Division of Scientific Computing
Department of Information Technology
Uppsala University

---

# NGSSC: HPC II
December 14, 2011

## Hands-on: experience with the parallel debugger TotalView and the parallel performance analyser from Sun Studio 12.1

The task of this computer lab is to get acquainted with the parallel debugger TotalView and the parallel performance analyser from Sun Studio 12.1.

To play with, you are provided with some parallel codes in C++ and Fortran. These are to be downloaded from `http://user.it.uu.se/~maya/Courses/NGSSC/HPCII/Files_2011/performance`.
You are welcome to use any of your available parallel codes, if you wish.

Preparatory work:

- Login on `kalkyl.uppmax.uu.se`
- Load the compilers and mpi modules: `module load pgi openmpi`
- Load TotalView: `module load totalview`
- Load Sun Studio: `module load java/sun_jdk1.6.0_04 sunstudio` (`module check avail`)
- copy the tar files the above url address and extract the directories.

**Exercise 1 (TotalView)**

1. Compile the code to be examined with the flag '-g'. There are makefiles for both cases.
2. Start TotalView
3. Try to put breakpoints, dive in subroutines, variables, etc
4. Stop the execution in different PEs and then continue to watch the progress of the execution.

As an example, you may compile using `Makefile_cg_dalig` and run `mpi_main_dalig`, where there is an error to be localized. To observe the error, do the following two runs and check the output (which should be the same in terms of residual norms and iterations performed, but it is not):

```
mpirun -np 1 mpi_main_dalig
mpirun -np 8 mpi_main_dalig
```

Some explanation about the algorithm will be provided during the lab.

**Exercise 2 (Sun Studio 12.1, Parallel performance analyser, MPI programs)**

1. Collect some performance data for your test application by issuing
   `collect -M OPENMPI mpirun -np xx -- executable`
2. Start sunstudio
3. Open the current experiment
4. Repeat the major steps from the video demo how to filter and reduce the amount of collected information, detect large amount of time spent during the execution and localize the source code where this happens.

Unless you want to analyse the performance of some of your own programs, you may try to do the following:

(i) Try to detect if there is a performance bottleneck in the code `wave`.

Compile with the corresponding Makefile. The program reads its input from the file `fort.1`, and by changing that you can adjust the size of the problem to be solved.

(ii) Try to understand what is the most time consuming part in the code - communication, certain serial function, both.

Compile using `Makefile_cg_fast` and run `mpi_main_fast`. This version of the code uses as input file `fort.1`, where the size of the local problems are determined. In this way, the code achieves almost perfect load balance between the PEs. However, with increasing the problem size

**Exercise 3 (Sun Studio 12.1, Parallel performance analyser, OpenMP programs)**
Play with the code `test_openmp_1.f`. The program is very simple - it finds the sum of the numbers from 1 to N, for given N.

1. The size of the problem is currently `5000`. Make it larger by changing the corresponding liens as follows:

```
allocate(ivar(50000000))
do i=1,50000000
```

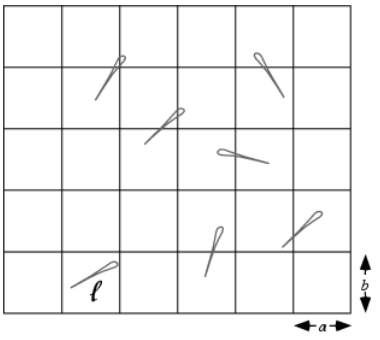
2. Run `test_openmp_1.f` for various number of threads (1,2,4,8). You see that the execution time does not scale with the number of threads.
3. Find the bottleneck, use SunStudio performance analyser.
4. Suggest a solution and time that to see the difference.
5. In `test_openmp_1.f` change the line, say to`do i=1,50000001`. Is there an indication for a memory leak during execution?
6. Try TotalView to see if it can detect the memory leak.

**Exercise 4 (Computing the value of $\pi$ using a Monte Carlo method)**

Buffon-Laplace-needle problem

Problem Statement: More than 200 years before Metropolis coined the name 'Monte Carlo' method, George Louis Leclerc, Comte de Buffon, proposed the following problem.
*'If a needle of length $\ell$ is dropped at random on the middle of a horizontal surface ruled with parallel lines a distance $d > \ell$ apart, what is the probability that the needle will cross one of the lines?'* This problem was first solved by Buffon (1777, pp. 100-104), but his derivation contained an error. A correct solution was given by Laplace (1812, pp. 359-362; Laplace 1820, pp. 365-369).
`http://mathworld.wolfram.com/Buffon-LaplaceNeedleProblem.html`

We reformulate somewhat the original problem in the following way. Imagine that a needle of length $\ell$ is dropped onto a floor with a grid of equally spaced parallel lines distances $a$ and $b$ apart, where $\ell$ is less than $a$ and $b$.




The probability that the needle will land on at least one line is given by

$$P(\ell, a, b) = \frac{2\ell(a+b) - \ell^2}{\pi\, ab}$$

(Uspensky 1937, p. 256; Solomon 1978, p. 4).

The idea now is to keep dropping this needle over and over on the table, and to record the statistics. Namely, we want to keep track of both the total number of times that the needle is randomly dropped on the table (call this N), and the number of times that it crosses a line (call this C).

If you keep dropping the needle, eventually you will find that the number $\frac{N(2\ell(a+b)-\ell^2)}{Cab}$ approaches the value of $\pi$.
(Note that for large $N$ the quantity $C/N$ approaches the probability $P(\ell, a, b)$.)

In order to get a reasonably accurate approximation of $\pi$ we need to perform a number of trials of order $10^6 - 10^8$. Since the separate trials are completely independent, we can perform those in parallel and sum up the result. This problem is an example of a trivial parallelism.

1. Study the implementation of the above algorithm (`pi_buffon_laplace.f`).

2. Compile and run the program for different numbers of processors (1, 2, 4,...). Amend the code in order to be able to time the execution. Do you observe that it scales? For instance, change the code so that you can have the same total number of trials, distributed evenly over a varying number of processors. What happens with the execution time when you increase the number of processors? Does the time on one PE reduce twice when two processors are used, compared to one PE? Or four times when four PEs are used?

3. How does the accuracy of the computed value of $\pi$ behaves on one and many PEs? Is it better to have more processors used, compared with one? How much more? What do you think is the reason for the error behaviour you observe - the parallelization of the code, the algorithm or something else?