02258 Parallel Computer Systems Fall 2022

Second Report

Tomás Estácio
Department of Applied
Mathematics and Computer
Science
Technical University of Denmark
Lyngby, Copenhagen, Denmark
s223187@dtu.dk

1 Databar exercise 4: Introduction to OpenMP

1.1 Writing a parallel OpenMP version of *calc.c*

In the previous Databar, we addressed the possibilities of use parallelization on the program called *calc.c* on specific parts of the code, because there were still some parts that needed to remain serial due to specific algorithmic limitations, like maintaining the order of prints and mutual dependencies.

Focusing more on the parts of the code where it is possible to perform parallelism, I was able to create a version of the program that takes advantage of multiple threads (up to 16) working at the same time, specifically distributing the iterations of two loops to the threads that were available.

To implement OpenMP parallelism in code, firstly I had to include the *omp.h* library available for C programming in the code, use the *#pragma omp parallel* directive to initialize the parallel region of the code, and perform only the first loop in parallel using the *#pragma omp for nowait schedule(static, chunksize)* directive and scheduler, as the loop can be faster when done in parallel and divided in chunks of iterations (*chunksize*) according to the number of threads executing, and does not need to wait for the threads to be synchronized. Then, I initialized another parallel region with a *for* loop inside, using the same directives previously described. However, after the execution of the loop, this time there was a need to synchronize the threads working on the loop, because they were about to enter a region with shared dependencies, so I used the *#pragma omp barrier* directive to do so and closed the parallel region. For each number of threads from 1 to 16, I executed the program 5 times, did the average of the values of the wall time and the CPU time, measured with the *time* linux command, and plot them in graphics. For the graphic that had the CPU time measured for different number of threads used, it is possible to conclude that, as the program is using multiple threads scheduled on multiple cores, they have more cycles spent on executing the actual logic of the program and not spent being in the "idle" state. For the graphic related to the wall time measured, it has an exponential decay while increasing the number of threads used, because the parallelization allows to perform more instructions at the same time, reducing the real time of the program.

Next, I plotted a graph for the measured speedup, using the formula:

$$S = \frac{Real\ time\ measured\ for\ a\ single\ thread}{Real\ time\ measured\ for\ n\ threads}$$

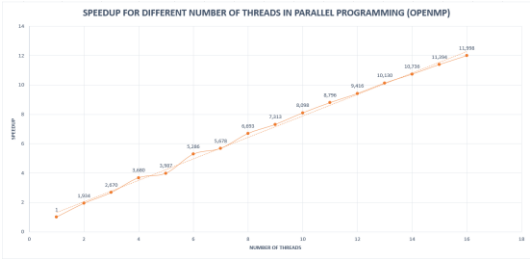

Figure 1: Graphic showing speedup measured with different number of threads

Following the speedup, it is also important to measure the parallel efficiency obtained when using different number of threads in a program, by calculating:

$$E = \frac{S}{p}$$

being *E* the parallel efficiency, *S* the speedup and *p* the number of threads used.

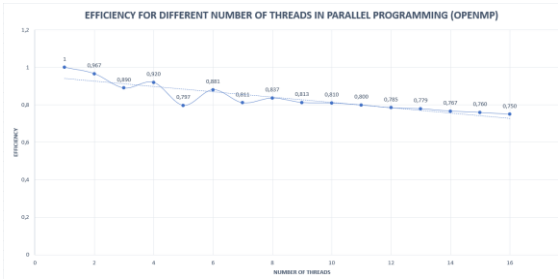

Figure 2: Graphic showing parallel efficiency with different number of threads

These results for the speedup and the parallel efficiency of the program are not optimal, as I got a speedup that is not proportional to the increase in the number of threads and a

decreasing trend for the efficiency of the program. There are a few reasons associated with these results, like load imbalance on the parallelized loops, serial code overhead and the parallel code overhead, such as threads start-up and termination time, synchronization for accessing to critical regions in the code (for example, when there is the trade of the values of the array *from* with the values of the array *to*, right after changing its elements) and OpenMP directives and schedulers overhead.

2 Databar exercise 5: MPI

2.1 Submitting MPI Jobs

For this Databar, initially I was asked to use a DTU's cluster system, managed and scheduled by the LSF application node. To do so, I analyzed the job script *run.sh* and verified that the command **BSUB** is used in the script to send jobs to the specified HPC's queue. It is also possible to see that the program that will be run by the script is *hello_mpi*, as it states in the last two lines of the file: *module load mpi*, that loads the MPI module before the compiler can be used, and *mpirun ./hello_mpi > mpi.log* that executes the program *hello_mpi.c*, assumed to be compiled beforehand using the *mpicc* command, and saves the output on the *mpi.log* file.

All the lines in the *run.sh* file that start with **#BSUB** are arguments to the resource manager interpret, like:

1. **#BSUB -q hpc**, specifies that the jobs to run will be placed in the queue, with the flag name "hpc". .
2. **#BSUB -J My_Application**, specifies the name of the job, in this case "MyApplication", to be easier to access to check its status.
3. **#BSUB -n** 4, reserves 4 cores for the job I want to run.
4. **#BSUB -R "rusage[mem=2GB]**, specifies that the job will run on a machine that offers at least 2GBytes per core of memory available
5. **#BSUB -R "span[hosts=1]"**, specifies that all the cores used must be on a single host, which means that it's not possible to request more cores than the number of physical cores present on a machine.
6. **#BSUB -M 3GB**, sets the limit of memory per processor for all the processors used by the job (if the job exceeds 3GBytes per core, it is killed).
7. **#BSUB -W 24:00**, sets the limit of wall time for the program.
8. **#BSUB -u** s223187@dtu.dk, sets the email address to receive updates on the job.
9. **#BSUB -B**, writes an email when the job starts.
10. **#BSUB -N**, writes an email when the job ends.
11. **#BSUB -o Output_%J.out**, specifies the output file.
12. **#BSUB -e Error_%J.err**, specifies the error file.

For the following question, I used a simple MPI program with the name *hello_mpi.c*, which initiates the parallel environment with a call of the *MPI_Init()* function. Upon initialization, the world communicator *MPI_COMM_WORLD* is called, defining a group of processes that can be referred to by a communicator handle, the "piece" of the program that hold the group of processes used. The functions *MPI_Comm_size()* and *MPI_Comm_rank()* gives the number of processes used in the program and the unique ranks that each process has. Next, the program prints a simple message that is going to be executed by all the 4 processors that are assigned to perform the task in parallel. Finally, to indicate that the parallel program should finish, we terminate it using the *MPI_Finalize()* function.

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv) {
  int rank, size;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   printf("Hello World, I am %d of %d\n", rank, size);

   MPI_Finalize();
   return 0;
}
```

Figure 3: Parallel program in C using MPI, "hello_mpi.c"

The MPI standard does not specify the order of the processes, as we can see in the output of the program, because the first process to print is the one with a rank of 0, the second has rank of 2, the third has rank of 3 and the fourth one has rank of 1. The output of this simple program is as expected, according to the analysis I've made of the code in the previous paragraph.

2.2 Distributed calculations

For this exercise, I was told to write a parallel version of the *calc.c* program using MPI. Firstly, I run the *MPI_Init* function to initialize the parallel environment, where multiple processes run at the same time and use the functions *MPI_COMM_SIZE* and *MPI_COMM_RANK* to determine the number of processes running and the rank of the process that is running. These values were useful to perform the division of the loops to parallelize, as I needed to divide the number of iterations of each loop so that each process only works on a part of the loop, taking advantage of the parallelism, leaving a few iterations, if the division between the *INTERVALS* variable and the number of processors has a rest, to be performed as serial by the process with rank 0 (root process), because it isn't a problem in terms of the global efficiency of the program. However, an MPI environment does not have a shared memory between the processes, so if some process changes the value of a shared variable, the value will not be automatically updated when another process tries to access it. With that in mind, after performing the parallelization of the first loop, I gather all the updated values of the *from* array in the root process with the *MPI_Gather* function. For the second loop inside the *while* loop, I perform the same parallelization described and then broadcast the first and last elements of the handled part by each process, the indexes of those elements and the elements that belong to the "tail" of the array, to every process, using the *MPI_Bcast* function. After the *while* loop finishes, I gather again all the updated values of the *to* array in the root process using *MPI_Gather*.

Since there are some *printf* functions calls in the program and I only needed that one process printed the values obtained, I made sure that only the process with rank 0 would print the values obtained, after I updated all the values of the arrays.

Using the same formula already discussed previously when measuring the speedup of the program, I plotted the following graphic:

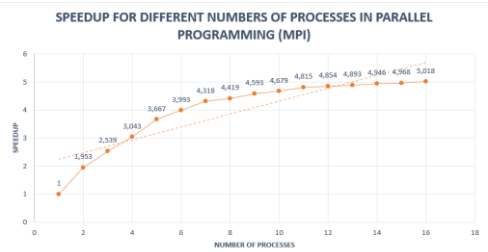

Figure 4: Graphic showing speedup with different numbers of processes

These values were obtained by analyzing the program, figure out the best regions to parallelize and then using some MPI performing tools to aggregate statistics of the program at run time (number of messages sent, amount of time spent on MPI environment, etc.), to have a better understanding of the performance problems associated with the program. After that, I was able to reduce the number of messages that I had to send when I realize that there was no need to send all the *from* array in each iteration of the *while* loop, just the first and last element of the part of the array worked by each process. However, comparing to the results obtained using OpenMP, there's a decrease in the speedup and in the parallel efficiency of the program, as we can analyze by the graphics.

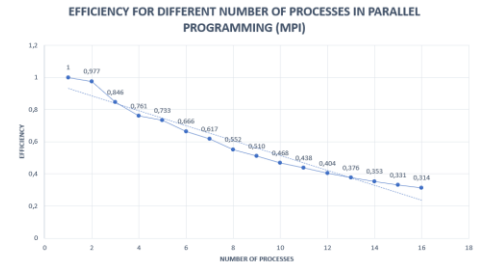

Figure 5: Graphic showing parallel efficiency with different numbers of processes

These results can be explained by the adding underlying overheads associated with the use of the MPI on the program, parallelized using a distributed system of processes in the HPC, which communicate by function calls in the code, complicating the code and decreasing efficiency, especially some of the functions used, like *MPI_Gather* and *MPI_Bcast*, due to being collective and blocking functions, meaning that the processes have to wait until the communication is completed, making synchronization of the processes in places where it is not necessary.

It is also possible to verify that the speedup of the program begins to slow down with the increase of the number of processes used, which can be explained by the fact that the more processes used also means more communication overhead between them.

Since the program used must access a lot of memory addresses, the use of MPI is justified because it offers more memory in real time. However, the communication overhead associated with the operations performed in shared variables and, therefore, the values need to be updated to all processes, has a large cost in the program efficiency.

3 Databar exercise 6: OpenCL on GPUs

3.1 Exercise01

For this Databar exercise, I started by downloading the file with the exercises regarding OpenCL programs running in CPUs and GPUs, starting with Exercise01.

Firstly, I was asked to examine the program *DeviceInfo.c*, which displays information about the platforms running OpenCL and the devices existing in each platform, such as the platform's name, vendor, OpenCL version, number of devices, device IDs, and the device's name, OpenCL version, maximum compute units, local memory size, global memory size, maximum buffer allocation size, work group size, and maximum dimensions of the work groups. The 4 OpenCL API calls are referenced bellow in the REFERENCES chapter, which describe the function and the arguments it takes.

After compiling and running the program for the CPU and the GPU, I get some information, as expected, regarding the software platform version of OpenCL on the HPC connected (platform) and the OpenCL version of the capabilities of the hardware (device). For the CPU, the platform used was AMD Accelerated Parallel Processing and, for the GPU, the platform used was NVIDIA CUDA.

3.2 Exercise02

The host program *vadd_c.c* has the purpose of computing the sum of two arrays, *a[]* and *b[]*, and storing the result in the array *c[]*. Firstly, there is the definition of the platform, setting up the platform and GPU device used, by using OpenCL API calls, like the ones described earlier. Next, the program creates the context and establishes a command queue to feed the selected device. Then, the program builds the program object, defining the source code for the kernel-program, referenced by the variable *KernelSource*, followed with the creation of the compute kernel called *vadd*. After these definitions, there is the setup of the memory objects, creating buffers for the arrays, writing the *a[]* and *b[]* arrays and using these to pass as arguments in commands to the kernel function, so that the kernel performs the operations necessary to achieve the goal of the program. After enqueuing the kernel and waiting for the execution of the kernel program, the programs calculate the time spent executing it and tests the results obtained with a certain tolerance. The 18 OpenMP API calls used in the program are referenced bellow in the REFERENCES chapter, which describe the function and the arguments it takes.

The output was as expected: prints information about the CPU used, gives an average execution time of the kernel function of 0,017546 seconds, and verifies all the results as correct.

3.3 Exercise05

For this exercise, I worked with the code of the accelerator kernel which performs a similar computation as the one in Exercise02, adding one array *d[]* to the operation, making it *c[] = a[]+b[]+d[]*. To change the code to run on a CPU, I had to change the second argument of the function call *clGetDeviceIDs()* that defines the type of OpenCL device used to *CL_DEVICE_TYPE_CPU*. The output of the program was the information about the device used, the run time of the kernel code performing the operation described, which was 0,005633 seconds, and the confirmation of the results obtained by the kernel.

To run now on a GPU, I had to set up the environment in the terminal used to access the GPUs available, change the type of OpenCL device used to *CL_DEVICE_TYPE_GPU*, clean the make file used before and execute the program, which gave a different output. Firstly, regarding the device info, the GPU has a maximum of 80 compute units, while the CPU only has a maximum of 24, because the GPU has more cores and can perform the operations required in parallel on multiple sets of data, like the arrays in this case. That way, it becomes easier to understand that the time spent by the kernel performing the operations is a lot less using the GPU (0,000045 seconds), comparing to the time spent using the CPU. In conclusion, the speedup of the program executed by the GPU is approximately 125 times better than the program executed by the CPU, which proves that has a better performance, mainly due to the massive parallelism we can perform on the same instructions repeatedly, but also because the program works with four arrays of 1024 float elements, so the memory management isn't a significant problem to the overall performance.

3.4 Calc revisited

The *calcOpenCL.c* program has some features which take advantage of the multiple cores of the GPU operating at the same time to perform the same operation inside the loops that were defined in the **KernelSource*, in separate compute kernels, since we had to use different global IDs, one for each loop. However, the program can still suffer some overhead associated with the memory communication between the host, the device and the workgroups, which are still considerable since the size of the array is very large.

After creating the program and setting up the environment to run it using the GPU available, I managed to achieve a run time of 0,3 seconds in average, using the *time* command, which is lower than any of the values obtained using OpenMP or MPI in the same program. For that reason, it is possible to assume that, taking advantage of the parallelization using the GPU, a better performance is achievable comparing to the use a CPU for this code, mainly due to the several repetitions of the same operations, which are perfect for the GPU to act on. To measure the kernel time in each the compute kernels created, I used the *wtime.c* function available, giving a better insight on the performance of this OpenCL implementation, getting an average time of 0,00695 seconds for the execution of the kernel that performs the initialization of the *from* array with the value 0,0 and of 0,006702 seconds for the execution of the second loop, the one inside the *while* loop on the *calc.c* program.

Finally, to achieve a reasonable measure of the speedup, I needed to divide the execution time obtained for the sequential program written with the execution time obtained using the parallelization of the GPU. Executing the sequential code 5 times with no compiler optimization flag, I got an average value of 4,387 seconds, and for the parallelized version I got an average of 0,299 seconds, so the speedup obtained was of 14,67, which is a larger number than any of the speedups obtained previously. For calculating efficiency, I decided to calculate the theorical peak performance = 2 * Number of cores used (2560) * Frequency (1245GHz) * Floating-point operations (5) = 31,187 TFlops. These values were deduced by analyzing the code and the specifications about the GPU used. Then, by dividing the actual peak performance possible of 7,0 TFlops with the theorical one, I get an efficiency of only 22,44% for the program.

REFERENCES

[1] Chapman & Hall, (2019), Introduction to High Performance Computing for Scientists and Engineers.

[2] Jun Zhang, Department of Computer Science, University of Kentucky, Parallel Computing, Chapter 7 Performance and Scalability, https://dl.icdst.org/pdfs/files3/9f9bff8c81b41a685f01ec5ed7e339a4.pdf.

[3] LSS Wiki, (15/05/2018), Parallel efficiency - simple approach, https://wikis.ovgu.de/lss/doku.php?id=guide:parallel_efficiency.

[4] Mark Roth, Micah J Best, Craig Mustard, Alexandra Fedorova, Deconstructing the Overhead in Parallel Applications, https://people.ece.ubc.ca/sasha/papers/iiswc-2012.pdf.

[5] Akira, How to Get Good Performance by Using OpenMP, http://akira.ruc.dk/~keld/teaching/IPDC_f10/Slides/pdf4x/4_Performance.4x.pdf.

[6] The Khronos Group Inc., clGetDeviceIDs, https://registry.khronos.org/OpenCL/sdk/1.0/docs/man/xhtml/clGetDeviceIDs.html.

[7] The Khronos Group Inc., clGetDeviceInfo, https://registry.khronos.org/OpenCL/sdk/1.0/docs/man/xhtml/clGetDeviceInfo.html.

[8] The Khronos Group Inc., clGetPlatformIDs, https://registry.khronos.org/OpenCL/sdk/1.0/docs/man/xhtml/clGetPlatformIDs.html.

[9] The Khronos Group Inc., clGetPlatformInfo, https://registry.khronos.org/OpenCL/sdk/1.0/docs/man/xhtml/clGetPlatformInfo.html.

[10] The Khronos Group Inc., clCreateContext, https://registry.khronos.org/OpenCL/sdk/1.0/docs/man/xhtml/clCreateContext.html.

[11] The Khronos Group Inc., clCreateCommandQueue, https://registry.khronos.org/OpenCL/sdk/1.0/docs/man/xhtml/clCreateCommandQueue.html.

[12] The Khronos Group Inc., clCreateProgramWithSource, https://registry.khronos.org/OpenCL/sdk/1.0/docs/man/xhtml/clCreateProgramWithSource.html.

[13] The Khronos Group Inc., clCreateKernel, https://registry.khronos.org/OpenCL/sdk/1.0/docs/man/xhtml/clCreateKernel.html.

[14] The Khronos Group Inc., clBuildProgram, https://registry.khronos.org/OpenCL/sdk/1.0/docs/man/xhtml/clBuildProgram.html.

[15] The Khronos Group Inc., clCreateBuffer, https://registry.khronos.org/OpenCL/sdk/1.0/docs/man/xhtml/clCreateBuffer.html.

[16] The Khronos Group Inc., clEnqueueWriteBuffer, https://registry.khronos.org/OpenCL/sdk/1.0/docs/man/xhtml/clEnqueueWriteBuffer.html.

[17] The Khronos Group Inc., clSetKernelArg, https://registry.khronos.org/OpenCL/sdk/1.0/docs/man/xhtml/clSetKernelArg.html.

[18] The Khronos Group Inc., clEnqueueNDRangeKernel, https://registry.khronos.org/OpenCL/sdk/1.0/docs/man/xhtml/clEnqueueNDRangeKernel.html.

[19] The Khronos Group Inc., clFinish, https://registry.khronos.org/OpenCL/sdk/1.0/docs/man/xhtml/clFinish.html.

[20] The Khronos Group Inc., clEnqueueReadBuffer, https://registry.khronos.org/OpenCL/sdk/1.0/docs/man/xhtml/clEnqueueReadBuffer.html.

[21] The Khronos Group Inc., clReleaseMemObject, https://registry.khronos.org/OpenCL/sdk/1.0/docs/man/xhtml/clReleaseMemObject.html.

[22] The Khronos Group Inc., clReleaseProgram, https://registry.khronos.org/OpenCL/sdk/2.0/docs/man/xhtml/clReleaseProgram.html.

[23] The Khronos Group Inc., clReleaseKernel, https://registry.khronos.org/OpenCL/sdk/1.0/docs/man/xhtml/clReleaseKernel.html.

[24] The Khronos Group Inc., clReleaseCommandQueue, https://registry.khronos.org/OpenCL/sdk/1.1/docs/man/xhtml/clReleaseCommandQueue.html.

[25] The Khronos Group Inc., clReleaseContext, https://registry.khronos.org/OpenCL/sdk/1.0/docs/man/xhtml/clReleaseContext.html.

[26] Microway, In-Depth Comparison of NVIDIA Tesla “Volta” GPU Accelerators, https://www.microway.com/knowledge-center-articles/in-depth-comparison-of-nvidia-tesla-volta-gpu-accelerators/.
[27] Simon McIntosh-Smith, Tom Deakin, (Nov 2014), Hands On OpenCL.