Programming of Supercomputers

Assignment 1
18.10.2013

Deadline: `1.11.2013 @ 08:00 CET`

This lab course deals with the parallelization and tuning of a generalized orthomin solver from the Fire benchmark suite.

The tasks you will have to complete during this semester are: *single-core optimization, analysis of I/O behavior, visualization, 3-steps parallelization and performance tuning.*

1 Sequential Optimization - GCCG Solver

Quite often the great speedup potential of single-core execution of applications is overseen by many developers and instead all the effort is put into the parallelization step. This leads to inefficient usage of resources and poor scalability.

This first assignment concentrates on optimizing the sequential behavior of the gccg solver and consist of the following subtasks:

1. Get to know the running environment (SuperMUC).
   Collect key characteristics (*frequency, number of CPUs/Cores, memory, etc.*) of the system fill in the "Runtime Env" worksheet.

2. Format the given source code according to the coding guidelines.

3. Experiment with different compiler optimization flags and carry out performance measurements using the Performance Application Programming Interface (PAPI).
   Measure the following metrics for the three different execution phases (*input, computation, output*) of each of the four input files (*tjunc, drall, pent, cojack.dat*):

   - Execution time
   - Mflops
   - L2 cache miss rate
   - L3 cache miss rate

   In order to be able to collect the performance data, you should first instrument the code using appropriate functions and counters of the *PAPI* library.
   The bare minimum of compiler flags to test is: *-g, -O1, -O2, -O3, -O3 with prefetching.*
   Fill in the tables in the given worksheets.

   Note: measured performance data has to be the average of at least three runs of the code.
   Note: extra points are given for performance improvement using a creative mixture of other compiler flags.

2 I/O Performance

Manipulating data files is very slow compared to accessing the main memory. This can be considerably improved by using a more concise data format such as binary.

1. Write a small tool to transform the text input files into a binary format;
2. Adapt the GCCG source code to read in correctly the new binary files;
3. Compare execution times for reading in from the two different formats;
4. Compare the storage requirements.

3 Visualization using ParaView (`http://www.paraview.org`)

When you start parallelizing your code, your should always be sure that the new results are still correct. This can be done by comparing either different variables, - e.g. *residual* and *number of iterations* - or the computed results.

A first impression on the results though, is best provided by visualisation. Use the popular visualization software *ParaView* to check the output of the Fire benchmark.

1. Extend the source code to store the results into a VTK file format. Use the provided `vol2mesh` and `write_result_vtk` functions.
2. Use ParaView to generate plots of `VAR`, `CGUP` and `SU` for the `tjunc.dat` input file.

4 Resources

On the course page you can find: the Fire Benchmark source code, 4 input data files, the coding guidelines and the template for the results data sheet.
http://lrr.in.tum.de/∼berariu/teaching/superprog1314.php

5 Submission Requirements

Your GCCG code must:

- compile by simply running `make` without any arguments and generate a `gccg` executable;
- accept 3 arguments in the following format:
  `gccg <format> <input file> <output prefix>`
  - format - file format to use (only `bin` or `text`)
  - input file - input data set
  - output prefix - a common prefix to be prepended to the name of all output files, e.g. **run1.**SU.vtk, **run2_**VAR.vtk, etc.
- generate an error if wrong arguments are given;
- generate SU.vtk, VAR.vtk and CGUP.vtk upon completion (with given prefix);

- generate a text file named `pstats.dat` containing the measured performance with the syntax:
  PHASE METRIC VALUE
  PHASE METRIC VALUE
  ....

  Use only 1 metric per line and 4 decimal places for all floating point numbers. For example:
  INPUT PAPI_L2_TCM 229024
  CALC PAPI_L2_TCA 5816261
  OUTPUT L2MissRate 3,94%
  ...

Your binary conversion tool must:

- compile using `make binconv` and generate a binconv executable;
- accept 3 arguments in the following format:
  `binconv <input file> <output file>`
- generate an error if wrong arguments are given.

Commit all assignment files to your team repository:

1. Store all source code files to a folder A1/code/: *.c, *.h, and Makefile
2. Create a spreadsheet with the collected performance data using the provided template. Store it as Data.ods or Data.xlsx to the A1/data/ folder.
3. Store the generated plots for tjunc.dat to the A1/plots/ folder:
   - SU.jpeg & SU.vtk (input: tjunc.dat)
   - VAR.jpeg & VAR.vtk (input: tjunc.dat)
   - CGUP.jpeg & CGUP.vtk (input: tjunc.dat)
4. Write a lab report and store it as Report.pdf to the A1/report/ folder. The report has to include:
   - a short description of the execution environment;
   - formulas used for computing the metrics (cache miss rate & Mflops);
   - comparison of the achieved Mflops against the peak system performance;
   - short discussion of the measurements data. Highlight relevant results by comparing optimization flags, input files and execution phases;
   - a statement on the performance of text vs. binary files.
5. Store any processing scripts or extra files into A1/scripts/

Please note:

- Code that does not compile will not be taken into consideration.
- The results delivered in your report will be checked against those measured by the automatic tests on your code.