Simgrid hands-on
13/06/14 – Sophia-Antipolis

Prerequisites:

- SimGrid (http://simgrid.gforge.inria.fr/):

A recent (two days old) version of SimGrid is available in the VM in simgrid_tools/simgrid . It is installed in the home folder, so smpicc and smpiff scripts should be directly in the PATH. It's a development version, please report any issue you encounter.

- NAS benchmarks (https://www.nas.nasa.gov/publications/npb.html):

We will use in this session the NAS benchmarks, as they represent a good subset of C/F77 MPI applications, with various specificities.

We will use four of them:

MG : Multi-Grid fortran code, with allreduce collective operations
DT : Data traffic. Not many communications, but huge ones.
LU : Lower-Upper Gauss-Seidel solver. C code, 50 iterations
EP : embarrassingly parallel. Converted to C for tutorial purpose

Makefile has been edited to use smpicc (for C codes) and smpiff (F77) compilation scripts, which is the only needed modification to use SMPI with NAS benchmarks.

Each benchmark is provided with several dataset sizes (CLASS), and has to be be compiled for a specific number of processes (NPROCS).To build each NAS, go to the NPB3.3-MPI folder and for a class B LU benchmark with 16 nodes, type (make sure smpicc and smpiff are in your PATH):

make LU CLASS=B NPROCS=16

Executable is then located in NPB3.3-MPI/bin folder (if an error occurs during compilation you may need to create the bin folder), with the name lu.B.16 (or dt.B.x for DT)

- Platforms files:

For this session, three 256 node platforms are provided :
- a 4 cabinets cluster (ethernet_cluster.xml), tuned to reproduce the behaviour of real Ethernet clusters
- a torus 4D (4*4*4*4) cluster (torus_cluster.xml)
- a fat-tree cluster (fat_tree-cluster.xml)

A hostfile allowing to deploy processes on all platforms is also provided (hostfile)

- Visualization:

For simple Gantt chart visualization of traces, we will use the Vite tool (https://gforge.inria.fr/scm/?group_id=1596), which should be already installed on the VM. To display a paje trace, just use “vite tracefile.trace”
For more complex visualization of resource utilization, the viva tool will be necessary. We won't use it in this session.

How to run a code with SMPI

- Important options:

When running a code with smpirun, some options are necessary:

- hostfile and platform: **-hostfile ./hostfile -platform ./myplatformfile.xml**

- number of processes to deploy: **-np x**

Some options are also strongly advised :

- Output the end timing of the simulation : **--cfg=smpi/display_timing:yes** . this makes it more easy to compare runs between them

- Power of the simulating platform : as we provide a power for the simulated nodes, we need to know the power in flops of the simulating one to provide better computation times. For this session, use --cfg=smpi/running_power:1Gf as nodes in the platform files also use this power. The default value being 20000 flops, timings would be very small and cause issues if this option is forgotten

- Do we want to privatize global variables? if simulation fails, try with **--cfg=smpi/privatize_global_variables:yes** to trigger the mmap-based dynamic switching of the global memory segments.

- Tracing? to output a paje trace you can then load with ViTe, just add **-trace** to the command line, a file smpi_simgrid.trace will be generated (add **--cfg=tracing/filename:myfile.trace**) to output with a different file name.

Example of a MG run with 128 nodes :

smpirun -hostfile hostfile -platform ./torus_cluster.xml -np 128 –cfg=smpi/privatize_global_variables:yes -trace --cfg=smpi/running_power:1Gf ./NPB3.3-MPI/bin/mg.A.128

- Example 1 – Replay :

Compile and run the LU, class A, 32 nodes benchmark.

This takes quite a long time to simulate (a couple minutes) ... but not too much because we reduced the number of iterations by a factor 5. This is because all code from all processes has to be really executed, and is serialized.

We provide several methods to speed things up. One of them is to capture a time independent trace of the running application (either live, or in simulation), and replay it on a different platform. This implies that the number of nodes may not be changed for this run.

To generate such a trace, use **-trace-ti** flag instead of -trace in smpirun. It is also advised to name the trace by using -**-cfg=tracing/filename:mytracefile.txt** . After the run of LU, a folder will have been created, containing one file per process as well as the file mytracefile.txt, (just a list of the other file names)

To replay this trace, use smpirun with the executable examples/smpi/smpi_replay, like this :

smpirun (classic SMPI parameters) -ext smpi_replay examples/smpi/smpi_replay mytracefile.txt

You can generate another trace from here to compare live and replay version.

- Example 2 – Sampling :

Use the EP benchmark, class B, 16 processes

The second method to speed up simulations is to sample the computation parts in the code. This means that the person doing the simulation needs to know the application and identify parts that are compute intensive and take time, while being regular enough not to ruin simulation accuracy. Furthermore there should not be any MPI calls inside such parts of the code.

This is simply done in C by calling **SMPI_SAMPLE_LOCAL/GLOBAL** macros, depending on the type of sampling you want to use.

An example is provided, in EP-sampling. Check in ep.c its behaviour, and build it, to compare simulation timings.

- Example 3 – Memory folding :

Use the DT benchmark, class C (85 nodes, not necessary for compilation, but for execution. A parameter has to be given while running: use **BH**)

This example allocates a lot of memory, and performs huge transfers. The transfers are not an issue in SimGrid, but the allocation of memory may be too much for a personal computer.

If you try to run the DT sample as such, there are good chances that **it will not finish (it needs 35 GB of memory)**. To avoid this, we provide a way for processes to share their memory allocations.

This will lose the content of the buffers, but if their content is not important for the accuracy of the simulation, it is a good trade-off. To do this, simply edit dt.c file, and replace allocations by SMPI_SHARED_MALLOC. Don't forget to also replace the calls to free (for these buffers only) by SMPI_SHARED_FREE.

Once done, recompile, and test again. If it crashes, you may have forgotten a free somewhere.

- Example 4 – Collective Communications :

Compile the MG benchmark, class A, for 128 nodes

Warning: MG seems to trigger Bus errors on the provided vm. This is under investigation, if you encounter this issue, please use allreduce_coll.c (smpicc allreduce_coll.c -o allreduce_coll). It can then be used with smpirun, by providing a size in Bytes as argument: try 1000

The default (for now) collective algorithms used in SMPI are naïve ones, which don't accurately simulate real collective operations from existing libraries.

MG uses a lot of allreduce operations, which can be tuned in SMPI, as algorithms used in OpenMPI, MPICH, and Star-MPI can be used. SMPI can use one of the following algorithms:

lr : logical ring reduce-scatter then logical ring allgather
rab1 : variations of the Rabenseifner algorithm : reduce_scatter then allgather
rab2 : variations of the Rabenseifner algorithm : alltoall then allgather
rab_rsag : variation of the Rabenseifner algorithm : recursive doubling reduce_scatter then recursive doubling allgather
rdb : recursive doubling

redbcast : reduce then broadcast, using default or tuned algorithms if specified : check http://simgrid.gforge.inria.fr/simgrid/3.12/doc/group__SMPI__API.html for the list
ompi_ring_segmented : ring algorithm used by OpenMPI
ompi : use openmpi selector for the allreduce operations
mpich : use mpich selector for the allreduce operations
default : naive one, by default – not realistic – avoid

A few others are available, to account for SMP capabilities. As we only use one process/node in these examples, they are not listed here.

To use one or another, add the command **--cfg=smpi/allreduce:algoname** to the command line (if you use the redbcast, you may want to use --cfg=smpi/reduce:algoname and --cfg=smpi/bcast:algoname as well, check documentation linked for the values). To have all collective operations in a run use OpenMPI or MPICH selector, you can use **--cfg=smpi/coll_selector:mpich(or ompi)**

What seems to best performing algorithm here for each platform ?

You can compare endtime, or see in the traces how they behave. You can activate the flag **--cfg=tracing/smpi/internals:yes** to see in the trace how each collective performs the operation.