Technische Universität München
Institut für Informatik
Prof. Dr. Thomas Huckle
Matous Sedlacek

WT 2011/2012

10.02.2012

Parallel Numerics

Exam

A handwritten sheet of paper (size A4, front and back page) may be used during the exam as mnemonic as well as the MPI operation reference distributed during the tutorials. No other material is allowed. The exam is to be solved within 90 minutes, and the answers are to be written in German or English. Please try to answer all parts of the questions precisely and briefly. For passing the exam you will need 17 out of 40 credits. The exam consists of 7 problems on 4 pages.

Problem 1: Miscellaneous **(≈ 8 credits)**

a) Describe the SPMD paradigm with a comparison to the SIMD and MIMD architecture classifications according to Michael J. Flynn. Refer also to synchronization of operations/instructions.
SPMD (single program, multiple data). All pe's execute same program, but each operates on different portion of problem data. Easier to program than true MIMD, but more flexible than SIMD. Although most parallel computers today are MIMD architecturally, they are usually programmed in SPMD style. SPMD works asynchronously. 1 credit for correct description. 1 credit for asynchronous operations/instructions similar to MIMD.

b) Describe briefly the difference between OpenMP and the Message Passing Interface (MPI)? Give 1 advantage and 1 disadvantage for each of them.
OpenMP is an API designed for shared memory systems. MPI is a standard for message passing, applicable to distributed memory systems.
Advantages of OpenMP: *easier to program and debug than MPI, directives can be added incrementally - gradual parallelization, can still run the program as a serial code, serial code statements usually don't need modification, code is easier to understand and maybe more easily maintained*
Disadvantages of OpenMP: *can only be run on shared memory computers, requires a compiler that supports OpenMP, mostly used for loop parallelization*
Advantages of MPI: *runs on either shared or distributed memory architectures, can be used on a wider range of problems than OpenMP, each process has its own local variables, distributed memory computers are less expensive than large shared memory computers*
Disadvantages of MPI: *requires more programming changes to go from serial to parallel version, can be harder to debug, performance is limited by the communcation network between the nodes, MPI cannot take advantage of multicore subsystems, i.e., cannot distinguish between subsystems in a hybrid environment*
1 credit for describing difference. 1 credit for correct advantages and disadvantages.

c) Given is the following C code fragment using MPI. Describe in detail wheather the code deadlocks or runs to completion.

```
...
int myrank;MPI_Request request;MPI_Status status;double a[8],b[8];
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
if( myrank == 0 ) {
```

```
    MPI_Irecv( b,8,MPI_DOUBLE,1,3,MPI_COMM_WORLD,&request );
    MPI_Send( a,8,MPI_DOUBLE,1,4,MPI_COMM_WORLD );
    MPI_Wait( &request,&status );
}
else if( myrank == 1 ) {
    MPI_Irecv( b,8,MPI_DOUBLE,0,4,MPI_COMM_WORLD,&request );
    MPI_Send( a,8,MPI_DOUBLE,0,3,MPI_COMM_WORLD );
    MPI_Wait( &request,&status );
}
MPI_Finalize();
...
```

Process 0 attempts to exchange messages with process 1. Each process begins by posting a receive for a message from the other. Then, each process blocks on a send. Finally, each process waits for its previously posted receive to complete. Each process completes its send because the other process has posted a matching receive. Each process completes its receive because the other process sends a message that matches. Barring system failure, the program runs to completion. (2 credits)

d) The solution of a linear system $Ax = b$ with a matrix $A \in \mathbb{R}^{n\times n}$ and $b \in \mathbb{R}^n$ can be obtained by using the SOR method. Here, the solution x is computed iteratively as a series $x^{(k)}$:

$$x_i^{(k+1)} = \frac{\omega}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right) + (1-\omega) x_i^{(k)} \qquad (1)$$

for $i = 1, \ldots, n,\ k = 0, 1, \ldots$

For $\omega = 1$ SOR coincides with the Gauss-Seidel method. Change the given SOR formula (1) such that it describes the Jacobi method!

With $\omega = 1$ the Jacobi method is

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right)$$

for $i = 1, \ldots, n,\ k = 0, 1, \ldots$

1 credit for $\omega = 1$, 1 credit for correct $x_j^{(k)}$. In case for given damped Jacobi, 1 credit is given.

Problem 2: Parallel evaluation of arithmetic expressions **(≈ 5 credits)**
Consider the arithmetic expression

$$(b_1 \cdot b_2 \cdot b_3 + b_4 + b_5) \cdot b_6 + b_7 - b_8, \quad b_i \in \mathbb{R}. \qquad (2)$$

a) How many time steps are necessary to evaluate (2) with its basic operations (addition, multiplication, subtraction, ...) on $p = 1$, $p = 2$, and $p = 3$ processors? Do **not!** modify the expression but use exactly the form as presented in (2). Show **and** explain the optimal evaluation scheme to proof your answer.

$p = 1$: With no parallelism there are 7 basic operations and 7 time steps necessary. (1 credit)

$p = 2$: Using two processors a possible evaluation scheme is

Time step	$p = 1$	$p = 2$
1	$b'_1 = b_1 \cdot b_2$	$b'_7 = b_7 - b_8$
2	$b''_1 = b'_1 \cdot b_3$	$b'_4 = b_4 + b_5$
3	$b'''_1 = b''_1 + b'_4$	
4	$b''''_1 = b'''_1 \cdot b_6$	
5	$b'''''_1 = b''''_1 + b'_7$	

Hence, 5 time steps are required. (1 credit)
$p = 3$: Using three processors a possible evaluation scheme is

Time step	$p = 1$	$p = 2$	$p = 3$
1	$b'_1 = b_1 \cdot b_2$	$b'_4 = b_4 + b_5$	$b'_7 = b_7 - b_8$
2	$b''_1 = b'_1 \cdot b_3$		
3	$b'''_1 = b''_1 + b'_4$		
4	$b''''_1 = b'''_1 \cdot b_6$		
5	$b'''''_1 = b''''_1 + b'_7$		

Hence, 5 time steps are required. (1 credit)
Only half credits without evaluation schemes.

b) Is it possible to reduce the number of time steps by modifying the arithmetic expression given in (2) when using $p = 3$ processors? Proof your answer!
The expression can be expanded to

$$b_1 \cdot b_2 \cdot b_3 \cdot b_6 + b_4 \cdot b_6 + b_5 \cdot b_6 + b_7 - b_8.$$

$p = 3$: Using three processors a possible evaluation scheme is

Time step	$p = 1$	$p = 2$	$p = 3$
1	$b'_1 = b_1 \cdot b_2$	$b'_4 = b_4 \cdot b_6$	$b'_5 = b_5 \cdot b_6$
2	$b''_1 = b'_1 \cdot b_3$	$b''_4 = b'_4 + b_7$	$b''_5 = b'_5 - b_8$
3	$b'''_1 = b''_1 \cdot b_6$	$b'''_4 = b''_4 + b''_5$	
4	$b''''_1 = b'''_1 + b'''_4$		

Hence, 4 time steps are required. This is a reduction of 1 time step in contrast to part a). (2 credits)

Problem 3: Storage costs for Matrix-Matrix multiplication (≈ 4 credits)
Consider the matrix-matrix product $C = A \cdot B$ where A, B, $C \in \mathbb{R}^{n \times n}$. To compute C, a 2D parallel algorithm can be used on a $\sqrt{p} \times \sqrt{p}$ processor grid, where p is the number of processors used. Each processor contains one block of each matrix. You may assume that p is a perfect square and that $\sqrt{p}$ divides n.
Example: for $p = 9$ the submatrices are distributed according to

$$\left(\begin{array}{c|c|c} A_{11}, B_{11}, C_{11} & A_{12}, B_{12}, C_{12} & A_{13}, B_{13}, C_{13} \\ \hline A_{21}, B_{21}, C_{21} & A_{22}, B_{22}, C_{22} & A_{23}, B_{23}, C_{23} \\ \hline A_{31}, B_{31}, C_{31} & A_{32}, B_{32}, C_{32} & A_{33}, B_{33}, C_{33} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} p_1 & p_2 & p_3 \\ \hline p_4 & p_5 & p_6 \\ \hline p_7 & p_8 & p_9 \end{array}\right).$$

What is the total storage per processor required for each of the following algorithms in a) and b)? You should include storage for the matrices and the message buffers necessary for the communication. Give a short explanation and the formula dependent on p and n. *Hint:* for instace, each processor will require $\frac{n^2}{p}$ storage for its A block.

a) Straightforward algorithm using broadcasts along both rows and columns of processors (i.e., p_1 will broadcast its blocks to p_2, p_3, p_4, and p_7)?
Each processor must store its own portions of A, B, and C, each of which is of size $\frac{n^2}{p}$. In addition, each processor requires enough buffer space to store the portions of A belonging to the other $\sqrt{p}-1$ processors in its row and column. Summing all these, the total storage required per processor is

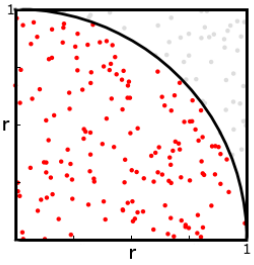
$$3\frac{n^2}{p} + 2(\sqrt{p}-1)\frac{n^2}{p} = \frac{n^2(3+2(\sqrt{p}-1))}{p} = \frac{n^2(2\sqrt{p}+1)}{p}.$$

1 credit formula, 1 credit description.

b) Cannon's algorithm?
In addition to storage for its portion of input and result matrices, i.e., $3\frac{n^2}{p}$, each processor requires two message buffers, each of size $\frac{n^2}{p}$, for circulating in ring fashion along both dimensions of the processor grid. Thus, the total storage required per processor is $5\frac{n^2}{p}$. 1 credit formula, 1 credit description.

Problem 4: Calculation of π with Monte Carlo and MPI (≈ 6 credits)

Consider a quarter circle with a radius $r = 1$ which is within a square in the first quadrant, with corners at $(0,0)$, $(0,1)$, $(1,0)$, and $(1,1)$. See also Figure to the right. By choosing random points within the square (analogy of throwing darts 'randomly' at a dart board), we are able to calculate whether or not each point is within the quarter circle or not. The ratio of the area of the quarter circle to the area of the square is given by



$$\frac{\text{Area of quarter circle}}{\text{Area of square}} = \frac{\frac{1}{4}r^2\pi}{r^2} \overset{r=1}{=} \frac{\pi}{4}. \tag{3}$$

(3) can be estimated by randomly choosing points (x, y) within the square and keeping track of the number of those points that fall within the quarter circle (those where $x^2 + y^2 <= 1$) versus the total number of points tried. Hence, π can be computed approximately by

$$\pi \approx \frac{4 \cdot \text{number of darts inside quarter circle}}{\text{total number of darts}} =: \frac{n_{\text{count}}}{n_{\text{darts}}} \quad \text{for} \quad n_{\text{darts}} \gg 1. \tag{4}$$

Given is the following C/MPI code fragment which stores the number of points which fall into the quarter circle in the variable `my_count`. According to the following questions, your code fragment should fit into the line 22.

```
7:  ...
8:  int my_id, nprocs, my_count = 0, count = 0, tag = 1;
9   int ndarts = 100000, master = 0;
10:  double x,y,z,pi;
11: MPI_Status status;
12: MPI_Init(&argc,&argv);
13: MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
14: MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
15:
```

```
16: for (int i = 0; i < ndarts; i++) {
17:     x = get_random_nmb();
18:     y = get_random_nmb();
19:     z = x*x + y*y;
20:     if (z <= 1) my_count++;
21: }
22:
23: // fill in your code here ...
24:
25: MPI_Finalize();
26: ...
```

a) The master process (rank 0) sums up the local `my_count`'s from all other processors (slaves) and computes π according to (4). Use **only!** the standard `MPI_Send()` and `MPI_Recv()` commands for the interprocess communication. Your code fragment should fit into line 22 of the above source code.

```
23:  if (my_id == 0) { // master gathers results from others
24:      count = my_count;
25:      for (int proc = 1; proc < nprocs; proc++) {
26:          MPI_Recv(&my_count,1,MPI_INT,proc,tag,MPI_COMM_WORLD,&status);
27:          count += my_count;
28:      }
29:      pi = 4 * ((double)count / (ndarts * nprocs));
30:  }
31:  else { // slaves send results to master
32:      MPI_Send(&my_count,1,MPI_INT,master,tag,MPI_COMM_WORLD);
33:  }
```

1 credit for if-else statement. 1 credit for correct MPI_Send() and MPI_Recv() usage. 1 credit for correct π computation.

b) Now every processor computes its own approximation to π. The master process sums up all these local results and computes the average value of π. Use an appropriate MPI function which enables to 'collect' all the local results. Write your code fragment which fits in line 22 of the above source code.

```
23:  double pi_current = 4.0 * (double)my_count / (double)ndarts;
24:  // Reduce local pi_current values across all workers on master
25:  double pi_sum;
26:  MPI_Reduce(&pi_current, &pi_sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
27:  if (my_id == 0) { // if master, calculate average
28:       pi = pi_sum / nprocs;
29:  }
```

1 credit for MPI_Reduce(). 1 credit for correct local π computation. 1 credit for correct average computation.

Problem 5: Sparse Approximate Inverses **(≈ 5 credits)**

The SAI algorithm computes a sparse matrix $M \in \mathbb{R}^{n\times n}$ which approximates $A \in \mathbb{R}^{n\times n}$ inversely, i.e., $M \approx A^{-1}$. It uses the ansatz

$$\min_{M\in\mathcal{P}} \|AM - I\|_F^2, \tag{5}$$

where $I \in \mathbb{R}^{n\times n}$ is the identity matrix and which requires an a priori chosen sparsity pattern $\mathcal{P}$ for M, i.e., one has to define the positions of the nonzeros in M a priori (by a given boolean matrix or sparsity set).

a) What is the inherent advantage of using the Frobeniusnorm approach according to (5)? Emphasize your answer with a formula corresponding to (5).
By using the Frobeniusnorm approach it is possible to decouple the computation of the preconditioner columns. Each column can be computed completely in parallel. The following equation is satisfied:

$$\min_{M\in\mathcal{P}} \|AM - I\|_F^2 = \sum_{k=1}^{n} \min_{M\in\mathcal{P}} \|AM_k - e_k\|_2^2,$$

where e_k denotes the kth unit vector and M_k the kth column of the preconditioner M. 1 credit description. 1 credit formula.

b) Compute the approximate inverse for the following system and preconditioner pattern:

$$\min_{M\in\mathcal{P}} \left\| \begin{pmatrix} 4 & 0 & -2 \\ 0 & 2 & 0 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} - \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \right\|_F^2 .$$

We compute the 3 columns of M idependently.

- *For M_1 we obtain the reduced system*

$$\begin{pmatrix} 4 & -2 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} m_{11} \\ m_{13} \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

  with the solution $m_{11} = \frac{1}{6}$ and $m_{13} = -\frac{1}{6}$.
- *For M_2 we obtain the reduced system*

$$2 \cdot m_{22} = 1$$

  with the solution $m_{22} = \frac{1}{2}$.
- *For M_3 we obtain the reduced system*

$$\begin{pmatrix} 4 & -2 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} m_{13} \\ m_{33} \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

  with the solution $m_{13} = \frac{1}{3}$ and $m_{33} = \frac{2}{3}$.

Hence, the approximate inverse is

$$M = \begin{pmatrix} \frac{1}{6} & 0 & \frac{1}{3} \\ 0 & \frac{1}{2} & 0 \\ -\frac{1}{6} & 0 & \frac{2}{3} \end{pmatrix}$$

1 credit for each column of M.

Problem 6: LU decomposition **(≈ 6 credits)**

For matrices A we want to compute an LU decomposition on a parallel computer. The algorithm uses a block decomposition of the following form

$$\begin{pmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & * \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & * \\ 0 & U_{22} & * \\ 0 & 0 & * \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix}. \tag{6}$$

and is based on two subproblems. The computation is to be performed column-wise for L and U.

a) Give the formulae and a short description to compute the first columns of the LU decomposition in (6).
A first LU decomposition of the small block $A_{11} = L_{11}U_{11}$ has to be performed. With U_{11} we can solve the resulting equations $L_{21}U_{11} = A_{21}$ and $L_{31}U_{11} = A_{31}$. 2 credits.

b) Based on results for the first columns in a), give the formulae and a short description for the computation of the second columns of the LU decomposition in (6).
Solving $L_{11}U_{12} = A_{12}$ for U_{12} we can compute $L_{22}U_{22} = A_{22} - L_{21}U_{12}$ via LU decomposition to obtain U_{22}. Using this, we finally solve $L_{32}U_{22} = A_{23} - L_{31}U_{12}$. 2 credits.

c) Describe the approach in general how to compute the LU decomposition.
After the second step b), we merge the already computed parts from the second column of L and U into the first column of L and U. Furthermore, we split the until now ignored parts from the $*$-columns into new coluns in L and U. As a results we can again start solving with a) for (6). Using this alternating procedure we finally obtain the complete LU decomposition of A. 2 credits.

Problem 7: Graphs and colouring **(≈ 6 credits)**
Given is the symmetric update

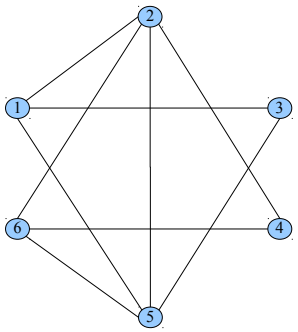
$$f(x) = \begin{pmatrix} f_1(x) \\ f_2(x) \\ f_3(x) \\ f_4(x) \\ f_5(x) \\ f_6(x) \end{pmatrix} = Ax = \frac{1}{5} \begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 \end{pmatrix} x.$$

where

$$x^{(k+1)} = f(x^{(k)}) \tag{7}$$

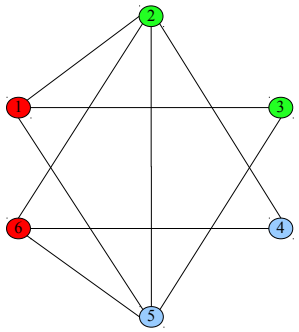
reflects the dependency between the update $x^{(k+1)}$ and the old solution $x^{(k)}$ in an iterative (e.g., Gauss-Seidel) algorithm. $A \in \mathbb{R}^{6\times 6}$ and $x \in \mathbb{R}^6$. Consider to overwrite the vector $x^{(k)}$ immediately.

a) Give the corresponding graph of f which describes the data dependency in the update (7), i.e. give $G(f)$.



1 credit for correct graph.

b) Give and explain a minimum colouring of $G(f)$ such that the update can be performed in parallel. How many steps have to be performed in parallel for the update?



There have to be performed 3 steps in parallel. 1 credit for correct graph. 1 credit for correct explanation.

c) Describe a renumbering of $G(f)$ according to the minimum coloring of $G(f)$ where the vertices of one color are numbered consecutively. Apply this permutation to rows and columns of A and give the resulting permuted matrix $\tilde{A}$.
Applying the full ordering

$$\begin{aligned} 1 &\to 1 \\ 2 &\to 3 \\ 3 &\to 4 \\ 4 &\to 5 \\ 5 &\to 6 \\ 6 &\to 2 \end{aligned}$$

to rows and columns of A we receive

$$\tilde{A} = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} D & * & * \\ * & D & * \\ * & * & D \end{pmatrix}, \quad \textit{with} \quad D = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

2 credits for correct permutation.

d) What is the advantage of this reordering when using Gauss-Seidel? How many components x_j can be updated simultaneously?
Considering the diagonal blocks D (or just the upper left diagonal block for the 2 elements) it is possible to perform the update between the corresponding components of x in parallel. Two components in x can be updated simultaneously. 1 credits for correct explanation how to read out the parallelism of $\tilde{A}$ and how many components can be updated simultaneously.