
OmpSs Examples and Exercises

BSC Programming Models

May 08, 2017

1	Introduction: Examples and Exercises	3
1.1	Contributing to the examples and exercises document	3
1.2	System configuration	4
1.3	The make utility	4
1.4	Building the examples	5
1.5	Job Scheduler: Minotauro	6
1.6	Job Scheduler: Marenostrum	6
2	Writing OmpSs programs	7
2.1	Data Management	7
2.1.1	Reusing device data among same device kernel executions	7
2.1.2	Forcing data back using a taskwait	8
2.1.3	Forcing data back using a task	9
2.2	Application's kernels	11
2.2.1	BlackScholes	11
2.2.2	Perlin Noise	12
2.2.3	N-Body	13
3	Examples Using OmpSs	15
3.1	Cholesky kernel	15
3.2	Stream Benchmark	16
3.3	Array Sum Benchmark (Fortran version)	16
4	Beginners Exercises	19
4.1	Matrix Multiplication	19
4.2	Dot Product	19
4.3	Multisort application	20
5	GPU Device Exercises	23
5.1	Saxpy kernel	23
5.2	Krist kernel	24
5.3	Matrix Multiply	24
5.4	NBody kernel	25
5.5	Cholesky kernel	26
6	MPI+OmpSs Exercises	27
6.1	Matrix multiply	27
6.2	Heat diffusion (Jacobi solver)	28

The information included in this document is provided “as is”, with no warranties whatsoever, including any warranty of merchantability, fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification, or sample. The document is not guaranteed to be complete and/or error-free at this stage and it is subject to changes without further notice. Barcelona Supercomputing Center will not assume any responsibility for errors or omissions in this document. Please send comments, corrections and/or suggestions to pm-tools at bsc.es. This document is provided for informational purposes only.

Note: A PDF version of this document is available in <http://pm.bsc.es/ompss-docs/examples/OmpSsExamples.pdf>, and all the example source codes in <http://pm.bsc.es/ompss-docs/examples/ompss-ee.tar.gz>

INTRODUCTION: EXAMPLES AND EXERCISES

This documentation contains examples and exercises using the OmpSs programming model. The main objective of this paper is to provide guidance in learning OmpSs programming model and serve as teaching materials in courses and tutorials. To find more complete applications please visit our BAR (BSC Application Repository) in the URL:

<http://pm.bsc.es/projects/bar>

1.1 Contributing to the examples and exercises document

The OmpSs documentation is written using Sphinx

<http://www.sphinx-doc.org/>

1. Make sure you have sphinx-doc in your machine

Ubuntu/Debian:

```
$ sudo apt-get install sphinx-doc python-sphinx texlive-latex-extra texlive-fonts-recommended
```

(Note: texlive- packages are required to build PDF documentation).

2. Make changes to .rst files

Start from index.rst to see the structure. Look at the .. toctree::, it lists the included files used to generate the documentation (toctree stands for “tree of the table of contents”).

Syntax of .rst is reStructuredText. You may want to read a quick introduction at

<http://www.sphinx-doc.org/rest.html>

The official reStructuredText documentation (if you want to dig further in the details) is in:

<http://docutils.sourceforge.net/rst.html#user-documentation>

3. Generate the documentation

- 3.1. Generate the HTML

```
$ make html
```

Now open your browser to .build/html/index.html and behold your contribution.

- 3.2. Generate the PDF

```
$ make latexpdf
```

Now open your PDF viewer to the .build/html/<docfile>.pdf (the file depends on the directory you chose in the step 0 above)

4. Commit your changes using git

```
$ git commit -a $ git push
```

It may happen that the remote repository changed where you were editing your local one. In that case, first do

```
$ git pull --rebase
```

and then proceed as above.

```
$ git commit -a $ git push
```

1.2 System configuration

In this section we describe how to tune your configure script and also how to use it to configure your environment. If you have a pre-configured package you can skip this section and simply run the Linux command `source` using your configure script:

```
$source configure.sh
```

The configure script is used to set all environment variables you need to properly execute OmpSs applications. Among other things it contains the `PATH` where the system will look for to find Mercurium compiler utility, and the MKL installation directory (if available) to run specific OmpSs applications (e.g. Cholesky kernel).

To configure your system you have to run the Linux command `source` using your configure script:

```
$source configure.sh
```

1.3 The make utility

In software development, Make is a utility that automatically builds executable programs and libraries from source code by reading files called makefiles which specify how to derive the target program. Though integrated development environments and language-specific compiler features can also be used to manage a build process, Make remains widely used, especially in Unix.

Make searches the current directory for the makefile to use, e.g. GNU make searches files in order for a file named one of `GNUmakefile`, `makefile`, `Makefile` and then runs the specified (or default) target(s) from (only) that file.

A makefile consists of rules. Each rule begins with a textual dependency line which defines a target followed by a colon (`:`) and optionally an enumeration of components (files or other targets) on which the target depends. The dependency line is arranged so that the target (left hand of the colon) depends on components (right hand of the colon). It is common to refer to components as prerequisites of the target:

```
target [target...] : [component...]  
[<TAB>] command-1  
[<TAB>] command-2  
...  
[<TAB>] command-n  
[target
```

Below is a very simple makefile that by default compiles the program `helloworld` (first target: `all`) using the GCC C compiler (`CC`) and using “-g” compiler option (`CFLAGS`). The makefile also provides a “clean” target to remove the generated files if the user desires to start over (by running `make clean`):

```
CC = gcc  
CFLAGS = -g  
LDFLAGS =
```



```

all: helloworld

helloworld: helloworld.o
    # Commands start with TAB not spaces !!!
    $(CC) $(LDFLAGS) -o $@ $^

helloworld.o: helloworld.c
    $(CC) $(CFLAGS) -c -o $@ $<

clean:
    rm -f helloworld helloworld.o

```

1.4 Building the examples

All the examples and exercises comes with a makefile (Makefile) configured to compile 3 different versions for each program. Each of the binary file name created by running make ends with a suffix which determines the version:

- program-p: performance version
- program-i: instrumented version
- 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.

Apart of building the program’s binaries, the make utility will also build shell scripts to run the program. Each exercise have two running scripts, one to run a single program execution (“run-once.sh”), the other will run multiples configurations with respect the number of threads, data size, etc (“multirun.sh”). Before submitting any job, make sure all environment variables have the values you expect to. Here is an example of the “run-once.sh” script:

```

#!/bin/bash
export NX_THREADS=4

./cholesky-p 4096 512 1

```

In some cases the shell script will contain job scheduler variables declared in top of the script file. A job scheduler script must contain a series of directives to inform the batch system about the characteristics of the job. These directives appear as comments in the job script file and the syntax will depend on the job scheduler system used.

With the running scripts it also comes a “trace.sh” file, which can be used to include all the environment variables needed to get an instrumentation trace of the execution. The content of this file is:

```

#!/bin/bash
export EXTRAE_CONFIG_FILE=extrae.xml
export NX_INSTRUMENTATION=extrae
$*

```

Additionally, you will need to change your running script in order to invoke the your program through the “trace.sh” script. Although you can also edit your running script adding all the environment variables related with the instrumentation, it is preferable to use this extra script to easily change in between instrumented and non-instrumented executions. When you want to instrument you will need to include “trace.sh” before your program execution command line:

```

#!/bin/bash
export NX_THREADS=1

```

```
./trace.sh ./cholesky-i 4096 512 1
```

Finally, the make utility will generate (if not already present in the directory) other configuration files as it is the case of “extrae.xml” file (used to configure Extrae plugin in order to get a Paraver trace, see “trace.sh” file).

1.5 Job Scheduler: Minotauro

The current section has a short explanation on how to use the job scheduler systems installed in BSC’s Minotauro machine. Slurm is the utility used in this machine for batch processing support, so all jobs must be run through it. These are the basic directives to submit jobs:

- mnsuubmit job_script submits a “job script” to the queue system (see below for job script directives).
- mnq: shows all the submitted jobs.
- mncancel <job_id> remove the job from the queue system, cancelling the execution of the processes, if they were still running.

A job must contain a series of directives to inform the batch system about the characteristics of the job. These directives appear as comments in the job script, with the following syntax:

```
# @ directive = value.
```

The job would be submitted using: “mnsuubmit <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).

1.6 Job Scheduler: Marenostrum

LSF is the utility used at MareNostrum III for batch processing support, so all jobs must be run through it. This section provides information for getting started with job execution at the Cluster. These are the basic commands to submit, control and check your jobs:

- bsub < job_script: submits a “job script” passed through standard input (STDIN) to the queue system.
- bjobs: shows all the submitted jobs
- bkill <job_id>: remove the job from the queue system, canceling the execution of the processes, if they were still running.
- bsc_jobs: shows all the pending or running jobs from your group.

WRITING OMPSS PROGRAMS

Following examples are written in C/C++ or Fortran using OmpSs as a programming model. With each example we provide simple explanations on how they are annotated and, in some cases, how they can be compiled (if a full example is provided).

2.1 Data Management

2.1.1 Reusing device data among same device kernel executions

Although memory management is completely done by the runtime system, in some cases we can assume a predefined behaviour. This is the case of the following Fortran example using an OpenCL kernel. If we assume runtime is using a write-back cache policy we can also determine that second kernel call will not imply any data movement.

kernel_1.cl:

```
__kernel void vec_sum(int n, __global int* a, __global int* b, __global int* res)
{
    const int idx = get_global_id(0);

    if (idx < n) res[idx] = a[idx] + b[idx];
}
```

test_1.f90:

```
! NOTE: Assuming write-back cache policy

SUBROUTINE INITIALIZE(N, VEC1, VEC2, RESULTS)
    IMPLICIT NONE
    INTEGER :: N
    INTEGER :: VEC1(N), VEC2(N), RESULTS(N), I
    DO I=1,N
        VEC1(I) = I
        VEC2(I) = N+1-I
        RESULTS(I) = -1
    END DO
END SUBROUTINE INITIALIZE

PROGRAM P
    IMPLICIT NONE
    INTERFACE
        !$OMP TARGET DEVICE(OPENCL) NDRANGE(1, N, 128) FILE(kernel_1.cl) COPY_DEPS
        !$OMP TASK IN(A, B) OUT(RES)
        SUBROUTINE VEC_SUM(N, A, B, RES)
    END INTERFACE
```

```

    IMPLICIT NONE
    INTEGER, VALUE :: N
    INTEGER :: A(N), B(N), RES(N)
  END SUBROUTINE VEC_SUM
END INTERFACE
INTEGER, PARAMETER :: N = 20
INTEGER :: VEC1(N), VEC2(N), RESULTS(N), I

CALL INITIALIZE(N, VEC1, VEC2, RESULTS)

CALL VEC_SUM(N, VEC1, VEC2, RESULTS)
! The vectors VEC1 and VEC2 are sent to the GPU. The input transfers at this
! point are: 2 x ( 20 x sizeof(INTEGER)) = 2 x (20 x 4) = 160 B.

CALL VEC_SUM(N, VEC1, RESULTS, RESULTS)
! All the input data is already in the GPU. We don't need to send
! anything.

!$OMP TASKWAIT
! At this point we copy out from the GPU the computed values of RESULTS
! and remove all the data from the GPU

! print the final vector's values
PRINT *, "RESULTS: ", RESULTS
END PROGRAM P

! Expected IN/OUT transfers:
! IN = 160B
! OUT = 80B

```

Compile with:

```
oclmfc -o test_1 test_1.f90 kernel_1.cl --ompss
```

2.1.2 Forcing data back using a taskwait

In this example, we need to copy back the data in between the two kernel calls. We force this copy back using a taskwait. Note that we are assuming write-back cache policy.

kernel_2.cl:

```

__kernel void vec_sum(int n, __global int* a, __global int* b, __global int* res)
{
    const int idx = get_global_id(0);

    if (idx < n) res[idx] = a[idx] + b[idx];
}

```

test_2.f90:

```

! NOTE: Assuming write-back cache policy
SUBROUTINE INITIALIZE(N, VEC1, VEC2, RESULTS)
  IMPLICIT NONE
  INTEGER :: N
  INTEGER :: VEC1(N), VEC2(N), RESULTS(N), I
  DO I=1,N
    VEC1(I) = I

```

```

VEC2(I) = N+1-I
RESULTS(I) = -1
END DO
END SUBROUTINE INITIALIZE

PROGRAM P
  IMPLICIT NONE
  INTERFACE
    !$OMP TARGET DEVICE(OPENCL) NDRANGE(1, N, 128) FILE(kernel_2.cl) COPY_DEPS
    !$OMP TASK IN(A, B) OUT(RES)
    SUBROUTINE VEC_SUM(N, A, B, RES)
      IMPLICIT NONE
      INTEGER, VALUE :: N
      INTEGER :: A(N), B(N), RES(N)
    END SUBROUTINE VEC_SUM
  END INTERFACE
  INTEGER, PARAMETER :: N = 20
  INTEGER :: VEC1(N), VEC2(N), RESULTS(N), I

  CALL INITIALIZE(N, VEC1, VEC2, RESULTS)

  CALL VEC_SUM(N, VEC1, VEC2, RESULTS)
  ! The vectors VEC1 and VEC2 are sent to the GPU. The input transfers at this
  ! point are: 2 x ( 20 x sizeof(INTEGER)) = 2 x (20 x 4) = 160 B.

  !$OMP TASKWAIT
  ! At this point we copy out from the GPU the computed values of RESULT
  ! and remove all the data from the GPU

  PRINT *, "PARTIAL RESULTS: ", RESULTS

  CALL VEC_SUM(N, VEC1, RESULTS, RESULTS)
  ! The vectors VEC1 and RESULT are sent to the GPU. The input transfers at this
  ! point are: 2 x ( 20 x sizeof(INTEGER)) = 2 x (20 x 4) = 160 B.

  !$OMP TASKWAIT
  ! At this point we copy out from the GPU the computed values of RESULT
  ! and remove all the data from the GPU

  ! print the final vector's values
  PRINT *, "RESULTS: ", RESULTS
END PROGRAM P

! Expected IN/OUT transfers:
! IN = 320B
! OUT = 160B

```

Compile with:

```
oclmfc -o test_2 test_2.f90 kernel_2.cl --ompss
```

2.1.3 Forcing data back using a task

This example is similar to the example 1.2 but instead of using a `taskwait` to force the copy back, we use a task with copies. Note that we are assuming `write-back` cache policy.

kernel_3.cl:

```

__kernel void vec_sum(int n, __global int* a, __global int* b, __global int* res)
{
    const int idx = get_global_id(0);

    if (idx < n) res[idx] = a[idx] + b[idx];
}

```

test_3.f90:

```

! NOTE: Assuming write-back cache policy

SUBROUTINE INITIALIZE(N, VEC1, VEC2, RESULTS)
    IMPLICIT NONE
    INTEGER :: N
    INTEGER :: VEC1(N), VEC2(N), RESULTS(N), I
    DO I=1,N
        VEC1(I) = I
        VEC2(I) = N+1-I
        RESULTS(I) = -1
    END DO
END SUBROUTINE INITIALIZE

PROGRAM P
    IMPLICIT NONE
    INTERFACE
        !$OMP TARGET DEVICE(OPENCL) NDRANGE(1, N, 128) FILE(kernel_3.cl) COPY_DEPS
        !$OMP TASK IN(A, B) OUT(RES)
        SUBROUTINE VEC_SUM(N, A, B, RES)
            IMPLICIT NONE
            INTEGER, VALUE :: N
            INTEGER :: A(N), B(N), RES(N)
        END SUBROUTINE VEC_SUM

        !$OMP TARGET DEVICE(SMP) COPY_DEPS
        !$OMP TASK IN(BUFF)
        SUBROUTINE PRINT_BUFF(N, BUFF)
            IMPLICIT NONE
            INTEGER, VALUE :: N
            INTEGER :: BUFF(N)
        END SUBROUTINE PRINT_BUFF
    END INTERFACE

    INTEGER, PARAMETER :: N = 20
    INTEGER :: VEC1(N), VEC2(N), RESULTS(N), I

    CALL INITIALIZE(N, VEC1, VEC2, RESULTS)

    CALL VEC_SUM(N, VEC1, VEC2, RESULTS)
    ! The vectors VEC1 and VEC2 are sent to the GPU. The input transfers at this
    ! point are: 2 x ( 20 x sizeof(INTEGER) ) = 2 x (20 x 4) = 160 B.

    CALL PRINT_BUFF(N, RESULTS)
    ! The vector RESULTS is copied from the GPU to the CPU. The copy of this vector in
    ! the memory of the GPU is not removed because the task 'PRINT_BUFF' does not modify it.
    ! Output transfers: 80B.
    ! VEC1 and VEC2 are still in the GPU.

    CALL VEC_SUM(N, VEC1, RESULTS, RESULTS)

```

```

! The vectors VEC1 and RESULTS are already in the GPU. Do not copy anything.

CALL PRINT_BUFF(N, RESULTS)
! The vector RESULTS is copied from the GPU to the CPU. The copy of this vector in
! the memory of the GPU is not removed because the task 'PRINT_BUFF' does not it.
! Output transfers: 80B.
! VEC1 and VEC2 are still in the GPU.

!$OMP TASKWAIT
! At this point we remove all the data from the GPU. The right values of the vector RESULTS are
! already in the memory of the CPU, then we don't need to copy anything from the GPU.

END PROGRAM P

SUBROUTINE PRINT_BUFF (N, BUFF)
  IMPLICIT NONE
  INTEGER, VALUE :: N
  INTEGER :: BUFF(N)

  PRINT *, "BUFF: ", BUFF
END SUBROUTINE PRINT_BUFF

! Expected IN/OUT transfers:
! IN = 160B
! OUT = 160B

```

Compile with:

```
oclmpc -o test_3 test_3.f90 kernel_3.cl --ompss
```

2.2 Application's kernels

2.2.1 BlackScholes

This benchmark computes the pricing of European-style options. Its kernel has 6 input arrays, and a single output. Offloading is done by means of the following code:

```

for (i=0; i<array_size; i+= chunk_size ) {
  int elements;
  unsigned int * cpf;
  elements = min(i+chunk_size, array_size) - i;
  cpf = cpf[i];
#pragma omp target device(cuda) copy_in( \
      cpf [i;elements], \
      S0 [i;elements], \
      K [i;elements], \
      r [i;elements], \
      sigma [i;elements], \
      T [i;elements]) \
  copy_out (answer[i;elements])
#pragma omp task firstprivate(local_work_group_size, i)
  {
    dim3 dimBlock(local_work_group_size, 1, 1);
    dim3 dimGrid(elements / local_work_group_size, 1, 1);
    cuda_bsoop <<<dimGrid, dimBlock>>> (&cpf[i], &S0[i], &K[i],

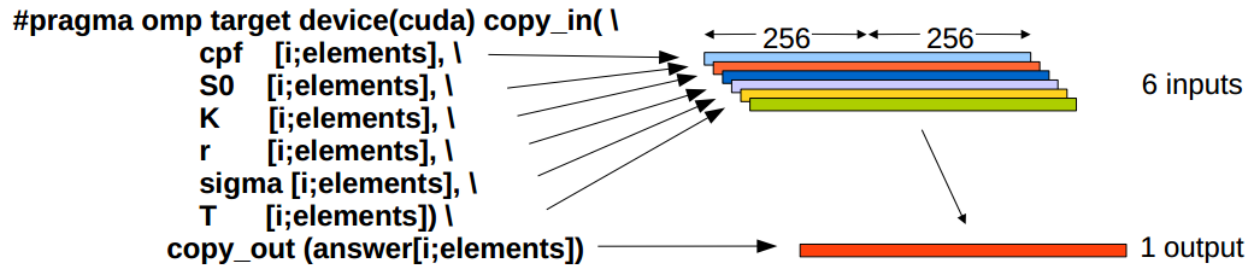
```

```

        &r[i], &sigma[i], &T[i], &answer[i]);
    }
}
#pragma omp taskwait

```

Following image shows graphically the annotations used to offload tasks to the GPUs available. Data arrays annotated with the `copy_in` clause are automatically transferred by the Nanos++ runtime system onto the GPU global memory. After the CUDA kernel has been executed, the `copy_out` clause indicates to the runtime system that the results written by the GPU onto the output array should be synchronized onto the host memory. This is done at the latest when the host program encounters the `taskwait` directive.



2.2.2 Perlin Noise

This benchmark generates an image consisting of noise, useful to be applied to gaming applications, in order to provide realistic effects. The application has a single output array, with the generated image. Annotations are shown here:

```

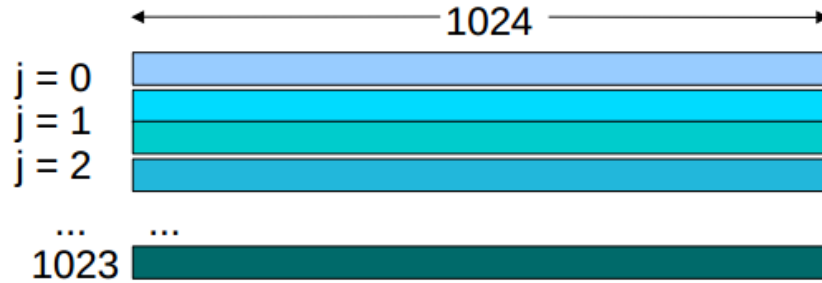
for (j = 0; j < img_height; j+=BS) {
    // Each task writes BS rows of the image
    #pragma omp target device(cuda) copy_deps
    #pragma omp task output (output[j*rowstride:(j+BS)*rowstride-1])
    {
        dim3 dimBlock;
        dim3 dimGrid;
        dimBlock.x = (img_width < BSx) ? img_width : BSx;
        dimBlock.y = (BS < BSy) ? BS : BSy;
        dimBlock.z = 1;
        dimGrid.x = img_width/dimBlock.x;
        dimGrid.y = BS/dimBlock.y;
        dimGrid.z = 1;

        cuda_perlin <<<dimGrid, dimBlock>>> (&output[j*rowstride], time, j, rowstride);
    }
}
#pragma omp taskwait noflush

```

In this example, the `noflush` clause eliminates the need for the data synchronization implied by the `taskwait` directive. This is useful when the programmer knows that the next task that will be accessing this result will also be executed in the GPUs, and the host program does not need to access it. The runtime system ensures in this case that the data is consistent across GPUs.

Following image shows the graphical representation of the data, and the way annotations split it across tasks.



```
for (j = 0; j < img_height; j+=BS) {
```

```
    #pragma omp target device(cuda) copy_deps
    #pragma omp task output (output[j*rowstride:(j*BS)*rowstride-1])
```

2.2.3 N-Body

This benchmark implements the gravitational forces among a set of particles. It works with an input array (`this_particle_array`), and an output array (`output_array`). Mass, velocities, and positions of the particles are kept updated alternatively in each array by means of a pointer exchange. The annotated code is shown here:

```
void Particle_array_calculate_forces_cuda ( int number_of_particles,
    Particle this_particle_array[number_of_particles],
    Particle output_array[number_of_particles],
    float time_interval )
{
    const int bs = number_of_particles/8;
    size_t num_threads, num_blocks;
    num_threads = ((number_of_particles < MAX_NUM_THREADS) ?
        Number_of_particles : MAX_NUM_THREADS );
    num_blocks = ( number_of_particles + MAX_NUM_THREADS ) / MAX_NUM_THREADS;
    #pragma omp target device(cuda) copy_deps
    #pragma omp task output( output_array) input(this_particle_array )
        calculate_forces_kernel_naive <<< num_blocks, MAX_NUM_THREADS >>>
            (time_interval, this_particle_array, number_of_particles,
                &output_array[first_local], first_local, last_local);
    #pragma omp taskwait
}
```


EXAMPLES USING OMPSS

In this section we include several OmpSs applications that are already parallelized (i.e. annotated with OmpSs directives). Users have not to change the code, but they are encouraged to experiment with them. You can also use that source directory to experiment with the different compiler and runtime options, as well as the different instrumentation plugins provided with your OmpSs installation.

3.1 Cholesky kernel

This example shows 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.

Note: You can download this code visiting the url http://pm.bsc.es/OmpSs_Examples_and_Exercises's (code) link. The Cholesky kernel is included inside the *01-examples*'s directory.

The kernel uses four different linear algorithms: potrf, trsm, gemm and syrkh. Following code shows the basic pattern for a Cholesky factorisation:

```
for (int k = 0; k < nt; k++) {
    // Diagonal Block factorization
    omp_potrf (Ah[k][k], ts, ts);

    // Triangular systems
    for (int i = k + 1; i < nt; i++)
        omp_trsm (Ah[k][k], Ah[k][i], ts, ts);

    // Update trailing matrix
    for (int i = k + 1; i < nt; i++) {
        for (int j = k + 1; j < i; j++)
            omp_gemm (Ah[k][i], Ah[k][j], Ah[j][i], ts, ts);

        omp_syrk (Ah[k][i], Ah[i][i], ts, ts);
    }
}
```

In this case we parallelize the code by annotating the kernel functions. So each call in the previous loop becomes the instantiation of a task. The following code shows how we have parallelized Cholesky:

```
#pragma omp task inout([ts][ts]A)
void omp_potrf(double * const A, int ts, int ld)
{
    ...
}
```

```
}  
  
#pragma omp task in([ts][ts]A) inout([ts][ts]B)  
void omp_trsm(double *A, double *B, int ts, int ld)  
{  
    ...  
}  
  
#pragma omp task in([ts][ts]A) inout([ts][ts]B)  
void omp_syrk(double *A, double *B, int ts, int ld)  
{  
    ...  
}  
  
#pragma omp task in([ts][ts]A, [ts][ts]B) inout([ts][ts]C)  
void omp_gemm(double *A, double *B, double *C, int ts, int ld)  
{  
    ...  
}
```

Note that for each of the dependences we also specify which is the matrix (block) size. Although this is not needed, due there is no overlapping among the different blocks, it will allow the runtime to compute dependences using the region mechanism.

3.2 Stream Benchmark

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 and another without barriers. 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.

Note: You can dowload this code visiting the url <http://pm.bsc.es> *OmpSs Examples and Exercises*'s (code) link. The Stream benchmark is included inside the *01-examples*'s directory.

3.3 Array Sum Benchmark (Fortran version)

This benchmark computes the sum of two arrays and stores the result in an other array.

Note: You can dowload this code visiting the url <http://pm.bsc.es> *OmpSs Examples and Exercises*'s (code) link. The Array Sum benchmark is included inside the *01-examples*'s directory.

In this case we annotate the algorithm using the Fortran syntax. The benchmark compute a set of array sums. The first inner loop initializes one array, that will be computed in the second inner loop. Dependences warrant proper execution and synchronization between initialization and compute results:

```
DO K=1, 1000  
    IF (MOD(K, 100) == 0) WRITE(0, *) 'K=', K  
    ! INITIALIZE THE ARRAYS  
    DO I=1, N, BS
```

```
!$OMP TASK OUT(VEC1(I:I+BS-1), VEC2(I:I+BS-1), RESULTS(I:I+BS-1)) &  
!$OMP PRIVATE(J) FIRSTPRIVATE(I, BS) LABEL(INIT_TASK)  
DO J = I, I+BS-1  
  VEC1(J) = J  
  VEC2(J) = N + 1 - J  
  RESULTS(J) = -1  
END DO  
!$OMP END TASK  
ENDDO  
! RESULTS = VEC1 + VEC2  
DO I=1, N, BS  
!$OMP TASK OUT(VEC1(I:I+BS-1), VEC2(I:I+BS-1)) IN(RESULTS(I:I+BS-1)) &  
!$OMP PRIVATE(J) FIRSTPRIVATE(I, BS) LABEL(ARRAY_SUM_TASK)  
DO J = I, I+BS-1  
  RESULTS(J) = VEC1(J) + VEC2(J)  
END DO  
!$OMP END TASK  
ENDDO  
ENDDO ! K  
!$OMP TASKWAIT
```


BEGINNERS EXERCISES

4.1 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 that you may consider before do the 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?).

4.2 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. A common implementation of this operation is shown below:

```
double dot_product(int N, int v1[N], int v2[N]) {
  double result = 0.0;
  for (long i=0; i<N; i++)
    result += v1[i] * v2[i];

  return result;
}
```

The example above is interesting from a programming model point of view because it accumulates the result of each iteration on a single variable called `result`. As we have already seen in this course, this kind of operation is called reduction, and it is a very common pattern in scientific and mathematical applications.

There are several ways to parallelize operations that compute a reduction:

- 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 the reduction 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.

Once we have introduced the dot product operation and the different ways of parallelizing a reduction, let's start this exercise. If you open the *dot-product.c* file, you will see that the `dot_product` function is a bit more complicated than the previous version.

```
double result = 0.0;
long j = 0;
for (long i=0; i<N; i+=CHUNK_SIZE) {
    actual_size = (N - i >= CHUNK_SIZE) ? CHUNK_SIZE : N - CHUNK_SIZE;
    C[j] = 0;

    #pragma omp task label( dot_prod ) firstprivate( j, i, actual_size )
    {
        for (long ii=0; ii<actual_size; ii++)
            C[j] += A[i+ii] * B[i+ii];
    }

    #pragma omp task label( increment ) firstprivate( j )
    result += C[j];

    j++;
}
```

Basically we have prepared our code to parallelize it, creating a private storage for each chunk and splitting the main loop into two different nested loops to adjust the granularity of our tasks (see `CHUNK_SIZE` variable). Apart from that, we have also annotated the tasks for you, but this parallel version is not ready, yet.

Goals of this exercise

- Find all the `#pragma omp` lines. As you can see, there are tasks, but we forgot to specify their dependencies.
- Tasks are executed asynchronously. Thus, at some point we have to wait for them. Where should we do that?
- 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.

4.3 Multisort application

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]);

    // Recursive merge: quarters to halves
    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);

    // Recursive merge: halves to whole
    merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);

} else {
    // Base case: using simpler algorithm
    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):

```
$NX_THREADS=4 gdb --args ./multisort-d 4096 64 128
```


GPU DEVICE EXERCISES

Almost all the programs in this section is available both in OpenCL and CUDA. From the point of view of an OmpSs programmer, the only difference between them is the language in which the kernel is written.

As OmpSs abstracts the user from doing the work in the host part of the code. Both OpenCL and CUDA have the same syntax. You can do any of the two versions, as they are basically the same, when you got one of them working, same steps can be done in the other version.

5.1 Saxpy kernel

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 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).

Kernel header file (kernel.h) have:

```
#pragma omp target device(cuda) copy_deps nrange( /*??*/ )
#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 nrange 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 nrange clause is correct. We can proceed to compile the source code, using the command 'make'. After it (if there are no compilation/link errors), we can execute it using one of the running scripts.

5.2 Krist kernel

Krist kernel 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 file (krist.h) 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).

Krist header file (krist.h) have:

```
#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: ND-range 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 one of the provided running scripts. Check if all environment variables are set to the proper values.

5.3 Matrix Multiply

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/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 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).

Matrix multiply header file (kernel.h) have:

```
//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, you have to search into the 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.

Note: 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 one of the running scripts.

5.4 NBody kernel

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. In this case we want to port a traditional SMP source code to another one which can exploit the benefits of CUDA/OpenCL. Someone already ported the kernel, so it does the same calculations than the previous SMP function.

Sources are not complete, we only have the C code which is calling a SMP function and a CUDA/OCL kernel, they do not interact with each other, `nbody.c` file have:

```
// Call the kernel
calculate_force_func(bs,time_interval,number_of_particles,this_particle_array, &output_array[i], i, ...
```

In this case there is nothing but a kernel ported by someone and a code calling a `smp` function. We’ll need to declare the kernel as an OmpSs task as we have done in previous examples.

Note: Use an intermediate header file and include it, it will work if we declare it on the `.c` file.

Once the kernel is correctly declared as a task, we can call it instead of the ‘old’ `smp` function. We can proceed to compile the source code, using the command ‘make’. After it (if there are no errors), we can execute it using one of the running scripts. In order to check results, you can use the command ‘diff `nbody_out-ref.xyz nbody_out.xyz`’.

Note: If someone is interested, you can try to do a NBody implementation which works with multiple GPUs can be done if you have finished early, you must split the output in different parts so each GPU will calculate one of this parts.

If we 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.

5.5 Cholesky kernel

This kernel is just like the SMP version found in the examples, but implemented in CUDA. It uses CUBLAS kernels for the `syrk`, `trsm` and `gemm` kernels, and a CUDA implementation for the `potrf` kernel (declared in a different file).

Your assignment is to annotate all CUDA tasks in the source code under the section “TASKS FOR CHOLESKY”.

MPI+OMPSS EXERCISES

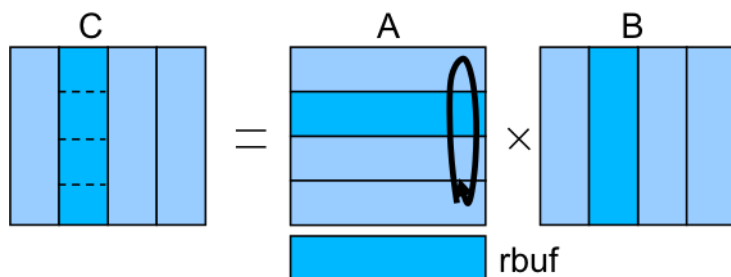
6.1 Matrix multiply

This code performs a matrix - matrix multiplication by means of a hybrid MPI/OmpSs implementation.

Note: You can download this code visiting the url http://pm.bsc.es/OmpSs_Examples_and_Exercises/s (code) link. This version of matrix multiply kernel is included inside the *04-mpi+ompss*'s directory.

Groups of rows of the matrix A are distributed to the different MPI processes. Similarly for the matrix B, groups of columns are distributed to the different MPI process. In each iteration, each process performs the multiplication of the A rows by the B columns in order to compute a block of C. After the computation, each process exchanges with its neighbours the set of rows of A (sending the current ones to the process i+1 and receiving the ones for the next iteration from the process i-1).

An additional buffer rbuf is used to exchange the rows of matrix A between the different iterations.



In this implementation of the code, two tasks are defined: one for the computation of the block of C and another for the communication. See the sample code snippet:

```

for ( it = 0; it < nodes; it++ ) {

    #pragma omp task // add in, out and inout
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, n, n, m, 1.0, (double *)a, m, (double *)B, n

    if (it < nodes-1) {
        #pragma omp task // add in, out and inout
        MPI_Sendrecv( a, size, MPI_DOUBLE, down, tag, rbuf, size, MPI_DOUBLE, up, tag, MPI_COMM_WORLD, &

    }

    i = (i+n)%m;           //next C block circular
    ptmp=a; a=rbuf; rbuf=ptmp; //swap pointers
}

```

The exercise is provided without the in, out and inout dependence clauses.

- Complete the pragmas in order to define the correct dependences
- Compile and run the example and check that the results are correct (the output of the computation already checks the correctness)
- Generate the tracefile and task graph of the execution

6.2 Heat diffusion (Jacobi solver)

This codes performs a ... hybrid MPI/OmpSs implementation.

Note: You can dowload this code visiting the url <http://pm.bsc.es> *OmpSs Examples and Exercises*'s (code) link. This version of matrix multiply kernel is included inside the *04-mpi+omps*'s directory.

Note: You need to specify the number of MPI tasks per node. In Marenostrum you can do this by adding `<<#BSUB -R "span[ptile=1]">>` to your job script.
