CHAPTER 6

Addressing Application Bottlenecks: Shared Memory

The previous chapters talked about the potential bottlenecks in your application and the system it runs on. In this chapter, we will have a close look at how the application code performs on the level of an individual cluster node. It is a fair assumption that there will also be bottlenecks on this level. Removing these bottlenecks will usually translate directly to increased performance, in addition to the optimizations discussed in the previous chapters.

In line with our top-down strategy, we will investigate how to improve your application code on the threading level. On this level, you will find several potential bottlenecks that can dramatically affect the performance of your application code; some of them are hardware related, some of them are related to your algorithm. The bottlenecks we discuss all come down to how the threads of your code interact with the underlying hardware. From the past chapters you already have an understanding of how this hardware works and what the important metrics and optimization goals are.

We will start with an introduction that covers how to apply Intel VTune Amplifier XE and a loop profiler to your application to gain a better understanding of the code's execution profile. The next topic is that of detecting sequential execution and load imbalances. Then, we will investigate how thread synchronization may affect the performance of the application code.

Profiling Your Application

Profiling the code is the first step toward gaining an understanding of what parts of your application are critical. As usual there are several options for performing this profiling and each option provides different insights into your application and the code it executes. Information of particular interest here is how much time the application spends in each part of the code. This analysis is useful because of the two insights it provides:

1. You get a detailed breakdown of the application runtime.
2. It tells you exactly what the points of interest are for code optimizations.

During the optimization work you will focus on the so-called hotspots that contribute most to the application runtime, because improving their performance will be most beneficial to overall runtime.

You have already seen a tool called PowerTOP in Chapter 4 that gives insight into what is currently running on the system. However, it does not show what exactly the running applications are executing. That is what the Linux tool suite `perf` is for.[1] It contains several tools to record and show performance data. One useful command is `perf top`, which continuously presents the currently active processes and the function they are currently executing. Figure 6-1 shows how the output of the interactive tool might look for a run of the HPCG benchmark.[2] The first column indicates what percentage of CPU time the function (listed in column 4 of a line) has consumed since the last update of the output. The second column shows in which process or shared library image the function is located. The `perf` tool also supports the recording of performance data and analyzing it offline with a command-line interface. Have a look at its documentation for a more detailed explanation.

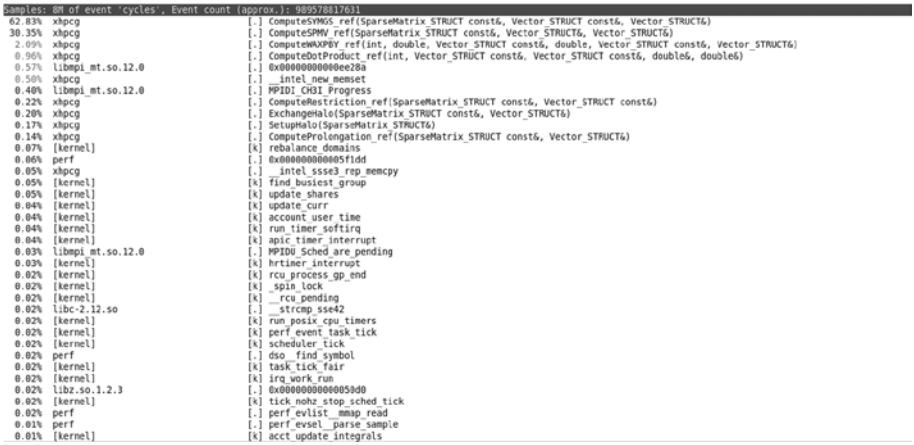


Figure 6-1. *Output of the `perf top` commmand with functions active in the HPCG application*

Although `perf` is a good start to monitor an application while it is running, most of the performance analysis needs to be done postmortem (i.e., after the application executed and performance data was collected). In this way it is possible to inspect the performance data and focus on a particular performance aspect or code region, without having to run the application all the time. This sets the stage for more visual and more powerful tools like Intel VTune Amplifier XE.

Using VTune Amplifier XE for Hotspots Profiling

Intel VTune Amplifier XE provides a unified graphical user interface (GUI) that supports the collection and analysis of performance data. It helps you configure the data collector and set up the application for a collection run. After the collection, you can then work with

the data. VTune Amplifier XE supports both event-based sampling using the processor's built-in performance monitoring units (PMUs) and sampling based on instrumentation of the binary code. In contrast to the Intel Trace Analyzer and Collector (see Chapter 5), the focus of VTune Amplifier XE is on shared-memory and intra-node analysis. The performance data is associated with the source code at all times, so you can easily determine which source line of the application contributed to the performance data.

The most important place to start is with the hotspots analysis to dissect the compute time of the application and relate that information to the application code. This gives a good overview of where the application spends the most compute time. The individual hotspots will be the focus areas of the optimization work to get the biggest bang for the buck. As a side benefit, the hotspots analysis also provides a first insight into how well the code executes on the machine. (We revisit this topic in Chapter 7.)

Hotspots for the HPCG Benchmark

As a first example, let's have a look at the HPCG benchmark. For educational purposes, we pretend that HPCG is an MPI-only code by compiling HPCG without OpenMP. We then try to identify OpenMP candidate loops to add multithreading to the code to make our assumed MPI-only a hybrid MPI/OpenMP code. Of course, in reality the OpenMP directives are already in the code, so we can double-check if we came to same parallelization strategy as the authors of HPCG.

It is a fair assumption that HPC codes are loopy codes that process bulk data in several key loops that will consume most of the compute time. Hence, we need to get a better understanding of the application code by looking at where the code spends time and how this time is spent in the hotspots. We also need to check if the time is spent in loop structures. To do that, we configure an analysis project in the VTune Amplifier XE GUI and run the following command in VTune Amplifier XE using the *Advanced Hotspot* method:

```
$ mpirun -np 48 ./hpcg.x
```

■ **Note** On most clusters it may not be possible to run the GUI. VTune Amplifier XE also supports data collection and analysis on remote systems and from the command line. If *Remote (SSH)* collection is selected in the project configuration, you can add the hostname and credentials for a remote system. You can also use the *Get Command Line* button in the GUI to get a command line that is ready for cut-and-paste to the cluster console or job script. After the collection has finished, you can copy the resulting data to your local machine for analysis within the GUI. For a command-line analysis, you do not need to create a project. You will see examples of how to use this feature later on in this section. You can find out more about collecting performance data and analyzing it with the command-line interface in the VTune Amplifier XE user's guide.[3]

Running this on our example machine gives us the result shown in Figure 6-2. The code executed for 383 seconds and consumed about 18,330 seconds of CPU time, out of which 10,991 seconds (almost 60 percent) is attributed to execution of a function called `ComputeSYMGS_ref`. Function `ComputeSPMV_ref` contributes another 5,538 seconds (30 percent) to the compute time. That makes up about 90 percent of the total CPU compute time. Thus, these two functions will be of interest when we're looking for optimization opportunities.

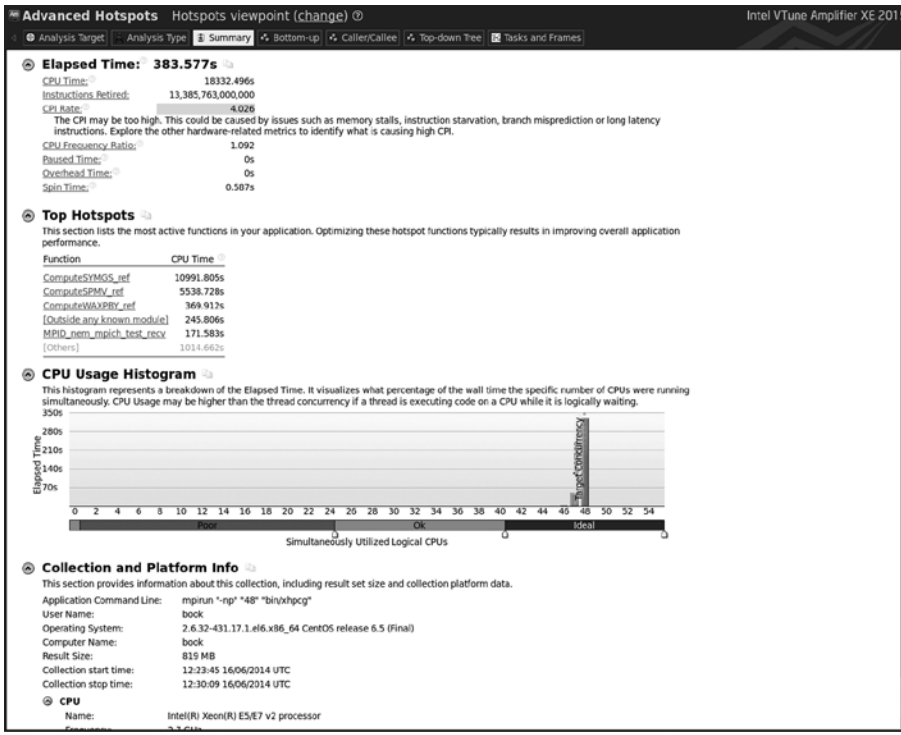


Figure 6-2. *Hotspot and parallelism summary of the HPCG benchmark*

So, the next step is to dig deeper into these functions to find out more about what they do and how they do it. We click on one of the hotspots or the *Bottom-up* button and VTune will show a screen similar to the one in Figure 6-3. Here all relevant functions are shown in more detail, together with their relevant execution time, their containing module (i.e., executable file, shared object, etc.), and the call stack that leads to the invocation of a hotspot. Of course, we will find our two suspect functions listed first and second, as in the Summary screen. As we are interested in finding out more about the hotspot, we change the filter to the *Loops and Functions* mode to let the tool also show hot loops. You can enable this mode by changing the *Loop Mode* filter to *Loops and Functions* in the filter area at the bottom of the GUI.

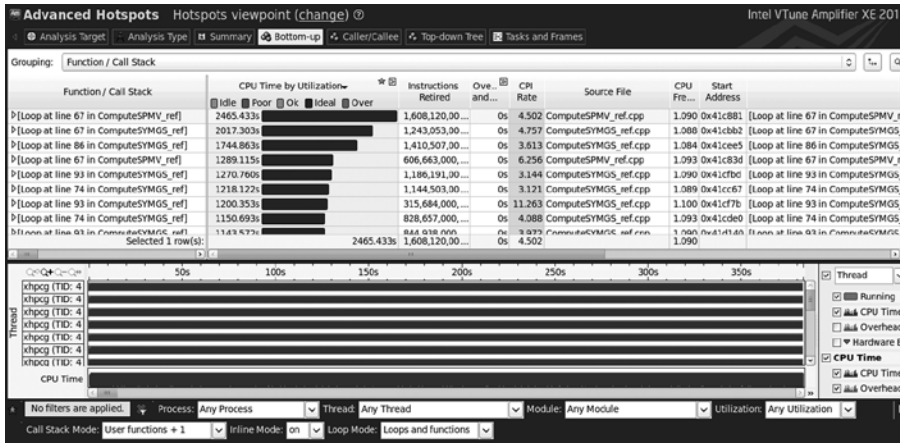


Figure 6-3. *Hotspots (loops and functions) for the HPCG benchmark*

You might be surprised to see that the order of the hotspots now seems to have changed. The functions `ComputeSYMGS_ref` and `ComputeSPMV_ref` are now at the tail of the ranking, which can be seen by scrolling down to the bottom of the upper pane of the screen shot in Figure 6-3. The new top hotspots are loops at several locations in these functions. The hottest loop is at line 67 in the function `ComputeSPMV_ref` and consumes 13 percent of the total compute time. This is a good candidate for parallelization, isn't it? We cannot tell without reading the source code, so we open the source code of the loop by double-clicking the line noting this loop within the VTune Amplifier XE GUI. Listing 6-1 shows the pertinent code of this hotspot.

Listing 6-1. Top Hotspot of the HPCG Benchmark

```
61   for (local_int_t i=0; i< nrow; i++) {
62     double sum = 0.0;
63     const double * const cur_vals = A.matrixValues[i];
64     const local_int_t * const cur_inds = A.mtxIndL[i];
65     const int cur_nnz = A.nonzerosInRow[i];
66
67     for (int j=0; j< cur_nnz; j++)
68       sum += cur_vals[j]*xv[cur_inds[j]];
69     yv[i] = sum;
70   }
```

As you can see, the code consists of two nested loops. VTune Amplifier XE identified the inner loop as the hotspot. Which loop should we select as the target for OpenMP parallelization? In this case, as in many others, the solution will be to parallelize the outer loop. But how do we know how many iterations these loops are executing?

Compiler-Assisted Loop/Function Profiling

Unfortunately, the hotspot analysis does not provide all the data that might be important to make sound decisions for our optimization work. Knowing about the CPU time for a particular hotspot only indicates how much time the code has spent there. It does not tell us how many times a hotspot or parts of it have been executed. For a loop hotspot that we consider for optimization, it will be important to know how many times the loop structure has been encountered from the surrounding code. In addition, we will be interested in the trip count of the loop—that is, how many iterations it executes. The minimum, maximum, and average number of trips through the loop suggest whether a loop might be amenable for certain optimizations, such as parallelization through OpenMP constructs. Hence, we need to complement the hotspots analysis with additional profiling to make sure we have all these bits of information ready to make an informed decision for optimizing the code.

Intel Composer XE ships with a compiler-assisted function and loop profiler that supplies the information we are interested in. To make use of these features requires a recompilation of the code with special command-line flags to augment the compiled code with code to monitor function calls and loop execution at runtime. The profiling can be enabled through the command-line arguments `-profile-functions`, `-profile-loops`, and `-profile-loops-report`. For example, the new command line to compile the HPCG benchmark might start with:

```
$ icc -profile-functions -profile-loops=all -profile-loops-report=2 ...
```

With these settings, the application will record runtime information for functions and loops, including trip counts for all loops. There are several caveats to keep in mind when using this feature, though. First, it only works with single-threaded, single-process applications. Second, it may add considerable overhead to the runtime of the application. The penalty depends on the code structure; many fine-grained functions and loops in the code will add more overhead than fewer large functions and loops. To reduce the overhead, you may try one or more of the command options listed in Table 6-1.

Table 6-1. *Additional Command-Line Options for the Compiler-Assisted Profiler*

Flag	Effect
`-profile-loops=inner`	Only profile inner loops
`-profile-loops=outer`	Only profile outer loops
`-profile-loops-report=1`	Report execution of loops, but no trip count

The loop profile for the HPCG example is given in Figure 6-4. When we compare Figure 6-4 with the hotspot profile shown in Figure 6-3, we can see that the hotspots and the loop profile do not match. This is no surprise; the loop profile was collected in single-rank mode—that is, with only one MPI process executing. In addition, a loop with a small trip count can exceed loops with large numbers of iterations if the loop body is large and demands a lot of compute time. Nevertheless, the loop profile contains an accurate itemization of the loops and their trip counts.

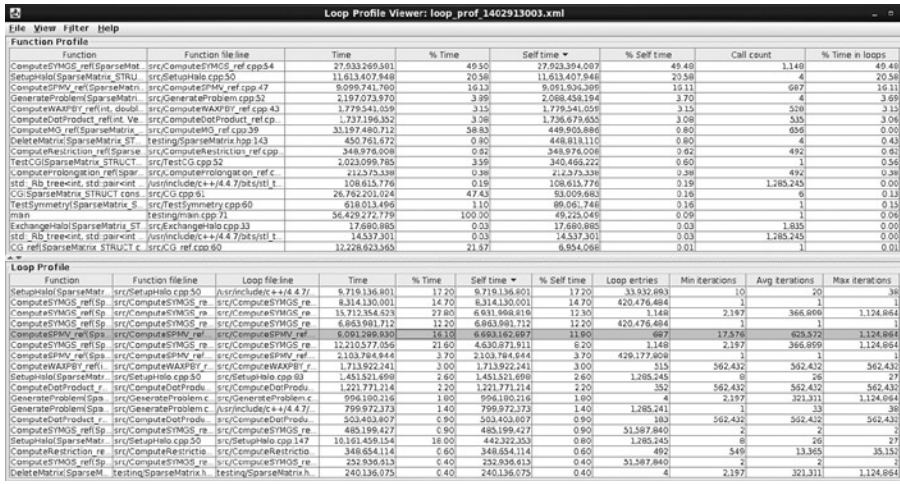


Figure 6-4. *Function and loop profile for the HPCG benchmark*

With the loop hotspots and the loop profile, we can now make an informed decision about which of the two loops in `ComputeSPMV_ref` to parallelize. The hotspot analysis told us that the inner loop is the hot loop. However, the loop profile tells us that the loop in line 67 has been encountered 429 million times with a minimum and maximum trip count of 1. It is easy to see that any parallelization would have done a very poor job on this loop. But there is also the highlighted outer loop showing up in the loop profile. It has been encountered 687 times with minimum and maximum trip count of 17,576 and 1.1 million, respectively. Also, the average trip count of about 625,572 iterations tells us that this loop will be an interesting candidate for parallelization. Of course, one still needs to check that there are no loop dependencies that would prevent parallelization. Inspecting the loop body, we can see that this loop can be executed in parallel. Although it is a good idea to check for loop-carried dependencies and data dependencies (Chapter 7) instead of blindly adding OpenMP parallelization pragmas to loops, tools such as Intel Inspector XE[4] or Valgrind[5] can be a great help in detecting and resolving issues introduced by multithreading.

EXERCISE 6-1

Run a hotspot analysis for your application(s) and determine the minimum, maximum, and average trip counts of its loops. Can you find candidates for parallelization?

Sequential Code and Detecting Load Imbalances

In a parallel program, the slowest thread determines the speed of the whole team working in parallel. All the faster threads will have to wait until the slowest straggler thread catches up and finishes its tasks. As a matter of fact, one of the challenges of parallel programming is that of ensuring all threads receive an equal share of the computational load. Please note that by "computational load" we are referring to the total number of cycles spent per thread for the parallel work. For instance, if the loop body takes a different amount of time to execute different iterations, the threads should not receive equal shares of the loop's iterations (for example, through static scheduling). *Sequential* portions of your application can be seen as a special form of load imbalance, as other threads and cores will be idle while the sequential code is executing in the master thread of the application.

The hotspots analysis for a particular parallel region of code in your application is a useful tool for detecting such load imbalances. VTune Amplifier XE indicates such problems through various elements in the analysis GUI. First, the tabular view (or *grid*) at the top contains, in column "User Time by Utilization," a color code to visualize the quality of parallel execution relative to the number of cores in the system. Red indicates that the hotspot was not using the machine properly and exposes too low an average degree of parallelism. Yellow stands for medium, whereas green suggests an ideal parallel execution. These color codes should not be taken as the only source of information, though; red or yellow hotspots always need closer investigation. Although load imbalances typically show up as a lower degree of parallelism, the red and yellow color codes can also be indicating too low a number of threads executing in parallel, owing to locks, lower number of threads requested, or sequential regions in the hotspots. In case you deliberately execute the application with a lower target thread count (for example, only the physical cores of a system with Intel Hyper-Threading Technology enabled), you can manually adjust the intervals for green, yellow, and red in the Summary tab of the VTune Amplifier XE GUI.

The second GUI element that uses color coding as a visual guide to performance data is the timeline view in the bottom part of the GUI. VTune Amplifier XE shows a horizontal bar for each of the threads in the application and provides insights into their behavior over time. A non-active thread is marked as light green, but once it consumes cycles its color turns to brown. The red color signals overhead, such as time spent waiting for a lock to be released or threads waiting to join a barrier. A load imbalance can easily be detected by looking at when the threads start and stop executing instructions compared to other threads of an OpenMP region, which are indicated by brackets at the top of the thread timeline.

Figure 6-5 shows the performance data and timeline that we collected for a run of the MiniFE application. We used the following command line to produce the performance data on a single node (eight MPI ranks with six OpenMP threads each):

```
$ export OMP_NUM_THREADS=6
$ mpirun -np 8 amplxe-cl -collect hotspots --result-dir miniFE-8x6 -- \
    miniFE.x -nx=500
```

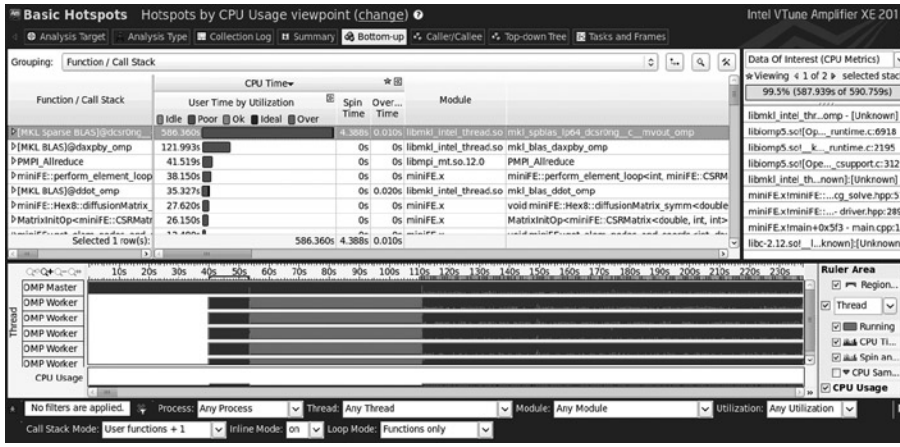


Figure 6-5. *Hotspot profile of the miniFE application to determine potential load imbalances*

Using this command line to collect performance data, VTune Amplifier XE produced eight different results databases (`miniFE-8x6.0` to `miniFE-8x6.7`), each of which contains the performance data for one of the eight MPI ranks. Figure 6-5 only shows the performance data for the first MPI rank. The other seven MPI ranks expose the same performance characteristics, and thus we can restrict ourselves to the one MPI rank in this case. For other applications, it will be required to check all MPI ranks and their performance data individually to make sure there are no outliers in the runtime profile.

Let us have a look at the timeline view at the bottom of Figure 6-5. The timeline shows several threads active over time. There are some particular areas of interest. First, we can observe that only one thread is executing for about 40 seconds before multithreading kicks in. We can also spot a second sequential part ranging for about 56 seconds in total, from 54 seconds to 110 seconds in the timeline. Zooming in and filtering the timeline, we can find out that the code is doing a matrix initialization in the first 40 seconds of its execution. About one-third of the compute time in this part is also attributed to an `MPI_Allreduce` operation. A similar issue leads to the sequential part that begins at 54 seconds of the execution. While this is not a true load imbalance in the code, because OpenMP is not active in these parts of the application, its exposure is similar to a load imbalance. From a timeline perspective, a load imbalance will look similar to what we see in Figure 6-5. In our example, finding a parallelization scheme to also parallelize the sequential fractions may boost application performance, owing to the amount of time spend in these parts of the application.

The general approach to solving a load imbalance is to first try to modify the loop scheduling of the code in question. Typically, OpenMP implementations prefer static scheduling that assigns equally large numbers of loop iterations to individual worker threads. While it is a good solution for loops with equal compute time per iteration, any unbalanced loop will cause problems. OpenMP defines several loop scheduling types that you can use to resolve the load imbalance. Although switching to fully dynamic schedules such as `dynamic` or `guided` appears to be a good idea, these scheduling

schemes tend to increase contention between many OpenMP threads, because a shared variable that maintains the work distribution. Static scheduling can still be used despite the load imbalance it introduces if the chunk size is adjusted down so that round-robin scheduling kicks in. Because each of the threads then receives a sequence of smaller blocks, there is a good chance that, on average, all the threads will receive compute-intensive and less compute-intensive loop chunks. At the same time, it ensures that each thread can compute all iterations it has to process, without synchronizing with the other threads through a shared variable.

Thread Synchronization and Locking

Thread synchronization is a double-edged sword. It keeps your data structures safe in that it allows you to control concurrent access and avoid race conditions on shared data; but if synchronization is introduced into the code, then parallelism may naturally suffer because synchronization constructs are meant to avoid concurrent execution of code regions. As a matter of fact, there will always be a tradeoff between limiting the degree of parallelism by introducing synchronization and choosing data structures and algorithms that need less synchronization for better parallelization.

In Table 5-8, you saw the performance of the MiniMD application on a workstation equipped with two Xeon processors. The data was for an execution that used Intel MPI on a single system to create several processes for execution. The version of MiniMD that we used to produce Table 5-8 also supported OpenMP-parallel execution instead of only MPI. So, a valid question is: Why did we use a message-passing library if there is shared memory available and if we could use multithreading instead? Let's hold that thought for a minute and just repeat the same benchmark, but now with OpenMP multithreading.

Figure 6-6 shows a speedup chart that compares the multiprocess MPI run with the multithread execution on the same machine. While the single-process and single-thread configuration exhibits the same performance behavior, there is a large gap between the MPI and the OpenMP versions. The OpenMP code is almost two times slower in all cases in comparison with the MPI version. In principle, an n-body algorithm should nicely scale with the number of cores, as shown by the MPI version. There is undoubtedly something going on in the OpenMP version of the code. Let's use VTune Amplifier XE to find out.

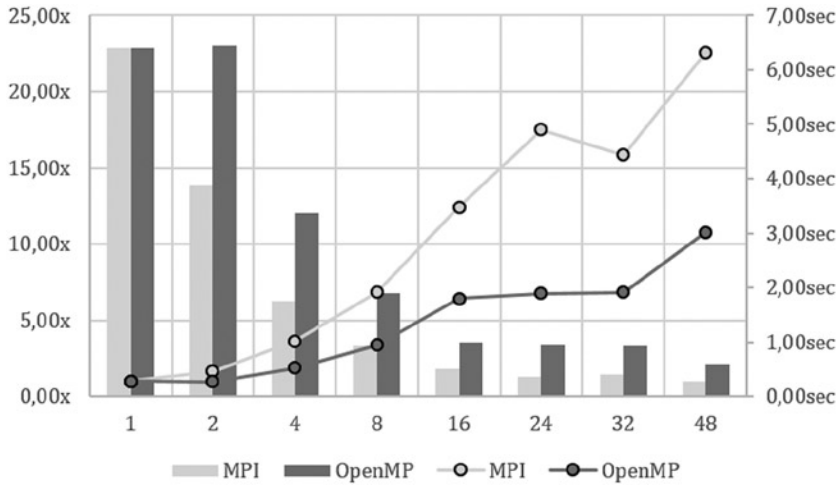


Figure 6-6. *Speedup graph (lines) and absolute runtime (bars) for the MiniMD benchmark*

We have executed the application with the following sequence of commands:

```
$ source /opt/intel/vtune_amplifier_xe/amplxe-vars.sh
$ amplxe-cl -collect advanced-hotspots -r omp -- \
    ./miniMD_intel --num_threads 24
$ amplxe-cl -collect advanced-hotspots -r mpi -- \
    mpirun -np 24 ./miniMD_intel
```

These commands instruct VTune Amplifier XE to collect two profiles:

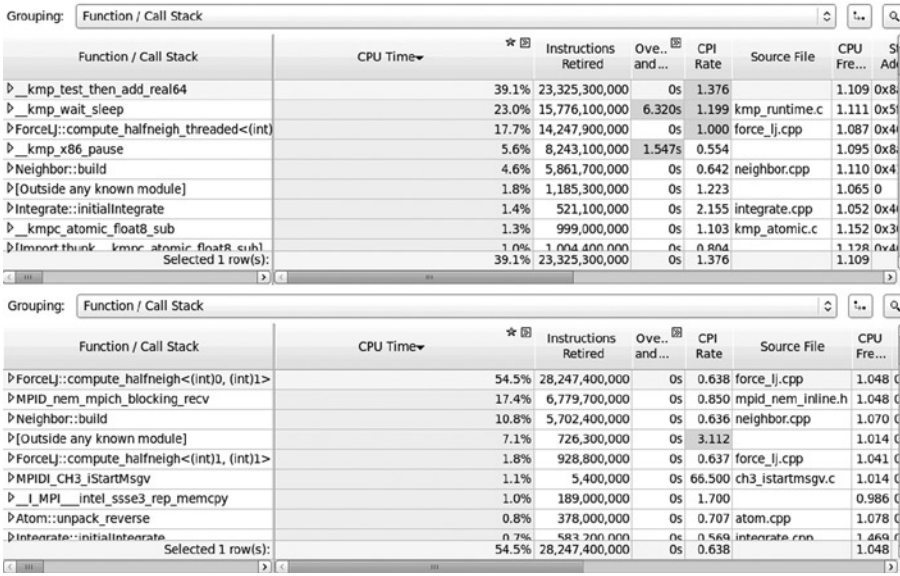
1. One process with 24 OpenMP threads
2. Twenty-four MPI ranks with one thread each

The collected profiles are named `omp` and `mpi`, respectively, through the `--result-dir` command line option of the collector.

The profiles are fundamentally different in what they represent from a data collection perspective. For the `omp` profile, VTune Amplifier XE monitored the performance events while MiniMD executed and created a performance database for just a single process with 24 threads. In the case of the `mpi` profile, the collector recognized that multiple MPI processes were spawned by the `mpirun` command. The performance database thus contains performance data from all 24 MPI ranks in a single profile.

Figure 6-7 shows the hotspots profiles of both executions. MPI is shown at the top, OpenMP at the bottom. As you can see from the hotspots profile, for MPI the hotspot is the `ForceLJ::compute_halfneigh` function, whereas for OpenMP it is a function called `__kmp_test_then_add_real64`. Functions that have a prefix `__kmp` in their name are compiler-internal functions used to implement OpenMP in Intel Composer XE.

In typical applications these functions show up in the hotspots profile from time to time and sometimes, as in this case, they are the culprit. To find out, we need to take a closer look at what the `__kmp_test_then_add_real64` function does.



Grouping: Function / Call Stack

Function / Call Stack	CPU Time	Instructions Retired	Ove... and ...	CPI Rate	Source File	CPU Fre...	Sl Ad
__kmp_test_then_add_real64	39.1%	23,325,300,000	0s	1.376		1.109	0x8
__kmp_wait_sleep	23.0%	15,776,100,000	6.320s	1.199	kmp_runtime.c	1.111	0x5
ForceLJ::compute_halfneigh_threaded<(int)	17.7%	14,247,900,000	0s	1.000	force_lj.cpp	1.087	0x4
__kmp_x86_pause	5.6%	8,243,100,000	1.547s	0.554		1.095	0x8
Neighbor::build	4.6%	5,861,700,000	0s	0.642	neighbor.cpp	1.110	0x4
[Outside any known module]	1.8%	1,185,300,000	0s	1.223		1.065	0
Integrate::initialIntegrate	1.4%	521,100,000	0s	2.155	integrate.cpp	1.052	0x4
__kmpc_atomic_float8_sub	1.3%	999,000,000	0s	1.103	kmp_atomic.c	1.152	0x3
[Import thunk ... kmpc_atomic_float8_sub]	1.0%	1,004,400,000	0s	0.804		1.128	0x4
Selected 1 row(s):	39.1%	23,325,300,000	0s	1.376		1.109	

Grouping: Function / Call Stack

Function / Call Stack	CPU Time	Instructions Retired	Ove... and ...	CPI Rate	Source File	CPU Fre...
ForceLJ::compute_halfneigh<(int)0, (int)1>	54.5%	28,247,400,000	0s	0.638	force_lj.cpp	1.048
MPID_nem_mpich_blocking_recv	17.4%	6,779,700,000	0s	0.850	mpid_nem_inline.h	1.048
Neighbor::build	10.8%	5,702,400,000	0s	0.636	neighbor.cpp	1.070
[Outside any known module]	7.1%	726,300,000	0s	3.112		1.014
ForceLJ::compute_halfneigh<(int)1, (int)1>	1.8%	928,800,000	0s	0.637	force_lj.cpp	1.041
MPIDI_CH3_iStartMsgv	1.1%	5,400,000	0s	66.500	ch3_istartmsgv.c	1.014
__I_MPI___intel_ssse3_rep_memcpy	1.0%	189,000,000	0s	1.700		0.986
Atom::unpack_reverse	0.8%	378,000,000	0s	0.707	atom.cpp	1.078
Integrate::initialIntegrate	0.7%	583,200,000	0s	0.569	integrate.cpp	1.469
Selected 1 row(s):	54.5%	28,247,400,000	0s	0.638		1.048

Figure 6-7. *Hotspots profiles for the MPI version (top) and OpenMP version (bottom) of MiniMD*

Let's have a closer look at it by double-clicking its line in the tabular view. This takes us to the assembly code of the function, because runtime libraries shipped with Intel Composer XE usually do not ship with full debugging symbols and source code, for obvious reasons. If you inspect the machine code, you will find that its main operation consuming a lot of time is a machine instruction `lock cmpxchg`. This instruction is an atomic compare-and-exchange operation, which is frequently used to implement an atomic add operation.

Functions like `__kmp_test_then_add_real64` and similar ones that implement OpenMP locks are hints that the code issues too many fine-grained atomic instructions. In case of MiniMD, the culprit is an `atomic` directive that protects the force update and that causes slowdown compared to the MPI version. It is also responsible for the limited scalability of the OpenMP version because it quickly becomes a bottleneck for an increased number of threads.

EXERCISE 6-2

Browse through the MiniMD code and try to find the OpenMP `atomic` constructs that cause the overhead in the OpenMP version. Can you find similar synchronization constructs in your application?

How such a synchronization issue can be resolved depends on the type of application, its data structures, and the algorithms used. For the MiniMD application, the synchronization is required because the effect of force on one atom also has an effect on the source of the force. According to Newton's third law, this effect is exactly the reverse force: if atom A is affected by a positive amount, then atom B, the source of the force, will be affected by a negative amount. The MPI version exploits this physical law to compute the force on atom A and then simply updates atom B without recomputing the force from scratch. This roughly cuts the computation required by 50 percent. Because of this optimization, multithreading becomes a bit more complex. For OpenMP, the atoms are distributed across the OpenMP threads. But as the computation for one atom requires an update of the forces for a second atom, synchronization must be added to avoid a race between the threads owning the atom. MiniMD already offers such a mode that can be enabled by setting the command-line option `--half_neigh 0`. Figure 6-8 compares the two modes of MiniMD. As you can see, performance and scalability are greatly improved by avoiding the excess synchronization.

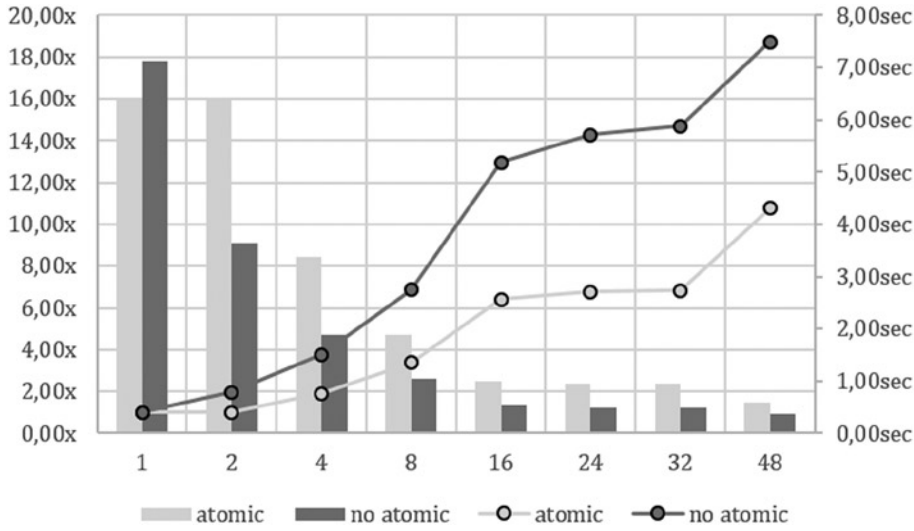


Figure 6-8. *Comparision of MiniMD with and without OpenMP* `atomic` *construct*

Another source of overhead are traditional locks, such as `omp_lock_t` or critical regions (`#pragma omp critical` in C/C++ or `!$omp critical` in Fortran). Whereas they share the property of mutual exclusion with their atomic instruction counterpart, they are typically more expensive and are widely used to protect code fragments and data structures that are more complex than simple updates of memory locations. VTune Amplifier XE offers a specialized analysis for problems that stem from these locks and helps to more easily pinpoint them in the code and their behavior at runtime. The analysis is called *Locks and Waits* and it specifically monitors the most commonly used

APIs to implement user-space locks. For each lock operation in the code, the analyzer helps browse through the participating threads (lock owner and waiting threads), the lock object involved, and the respective source code locations where the lock was acquired and released.

Dealing with Memory Locality and NUMA Effects

With the algorithmic improvements to obtain higher degrees of parallelism, we can now investigate how best to execute the parallel code on today's hardware. If you recall the features of the platform architecture that were described in Chapter 2, you will remember that a compute node typically contains several sockets with locally attached memory (NUMA), last level cache, and the compute cores with their private L1 and L2 caches. Because of the different bandwidth characteristics of local memory and remote memory, the placement of data and computation (i.e., threads and processes) becomes an important optimization target. It is key to keep data and computation on the same NUMA region to ensure lowest latency and highest memory bandwidth for the data accesses performed.

You may recall that each virtual page of the virtual memory associated with a process is backed up by a physical page that resides on one of the memory modules in one NUMA region. The Linux kernel uses a default strategy called *first touch* to allocate the physical pages. When an application allocates memory (for example, by calling `malloc` in C/C++), it receives a pointer to the allocated memory. However, the Linux kernel does not yet create any new physical pages unless the memory is accessed, or "touched." When a thread first touches the data by reading from it or writing into it, the physical page is allocated in the NUMA region that belongs to the core running that thread.

■ **Note** The `numactl` command introduced in Chapter 2 can also change the default allocation strategy of the Linux kernel. The argument `--localalloc` enables the standard Linux allocation strategy. With `--preferred` you can ask to place physical pages on a specific NUMA region, whereas `--membind` enforces placement on NUMA regions. Finally, the `--interleave` option interleaves the physical pages on several NUMA regions in a round-robin fashion. You can find additional details about this in the `man` page of the `numactl` command.

In a real application, this may severely penalize performance. If data is frequently accessed from a thread that runs on a different socket than the one that it ran on during allocation, the application will suffer from the lower bandwidth and higher latency of the remote data access. Figure 6-9 shows the achieved bandwidth of the STREAM Triad benchmark on our example machine. For the black line ("local memory") in the chart, we have executed the benchmark on socket 0 and used `numactl` to force memory allocation to the local memory:

```
$ (for i in `seq 1 12`; do OMP_NUM_THREADS=$i numactl --cpunodebind
   0 --membind 0 ./stream;
done) | grep "Triad:" | awk '{print $2}'
```

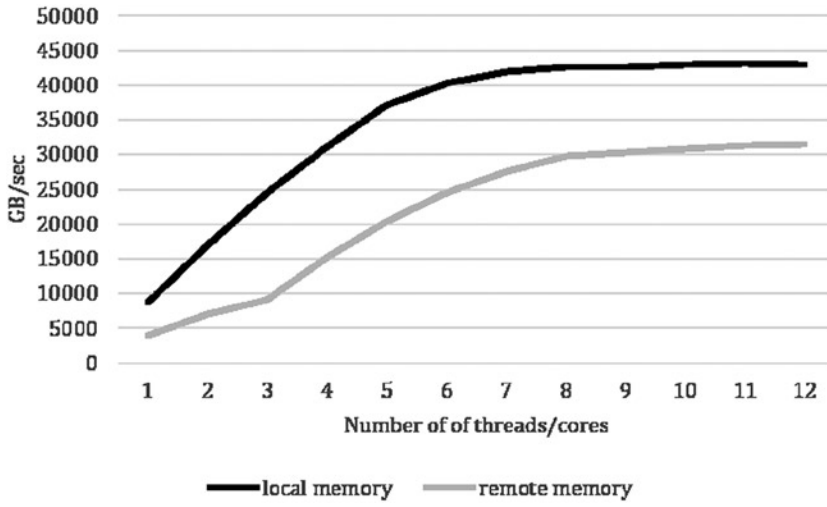


Figure 6-9. *Bandwidth as measured by the STREAM Triad benchmark*

The gray line shows the bandwidth we have obtained by forcing memory allocation to the second NUMA region, while keeping the threads on the first socket:

```
$ (for i in `seq 1 12`; do OMP_NUM_THREADS=$i numactl --cpunodebind
   0 --membind 1 ./stream;
done) | grep "Triad:" | awk '{print $2}'
```

It is easy to see how much available memory bandwidth we lost by choosing a wrong placement for data and computation. It is key to tie data and computation together on the same NUMA region whenever possible. This will greatly improve application performance. If the application is too complex to improve its NUMA awareness, you can still investigate if interleaved page allocation or switching off the NUMA mode in the BIOS improves overall performance. With these settings, the memory allocations are then distributed across the whole machine and thus all accesses are going equally to local and remote memory, on average.

If you wish to optimize the application and improve its NUMA awareness, then there are several ways to accomplish this mission. First, there are ways to bind threads and processes to individual NUMA regions so that they stay close to their data. We used the `numactl` command earlier to do this, but Linux offers several other APIs (for instance, `sched_setaffinity`) or tools (for example, `taskset`) to control process and threads in a machine-dependent manner. You may also recall the `I_MPI_PIN` environment variable and its friends (see Chapter 5) that enable a more convenient way of controlling process placement for MPI applications. Of course, typical OpenMP implementations also provide similar environment variables. (We will revisit this topic later in this chapter, when we look at hybrid MPI/OpenMP applications.)

Second, you can exploit the first-touch policy of the operating system in a threaded application. The key idea here is to use the same parallelization scheme to initialize data and to make sure that the same parallelization scheme is also used for computation. Listing 6-2 shows an example of a (very) naïve matrix-vector multiplication code that uses OpenMP for multithreading. Apart from the `compute` function, which computes the result of the matrix-matrix multiplication, the code contains two initialization functions (`init` and `init_numa_aware`). In the `init` function, the master thread allocates all data structures and then initializes the data sequentially. With the first-touch policy of the Linux kernel, all physical pages will therefore reside on the NUMA region that executed the master thread. The `init_numa_aware` function still uses the master thread to allocate the data through `malloc`. However, the code then runs the initialization in an OpenMP parallel `for` loop with the same loop schedule as the accesses in the `compute` function happen for the `A` and `c` arrays. Because each OpenMP thread now touches the same data for `A` and `c` it is supposed to work on, the physical pages are distributed across the NUMA regions of the machine and locality is improved.

Listing 6-2. Simplistic Matrix-vector Multiplication with NUMA-aware Memory Allocation

```
void init() {
    A = (double*) malloc(sizeof(*A) * n * n);
    b = (double*) malloc(sizeof(*b) * n);
    c = (double*) malloc(sizeof(*c) * n);
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            A[i*n+j] = ((double) rand())/((double) RAND_MAX);

    for (int i = 0; i < n; i++) {
        b[i] = ((double) rand())/((double) RAND_MAX);
        c[i] = 0.0;
    }
}

void init_numa_aware() {
    A = (double*) malloc(sizeof(*A) * n * n);
    b = (double*) malloc(sizeof(*b) * n);
    c = (double*) malloc(sizeof(*c) * n);
#pragma omp parallel
    {
#pragma omp for
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                A[i*n+j] = ((double) rand())/((double) RAND_MAX);
```

```
#pragma omp for
        for (int i = 0; i < n; i++) {
            b[i] = ((double) rand())/((double) RAND_MAX);
            c[i] = 0.0;
        }
    }
}

void compute() {
#pragma omp parallel for
    for(int i = 0; i < n; i++)
        for(int j = 0; j < n; j++)
            c[i] += A[i*n+j] * b[j];
}
```

The array `b` is a special case in this example. If you consider the `compute` function, you will see that `b` is read equally from all threads. So at first glance it does not seem to make a real difference if we used a NUMA-aware allocation or just allocate it in a single NUMA region. Unless the matrix size becomes unreasonably large, it will likely be that `b` will fit in the last-level cache of the individual sockets, so that no NUMA effects can be measured.

Of course, all this only happens if the working size of the application requires allocation of several physical pages so that they can be distributed across the different NUMA regions. The data also needs to be large enough so that the caches are not effective and that out-of-cache data accesses happen. For a perfectly cache-optimized code, the effect of this optimization may be low or even negligible. If threads frequently access a large, shared, but read-only data structure (like `b`) that does not fit the LLC of the sockets, then distributing it across several NUMA regions will still likely benefit performance. In this case, distributing the data helps avoid overloading a single NUMA region with memory accesses from other NUMA regions.

The effect of parallel data allocation in Listing 6-2 can be visualized nicely with the STREAM Triad benchmark. Figure 6-10 summarizes different thread placements and the effect of NUMA-aware allocation on memory bandwidth. The `compact` (gray solid and dashed line) in the chart indicates that the OpenMP runtime was instructed to first fill a socket with threads before placing threads on the second socket. “Scatter” (black solid and dashed line) distributes the threads in round-robin fashion. (We will have a closer look at these distribution schemes in the next section).

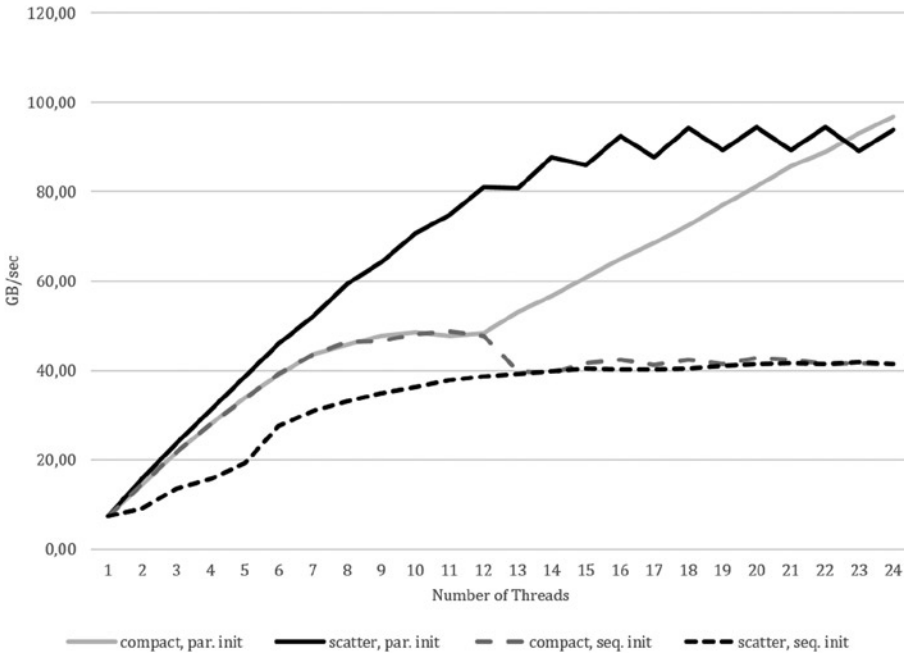


Figure 6-10. *STREAM Triad bandwidth with NUMA-aware allocation across multiple NUMA regions*

What you can observe from Figure 6-10 is that NUMA awareness always provides best results, as it fully exploits the capabilities of the memory subsystem. If threads are kept close to each other (`compact`), adding the second NUMA region contributes additional memory bandwidth, which is expected. For the `scatter` distribution, the memory bandwidth of the two NUMA regions of the system contributes to the aggregate memory bandwidth when at least two threads are executing. However, memory bandwidth will be up to a factor of two less if memory is allocated in only one NUMA region.

Unfortunately, NUMA-aware data allocation is not possible in all cases. One peculiar example is MPI applications that employ OpenMP threads. In many cases, these applications use the `MPI_THREAD_FUNNELED` or `MPI_THREAD_SERIALIZED` modes in which only one thread performs the MPI operations. If messages are received into a newly allocated buffer, then the first-touch policy automatically allocates the backing store of the buffer on a single NUMA region in which the communicating thread was executing. If you wish to run OpenMP threads across multiple NUMA regions and still maintain NUMA awareness, things tend to become complex and require a lot of thought and fine-tuning. Depending on how long the data will be live in the buffer and how many accesses the threads will make, it might be beneficial to either make a multithreaded copy of the buffer so that the accessing threads also perform the first touch, or use the Linux kernel's interface for page migration to move the physical pages into the right NUMA domain. However, these will be costly operations that need to be amortized by enough data accesses. Plus, implementing the migration strategies adds a lot of boilerplate code to the

application. The easiest way of solve this is to use one MPI rank per NUMA region and restrict OpenMP threading to that region only. In this case, there are no changes required to the application code, but you will need to properly bind threads and processes to the NUMA regions and their corresponding cores.

Thread and Process Pinning

Besides the aforementioned need to properly place processes and threads to get a better data locality in NUMA systems, thread and process pinning also offer other benefits that may lead to performance improvements.

As shown in Figure 6-10, putting threads or processes far apart in the system (`scatter`)—that is, on different sockets of the machine—can improve the aggregated memory bandwidth. As each socket has its own memory subsystem, the threads on different sockets do not compete for the same memory channels and thus receive more memory bandwidth in total. The same applies to the total amount of last-level cache (LLC) available to the application.

On the other hand, scattered distribution has some disadvantages. If threads communicate a lot by reading and writing to variables and data structures shared between them, then communication across the QPI link can easily become a bottleneck. The same applies to synchronization constructs such as barriers, locks, and atomic operations. Synchronization constructs are much more efficient if the participating threads are on the same socket. This is because the memory operations involved in implementing the synchronization are much faster when running from the same shared (last-level) cache instead of involving communication over the QPI links of the system. While synchronization is a good reason to keep threads as close as possible, it conflicts with the above benefits of spreading the threads across the system. In general, one can only hope to find a good tradeoff between the conflicting benefits and to approximate the ideal placement configuration.

Controlling OpenMP Thread Placement

Intel Composer XE, and its implementation of OpenMP, offers two ways to control thread placement in an application:

1. `KMP_AFFINITY` environment variable
2. `OMP_PROC_BIND` interface of the OpenMP 4.0 API

For a long time, before the OpenMP API version 4.0 was released, `KMP_AFFINITY` was the standard way of controlling thread placement for the Intel implementation of OpenMP. Through this environment variable, you can control thread placement on several levels ranging from abstract placement policies to a fine-grained mapping of OpenMP threads to sockets, cores, and hyper-threads. The settings of `KMP_AFFINITY` are effective for the whole application process—that is, if the process spawns multiple parallel regions, the same settings pertain for all parallel regions. `KMP_AFFINITY` also supports only one level of parallelism, but no nested OpenMP parallel regions.

Listing 6-3 shows the effect of different values for the KMP_AFFINITY variable on the thread placement. It shows how 18 threads are mapped to the cores of our two-socket example machine. For the compact placement, all 18 threads will be assigned to the first socket. The scatter strategy assigns the threads to the sockets of the machine in a round-robin fashion; even thread IDs are assigned to the first socket, threads with odd ID execute on the second socket. We can check this allocation by adding the verbose modifier to the KMP_AFFINITY environment variable, which requests to print information about the machine structure and how the threads are assigned to the (logical) cores of the system (Listing 6-3). To make sense of the different IDs and the underlying machine structure, you may use the cpuinfo tool introduced in Chapter 5.

Listing 6-3. OpenMP Thread Pinning with Additional Information Printed for Each OpenMP Thread

```
$ OMP_NUM_THREADS=18 KMP_AFFINITY=granularity=thread,compact,verbose \
    ./my_app
OMP: Info #204: KMP_AFFINITY: decoding x2APIC ids.
OMP: Info #202: KMP_AFFINITY: Affinity capable, using global cpuid leaf 11
info
OMP: Info #154: KMP_AFFINITY: Initial OS proc set respected: {0,1,2,3,4,5,6,
7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,3
3,34,35,36,37,38,39,40,41,42,43,44,45,46,47}
OMP: Info #156: KMP_AFFINITY: 48 available OS procs
OMP: Info #157: KMP_AFFINITY: Uniform topology
OMP: Info #179: KMP_AFFINITY: 2 packages x 12 cores/pkg x 2 threads/core (24
total cores)
OMP: Info #206: KMP_AFFINITY: OS proc to physical thread map:
OMP: Info #171: KMP_AFFINITY: OS proc 0 maps to package 0 core 0 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 24 maps to package 0 core 0 thread 1
[...]
OMP: Info #144: KMP_AFFINITY: Threads may migrate across 1 innermost levels
of machine
OMP: Info #242: KMP_AFFINITY: pid 85939 thread 0 bound to OS proc set {0}
OMP: Info #242: KMP_AFFINITY: pid 85939 thread 1 bound to OS proc set {24}
OMP: Info #242: KMP_AFFINITY: pid 85939 thread 2 bound to OS proc set {1}
OMP: Info #242: KMP_AFFINITY: pid 85939 thread 3 bound to OS proc set {25}
OMP: Info #242: KMP_AFFINITY: pid 85939 thread 4 bound to OS proc set {2}
OMP: Info #242: KMP_AFFINITY: pid 85939 thread 5 bound to OS proc set {26}
OMP: Info #242: KMP_AFFINITY: pid 85939 thread 6 bound to OS proc set {3}
OMP: Info #242: KMP_AFFINITY: pid 85939 thread 7 bound to OS proc set {27}
OMP: Info #242: KMP_AFFINITY: pid 85939 thread 8 bound to OS proc set {4}
OMP: Info #242: KMP_AFFINITY: pid 85939 thread 9 bound to OS proc set {28}
OMP: Info #242: KMP_AFFINITY: pid 85939 thread 10 bound to OS proc set {5}
OMP: Info #242: KMP_AFFINITY: pid 85939 thread 11 bound to OS proc set {29}
OMP: Info #242: KMP_AFFINITY: pid 85939 thread 12 bound to OS proc set {6}
OMP: Info #242: KMP_AFFINITY: pid 85939 thread 13 bound to OS proc set {30}
OMP: Info #242: KMP_AFFINITY: pid 85939 thread 14 bound to OS proc set {7}
OMP: Info #242: KMP_AFFINITY: pid 85939 thread 16 bound to OS proc set {8}
```

```
OMP: Info #242: KMP_AFFINITY: pid 85939 thread 15 bound to OS proc set {31}
OMP: Info #242: KMP_AFFINITY: pid 85939 thread 17 bound to OS proc set {32}
[...]

$ OMP_NUM_THREADS=18 KMP_AFFINITY=granularity=thread,scatter,verbose \
    ./my_app
OMP: Info #204: KMP_AFFINITY: decoding x2APIC ids.
OMP: Info #202: KMP_AFFINITY: Affinity capable, using global cpuid leaf 11
info
OMP: Info #154: KMP_AFFINITY: Initial OS proc set respected: {0,1,2,3,4,5,6,
7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,3
3,34,35,36,37,38,39,40,41,42,43,44,45,46,47}
OMP: Info #156: KMP_AFFINITY: 48 available OS procs
OMP: Info #157: KMP_AFFINITY: Uniform topology
OMP: Info #179: KMP_AFFINITY: 2 packages x 12 cores/pkg x 2 threads/core (24
total cores)
OMP: Info #206: KMP_AFFINITY: OS proc to physical thread map:
OMP: Info #171: KMP_AFFINITY: OS proc 0 maps to package 0 core 0 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 24 maps to package 0 core 0 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 1 maps to package 0 core 1 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 25 maps to package 0 core 1 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 2 maps to package 0 core 2 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 26 maps to package 0 core 2 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 3 maps to package 0 core 3 thread 0
[...]
OMP: Info #242: KMP_AFFINITY: pid 85979 thread 0 bound to OS proc set {0}
OMP: Info #242: KMP_AFFINITY: pid 85979 thread 1 bound to OS proc set {12}
OMP: Info #242: KMP_AFFINITY: pid 85979 thread 2 bound to OS proc set {1}
OMP: Info #242: KMP_AFFINITY: pid 85979 thread 3 bound to OS proc set {13}
OMP: Info #242: KMP_AFFINITY: pid 85979 thread 4 bound to OS proc set {2}
OMP: Info #242: KMP_AFFINITY: pid 85979 thread 5 bound to OS proc set {14}
OMP: Info #242: KMP_AFFINITY: pid 85979 thread 6 bound to OS proc set {3}
OMP: Info #242: KMP_AFFINITY: pid 85979 thread 7 bound to OS proc set {15}
OMP: Info #242: KMP_AFFINITY: pid 85979 thread 8 bound to OS proc set {4}
OMP: Info #242: KMP_AFFINITY: pid 85979 thread 9 bound to OS proc set {16}
OMP: Info #242: KMP_AFFINITY: pid 85979 thread 10 bound to OS proc set {5}
OMP: Info #242: KMP_AFFINITY: pid 85979 thread 11 bound to OS proc set {17}
OMP: Info #242: KMP_AFFINITY: pid 85979 thread 12 bound to OS proc set {6}
OMP: Info #242: KMP_AFFINITY: pid 85979 thread 13 bound to OS proc set {18}
OMP: Info #242: KMP_AFFINITY: pid 85979 thread 14 bound to OS proc set {7}
OMP: Info #242: KMP_AFFINITY: pid 85979 thread 16 bound to OS proc set {8}
OMP: Info #242: KMP_AFFINITY: pid 85979 thread 15 bound to OS proc set {19}
OMP: Info #242: KMP_AFFINITY: pid 85979 thread 17 bound to OS proc set {20}
[...]
```

If you carefully inspect the printout of Listing 6-3, it appears that the OpenMP runtime system has assigned the threads in a way that we did not expect in the first place. The `compact` policy assigned multiple OpenMP threads to the same physical core

(e.g., thread 0 and 1 to cores 0 and 24, respectively), whereas for `scatter`, it assigned different physical cores. Due to SMT, each physical core appears as two logical cores that may execute threads. With `compact`, we have requested from the OpenMP runtime to fill one socket first, before utilizing the second socket. The most compact thread placement is to put thread 0 to logical core 0 and use logical core 24 for thread 1, and so on. Thinking of a compact placement, this might not be what we have intended to do; you might have expected something along the line of placing 12 threads on the first socket and deploy the remaining six threads on the other socket.

The syntax for `KMP_AFFINITY` provides modifiers to further control its behavior. We already silently used `granularity` in Listing 6-3. You can use it to tell the Intel OpenMP implementation whether an OpenMP thread is to be assigned to a single logical core (`granularity=thread`) or to the hardware threads of a physical core (`granularity=core`). Once you have played a bit with these two settings, you will see that neither will deploy the 18 threads of our example to two sockets. The solution is to use `compact,1` as the policy. The effect is shown in Listing 6-4, in which 12 threads have been deployed to the first socket, and the remaining six threads have been assigned to the second socket. The documentation of Intel Composer XE[6] can give you more information on what `compact,1` means and what other affinity settings you can use.

Listing 6-4. Compact KMP_AFFINITY Policy Across Two Sockets of the Example Machine

```
$ OMP_NUM_THREADS=18 KMP_AFFINITY=granularity=thread,compact,1,verbose ./
my_app
[...]
OMP: Info #242: KMP_AFFINITY: pid 86271 thread 0 bound to OS proc set {0}
OMP: Info #242: KMP_AFFINITY: pid 86271 thread 1 bound to OS proc set {1}
OMP: Info #242: KMP_AFFINITY: pid 86271 thread 3 bound to OS proc set {3}
OMP: Info #242: KMP_AFFINITY: pid 86271 thread 2 bound to OS proc set {2}
OMP: Info #242: KMP_AFFINITY: pid 86271 thread 4 bound to OS proc set {4}
OMP: Info #242: KMP_AFFINITY: pid 86271 thread 5 bound to OS proc set {5}
OMP: Info #242: KMP_AFFINITY: pid 86271 thread 6 bound to OS proc set {6}
OMP: Info #242: KMP_AFFINITY: pid 86271 thread 7 bound to OS proc set {7}
OMP: Info #242: KMP_AFFINITY: pid 86271 thread 8 bound to OS proc set {8}
OMP: Info #242: KMP_AFFINITY: pid 86271 thread 9 bound to OS proc set {9}
OMP: Info #242: KMP_AFFINITY: pid 86271 thread 10 bound to OS proc set {10}
OMP: Info #242: KMP_AFFINITY: pid 86271 thread 11 bound to OS proc set {11}
OMP: Info #242: KMP_AFFINITY: pid 86271 thread 12 bound to OS proc set {12}
OMP: Info #242: KMP_AFFINITY: pid 86271 thread 13 bound to OS proc set {13}
OMP: Info #242: KMP_AFFINITY: pid 86271 thread 14 bound to OS proc set {14}
OMP: Info #242: KMP_AFFINITY: pid 86271 thread 16 bound to OS proc set {16}
OMP: Info #242: KMP_AFFINITY: pid 86271 thread 15 bound to OS proc set {15}
OMP: Info #242: KMP_AFFINITY: pid 86271 thread 17 bound to OS proc set {17}
```

With version 4.0 of the OpenMP API specification, OpenMP now defines a common way to deal with thread placement in OpenMP applications. In OpenMP terms, a *place* denotes an entity that is capable of executing an OpenMP thread and is described as an unordered list of numerical IDs that match the processing elements of the underlying

hardware. For Intel processors, these IDs are the core IDs as they appear in the operating system (e.g., as reported in `/proc/cpuinfo` or by `KMP_AFFINITY=verbose`). A *place list* contains an ordered list of places and is defined through the `OMP_PLACES` environment variable. The place list can also contain abstract names for places, such as `threads` (logical cores), `cores` (physical cores), or `sockets` (the sockets in the machine).

OpenMP also defines three placement policies with respect to an existing place list:

- `master`: Assign all threads of a team to the same place as the master thread of the team.
- `close`: Assign OpenMP threads to places such that they are close to their parent thread.
- `spread`: Sparsely distribute the OpenMP threads in the place list, dividing the place list into sublists.

In contrast to `KMP_AFFINITY`, the OpenMP placement policies can be used on a per-region basis by using the `proc_bind` clause at a `parallel` construct in the OpenMP code. It also supports nested parallelism through a list of policies separated by commas for the `OMP_PROC_BIND` variable. For each nesting level, one can specify a particular policy that becomes active, once a parallel region on that level starts executing. This is especially useful for applications that either use nested parallelism or that need to modify the thread placement on a per-region basis.

Listing 6-5 contains a few examples of different thread placements using `OMP_PLACES` and `OMP_PROC_BIND`. The first example has the same effect as the compact placement in Listing 6-4, whereas the second example assigns the threads in a similar fashion as the `scatter` policy of `KMP_AFFINITY`.

Listing 6-5. Examples for Using `OMP_PLACES` and `OMP_PROC_BIND`

```
$ OMP_NUM_THREADS=18 OMP_PROC_BIND=close OMP_PLACES=threads \
    KMP_AFFINITY=verbose ./my_app
[...]
OMP: Info #242: KMP_AFFINITY: pid 86565 thread 0 bound to OS proc set {0}
OMP: Info #242: OMP_PROC_BIND: pid 86565 thread 3 bound to OS proc set {25}
OMP: Info #242: OMP_PROC_BIND: pid 86565 thread 14 bound to OS proc set {7}
OMP: Info #242: OMP_PROC_BIND: pid 86565 thread 16 bound to OS proc set {8}
OMP: Info #242: OMP_PROC_BIND: pid 86565 thread 8 bound to OS proc set {4}
OMP: Info #242: OMP_PROC_BIND: pid 86565 thread 12 bound to OS proc set {6}
OMP: Info #242: OMP_PROC_BIND: pid 86565 thread 11 bound to OS proc set {29}
OMP: Info #242: OMP_PROC_BIND: pid 86565 thread 7 bound to OS proc set {27}
OMP: Info #242: OMP_PROC_BIND: pid 86565 thread 4 bound to OS proc set {2}
OMP: Info #242: OMP_PROC_BIND: pid 86565 thread 10 bound to OS proc set {5}
OMP: Info #242: OMP_PROC_BIND: pid 86565 thread 5 bound to OS proc set {26}
OMP: Info #242: OMP_PROC_BIND: pid 86565 thread 6 bound to OS proc set {3}
OMP: Info #242: OMP_PROC_BIND: pid 86565 thread 13 bound to OS proc set {30}
OMP: Info #242: OMP_PROC_BIND: pid 86565 thread 9 bound to OS proc set {28}
OMP: Info #242: OMP_PROC_BIND: pid 86565 thread 1 bound to OS proc set {24}
OMP: Info #242: OMP_PROC_BIND: pid 86565 thread 17 bound to OS proc set {32}
```

```
OMP: Info #242: OMP_PROC_BIND: pid 86565 thread 15 bound to OS proc set {31}
OMP: Info #242: OMP_PROC_BIND: pid 86565 thread 2 bound to OS proc set {1}
[...]

$ OMP_NUM_THREADS=18 OMP_PROC_BIND=spread OMP_PLACES=cores \
    KMP_AFFINITY=verbose ./my_app
[...]
OMP: Info #242: KMP_AFFINITY: pid 86690 thread 0 bound to OS proc set {0,24}
OMP: Info #242: OMP_PROC_BIND: pid 86668 thread 1 bound to OS proc set {2,26}
OMP: Info #242: OMP_PROC_BIND: pid 86668 thread 2 bound to OS proc set {3,27}
OMP: Info #242: OMP_PROC_BIND: pid 86668 thread 3 bound to OS proc set {4,28}
OMP: Info #242: OMP_PROC_BIND: pid 86668 thread 4 bound to OS proc set {6,30}
OMP: Info #242: OMP_PROC_BIND: pid 86668 thread 5 bound to OS proc set {7,31}
OMP: Info #242: OMP_PROC_BIND: pid 86668 thread 6 bound to OS proc set {8,32}
OMP: Info #242: OMP_PROC_BIND: pid 86668 thread 7 bound to OS proc set {10,34}
OMP: Info #242: OMP_PROC_BIND: pid 86668 thread 8 bound to OS proc set {11,35}
OMP: Info #242: OMP_PROC_BIND: pid 86668 thread 9 bound to OS proc set {12,36}
OMP: Info #242: OMP_PROC_BIND: pid 86668 thread 10 bound to OS proc set {14,38}
OMP: Info #242: OMP_PROC_BIND: pid 86668 thread 11 bound to OS proc set {15,39}
OMP: Info #242: OMP_PROC_BIND: pid 86668 thread 12 bound to OS proc set {16,40}
OMP: Info #242: OMP_PROC_BIND: pid 86668 thread 13 bound to OS proc set {18,42}
OMP: Info #242: OMP_PROC_BIND: pid 86668 thread 14 bound to OS proc set {19,43}
OMP: Info #242: OMP_PROC_BIND: pid 86668 thread 15 bound to OS proc set {20,44}
OMP: Info #242: OMP_PROC_BIND: pid 86668 thread 16 bound to OS proc set {22,46}
OMP: Info #242: OMP_PROC_BIND: pid 86668 thread 17 bound to OS proc set {23,47}
[...]
```

For more information on how to use `KMP_AFFINITY` and the OpenMP interface for threaded applications, see the user's guide of Intel Composer XE. For more advanced usage scenarios, the documentation also contains useful information on how programmers can use special runtime functions that allow for specific control of all aspects of thread pinning.

EXERCISE 6-3

Use different settings for `KMP_AFFINITY` and `OMP_PROC_BIND`, and conduct performance runs with these settings. What are the best settings for your application?

Thread Placement in Hybrid Applications

Process and thread placement may also lead to performance improvements for MPI/OpenMP hybrid applications. Depending on how many MPI ranks you are running per node, you may need to consider thread placement and find the ideal placement, similarly to what we have discussed for purely threaded applications.

If you configure the application to run only a single MPI rank per node, so that the remaining cores of the node are used to execute OpenMP threads, you'll need to place the threads appropriately to avoid NUMA issues and to make sure that the operating system keeps the threads where their data has been allocated.

If the application runs with one or more MPI ranks per socket, thread placement will be less of an issue. If the MPI rank is bound to a certain socket (the default for Intel MPI), the threads of each MPI process are automatically confined to execute on the same set of cores (or socket) that are available for their parent process (see Listing 6-6). Since now the MPI ranks' threads cannot move away from their executing socket, the NUMA issue is automatically solved. Data allocation and computation will always be performed on the same NUMA region. Pinning threads to specific cores might still lead to improvements, since it effectively avoids cache invalidations of the L1 and L2 caches that may happen owing to the threads' wandering around on different cores of the same socket.

In Listing 6-6, we instruct both the Intel MPI Library and the Intel OpenMP runtime to print their respective process and thread placements for MiniMD on a single node with two MPI ranks. As you can see, the Intel MPI Library automatically deploys one MPI rank per socket and restricts execution of the OpenMP threads to the cores of each socket. We can use this as a starting point and apply what we saw earlier in this section. Adding the appropriate `KMP_AFFINITY` settings, we can now make sure that each OpenMP thread is pinned to the same core during execution (shown in Listing 6-7).

Listing 6-6. Default Process and Thread Placement for an MPI/OpenMP Hybrid Application

```
$ I_MPI_DEBUG=4 KMP_AFFINITY=verbose mpirun "-prepend-rank  -np 2 \
    ./miniMD_intel --num_threads 12
[0] [0] MPI startup(): Single-threaded optimized library
[0] [0] MPI startup(): shm data transfer mode
[1] [1] MPI startup(): shm data transfer mode
[0] [0] MPI startup(): Rank    Pid      Node name  Pin cpu
[0] [0] MPI startup(): 0       87096    book       {0,1,2,3,4,5,6,7,8,9,
10,11,24,25,26,27,28,29,30,31,32,33,34,35}
[0] [0] MPI startup(): 1       87097    book       {12,13,14,15,16,17,18,19,
20,21,22,23,36,37,38,39,40,41,42,43,44,45,46,47}
[...]
[0] OMP: Info #242: KMP_AFFINITY: pid 87135 thread 0 bound to OS proc set
{0,1,2,3,4,5,6,7,8,9,10,11,24,25,26,27,28,29,30,31,32,33,34,35}
[0] OMP: Info #242: KMP_AFFINITY: pid 87135 thread 1 bound to OS proc set
{0,1,2,3,4,5,6,7,8,9,10,11,24,25,26,27,28,29,30,31,32,33,34,35}
[0] OMP: Info #242: KMP_AFFINITY: pid 87135 thread 3 bound to OS proc set
{0,1,2,3,4,5,6,7,8,9,10,11,24,25,26,27,28,29,30,31,32,33,34,35}
[0] OMP: Info #242: KMP_AFFINITY: pid 87135 thread 2 bound to OS proc set
{0,1,2,3,4,5,6,7,8,9,10,11,24,25,26,27,28,29,30,31,32,33,34,35}
[0] OMP: Info #242: KMP_AFFINITY: pid 87135 thread 4 bound to OS proc set
{0,1,2,3,4,5,6,7,8,9,10,11,24,25,26,27,28,29,30,31,32,33,34,35}
[0] OMP: Info #242: KMP_AFFINITY: pid 87135 thread 5 bound to OS proc set
{0,1,2,3,4,5,6,7,8,9,10,11,24,25,26,27,28,29,30,31,32,33,34,35}
```

```
[0] OMP: Info #242: KMP_AFFINITY: pid 87135 thread 6 bound to OS proc set
{0,1,2,3,4,5,6,7,8,9,10,11,24,25,26,27,28,29,30,31,32,33,34,35}
[0] OMP: Info #242: KMP_AFFINITY: pid 87135 thread 8 bound to OS proc set
{0,1,2,3,4,5,6,7,8,9,10,11,24,25,26,27,28,29,30,31,32,33,34,35}
[0] OMP: Info #242: KMP_AFFINITY: pid 87135 thread 7 bound to OS proc set
{0,1,2,3,4,5,6,7,8,9,10,11,24,25,26,27,28,29,30,31,32,33,34,35}
[0] OMP: Info #242: KMP_AFFINITY: pid 87135 thread 9 bound to OS proc set
{0,1,2,3,4,5,6,7,8,9,10,11,24,25,26,27,28,29,30,31,32,33,34,35}
[0] OMP: Info #242: KMP_AFFINITY: pid 87135 thread 10 bound to OS proc set
{0,1,2,3,4,5,6,7,8,9,10,11,24,25,26,27,28,29,30,31,32,33,34,35}
[0] OMP: Info #242: KMP_AFFINITY: pid 87135 thread 11 bound to OS proc set
{0,1,2,3,4,5,6,7,8,9,10,11,24,25,26,27,28,29,30,31,32,33,34,35}
[1] OMP: Info #242: KMP_AFFINITY: pid 87136 thread 0 bound to OS proc set
{12,13,14,15,16,17,18,19,20,21,22,23,36,37,38,39,40,41,42,43,44,45,46,47}
[1] OMP: Info #242: KMP_AFFINITY: pid 87136 thread 1 bound to OS proc set
{12,13,14,15,16,17,18,19,20,21,22,23,36,37,38,39,40,41,42,43,44,45,46,47}
[1] OMP: Info #242: KMP_AFFINITY: pid 87136 thread 2 bound to OS proc set
{12,13,14,15,16,17,18,19,20,21,22,23,36,37,38,39,40,41,42,43,44,45,46,47}
[1] OMP: Info #242: KMP_AFFINITY: pid 87136 thread 3 bound to OS proc set
{12,13,14,15,16,17,18,19,20,21,22,23,36,37,38,39,40,41,42,43,44,45,46,47}
[1] OMP: Info #242: KMP_AFFINITY: pid 87136 thread 4 bound to OS proc set
{12,13,14,15,16,17,18,19,20,21,22,23,36,37,38,39,40,41,42,43,44,45,46,47}
[1] OMP: Info #242: KMP_AFFINITY: pid 87136 thread 5 bound to OS proc set
{12,13,14,15,16,17,18,19,20,21,22,23,36,37,38,39,40,41,42,43,44,45,46,47}
[1] OMP: Info #242: KMP_AFFINITY: pid 87136 thread 6 bound to OS proc set
{12,13,14,15,16,17,18,19,20,21,22,23,36,37,38,39,40,41,42,43,44,45,46,47}
[1] OMP: Info #242: KMP_AFFINITY: pid 87136 thread 7 bound to OS proc set
{12,13,14,15,16,17,18,19,20,21,22,23,36,37,38,39,40,41,42,43,44,45,46,47}
[1] OMP: Info #242: KMP_AFFINITY: pid 87136 thread 8 bound to OS proc set
{12,13,14,15,16,17,18,19,20,21,22,23,36,37,38,39,40,41,42,43,44,45,46,47}
[1] OMP: Info #242: KMP_AFFINITY: pid 87136 thread 9 bound to OS proc set
{12,13,14,15,16,17,18,19,20,21,22,23,36,37,38,39,40,41,42,43,44,45,46,47}
[1] OMP: Info #242: KMP_AFFINITY: pid 87136 thread 10 bound to OS proc set
{12,13,14,15,16,17,18,19,20,21,22,23,36,37,38,39,40,41,42,43,44,45,46,47}
[1] OMP: Info #242: KMP_AFFINITY: pid 87136 thread 11 bound to OS proc set
{12,13,14,15,16,17,18,19,20,21,22,23,36,37,38,39,40,41,42,43,44,45,46,47}
[...]
```

Listing 6-7. Hybrid MPI/OpenMP Application with Thread-to-Core Pinning

```
$ I_MPI_DEBUG=4 KMP_AFFINITY=granularity=thread,compact,1,verbose \
    mpirun -prepend-rank -np 2
    ./miniMD_intel --num_threads 12
[0] [0] MPI startup(): Single-threaded optimized library
[0] [0] MPI startup(): shm data transfer mode
[1] [1] MPI startup(): shm data transfer mode
[0] [0] MPI startup(): Rank    Pid      Node name  Pin cpu
```

```
[0] [0] MPI startup(): 0       87377    book
{0,1,2,3,4,5,6,7,8,9,10,11,24,25,26,27,28,29,30,31,32,33,34,35}
[0] [0] MPI startup(): 1       87378    book
{12,13,14,15,16,17,18,19,20,21,22,23,36,37,38,39,40,41,42,43,44,45,46,47}
[...]
[0] OMP: Info #242: KMP_AFFINITY: pid 87377 thread 0 bound to OS proc set {0}
[0] OMP: Info #242: KMP_AFFINITY: pid 87377 thread 1 bound to OS proc set {1}
[0] OMP: Info #242: KMP_AFFINITY: pid 87377 thread 2 bound to OS proc set {2}
[0] OMP: Info #242: KMP_AFFINITY: pid 87377 thread 3 bound to OS proc set {3}
[0] OMP: Info #242: KMP_AFFINITY: pid 87377 thread 4 bound to OS proc set {4}
[0] OMP: Info #242: KMP_AFFINITY: pid 87377 thread 5 bound to OS proc set {5}
[0] OMP: Info #242: KMP_AFFINITY: pid 87377 thread 6 bound to OS proc set {6}
[0] OMP: Info #242: KMP_AFFINITY: pid 87377 thread 7 bound to OS proc set {7}
[0] OMP: Info #242: KMP_AFFINITY: pid 87377 thread 8 bound to OS proc set {8}
[0] OMP: Info #242: KMP_AFFINITY: pid 87377 thread 9 bound to OS proc set {9}
[0] OMP: Info #242: KMP_AFFINITY: pid 87377 thread 10 bound to OS proc set {10}
[0] OMP: Info #242: KMP_AFFINITY: pid 87377 thread 11 bound to OS proc set {11}
[1] OMP: Info #242: KMP_AFFINITY: pid 87378 thread 0 bound to OS proc set {12}
[1] OMP: Info #242: KMP_AFFINITY: pid 87378 thread 1 bound to OS proc set {13}
[1] OMP: Info #242: KMP_AFFINITY: pid 87378 thread 2 bound to OS proc set {14}
[1] OMP: Info #242: KMP_AFFINITY: pid 87378 thread 3 bound to OS proc set {15}
[1] OMP: Info #242: KMP_AFFINITY: pid 87378 thread 4 bound to OS proc set {16}
[1] OMP: Info #242: KMP_AFFINITY: pid 87378 thread 5 bound to OS proc set {17}
[1] OMP: Info #242: KMP_AFFINITY: pid 87378 thread 6 bound to OS proc set {18}
[1] OMP: Info #242: KMP_AFFINITY: pid 87378 thread 7 bound to OS proc set {19}
[1] OMP: Info #242: KMP_AFFINITY: pid 87378 thread 8 bound to OS proc set {20}
[1] OMP: Info #242: KMP_AFFINITY: pid 87378 thread 9 bound to OS proc set {21}
[1] OMP: Info #242: KMP_AFFINITY: pid 87378 thread 10 bound to OS proc set {22}
[1] OMP: Info #242: KMP_AFFINITY: pid 87378 thread 11 bound to OS proc set {23}
[...]
```

Summary

This chapter was all about optimizations on the threading level of the application to achieve better performance on a single node.

If your application is using only MPI to exchange messages on the process level and you are thinking about multithreading, this chapter showed how you can create a hotspot and loop profile to get a better understanding of the application behavior. This is your foundation for making informed decisions about where to apply OpenMP (or other threading models) to your code to move it to a hybrid MPI/OpenMP solution.

The hotspot profile is the tool for getting to know about optimization and parallelization candidates. The hotspots are always the optimization candidates that you will investigate closely and in depth so that you can find bottlenecks in these parts of your code. We have presented some of the most common application bottlenecks, such as sequential and load imbalanced parts of code, excessive thread synchronization, and issues introduced by the NUMA.

References

1. "Perf: Linux profiling with performance counters," https://perf.wiki.kernel.org/index.php/Main_Page.
2. J. Dongarra and M. A. Heroux, *Toward a New Metric for Ranking High Performance Computing Systems* (Albuquerque, NM: Sandia National Laboratories, 2013).
3. *Intel VTune Amplifier XE User's Guide* (Santa Clara, CA: Intel Corporation, 2014).
4. "Intel® Inspector XE 2015," https://software.intel.com/intel-inspector-xe.
5. Valgrind Developers, "Valgrind," http://valgrind.org/.
6. *User and Reference Guide for the Intel C++ Compiler 15.0* (Santa Clara, CA: Intel Corporation, 2014).