School of Computing, University of Leeds, 2013–2014

COMP3920: Parallel Scientific Computing

Coursework 2 – Modelling Heat Diffusion

Deadline: 10:00am, Friday 29th November 2013

Introduction

This coursework is designed to help you to understand (a) the behaviour of the approximation errors of scientific computing algorithms as the problem size is increased to improve accuracy, (b) the convergence behaviour of iterative algorithms for solving linear systems of equations and (c) the properties of different parallel partitioning strategies, investigating the resulting parallel performance as the problem size and the number of processors is increased. It is worth 20% of the overall grade for this module.

Two programs are provided, `heat_rows.c` and `heat_blocks.c`. Each one implements a different parallel strategy, respectively strip and block partitioning, for calculating the heat distribution in the interior of a metal plate when its boundary is held at a fixed temperature. The linear system of equations generated by the discrete form of this problem is solved using a Jacobi iteration. The derivation of this system and the program structure will be discussed in some detail during the lectures.

Compile and run the two programs to make sure that you know what the output should look like. Both codes are set up to produce exactly the same output: with the parameter values assigned in the codes you should (after 13544 iterations) get a single file called `heat_solution.dat` with $(N+1)^2 = 10201$ lines of three columns: row index, column index, solution value. The values in the final column (all close to 1.0 in this case) represent the temperatures at the nodes of the finite difference mesh. A very simple Matlab script, `draw_heat.m`, is also supplied which will plot the output.

Note that the strip partitioning implemented in `heat_rows.c` is very natural to use in C because the two-dimensional array stores each row in a contiguous block of memory (see the function `matrix` for details). Consequently, when sending a row of data to another processor, as required in this algorithm, only a pointer to the start of the row is required in addition to the length of the row. In contrast, partitioning the sample points into strips of *columns* would not be so straightforward: each processor would need to send to and receive from its two neighbours a column of values that are not stored in contiguous memory. Hence it is necessary to use buffers which are filled before the send and emptied after the receive has been completed. These intermediate buffers are also necessary for block partitioning, and implemented in `heat_blocks.c`.

As always, please ensure that you do not hinder anyone else's work when you use machines remotely and use `mpdallexit` when logging off.

Tasks

1. The code is currently set up to solve the problem where the temperature around the boundary of the whole domain is held at a value of 1.0. The exact solution to this problem gives a temperature of 1.0 throughout the domain. This is nice and simple but of limited use when testing the algorithm.

   Modify the file `heat_rows.c` so that the exact solution to the problem is instead given by $u = e^{3y} \cos 3x$ and the convergence tolerance for the iteration is 10^{-12}. Run this code with different values of N (*e.g.* 8, 16, 32, 64, 128, 256, ...) and construct a table which displays the error in the final solution (as output by the code) for each value of N. You will need to compile your code with the `-lm` option to include the mathematical functions. Draw a graph which plots $\log |\text{Error}|$ against $\log N$.

   Describe the behaviour of the error as N is increased. Comment on the behaviour you see in this graph and relate it to the behaviour of the error you see in the table.

   [11 marks]

2. Repeat the numerical experiments you carried out in Task 1, but using a convergence tolerance for the iteration of 10^{-5}. Construct another table displaying errors in the final solution and another graph which plots $\log |\text{Error}|$ against $\log N$. Compare your results with those obtained in Task 1. Comment on the similarities and differences you see in the behaviour of the error as N is increased. **[8 marks]**

3. Modify the *original file* `heat_rows.c`, in which $u = 1$ is specified as the exact solution, so that $N = 61$ (giving 60×60 unknown interior values to calculate) and it runs for a fixed number of iterations (`MaxIter = 8000`) instead of stopping when a specified convergence tolerance is reached.

   Investigate the scalability of this code on the workstations in DEC10 or ENIAC. You should look at 1, 2, 4, 8 and (when there are enough available machines) 16 processors and consider a variety of problem sizes (*e.g.* $N = 61, 121, 241, 481, 961, 1921, \ldots$). Present the runtimes you obtain for different values of N and different numbers of processors in a table. For larger values of N you should comment out the call to the function `write_file`, otherwise you will create some very large files in your filespace.

   In a single figure, plot separate graphs for each of the values of N tested which show the variation of the speedup against the number of processors used (*cf.* the graphs of speedup in the lecture slides). In a separate figure you should also plot graphs, for all of the problem sizes tested, of the efficiency against the number of processors used.

   Comment on the behaviour of the runtimes as the number of processors and the problem size are varied. You should briefly explain the behaviour you see, particularly if it doesn't match what you would expect from the theory. **[17 marks]**

4. Modify the program `heat_blocks.c` so that it also runs for a fixed number of iterations (`MaxIter = 8000`).

   As in Task 3, investigate the scalability of your modified code, using the same range of problem sizes, but this time running on 1, 4, 9 and 16 processors. Note that the code can only deal with cases where $\sqrt{p}$ is an integer and $N-1$ is a multiple of $\sqrt{p}$.

   Present the runtimes you obtain in a table which can be compared with the results of Task 3. Also plot graphs of speedup and efficiency, as described in Task 3.

   Compare this set of runtimes with those you obtained in Task 3. Based on your results, which of the two approaches to partitioning appears to scale better as the number of processors is increased? You should justify your answer. Comment on how your observations about the similarities and differences between the two sets of runtimes support (or contradict) the results obtained from the time complexity analysis carried out in the lectures. **[15 marks]**

The remainder of this coursework considers the effect of using different iterative algorithms for approximating the temperature distribution. All results from now on should be obtained using strip partitioning (based on `heat_rows.c`), an exact solution of $u = 1$, and $N = 241$.

5. For a convergence tolerance of 10^{-12} the Jacobi iteration converges in 220567 iterations. This is frustratingly slow, even in parallel.

   If the code is run on a single processor, the Jacobi iteration can be replaced by a Gauss-Seidel iteration with lexicographic ordering. Change the function `iteration` so that it carries out a lexicographic Gauss-Seidel iteration on each processor. You do not need to modify any other part of the code or to attempt to further improve the efficiency of the implementation. Run your modified code with the same parameters as the Jacobi iteration and write down the number of iterations it takes the new algorithm to converge with a tolerance of 10^{-12} on one processor.

   Write down the number of iterations it takes for the Jacobi and lexicographic Gauss-Seidel algorithms to converge with tolerances of 10^{-6}, 10^{-8} and 10^{-10} on one processor. Compare these two sets of numbers and describe any relationship you see between the convergence rates of the two algorithms.

   It should be possible to run this code on multiple processors. Describe what your implementation will actually do in this scenario, *e.g.* when run on 2 processors. You should consider whether each node is updated using information from the previous iteration (as in Jacobi) or the current iteration (as in Gauss-Seidel). Run your code on 2, 3 and 4 processors with a convergence tolerance of 10^{-12} and write down the number of iterations the algorithm takes to converge in each case. Compare these numbers with the number of iterations taken on one processor, describing any differences you see.

   [13 marks]

6. For this problem it is more typical to implement a red-black ordering for the Gauss-Seidel iteration (as described in the lectures). Modify the original file `heat_rows.c` so that it carries out a red-black Gauss-Seidel iteration instead of a Jacobi iteration. Note that it will be necessary to split each iteration into two stages (an update for the red nodes followed by an update for the black nodes) with two lots of communication per iteration (one immediately before the red updates and one immediately before the black updates). You should also try to make your implementation reasonably memory-efficient.

   When testing your code you should choose the number of processors p so that it makes $(N-1)/p$ an even integer. Write down the number of iterations it takes for the red-black Gauss-Seidel algorithm to converge with tolerances of 10^{-6}, 10^{-8}, 10^{-10} and 10^{-12}, on 1 and 4 processors. Compare these numbers with the numbers of iterations taken by the lexicographic Gauss-Seidel algorithm, describing any similarities and differences you see.

   Run your code with values of p for which $(N-1)/p$ is not an even integer. Describe any differences you see in your results and explain why you see them. **[18 marks]**

7. The Successive Over-Relaxation (SOR) iteration can be written as

$$(u_{i,j}^{SOR})^{(k+1)} \;=\; (1-\omega)(u_{i,j})^{(k)} + \omega(u_{i,j}^{GS})^{(k+1)}\,,$$

   in which ω is a "relaxation" parameter, typically chosen to have a value between 0 and 2, and $(u_{i,j}^{GS})^{(k+1)}$ is the updated value according to the Gauss-Seidel iteration you have implemented in Task 6, *i.e.* taking $\omega = 1$ should reproduce the Gauss-Seidel iteration. Modify the iteration in your code with the red-black ordering, produced in Task 6, so that it carries out the SOR iteration.

   Varying the value of ω will vary the rate at which the iteration converges. How many iterations does it take your code to converge for a tolerance of 10^{-12} when $\omega = 1.5$? Describe how the rate of convergence changes as ω is varied between 0 and 2.5. Find, to two decimal places, the optimal value of ω, *i.e.* the one that gives the most rapid convergence for a tolerance of 10^{-12}. Write down the number of iterations it takes to reach convergence for this value of ω. **[10 marks]**

8. Compare the runtimes on a single processor for the Jacobi, red-black Gauss-Seidel and red-black SOR with optimal ω. Discuss the relative speeds of the three algorithms, including a brief discussion of the relative runtimes per iteration.

   Compare the runtimes of the three algorithms on multiple processors (using strip partitioning). Does parallelisation change significantly their relative speeds?

   Finally, consider all of the algorithms you have implemented. For a convergence tolerance of 10^{-12}, what is the shortest runtime you can achieve? Which algorithm did you use to achieve this and how many processors (on how many hosts) did you use to obtain this runtime? How many times faster have you made your solver by doing this coursework, compared to using the Jacobi iteration on a single processor? **[8 marks]**

Note that, if you are unable to get the red-black ordering to work, you can still obtain some marks for Tasks 7 and 8 by doing the comparison for lexicographic Gauss-Seidel and SOR implementations on a single processor and submitting those results.

Total: 100 marks

Deliverables

Please submit your coursework in the form of a short report, via the postbox in the SSO. The report should include the following.

1. Tables of errors for Tasks 1 and 2, tables of iteration counts for Tasks 5, 6 and 7 and tables of runtimes for Tasks 3, 4 and 8.
2. Graphs illustrating the error behaviour for Tasks 1 and 2, and the variation of speedup and efficiency requested for Tasks 3 and 4.
3. A brief description of the modifications you have made to the original code to produce your solutions to Tasks 1, 3 and 5, and the modifications you made to your Gauss-Seidel code to produce your SOR iteration for Task 7. You do *not* need to submit your full code for these tasks.
4. A statement of the number of processors you have used on each host in your MPD ring when you have been obtaining the timings shown in the tables for Tasks 3 and 4.
5. Discussions of the results obtained in all Tasks.
6. Your full code for Task 6.