Full Throttle: OpenMP* 4.0

By Michael Klemm, Senior Application Engineer, Intel and Christian Terboven, Deputy Head of HPC Group, RWTH Aachen University

Introduction

“Multicore is here to stay.” This single sentence accurately describes the situation of application developers and the hardware evolution they are facing. Since the introduction of the first dual-core CPUs, the number of cores has kept increasing. The advent of the Intel® Xeon Phi™ coprocessor has pushed us into the world of manycore—where up to 61 cores with 4 threads each impose new requirements on the parallelism of applications to exploit the capabilities of the hardware.

It is not only the ever-increasing number of cores that requires more parallelism in an application. Over the past years, the width of SIMD (Single Instruction Multiple Data) registers has been growing. While the early single instruction multiple data (SIMD) instructions of Intel® MMX™ technology used 64-bit registers, our newest family member, Intel® Advanced Vector Instructions 512 (Intel® AVX-512), runs with 512-bit registers. That’s an awesome 16 floating-point numbers in single precision, or eight double-precision numbers that can be computed in one go. If your application does not exploit these SIMD capabilities, you can easily lose a factor of 16x or 8x compared to the peak performance of the CPU.
It is key for application developers to keep up with the evolution of the hardware in terms of both number of cores and SIMD capabilities. Today’s applications must exploit multiple levels of parallelism to make use of the compute power of today’s and tomorrow’s CPUs. Multithreading alone will not scale to the future. In addition, applications increasingly make use of plugins and libraries that are also written with parallelism in mind. Programmers now need to orchestrate the cooperation between their application code, third-party libraries, and hardware to achieve an efficient solution.

One possible solution to this are task-based programming models to describe how the application can be decomposed into concurrent tasks that can be independently executed. In contrast to traditional thread-based programming, benefits include a much more flexible mapping of application tasks to the execution units, and a much easier interaction between components. Each component can just create tasks, which then automatically intermix with all other tasks of the application. Programming models such as Intel® Cilk™ Plus, Intel® Threading Building Blocks (Intel® TBB), or C++11’s `async` feature are good examples.

**My Name is OpenMP**

Since its introduction in 1997, the OpenMP API emerged as a de-facto standard for shared-memory parallel programming. With a focus on technical and scientific computing, it supports C, C++, and Fortran. OpenMP compilers can be found for all mainstream platforms. OpenMP is “open,” since anyone can implement parts or the full specification without any licensing costs.

OpenMP consists of directives that describe how the code should be parallelized by the OpenMP compiler. It also defines an API and environment variables to control the runtime behavior of the code. The directives are implemented through pragmas in C/C++ or special comments in Fortran. Because of this, OpenMP code can usually also be compiled into a sequential version by ignoring the pragmas or comments. OpenMP makes it possible to incrementally add parallelism to existing code as well as to focus on the compute-intensive parts of the code.

The fundamentals of OpenMP are parallel regions, in which the “master thread” is joined by so-called “worker threads.” Outside of the parallel regions, the code runs sequentially in the master thread. Worksharing constructs provide a means to distribute work across the team of threads (Figure 1). In this example, the `for` loop is cut into the equal-size chunks which are assigned to the threads of the team. If this static distribution does not fit,

```cpp
1 const int N = 1000000;
2 double A[N], B[N], C[N];
3 // initialize A, B, and C
4 // ...
5
6 #pragma omp parallel for
7 for (int i = 0; i < N; i++) {
8     A[i] = B[i] + C[i];
9 }
```

Simple example of a parallel region with worksharing of a `for` loop
because it creates a load imbalance, the schedule clause can change the default distribution scheme. For instance, \texttt{schedule(dynamic, c)} creates chunks of size \(c\), and idle threads grab the next available chunk.

OpenMP also provides mechanisms to describe the visibility of data to the threads ("scoping"). It is important to tell the OpenMP compiler what data must remain in the shared memory domain and what data needs to be private to the individual threads. OpenMP defines clauses that can be added to the directives to control the scope of variables. The \texttt{shared} clause keeps a variable in the shared space, while the \texttt{private} clause creates a thread-private copy of a variable. Shared scope is the default for variables that are declared outside of a parallel region. In Figure 1, this applies to the variables \(A, B, C\), and the constant \(N\). Private variables can store different values for different threads, as needed by the loop counter \(i\), in the example. Private copies are by default created without initializing; \texttt{firstprivate} can be used to assign the value of the variable outside of the parallel region.

The latest version 4.0 of the OpenMP API specification not only includes minor bug fixes and improvements to existing features, it now supports a good share of features introduced with Fortran 2003. OpenMP affinity defines a common way to express thread affinity to execution units of the hardware. Version 4.0 also comes with major feature enhancements, some of which will be discussed in more detail here. Task groups improve tasking by providing a better way to express synchronization of a set of tasks and to handle cancellation, which allows to stop parallel execution. SIMD pragmas extend the thread-parallel execution to data-parallel SIMD machine instructions, while user-defined reductions let programmers specify arbitrary reduction operations. Possibly the biggest addition to OpenMP is support for offloading computation to coprocessor devices.

**Talk the Talk, Task the Task**

The growing number of cores (and threads) make it harder to fully utilize the cores with traditional worksharing constructs for parallel loops. Irregular algorithms, such as recursions and traversals of graphs, require a completely different approach to parallelism. Task-based models blend well with the requirements of these algorithms, since tasks can be created in a much more flexible way.

An OpenMP task may be treated as a small package that consists of a piece of code to be executed and all the data needed for execution. An OpenMP task is created through the \texttt{#pragma omp task} directive to mark a piece of code and data for concurrent execution. The OpenMP runtime system takes care of mapping the created tasks to the threads of a parallel region. It may defer the task for later execution by adding it to a task queue or it may execute the task immediately.

**Figure 2** shows a task-parallel version of a very simple, brute-force Sudoku* solver. The idea of the algorithm is:

1. Find an empty field without a number
2. Insert a number
3. Check the Sudoku board
4. If the solution is invalid, try the next possible number
5. If the solution is valid, go to the next field and start over
void main() {
    // setup data structures
    // ...

#pragma omp parallel    // start parallel region
{
    // limit to one thread
#pragma omp single
{
    // group all tasks
#pragma omp taskgroup
{
    solve_parallel(0, 0, sudoku);
    }
}
} // end omp parallel

void solve_parallel(int x, int y, CSudokuBoard* sudoku, CSudokuBoard* & solution) {
    if (x == sudoku->getFieldSize()) {   // end of Sudoku line
        y++;   x = 0;
        if(y == sudoku->getFieldSize())   // end of Sudoku field
            return true;
    }

    if (sudoku->get(y, x) > 0) {   // field already set
        return solve_parallel(x+1, y, sudoku); // tackle next field
    }

    for (int i = 1; i <= sudoku->getFieldSize(); i++) {  // try all possible numbers
        if (!sudoku->check(x, y, i)) {   // if number fits, set it
            CSudokuBoard* new_sudoku = new CSudokuBoard(*sudoku);  // create new solver task
            new_sudoku->set(y, x, i);
            if (solve_parallel(x+1, y, new_sudoku)) {   // tackle next field
                if (!solution) {  // if new solution, save it
                    solution = new_sudoku;
                }
            }
            delete new_sudoku;  // clean up
        }
    }

    if (sudoku->get(y, x) > 0) {   // field already set
        return solve_parallel(x+1, y, sudoku); // tackle next field
    }

    for (int i = 1; i <= sudoku->getFieldSize(); i++) {   // try all possible numbers
        if (!sudoku->check(x, y, i)) {  // if number fits, set it
            CSudokuBoard* new_sudoku = new CSudokuBoard(*sudoku);  // create new solver task
            new_sudoku->set(y, x, i);
            if (solve_parallel(x+1, y, new_sudoku)) {   // tackle next field
                if (!solution) {  // if new solution, save it
                    solution = new_sudoku;
                }
            }
            delete new_sudoku;  // clean up
        }
    }

    // request cancellation
    #pragma omp cancel taskgroup
}

    // await completion of child tasks
#pragma omp taskwait
    // no solution found, reset field
}

Sudoku* solver example with OpenMP* tasks and cancellation
To execute tasks, a parallel region first creates the team of threads. In our example, only one thread needs to start with the Sudoku algorithm, as the algorithm starts task creation when fired up. The algorithm creates tasks in step four above. Trying to solve the Sudoku board for different configurations of a number of a given field can be parallelized: each task can try a different number and check the board for a valid solution.

The example also shows that combining C++ classes and OpenMP is easy. The variable `sudoku` is a pointer to instances of the `CSudokuBoard` class and the `firstprivate` indicates that each task received a private copy of that pointer. The tasks can then create a copy of the instance by invoking its copy constructor.

All threads of a team participate in executing deferred tasks from the task queue. If threads run into a barrier or some other synchronization construct, they can check for available tasks and execute them. It is explicitly allowed by OpenMP that undersupplied threads steal tasks from overloaded threads.

OpenMP offers several synchronization constructs to synchronize the execution of tasks. Barriers guarantee that all tasks created before reaching a barrier have been executed when the barrier is left by the team. The `taskwait` construct waits for the completion of all child tasks created by a parent task. Finally, the `taskgroup` construct logically groups all tasks created within the construct and establishes an implicit `taskwait` for all tasks of the group at the end of the construct.

**Stopping Midstream**

Sometimes you may want to abort a parallel computation because of an unforeseen situation (e.g., a failure or error) that prevents execution from continuing cleanly. Another reason might be that the result has been computed and it does not make much sense to continue execution. Examples are numerical algorithms or search algorithms. With OpenMP 3.1, programmers could not easily implement such algorithms with OpenMP, since it did not support stopping a parallel execution once it had been started. For some OpenMP constructs, there have been workarounds. For instance, stopping a `for` worksharing construct involved an `if` statement that turned the `for` loop into an empty loop that ran to its natural end without doing any more work. Despite being a dirty hack, the loop continued to run, which costs precious energy and consumes time.

OpenMP 4.0 solves this problem by introducing directives to cleanly abort parallel execution. The key directive is the `cancel` directive to request termination of the current OpenMP region. The requesting thread immediately stops execution and notifies the remaining threads about the request for termination. If they receive a notification, the threads check at so-called “cancellation points” and stop execution.

The `cancel` directive supports termination of parallel regions, worksharing constructs, and task groups. Cancellation points are automatically inserted at barriers and the `cancel` directive. Programmers can add additional cancellation points to the code through the `cancellation point` directive. When cancellation occurs, the OpenMP runtime does not release any acquired resources such as allocated memory, locks, or open files. It is the programmer’s responsibility to clean up before cancellation is requested, or before a thread hits a cancellation point that triggers cancellation.
The Sudoku example in Figure 2 uses cancellation to stop searching for a solution of the Sudoku board. If one of the solver tasks found a new solution, it first checks whether some other task has already found another solution. If it is the first solution found, the task saves the solution and requests cancellation. We use a critical region to avoid a potential race condition on the solution and to avoid two tasks asking for cancellation at the same time.

In the example, we rely on the behavior of task cancellation. All tasks that have started execution may run to completion and are not aborted unless they contain a cancellation point. All other tasks sent to the waiting queue are discarded and considered completed. This explains why each task must always check for a solution already found. We cannot know if a task might start execution ahead of time. Hence, we need a safety net to avoid storing a duplicate solution if a task slips into execution shortly after one task has found solution and requested cancellation.

SIMD Me

OpenMP 3.0 and earlier versions focused solely on multithreading and left other topics, such as data-parallel SIMD instructions, to other paradigms. If a programmer wanted to exploit the SIMD features of modern processors, she was left hoping for the compiler’s auto-vectorizer to be smart enough to insert appropriate SIMD instructions. Otherwise, she had to use vendor-specific extensions, which are not easily portable and were found to be problematic in combination with OpenMP’s parallelization directives.

OpenMP 4.0 aims to improve this situation. It defines new constructs, allowing for the portable description of SIMD expressions and their combination with parallelization directives. The main building block for this is the simd construct to vectorize loops (Figure 3). It advises the compiler to introduce appropriate SIMD instructions.

```plaintext
#pragma omp declare simd aligned(a,b) notinbranch
float min(float a, float b) {
  return a < b ? a : b;
}

#pragma omp declare simd aligned(x) uniform(y) notinbranch
float distance(float x, float y) {
  return (x - y) * (x - y);
}

void distance_update(float *a, float *b, float *y, int vlen) {
  float *ptr = b;
  #pragma omp parallel for simd safelen(16) linear(ptr:1) aligned(a,b,y)
  for (int i=0; i<vlen; i++) {
    y[i] = min(sqrt(distance(a[i], 1.0)), ptr);
    ptr += 1;
  }
}
```

Vectorization of functions and loop with OpenMP* SIMD directives
into the serial code. The `for simd` and `parallel for simd` constructs combine this aspect with the well-known loop-level thread parallelization, allowing for an efficient combination of both paradigms simultaneously.

The syntax of the new construct closely matches the existing OpenMP worksharing constructs and supports a large set of their established clauses (such as `private`, `reduction`, `collapse`, etc.), albeit with slightly revised semantics. Additionally, there are some new clauses that aid the compiler in creating efficient SIMD code. The `safelen` clause defines the maximum possible vector length for a loop, for example, in the presence of dependencies between loop iterations with a specific stride. Similarly important is `linear`, expressing a linear dependence of a variable to the loop counter. The `aligned` clause specifies data alignment to help the compiler choose optimal load and store instructions.

A challenge of vectorizing codes is when the loop body contains function calls, as shown in Figure 3. If there is no SIMD version of the functions `min` and `distance`, then the compiler cannot generate SIMD instructions. For many routines of the standard library, modern compilers already offer vectorized versions (e.g., `sqrt`, `sin`, `cos`). For all other functions, the programmer has to help again. In Figure 3, the `declare simd` construct instructs the compiler to generate an additional vector version by promoting scalar arguments to vectors. If necessary, the `uniform` clause avoids SIMD promotion for certain arguments. The `notinbranch` clause assures that the function will never be called from within a conditional branch (e.g., in the body of an `if` statement), hence allowing further compiler optimizations. The `inbranch` clause asserts the opposite.

Reducers Everywhere

Reductions are involved whenever a team of threads cooperatively work on a problem and have to produce a single, global result. Each thread receives a private copy of a variable to collect their local intermediate results. Shortly before parallel execution ends, the OpenMP runtime system collects all the intermediate results from all threads and reduces them into the global result. Before OpenMP 4.0, only predefined reductions operations, such as addition or minimum/maximum on primitive types of the base language (e.g., `int` or `float`), have been supported. Programmers previously had to write their own reduction code if derived data types or more complex operations had to be used. This resulted in more or less complex code patterns that had to be maintained by programmers.

OpenMP 4.0 includes support for user-defined reductions. Programmers can now define arbitrary reduction operations on arbitrarily complex data types. Figure 4 uses the new feature to implement a parallel algorithm to compute the bounding box of a cloud of 2D points. The bounding box is the smallest rectangle that contains all points. It is computed in 2D by determining the left-most and right-most, as well as lowest and highest, point of the cloud. These locations give you the lower left and upper right corner of the rectangle.
For Figure 4, we rely on a simple class `Point2D` that stores the x and y coordinates of a 2D point, and a class `Rectangle` that stores a rectangle consisting of two 2D points. The `declare reduction` directives in the example declare new user-defined reduction operations. Separated by colons, the directive introduces names for the reduction operation (`minp` and `maxp`) and defines the data types for which the reduction operation is effective (`Point2D`). The last part contains a C/C++ expression or Fortran statement that describes how to combine two local results into a new intermediate result. This expression or statement is applied repeatedly until all intermediate results have been combined into the global result. The `initializer` clause specifies how the thread-private copy of the reduction variable is initialized.

The `for` loop iterates all elements of a `std::vector` that stores all the points of the cloud. For each point, it computes the running minimum and maximum of x and y coordinates to find the corners of the minimal bounding box. To parallelize the code, we introduce a `parallel for` construct to distribute the loop across a team of threads. Thus, each thread computes only a local bounding box that contains only the points assigned to the thread. The missing piece is how to combine all the local bounding boxes into the global one. The `reduction` clause at the `parallel for` construct performs this operation by applying the new reduction operations `minp` and `maxp`. The `omp_in` and `omp_out` variables in the combiner expression refer to the left and right operand when applying the reduction expression. In the example, these operands are passed to the minimum/maximum operation to enlarge the bounding box.
High Speed

Support to offload computation to attached devices, such as the Intel Xeon Phi coprocessor, is probably the most groundbreaking feature of OpenMP 4.0. OpenMP 4.0 introduces a new device model that extends the traditional threading model for shared memory to support offload computations. Having an OpenMP specification for offloading provides an industry-wide solution superior in every way to the OpenACC* it supersedes. Unlike OpenACC, OpenMP allows for the full use of a wide variety of devices instead of restricting what can be offloaded. Unlike OpenACC, there is no restriction on the number of devices supported, though all devices used need to be of the same architecture. OpenMP defines new constructs and directives to facilitate transfer of control between the host and the devices, as well as to issue data transfers.

The `target` construct (see Figure 5) transfers control from the host thread to the coprocessor device, and creates a device data environment to contain all the data needed to execute code on the target device. The `map` clause at the target construct controls the allocation of data in the data environment, as well as the direction of data transfers. Figure 6 shows the different transfer types supported by `map`. In the example, we use `map(to:x[:N])` to transfer the array `x` from the host to the target device. The syntax `x[:N]` is shorthand for `x[0:N]` and describes `N` array elements starting from index 0, that is, all array elements in `x`.

```c
1 int n = 10240; float a = 2.0f; float b = 3.0f;
2 float *x = (float*) malloc(n * sizeof(float));   // init x
3 float *y = (float*) malloc(n * sizeof(float));   // init y
4
5 #pragma omp target data map(to:x[:N])
6 {
7   int num_blcks = 61;
8   int num_thrds = 4;
9   #pragma omp target map(tofrom:y[:N])
10  #pragma omp teams num_teams(num_blcks) num_threads(num_thrds)
11  #pragma omp distribute
12    for (int b = 0; b < n; b += num_blcks) {
13      #pragma omp parallel for
14        for (int i = b; i < b + num_blcks; ++i){
15          y[i] = a*x[i] + y[i];
16        }
17    }
18
19    // do something with y
20    // ...
21
22 #pragma omp target map(tofrom:y[:N])
23 #pragma omp teams distribute parallel for \
24   num_teams(num_blcks) num_threads(num_thrds)
25   for (int i = 0; i < n; ++i){
26     y[i] = b*x[i] + y[i];
27   }
28
29 free(x); free(y);
```

Offloading computation from the host to a coprocessor device
Since coprocessors are using the PCI Express* bus to talk to the host system, avoiding unnecessary data transfers is very important. The target data construct establishes a device data environment, but does not transfer the control flow. With this construct, programmers can transfer data to the target device and keep it there, while the control flow is sent back and forth through several target constructs. If the target construct notices that data is already present on the device, it will then avoid the data transfer. The target update directive can be used to issue data exchanges, if data on either the host or the devices is outdated.

Let's dissect the example of Figure 5 and explain how the different constructs interact. The example implements the well-known SAXPY operation $y = a \times x + y$. Since the code executes the operation two times, the code creates a device data environment for safe transfers of $x$ across different invocations. The target construct in the example will notice that $x$ is already available on the coprocessor, and only issue a data transfer for $y$. The same is true for the second target construct in the example.

The constructs mentioned so far only transfer control or data between the host and the devices. They do not automatically parallelize the code that is transferred over. Programmers need to use standard OpenMP features to create parallelism on the target devices. In our example, a simple parallel for construct would be sufficient to create a team of threads to execute the SAXPY loop in parallel. However, this may be inefficient, since the overhead of team creation and synchronization for the full coprocessor with 244 threads will be considerable.

To make the solution more efficient, the parallelization exploits the hierarchical architecture of four hyper-threads per physical coprocessor core. OpenMP 4.0 offers new constructs to map these hierarchies to program code:
The team’s constructs creates a league of independent thread teams, whose master thread executes the code of the construct.

The distribute construct is a new worksharing construct to distribute a loop across the master threads of a league. The distribute construct does not have an implicit barrier at the end.

As usual, the existing parallel for construct distributes a loop within a thread team.

The SAXPY code creates one thread team for each of the 61 physical cores of the coprocessor. As a first level of parallelism, it then distributes the outer loop over \( b \) across the created thread teams. The second level of parallelism consists of a parallel for to create four threads for each hyper-thread of the physical cores. The second invocation of SAXPY shows the syntactic sugar of the combined teams distribute parallel for constructs to make the code shorter.

The transfer of control from the host to a device is a blocking operation. That is, the host thread waits until the control flow on the target has finished and all data has been transferred. If both host and device need to fulfill concurrent tasks, programmers may use existing OpenMP features. For instance, programmers can wrap a target or target update construct in an OpenMP task to execute these constructs in another OpenMP thread.

**Conclusion**

With version 4.0, OpenMP makes a quantum leap forward. The newly introduced features open up a new world of heterogeneous programming. OpenMP 4.0 provides new features that make it an interesting programming model with a wider reach than technical and scientific computing. Intel continues its commitment to OpenMP as a parallel programming model. Intel® Composer XE 2013 fully supports the previous OpenMP API specification 3.1 and future versions will supply a first-class, high-performance implementation of OpenMP 4.0. With SP1, Intel Composer XE 2013 for C, C++, and Fortran supports a subset of new features of OpenMP 4.0, including OpenMP target constructs, SIMD directives, and the OpenMP affinity feature. Future versions of Intel Composer XE will ship with support for user-defined reductions and task dependencies, as well as support for cancellation. The work for OpenMP version 4.1 and 5.0 has already begun by the OpenMP Architecture Review Board, and Intel is on board to support the development of the next, even better, OpenMP.
Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that
are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and
other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on
microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended
for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel
microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding
the specific instruction sets covered by this notice.

Notice revision #20110804