TMA4280: Introduction to Supercomputing

Problem set 6

Spring 2014

NOTE THAT:

- this exercise is mandatory;
- you can work on this problem alone or with up to two other students;
- a report describing the solution should be written (max 12 pages);
- please write your *name(s)* on the report;
- please include the source code as separate files rather than inline in the report (save the forest!)
- please make sure that you have answered all the questions;
- the due date is Friday, April 4, 2014;
- the report will count 30% towards the final grade.

©Einar M. Rønquist
Department of Mathematical Sciences
NTNU, N-7491 Trondheim, Norway

Consider the solution of the two-dimensional Poisson problem

$$-\nabla^2 u = f \qquad \text{in } \Omega = (0,1) \times (0,1) \tag{1}$$

$$u = 0 \qquad \text{on } \partial\Omega\,. \tag{2}$$

Here, f is a given right hand side and u is the solution.

Assume that we discretize this problem on a regular finite difference grid with $(n+1)$ points in each spatial direction. Hence, the mesh spacing $h = 1/n$. We use the standard 5-point stencil to discretize the Laplace operator.

In order to solve the system of algebraic equations, we apply the diagonalization methods discussed in class. In particular, we apply the Discrete Sine Transform (DST) in order to obtain a solution of this problem in $\mathcal{O}(n^2 \log n)$ floating point operations.

a) Write a program to solve the Poisson problem on p processors using the algorithms described above. You can choose $f = 1$ for the initial development.

   Use the provided routines to compute the DST and its inverse based on FFT. See Appendix A for details on compiling, linking and running the *serial* version of the Poisson solver. Use the MPI communication library in order to develop your program for distributed memory parallel computers, and OpenMP to utilize t threads in each MPI process. See Appendix C for comments on the transpose operation.

b) Run your parallel program on `kongull`, the cluster at NTNU.
   Obtain detailed timing results for different combinations of $n = 2^k$ and p, t.
   In particular, you should demonstrate that your program functions correctly for (selected) values of p in the range $1 \leq p \leq 36$.
   Follow the procedure described in Appendix B.

   Note 1. Having available a program that runs extremely fast is of no use unless the final answer is correct.

   Note 2. It is sufficient (and strongly recommended!) that you test the correctness of your program on a *small* problem size before solving larger problems.

c) Run your program with $n = 16384$ and $p{*}t = 36$, i.e., solve the Poisson problem with about 270 million grid points on 36 processors. Does the hybrid model work better, worse or equivalent compared to the pure distributed memory model? Explain your observations.

d) Report the speedup, S_p, as well as the parallel efficiency, η_p, for different values of n and p. The parallel efficiency on p processors is defined as $\eta_p = \frac{S_p}{p}$.
How do your timing results scale with the problem size n^2 for a fixed number of processors? Is the scaling as expected? Do you see an improved speedup if you increase the problem size?

Note 4: You need to submit your individual jobs to a batch queue. This is very important on supercomputer systems because this is the only way to ensure consistent and reliable timing results.

e) Modify the given data f to be a function of your own choice. As an example, you could choose f to be a smooth function like $f(x, y) = e^x \cdot \sin(2\pi x) \cdot \sin(\pi y)$. Another example is to let f represent 2 point sources, e.g., $f = 0$ in the whole domain except at 2 chosen (grid)points (x_1, y_1) and (x_2, y_2) inside the domain where $f(x_1, y_1) = 1$ and $f(x_2, y_2) = -1$.

Run your program with the new right hand side f for a particular n and a particular p. Do you have to modify anything related to the parallel implementation when you change f, i.e., when you solve a different Poisson problem?

f) Discuss how you would modify the numerical algorithm to deal with the case where $u \neq 0$ on $\partial\Omega$, i.e., in the case of non-homogeneous Dirichlet boundary conditions. You do not have to implement this.

g) In this exercise (as well as in the lectures), we have assumed that the domain is the unit square, i.e., $\Omega = (0, 1) \times (0, 1)$. Discuss how you would modify the Poisson solver based on diagonalization techniques if the domain Ω instead is a rectangle with sides L_x and L_y, i.e., $\Omega = (0, L_x) \times (0, L_y)$. You can still assume a regular finite difference grid with $(n + 1)$ points in each spatial direction. Does this extension of the original method change anything in terms of your parallel implementation? You do not have to implement this case.

General comments:

It will in general be emphasized that the program is well parallelized and load balanced. It will further be of importance that the program is well organized. It is allowed to use more memory than the minimum required, if it speeds up the program, but this should be done with care.

A report describing results of parallelization of a scientific problem typically contains:

- a description of the mathematical/numerical problem,
- a discussion of possible solution strategies,
- a short explanation of the finished program,
- description of the parallel computer on which the numerical results were obtained, which compiler and compiler options you used to compile the program, as well as any other relevant information such as sublibraries utilized,
- numerical results (preferably plot(s) of the result(s)),
- analysis (both theoretical and experimental) of the performance of the algorithm and implementation (time usage, speedup and efficiency as a function of problem dimension and number of processors),
- a discussion of bottlenecks and possible optimization, and
- possibly a listing of the relevant parts of the source code in the appendix.

APPENDIX A. Compiling and linking The serial code ships with a CMake based build system for portability and simplicity. You can generate a build system using

```
module load intel/compilers/11.1.059
module load cmake
CC=icc FC=ifort cmake . -DCMAKE_BUILD_TYPE=Release
```

assuming you are located in the folder with the shipped CMakeLists.txt. This will, on success, generate a Makefile, and you can then build and run the program using

```
make
./poisson 128
```

The first statement builds the program, while the second statement runs the code on a single processor with $n = 128$. Note that n needs to be a power of 2 in order to use the discrete sine transform provided here. Note that by default CMake will not show you the compiler commands. You can see these by doing

```
make VERBOSE=1
```

APPENDIX B. Verification of correctness

One way to verify that the code works correctly is to do a *convergence test*. Following this approach, we first assume an exact solution to our Poisson problem. For example, we can assume that the exact solution is given as

$$u(x, y) = \sin(\pi x) \cdot \sin(2\pi y). \tag{3}$$

This solution satisfies the homogeneous boundary conditions $u = 0$ on $\partial\Omega$.

Next, we evaluate $-\nabla^2 u$ which should be equal to f, i.e.,

$$f(x, y) = -\nabla^2 u = 5\pi^2 \cdot \sin(\pi x) \cdot \sin(2\pi y). \tag{4}$$

Assuming the given data $f(x, y)$ as specified in (4), we now solve the Poisson problem numerically. We then compare the numerically computed solution with the exact solution at the finite difference points. The maximum pointwise error should then decrease to zero as $\mathcal{O}(h^2)$, i.e., if we increase n with a factor of two (or equivalently, decrease h with a factor of two), the error should decrease with a factor of four. Remember that finding the maximum pointwise error using multiple processors will require communication!

APPENDIX C. Comments on the transpose operation

The implementation of the transpose operation is trivial in a serial context. In a parallel context, using a distributed memory programming model, message passing has to be used. In this case, the transpose of a matrix will involve all-to-all communication.

Part of the solution algorithm in this exercise requires a parallel implementation of the transpose operation. We have given a matrix of a certain dimension. We can distribute the matrix such that each processor is responsible for a certain number of columns (or rows).

The most convenient way to implement the transpose operation is to use the MPI library function MPI_ALLTOALLV in FORTRAN (or MPI_Alltoallv in C); see Figure 1. For a detailed description of this function, see reference 1, page 174, or simply input the function name into an internet search engine.

Reference

1. Snir, Otto, Huss-Lederman, Walker, and Dongarra,
MPI: The Complete Reference, The MIT Press, Cambridge, MA.

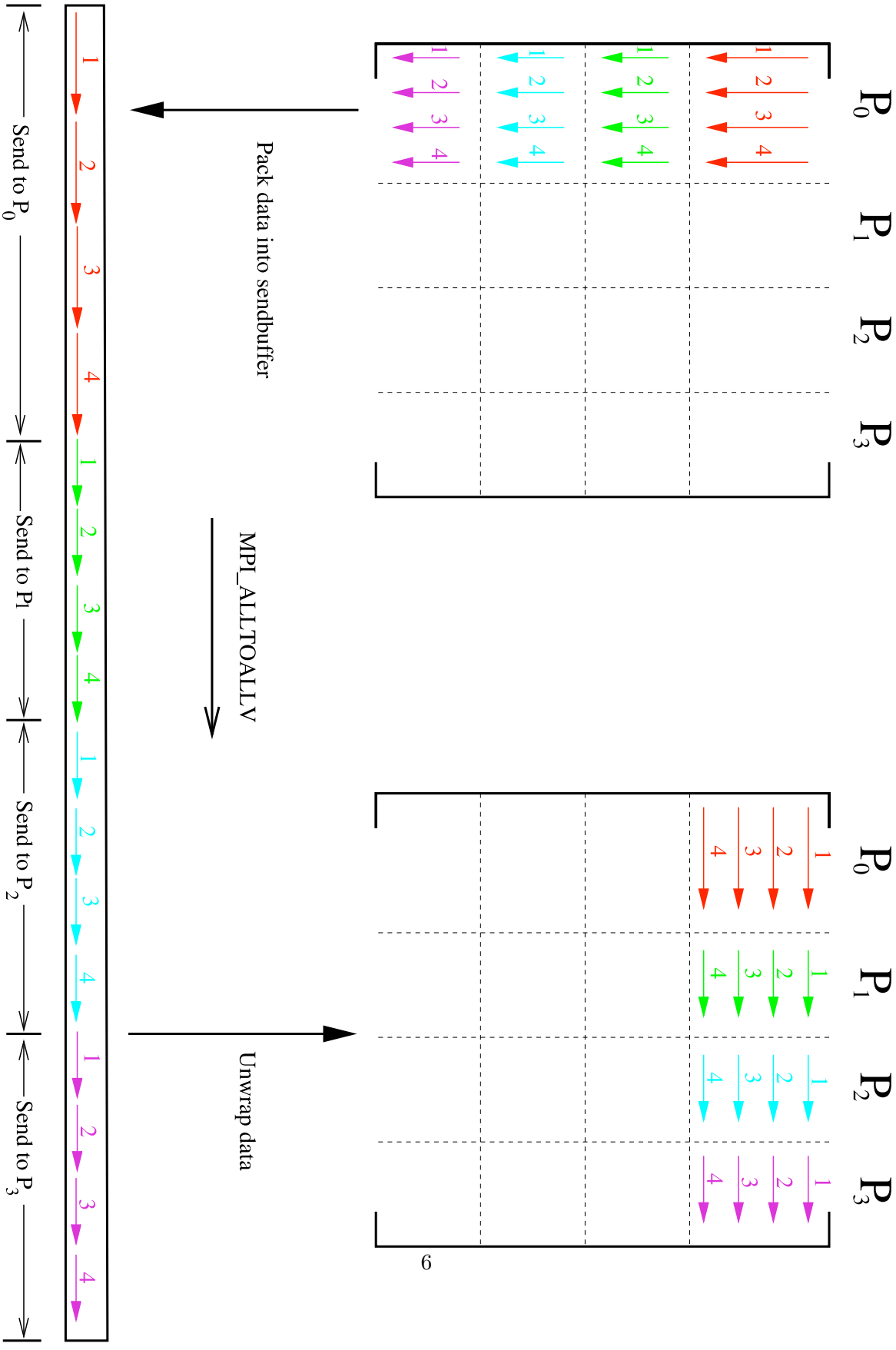

Figure 1: The transpose operation using message passing: the packing and unpacking of data. The figure is due to Bjarte Hægland.