

Programming with OmpSs-2

Table of contents:

- [Quick Overview](#)
- [Quick Setup on DEEP System for a Shared-Memory Parallel Application](#)
- [Using the Repositories](#)
- Examples:
 - [A Step-By-Step Detailed Guide to Execute the Multisaxpy Benchmark](#)
 - [Dot-product Benchmark](#)
 - [Mergesort Benchmark](#)
 - [Nqueens Benchmark](#)
 - [Matmul Benchmark](#)
 - [Cholesky Nenchmark \(OmpSs-2+MKL\)](#)
 - [Nbody Nenchmark \(MPI+OmpSs-2+TAMPI\)](#)
 - [Heat Benchmark \(MPI+OmpSs-2+TAMPI\)](#)

Quick Overview

OmpSs-2 is a programming model composed of a set of directives and library routines that can be used in conjunction with a high-level programming language (such as C, C++ or Fortran) in order to develop concurrent applications. Its name originally comes from two other programming models: **OpenMP** and **StarSs**. The design principles of these two programming models constitute the fundamental ideas used to conceive the OmpSs philosophy.

OmpSs-2 **thread-pool** execution model differs from the **fork-join** parallelism implemented in OpenMP.

A **task** is the minimum execution entity that can be managed independently by the runtime scheduler. **Task dependences** let the user annotate the data flow of the program and are used to determine, at runtime, if the parallel execution of two tasks may cause data races.

The reference implementation of OmpSs-2 is based on the **Mercurium** source-to-source compiler and the **Nanos6** runtime library:

- Mercurium source-to-source compiler provides the necessary support for transforming the high-level directives into a parallelized version of the application.
- Nanos6 runtime library provides services to manage all the parallelism in the user-application, including task creation, synchronization and data movement, as well as support for resource heterogeneity.

Additional information about the OmpSs-2 programming model can be found at:

- OmpSs-2 official website. [?https://pm.bsc.es/ompss-2](https://pm.bsc.es/ompss-2)
- OmpSs-2 specification. [?https://pm.bsc.es/ftp/ompss-2/doc/spec](https://pm.bsc.es/ftp/ompss-2/doc/spec)
- OmpSs-2 user guide. [?https://pm.bsc.es/ftp/ompss-2/doc/user-guide](https://pm.bsc.es/ftp/ompss-2/doc/user-guide)
- OmpSs-2 examples repository. [?https://pm.bsc.es/gitlab/ompss-2/examples](https://pm.bsc.es/gitlab/ompss-2/examples)
- OmpSs-2 manual with examples and exercises. [?https://pm.bsc.es/ftp/ompss-2/doc/examples/index.html](https://pm.bsc.es/ftp/ompss-2/doc/examples/index.html)
- Mercurium official website. [?Link 1](#), [?Link 2](#)
- Nanos official website. [?Link 1](#), [?Link 2](#)

Quick Setup on DEEP System for a Shared-Memory Parallel Application

We highly recommend to interactively log in a **cluster module (CM) node** to begin using OmpSs-2. To request an entire CM node for an interactive session, please execute the following command to use all the 48 available threads:

```
srunk -p dp-cn -N 1 -n 1 -c 48 --pty /bin/bash -i
```

Note that the command above is consistent with the actual hardware configuration of the cluster module with **hyper-threading enabled**.

OmpSs-2 has already been installed on DEEP and can be used by simply executing the following commands:

```
modulepath="/usr/local/software/skylake/Stages/2018b/modules/all/Core:$modulepath"
```

```
modulepath="/usr/local/software/skylake/Stages/2018b/modules/all/Compiler/mpi/intel/2019.0.117-GCC-7.3.0:$modulepath"
```

```
modulepath="/usr/local/software/skylake/Stages/2018b/modules/all/MPI/intel/2019.0.117-GCC-7.3.0/psmpi/5.2.1-1-mt:$modulepath"
```

```
export MODULEPATH="$modulepath:$MODULEPATH"
```

```
module load OmpSs-2
```

Remember that OmpSs-2 uses a **thread-pool** execution model which means that it **permanently uses all the threads** present on the system. Users are strongly encouraged to always check the **system affinity** by running the **NUMA command** `srtn numactl --show`:

```
$ srtn numactl --show
policy: default
preferred node: current
physcpubind: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
cpubind: 0 1
nodebind: 0 1
membind: 0 1
```

as well as the **Nanos6 command** `srtn nanos6-info --runtime-details | grep List`:

```
$ srtn nanos6-info --runtime-details | grep List
Initial CPU List 0-47
NUMA Node 0 CPU List 0-35
NUMA Node 1 CPU List 12-47
```

System affinity can be used to specify, for example, the ratio of MPI and OmpSs-2 processes for a hybrid application and can be modified by user request in different ways:

- Via the command `srtn` or `salloc`. However, if the affinity given by SLURM does not correspond to the resources requested, it should be reported to the system administrators.
- Via the command `numactl`.
- Via the command `taskset`.

Using the Repositories

All the examples shown here are publicly available at [?https://pm.bsc.es/gitlab/ompss-2/examples](https://pm.bsc.es/gitlab/ompss-2/examples). Users must clone/download each example's repository and then transfer it to a DEEP working directory.

System Configuration

Please refer to section [Quick Setup on DEEP System](#) to get a functional version of OmpSs-2 on DEEP. It is also recommended to run OmpSs-2 via an interactive session on a cluster module (CM) node.

Building and Running the Examples

All the examples come with a Makefile already configured to build (e.g. `make`) and run (e.g. `make run`) them. You can clean the directory with the command `make clean`.

Controlling the Available Threads

In order to limit or constraint the available threads for an application, the Unix **taskset** tool can be used to launch applications with a given thread affinity. In order to use `taskset`, simply precede the application's binary with `taskset` followed by a list of CPU IDs specifying the desired affinity:

```
taskset -c 0,2-4 ./application
```

The example above will run **application** with 4 cores: 0, 2, 3, 4.

Creating Dependency Graphs

Nanos6 allows for a graphical representation of data dependencies to be extracted. In order to generate said graph, run the application with the **NANOS6** environment variable set to **graph**:

```
NANOS6=graph ./application
```

By default graph nodes will include the full path of the source code. To remove it, set the following environment variable:

```
NANOS6_GRAPH_SHORTEN_FILENAMES=1
```

The result will be a PDF file with several pages, each representing the graph at a certain point in time. For best results, we suggest to display the PDF with **single page** view, showing a full page and to advance page by page.

Obtaining Statistics

Another equally interesting feature of Nanos6 is obtaining statistics. To do so, simply run the application as:

```
NANOS6=stats ./application or also NANOS6=stats-papi ./application
```

The first collects timing statistics while the second also records hardware counters (compilation with PAPI is needed for the second). By default, the statistics are emitted standard error when the program ends.

Tracing with Extrae

A **trace.sh** file can be used to include all the environment variables needed to get an instrumentation trace of the execution. The content of this file is as follows:

```
#!/bin/bash
export EXTRAE_CONFIG_FILE=extrae.xml
export NANOS6="extrae"
$*
```

Additionally, you will need to change your running script in order to invoke the program through this trace.sh script so that it looks like:

```
./trace.sh ./application
```

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 between instrumented and non-instrumented executions. When in need to instrument your execution, simply include trace.sh before the program invocation. Note that the **extrae.xml** file, which is used to configure the Extrae library to get a Paraver trace, is also needed.

A Step-By-Step Detailed Guide to Execute the Multisaxpy Benchmark

Users must clone/download this example's repository from <https://pm.bsc.es/gitlab/ompss-2/examples/multisaxpy> and transfer it to a DEEP working directory.

Description

This benchmark runs several SAXPY operations. SAXPY is a combination of scalar multiplication and vector addition (a common operation in computations with vector processors) and constitutes a level 1 operation in the Basic Linear Algebra Subprograms (BLAS) package.

There are **7 implementations** of this benchmark.

Execution Instructions

```
./multisaxpy SIZE BLOCK_SIZE ITERATIONS
```

where:

- SIZE is the number of elements of the vectors used on the SAXPY operation.
- The SAXPY operation will be applied to the vector in blocks that contains BLOCK_SIZE elements.
- ITERATIONS is the number of times the SAXPY operation is executed.

Downloading, Building and Executing this Benchmark

Clone the repository to your local machine:

```
git clone https://pm.bsc.es/gitlab/ompss-2/examples/multisaxpy
```

and upload it to the `/work/cdeep/USERNAME/` directory (which might not exist yet) of the DEEP cluster:

```
scp -r multisaxpy/ USERNAME@deep.fz-juelich.de:~/work/cdeep/USERNAME/
```

Now connect to the DEEP login node:

```
ssh -X USERNAME@deep.fz-juelich.de
```

and from there open the *multisaxpy* folder:

```
cd /work/cdeep/USERNAME/multisaxpy
```

and request an interactive cluster module (CM) node in order to use all the available 48 threads to run a pure OmpSs-2 application:

```
srun -p dp-cn -N 1 -n 1 -c 48 --pty /bin/bash -i
```

Load the OmpSs-2 module via the following commands:

```
modulepath="/usr/local/software/skylake/Stages/2018b/modules/all/Core:$modulepath"
```

```
modulepath="/usr/local/software/skylake/Stages/2018b/modules/all/Compiler/mpi/intel/2019.0.117-GCC-7.3.0:$modulepath"
```

```
modulepath="/usr/local/software/skylake/Stages/2018b/modules/all/MPI/intel/2019.0.117-GCC-7.3.0/psmpi/5.2.1-1-mt:$modulepath"
```

```
export MODULEPATH="$modulepath:$MODULEPATH"
```

```
module load OmpSs-2
```

and check the affinity via the command `srun numactl --show` which should report the following:

```
$ srun numactl --show
policy: default
preferred node: current
physcpubind: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
cpubind: 0 1
nodebind: 0 1
membind: 0 1
```

Now you should be able to clean, build and execute this benchmark via the command `make`:

```
$ make clean
rm -f 01.multisaxpy_seq 02.multisaxpy_task_loop 03.multisaxpy_task 04.multisaxpy_task+dep 05.multisaxpy_task+weakdep 06.multisaxpy_task+weakdep

$ make
mcxx --ompss-2 01.multisaxpy_seq.cpp main.cpp -o 01.multisaxpy_seq -lrt
mcxx --ompss-2 02.multisaxpy_task_loop.cpp main.cpp -o 02.multisaxpy_task_loop -lrt
mcxx --ompss-2 03.multisaxpy_task.cpp main.cpp -o 03.multisaxpy_task -lrt
03.multisaxpy_task.cpp:3:13: info: adding task function 'axy_task' for device 'smp'
03.multisaxpy_task.cpp:12:3: info: call to task function '::axy_task'
03.multisaxpy_task.cpp:3:13: info: task function declared here
mcxx --ompss-2 04.multisaxpy_task+dep.cpp main.cpp -o 04.multisaxpy_task+dep -lrt
```

```

04.multisaxpy_task+dep.cpp:3:13: info: adding task function 'axpy_task' for device 'smp'
04.multisaxpy_task+dep.cpp:12:3: info: call to task function '::axpy_task'
04.multisaxpy_task+dep.cpp:3:13: info: task function declared here
mcxx --ompss-2 05.multisaxpy_task+weakdep.cpp main.cpp -o 05.multisaxpy_task+weakdep -lrt
05.multisaxpy_task+weakdep.cpp:3:13: info: adding task function 'axpy_task' for device 'smp'
05.multisaxpy_task+weakdep.cpp:12:3: info: call to task function '::axpy_task'
05.multisaxpy_task+weakdep.cpp:3:13: info: task function declared here
mcxx --ompss-2 06.multisaxpy_task_loop+weakdep.cpp main.cpp -o 06.multisaxpy_task_loop+weakdep -lrt
mcxx --ompss-2 07.multisaxpy_task+reduction.cpp main.cpp -o 07.multisaxpy_task+reduction -lrt
07.multisaxpy_task+reduction.cpp:14:13: info: reduction of variable 'yy' of type 'double [elements]' solved to 'operator +
<openmp-builtin-reductions>:1:1: info: reduction declared here
07.multisaxpy_task+reduction.cpp:21:13: info: reduction of variable 'y' of type 'double [N]' solved to 'operator +
<openmp-builtin-reductions>:1:1: info: reduction declared here

$ make run
./01.multisaxpy_seq 16777216 8192 100
size: 16777216, bs: 8192, iterations: 100, time: 3.30132, performance: 0.508197
NANOS6_SCHEDULER=fifo ./02.multisaxpy_task_loop 16777216 8192 100
size: 16777216, bs: 8192, iterations: 100, time: 0.411888, performance: 4.07325
./03.multisaxpy_task 16777216 8192 100
size: 16777216, bs: 8192, iterations: 100, time: 0.648536, performance: 2.58694
./04.multisaxpy_task+dep 16777216 8192 100
size: 16777216, bs: 8192, iterations: 100, time: 1.04207, performance: 1.60998
./05.multisaxpy_task+weakdep 16777216 8192 100
size: 16777216, bs: 8192, iterations: 100, time: 1.09049, performance: 1.5385
NANOS6_SCHEDULER=fifo ./06.multisaxpy_task_loop+weakdep 16777216 8192 100
size: 16777216, bs: 8192, iterations: 100, time: 8.91, performance: 0.188296
./07.multisaxpy_task+reduction 16777216 8192 100
size: 16777216, bs: 8192, iterations: 100, time: 7.03558, performance: 0.238462

```

Override the Number of Threads Used

Let's have a closer look at the third implementation, i.e. *03.multisaxpy_task*, which took 0.648536 seconds to finish using 48 threads. Remember that a full CM node features 48 threads (0-47) divided in two sockets: 0-11,24-35 for the first socket and 12-23,36-47 for the second socket. **Notice that they are indeed not consecutive!**

We can change the threads used by OmpSs-2 with the Linux command `taskset`. For example, the command to run this binary with 24 threads interleaved between the two sockets would be:

```
taskset -c 0-23 ./03.multisaxpy_task 16777216 8192 100
```

Similarly, to run this benchmark using all the 24 threads of the second socket use the following command:

```
taskset -c 12-23,36-47 ./03.multisaxpy_task 16777216 8192 100
```

You can also try to run this example with only 12 threads of the first socket:

```
taskset -c 0-11 ./03.multisaxpy_task 16777216 8192 100
```

or 12 threads interleaved between the two sockets:

```
taskset -c 0-5,12-17 ./03.multisaxpy_task 16777216 8192 100
```

Changing the number of threads assigned to OmpSs-2 affects the performance of the application and not necessarily in a negative way, e.g. see below:

```

$ ./03.multisaxpy_task 16777216 8192 100
size: 16777216, bs: 8192, iterations: 100, time: 0.653537, performance: 2.56714
$ taskset -c 0-23 ./03.multisaxpy_task 16777216 8192 100
size: 16777216, bs: 8192, iterations: 100, time: 0.686265, performance: 2.44471
$ taskset -c 12-23,36-47 ./03.multisaxpy_task 16777216 8192 100
size: 16777216, bs: 8192, iterations: 100, time: 0.650363, performance: 2.57967
$ taskset -c 0-11 ./03.multisaxpy_task 16777216 8192 100

```

```
size: 16777216, bs: 8192, iterations: 100, time: 0.55417, performance: 3.02745
$ taskset -c 0-5,12-17 ./03.multisaxpy_task 16777216 8192 100
size: 16777216, bs: 8192, iterations: 100, time: 0.705859, performance: 2.37685
```

Creating a Dependency Graph

Let's continue with the same example used above and create a dependency graph using only 12 threads of one socket (e.g. the second), which demonstrated to be the affinity giving the best results. Furthermore, we are not longer interested in running 100 iterations (nor using a large number of elements) for graph purposes and hence only one iteration will suffice to generate a complete graph of this application. Run the following command:

```
NANOS6=graph taskset -c 12-23 ./03.multisaxpy_task 196608 8192 1
```

Once it has finished it should have created a script with the name *graph-XXXXX-YYYYYYYYY-script.sh* and a directory *graph-XXXXX-YYYYYYYYY-components*. Execute said script by typing the following (note that it requires the tool dot):

```
bash graph-XXXXX-YYYYYYYYY-script.sh
```

to merge the intermediate results into a single PDF file which should look like this:

which illustrates 24 tasks executed in parallel using 12 threads.

Obtaining statistics

The visual execution of tasks can be further complemented with statistics. Executing the following command:

```
NANOS6=stats taskset -c 12-23 ./03.multisaxpy_task 196608 8192 1
```

will give you the information below:

```
$ NANOS6=stats taskset -c 12-23 ./03.multisaxpy_task 196608 8192 1
size: 196608, bs: 8192, iterations: 1, time: 0.000241, performance: 0.815801
STATS      Total CPUs      12
STATS      Total time      2.42573e+07      ns
STATS      Total threads   12
STATS      Mean threads per CPU      1
STATS      Mean tasks per thread    2.08333

STATS      Mean thread lifetime    3.65355e+09      %
STATS      Mean thread running time    100      %
STATS      Mean effective parallelism    0.123268

STATS      All Tasks instances      25
STATS      All Tasks mean instantiation time    1445      ns      0.885064      %
STATS      All Tasks mean pending time      0      ns      0      %
STATS      All Tasks mean ready time    32446      ns      19.8732      %
STATS      All Tasks mean execution time    119605      ns      73.2582      %
STATS      All Tasks mean blocked time    3702      ns      2.26748      %
STATS      All Tasks mean zombie time    6067      ns      3.71604      %
STATS      All Tasks mean lifetime    163265      ns

STATS      03.multisaxpy_task.cpp:3:13 instances      24
STATS      03.multisaxpy_task.cpp:3:13 mean instantiation time    1251      ns      1.75051      %
STATS      03.multisaxpy_task.cpp:3:13 mean pending time      0      ns      0      %
STATS      03.multisaxpy_task.cpp:3:13 mean ready time    32944      ns      46.0981      %
STATS      03.multisaxpy_task.cpp:3:13 mean execution time    31079      ns      43.4884      %
STATS      03.multisaxpy_task.cpp:3:13 mean blocked time      0      ns      0      %
STATS      03.multisaxpy_task.cpp:3:13 mean zombie time    6191      ns      8.66298      %
STATS      03.multisaxpy_task.cpp:3:13 mean lifetime    71465      ns

STATS      main instances      1
STATS      main mean instantiation time    6089      ns      0.2573      %
```

```

STATS      main mean pending time      0      ns      0      %
STATS      main mean ready time      20505      ns      0.866471      %
STATS      main mean execution time    2244241      ns      94.8339      %
STATS      main mean blocked time     92553      ns      3.91097      %
STATS      main mean zombie time      3108      ns      0.131333      %
STATS      main mean lifetime      2366496      ns

STATS      Phase 1 03.multisaxpy_task.cpp:3:13 instances      24
STATS      Phase 1 03.multisaxpy_task.cpp:3:13 mean instantiation time      1251      ns      1.75051      %
STATS      Phase 1 03.multisaxpy_task.cpp:3:13 mean pending time      0      ns      0      %
STATS      Phase 1 03.multisaxpy_task.cpp:3:13 mean ready time      32944      ns      46.0981      %
STATS      Phase 1 03.multisaxpy_task.cpp:3:13 mean execution time    31079      ns      43.4884      %
STATS      Phase 1 03.multisaxpy_task.cpp:3:13 mean blocked time      0      ns      0      %
STATS      Phase 1 03.multisaxpy_task.cpp:3:13 mean zombie time      6191      ns      8.66298      %
STATS      Phase 1 03.multisaxpy_task.cpp:3:13 mean lifetime      71465      ns

STATS      Phase 1 instances      24
STATS      Phase 1 mean instantiation time      1251      ns      1.75051      %
STATS      Phase 1 mean pending time      0      ns      0      %
STATS      Phase 1 mean ready time      32944      ns      46.0981      %
STATS      Phase 1 mean execution time    31079      ns      43.4884      %
STATS      Phase 1 mean blocked time      0      ns      0      %
STATS      Phase 1 mean zombie time      6191      ns      8.66298      %
STATS      Phase 1 mean lifetime      71465      ns
STATS      Phase 1 effective parallelism      0.165278

```

Additionally, you can get information related to hardware counters via PAPI. For this, firstly load the PAPI module:

```
module load PAPI/5.6.0
```

and then execute:

```
NANOS6=stats-papi taskset -c 12-23 ./03.multisaxpy_task 196608 8192 1
```

to get statistics:

```

$ NANOS6=stats-papi taskset -c 12-23 ./03.multisaxpy_task 196608 8192 1
size: 196608, bs: 8192, iterations: 1, time: 0.000236, performance: 0.833085
STATS      Total CPUs      12
STATS      Total time      3.06985e+07      ns
STATS      Total threads    12
STATS      Mean threads per CPU      1
STATS      Mean tasks per thread    2.08333

STATS      Mean thread lifetime    2.88807e+09      %
STATS      Mean thread running time    100      %
STATS      Mean effective parallelism    0.13271

STATS      All Tasks instances      25
STATS      All Tasks mean instantiation time    2708      ns      1.52238      %
STATS      All Tasks mean pending time      0      ns      0      %
STATS      All Tasks mean ready time      9032      ns      5.07761      %
STATS      All Tasks mean execution time    162959      ns      91.6123      %
STATS      All Tasks mean blocked time      1105      ns      0.621209      %
STATS      All Tasks mean zombie time      2075      ns      1.16652      %
STATS      All Tasks mean lifetime    177879      ns
STATS      All Tasks Real frequency    0.658047      GHz
STATS      All Tasks Virtual frequency    0.782649      GHz
STATS      All Tasks IPC      1.66625
STATS      All Tasks L2 data cache miss ratio    3.203
STATS      All Tasks Real nsecs      3804026      nsecs

```

STATS	All Tasks Virtual nsecs	3198406	nsecs				
STATS	All Tasks Instructions	4171011	instructions				
STATS	All Tasks Total cycles	2503229					
STATS	All Tasks Instr completed	4171011					
STATS	All Tasks L2D cache accesses	16754					
STATS	All Tasks L2D cache misses	53663					
STATS	All Tasks Reference cycles	2054784					
STATS	03.multisaxpy_task.cpp:3:13 instances	24					
STATS	03.multisaxpy_task.cpp:3:13 mean instantiation time	2498	ns	4.60435	%		
STATS	03.multisaxpy_task.cpp:3:13 mean pending time	0	ns	0	%		
STATS	03.multisaxpy_task.cpp:3:13 mean ready time	8237	ns	15.1826	%		
STATS	03.multisaxpy_task.cpp:3:13 mean execution time	41452	ns	76.405	%		
STATS	03.multisaxpy_task.cpp:3:13 mean blocked time	0	ns	0	%		
STATS	03.multisaxpy_task.cpp:3:13 mean zombie time	2066	ns	3.80808	%		
STATS	03.multisaxpy_task.cpp:3:13 mean lifetime	54253	ns				
STATS	03.multisaxpy_task.cpp:3:13 Real frequency	3.16748	GHz				
STATS	03.multisaxpy_task.cpp:3:13 Virtual frequency	3.18873	GHz				
STATS	03.multisaxpy_task.cpp:3:13 IPC	1.72954					
STATS	03.multisaxpy_task.cpp:3:13 L2 data cache miss ratio	3.96831					
STATS	03.multisaxpy_task.cpp:3:13 Real nsecs	755566	nsecs				
STATS	03.multisaxpy_task.cpp:3:13 Virtual nsecs	750532	nsecs				
STATS	03.multisaxpy_task.cpp:3:13 Instructions	4139211	instructions				
STATS	03.multisaxpy_task.cpp:3:13 Total cycles	2393243					
STATS	03.multisaxpy_task.cpp:3:13 Instr completed	4139211					
STATS	03.multisaxpy_task.cpp:3:13 L2D cache accesses	13316					
STATS	03.multisaxpy_task.cpp:3:13 L2D cache misses	52842					
STATS	03.multisaxpy_task.cpp:3:13 Reference cycles	1964416					
STATS	main instances	1					
STATS	main mean instantiation time	7755	ns	0.246588	%		
STATS	main mean pending time	0	ns	0	%		
STATS	main mean ready time	28131	ns	0.894488	%		
STATS	main mean execution time	3079121	ns	97.9076	%		
STATS	main mean blocked time	27636	ns	0.878749	%		
STATS	main mean zombie time	2284	ns	0.0726249	%		
STATS	main mean lifetime	3144927	ns				
STATS	main Real frequency	0.0360792	GHz				
STATS	main Virtual frequency	0.0449312	GHz				
STATS	main IPC	0.289128					
STATS	main L2 data cache miss ratio	0.238802					
STATS	main Real nsecs	3048460	nsecs				
STATS	main Virtual nsecs	2447874	nsecs				
STATS	main Instructions	31800	instructions				
STATS	main Total cycles	109986					
STATS	main Instr completed	31800					
STATS	main L2D cache accesses	3438					
STATS	main L2D cache misses	821					
STATS	main Reference cycles	90368					
STATS	Phase 1 03.multisaxpy_task.cpp:3:13 instances	24					
STATS	Phase 1 03.multisaxpy_task.cpp:3:13 mean instantiation time	2498	ns	4.60435	%		
STATS	Phase 1 03.multisaxpy_task.cpp:3:13 mean pending time	0	ns	0	%		
STATS	Phase 1 03.multisaxpy_task.cpp:3:13 mean ready time	8237	ns	15.1826	%		
STATS	Phase 1 03.multisaxpy_task.cpp:3:13 mean execution time	41452	ns	76.405	%		
STATS	Phase 1 03.multisaxpy_task.cpp:3:13 mean blocked time	0	ns	0	%		
STATS	Phase 1 03.multisaxpy_task.cpp:3:13 mean zombie time	2066	ns	3.80808	%		
STATS	Phase 1 03.multisaxpy_task.cpp:3:13 mean lifetime	54253	ns				
STATS	Phase 1 03.multisaxpy_task.cpp:3:13 Real frequency	3.16748	GHz				
STATS	Phase 1 03.multisaxpy_task.cpp:3:13 Virtual frequency	3.18873	GHz				
STATS	Phase 1 03.multisaxpy_task.cpp:3:13 IPC	1.72954					

```

STATS      Phase 1 03.multisaxpy_task.cpp:3:13 L2 data cache miss ratio      3.96831
STATS      Phase 1 03.multisaxpy_task.cpp:3:13 Real nsecs                  755566      nsecs
STATS      Phase 1 03.multisaxpy_task.cpp:3:13 Virtual nsecs              750532      nsecs
STATS      Phase 1 03.multisaxpy_task.cpp:3:13 Instructions                4139211     instructions
STATS      Phase 1 03.multisaxpy_task.cpp:3:13 Total cycles                2393243
STATS      Phase 1 03.multisaxpy_task.cpp:3:13 Instr completed                4139211
STATS      Phase 1 03.multisaxpy_task.cpp:3:13 L2D cache accesses             13316
STATS      Phase 1 03.multisaxpy_task.cpp:3:13 L2D cache misses              52842
STATS      Phase 1 03.multisaxpy_task.cpp:3:13 Reference cycles              1964416

```

```

STATS      Phase 1 instances          24
STATS      Phase 1 mean instantiation time      2498      ns      4.60435      %
STATS      Phase 1 mean pending time           0      ns      0      %
STATS      Phase 1 mean ready time            8237      ns      15.1826      %
STATS      Phase 1 mean execution time         41452      ns      76.405      %
STATS      Phase 1 mean blocked time           0      ns      0      %
STATS      Phase 1 mean zombie time           2066      ns      3.80808      %
STATS      Phase 1 mean lifetime              54253      ns
STATS      Phase 1 Real frequency              3.16748      GHz
STATS      Phase 1 Virtual frequency           3.18873      GHz
STATS      Phase 1 IPC                      1.72954
STATS      Phase 1 L2 data cache miss ratio      3.96831
STATS      Phase 1 Real nsecs                  755566      nsecs
STATS      Phase 1 Virtual nsecs              750532      nsecs
STATS      Phase 1 Instructions                4139211     instructions
STATS      Phase 1 Total cycles                2393243
STATS      Phase 1 Instr completed            4139211
STATS      Phase 1 L2D cache accesses           13316
STATS      Phase 1 L2D cache misses            52842
STATS      Phase 1 Reference cycles            1964416
STATS      Phase 1 effective parallelism        0.217033

```

Tracing with Extrae

THIS SECTION IS WORK IN PROGRESS, PLEASE IGNORE IT

To get traces of this benchmark using Extrae firstly load the corresponding module:

```
module load Extrae/3.6.1
```

and charge the Extrae environment in your active session:

```
source /usr/local/software/skylake/Stages/2018b/software/Extrae/3.6.1-ipsmpi-2018b-mt/etc/extrae.sh
```

Then copy a preconfigured *extrae.xml* file to instrument OmpSs-2 to your current working directory *multisaxpy/*:

```
cp /usr/local/software/skylake/Stages/2018b/software/Extrae/3.6.1-ipsmpi-2018b-mt/share/example/OMPSS/extrae.xml
.
```

The next step is to create a new file *trace.sh*:

```
touch trace.sh
```

with the necessary permission to be executed:

```
chmod +x trace.sh
```

and fill it with the following text:

```
#!/bin/bash
export EXTRAE_CONFIG_FILE=extrae.xml
export NANOS6="extrae"
```

\$*

Now execute the benchmark keeping its original size but only 20 iterations with the following command:

```
taskset -c 12-23 ./trace.sh ./03.multisaxpy_task 16777216 8192 20
```

References

- <https://pm.bsc.es/gitlab/ompss-2/examples/multisaxpy>
- <https://pm.bsc.es/ftp/ompss-2/doc/examples/local/sphinx/03-fundamentals.html>
- <https://en.wikipedia.org/wiki/AXPY>

Dot-product Benchmark

Users must clone/download this example's repository from <https://pm.bsc.es/gitlab/ompss-2/examples/dot-product> and transfer it to a DEEP working directory.

Description

This benchmark runs a dot-product operation. The dot-product (also known as scalar product) is an algebraic operation that takes two equal-length sequences of numbers and returns a single number.

There are **3 implementations** of this benchmark.

Execution Instructions

```
./dot_product SIZE CHUNK_SIZE
```

where:

- SIZE is the number of elements of the vectors used on the dot-product operation.
- The dot-product operation will be applied to the vector in blocks that contains CHUNK_SIZE elements.

References

- <https://pm.bsc.es/gitlab/ompss-2/examples/dot-product>
- https://en.wikipedia.org/wiki/Dot_product

Mergesort Benchmark

Users must clone/download this example's repository from <https://pm.bsc.es/gitlab/ompss-2/examples/mergesort> and transfer it to a DEEP working directory.

Description

This benchmark is a recursive sorting algorithm based on comparisons.

There are **6 implementations** of this benchmark.

Execution Instructions

```
./mergesort N BLOCK_SIZE
```

where:

- N is the number of elements to be sorted. Mandatory for all versions of this benchmark.

- BLOCK_SIZE is used to determine the threshold when the task becomes *final*. If the array size is less or equal than BLOCK_SIZE, the task will become final, so no more tasks will be created inside it. Mandatory for all versions of this benchmark.

References

- <https://pm.bsc.es/gitlab/ompss-2/examples/mergesort>
- https://en.wikipedia.org/wiki/Merge_sort

Nqueens Benchmark

Users must clone/download this example's repository from <https://pm.bsc.es/gitlab/ompss-2/examples/nqueens> and transfer it to a DEEP working directory.

Description

This benchmark computes, for a NxN chessboard, the number of configurations of placing N chess queens in the chessboard such that none of them is able to attack any other. It is implemented using a branch-and-bound algorithm.

There are **7 implementations** of this benchmark.

Execution Instructions

```
./n-queens N [threshold]
```

where:

- N is the chessboard's size. Mandatory for all versions of this benchmark.
- threshold is the number of rows of the chessboard that will generate tasks.

The remaining rows (N - threshold) will not generate tasks and will be executed in serial mode. Mandatory from all versions of this benchmark except from 01 (sequential version) and 02 (fully parallel version).

References

- <https://pm.bsc.es/gitlab/ompss-2/examples/nqueens>
- https://en.wikipedia.org/wiki/Eight_queens_puzzle

Matmul Benchmark

Users must clone/download this example's repository from <https://pm.bsc.es/gitlab/ompss-2/examples/matmul> and transfer it to a DEEP working directory.

Description

This benchmark runs a matrix multiplication operation $C = A \cdot B$, where A has size N?M, B has size M?P, and the resulting matrix C has size N?P.

There are **3 implementations** of this benchmark.

Execution Instructions

```
./matmul N M P BLOCK_SIZE
```

where:

- N is the number of rows of the matrix A.
- M is the number of columns of the matrix A and the number of rows of the matrix B.
- P is the number of columns of the matrix B.
- The matrix multiplication operation will be applied in blocks that contains BLOCK_SIZE?BLOCK_SIZE elements.

References

- <https://pm.bsc.es/gitlab/ompss-2/examples/matmul>
- <https://pm.bsc.es/ftp/ompss-2/doc/examples/local/sphinx/02-examples.html>
- https://en.wikipedia.org/wiki/Matrix_multiplication_algorithm

Cholesky Benchmark (OmpSs-2+MKL)

Users must clone/download this example's repository from <https://pm.bsc.es/gitlab/ompss-2/examples/cholesky> and transfer it to a DEEP working directory.

Description

This benchmark is a decomposition of a Hermitian, positive-definite matrix into the product of a lower triangular matrix and its conjugate transpose. This Cholesky decomposition is carried out with OmpSs-2 using tasks with priorities.

There are **3 implementations** of this benchmark.

The code uses the CBLAS and LAPACK interfaces to both BLAS and LAPACK. By default we try to find MKL, ATLAS and LAPACK from the MKLROOT, LIBRARY_PATH and C_INCLUDE_PATH environment variables. If you are using an implementation with other linking requirements, please edit the LIBS entry in the makefile accordingly.

The Makefile has three additional rules:

- **run:** runs each version one after the other.
- **run-graph:** runs the OmpSs-2 versions with the graph instrumentation.
- **run-extrae:** runs the OmpSs-2 versions with the extrae instrumentation.

For the graph instrumentation, it is recommended to view the resulting PDF in single page mode and to advance through the pages. This will show the actual instantiation and execution of the code. For the extrae instrumentation, extrae must be loaded and available at least through the LD_LIBRARY_PATH environment variable.

Execution Instructions

```
./cholesky SIZE BLOCK_SIZE
```

where:

- SIZE is the number of elements per side of the matrix.
- The decomposition is made by blocks of BLOCK_SIZE by BLOCK_SIZE elements.

References

- <https://pm.bsc.es/gitlab/ompss-2/examples/cholesky>
- <https://pm.bsc.es/ftp/ompss-2/doc/examples/02-examples/cholesky-mkl/README.html>
- https://en.wikipedia.org/wiki/Eight_queens_puzzle