPI_FABRICS`	Description
`sock`	`tcp`	TCP/IP-capable network fabrics, such as Ethernet and InfiniBand (the latter through IP over IB). Normally the slowest available fabric.
`shm`	`shm`	Shared memory only. Normally the fastest available fabric, but for very large messages where fast interconnects may win intranode.
`ssm`	`shm:tcp`	Shared memory + TCP/IP. Good for multicore clusters built on Ethernet.
`rdma`	`dapl`	Direct Access Programming Library (DAPL). Good for DAPL-capable network fabrics, such as InfiniBand or iWarp.
`rdssm`	`shm:dapl`	Shared-memory + DAPL. The default and fastest choice in most cases. See above for details.
`N/A`	`ofa`	Open Fabric Association (OFA)-capable network fabric including InfiniBand.* Comparable to DAPL but with some advantages, like multirail and checkpoint/restart support.

(*continued*)

Table 5-9. *(continued)*

I_MPI_DEVICE	I_MPI_FABRICS	Description
N/A	shm:ofa	Shared memory + OFA-capable network fabric. See above for details.
N/A	tmi	Tag Matching Interface (TMI)-capable network fabric including Intel True Scalue Fabric. Includes shared memory support internally, so there is no point in using the `shm:tmi` combination.

Next, you will observe that this application suffers from noticeable load imbalance *and* MPI overhead (called "interconnect" in the imbalance diagram; see Figure 5-11).

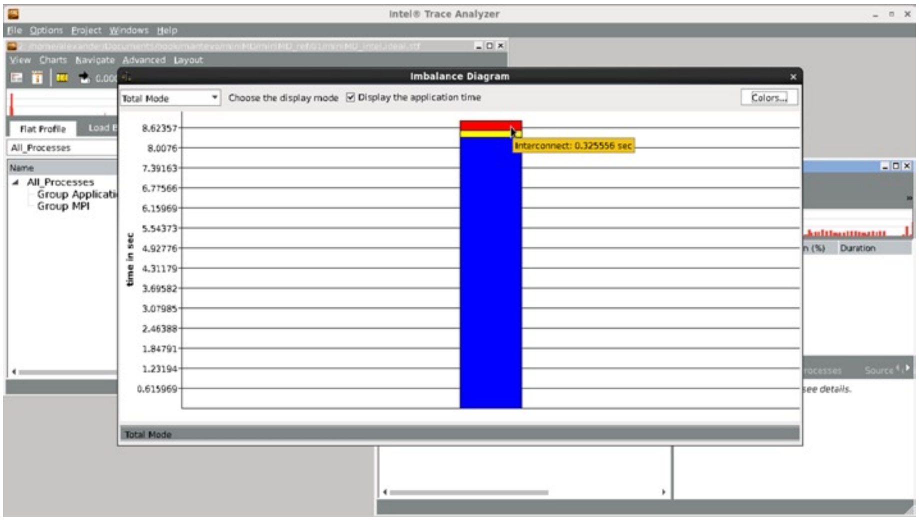
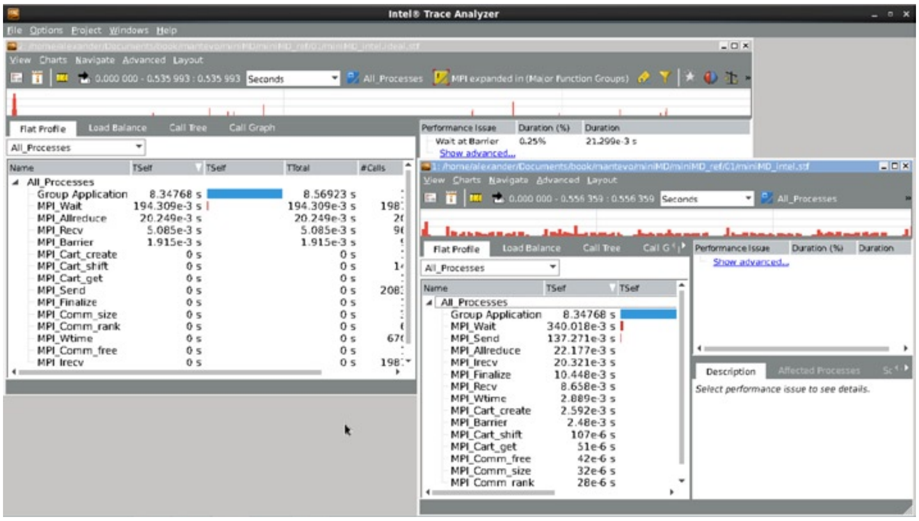


Figure 5-11. *MiniMD trace file in ITAC imbalance diagram (Workstation, 16 MPI processes)*

There is something to haul on the MPI side of the equation, at least on the default workload `in.lj.miniMD`. We can find out what exactly is contributing to this by comparing the real and ideal traces, ungrouping the *Group MPI* and sorting the list by *TSelf* (see Figure 5-12).



Performance Issue	Duration (%)	Duration
Wait at Barrier	0.25%	21.299e-3 s

Name	TSelf	TSelf	TTotal	#Calls
All_Processes				
Group Application	8.34768 s		8.56923 s	
MPI_Wait	194.309e-3 s		194.309e-3 s	198
MPI_Allreduce	20.249e-3 s		20.249e-3 s	2
MPI_Recv	5.085e-3 s		5.085e-3 s	9
MPI_Barrier	1.915e-3 s		1.915e-3 s	
MPI_Cart_create	0 s		0 s	
MPI_Cart_shift	0 s		0 s	1
MPI_Cart_get	0 s		0 s	
MPI_Send	0 s		0 s	208
MPI_Finalize	0 s		0 s	
MPI_Comm_size	0 s		0 s	
MPI_Comm_rank	0 s		0 s	
MPI_Wtime	0 s		0 s	67
MPI_Comm_free	0 s		0 s	
MPI_Irecv	0 s		0 s	198

Name	TSelf	TSelf
All_Processes		
Group Application	8.34768 s	
MPI_Wait	340.018e-3 s	
MPI_Send	137.271e-3 s	
MPI_Allreduce	22.177e-3 s	
MPI_Irecv	20.321e-3 s	
MPI_Finalize	10.448e-3 s	
MPI_Recv	8.658e-3 s	
MPI_Wtime	2.889e-3 s	
MPI_Cart_create	2.592e-3 s	
MPI_Barrier	2.48e-3 s	
MPI_Cart_shift	107e-6 s	
MPI_Cart_get	51e-6 s	
MPI_Comm_free	42e-6 s	
MPI_Comm_size	32e-6 s	
MPI_Comm_rank	28e-6 s	

Figure 5-12. *MiniMD ideal and real traces compared (Workstation, 16 MPI processes)*

Compare the ideal trace in the upper left corner with the real trace in the lower right corner. The biggest part of the improvement comes from halving the time spent in the `MPI_Wait`. Most of the remaining improvement can be attributed to the reduction of the `MPI_Send` and `MPI_Irecv` durations to zero in the ideal trace, not to mention the `MPI_Finalize`. Contrary to this, the time spent in the `MPI_Allreduce` changes only slightly.

By the looks of it, MPI issues might be induced by the load imbalance rather than intrinsic communication overhead, but we cannot see this right now, for sure. Hence, we should look into the communication pattern first. This is even more the case because the relative portion of the MPI time is noticeable on this workload, and the increase of the time step parameter in the input file to the more representative value of 1000 drives this portion from 7 percent down to only 5.5 percent, on average (to be continued).

EXERCISE 5-8

Analyze and address the load imbalance in miniMD. What causes it? Replace miniMD with your favorite application and address the load imbalance there, provided this is necessary. What causes the imbalance?

Optimizing MPI Performance

If MPI overhead clearly dominates the overhead caused by the load imbalance, or if you simply do not see a practical way of addressing the load imbalance within the constraints of the target application and available time, you can still do well by addressing MPI performance issues.

Classifying the MPI Performance Issues

Several causes may lead to the MPI performance being lower than expected. They can be attributed roughly to the interaction of the main components of the system that include the platform, the MPI library, and the application involved.

The MPI library itself may not be optimally tuned for the platform at hand. Even though great care is taken to tune, for example, the Intel MPI library out of the box for the most modern Intel architectures, your system may be a bit different or a bit older than that covered by the default tuning process. In this case, the Intel MPI library can and should be tuned for the platform as a whole.

Also, the MPI library may not be optimally tuned for the application involved. The easiest example to show this is an application that is more latency than bandwidth bound, and thus not the one for which the Intel MPI library was tuned by default. Another example is an application that uses a specific number of MPI processes and several collective operations or point-to-point communication patterns that are not well represented by the Intel MPI Benchmarks used predominantly to tune Intel MPI. These would include the OSU benchmarks that focus on the network saturation exchanges.[14] If your application behaves like this, you may need to re-tune Intel MPI for it.

This relationship can be reversed, as well. Indeed, just as the Intel MPI library may be considered suboptimally tuned for a particular application, the application itself may be doing things that are bad for Intel MPI in particular and any MPI in general. Sometimes this involves interaction with the platform, sometimes it does not. For example, the MPI library usage of the cache may be competing with the application usage of it. Your methods will change depending on what you have to address.

If a MPI/platform interaction is involved, an application may be using intrinsically higher latency (e.g., internode) links for short messages. A high-quality MPI implementation like Intel MPI may sometimes be able to work around this by, say, rearranging collective operations so that the local part of the communication is done first. However, sometimes you will have to help it out.

If, however, the application is doing something intrinsically bad for any MPI, the main goal is to change the application to do the right thing. One fairly common example is a well-intentioned desire of some application developers to replace the collective operations that may not have been optimally tuned in the past by manual, point-to-point implementations thereof included into the application itself, sometimes in a pretty implicit form. This may indeed bring some performance improvement, but more often than not it does quite the opposite. Another example is the much beloved packing of noncontiguous data types into dense arrays and sending of them across and then unpacking them at the other end. Again, sometimes this makes sense, but more often it does not.

Addressing MPI Performance Issues

There is a bit of a chicken-and-egg problem once you turn toward optimizing MPI communication: what comes under scrutiny first—the platform, the MPI, or the application? The number of components and complexity of their direct and implicit interactions make it relatively difficult to give fast and ready advice for all possible situations. As a rule of thumb, keep in mind the following priorities:

1. *Map* the application upon the target platform. This includes, in particular, selection of the fastest communication fabrics, proper process layout and pinning, and other settings that affect the way application and platform interact via MPI mediation.

2. *Tune* the Intel MPI library for the platform and/or the application involved. If your application is bandwidth bound, you are likely to do well with the platform-specific tuning. If your application differs, you may need to cater to its particular needs.

3. *Optimize* the application for Intel MPI library. This includes typical MPI optimizations valid for any MPI implementation and specific Intel MPI tricks for varying levels of complexity and expected return on investment.

As usual, you will have to iterate until convergence or timeout. We will go through these steps one by one in the following sections. However, if in a particular case you perceive the need for bypassing some steps in favor of others, feel free to do so, but beware of spending a lot of time addressing the wrong problem first.

You will notice that we differentiate between optimization and tuning. *Optimization* is a wider term that may include tuning. *Tuning* normally concerns changing certain environment settings that affect performance of the target application. In other words, optimization may be more intrusive than tuning because deep optimization may necessitate source code modifications.

Mapping Application onto the Platform

Before you start the process of MPI optimization in earnest, you have to make sure that you are actually trying to optimize the application configuration that is suitable for the platform involved. The biggest potential problem here is improper process layout and pinning that may exercise slow communication paths where fast paths are needed and indeed possible.

Understanding Communication Paths

Intranode communication paths are typically the fastest the closer to the processor you get, with the shared memory ruling the realm, intranode busses like PCI Express coming next, and networking equipment bringing up the rear. However, in some cases the situation may be different, and the seemingly slower paths, like InfiniBand, may offer better bandwidth (see Figures 5-1 and 5-2), even intranode. You should definitely make yourself familiar with the quirks of the platform involved via extensive low-level benchmarking described earlier in this chapter.

A very important aspect of tuning is the selection of a proper communication fabrics combination for a particular job. Even though Intel MPI will try to choose the fastest possible fabrics automatically, in certain situations you will have to help it out. This is particularly true of the heterogeneous installations with the Intel Xeon Phi coprocessor involved, where there are so many paths to explore.

Beyond that, you have already seen several examples of one simple pinning setting's dramatically changing the behavior of certain benchmarks and applications. Generally speaking, if your application is latency bound, you will want its processes to share as much of the memory and I/O subsystem paths as possible. This means, in part, that you will try to put your processes onto adjacent cores, possibly even virtual ones. If your application is bandwidth bound, you will do better sharing as little of the memory subsystem paths as possible. This means, in part, putting your MPI processes on different processor sockets, and possibly even different nodes, if you use a cluster.

Selecting Proper Communication Fabrics

The Intel MPI Library selects reasonable communication fabric(s) by default. You can always find out what has been selected by setting the environment variable `I_MPI_DEBUG` to `2`. If the default selection does not look right, you can change this by using one of the two environment variables, the older `I_MPI_DEVICE` and the newer `I_MPI_FABRICS` environment variables, and their respective relations. Table 5-9 gives a brief overview of what you can do.

Using Scalable Datagrams

Note that when you use a DAPL-capable fabric, with or without shared memory involvement, you can select a scalable connectionless DAPL UD transport by setting the environment variable `I_MPI_DAPL_UD` to `enable`. This may make sense if your job runs on thousands of processes. Pure connection-oriented DAPL will normally be faster below this threshold.

Specifying a Network Provider

In certain situations, you will have to specify further details of the lower-level networking configuration. This is most often the case when you have more than one version of the DAPL stack installed on the system. You will probably have to ask around to determine whether or not you need to set the `I_MPI_DAPL_PROVIDER` and `I_MPI_DAPL_UD_PROVIDER` variables, and if so, what values to use when.

Using IP over IB

Another trick is to switch over to IP over IB (IPoIB) when using the TCP transport over InfiniBand. Here is how you can do this:

```
$ export I_MPI_TCP_NETMASK=ib0                  # for IP over IB or
$ export I_MPI_TCP_NETMASK=192.169.0.0          # for a particular subnet
```

Controlling the Fabric Fallback Mechanism

A word of caution for benchmarking: Intel MPI library will normally fall back upon the TCP communication if the primary fabric refuses to work, for some reason. This is a useful feature out in the field, where running a program reliably may be more important than running it fast. If you want to control the fallback path, enter this:

```
$ mpirun -genv I_MPI_FABRICS_LIST dapl,tcp -np <number of processes> ./your_app
```

However, this feature may be outright misleading during benchmarking. To make sure you are indeed using the fabric you selected, you may want to disable the Intel MPI built-in fallback mechanism by setting the environment variable `I_MPI_FALLBACK` to `disable`.

Using Multirail Capabilities

If your installation supports multirail capability, which modern InfiniBand hardware normally does by providing more than one port and possibly even InfiniBand adapter per node, you can exploit this over the OFA fabric. Just enter these magic commands depending on the number of adapters and ports you have:

```
$ export I_MPI_FABRICS=shm:ofa
$ export I_MPI_OFA_NUM_ADAPTERS=<n>  # e.g. 2 (1 by default)
$ export I_MPI_OFA_NUM_PORTS=<n>     # e.g. 2 (1 by default)
```

Detecting and Classifying Improper Process Layout and Pinning Issues

It is relatively easy to detect signs of improper process layout. Once you fire up ITAC on a trace file, you may either see exchange volumes spread very unevenly between the processes in the Message Profile chart (which by itself might be a sign of load imbalance that we have addressed), or you may notice overly long message lines crisscrossing substantial portions of the application event timeline. Normally, these latter messages will also lead to exorbitant wait times that may be picked up by the Performance Assistant and shown in the respective chart. This kind of problem can be observed both intra- and internode, as well as in the mixed configurations.

Now, any of these nice pictures will not tell you what is actually *causing* the observed issues. You will have to find that out yourself. In general, there are several ways of attacking this problem once you understand the root cause:

1. Rearrange the MPI processes and/or change their pinning at job startup to make offending messages go along the fastest possible communication path. This is the least intrusive method, which we will concentrate on below.

2. Use virtual process topologies to make MPI rearrange the process ranks according to the expected intensity of the interprocess communication. This implies that the MPI implementation does rearrange processes when asked to do so. Intel MPI does not do this at the moment, so we will basically gloss over this approach.

3. Rewrite the application to use a different communication pattern, or choose an alternative algorithm for the offending MPI collective operation. This is a more intrusive approach that we will consider when dealing with the MPI tuning and application modification later in this chapter.

Process pinning acts one level below the process layout. When you choose the process layout, you basically tell Intel MPI what node to put any particular MPI process on. Where exactly it lands on this node is decided by the process pinning. The ways to detect issues arising from improper process pinning are basically comparable to those recommended for the process layout investigation.

In the presence of NUMA, you will also have to mind the relationship between the processes and their memory. If the memory is located "close" to the process (in the NUMA sense), performance may be substantially better compared to when the process memory sits a few processor interconnect hops away. In the latter case, you will notice the platform latency and bandwidth limitations biting in much sooner than expected from the theoretical estimates and the low-level MPI benchmarking.

Finally, and less obviously, NUMA considerations may apply not only to the memory but also to the peripherals, like networking cards or interconnect busses. Again, if a card used for communication by a given process sits next to it in the node hierarchy, respective communication will most likely be noticeably faster compared to when the card sits several hops away. Add to this the unavoidable relationship between the memory and the networking cards, and you will get a pretty mess to clean up.

The overall picture gets even more complicated once you add dynamic processes to the mix. This includes process spawning and process attachment, especially in the heterogeneous environments. As it's still a relatively rarely used set of features, we will only touch upon them in this book.

Controlling Process Layout

The default process layout induced by the Intel MPI library is the so-called group round robin. This means that, by default, consecutive MPI ranks are placed on one node until all the available virtual cores are occupied. Once one node is fully loaded, the next node is dealt with in the same manner if it is available. Otherwise, the processes wraps around back to the very first node used, and so on.

There are several ways to control the process layout. The first of them acts *a priori*, at the job startup. The other method kicks in when the processes have been started. It uses the so-called virtual topologies defined by the MPI standard—the communicators created by using the `MPI_Cart_create`, `MPI_Graph_create`, and friends. This latter method presumes that the underlying MPI implementation indeed rearranges the MPI process ranks when asked to do so by the application programmer.

Controlling the Global Process Layout

Several methods exist to specify the process layout at startup, with varying degrees of brevity and precision. The easiest of them is use of the -ppn option and friends, including the environment variable I_MPI_PERHOST. You set the I_MPI_PERHOST environment variable to control process layout in the following manner:

```
$ export I_MPI_PERHOST=1               # makes round-robin distribution
$ export I_MPI_PERHOST=all             # maps processes to all virtualCPUs
                                         on a node (default)
$ export I_MPI_PERHOST=allcores        # maps processes to all physicalCPUs
                                         on a node
```

Alternatively, you can use one of the following mpirun options:

```
-perhost <number>       # group round-robin distribution with number of
                          processes per node
-ppn <number>           # "group round-robin", same as '-perhost <number>'
-grr <number>           # "group round-robin", same as '-perhost <number>'
-rr                     # round-robin distribution, same as '-perhost 1'
```

For example, this will put only two processes on each node:

```
$ mpirun -ppn 2 -np <number of processes> ./your_app
```

You will normally want to use the default process layout for pure MPI applications. For hybrid programs, you may want to decrease the number of processes per node accordingly, so as to leave enough cores for the OpenMP or another threading library to use. Finally, and especially in benchmarking the internode rather than the intranode communication, you will need to go down to one process per node.

Controlling the Detailed Process Layout

More detailed process layout control methods include the so-called long mpirun notation and the -hostfile, -machinefile, and, in the case of the scalable Hydra process manager only, the hosts options. Each of them essentially prescribes what processes to put where. The long notation is probably the most illustrative of all, so we will briefly review it here. You can look up the rest of the control possibilities in the *Intel MPI Library Reference Manual*.[15] Note that use of specific process placement is very common in benchmarking when you really want to make sure rank 0 sits here, rank 1 sits there, and so on. This may contribute substantially to the reproducibility of the results.

In normal operational mode, you will probably use the long notation more often when dealing with Intel Xeon Phi co-processor than otherwise, so let's demonstrate it in that case (here and elsewhere we split the overly long run strings into several lines by using the shell backslash/new line notation):

```
$ mpirun -genv I_MPI_MIC enable \
         -host `hostname` -np 2 ./your_app : \
         -host `hostname`-mic0 -np 16 ./your_app.mic
```

You can see that the run string is separated into two parts by the colon (`:`). The first half prescribes two MPI processes to be started on the host CPU. The second half puts 16 MPI processes upon the Intel Xeon Phi coprocessor connected to this CPU. This coprocessor conventionally bears the name of the host node plus the extension `-mic0`.

Setting the Environment Variables at All Levels

Note that you can set environment variables, such as those controlling the process pinning, either generally for all parts using the `-genv` option *before* the first `-host` option or individually in each part using the `-env` option, preferably *after* the respective `-host` option. Here is a good mixed example:

```
$ mpirun -genv I_MPI_MIC enable \
         -host `hostname` -env I_MPI_PIN_DOMAIN 4 -np 2 ./your_app : \
         -host `hostname`-mic0 -env I_MPI_PIN_DOMAIN 16 -np 4 ./your_app.mic
```

This particular command will turn on the Intel Xeon Phi coprocessor support, and then create OpenMP domains of four cores on the host processes and 16 cores on the Intel Xeon Phi coprocessor.

Controlling the Process Pinning

The Intel MPI library ships in several variants. The main ones are the sequential optimized library and the multithreaded optimized library. In the former library, the maximum supported thread level is `MPI_THREAD_SINGLE`. In the latter library, the maximum supported thread level is `MPI_THREAD_MULTIPLE`, with the default being `MPI_THREAD_FUNNELED`. More than one library is shipped so as to achieve maximum possible performance in each use case. Owing to Intel MPI development's constant work on optimization, it is not impossible that only the multithreaded library will be included in the delivery in the future, so we will concentrate on that right away.

The default process pinning imposed by the Intel MPI library is geared toward hybrid applications. It is roughly described by the following settings:

```
I_MPI_PIN=on
I_MPI_PIN_MODE=pm
I_MPI_PIN_DOMAIN=auto,compact
I_MPI_PIN_RESPECT_CPUSET=on
I_MPI_PIN_RESPECT_HCA=on
I_MPI_PIN_CELL=unit
I_MPI_PIN_ORDER=compact
```

There are several important aspects to keep in mind:

1. Process pinning is turned on by default. You may want to control this by setting the environment variable `I_MPI_PIN` to the values of `disable` or `enable` (likewise, `off` and `on`, or `false` and `true`, or just `0` and `1`, respectively).

2. The default process pinning is imposed by the process management infrastructure rather than the library itself. This has some far-reaching ramifications with respect to the memory and peripherals affinity we are going to consider in the next section. You probably do not want to interfere with this unless your job manager starts making trouble here.

3. There are two major methods of controlling the pinning, one of which focuses on hybrid ones (via the `I_MPI_PIN_DOMAIN` and friends) while the other is better suited for pure MPI programs (via the `I_MPI_PIN_PROCESSOR_LIST` and friends). If the former method is used, it normally overrides the latter if that is used as well.

4. The default `I_MPI_PIN_DOMAIN` value `auto` means that the domain size is defined by the formula *size=#cpu/#proc*, where *#cpu* is the number of virtual processors on the node and *#proc* is the number of the MPI processes started on the node. It is this domain into which all the threads belonging to the respective MPI process are placed. The qualifier `compact` above leads to the domains' being put as close to each other as possible in the sense of sharing the processor resources like caches. If you do not want this, you can try values of `scatter` and `platform` to go for the least possible resource sharing and the platform-specific thread ordering, respectively.

5. The default pinning takes into account the platform affinity setting (cf. `cpuset` command) and the locality of the InfiniBand networking cards (called host channel adapter, or HCA). It also prescribes targeting the virtual cores (`unit`) and compact domain ordering (`compact`) in the absence of respective qualifiers in the values of the `I_MPI_PIN_PROCESSOR_LIST` and `I_MPI_PIN_DOMAIN` environment variables.

There may be small deviations between the description given and the realities of the default pinning, so you should look into the aforementioned *Intel MPI Library Reference Manual* to learn all the details.

If you want to use OpenMP in your program, you better change the value `auto` to `omp`, in which case the size of the domain will be defined by the OpenMP specific means, like the value of the environment variable `OMP_NUM_THREADS` or `KMP_NUM_THREADS`. Likewise, the pinning inside the domain will be determined according to the OpenMP specific settings like `KMP_AFFINITY`, which we will consider in detail in Chapter 6.

Like the I_MPI_PIN_DOMAIN, the I_MPI_PIN_PROCESSOR_LIST has many possible values. The most practical values are as follows:

```
$ export I_MPI_PIN_PROCESSOR_LIST=all          # all virtual cores
$ export I_MPI_PIN_PROCESSOR_LIST=allcores     # all physical cores
$ export I_MPI_PIN_PROCESSOR_LIST=allsocks     # all processor sockets
```

When you start playing with exact process placement upon specific cores, both I_MPI_PIN_DOMAIN and I_MPI_PIN_PROCESSOR_LIST will help you by providing the list-oriented, bit mask–based, and symbolic capabilities to cut the cake exactly the way you want, and if you wish, by using more than one method. You will find them all fully described in the *Intel MPI Library Reference Manual*.

Controlling Memory and Network Affinity

There are no special means of controlling memory affinity in the Intel MPI library per se. However, as mentioned in the previous section, the library facilitates the operating system doing the right thing by setting the process pinning before the process launch. Under normal conditions, this means that the processor running a particular process will be located closely to the memory this process uses. At the same time, it is possible to use the system and third-party tools to affect the memory affinity (cf. numactl command), which will be reviewed in Chapter 6.

Contrary to this, networking affinity enjoys some level of support in Intel MPI Library, as represented by the I_MPI_PIN_RESPECT_HCA setting mentioned here. There are other settings available, but they are considered experimental at the moment and are reserved for the MPI implementors until better times.

Example 4: MiniMD Performance Investigation on Xeon Phi

Let's see what happens to the miniMD application and its mapping on the platform if we add Intel Xeon Phi coprocessors to the mix. First, you will need to get access to a machine that has them. In our case, we used the same cluster that happens to have several Intel Xeon Phi equipped nodes. Running the application on Intel Xeon Phi is only a bit more complicated than on the normal Xeon. You need to build the executable program separately for Intel Xeon and for Intel Xeon Phi. In the case of the miniMD, this is accomplished by the following commands (see also 1build.sh):

```
$ make intel            # for Intel Xeon
$ mv ./miniMD_intel ./miniMD_intel.host
$ make clean
$ make intel KNC=yes    # for Intel Xeon Phi
$ mv ./miniMD_intel ./miniMD_intel.mic
```

Here, we renamed both executables to keep them separate and distinguishable from each other and from the plain Xeon executable we may need to rebuild later on. This way we cannot spoil our executable programs by accident.

Running the program is similar to the workstation:

```
$ export I_MPI_MIC=enable
$ mpiexec.hydra \
      -env LD_LIBRARY_PATH /opt/intel/impi_latest/mic/lib:$MIC_LD_LIBRARY_PATH \
      -host `hostname`-mic0 -np 16 ./miniMD_intel.mic
```

These environment settings make sure that the Intel Xeon Phi coprocessor is found and that the path settings there are correct. If we compare performance of the programs on Intel Xeon and Intel Xeon Phi at different process counts, we get the results shown in Table 5-10:

Table 5-10. *MiniMD Execution Time on Intel Xeon or Intel Xeon Phi (Seconds, Cluster)*

MPI proc.	Xeon	Xeon Phi	Ratio, times
1	8.13	52.72	6.48
2	4.08	26.90	6.60
4	2.08	14.22	6.85
8	1.06	7.02	6.62
16	0.56	3.85	6.92
24	0.38	2.65	6.90
32	0.43	2.04	4.77
48	0.30	1.47	4.82
64		1.38	
96		1.12	
128		1.18	

As usual, we performed three runs at each process count and analyzed the results for variability, which was all below 1 percent in this case. You have certainly gotten used to this procedure by now, so that we can skip the details. From this table we can derive that a Xeon is roughly 6.5 to 6.9 times faster than Xeon Phi for the same number of MPI processes, as long as Xeon cores are not saturated. Note that this relationship holds for the core-to-core comparison (one MPI process results) and for MPI-to-MPI comparison (two through 24 MPI processes). So, you will need between 6.5 and 12 times more Xeon Phi processes to beat Xeon. Note that although Xeon Phi saturates later, at around 64 to 96 MPI processes, it never reaches Xeon execution times on 48 MPI processes. The difference is again around six times.

It may be interesting to see how speedup and efficiency compare to each other in the case of Xeon and Xeon Phi platforms; see Figure 5-13.

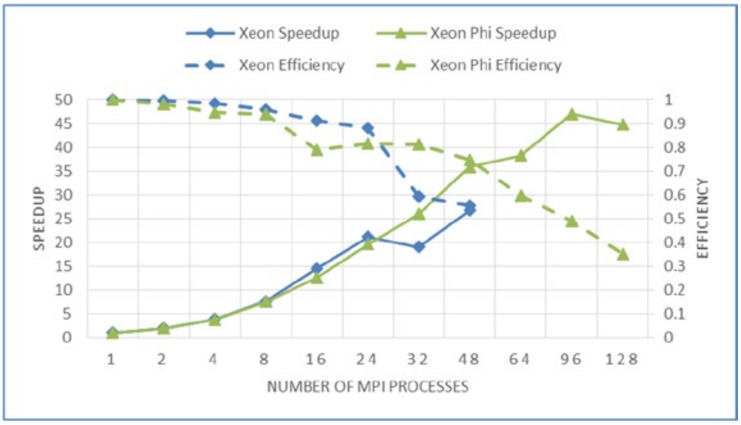


Figure 5-13. *MimiMD speedup and efficiency on Xeon and Xeon Phi platforms (cluster)*

Here, speedup is measured by the left-hand vertical axis, while efficiency goes by the right-hand one. Looking at this graph, we can draw a number of conclusions:

1. We can see that Xeon efficiency surpasses Xeon Phi's and goes very much along the ideal speedup curve until Xeon efficiency drops dramatically when we go beyond 24 MPI processes and start using virtual rather than physical cores. It then recovers somewhat by sheer weight of the resources applied.

2. Since this Xeon Phi unit has 61 physical cores, we observe a comparable effect at around 61 MPI processes as well.

3. Xeon surpasses Xeon Phi on efficiency until the aforementioned drop, when Xeon Phi takes over.

4. Xeon Phi becomes really inefficient and stops delivering speedup growth on a large number of MPI processes. It is possible that OpenMP threads might alleviate this somewhat.

5. There is an interesting dip in the Xeon Phi efficiency curve at around 16 MPI processes. What it is attributed to may require extra investigation.

If you try to use both Xeon and Xeon Phi at once, you will have to not only balance their respective numbers but also keep in mind that the data traversing the PCI Express bus may move slower than inside Xeon and Xeon Phi, and most likely will move slower most of the time, apart from large messages inside Xeon Phi. So, if you start with the aforementioned proportion, you will have to play around a bit before you get to the nearly ideal distribution, not to mention doing the process pinning and other tricks we have explored. A good spot to start from would probably be 16 to 24 MPI processes on Xeon and 64 to 96 MPI processes on Xeon Phi.

The required command will look as follows:

```
$ export I_MPI_MIC=1
$ export I_MPI_DAPL_PROVIDER_LIST=ofa-v2-mlx4_0-1u,ofa-v2-scif0
$ mpiexec.hydra -host `hostname` -np 16 ./miniMD_intel.host : \
    -env LD_LIBRARY_PATH /opt/intel/impi_latest/mic/lib:$MIC_LD_LIBRARY_PATH \
    -host `hostname`-mic0 -np 96 ./miniMD_intel.mic
```

Table 5-11 shows a result of our quick testing on the same platform:

Table 5-11. *MiniMD Execution Time on Intel Xeon and Intel Xeon Phi with Local Minima Highlighted (Seconds, Cluster)*

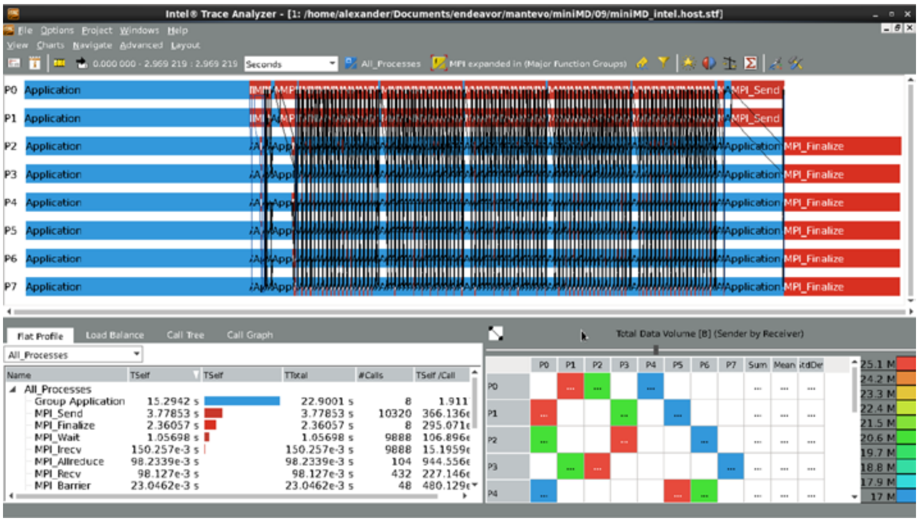
Xeon/Phi	48	64	96	128
8	1.396	1.349	1.140	1.233
16	1.281	1.324	*1.133*	1.134
24	1.190	1.256	1.137	1.222
48	*0.959*	1.219	1.157	1.093

We placed Xeon process counts along the vertical axis and Xeon Phi process counts along the horizontal axis. This way we could obtain a sort of two-dimensional data-distribution picture represented by numbers. Also, note that we prudently under- and overshot the guesstimated optimal process count ranges, just in case our intuition was wrong. And as it happens, it was wrong! We can observe two local minima: one for the expected 16:96 Xeon to Xeon Phi process count ratio. However, the better global minimum is located in the 48:48 corner of the table. And if we compare it to the best we can get on 48 Xeon–based MPI processes alone, we see that Xeon Phi's presence draws the result *down* by more than three times.

One can use ITAC to see what exactly is happening: is this imbalance induced by the aforementioned Xeon to Xeon Phi core-to-core performance ratio that has not been taken into account during the data distribution? Or is it by the communication overhead basically caused by the PCI Express bus? It may be that both effects are pronounced to the point of needing a fix. In particular, if the load imbalance is a factor, which it most likely is because the data is likely split between the MPI processes proportional to their total number, without accounting for the relative processor speed, one way to fight back would be to create a bigger number of OpenMP threads on the Xeon Phi part of the system. Quite unusually, you can control the number of threads using the program's own `-t` option. For example, the following command uses one of the better miniMD configurations while generating a valid ITAC trace file:

```
$ mpiexec.hydra -trace -host `hostname` -np 2 ./miniMD_intel.host -t 12 :\
    -env LD_LIBRARY_PATH \ /opt/intel/impi_latest/mic/lib:/opt/intel/itac_
latest/mic/lib:$MIC_LD_LIBRARY_PATH \
    -host `hostname`-mic0 -np 6 ./miniMD_intel.mic -t 32
```

Even a quick look at the resulting trace file shows that load imbalance caused by the platform heterogeneity is indeed the root cause of all the evil here, as shown in Figure 5-14.


Intel® Trace Analyzer - [1: /home/alexander/Documents/endeavor/mantevo/miniMD/09/miniMD_intel.host.stf]
File Options Project Windows Help
View Charts Navigate Advanced Layout
0.000 000 - 2.969 219 : 2.969 219
Seconds
All_Processes
MPI expanded in (Major Function Groups)
P0 Application MPI_Send
P1 Application MPI_Send
P2 Application MPI_Finalize
P3 Application MPI_Finalize
P4 Application MPI_Finalize
P5 Application MPI_Finalize
P6 Application MPI_Finalize
P7 Application MPI_Finalize
Flat Profile
Load Balance
Call Tree
Call Graph
Total Data Volume [B] (Sender by Receiver)
P0 P1 P2 P3 P4 P5 P6 P7 Sum Mean StdDev
25.1 M
24.2 M
23.3 M
22.4 M
21.5 M
20.6 M
19.7 M
18.8 M
17.9 M
17 M

All_Processes

Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
All_Processes					
Group Application	15.2942 s		22.9001 s	8	1.911
MPI_Send	3.77853 s		3.77853 s	10320	366.136e
MPI_Finalize	2.36057 s		2.36057 s	8	295.071e
MPI_Wait	1.05698 s		1.05698 s	9888	106.896e
MPI_Irecv	150.257e-3 s		150.257e-3 s	9888	15.1959e
MPI_Allreduce	98.2339e-3 s		98.2339e-3 s	104	944.556e
MPI_Recv	98.127e-3 s		98.127e-3 s	432	227.146e
MPI_Barrier	23.0462e-3 s		23.0462e-3 s	48	480.129e

Figure 5-14. *MiniMD trace file in ITAC (cluster, 2 Xeon processes, 6 Xeon Phi processes)*

Here, processes *P0* and *P1* sit on the Xeon, while the rest of them sit on the Xeon Phi. The difference in their relative speed is very clear from the direct visual comparison of the corresponding (blue) computing sections. We can discount the `MPI_Finalize` duration because it is most likely caused by the ITAC data post-processing. However, the `MPI_Send` and `MPI_Wait` times are out of all proportion.

Further analysis of the data-exchange pattern reveals that two closely knit groups take four processes each, with somewhat lower exchange volumes between the groups (not shown). Moreover, a comparison of the transfer rates that can be done by clicking on the *Message Profile* and selecting *Attribute to show/Maximum Transfer Rate* shows that the PCI Express links achieve at most 0.2 bytes per tick while up to 2 bytes per tick are possible inside Xeon and up to 1.1 bytes per tick inside Xeon Phi (not shown). This translates to about 0.23 GiB/s, 2.3 GiB/s, and 1.1 GiB/s, respectively, with some odd outliers.

Hence, we can hope for performance improvement if we do the following:

1. Split the Xeon and Xeon Phi portions into equal-size process groups (say, 4 vs. 4). This should match the data split performed by the program because currently the first two processes of the first group sit on Xeon and the other two are on Xeon Phi.

2. Use up to 12 times fewer threads on Xeon than on Xeon Phi (say, 4 vs. 48). This should compensate for the relative difference in processor speed.

3. Pray that the lower exchange volume in the fringes will not overload the PCI Express links. The difference in volumes (25 MiB vs. 17 MiB) is, however, rather small and may not suffice.

Indeed, if we follow these recommendations and change the run string accordingly, we get a substantial reduction in the program execution time (from 1.58 seconds to 1.26 seconds) despite the fact that we used *fewer* cores on Xeon and the same number of cores on Xeon Phi. This is, however, only the beginning of the journey, because we are still far away from the best Xeon-only result obtained so far (0.3 seconds; see Table 5-10). Given the prior treatise in this book, and knowing how to deal with the load imbalance in general, you can read other sources dedicated to Intel Xeon Phi programming if you want to pursue this path.[16] If, after that, the heterogeneity still shows through the less than optimal data-exchange paths, especially across the PCI Express lane, you can address this in other ways that we will discuss further along in this chapter.

EXERCISE 5-9

Find out the optimal MPI process to the OpenMP thread ratio for miniMD using a heterogeneous platform. Quantify this ratio in comparison to the relative component speeds. How much of the effect can be attributed to the computation and communication parts of the heterogeneity?

Example 5: MiniGhost Performance Investigation

Let's take on a beefier example this time. Instead of going for the realistic but relatively small workloads we used in the case of miniFE and miniMD earlier, we'll deal with the miniGhost from the NERSC-8 Trinity benchmark set.[17] This finite difference calculation will nicely complement the finite element and molecular dynamics programs we have considered so far. However, for Trinity, being a record-setting procurement, even the smallest configuration of its miniGhost benchmark will certainly overwhelm our workstation, so we will first have to reduce the size of the workload in order to make sense of it.

Using the benchmarking methods described earlier, you will find that setting the domain size to 200 cubed will do the trick. Moreover, you will learn that the best performance is achieved by taking 12 processes per node and running four OpenMP threads per process, and by splitting the task into 1:3:4 slabs in the X, Y, and Z directions, respectively. By the way, for the program to build, you will have to change the `Makefile` to reference Intel compilers, and also add the `-qopenmp` flag to the `OPT_F` and add the `-lifcore` library to the `LIBS` variables there. It is quite usual that some minor adjustments are necessary.

Long story made short, here is the run string we used for the workstation launch:

```
$ export OMP_NUM_THREADS=4
$ mpirun -np 12 ./miniGhost.x --scaling 1 --nx 200 --ny 200 --nz 200
  --num_vars 40 \
    --num_spikes 1 --debug_grid 1 --report_diffusion 21 --percent_sum 100 \
    --num_tsteps 20 --stencil 24 --comm_method 10 --report_perf 1 --npx 1
      --npy 3 --npz 4 \
    --error_tol 8
```

Built-in statistics output shows the role distribution among the top three MPI calls, as illustrated in Listing 5-15:

Listing 5-15. MiniGhost Statistics (Workstations, 12 MPI Processes, 4 OpenMP Threads per Process)

```
#                         [time]         [calls]       <%mpi>        <%wall>
# MPI_Allreduce           3.17148        9600          54.74         3.47
# MPI_Waitany             2.23135        1360          38.51         2.44
# MPI_Init                0.371742       12            6.42          0.41
```

High relative cost of the `MPI_Allreduce` makes it a very attractive tuning target. However, let us try the full-size workload first. Once we proceed to run this benchmark in its "small" configuration on eight cluster nodes and 96 MPI processes, we will use the following run string inspired in part by the one we used on the workstation (here, we highlighted deviations from the original script `run_small.sh`):

```
$ export OMP_NUM_THREADS=4
$ export I_MPI_PERHOST=12
$ mpirun -np 96 ./miniGhost.x --scaling 1 --nx 672 --ny 672 --nz 672
  --num_vars 40 \
    --num_spikes 1 --debug_grid 1 --report_diffusion 21 --percent_sum 100 \
    --num_tsteps 20 --stencil 24 --comm_method 10 --report_perf 1 --npx 4
      --npy 4 --npz 6 \
    --error_tol8
```

The irony of benchmarking in the context of a request for proposals (RFP) like NERSC-8 Trinity is that we cannot change the parameters of the benchmarks and may not be allowed to change the run string, either. This means that we will probably have to go along with the possibly suboptimal data split between the MPI processes this time; although looking at the workstation results, we would prefer to leave as few layers along the X axis as possible. However, setting a couple of environment variables upfront to ask for four instead of one OpenMP threads, and placing 12 MPI processes per node, might be allowed. Thus, our initial investigation did influence the mapping of the application to the platform, and we know that we may be shooting below the optimum in the data-distribution sense.

Further, it is interesting to see what is taking most of the MPI time now. The built-in statistics show a slightly different distribution; see Listing 5-16:

Listing 5-16. MiniGhost Statistics (Cluster, 8 Nodes, 12 MPI Processes per Node, 4 OpenMP Threads per Process)

```
#                           [time]        [calls]       <%mpi>        <%wall>
# MPI_Init                  149.771       96            44.95         4.17
# MPI_Allreduce             96.3654       76800         28.92         2.68
# MPI_Waitany               79.7788       17920         23.94         2.22
```

The sharp hike in relative `MPI_Init` cost is probably explained by the presence of the relatively slower network. It may also be explained by all the threads being busy when the network stack itself needs some of them to process the connection requests. Whatever the reason, this overhead looks abnormally high and certainly deserves further investigation.

This way or another, the `MPI_Init`, `MPI_Allreduce`, and `MPI_Waitany` take about 99 percent of all MPI time, between them. At least the first two calls may be amenable to the MPI-level tuning, while the last one may indicate some load imbalance (to be continued).

EXERCISE 5-10

Find the best possible mapping of your favorite application on your favorite platform. Do you do better with the virtual or the physical cores? Why?

Tuning the Intel MPI Library

Once you are certain that the application is properly mapped onto the platform, it makes sense to turn to the way the MPI Library is exploiting this situation. This is where MPI tuning for the platform comes into play. As mentioned above, you are likely to go this way if your application falls into the wide class of bandwidth-bound programs. In the case of latency-bound applications, you will probably want to use the application-specific tuning described later in this section.

Tuning Intel MPI for the Platform

There are two ways to tune Intel MPI for the platform: automatically by using the `mpitune` utility or manually.

If you elect to use the `mpitune` utility, run it once after installation and each time after changes in cluster configuration. The best configuration of the automatically selected Intel MPI tuning parameters is recorded for each combination of the communication device, the number of nodes, the number of MPI ranks, and the process layout. The invocation string is simple in this case:

```
$ mpitune
```

Be aware that this can take a lot of time, so it may make sense to run this job overnight. Note also that for this mode to work, you should have the writing permission for the `etc` subfolder of the Intel MPI Library installation directory, or use the `-od` option to select a different output directory.

Once the `mpitune` finishes, you can reuse the recorded values in any run by adding the `-tune` option to the normal `mpirun` invocation string; for example:

```
$ mpirun -tune -np 32 ./your_app
```

You can learn more about the `mpitune` utility in the *Tutorial: MPI Tuner for Intel MPI Library for Linux* OS*.[18] If you elect to do the tuning manually, you will have to dig into the MPI internals quite a bit. There are several groups of tuning parameters that you will need to deal with for every target fabric, number of processes, their layout, and the pinning. They can be split into point-to-point, collective, and other magical settings.

Tuning Point-to-Point Settings

Point-to-point operations form the basis of most MPI implementations. In particular, Intel MPI uses point-to-point communication extensively for the collective and (however counterintuitive this may seem) one-sided communications. Thus, the tuning should start with the point-to-point settings.

■ **Note** You can output some variable settings using the `I_MPI_DEBUG` value of `5`.

Adjusting the Eager and Rendezvous Protocol Thresholds

MPI implementations normally support two communication protocols:

- *Eager protocol* sends data immediately regardless of the availability of the matching receive request on the other side. This protocol is used normally for short messages, essentially trading better latency on the sending side for the danger of having to allocate intermediate buffers on the receiving side when the respective receive operation has not yet been posted.
- *Rendezvous protocol* notifies the receiving side on the data pending, and transfers it only once the matching receive request has been posted. This protocol tries to avoid the cost of the extra buffer allocation on the receiving side at the sacrifice of, typically, two extra short messages used for the notification and acknowledgment.

The protocol switchover point is controlled by the environment variable `I_MPI_EAGER_THRESHOLD`. Below and at this integral value that currently defaults to 256 KiB, the eager protocol is used. Above it, the rendezvous protocol kicks in. As a rule of thumb, the longer the messages you want to send immediately, the higher will be your optimal eager threshold.

Changing DAPL and DAPL UD Eager Protocol Threshold

Specifics of the Intel MPI Library add another, lower-level eager/rendezvous protocol threshold to the DAPL and DAPL UD communication paths. This has to do with how messages are sent between the processes using Remote Direct Memory Access (RDMA) methods. Basically, the lower-level eager protocol tries to avoid the cost of extra memory registration, while the rendezvous protocol goes for this registration to speed up the resulting data transfer by bypassing any intermediate buffers.

As in the case of the high-level eager threshold, each of the fabrics has its own threshold, called `I_MPI_DAPL_DIRECT_COPY_THRESHOLD` and `I_MPI_DAPL_UD_DIRECT_COPY_THRESHOLD`, respectively. When setting these environment variables, you will have to balance the desire to send messages off immediately with the increase in memory consumption associated with the raised value of the respective threshold.

Bypassing Shared Memory for Intranode Communication

It may happen on certain platforms that fabric performance overtakes the shared memory performance intranode. If it happens at all, it normally occurs at around 350 KiB message size. If your preliminary benchmarking reveals this situation, set the environment variable `I_MPI_SHM_BYPASS` to `enable`. This will make Intel MPI use the DAPL or TCP fabrics, if selected, for message sizes larger than the value of the environment variable `I_MPI_INTRANODE_EAGER_THRESHOLD` that currently defaults to 256 KiB for all fabrics but `shm`.

Bypassing the Cache for Intranode Communication

As a final note on point-to-point thresholds, there is a way to control what variant of the memory copying is used by the shared memory communication path. If you set the environment variable `I_MPI_SHM_CACHE_BYPASS` to `enable`, Intel MPI Library will use the normal, cache-mediated memory for messages below the values of the `I_MPI_SHM_CACHE_BYPASS_THRESHOLDS` and special non-temporal memory copy for larger messages. If activated, this feature may prevent the so-called cache pollution by data that will be pushed out of cache by additional incoming message segments anyway.

This last is a fairly advanced control, so you should approach it with care and read the respective part of the *Intel MPI Library Reference Manual*. The default values set to half of the size of L2 cache are normally adequate, but you may want to set them to the size of the L1 cache if you feel adventurous; for example:

```
$ export I_MPI_SHM_CACHE_BYPASS_THRESHOLDS=16384,16384,-1,16384,-1,16384
$ mpirun -np 2 -genv I_MPI_FABRICS shm IMB-MPI1 PingPong
```

Choosing the Best Collective Algorithms

Now that you are sure of your fabric selection and the most important point-to-point thresholds, it is the right time to proceed to tuning the collective operations. Certainly, you should make a list of operations that are relevant to your task. Looking into the built-in statistics output by the Intel MPI Library is a good first step here.

As it happens, Intel MPI Library provides different algorithms for each of the many MPI collective operations. Each of these algorithms has its strengths and weaknesses, as well as its possible limitations on the number of processes and message sizes it can sensibly handle.

■ **Note** You can output default collective settings using the `I_MPI_DEBUG` value of `6`.

You can use the environment variables named after the pattern `I_MPI_ADJUST_<opname>`, where the `<opname>` is the name of the respective collective operation. This way you come to the variable names like `I_MPI_ADJUST_ALLREDUCE`.

If we consider the case of the `MPI_Allreduce` a little further, we will see that there are no less than eight different algorithms available for this operation alone. Once again, the *Intel MPI Library Reference Manual* is your friend. Here, we will only be able to give some rules of thumb as to the algorithm selection by their class. To see how this general advice fits your practical situation, you will have to run a lot of benchmarking jobs to determine where to change the algorithm, if at all. A typical invocation string looks as follows:

```
$ mpirun -genv I_MPI_ADJUST_ALLREDUCE 4 -np 16 IMB-MPI1 Allreduce
```

You can certainly use any other benchmark, or even application, you want for this tuning. We will stick to the IMB here, out of sheer weight of experience. This way or another, you will end up with pretty fancy settings of the following kind that will have to be put somewhere (most likely, a configuration file):

```
$ export I_MPI_ADJUST_ALLGATHER= \
    '1:4-11;4:11-15;1:15-27;4:27-31;1:31-32;2:32-51;3:51-5988;4:5988-13320'
```

Well, it's your choice. Now, going through the most important collective operations in alphabetical order, in Table 5-12, we issue general recommendations based on extensive research done by Intel engineers.[19] You should take these recommendations with a grain of salt, for nothing can beat your own benchmarking.

Table 5-12. *Intel MPI Collective Algorithm Recommendations*

Operation	Algorithm	Small msgs	Large msgs	Rec. PPN
MPI_Allgather	(1) Recursive Doubling	+	+	1*
	(2) Bruck's	+	+	1*
	(3) Ring		+	any
	(4) Topological Gatherv/Bcast	+		>1
MPI_Allreduce	(1) Recursive Doubling	+		
	(2) Rabenseifner's	+	+	
	(3) Reduce/Bcast		+**	1
	(4) Topological Reduce/Bcast		+**	>1
	(5) Binomial Tree	+		1
	(6) Topological Binomial Tree	+		>1
	(7) Shumilin's Ring		+**	
	(8) Ring		+	
MPI_Alltoall	(1) Bruck's	+		
	(2) Isend/Irecv		+	
	(3) Pairwise Exchange		+	
	(4) Plum's	+	+	

(*continued*)

Table 5-12. (*continued*)

Operation	Algorithm	Small msgs	Large msgs	Rec. PPN
MPI_Barrier	(1) Dissemination	N/A	N/A	1
	(2) Recursive Doubling	N/A	N/A	1
	(3) Topology Dissemination	N/A	N/A	>1
	(4) Topology Recursive Doubling	N/A	N/A	>1
	(5) Binominal Gather/Scatter	N/A	N/A	1
	(6) Topology Binominal Gather/Scatter	N/A	N/A	>1
MPI_Bcast	(1) Binomial Tree	+		1
	(2) Recursive Doubling	+	+	1
	(3) Ring		+	1
	(4) Topological Binomial Tree	+		>1
	(5) Topological Recursive Doubling	+	+	>1
	(6) Topological Ring		+	>1
	(7) Shumilin's		+**	
MPI_Gather &	(1) Binomial Tree	+	+	1
MPI_Scatter	(2) Topological Binomial Tree	+	+	>1
	(3) Shumilin's		+	
MPI_Reduce	(1) Shumilin's		+**	1
	(2) Binomial Tree	+		1
	(3) Topological Shumilin's		+**	>1
	(4) Topological Binomial Tree	+		>1
	(5) Rabenseifner's	+	+	1
	(6) Topological Rabenseifner's	+	+	>1

**Only for large messages, otherwise any PPN.*

***For buffers larger than the number of processes times the algorithm specific segment size.*

Tuning Intel MPI Library for the Application

Again, you can tune Intel MPI for a particular application either automatically using the `mpitune` utility or manually. The `mpitune` invocation string is a little more complicated in this case (the use of backslashes and quotes is mandatory):

```
$ mpitune --application \"mpiexec -np 32 ./my_app\" --of ./my_app.conf
```

This way you can tune Intel MPI for any kind of MPI application by specifying its command line. By default, performance is measured as the inverse of the program execution time. To reduce the overall tuning time, use the shortest representative application workload (if applicable). Again, this process may take quite a while to complete.

Once you get the configuration file, you can reuse it any time in the following manner:

```
$ mpirun -tune ./my_app.conf -np 32 ./my_app
```

Note that here you not only mention the file name but also use the same number of processes and generally the same run configuration as in the tuning session. (You can learn more about this tuning mode in the aforementioned tuning tutorial.)

If you elect to tune Intel MPI manually, you will basically have to repeat all that you did for the platform-specific tuning described in the previous section, with the exception of using your application or a set of representative kernels instead of the IMB for the procedure. Certainly, you will do better instead by addressing only those point-to-point patterns and collective operations at the number of processes, their layout and pinning, and message sizes that are actually used by the target application. The built-in statistics and ITAC output will help you in finding out what to go for first.

Using Magical Tips and Tricks

Sometimes you will have to foray beyond the normal tuning of the point-to-point and collective operations. Use the following expert advice sparingly: the deeper you get into this section, the closer you are moving toward Intel MPI open heart surgery.

Disabling the Dynamic Connection Mode

Intel MPI establishes connections on demand if the number of processes is higher than 64. This saves some time at startup and may diminish the total number of connections established, so it is an important scalability feature. However, it may also lead to certain delays during the first exchange that require a new connection to be set up. Set the environment variable `I_MPI_DYNAMIC_CONNECTION` to `disable` in order to establish all connections upfront.

Applying the Wait Mode to Oversubscribed Jobs

Sometimes applications do a lot of I/O and may profit from running in the so-called oversubscribed mode—that is, in the mode with the number of processes that exceeds the number of the available cores. In these rare cases, try to set the environment variable `I_MPI_WAIT_MODE` to `enable` so as to make MPI processes wait for an interrupt to be delivered to them instead of polling the fabrics for the new messages. Even though Intel MPI possesses a rather elaborate back-off strategy in the default polling mode, going for the outright wait mode (also called event-driven mode) may bring a quantum leap in performance under some circumstances.

Fine-Tuning the Message-Passing Progress Engine

Deep inside any MPI implementation there sits a vital component called the *progress engine*. It is this component that actually pushes bytes into the fabric layers and makes sure they proceed to their respective destinations, reach them, and are put into the user buffers on the other side.

Typically, this component is called (or, in implementor speak, "kicked") every time there is a substantial call into the MPI Library that can be implicated in moving data across the wires. Examples of this class include the `MPI_Send`, `MPI_Recv`, `MPI_Wait`, `MPI_Test`, `MPI_Probe`, all collective operations, their multiple friends and relations, and some other calls. This approach is called *synchronous invocation of the progress engine*. On a level with this, an MPI implementation can offer asynchronous capabilities by, say, running part of the progress engine in a background thread.

This way or another, this component is faced with a difficult existential dilemma. On one hand, it needs to be reactive to new messages coming and going, in order to achieve acceptable latency. On the other hand, it should try to avoid using up too much of the processor's time, for this would make the overall system performance go down. To address this dilemma, various MPI implementations offer so-called back-off strategies that try to find the right balance between reactivity and resource consumption. Of course, there are multiple settings that control this strategy, and the default tuning tries to select them so that a typical application will do alright.

Intel MPI has elaborate and finely tuned back-off mechanisms. Should you become dissatisfied with the default settings, however, try to increase the `I_MPI_SPIN_COUNT` value from the default of `1` for one process per node and `250` for more than one process per node. This will change the number of times the progress engine spins, waiting for a message or connection request, before the back-off strategy kicks in. Higher values will favor better latency, to a degree. If you raise this value too much, you will start burning too many CPU cycles, polling the memory needlessly.

If you run more than one process per node that use the shared memory channel for data exchange, try to increase the `I_MPI_SHM_SPIN_COUNT` value above its default of `100`. This may benefit multicore platforms when the application uses topological algorithms for collective operations.

Reducing the Pre-reserved DAPL Memory Size

Large-scale applications may experience memory resource pressures due to a big number of pre-allocated buffers pinned to the physical memory pages. If you do not want to go for the DAPL UD mode, use the environment variable `I_MPI_DAPL_BUFFER_NUM` to decrease the number of buffers for each pair in a process group. The default value is `16`.

If you increase this value, you may notice better latency on short messages (see the low-level eager protocol threshold mentioned earlier). In addition, if your application mostly sends short messages, you can try to reduce the DAPL buffer size by changing the environment variable `I_MPI_DAPL_BUFFER_SIZE`. The default value is `23808`.

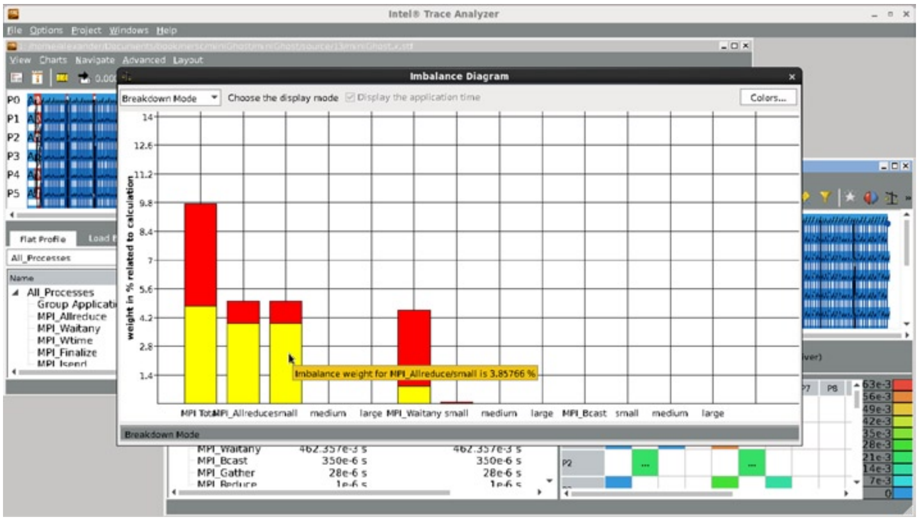
Finally, you can try to set the environment variable `I_MPI_DAPL_SCALABLE_PROGRESS` to `enable` for high process count. This is done automatically for more than 128 processes, though.

What Else?

Here is an assorted mix of tips and tricks you may try in your spare time:

- `I_MPI_SSHM=1` Turns on the scalable shared memory path, which might be useful on the latest multicore Intel Xeon processors and especially on the many-core Intel Xeon Phi coprocessor.
- `I_MPI_OFA_USE_XRC=1` Turns on the extensible reliable connection (XRC) capability that may improve scalability for several thousand nodes.
- `I_MPI_DAPL_UD_RDMA_MIXED=1` Makes DAPL UD use connectionless datagrams for short messages and connection-oriented RDMA for long messages.
- `I_MPI_DAPL_TRANSLATION_CACHE_AVL_TREE=1` May be useful for applications sending a lot of long messages over DAPL.
- `I_MPI_DAPL_UD_TRANSLATION_CACHE_AVL_TREE=1` Same for DAPL UD.

Of course, even this does not exhaust the versatile toolkit of tuning methods available. Read the Intel MPI documentation, talk to experts, and be creative. This is what this work is all about, right?


Intel® Trace Analyzer
File Options Project Windows Help
View Charts Navigate Advanced Layout
Imbalance Diagram
Breakdown Mode
Choose the display mode
Display the application time
Colors...
weight in % related to calculation
MPI Total MPI_Allreduce small medium large MPI_Waitany small medium large MPI_Bcast small medium large
Breakdown Mode
Flat Profile
All_Processes
Name
All_Processes
Group Application
MPI_Allreduce
MPI_Waitany
MPI_Wtime
MPI_Finalize
MPI_Isend

Imbalance weight for MPI_Allreduce/small is 3.85766 %

```
MPI_Waitany   462.357e-3 s   462.357e-3 s
MPI_Bcast     350e-6 s       350e-6 s
MPI_Gather    28e-6 s        28e-6 s
MPI_Reduce    1e-6 s         1e-6 s
```

Figure 5-15. *MiniGhost trace file in ITAC imbalance diagram breakdown mode (Workstation, 12 MPI processes, 4 OpenMP threads)*

Example 5 (cont.): MiniGhost Performance Investigation

Figure 5-15 shows the split of the MPI and load imbalance issues in the breakdown mode.

So, the total MPI overhead is evenly split between the `MPI_Allreduce` and the `MPI_Waitany`. Most of the `MPI_Allreduce` overhead is induced by load imbalance on small messages, while most of the `MPI_Waitany` overhead is caused by actual communication that we will analyze later on. We can assume that the picture will be qualitatively the same on the cluster. So, if you decide to address the `MPI_Allreduce` performance right away, which is not recommended, you can do some benchmarking at the target node counts for all `MPI_Allreduce` algorithms to see whether there is anything to haul there. Given several MPI processes per node and short messages dominating the `MPI_Allreduce` overhead, topology-aware algorithm number 6 is going to be your first preference (see Table 5-12). Such a trial is very easy to perform. Just enter the following command before the launch:

```
$ export I_MPI_ADJUST_ALLREDUCE=6
```

A quick trial we performed confirmed that algorithm number 6 was among the best for this workload. However, algorithms 1 and 2 fared just as well and were only 0.2 seconds below the default one. Hence, most likely, optimization of the program source code aimed at reduction of the irregularity of the exchange pattern will bring more value if done upfront here. That may include both load imbalance correction and tuning of the communication per se, because they may be interacting detrimentally with each other (to be continued).

EXERCISE 5-11

Try your hand at both platform- and application-specific Intel MPI tuning, using your favorite platform and application. Gauge the overall performance improvement. Identify the cases where platform-specific tuning goes against the application-specific one.

Optimizing Application for Intel MPI

At last, it is time to turn to the application itself. That is, unless you noticed much earlier a grave and apparent problem that went against all good MPI programming practices. In that case, you may want to try and fix that problem first, provided you make double sure it is that problem that is causing trouble—as usual.

You can sensibly apply the advice contained in this section only if you have access to the application source code. It may be way out of reach in most industrial situations in the field. This situation is, however, different if you are using open-source software or have been graciously granted a source code license to a piece of closed-source code. Thus, we are talking about real optimization rather than tuning here, and real optimization takes time—a luxury that you most likely will not have under real conditions.

There are quite a few things that can go wrong. This book is not a guide to MPI programming per se, so we will be brief and will focus on the most important potential issues.

Avoiding MPI_ANY_SOURCE

Try to make your exchanges deterministic. If you have to use the `MPI_ANY_SOURCE`, be aware that you may be paying quite a bit on top for every message you get. Indeed, instead of waiting on a particular communication channel, as prescribed by a specific receive operation, in the case of `MPI_ANY_SOURCE` the MPI Library has to poll all existing connections to see whether there is anything matching on input. This means extensive looping and polling, unless you went for the wait mode described earlier. Note that use of different message tags is not going to help here, because the said polling will be done still.

Generally, all kinds of nondeterminism are detrimental and should be avoided, if possible. One way this cannot be done is when a server process distributes some work among the slave processes and waits to them to report back. However the work is apportioned, some will come back earlier than others, and enforcing a particular order in this situation might slow down the overall job. In all other cases, though, try to see whether you can induce order and can benefit from doing that.

Avoiding Superfluous Synchronization

Probably the worst thing application programmers do, over and over again, is superfluous synchronization. It is not uncommon to see, for example, iterations of a computational loop separated by an `MPI_Barrier`. If you program carefully and remember that MPI guarantees reliable and ordered data delivery between any pair of processes, you can skip this synchronization most of the time. If you are still afraid of missing things or mixing them up, start using the MPI message tags to instill the desired order, or create a communicator that will ensure all messages sent within it will stay there.

Another aspect to keep in mind is that, although collective operations are not required to synchronize processes by the MPI standard (with the exception of the aforementioned `MPI_Barrier`, of course), some of them may do this, depending on the algorithm they use. This may be a boon in some cases, because you can exploit this side effect to your ends. You should avoid doing so, however, because if the algorithm selection is changed for some reason, you may end up with no synchronization point where you implied one, or vice versa.

About the only time when you may want to introduce extra synchronization points is in the search for the load imbalance and its sources. In that case, having every iteration or program stage start at approximately the same time across all the nodes involved may be beneficial. However, this may also tilt the scale so that you will fail to see the real effect of the load imbalance.

Using Derived Datatypes

There are a few more controversial topics besides the one related to the derived datatypes (one word, as it appears in the MPI standard). As you may remember, these are opaque MPI objects that basically describe the data layout in memory. They can be used almost without limitation in any imaginable data-transfer operation in MPI.

Unfortunately, they suffer from a bad reputation. In the early days of MPI, the implementors could not always make data transfer efficient in the presence of the derived datatypes. This may still be the case now in some implementations, especially if the datatype involved is, well, too involved. Owing to this mostly ungrounded fear, application programmers try to use contiguous data buffers; and if they have to work with noncontiguous data structures, they do the packing in and out themselves by hand or by using the `MPI_Pack`/`MPI_Unpack` calls.

For most of the time, though, this is a thing of the past. You can actually win quite a bit by using the derived datatypes, especially if the underlying MPI implementation provides native support for them. Modern networks and memory controllers can do scatter, gather, and some other manipulations with the data processed on the fly, without any penalty you would need to take care of at this level. Moreover, buffer management done inside the MPI library, as well as packing and unpacking if that ever becomes necessary, is implemented using techniques that application programmers may simply have no everyday access to. Of course, if you try hard enough, you will write your own specific memory copy utility or a datatype unrolling loop that will do better than the generic procedure used by your MPI implementation. Before you go to this trouble, however, make sure you prove it's worth doing.

Using Collective Operations

Another rudimentary fear widespread among application programmers is that of suboptimal collective operations. Especially, older codes will go to great pains tore-implement all collective operations they need on the basis of the earlier status of MPI implementations.

Again, this is mostly a thing of the past. Unless you know a brilliant new algorithm that beats, hands down, all that can be extracted by the MPI tuning described earlier, you should try to avoid going for the point-to-point substitute. Moreover, you may actually win big by replacing the existing homegrown implementations with an equivalent MPI collective operation. There may be exceptions to this recommendation, but you will have to justify any efforts very carefully in this case.

Betting on the Computation/Communication Overlap

Well, don't. Most likely you will lose out. That is, there is some overlap in certain cases, but you have to measure its presence and real effect before you can be sure. Let's look into a couple of representative cases where you can hope to get something in return for the effort of converting mostly deterministic blocking communication into the controlled chaos of nonblocking transfers (again, this is the way the MPI standard decided to refer to these operations).

This method may be effective if you notice that blocking calls make the program stall and you have eliminated all other possible reasons for this happening. That is, your program is soundly mapped onto the platform, is well load balanced, and runs on top of a tuned MPI implementation. If in this case you still see that some processes stall in vastly premature receive operations; or, on the contrary, you can detect an inordinately high amount of unexpected receives (that is, messages arrive before the respective receive operation is posted); or if your sending processes are waiting for the data to be pumped out, you may need to act. A particular case of unnecessary serialization that happens when processes wait for each other in turn is well described in the *Tutorial: Detecting and Removing Unnecessary Serialization.*[20]

The replacement per se is rather trivial, at least at first. Every blocking send operation is replaced by its nonblocking variant, like `MPI_Send` by `MPI_Isend` or `MPI_Recv` by `MPI_Irecv`, with the closing call like `MPI_Wait` or `MPI_Test` issued later in the program. You can also group several operations by using the `MPI_Waitall`, `MPI_Waitsome`, and `MPI_Waitany`, and their `MPI_Test` equivalents. Here, you will do well by ordering the requests passed to these calls so that those most likely to be completed first come first. Normally, you want to post a receive operation just in time for the respective send operation to match it on the other side. You may even go for special variations on the send operations, like buffered, synchronous, or ready sends, in case this is warranted by your application and it brings a noticeable performance benefit. This can be done with or without making them nonblocking, by the way. Moreover, you can even generate so-called generic requests or use persistent operations to represent these patterns, provided doing so brings the desired performance benefit.

What is important to understand before you dive in is that the standard `MPI_Send` can be mapped onto any blocking send operation depending on the message size, internal buffer status in the MPI library, and some other factors. Most often, small messages will be sent out eagerly in order to return control back to the application as soon as possible. To this end, even a copy of the user buffer may be made, as in the buffered send, if the message passing machinery appears overloaded at the moment. In any case, this is almost equivalent to a nonblocking send operation, with the very next MPI call implicated in the data transfer in any way actually kicking the progress engine and doing what an `MPI_Isend` and/or `MPI_Test` would have done at that moment. Changing this blocking operation to a nonblocking one would probably be futile, in many cases.

Likewise, large messages will probably be sent using the rendezvous protocol mentioned above. In other words, the standard send operation will effectively become a synchronous one. Depending on the MPI implementation details, this may or may not be equivalent to just calling the `MPI_Ssend`. Once again, in absence of a noticeable computation/communication overlap, you will not see any improvement if you replace this operation with a nonblocking equivalent.

More often than not, what does make sense is trying to do bilateral exchanges by replacing a pair of sends and receives that cross each other by the `MPI_Sendrecv` operation. It may happen to be implemented so that it exploits the underlying hardware in a way that you will not be able to reach out for unless you let MPI handle this transfer explicitly. Note, however, that a careless switch to nonblocking communication may actually introduce extra serialization into the program, which is well explained in the aforementioned tutorial.

Another aspect to keep in mind is that for the data to move across, something or someone—in the latter case, you—will need to give the MPI library a chance to help you. If you rely on asynchronous progress, you may feel that this matter has been dealt with. Actually, it may or it may not have been, and even if it has been addressed, doing some relevant MPI call in between, be aware that even something apparently pointless, like an `MPI_Iprobe` for a message that never comes, may speed up things considerably. This happens because synchronous progress is normally less expensive than asynchronous.

Once again, here the MPI implementation faces a dilemma, trading latency for guarantee. Synchronous progress is better for latency, but it cannot guarantee progress unless the program issues MPI calls relatively often. Asynchronous progress can provide the necessary guarantee, especially if there are extra cores or cards in the system doing just this. However, the context switch involved may kill the latency. It is possible that in the future, Intel MPI will provide more controls to influence this kind of behavior. Stay tuned; until then, be careful about your assumptions and measure everything before you dive into chaos.

Finally, believe it or not, blocking transfers may actually help application processes self-organize during the runtime, provided you took into account their natural desires. If your interprocess exchanges are highly regular, it may make sense to do them in a certain order (like north-south, then east-west, and so on). After initial shaking in, the processes will fall into lockstep with each other, and they will proceed in a beautifully synchronized fashion across the computation, like an army column marching to battle.

Replacing Blocking Collective Operations by MPI-3 Nonblocking Ones

Intel MPI Library 5.0 provides MPI-3 functionality while maintaining substantial binary compatibility with the Intel MPI 4.x product line that implements the MPI-2.x standards.[21] Thus, you can start experimenting with the most interesting features of the MPI-3 standard right away. We will review only the nonblocking collective operations here, and bypass many other features.[22] In particular, we will not deal with the one-sided operations and neighborhood collectives, for their optimization is likely to take some time yet on the implementor side. Of course, if you want to experiment with these new features, nobody is going to stop you. Just keep in mind that they may be experimenting with you in return.

Contrary to this, nonblocking collective operations are relatively mature, even if their tuning may still need to be improved. You can replace any blocking collective operation (including, surprisingly, the `MPI_Barrier`) by a nonblocking version of it, add a corresponding closing call later in the program, and enjoy—what?

Let's see in more detail what you may hope to enjoy. First, your program will become more complicated, and you will not be able to tell what is happening with the precision afforded by the blocking collectives. This is a clear downside. Even in the case of the `MPI_Ibarrier`, you will not be able to ascertain when exactly the synchronization happens, whether in the `MPI_Ibarrier` call itself (which is possible) or in the matching closing call (which is probably desired). All depends on the algorithm selected by the implementation, and this you can control only externally, if at all.

Next, tuning of the settings for the blocking collectives may not influence the nonblocking ones and vice versa. Indeed, tuning of the nonblocking operations may not be controllable by you at this moment, at all. In addition, the MPI standard specifically clarifies that the blocking and nonblocking settings may be independent of each other, for the sake of making proper choices on the actual performance benefits observed. This is another clear downside.

On the bright side, you can use more than one nonblocking collective at a time over any communicator, and hope to exploit the computation/communication overlap in as much as is supported by the MPI library involved. In the Intel MPI Library, you may profit from setting the environment variable `MPICH_ASYNC_PROGRESS` to `enable`.

EXERCISE 5-12

If your application fares better with the MPI-3 nonblocking collectives inside, let us know; we are looking for good application examples to justify further tuning of this advanced MPI-3 standard feature.

Using Accelerated MPI File I/O

If your program relies on MPI file I/O, you can speed it up by telling Intel MPI what parallel file system you are using. If this is PanFS,[23] PVFS2,[24] or Lustre,[25] you may obtain noticeable performance gain because Intel MPI will go through a special code path designed for the respective file system. To achieve this, enter the following commands:

```
$ export I_MPI_EXTRA_FILE_SYSTEM=on
$ export I_MPI_EXTRA_FILE_SYSTEM_LIST=panfs,pvfs2,lustre
```

You can mention only those file systems that interest you in the second line, of course.

Example 5 (cont.): MiniGhost Performance Investigation

Analysis of the full miniGhost trace file done on the "small" problem size on the cluster basically confirms all findings observed on the workstation (not shown). Surprisingly, ITAC traces do not show `MPI_Init` anomaly in either case. Possibly, we have to do with a so-called Heisenbug that disappears due to observation.

That phenomenon aside, if we can fix the substantially smaller workstation variant, we should see gains in the bigger cluster case. This can be further helped by adjusting the workstation run configuration so that it fully resembles the situation within one node of the "small" cluster run by 12 MPI processes, four OpenMP threads, and respective process layout and grid size. Thus, the problems to be addressed, in order of decreasing importance, are as follows:

1. `MPI_Init` overhead visible only in the built-in statistics output.

2. Load imbalance that hinders proper `MPI_Allreduce` performance. This is the biggest issue at hand.

3. Communication related to the `MPI_Waitany` that may be interacting with the load imbalance and detrimentally affecting the `MPI_Allreduce` as well.

The `MPI_Init` overhead may need to be confirmed by repeated execution and independent timing of the `MPI_Init` invocation using the `MPI_Wtime` to be embedded into the main program code for this purpose (see file `main.c`). If this confirms that the effect manifested by the statistics output is consistently observable in other ways, we can probably discount the ITAC anomaly as a Heisenbug. At the moment of this writing, however, our bets were on the involuntary change of the job manager queue that may have contributed to this effect.

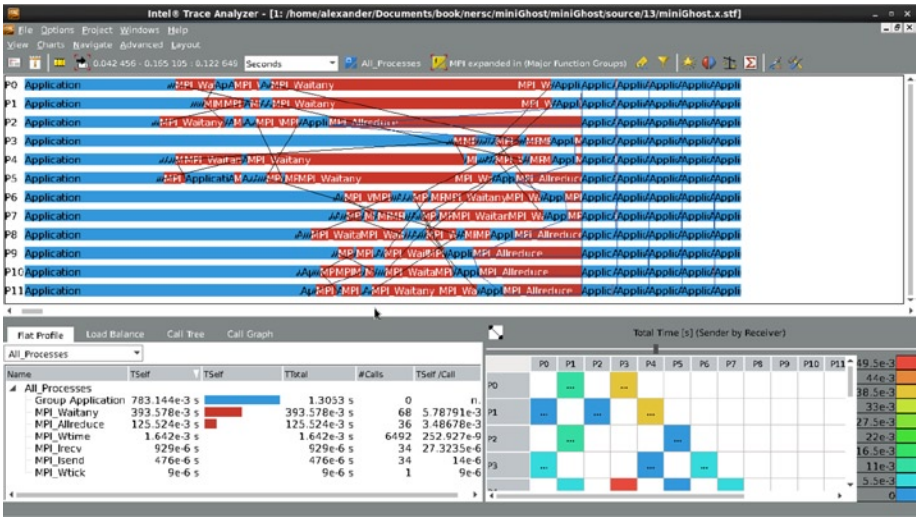
The earlier statistics measurements were done in a queue set up for larger jobs, while the later ITAC measurements used another queue set up for shorter jobs, because the larger queue became overloaded and nothing was moving there, as it usually does under time pressure. This resulted in the later jobs being put onto another part of the cluster, with comparable processors but with a possibly better connectivity. This once again highlights the necessary of keeping your environment unchanged throughout the measurement series, and of doing the runs well ahead of the deadlines.

Load imbalance aside, we may have to deal with the less than optimal process layout (4x4x6) prescribed by the benchmark formulation. Indeed, when we tried other process layouts within the same job manager session, we observed that the communication along the X axis was stumbling—and more so as more MPI processes were placed along it; see Table 5-13:

Table 5-13. *MiniGhost Performance Dependency on the Process Layout (Cluster, 8 Nodes, 96 MPI Processes, 4 OpenMP Threads per Process)*

Layout, XxYxZ	Performance, GFLOPS	Time, Sec
4x4x6	3.69E+03	3.55E+01
1x8x12	3.72E+03	3.52E+01
8x1x12	3.55E+03	3.69E+01
8x12x1	3.41E+03	3.85E+01
1x1x96	3.10E+03	4.23E+01
1x96x1	3.11E+03	4.21E+01
96x1x1	1.72E+03	7.62E+01

Let's try to understand what exactly is happening here. If you view a typical problematic patch of the miniGhost trace file in ITAC, you will notice the following picture replicated many times across the whole event timeline, at various moments and at different time scales, as shown in Figure 5-16.


Intel® Trace Analyzer - [1: /home/alexander/Documents/book/nersc/miniGhost/miniGhost/source/13/miniGhost.x.stf]
File Options Project Windows Help
View Charts Navigate Advanced Layout
0.042 456 - 0.165 105 : 0.122 649
Seconds
All_Processes
MPI expanded in (Major Function Groups)
P0 P1 P2 P3 P4 P5 P6 P7 P8 P9 P10 P11
Application
MPI_Waitany
MPI_Allreduce
Flat Profile
Load Balance
Call Tree
Call Graph
All_Processes
Total Time [s] (Sender by Receiver)

Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
All_Processes					
Group Application	783.144e-3 s		1.3053 s	0	n.
MPI_Waitany	393.578e-3 s		393.578e-3 s	68	5.78791e-3
MPI_Allreduce	125.524e-3 s		125.524e-3 s	36	3.48678e-3
MPI_Wtime	1.642e-3 s		1.642e-3 s	6492	252.927e-9
MPI_Irecv	929e-6 s		929e-6 s	34	27.3235e-6
MPI_Isend	476e-6 s		476e-6 s	34	14e-6
MPI_Wtick	9e-6 s		9e-6 s	1	9e-6

Figure 5-16. *Typical MiniGhost exchange pattern (Workstation, 12 MPI processes, 4 OpenMP threads)*

This patch corresponds to the very first and most expensive exchange during the program execution. Rather small per se, it becomes a burden due to endless replication; all smaller MPI communication segments after this one follow the same pattern or at least have a pretty imbalanced `MPI_Allreduce` inside (not shown). It is clear that the first order of the day is to understand why the `MPI_Waitany` has to work in such irregular circumstances, and then try to correct this. It is also possible that the `MPI_Allreduce` will recover its dignity when acting in a better environment.

By the looks of it, the pattern in Figure 5-16 resembles a typical neighbor exchange implemented by nonblocking MPI calls. Since the very first `MPI_Allreduce` is a representative one, we have no problem identifying where the prior nonblocking exchange comes from: a bit of source code and log file browsing lead us to the file called `MG_UNPACK_BSPMA.F`, where the waiting is done using the `MPI_Waitany` on all `MPI_Request` items filled by the prior calls to `MPI_Isend` and `MPI_Irecv` that indeed represent a neighbor data exchange. In addition to this, as the name of the file suggests and the code review confirms, the data is packed and unpacked using the respective MPI calls. From this, at least three optimization ideas of different complexity emerge:

1. *Relatively easy*: Use the `MPI_Waitall` or `MPI_Waitsome` instead of the fussy `MPI_Waitany`. The former might be able to complete all or at least more than one request per invocation, and do this in the most appropriate order defined by the MPI implementation. However, there is some internal application statistics collection that is geared toward the use of `MPI_Waitany`, so more than just a replacement of one call may be necessary technically.

2. *Relatively hard*: Try to replace the nonblocking exchange with the properly ordered blocking `MPI_Sendrecv` pairs. A code review shows that the exchanges are aligned along the three spatial dimensions, so that a more regular messaging order might actually help smoothe the data flow and reduce the observed level of irregularity. If this sounds too hard, even making sure that all `MPI_Irecv` are posted shortly before the respective `MPI_Isend` might be a good first step.

3. *Probably impossible*: Use the MPI derived datatypes instead of the packing/unpacking. Before this deep modification is attempted, it should be verified that packing/unpacking indeed matters.

This coding exercise is only sensible once the `MPI_Allreduce` issue has been dealt with. For that we need to look into the node-level details in the later chapters of this book, and then return to this issue. This is a good example of the back-and-forth transition between optimization levels. Remember that once you introduce any change, you will have to redo the measurements and verify that the change was indeed beneficial. After that is done, you can repeat this cycle or proceed to the node optimization level we will consider in the following chapters, once we've covered more about advanced MPI analysis techniques (to be continued in Chapter 6).

EXERCISE 5-13

Investigate the miniGhost `MPI_Init` overhead and clarify whether this is a Heisenbug or not. If it is, contact Intel Premier and report the matter.

EXERCISE 5-14

Return here once the `MPI_Allreduce` load imbalance has been dealt with, and implement one of the proposed source code optimizations. Gauge its effect on the miniGhost benchmark, especially at scale. Was it worth the trouble?

Using Advanced Analysis Techniques

We have barely scratched the surface of capabilities offered by the Intel MPI and Intel Trace Analyzer and Collector. This section introduces more advanced features that you may need in your work, but you will have to read more about them before you can start to use them.

Automatically Checking MPI Program Correctness

We started with the premise of a correctly written parallel application. Here's the truth, though: there are none. That is, there are some applications that manifest no apparent errors at the moment. Even as we were writing this book, we detected several errors in candidate programs of various levels of maturity, from our own naïve code snippets to the venerable, internationally recognized, and widely used benchmarks. Some programs would not build, some would not run, some would break on ostensibly valid input data, and so on. This is all a fact of life.

Fortunately, if you use Intel MPI and ITAC, you can mitigate at least some of the risk in trying to optimize an erroneous program. Just add option `-check_mpi` to your application build string or the `mpirun`, run string, and the ITAC correctness checking library will start watching all MPI transfers and checking them for many issues, including incorrect parameters, potential or real deadlocks, race conditions, data corruption, and more.

This may cost quite some a bit at runtime, especially if you ask for the buffer check sums to be computed and verified, or you used the `valgrind` in addition to check the memory access patterns. However, in return you will get at least some of that warm and fuzzy feeling that is otherwise unknown to programmers in general and to parallel programmers in particular; that feeling, though, is the almost certain yet dangerously wrong belief that your program is MPI bug free.

Comparing Application Traces

You have seen how we compared real and ideal traces (see Figure 5-12). Actually, this is a generic feature you can apply to any two traces. While comparing two unrelated traces might be a bit off topic, comparing two closely related traces may reveal interesting things. For example, you can compare two different runs of the same application, done on different process counts or just having substantially different performance characteristics on the same process count. Looking into the traces side by side will help you spot where they differ. This is how to go about it:

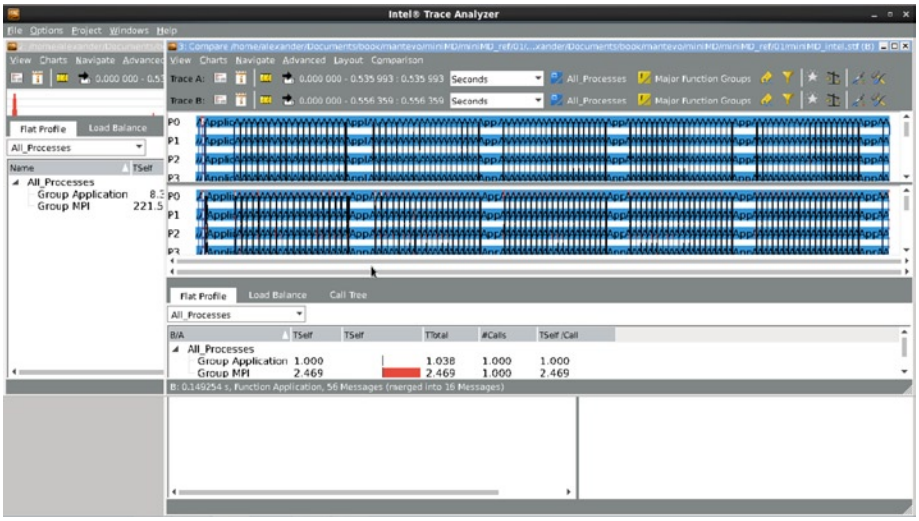
1. Easiest of all, open two trace files you want to compare in one ITAC session when starting up. This is how you can do this:

   ```
   $ traceanalyzer trace1.stf trace2.stf
   ```

2. If you are already in an ITAC session where you have been analyzing a certain trace file, open another file via the global File/Open menu, and then use the File/Compare item.

3. If you want to do this by hand, open the files in any way described above, configure to your liking the charts you want to compare, and then use the global View/Arrange menu item to put them side by side or on top of each other.

For example, if you do any of this for the real and ideal files illustrated in Figure 5-12, you will get the results shown in Figure 5-17.


Intel® Trace Analyzer
File Options Project Windows Help
View Charts Navigate Advanced Layout Comparison
Trace A: 0.000 000 - 0.535 993 : 0.535 993 Seconds All_Processes Major Function Groups
Trace B: 0.000 000 - 0.556 359 : 0.556 359 Seconds All_Processes Major Function Groups
P0
P1
P2
P3
Flat Profile
Load Balance
Call Tree
All_Processes

B/A	TSelf	TSelf	TTotal	#Calls	TSelf /Call
All_Processes					
Group Application	1.000		1.038	1.000	1.000
Group MPI	2.469	🟥	2.469	1.000	2.469

B: 0.149254 s, Function Application, 56 Messages (merged into 16 Messages)

Figure 5-17. *MiniMD real and ideal traces compared side by side (Workstation, 16 MPI processes)*

This view confirms the earlier observation that, although there may be up to 2.5 times improvement to haul in the MPI area, the overall effect on the total program's execution time will be marginal. Another interesting view to observe is the Breakdown Mode in the imbalance diagram shown in Figure 5-18 (here we again changed the default colors to roughly match those in the event timeline).

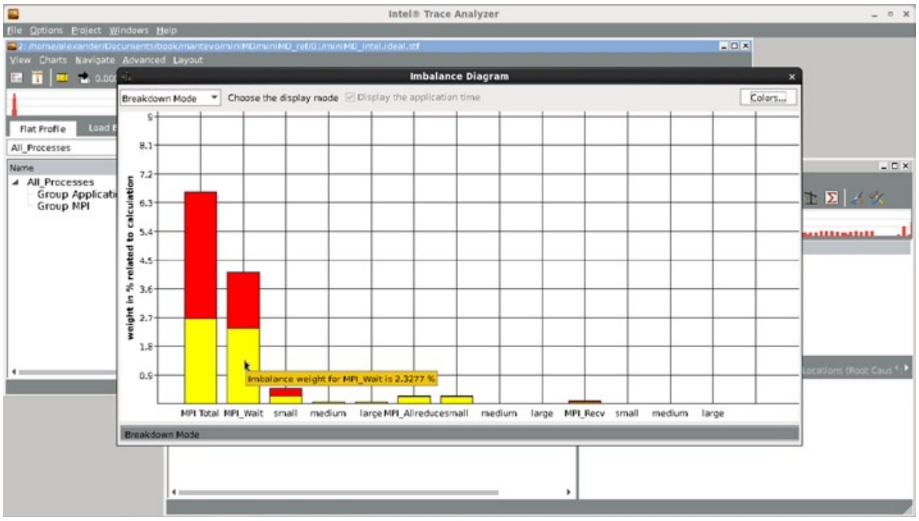


Figure 5-18. *MiniMD trace file in ITAC imbalance diagra breakdown mode (Workstation, 16 MPI processes)*

From this view you can conclude that `MPI_Wait` is probably the call to investigate as far as pure MPI performance is concerned. The rest of the overhead comes from the load imbalance. If you want to learn more about comparing trace files, follow up with the aforementioned serialization tutorial.

Instrumenting Application Code

Once in a while you may want to know exactly what is happening in the user part of the application, rather than just observe the blue Group Application mentioned in the respective ITAC charts. In this case you can use several features provided by Intel tools to get this information:

1. Probably the easiest is to ask Intel compiler do the job for you. Add the compiler option `-tcollect` to get all source code functions instrumented to leave a trace in the ITAC trace file. This option needs to be used both at the compilation and at the linkage steps. If the number of the resulting call tracing events is too high, use the `-tcollect-filter` variety to limit their scope. You may—and probably should—apply these features selectively to those files that interest you most; otherwise, the trace file size may explode. You can find more details in the ITC documentation.[26]

2. If you want complete control and are willing to invest some time, use the ITAC instrumenting interface described in the documentation mentioned above. An instrumentation source code example that comes with the ITAC distribution will be a good starting point here.

You can learn more about these advanced topics and also control the size of the trace file, the latter which is very important if you want to analyze a long running or a highly scalable application, in the *Tutorial: Reducing Trace File Size*.[27]

Correlating MPI and Hardware Events

As a final point before we close the MPI optimization, we give a recommendation on how to correlate the ITAC trace events with the hardware events, including those registered by the Intel VTune Amplifier XE data collection infrastructure.[28] As usual, there is more than one way to do this.

Collecting and Analyzing Hardware Counter Information in ITAC

Believe it or not, you can collect and display a lot of hardware counter information right in the ITAC. Its facilities are not as extensive and automated as those of VTune Amplifier XE; however, they can give you a good first hack at the problem. You can read about this topic in the ITAC documentation. Note that quite a bit of hacking will be required upfront.

Collecting and Analyzing Hardware Counter Information in VTune

If you have no time for hacking, you can choose the normal way. First, you need to launch VTune Amplifier XE. There are, again, several methods to do so:

1. Launch `amplxe-cl` with `mpirun`. For example: Collect 1 result/rank/node from M nodes – M result directories in total:

    ```
    $ mpirun <machine file> -np <N> ... amplxe-cl -collect <analysis
    type> ./your_app
    ```

    Assumptions: N>M and at least 1 rank/node.

2. Collect 2 hotspots on the host ‘hostname’:

    ```
    $ mpirun -host 'hostname' -np 14 ./a.out : \
             -host 'hostname' -np 2 amplxe-cl –r foo -c hotspots
              ./your_app
    ```

3. Launch `mpirun` with `amplxe-cl`. For example: Collect N ranks in one result file on a node (e.g., ‘hostname’):

    ```
    $ amplxe-cl -collect <analysis type> ... -- mpirun -host
    ‘hostname’ -np <N> ./your_app
    ```

Limitation: Currently collects only on the `localhost`.

We will cover the rest of this topic in Chapter 6, but you may want to read the *Tutorial: Analyzing MPI Application with Intel Trace Analyzer and Intel VTune Amplifier XE* as well.[29]

Summary

We presented MPI optimization methodology in this chapter in its application to the Intel MPI Library and Intel Trace Analyzer. However, you can easily reuse this procedure with other tools of your choice.

It is (not so) surprising that the literature on MPI optimization in particular is rather scarce. This was one of our primary reasons for writing this book. To get the most out of it, you need to know quite a bit about the MPI programming. There is probably no better way to get started than by reading the classic *Using MPI* by Bill Gropp, Ewing Lusk, and Anthony Skjellum[30] and *Using MPI-2* by William Gropp, Ewing Lusk, and Rajeev Thakur.[31] If you want to learn more about the Intel Xeon Phi platform, you may want to read *Intel Xeon Phi Coprocessor High-Performance Programming* by Jim Jeffers and James Reinders that we mentioned earlier. Ultimately, nothing will replace reading the MPI standard, asking questions in the respective mailing lists, and getting your hands dirty.

We cannot recommend any specific book on the parallel algorithms because they are quite dependent on the domain area you are going to explore. Most likely, you know all the most important publications and periodicals in that area anyway. Just keep an eye on them; algorithms rule this realm.

References

1. MPI Forum, "MPI Documents," www.mpi-forum.org/docs/docs.html.
2. H. Bockhorst and M. Lubin, "Performance Analysis of a Poisson Solver Using Intel VTune Amplifier XE and Intel Trace Analyzer and Collector," to be published in TBD.
3. Intel Corporation, "Intel MPI Benchmarks," http://software.intel.com/en-us/articles/intel-mpi-benchmarks/.
4. Intel Corporation, "Intel(R) Premier Support," www.intel.com/software/products/support.
5. D. Akin, "Akin's Laws of Spacecraft Design," http://spacecraft.ssl.umd.edu/old_site/academics/akins_laws.html.
6. A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary, "HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers," www.netlib.org/benchmark/hpl/.
7. Intel Corporation, "Intel Math Kernel Library – LINPACK Download," http://software.intel.com/en-us/articles/intel-math-kernel-library-linpack-download.
8. A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary, "HPL FAQs," www.netlib.org/benchmark/hpl/faqs.html.
9. "BLAS (Basic Linear Algebra Subprograms)," www.netlib.org/blas/.
10. Sandia National Laboratory, "HPCG - Home," https://software.sandia.gov/hpcg/.
11. "Home of the Mantevo project," http://mantevo.org/.
12. Intel Corporation, "Configuring Intel Trace Collector," https://software.intel.com/de-de/node/508066.
13. Sandia National Laboratory, "LAMMPS Molecular Dynamics Simulator," http://lammps.sandia.gov/.
14. Ohio State University, "OSU Micro-Benchmarks," http://mvapich.cse.ohio-state.edu/benchmarks/.
15. Intel Corporation, "Intel MPI Library - Documentation," https://software.intel.com/en-us/articles/intel-mpi-library-documentation.
16. J. Jeffers and J. Reinders, *Intel Xeon Phi Coprocessor High-Performance Programming* (Waltham, MA: Morgan Kaufman Publ. Inc., 2013).
17. "MiniGhost," www.nersc.gov/users/computational-systems/nersc-8-system-cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/minighost/.
18. Intel Corporation, "Tutorial: MPI Tuner for Intel MPI Library for Linux OS," https://software.intel.com/en-us/mpi-tuner-tutorial-lin-5.0-pdf.

19. M. Chuvelev, “Collective Algorithm Models,” Intel Corporation, Internal technical report, 2013.

20. Intel Corporation, “Tutorial: Detecting and Removing Unnecessary Serialization,” https://software.intel.com/en-us/itac_9.0_serialization_pdf.

21. A. Supalov and A. Yalozo, “20 Years of the MPI Standard: Now With a Common Application Binary Interface,” *The Parallel Universe* 18, no. 1 (2014): 28–32.

22. M. Brinskiy, A. Supalov, M. Chuvelev, and E. Leksikov, “Mastering Performance Challenges with the New MPI-3 Standard,” *The Parallel Universe* 18, no. 1 (2014): 33–40.

23. “PanFS Storage Operating System,” www.panasas.com/products/panfs.

24. “Parallel Virtual File System, Version 2,” www.pvfs.org/.

25. “Lustre - OpenSFS,” http://lustre.opensfs.org/.

26. Intel Corporation, “Intel Trace Analyzer and Collector - Documentation,” https://software.intel.com/en-us/articles/intel-trace-analyzer-and-collector-documentation.

27. Intel Corporation, “Tutorial: Reducing Trace File Size,” https://software.intel.com/en-us/itac_9.0_reducing_trace_pdf.

28. Intel Corporation, “Intel VTune Amplifier XE 2013,” https://software.intel.com/en-us/intel-vtune-amplifier-xe.

29. Intel Corporation, “Tutorial: Analyzing MPI Application with Intel Trace Analyzer and Intel VTune Amplifier XE,” https://software.intel.com/en-us/itac_9.0_analyzing_app_pdf.

30. W. Gropp, E. L. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interf*ace, 2nd. ed. (Cambridge, MA: MIT Press, 1999).

31. W. Gropp, E. L. Lusk, and R. Thakur, *Using MPI-2: Advanced Features of the Message Passing Interface* (Cambridge, MA: MIT Press, 1999).