Software

Using OpenMP* effectively on Theta - Hands-on labs

Carlos Rosales-Fernandez

Access and getting the files

In this hands-on session you will practice OpenMP* tasking, SIMD, and affinity settings. Three exercises are provided, with submission scripts for Theta and proposed solutions.

To get started, copy the files to a directory of your choosing in the **/projects** area:

```
$ tar xzvf /projects/SDL_Workshop/crosales/SDL_2018/omp.tar.gz
```

Then change into the **omp** directory:

```
$ cd ./omp
```

Lab 1 - Affinity control with OpenMP*

Lab 1 - OpenMP* Affinity Control

We will use a simple hand-written matrix-matrix multiplication example to illustrate the effect of affinity on runtime.

To get started, change into the **affinity** directory:

```
$ cd ./affinity
```

Inside this directory you will find a simple **build.sh** script and COBALT submission script - **affinity.run.**

Start by executing the build script:

```
$ ./build.sh
```

This will generate the **mat.omp** executable that you need to complete this exercise.

Lab 1 - OpenMP* Affinity Control

Submit the **affinity.run** script to run the example code with a variety of affinity settings and thread counts:

```
$ qsub ./affinity.run
```

This will generate an output file, **affinity.out**, which contains details of each run configuration and the approximate performance achieved.

Inspect the output file and try to answer the following questions:

- What seems to be the best affinity setting combination for this code?
- What is the speedup achieved by using optimal affinity settings?
- Can you modify the submission script to add other affinity settings (or thread counts) and test to see if there are alternatives that work better?

Lab 1 - Solution

The best combination should be using the following:

- OMP_NUM_THREADS=64
- OMP_PLACES=cores
- OMP_PROC_BIND=spread

Note the following characteristics:

- Since KNL is capable of issuing 2 vector instructions per core per cycle from a single thread ,there is no need to go over 64 threads to achieve maximum performance in a code of this type - Feel free to try and measure the performance.
- Using a compact affinity setting leaves cores unused and leads to lower overall performance.

Lab 2 - Basic Task concepts

Lab 2 - Basic Task Generation and Execution

In this example you will build a simple code that uses tasks to print out the simple sentence:

```
Hello World from OpenMP!
```

First, change to the basic directory:

```
$ cd ./basic
```

Now edit the provided sequential version **basic.c** so that each of the words in the sentence is printed to screen from a separate task. Remember that you will have to:

- Define a parallel region
- Generate the tasks within a single construct

Compile your new version (don't forget the -**qopenmp** flag) and ensure there are no compilation errors.

Lab 2 - Testing

Now launch the provided **basic.run** script so that you can see the output of your code when using multiple threads:

```
$ qsub ./basic.run
```

The script assumes your executable is called **a.out**, and profides the output in file **basic.out**.

Did the sentence come out correctly? It is unlikely, unless you used any type of synchronization - if you did you are ahead of the game - congratulations!

Now try to come up with **two** implementations that write the output in order while still using the same number of tasks. Do not worry about serialization - this exercise is not about performance, but methodology.

Lab 2 - Solution 1

In solution 1 we simply place a **taskwait** statement in between each printf command, so that the output is serialized.

This is a simple way of ensuring order but, in more complex problems it completely defies the purpose of using OpenMP* in the first place.

```
#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp task
        printf("Hello ");
        #pragma omp taskwait
        #pragma omp task
        printf("World ");
        #pragma omp taskwait
        #pragma omp task
        printf("from");
        #pragma omp taskwait
        #pragma omp task
        printf("OpenMP!");
    }
}
```

Lab 2 - Solution 2

In solution 2 we use the alternative method of defining dependencies among tasks.

In this simple example the result is the same - complete reordering at the expense of full serialization.

But in more complex codes defining dependencies may allow for greater parallel execution opportunities at runtime.

```
#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp task depend(out:a)
        a = printf("Hello ");

        #pragma omp task depend(in:a) depend(out:b)
        b = printf("World ");

        #pragma omp task depend(in:b) depend(out:c)
        c = printf("from");

        #pragma omp task depend(in:c)
        printf("OpenMP!");
    }
}
```

Lab 3 - Fibonacci generator

Lab 3 - A Simple Fibonacci Number Generator

The Fibonacci series is an integer series defined by having numbers which, after the first one, are the sum of the previous two in the series:

```
1, 1, 2, 3, 5, 8, 13, 21, …
```

A simple Fibonacci generator can be coded as a recursive function:

```
int main(int argc,
         char *argv[])
{
    ...
    answer = fib( number );
    ...
}
```

```
int fib( int n)
{
    if( n < 2 ) return n;
    int i = fib( n - 1 );
    int j = fib( n - 2 );
    return i+j;
}
```

Your mission, should you choose to accept it, is to create a new version of this function that can be executed in parallel using OpenMP* constructs.

The following slides guide you through the process, and point to a solution in case you get stuck.

Lab 3 - Getting started

First go to the Fibonacci directory:

```
$ cd ../fibonacci
```

Inside this directory you will find three subdirectories named ver0, ver1, ver2. They each correspond to a version of the Fibonnaci number generator:

- ver0 - serial implementation, for reference and getting started.
- ver1 - proposed simple tasking solution
- ver2 - proposed refined tasking solution

Start by making a copy of version 0 so that you can work with it and still have a clear reference code to go back to:

```
$ cp ./ver0/* ./
```

Lab 3 - Some Hints

I'm not going to tell you exactly how to do this, but remember two critical things:

1. You MUST initiate the task generation process inside a single region within a parallel OpenMP* region - in this case main would be the right place to do this.
2. If you look inside the fib.c source file you will see that the fib() function either returns immediately or has two independent tasks to perform.
3. Once those tasks are performed their value is added and returned - perhaps an appropriate location for a synchronization point.

Try to use this hints and what you have learned to parallelize this recursive code using OpenMP* tasks.

Next slide has the answer if you get stuck!

Lab 3 - Proposed Solution (ver1)

Our proposed solution has a single task entering the function fib() from main(). It then generates two additional tasks to execute calls to fib() independently for (n-1) and (n-2):

```
int main(int argc,
         char *argv[])
{
    ...
    #pragma omp parallel
    {
        #pragma omp single
        {
            #pragma omp task
            answer = fib( number );
        }
    }
    ...
}
```

```
int fib( int n)
{
    if( n < 2 ) return n;
    int i, j;
    #pragma omp task shared(i)
    {
        i = fib( n - 1 );
    }
    #pragma omp task shared(j)
    {
        j = fib( n - 2 );
    }
    #pragma omp taskwait
        return i+j;
}
```

Lab 3 - Analysis of the Solution

Whether using your own version or the proposed solution in directory **ver1**, submit a quick job to determine how scalable your implementation is:

```
$ qsub ./tasking.run
```

This will save the number of threads and the time taken to determine the 41st number in the Fibonacci series to an output file called **tasking.out**.

- What is the best speedup you can get out of this code, from 4 to 128 threads?
- Is this faster or slower than the original serial implementation?
- Can you think of any way to improve the proposed solution?

Lab 3 - A Better Solution (ver2)

It turns out that the proposed solution in **ver1** works correctly, but generates excessive overhead by generating too many tasks.

Ideally one would include a variable threshold below which a serial function is used rather than a parallelized one. This is what the solution in the directory **ver2** provides.

Try to develop your own version of this hybrid code that enables better workload balance or, if you prefer, look at the solution provided in **ver2** and described in the next slide.

Go to the **ver2** directory (or use your own solution) to submit the **tasking.run** script to complete a new scalability analysis. Can you see the difference in scalability and speedup?

Feel free to change the value of the defined “SPLITTER” variable and observe its effects on performance. Remember you will need to recompile the code each time you make a change to this variable.

Lab 3 - Proposed Solution (ver2)

Our proposed solution does not create a new task once a small enough **n** is reached:

```
int main(int argc,
         char *argv[])
{
    ...
    #pragma omp parallel
    {
        #pragma omp single
        {
            #pragma omp task
            answer = fib( number );
        }
    }
    ...
}
```

```
int fib( int n)
{
    if( n < 2 ) return n;
    int i, j;
    #pragma omp task shared(i) if(n>30)
    {
        i = fib( n - 1 );
    }
    #pragma omp task shared(j) if(n>30)
    {
        j = fib( n - 2 );
    }
    #pragma omp taskwait
        return i+j;
}
```

Legal Disclaimer & Optimization Notice <w/o benchmarks>

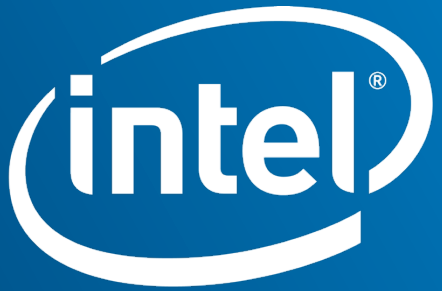
Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

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



Software