Programming with OmpSs (A brief introduction)

Hardware Configuration and Login Information

-- **Minotauro** -- - 128 Bull Blade B505 (126 + 2 login) - 2 Intel E5649 (6-core/2.53GHz/12MB cache) - 24GB RAM - 2 NVIDIA M2090 - 1x SSD 250GB - Network -2x IB HCA QDR (GPUDirect) -2x 1Gb Eth (admin / GPFS)	Change password node: - **dl01.bsc.es** Login nodes: - **mt1.bsc.es** - **mt2.bsc.es** username: **nct99999**[*] passwd: ****** (*) Last five digits and password will be provided during the hands-on session.

Getting the exercises (*and configuring OmpSs*)

Connect to a Minotauro's *Login Node* (see information above about available login nodes) using your *username* and your *password*. It is important to enable X11 forwarding when starting your ssh session due we will use visualization tools.

Copy the .tar.gz file from ~nct01075/ directory to your home and unpack it (using **tar**). It will create a new directory called ompss-exercises-MT. Inside the directory you will find a configure script file. Make sure you execute **source** on that file to configure your environment in order to use OmpSs tools.

```
$ ssh -X username@login-node
login as: nct99999
nct99999@login-node's password: ******
Last login: Thu Jan 01 00:00:00 1970 from 00.00.00.00
nct99999@login-node:~>tar -xvf ~nct01075/ompss-exercises-MT.tar.gz
nct99999@login-node:~>cd ompss-exercises-MT
nct99999@login-node:~>source configure.sh
```

All OmpSs exercises come with a makefile (***Makefile***). Some of them are also configured to compile 4 different versions for each program. Binary files end with a suffix which determines the version: ***program*-s** (sequential version, ignoring directives), ***program*-p** (performance version), ***program*-i** (instrumented version) and ***program*-d** (debug version). You can actually select which version you want to compile by executing: "make *program-version*" (e.g. in the Cholesky kernel you can compile the performance version executing "make cholesky-p". By default (running make with no parameters) all the versions are compiled.

Each exercise may have two job scripts. One is running a single program execution (***run-1.sh***), the other is running multiple configurations with respect the number of threads, data size, etc (***run-n.sh***). Before submitting any job, make sure all environment variables have the values you expect to. The job would be submitted using: "mnsubmit <job_script>". While the jobs are queued you can check their status using the command "mnq" (it may take a while to start executing). Once a job has been executed you will get two files. One for *console standard output* (with .out extension) and other for *console standard error* (with .err extension).

Programming with OmpSs (A brief introduction)

1. (a) Compiling and Executing OmpSs Programs

Cholesky

In this exercise we will work with the Cholesky kernel. This algorithm is a decomposition of a Hermitian, positive-definite matrix into the product of a lower triangular matrix and its conjugate transpose.

Edit Makefile in order to build performance, instrumented and debug OmpSs versions. Add compiler common flags in CFLAGS, performance flags in CFLAGS_P, instrumentation flags in CFLAGS_I and debug flags in CFLAGS_D makefile variables.

```
PROGRAM  = cholesky

CFLAGS   = # compiler common flags
CFLAGS_P = # compiler performance flags (if needed)
CFLAGS_I = # compiler instrumented flags
CFLAGS_D = # compiler debug flags
```

Compile the program (running "make") and execute performance version (using "mnsubmit <*job_script*>"). Remember that for every application we provide two running scripts (see Executing in Queues section). Before submitting any job, make sure it uses the correct version you want to run and it declares all enviroment variables you want to declare. Once the job has been executed, check the console files (.out and .err) and discuss the results.

In order to create a dependence graph we have to submit the instrumented version of the application. You can create it using a single-execution script (**run-1.sh**) running the instrumented version (**cholesky-i**). Before submitting the job check if the corresponding runtime options (e.g. NX_INSTRUMENTATION=graph) have been declared. In order to get a complete dependence graph output you will need to execute your program with just one thread (i.e. NX_THREADS=1) and no task creation throttling policy (i.e. NX_THROTTLE=dummy). You can use the document viewer tool **evince** just to visualize the graph:

```
nct99999@login-node:~>evince graph.pdf
```

Stream

The stream benchmark is part of the HPC Challenge benchmarks (http://icl.cs.utk.edu/hpcc/) and here we present two versions: one that inserts barriers (**stream_barr**) and another without barriers, but using dependences (**stream_deps**). The behavior of version with barriers resembles the OpenMP version, where the different functions (Copy, Scale, ...) are executed one after another for the whole array while in the version without barriers, functions that operate on one part of the array are interleaved and the OmpSs runtime keeps the correctness by means of the detection of data-dependences.

As in the previous exercise: check the Makefile compiler flags, compile the program, submit a performance version and compare stream barrier and dependence versions. You can also get a dependence graph and compare results between these two versions. (NOTE: In order to better undestand the problem structure you can reduce the number of iterations that performs the application from 10 to just 1. Edit **stream.c** file, look for #define NTIMES 10, and change this value to 1).

1. (b) Getting Paraver Traces

In order to create a paraver tracefile program execution we have to submit the instrumented version of the application. You can create it using a single-execution script (**run-1.sh**) running the instrumented version. Before submitting the job check if the corresponding runtime options (e.g. NX_INSTRUMENTATION=extrae) have been declared.

Once the job has been executed we will have a trace in the job's working directory. A trace is composed of three different files: the trace file (.prv), the trace configuration (.pcf) and other information (.row). Execute Paraver command (**wxparaver**) and open the trace through the menu option [File / Load Trace...] or you can also open the trace by executing Paraver with the tracefile as a parameter (e.g. “wxparaver cholesky-i_001.prv”). Use the Paraver config files included in the paraver directory to visualize the program execution. You can use, among others, **thread_state.cfg** (included in runtime subdirectory) or **task_name_and_location.cfg** (included in tasks subdirectory), but feel free to load other configuration files and experiment with them.

2. Basic Parallelization using OmpSs

Dot Product

The dot product is an algebraic operation that takes two equal-length sequences of numbers and returns a single number obtained by multiplying corresponding entries and then summing those products. This is the code that you can find in **dot_product()** function:

```
   for (long i=0; i<N; i+=CHUNK_SIZE) {
      actual_size = (N-CHUNK_SIZE>=CHUNK_SIZE)?CHUNK_SIZE:(N-CHUNK_SIZE);
      C[j]=0;
      for (long ii=0; ii<actual_size; ii++)
         C[j]+= A[i+ii] * B[i+ii];
      }
      acc += C[j];
      j++;
   }
```

This simple mathematical operation is performed on 2 vectors of N-dimension and returns a scalar. It is interesting (at least from the programming model point of view) because you have to accumulate your result on a single variable (**acc**), and you want to do this from multiple threads at the same time. This is called a reduction and there are several ways to do it:

- Protect the reduction with a lock or atomic clause, so that only one thread increments the variable at the same time. Note that locks are expensive.
- Specify that there is a dependency on this variable, but choose carefully, you don't want to serialize the whole execution! In this exercise we are incrementing a variable, and the sum operation is commutative. OmpSs has a type of dependency called commutative, designed specifically for this purpose.
- Use a vector to store intermediate accumulations. Tasks operate on a given position of the vector (the parallelism will be determined by the vector length), and when all the tasks are completed, the contents of the vector are summed.

In that code we have already annotated the tasks for you, but this parallel version is not ready, yet.

- Find all the #pragma omp lines. As you can see, there are tasks, but we forgot to specify the dependencies.
- There is a task with a label dot_prod. What are the inputs of that task? Does it have an output? What is the size of the inputs and outputs? Annotate the input and output dependencies.
- Below the dot_prod task, there is another task labeled as increment. What does it do? Do you see a difference from the previous? You have to write the dependencies of this task again, but this time think if there is any other clause (besides in and out) that you can use in order to maximize parallelism.

Compile the program (running "make") and execute it (using "mnsubmit <*job_script*>"). Before submitting any job, make sure it uses the correct version you want to run and it declares all enviroment variables you want to declare. Once the job has been executed, check the console files (.out and .err) and discuss the results.

Multisort (explicit synchronization)

Multisort application, sorts an array using a divide and conquer strategy. The vector is split into 4 chunks, and each chunk is recursively sorted (as it is recursive, it may be even split into other 4 smaller chunks), and then the result is merged. When the vector size is smaller than a configurable threshold (MIN_SORT_SIZE) the algorithm switches to a sequential sort version.

```
   if (n >= MIN_SORT_SIZE*4L) {
      // Recursive decomposition
      multisort(n/4L, &data[0], &tmp[0]);
      multisort(n/4L, &data[n/4L], &tmp[n/4L]);
      multisort(n/4L, &data[n/2L], &tmp[n/2L]);
      multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);

      merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
      merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);

      merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
   } else {
      // Base case
      basicsort(n, data);
   }
```

As the code is already annotated with some task directives, try to compile and run the program. Is it verifying? Why do you think it is failing? Running an unmodified version of this code may also raise a Segmentation Fault. Investigate which is the cause of that problem. Although it is not needed, you can also try to debug program execution using **gdb** debugger (with the ompss debug version):

```
nct99999@login-node:~> NX_THREADS=4 gdb --args ./multisort-d 4096 64 128
```

After solving the problem, compile the program (running “make”) and execute it (using “mnsubmit <*job_script*>”). Before submitting any job, make sure it uses the correct version you want to run and it declares all enviroment variables you want to declare. Once the job has been executed, check the console files (.out and .err) and discuss the results.

Matrix Multiplication

This example performs the multiplication of two matrices (A and B) into a third one (C). Since the code is not optimized, not very good performance results are expected. Think about how to parallelize (using OmpSs) the following code found in **compute()** function:

```
  for (i = 0; i < DIM; i++)
    for (j = 0; j < DIM; j++)
      for (k = 0; k < DIM; k++)
        matmul ((double *)A[i][k], (double *)B[k][j], (double *)C[i][j], NB);
```

This time you are on your own: you have to identify what code must be a task. There are a few hints and this example is quite simple, compared to the previous exercise.

- Have a look at the compute function. It is the one that the main procedure calls to perform the multiplication. As you can see, this algorithm operates on blocks (to increase memory locality and to parallelize operations on those blocks).
- Now go to the matmul function. As you can see, this function performs the multiplication on a block level.
- Annotate the tasks that you consider are necessary, and don't forget to ensure that all of them are finished before returned the result of the matrix multiplication (would it be necessary any synchronization directive to guarantee that result has been already computed?).

Compile the program (running “make”) and execute it (using “mnsubmit <*job_script*>”). Before submitting any job, make sure it uses the correct version you want to run and it declares all enviroment variables you want to declare. Once the job has been executed, check the console files (.out and .err) and discuss the results.

3. Heterogeneity Support in OmpSs

Hybrid Cholesky (CPUs + CUDA GPUs)

In this exercise we will come back to Cholesky Factorization but this time, the four kernels involved in the computation, have been coded using two different architectures (smp and cuda).

```
   for (k = 0; k < nt; k++) {
        CALL_POTRF_TILE(Ah[k*nt + k], ts);
        // Triangular systems
        for (i = k + 1; i < nt; i++)
            trsm_tile(Ah[k*nt + k], Ah[k*nt + i], ts, (nt-i)+10);
        // update trailing matrix
        for (i = k + 1; i < nt; i++) {
            for (j = k + 1; j < i; j++)
                gemm_tile(Ah[k*nt + i], Ah[k*nt + j], Ah[j*nt + i], ts);
            syrk_tile(Ah[k*nt + i], Ah[i*nt + i], ts, (nt-i)+10);
        }
    }
```

Compile the program (running "make") and execute performance version (using "mnsubmit <*job_script*>"). Remember that for every application we provide two running scripts (see Executing Jobs in Queues section). Before submitting any job, make sure it uses the correct version you want to run and it declares all enviroment variables you want to declare. Once the job has been executed, check the console files (.out and .err) and discuss the results.

Submit an instrumented version of the application in order to get a paraver trace using a matrix size of 8192x8192 and a block size of 1024x1024 (i.e. ./cholesky_hyb-i 8192 1024 0). Analyse the trace using **wxparaver** and see which tasks are executed on CPUs and which on GPUs.

Increase the matrix size from 8192x8192 to 16384x16384 keeping the block size of 1024x1024. That will increment the number of tasks. Get a paraver trace and compare the results with the previous execution.

DGEMM (CPUs vs OpenCL GPUs)

In this exercise we will compare performance between CPUs and GPUs. First take a look at the NDRange clause in the target directive when creating a OpenCL task. Default configuration uses a single thread of the GPU device (i.e. ndrange(2,...,...,1,1) values).

```
#pragma omp target device(opencl) copy_deps ndrange(2,nb/4,nb/4,1,1) file(dgemm.cl)
#pragma omp task in([lda*K] A, [ldb*K] B) inout([ldc*K] C)
void dgemm2_6_0_4_ext (unsigned int M, unsigned int N, unsigned int K,
     double alpha, double *A, unsigned int lda, double* B, unsigned int ldb,
     double beta, double *C, unsigned int ldc);
```

Change the NDRange configuration to use different number of GPU threads and measure the performance you get for each of them (try: 1,2 2,1 2,2 4,4 8,8 etc.). When executed, the application reports performance on GPUs and CPUs.

You can also replace the OpenCL kernel file (configured by default to use dgemm1_5_1_0_0_0.cl-config-4-4-8-8 file) and use dgemm2_6_0_4.cl-config-2-4-16-16. Changing the kernel file also requieres to change the NDRange clause. Follow these steps to replace the kernel file:

```
nct99999@login-node:~>rm dgemm.cl
nct99999@login-node:~>ln -s dgemm2_6_0_4.cl-config-2-4-16-16 dgemm.cl
```

Change the NDRange configuration this way:

```
#pragma omp target device(opencl) copy_deps ndrange(2,nb/2,nb/4,16,16) file(dgemm.cl)
#pragma omp task in([lda*K] A, [ldb*K] B) inout([ldc*K] C)
void dgemm2_6_0_4_ext (unsigned int M, unsigned int N, unsigned int K,
     double alpha, double *A, unsigned int lda, double* B, unsigned int ldb,
     double beta, double *C, unsigned int ldc);
```

Get performance results and compare paraver traces for different number of GPU threads (you can also try any of the following configurations: 1,2 2,1 2,2 4,4 8,8 etc.).

Saxpy (OpenCL and CUDA)

In this exercise we will work with the Saxpy kernel. This algorithm sums one vector with another vector multiplied by a constant.

The sources are not complete, but the standard structure for OmpSs CUDA/Kernel is complete:

- There is a kernel in the files (kernel.cl/kernel.cu) in which the kernel code (or codes) is defined.
- There is a C-file in which the host-program code is defined.
- There is a kernel header (kernel_header.h) file which declares the kernel as a task, this header must be included in the C-file and can also be included in the kernel file (not strictly needed).

```
#pragma target device(cuda) copy_deps ndrange( /*???*/ )
#pragma omp task in([n]x) inout([n]y)
__global__ void saxpy(int n, float a, float* x, float* y);
```

As you can see, we have two vectors (x and y) of size n and a constant a. They specify which data needs to be copied to our runtime. In order to get this program working, we only need to specify the NDRange clause, which has three members:

- First one is the number of dimensions on the kernel (1 in this case).
- The second one is the total number of kernel threads to be launched (as one kernel thread usually calculates a single index of data, this is usually the number of elements, of the vectors in this case).
- The third one is the group size (number of threads per block), in this kind of kernels which do not use shared memory between groups, any number from 16 to 128 will work correctly (optimal number depends on hardware, kernel code…).

When the NDRange clause is correct. We can proceed to compile the source code, using the command “make”. After it (if there are no errors), we can execute it using “mnsubmit run-1.sh”.

Krist (OpenCL and CUDA)

In this exercise we will work with the Krist kernel. This algorithm is used on crystallography to find the exact shape of a molecule using Rntgen diffraction on single crystals or powders. We'll execute the same kernel many times.

The sources are not complete, but the standard structure for OmpSs CUDA/Kernel is complete:

- There is a kernel in the files (kernel.cl/kernel.cu) in which the kernel code (or codes) is defined.
- There is a C-file in which the host-program code is defined.
- There is a kernel header (krist_auxiliar_header.h) file which declares the kernel as a task, this header must be included in the C-file and can also be included in the kernel file (not strictly needed).

```
#pragma omp target device(cuda) copy_deps /* ndrange? */
#pragma omp task /* in and outs? */
__global__ void cstructfac(int na, int number_of_elements, int nc, float f2, int NA,
TYPE_A* a, int NH, TYPE_H* h, int NE, TYPE_E* output_array);
```

As you can see, now we need to specify the ndrange clause (same procedure than previous exercise) and the inputs and outputs. As we have done before for SMP (hint: Look at the source code of the kernel in order to know which arrays are read and which ones are written). The total number of elements which we'll process (not easy to guess by reading the kernel) is "number_of_elements".

Remind: Ndrange clause has three members:

- First one is the number of dimensions on the kernel (1 in this case).
- The second one is the total number of kernel threads to be launched (as one kernel threads usually calculates a single index of data, this is usually the number of elements, of the vectors in this case).
- The third one is the group size (number of threads per block), in this kind of kernels which do not use shared memory between groups, any number from 16 to 128 will work correctly (optimal number depends on hardware, kernel code…

Once the NDRange clause is correct and the input/outputs are correctly defined. We can proceed to compile the source code, using the command "make". After it (if there are no errors), we can execute it using "mnsubmit run-1.sh".

Perlin Noise (CUDA)

In this exercise we will work with the Perlin Noise algorithm which is a procedural texture primitive used to simulate elements from nature, and is especially useful in circumstances where computer memory is limited.

Complete the target and task directive annotating the cuda_perlin kernel at perlin_kernel.cuh file:

```
#pragma omp target /*...*/
#pragma omp task /*...*/
__global__ void cuda_perlin (pixel output[], float time, int j, int rowstride, int
img_width, int BS);
```

Hints: Does the task directive need dependences? Which values you will use in the NDRange clause? Does the device to declare copy clauses?

Once you consider the kernel is properly declared with the task and target directive, you can proceed to compile the source code, using the command “make” and execute it using “mnsubmit run-1.sh”.

Matmul (OpenCL and CUDA)

In this exercise we will work with the Matmul kernel. This algorithm is used to multiply two 2D-matrices and store the result in a third one. Every matrix has the same size.

This is a blocked-matmul multiplications, this means that we launch many kernels and each one of this kernels will multiply a part of the matrix. This way we can increase parallelism, by having many kernels which may use as many GPUs as possible.

Sources are not complete, but the standard file structure for OmpSs CUDA/OpenCL Kernel is complete:

- There is a kernel in the files (kernel.cl/kernel.cu) in which the kernel code (or codes) is defined.
- There is a C-file in which the host-program code is defined.
- There is a kernel header file which must declares the kernel as a task, this header must be included in the C-file and can also be included in the kernel file (not strictly needed): matmul_auxiliar_header.h

```
// Kernel declaration as a task should be here
// Remember, we want to multiply two matrices, (A*B=C) where all of them have size NB*NB
```

In this header, there is no kernel declared as a task yet, so you have to search into the matmul_kernel.cu/cl file in order to see which kernel you need to declare, declare the kernel as a task, by placing its declaration and the pragmas over it.

Hint: In this case as we are multiplying a two-dimension matrix, so the best approach is to use a two-dimension ndrange.

In order to get this program working, we only need to specify the NDRange clause, which has five members:

- First one is the number of dimensions on the kernel (2 in this case).
- The second and third ones are the total number of kernel threads to be launched (as one kernel threads usually calculates a single index of data, this is usually the number of elements, of the vectors in this case) per dimension.
- The fourth and fifth ones are the group size (number of threads per block), in this kind of kernels which do not use shared memory between groups, any number from 16 to 32 (per dimension) should work correctly (depending on the underlying hardware).

Once the ndrange clause is correct and the input/outputs are correctly defined. We can proceed to compile the source code, using the command “make”. After it (if there are no errors), we can execute it using “mnsubmit run-1.sh”.

NBody (OpenCL and CUDA)

In this exercise we will work with the NBody kernel. This algorithm numerically approximates the evolution of a system of bodies in which each body continuously interacts with every other body

As in the previous exercise there is a kernel header file (kernel_header.cuh/clh) which must declares the kernel as a task, this header must be included in the C-file and can also be included in the kernel file (although not strictly needed). As the kernel is not declared as a task yet, you have to search into the cuda/opencl_kernel.cu/cl file in order to see which kernel you need to declare, declare the kernel as a task, by placing its declaration and the pragmas over it.

Once the kernel is correctly declared as a task, and is configured with the proper NDRange clause, we can proceed to compile the source code, using the command “make”. After it (if there are no errors), we can execute it using “mnsubmit run-1.sh”. In order to check results, you can use the command “diff nbody_out-ref.xyz nbody_out.xyz”.

* Extra: If someone is interested, you can try to do a NBody implementation which works with multiple GPUs. You must split the output in different parts so each GPU will calculate one of this parts. If you check the whole source code in nbody.c (not needed), you can see that the “Particle_array_calculate_forces_cuda” function in kernel.c is called 10 times, and in each call, the input and output array are swapped, so they act like their counter-part in the next call. So when we split the output, we must also split the input in as many pieces as the previous output.