CHAPTER 1

No Time to Read This Book?

We know what it feels like to be under pressure. Try out a few quick and proven optimization stunts described below. They may provide a good enough performance gain right away.

There are several parameters that can be adjusted with relative ease. Here are the steps we follow when hard pressed:

- Use Intel MPI Library[1] and Intel Composer XE[2]
- Got more time? Tune Intel MPI:
  - Collect built-in statistics data
  - Tune Intel MPI process placement and pinning
  - Tune OpenMP thread pinning
- Got still more time? Tune Intel Composer XE:
  - Analyze optimization and vectorization reports
  - Use interprocedural optimization

Using Intel MPI Library

The Intel MPI Library delivers good out-of-the-box performance for bandwidth-bound applications. If your application belongs to this popular class, you should feel the difference immediately when switching over.

If your application has been built for Intel MPI compatible distributions like MPICH,[3] MVAPICH2,[4] or IBM POE,[5] and some others, there is no need to recompile the application. You can switch by dynamically linking the Intel MPI 5.0 libraries at runtime:

```
$ source /opt/intel/impi_latest/bin64/mpivars.sh
$ mpirun -np 16 -ppn 2 xhpl
```

If you use another MPI and have access to the application source code, you can rebuild your application using Intel MPI compiler scripts:

- Use `mpicc` (for C), `mpicxx` (for C++), and `mpifc`/`mpif77`/`mpif90` (for Fortran) if you target GNU compilers.
- Use `mpiicc`, `mpiicpc`, and `mpiifort` if you target Intel Composer XE.

Using Intel Composer XE

The invocation of the Intel Composer XE is largely compatible with the widely used GNU Compiler Collection (GCC). This includes both the most commonly used command line options and the language support for C/C++ and Fortran. For many applications you can simply replace `gcc` with `icc`, `g++` with `icpc`, and `gfortran` with `ifort`. However, be aware that although the binary code generated by Intel C/C++ Composer XE is compatible with the GCC-built executable code, the binary code generated by the Intel Fortran Composer is not.

For example:

```
$ source /opt/intel/composerxe/bin/compilervars.sh intel64
$ icc -O3 -xHost -qopenmp  -c example.o example.c
```

Revisit the compiler flags you used before the switch; you may have to remove some of them. Make sure that Intel Composer XE is invoked with the flags that give the best performance for your application (see Table 1-1). More information can be found in the Intel Composer XE documentation.[6]

Table 1-1. *Selected Intel Composer XE Optimization Flags*

GCC	ICC	Effect
`-O0`	`-O0`	Disable (almost all) optimization. Not something you want to use for performance!
`-O1`	`-O1`	Optimize for speed (no code size increase for ICC)
`-O2`	`-O2`	Optimize for speed and enable vectorization
`-O3`	`-O3`	Turn on high-level optimizations
`-ftlo`	`-ipo`	Enable interprocedural optimization
`-ftree-vectorize`	`-vec`	Enable auto-vectorization (auto-enabled with `-O2` and `-O3`)
`-fprofile-generate`	`-prof-gen`	Generate runtime profile for optimization
`-fprofile-use`	`-prof-use`	Use runtime profile for optimization
	`-parallel`	Enable auto-parallelization
`-fopenmp`	`-qopenmp`	Enable OpenMP
`-g`	`-g`	Emit debugging symbols
	`-qopt-report`	Generate the optimization report
	`-vec-report`	Generate the vectorization report
	`-ansi-alias`	Enable ANSI aliasing rules for C/C++

(*continued*)

Table 1-1. (*continued*)

GCC	ICC	Effect
`-msse4.1`	`-xSSE4.1`	Generate code for Intel processors with SSE 4.1 instructions
`-mavx`	`-xAVX`	Generate code for Intel processors with AVX instructions
`-mavx2`	`-xCORE-AVX2`	Generate code for Intel processors with AVX2 instructions
`-mcpu=native`	`-xHost`	Generate code for the current machine used for compilation

For most applications, the default optimization level of `-O2` will suffice. It runs fast and gives reasonable performance. If you feel adventurous, try `-O3`. It is more aggressive but it also increases the compilation time.

Tuning Intel MPI Library

If you have more time, you can try to tune Intel MPI parameters without changing the application source code.

Gather Built-in Statistics

Intel MPI comes with a built-in statistics-gathering mechanism. It creates a negligible runtime overhead and reports key performance metrics (for example, MPI to computation ratio, message sizes, counts, and collective operations used) in the popular IPM format.[7]

To switch the IPM statistics gathering mode on and do the measurements, enter the following commands:

```
$ export I_MPI_STATS=ipm
$ mpirun -np 16 xhpl
```

By default, this will generate a file called `stats.ipm`. Listing 1-1 shows an example of the MPI statistics gathered for the well-known High Performance Linpack (HPL) benchmark.[8] (We will return to this benchmark throughout this book, by the way.)

Listing 1-1. MPI Statistics for the HPL Benchmark with the Most Interesting Fields Highlighted

```
Intel(R) MPI Library Version 5.0

Summary MPI Statistics
Stats format: region
Stats scope : full

##############################################################################
#
# command : /home/book/hpl/./xhpl_hybrid_intel64_dynamic (completed)
# host    : esg066/x86_64_Linux             mpi_tasks : 16 on 8 nodes
# start   : 02/14/14/12:43:33               wallclock : 2502.401419 sec
# stop    : 02/14/14/13:25:16               %comm     : 8.43
# gbytes  : 0.00000e+00 total               gflop/sec : NA
#
##############################################################################
# region  : *   [ntasks] = 16
#
#                       [total]       <avg>         min           max
# entries               16            1             1             1
# wallclock             40034.7       2502.17       2502.13       2502.4
# user                  446800        27925         27768.4       28192.7
# system                1971.27       123.205       102.103       145.241
# mpi                   3375.05       210.941       132.327       282.462
# %comm                               8.43032       5.28855       11.2888
# gflop/sec             NA            NA            NA            NA
# gbytes                0             0             0             0
#
#
#                       [time]        [calls]       <%mpi>        <%wall>
# MPI_Send              2737.24       1.93777e+06   81.10         6.84
# MPI_Recv              394.827       16919         11.70         0.99
# MPI_Wait              236.568       1.92085e+06   7.01          0.59
# MPI_Iprobe            3.2257        6.57506e+06   0.10          0.01
# MPI_Init_thread       1.55628       16            0.05          0.00
# MPI_Irecv             1.31957       1.92085e+06   0.04          0.00
# MPI_Type_commit       0.212124      14720         0.01          0.00
# MPI_Type_free         0.0963376     14720         0.00          0.00
# MPI_Comm_split        0.0065608     48            0.00          0.00
# MPI_Comm_free         0.000276804   48            0.00          0.00
# MPI_Wtime             9.67979e-05   48            0.00          0.00
# MPI_Comm_size         9.13143e-05   452           0.00          0.00
# MPI_Comm_rank         7.77245e-05   452           0.00          0.00
# MPI_Finalize          6.91414e-06   16            0.00          0.00
# MPI_TOTAL             3375.05       1.2402e+07    100.00        8.43
##############################################################################
```

From Listing 1-1 you can deduce that MPI communication occupies between 5.3 and 11.3 percent of the total runtime, and that the `MPI_Send`, `MPI_Recv`, and `MPI_Wait` operations take about 81, 12, and 7 percent, respectively, of the total MPI time. With this data at hand, you can see that there are potential load imbalances between the job processes, and that you should focus on making the `MPI_Send` operation as fast as it can go to achieve a noticeable performance hike.

Note that if you use the full IPM package instead of the built-in statistics, you will also get data on the total communication volume and floating point performance that are not measured by the Intel MPI Library.

Optimize Process Placement

The Intel MPI Library puts adjacent MPI ranks on one cluster node as long as there are cores to occupy. Use the Intel MPI command line argument `-ppn` to control the process placement across the cluster nodes. For example, this command will start two processes per node:

```
$ mpirun -np 16 -ppn 2 xhpl
```

Intel MPI supports process pinning to restrict the MPI ranks to parts of the system so as to optimize process layout (for example, to avoid NUMA effects or to reduce latency to the InfiniBand adapter). Many relevant settings are described in the *Intel MPI Library Reference Manual*.[9]

Briefly, if you want to run a pure MPI program only on the physical processor cores, enter the following commands:

```
$ export I_MPI_PIN_PROCESSOR_LIST=allcores
$ mpirun -np 2 your_MPI_app
```

If you want to run a hybrid MPI/OpenMP program, don't change the default Intel MPI settings, and see the next section for the OpenMP ones.

If you want to analyze Intel MPI process layout and pinning, set the following environment variable:

```
$ export I_MPI_DEBUG=4
```

Optimize Thread Placement

If the application uses OpenMP for multithreading, you may want to control thread placement in addition to the process placement. Two possible strategies are:

```
$ export KMP_AFFINITY=granularity=thread,compact
$ export KMP_AFFINITY=granularity=thread,scatter
```

The first setting keeps threads close together to improve inter-thread communication, while the second setting distributes the threads across the system to maximize memory bandwidth.

Programs that use the OpenMP API version 4.0 can use the equivalent OpenMP affinity settings instead of the KMP_AFFINITY environment variable:

```
$ export OMP_PROC_BIND=close
$ export OMP_PROC_BIND=spread
```

If you use `I_MPI_PIN_DOMAIN`, MPI will confine the OpenMP threads of an MPI process on a single socket. Then you can use the following setting to avoid thread movement between the logical cores of the socket:

```
$ export KMP_AFFINITY=granularity=thread
```

Tuning Intel Composer XE

If you have access to the source code of the application, you can perform optimizations by selecting appropriate compiler switches and recompiling the source code.

Analyze Optimization and Vectorization Reports

Add compiler flags `-qopt-report` and/or `-vec-report` to see what the compiler did to your source code. This will report all the transformations applied to your code. It will also highlight those code patterns that prevented successful optimization. Address them if you have time left.

Here is a small example. Because the optimization report may be very long, Listing 1-2 only shows an excerpt from it. The example code contains several loop nests of seven loops. The compiler found an OpenMP directive to parallelize the loop nest. It also recognized that the overall loop nest was not optimal, and it automatically permuted some loops to improve the situation for vectorization. Then it vectorized all inner-most loops while leaving the outer-most loops as they are.

Listing 1-2. Example Optimization Report with the Most Interesting Fields Highlighted

```
$ ifort -O3 -qopenmp -qopt-report -qopt-report-file=stdout -c example.F90

    Report from: Interprocedural optimizations [ipo]

[...]

OpenMP Construct at example.F90(8,7)
remark #15059: OpenMP DEFINED LOOP WAS PARALLELIZED
OpenMP Construct at example.F90(25,7)
remark #15059: OpenMP DEFINED LOOP WAS PARALLELIZED

[...]
```

```
LOOP BEGIN at example.F90(9,2)
   remark #15018: loop was not vectorized: not inner loop

   LOOP BEGIN at example.F90(12,5)
      remark #25448: Loopnest Interchanged : ( 1 2 3 4 ) --> ( 1 4 2 3 )
      remark #15018: loop was not vectorized: not inner loop

      LOOP BEGIN at example.F90(12,5)
         remark #15018: loop was not vectorized: not inner loop

[...]

                  LOOP BEGIN at example.F90(15,8)
                     remark #25446: blocked by 125   (pre-vector)
                     remark #25444: unrolled and jammed by 4   (pre-vector)
                     remark #15018: loop was not vectorized: not inner loop

                     LOOP BEGIN at example.F90(13,6)
                        remark #25446: blocked by 125   (pre-vector)
                        remark #15018: loop was not vectorized: not inner loop

                        LOOP BEGIN at example.F90(14,7)
                           remark #25446: blocked by 128   (pre-vector)
                           remark #15003: PERMUTED LOOP WAS VECTORIZED
                        LOOP END

                        LOOP BEGIN at example.F90(14,7)
                        Remainder
                           remark #25460: Loop was not optimized
                        LOOP END
                     LOOP END
                  LOOP END

[...]

            LOOP END
         LOOP END
      LOOP END
   LOOP END
LOOP END

LOOP BEGIN at example.F90(26,2)
   remark #15018: loop was not vectorized: not inner loop

   LOOP BEGIN at example.F90(29,5)
      remark #25448: Loopnest Interchanged : ( 1 2 3 4 ) --> ( 1 3 2 4 )
      remark #15018: loop was not vectorized: not inner loop
```

```
      LOOP BEGIN at example.F90(29,5)
         remark #15018: loop was not vectorized: not inner loop

         LOOP BEGIN at example.F90(29,5)
            remark #15018: loop was not vectorized: not inner loop

            LOOP BEGIN at example.F90(29,5)
               remark #15018: loop was not vectorized: not inner loop

               LOOP BEGIN at example.F90(29,5)
                  remark #25446: blocked by 125   (pre-vector)
                  remark #25444: unrolled and jammed by 4   (pre-vector)
                  remark #15018: loop was not vectorized: not inner loop

[...]
               LOOP END
            LOOP END
         LOOP END
      LOOP END
   LOOP END
LOOP END
```

Listing 1-3 shows the vectorization report for the example in Listing 1-2. As you can see, the vectorization report contains the same information about vectorization as the optimization report.

Listing 1-3. Example Vectorization Report with the Most Interesting Fields Highlighted

```
$ ifort -O3 -qopenmp -vec-report=2 -qopt-report-file=stdout -c example.F90

[...]

LOOP BEGIN at example.F90(9,2)
   remark #15018: loop was not vectorized: not inner loop

   LOOP BEGIN at example.F90(12,5)
      remark #15018: loop was not vectorized: not inner loop

      LOOP BEGIN at example.F90(12,5)
         remark #15018: loop was not vectorized: not inner loop

         LOOP BEGIN at example.F90(12,5)
            remark #15018: loop was not vectorized: not inner loop

            LOOP BEGIN at example.F90(12,5)
               remark #15018: loop was not vectorized: not inner loop
```

```
                LOOP BEGIN at example.F90(12,5)
                   remark #15018: loop was not vectorized: not inner loop

                   LOOP BEGIN at example.F90(15,8)
                      remark #15018: loop was not vectorized: not inner loop

                      LOOP BEGIN at example.F90(13,6)
                         remark #15018: loop was not vectorized: not inner loop

                         LOOP BEGIN at example.F90(14,7)
                            remark #15003: PERMUTED LOOP WAS VECTORIZED
                         LOOP END

[...]

                      LOOP END
                   LOOP END

                   LOOP BEGIN at example.F90(15,8)
                   Remainder
                      remark #15018: loop was not vectorized: not inner loop

                      LOOP BEGIN at example.F90(13,6)
                         remark #15018: loop was not vectorized: not inner loop

[...]

                         LOOP BEGIN at example.F90(14,7)
                            remark #15003: PERMUTED LOOP WAS VECTORIZED
                         LOOP END

[...]

                      LOOP END
                   LOOP END
                LOOP END

[...]

             LOOP END
          LOOP END
       LOOP END
    LOOP END
LOOP END

[...]
```

Use Interprocedural Optimization

Add the compiler flag `-ipo` to switch on interprocedural optimization. This will give the compiler a holistic view of the program and open more optimization opportunities for the program as a whole. Note that this will also increase the overall compilation time.

Runtime profiling can also increase the chances for the compiler to generate better code. Profile-guided optimization requires a three-stage process. First, compile the application with the compiler flag `-prof-gen` to instrument the application with profiling code. Second, run the instrumented application with a typical dataset to produce a meaningful profile. Third, feed the compiler with the profile (`-prof-use`) and let it optimize the code.

Summary

Switching to Intel MPI and Intel Composer XE can help improve performance because the two strive to optimally support Intel platforms and deliver good out-of-the-box (OOB) performance. Tuning measures can further improve the situation. The next chapters will reiterate the quick and dirty examples of this chapter and show you how to push the limits.

References

1. Intel Corporation, “Intel(R) MPI Library,” http://software.intel.com/en-us/intel-mpi-library.
2. Intel Corporation, “Intel(R) Composer XE Suites,” http://software.intel.com/en-us/intel-composer-xe.
3. Argonne National Laboratory, “MPICH: High-Performance Portable MPI,” www.mpich.org.
4. Ohio State University, “MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE,” http://mvapich.cse.ohio-state.edu/overview/mvapich2/.
5. International Business Machines Corporation, “IBM Parallel Environment,” www-03.ibm.com/systems/software/parallel/.
6. Intel Corporation, “Intel Fortran Composer XE 2013 - Documentation,” http://software.intel.com/articles/intel-fortran-composer-xe-documentation/.
7. The IPM Developers, “Integrated Performance Monitoring - IPM,” http://ipm-hpc.sourceforge.net/.
8. A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary, “HPL : A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers,” 10 September 2008, www.netlib.org/benchmark/hpl/.
9. Intel Corporation, “Intel MPI Library Reference Manual,” http://software.intel.com/en-us/node/500285.