StarSs hands-on

Single node examples

Initializations

Set the path and LD_LIBRARY_PATH for the compiler and runtime with the env_RES.sh script:

```
> source env_RES.sh
```

Matrix multiply

Compiling and executing with OmpSs (SMPSs syntax)

Change to simple*_matmul* directory. The implementations of the matrix multiply example is based on a hypermatrix (matrix of pointers to blocks of BSIZE x BSIZE floats). The example implements a blocked matrix multiplication where each task multiplies a block of A by a block of B at leaves the result in a block of C.

Open the file simple_matmul.c. Have a look to the code and observe the compiler directives.

Type “sscc –h” to see the sscc compiler options.
Compile the example with:

```
sscc -k –-verbose  -o simple_matmul -O3  simple_matmul.c
```

The option “--verbose” shows the different compilation steps. The option “-k” generates the different intermediate files.

Check the compilation steps and generated files. Look into the generated files and identify the glue code necessary to link to the NANOS++ runtime libraries.

Execute with 1,2 and 4 threads. For example, to execute with 4 threads:

```
export NX_PES=4
./simple_matmul 16
```

(the parameter "16" indicates the size of the matrix in number of blocks; initially blocks are of size 128 x 128 floats, the total size of the matrix in this case is 2048).

Compiling and executing with OmpSs (OmpSs syntax)

Change to simple_*matmul_ompss* directory. Open the file simple_matmul_ompss.c. Have a look to the code and observe the compiler directives. The original SMPSs directives have been replaced by their equivalent for OmpSs. In this case, the "start" and "finish" compiler directives are not required, but we added a "taskwait" to wait for all tasks before we measure time.

Compile with:

```
mcc -–ompss -O3 simple_matmul_ompss.c  -o simple_matmul_ompss
```

And execute with:

```
Export NX_PES=4
./simple_matmul_ompss 16
```

With the OmpSs syntax, it is not mandatory to indicate all the arguments in the input/output argument, but at least the minimum set that guarantees that the dependences are kept. Modify the compiler directive in such a way that the number of input/output clauses are reduced but still the program behaves correctly.

Stream

Generating a tracefile

This example is the implementation in OmpSs of the STREAM benchmark (it is located in the *stream* directory). Compared with the official OpenMP version, this implementation encapsulates the operations in the benchmark (copy, scale, add and triadd) in StarSs tasks. There are two versions: stream.c and stream_nb.c. The first one emulates the behaviour of the OpenMP version, by inserting taskwaits between each set of tasks. The second version eliminates these taskwaits, allowing the runtime to detect the dependencies between the tasks.

You can run both examples and also, you can extract traces for both cases (reduce first the size of N before, to get a smaller tracefile).

For example:

```
> mcc --ompss -O3 stream.c -o stream

> export NX_PES=4
> ./stream
```

Generating tracefiles
Compile this example again, with the *"–instrumentation"* option and setting the "NX_INSTRUMENTATION=extrae" before execution. For example:

```
> mcc ––ompss  -–instrumentation –O3 stream.c –o stream_trace
> export NX_PES=4
> export NX_INSTRUMENTATION=extrae
> ./stream_trace
```

When running, it will generate a Paraver tracefile, composed of three files: stream_trace_001.prv, stream_trace_001.row and stream_trace_001.pcf. Open the paraver tracefile with Paraver (use the paraver.sh script for this):

```
../paraver.sh
```

(Those interested can install in their system the Paraver browser. It is downloadable from:

 http://www.bsc.es/plantillaC.php?cat_id=625
)

Browse with the File -> Load Trace menu to open your tracefile. Load the configuration file "user_functions.cfg" from the ../paraver_cfgs directory with the File-> Load Configuration menu.

The graphical window shows the different threads of the execution (main thread and one for each additional CPU used). Shows the tasks executed by each thread. Select the option "Info Panel" from the right-click menu. Then select the "Colors" tab. This will show a legend mapping colours to task types.

Select the View -> Event Flags from the right-click menu. This will show small green flags, denoting the occurrence of events. In this case, there is one such flag per task instance. You can zoom to see more detail by left-clicking in the window and dragging. Also, by double left-clicking on top of a task, the tab "What/Where" will give you more information about the task, like task type and duration.

Open the "tread_state.cfg" configuration file. In this case, different colours denote different states of the thread, for example:

- RUNTIME: the thread is busy performing some runtime operation
- RUNNING: the thread is executing a task
- SYNCHRONIZATION: the thread is waiting for synchronization event (for example, for all tasks to finish)
- ...

Align the two windows in the same time period. To do this, select Copy from the right-click menu in one window and select "Paste -> time" from the same menu in the other window. With this view of the two windows you can analyze what is the thread doing while it is not executing tasks. For example, at the beggining the main thread spends a lot of time in the states "CREATION" (that generates tasks) and "SCHEDULING" (that schedules tasks to threads) and it is not able to execute any task.

Open the "3DH_duration_user_functions.cfg" configuration file. This is a different type of configuration file that shows a histogram. The X-axis show the frequency of the tasks' duration for a given range. If you move the cursor over the colored segments, the bottom of the window will show the actual range represented by the column of the segment and the actual value (number of tasks in the range in this case) for that segment.

Another interesting view is the one give by the task.cfg configuration file, where different colours identify the different tasks being executed at each moment.

There is a set of different configuration files that evaluate different aspects such, all of them available in the paraver_cfgs directory. This is useful for analyzing the performance obtained by the applications and to graphically see the parallelism achieved.

Compare the tracefiles of both examples (stream and stream_nb with Paraver. What can you see?

Fibonacci

The Fibonacci numbers are the numbers in the following integer sequence:

    0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

By definition, the first two numbers in the Fibonacci sequence are 0 and 1, and each subsequent number is the sum of the previous two.

The directory fibonacci contains the serial implementation of a C program that computes the Fibonacci numbers. Modify the sequential implementation of the Fibonacci to convert it to StarSs parallel version. Remember that OmpSs supports recursivity in the tasks (tasks that call to other tasks).

Multiple node, MPI/OmpSs examples

Initializations

Set the path and LD_LIBRARY_PATH for OmpSs with the env_RES.sh script:

```
> source env_RES.sh
```

Mod2am

This example implements de mod2am benchmark that multiplies two matrixes. It is a hybrid MPI/OmpSs application, that encapsulates MPI calls in tasks.

Identify the code of the tasks.

Compile and run the application, using the Makefile and execute it with the run_1.sh script.

```
> make
```

```
> mnsubmit run_1.sh
```

Modify the run.sh script and the mod2am.in file to run with different number of MPI processes, different number of OmpSs threads, and different matrix sizes. Compare the performance achieved in each case.

Compare the performance if the communications are not encapsulated into tasks.