HPC 563: Classical Simulation

Dominik Dahlem

April 3, 2008

Chapter 1

Setup

1.1 General Approach

- GNU autotools is used to maintain this project:
  autoconf version 2.61 automake version 1.10
- Getopt is used to parse command-line parameters.
  A help message can be displayed by specifying "-?" or "-h" to the command-line of the executable.
- The GSL library was chosen to provide some mathematical functions (i.e., `GSL_MAX` and `GSL_MIN`) and to control the floating point environment of the application.
- The cunit library was used to implement the unit tests.
- MPI is used to provide parallel execution using 1D decomposition of the linear system into blocks of rows assigned to participating nodes.
- The application was implemented with OpenMP support. The problem domain lends itself to multi-threaded programming, especially expensive operations such as the matrix-vector multiply and the initialisation of the matrix. OpenMP was not tested on the IITAC cluster though, because the configure script didn't recognise any support for OpenMP. So, OpenMP was tested with gcc 4.2.3.

1.2 Configuration

Each feature of the application is configurable. That means that the libraries required for certain features are checked when the configure script is executed

and an appropriate message is presented to the user. Some features, such as MPI, GSL, and unit tests have inter-dependencies which are enforced.

- The assignment ships with a configure script generated by autotools. The following options are supported:
  - –**enable-test**: enables testing using cunit. The unit tests require GSL to be configured as well. An error message is produced, if the GSL library is not configured or not available.
  - –**enable-gcov**: enables the coverage analysis using gcov and lcov. Call `make lcov` to generate the HTML report.
  - –**enable-gsl**: enables the GSL library.
  - –**enable-mpi**: enables the parallel execution of the application using MPI. If MPI is enabled the unit tests are disabled, because the tests cannot be executed in a parallel fashion.
  - –**enable-openmp**: enables OpenMP in order to support multi-threaded execution.
  - –**enable-debug**: enables debug messages and asserts in some of the functions.
  - –**enable-report**: Enable the report generation which is written in latex.
  - –**enable-plot**: Enable the plot generation

1.3 Make

The project is set up in such a way that all sources can be built with a single make command in the root project directory. Additionally, it is also possible to cd into a c module or the doc directory to build those individually.

- Installing the application was not considered.
- To build the project do one of the following:
  1. No gcov, no tests, no MPI

     ```
     ./configure
     make
     ```

     Compile the sources
  2. No gcov, no tests, MPI enabled

```
./configure --enable-mpi
make
```

Compile the sources for parallel execution

3. No gcov, no tests, MPI and OpenMP enabled

   ```
   ./configure --enable-mpi --enable-openmp
   make
   ```

   Compile the sources for hybrid execution, i.e., that means that the coarser tasks are distributed among several remote nodes, while the finer level is parallelised on a single machine with threads.

4. No gcov

   ```
   ./configure --enable-test
   make check
   ```

   This command will compile everything and run the cunit tests.

5. Enable gcov and testing

   ```
   ./configure --enable-test --enable-gcov
   make lcov
   ```

   This command will compile everything, run the cunit tests, and generate a snazzy HTML coverage report using lcov [1].

6. Enable the report generation

   ```
   ./configure --enable-report --enable-plot
   make
   ```

   This command will compile the c, the gnuplot graphs, and the latex sources.

1.4 Assignment Layout

The assignment is structured in the following way:

```
- src          (the sources)
- test         (unit tests)
- doc          (documentation)
   - coverage  (coverage analysis)
   - eval      (gnuplot script and surface plots)
   - report    (the report)
```

[1] http://ltp.sourceforge.net/coverage/lcov.php

A Doxygen configuration file is provided to generate the code documentation in HTML. doxygen support is integrated into the makefiles. Run: `make doxygen-doc`

```
- doc
   - doxygen
      - html
```

The generated doxygen report details the inter-relationships between the implemented modules and the source files.

The lcov coverage report is provided in the coverage folder. Though, only the infrastructural modules are tested, such as vector and matrix operations and the conjugate gradient method.

Chapter 2

Approach

This chapter outlines the general approach taken for the development of the parallel conjugate gradient solver.

2.1 Configuration of the Code

The C code is implemented in such a way that the features, i.e., MPI, OpenMP, debug, and GSL, can be configured individually. As a result the code is portable across a number of platforms and does not depend on a specific feature to be available. The disadvantage of this approach is that the code becomes a bit harder to maintain, because the features are enclosed in `ifdef` statements.

2.2 Data Structures

The two main data structures, vector and matrix, are implemented as C `structs` with supporting routines, such as allocation, free, add, scale, dot product, multiply, etc. The vector structure provides a pointer to a double primitive type and the length of the vector. The matrix consists of a matrix type (symmetric band SB, and generic band GB), the matrix storage format (compressed diagonal storage CDS), a pointer to diagonal vectors, the number of diagonals, and an index which determines the relative positions of the specified diagonals. Based on these characteristics different matrix-vector multiply routines apply (see Section 2.4.2).

2.3 Domain Decomposition

The finite difference scheme for Poisson's equation is decomposed into equally-sized rows among the available nodes. If the square dimension of the finite difference scheme is not divisible by the number of configured nodes then the dimension is adjusted and appended to the matrix and the respective vectors in the linear system.

The 1D decomposition does not require MPI communication. Instead, each node is responsible for setting up the respective data structures in the appropriate way.

2.4 MPI

2.4.1 Dot Product

The dot product is relatively easy to calculate in a cluster environment. Each node computes the partial dot product of the local vector view. Once the local sum is complete, the `MPI_Allreduce()` routine is executed which sums the local parts into a global result. This result is visible to all nodes. However, the communication just involves sending the partial dot product value to the MPI environment.

2.4.2 Matrix-Vector Multiply

While the serial code takes advantage of the symmetry of the matrix, the parallel code sets up a 5-banded matrix on each node. Both matrix-vector multiplications are solved with different functions that take advantage of the given structure. In fact the 5-banded matrix-vector multiplication is implemented as a generic band matrix-vector multiply. Consequently, the conjugate gradient solver can be used for any 2 dimensional finite difference problem. The solver interprets the structure of the matrix and delegates to the appropriate multiplication routine. However, the solver does not test for the well-posedness of the problem.

MPI communication only takes place to retrieve the global u-vector before calculating the matrix-vector product. For this the `MPI_Allgather()` method is called, which results in any node having a global u-vector.

2.5 OpenMP

Much of the effort in implementing the parallel conjugate gradient solver went into parallelising the coarse tasks among configured nodes in a cluster with MPI. However, there still are operations on a finer scale that do not take advantage of multi-processor nodes. Therefore, the GNU profiler (gprof) was used in order to learn more about where the application is spending most of its time. This should indicate which routines can be further improved using multi-threaded execution. OpenMP, which is part of the gcc 4.2 and newer, provides a neat way of declaring which parts of the code should be parallelised using `pragma` directives.

The following steps were executed in the root directory of the project to retrieve a flat execution profile of the application:

```
./configure CFLAGS=''-pg'' --enable-gsl
make clean all
./src/main/pdepoiss_solv -d 0.01
gprof ./src/main/pdepoiss_solv gmon.out
```

The flat profile looks like this:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 56.69      0.85     0.85      403     0.00     0.00  dcdssbmv
 16.01      1.09     0.24      805     0.00     0.00  dotProduct
 12.67      1.28     0.19      805     0.00     0.00  daxpy
 10.00      1.43     0.15      402     0.00     0.00  add
  4.67      1.50     0.07      402     0.00     0.00  scale
  0.00      1.50     0.00    63997     0.00     0.00  bound_cond
```

This profile shows that two methods, `dotProduct()` and `dcdssbmv()` , are responsible for more than 70% of the runtime of the application.

In order to improve the performance, the matrix-vector multiply (`dcdssbmv`) was declared to run in a parallel execution contexts.

The results of the speedup $S_p = T_1/T_p$ and the efficiency $E_p = S_p/p$ are presented in Table 2.1.

The results were recorded using a more fine-grained mesh with a delta value of $d = 0.005$ which results in 501 grid points in both dimensions. These results show that the highest speedup is obtained with 2 threads. Using more threads involves a higher overhead for thread management, because the test runs were performed on a 2-core CPU [1].

[1] i686 Intel(R) Pentium(R) D CPU 3.00GHz GenuineIntel GNU/Linux

Threads (p)	**Execution Time** (T)	**Speedup** (S)	**Efficiency** (E)
1	15.73	1	1
2	12.43	1.26	0.63
4	13.22	1.19	0.3
8	13.93	1.13	0.14

Table 2.1: Performance Measures with 501 Grid Points

Chapter 3

Execution

The conjugate gradient solver for Poisson's equation accepts a number of command-line arguments which are parsed with getopt. The executable resides in the src/main folder and can be called with

```
pdepoiss_solv
```

- -s : Space dimension (Note: specify grid dimensions or the delta value)
- -t : Time dimension (Note: specify grid dimensions or the delta value)
- -d : Delta value (Note: specify grid dimensions or the delta value)
- -r : Error threshold for the conjugate gradient method
- -f : File name for the result surface
- -1 : Lower bound of the domain in the x dimension
- -2 : Upper bound of the domain in the x dimension
- -3 : Lower bound of the domain in the y dimension
- -4 : Upper bound of the domain in the y dimension
- -?/-h : The help message of the application

The space and time dimensions are required to be equal. This is ensured by the application itself. The default value for both is 26. If only one dimension is specified on the command-line then both are set to the highest value, i.e., 26 if the specified value is smaller than 26 or otherwise the specified value. The delta value will be calculated based on this setting.

Alternatively, the delta value can be specified upon which the space and time dimensions are derived.

The domain of the function space can be specified as well. The default values are set to the square region provided in the assignment description.

The file name of the gnuplot data for the surface plot can be specified which is set to result.dat as default name.

3.1 Environment Variables

GSL and OpenMP allow some environment variables to be set for the application to control the floating point environment and the multi-threading behaviour. The following list details the ones used throughout the development of the application and to retrieve the results.

- **GSL_IEEE_MODE**: Sets control variables values for the floating point environment. The variables used are *double-precision*, *mask-denormalized*, and *mask-underflow*. This ignores any errors related to denormalised and under-flowing small numbers, but it traps overflows, division by zero, and invalid operations [1].
- **OMP_NUM_THREADS**: Controls the maximum number of threads available to the application. Different values were used ranging from 2-8.
- **OMP_NESTED**: Indicates to the OpenMP environment whether nested threads are allowed. Some parts of the code support thread nesting. These parts can be optimised with setting this variable to true.

To summarise, the following command-line was used to run the application:

```
GSL_IEEE_MODE="double-precision,mask-underflow,mask-denormalized" \
    ./src/main/pdepoiss_solv -d 0.01
```

Additionally, if OpenMP was compiled in, the variable *OMP_NUM_THREADS* was set to 2, i.e., it was exported to the shell before executing the application. The configure script, however, did not recognise OpenMP on the IITAC cluster. So, the only option enabled on the IITAC cluster was *–enable-mpi*.

[1] http://www.gnu.org/software/gsl/manual/

Chapter 4

Results

This chapter presents the solution to the parallel conjugate gradient method to solve Poisson's equation

$$\bigtriangledown^2 u = -f \tag{4.1}$$

with $u \equiv u(x, y)$ on the square region

$$ABCD, A = (-0.5, -2), B = (2, -2), C = (2, 0.5), D = (-0.5, 0.5), \tag{4.2}$$

where the source density s given by

$$f(x, y) = 4\cos(x + y)\sin(x - y) \tag{4.3}$$

The exact solution and boundary conditions are given by

$$u = \cos(x + y)\sin(x - y) \tag{4.4}$$

which results in the following graph

Two simulation runs were performed – one in serial mode and the other one in parallel mode using the Message Passing Interface (MPI) with 16 configured nodes. Both runs are configured with $\delta = 0.01$ which yields 251 grid points. Figure 4.2 presents both graphs with the gradient on the surface illustrating the error (Equation 4.5) in the conjugate gradient method compared to the analytical solution (Equation 4.4).

$$\epsilon = |u(x, y) - v_{i,j}| \tag{4.5}$$

The error distribution of both the serial and parallel execution on the surface of Figure 4.2 are very similar. So similar in fact, that it is not distinguishable with the bare eye. The root directory contains an awk script

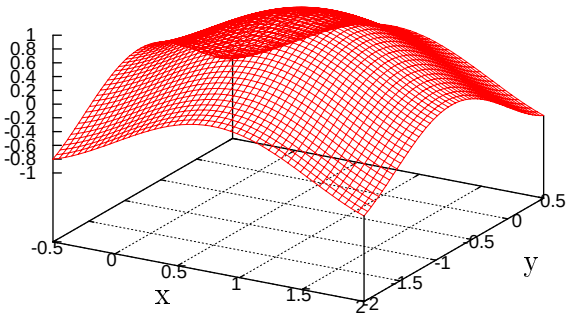

Figure 4.1: Exact Solution of Poisson's Equation

that streams through the error column in the output file and presents the maximum error. The difference is in the 15th decimal digit.

Surprisingly, the error distribution on the surface of both plots is not smooth. The reason for that is the relatively low default error threshold of $1e^{-6}$ which results in ca. 400 conjugate gradient iterations. Increasing the error threshold to $1e^{-12}$ will produce a much smoother error surface (see Figure 4.3), where the maximum error is in the centre of the plot (which is more what was expected in the first place).

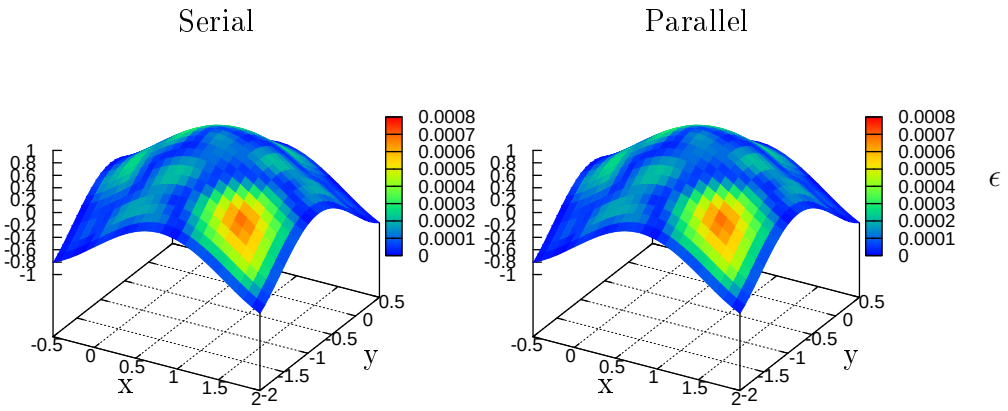

Figure 4.2: Approximate Solution of Poisson's Equation $\bigtriangledown^2 u = -f$

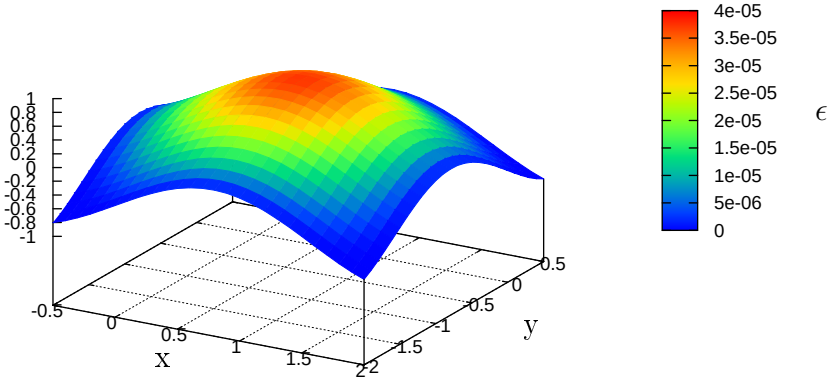

Figure 4.3: Serial Approximation of Poisson's Equation with an error threshold of $1e^{-12}$

Chapter 5

Problems

During the course of the implementation, some problems were encountered and dealt with. Some others were not tackled. The latter ones are:

- The parallel implementation does not take advantage of the symmetry of the matrix. That means that the matrix was set up as a 5-band general matrix and solved respectively.
- In some configurations floating point exceptions occur and manifest themselves as NaNs in the result.

The latter one is a major flaw that is quite difficult to pinpoint. In an early implementation of the solution, a similar error occurred in the serial version, because the conjugate iteration was performed until a maximum loop count of the dimension of the matrix was exceeded. This lead to floating point operations with very small numbers and was discovered using gdb to handle the *SIGFPE* signal. Knowing, the source of the floating point exception hinted at a more optimal conjugate gradient method implementation which did take the error threshold into account. This reduced the iteration count significantly and avoided floating operation of very small numbers. The conjugate gradient method would stop once this error threshold was exceeded.

However, debugging the parallel version proved too difficult to finally resolve this issue.