Modelling and implementation of algorithms in applied mathematics using MPI

Lecture 2: Poisson Problem and Finite Differences, First Steps in MPI

G. Rapin

Brazil
March 2011

Outline

1 Poisson Problem and Finite Differences

2 First Steps in MPI

Outline

1 Poisson Problem and Finite Differences

2 First Steps in MPI

Poisson Problem

- Let $\Omega \subset \mathbb{R}^d$ be a bounded domain.
- The *Laplace-Operator* is given by

$$\triangle u := \sum_{i=1}^{d} \frac{\partial^2 u}{\partial x_i^2}$$

  for a function $u : \Omega \to \mathbb{R}$.
- The *Poisson Problem* is defined as follows: Find a function $u \in C^2(\Omega) \cap C(\overline{\Omega})$ satisfying

$$\left\{ \begin{array}{rcll} -\triangle u & = & f & \text{in } \Omega \\ u & = & 0 & \text{on } \partial\Omega \end{array} \right. \tag{1}$$

- Applications in electrostatics, mechanical engineering and theoretical physics

One-dimensional Poisson Problem in $(0,1)$

Find a function

$$u : [0,1] \quad \to \quad \mathbb{R},$$

such that

$$\begin{aligned} -u''(x) &= e^x, \quad x \in (0,1) \\ u(0) &= u(1) = 0 \end{aligned}$$

Problem: In general there is no analytical solution.

Goal: Approximation of the solution.

Finite Differences

- Discretisation: $0 = x_0 < \cdots < x_n = 1$ with $x_i = \frac{i}{n}$
- Differential quotient:

$$u''(x_i) \sim \frac{u(x_{i-1}) - 2u(x_i) + u(x_{i+1})}{h^2}, \quad h := \frac{1}{n}$$

- Inserting of $-u''(x) = e^x$ yields

$$-u(x_{i-1}) + 2u(x_i) - u(x_{i+1}) = h^2 e^{x_i}, \quad i = 1, \ldots, n-1$$

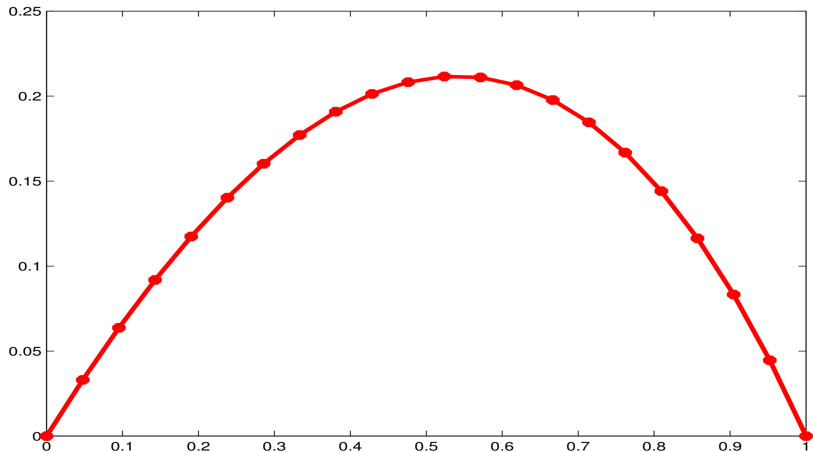
- Boundary Conditions $\Rightarrow$ $u(x_0) = u(x_n) = 0$.
- $\Rightarrow$ linear system for $u(x_1), \ldots, u(x_{n-1})$.

Discrete Problem

- Set $z = (z_1, \dots, z_{n-1})^t = (u(x_1), \dots, u(x_{n-1}))^t$.
- Solve the linear system $Az = F$ with

$$A := \begin{pmatrix} 2 & -1 & & & 0 \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ 0 & & & -1 & 2 \end{pmatrix}, \quad F := h^2 \begin{pmatrix} e^{\frac{1}{n}} \\ \vdots \\ e^{\frac{n-1}{n}} \end{pmatrix}.$$

Solution for $n = 21$



Two-dimensional Poisson Problem in $\Omega = (0,1)^2$

- For simplicity we consider $\Omega = (0,1) \times (0,1)$.
- Define a grid on Ω using the grid size $h = \frac{1}{N}$, $N \in \mathbb{N}$.

Set of grid points

$$Z_h := \{(x,y) \in \overline{\Omega} \mid x = z_1 h,\ y = z_2 h \text{ with } z_1, z_2 \in \mathbb{Z}\}.$$

- Let $\omega_h := Z_h \cap \Omega$ be the interior points.
- Set of points on the boundary are defined by $\gamma_h := Z_h \cap \partial\Omega$.

Discretisation of the 2D-Poisson Problem

- Let $\mathcal{O}$ be the *Landau symbol*. $g = \mathcal{O}(h^k)$ means $\lim_{h\to 0} \sup \frac{|g(h)|}{|h^k|} < \infty$.
- Differential quotient w.r.t. x

$$\frac{u(x-h,y) - 2u(x,y) + u(x+h,y)}{h^2} = \frac{\partial^2 u}{\partial x^2}(x,y) + \mathcal{O}(h^2).$$

- Differential quotient w.r.t. y

$$\frac{u(x,y-h) - 2u(x,y) + u(x,y+h)}{h^2} = \frac{\partial^2 u}{\partial y^2}(x,y) + \mathcal{O}(h^2).$$

- Therefore, $\triangle u$ can be approximated by

$$\triangle u - \frac{1}{h^2}\Big(u(x,y-h) + u(x-h,y) \\ -4u(x,y) + u(x,y+h) + u(x+h,y)\Big) = \mathcal{O}(h^2).$$

Discretisation of the 2D-Poisson Problem

- Setting $u_{i,j} := u(ih, jh)$ we get for $-\triangle u = f$ on ω_h

$$-u_{i,j-1} - u_{i-1,j} + 4u_{i,j} - u_{i+1,j} - u_{i,j+1} = h^2 f_{ij}, \quad i,j \in \{1, \dots, N-1\}$$

  with $f_{ij} := f(ih, jh)$.
- The boundary conditions yield

$$u_{0,i} = u_{N,i} = u_{i,0} = u_{i,N} = 0, \quad i = 0, \dots, N.$$

- Sorting the unknown in a lexicographic order

$$\begin{array}{cccc} (h,h), & (2h,h), & \dots & ((N-1)h,h) \\ (h,2h) & (2h,2h), & \dots & ((N-1)h,2h) \\ \vdots & \vdots & \vdots & \vdots \\ (h,(N-1)h) & (2h,(N-1)h) & \dots & ((N-1)h,(N-1)h). \end{array}$$

  we get the unknowns $U_{i+(N-1)(j-1)} = u_{i,j}$.

Discretisation of the 2D-Poisson Problem

- We obtain the following linear system for $U = (U_i)_{i=1}^{(N-1)^2}$:

$$AU = F. \tag{2}$$

with $F := (f_i)_{i=1}^{(N-1)^2}$, $f_{i+(N-1)(j-1)} = f(ih, jh)$, $i, j \in \{1, \ldots, N-1\}$ and

$$\begin{aligned} A &:= \frac{1}{h^2} tridiag(-I_{N-1}, T, -I_{N-1}) \in \mathbb{R}^{(N-1)^2 \times (N-1)^2}, \\ T &:= tridiag(-1, 4, -1) \in \mathbb{R}^{(N-1) \times (N-1)}. \end{aligned}$$

I_k is the k-dimensional identity matrix.

Solution of 2D-Poisson Problem

Theorem

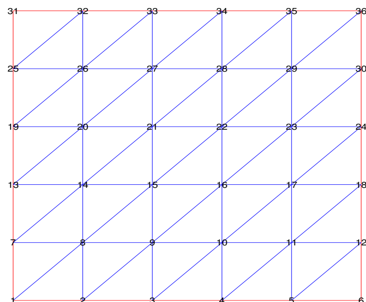
The linear system possesses a unique solution. For sufficiently smooth solutions u of the continuous problem we get

$$\|R_h u - U\|_\infty \leq Ch^2 \|u\|_{C^4(\overline{\Omega})}.$$

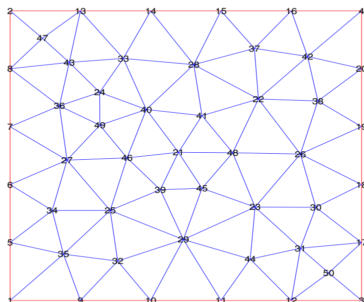
C is a constant independent of h and u. $R_h u \in \mathbb{R}^{(N-1)^2}$ is the restriction of the solution on the interior grid points.

- Thus, the approximation is of second order.
- The linear system possesses $(N-1)^2$ unknowns. In three dimensions we would get $(N-1)^3$ unknowns.

Excursion to Finite Elements



structured grid

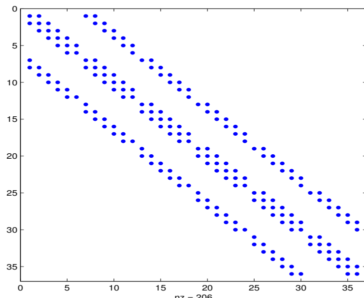


unstructured grid

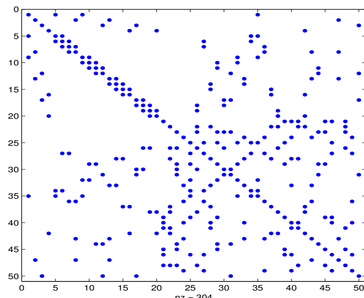
Remarks

- Finite Element methods are the most popular approach for solving this kind of problems.
- Finite Element methods are defined on grids and are equipped with certain local basis functions.

Sparsity Pattern of the corresponding matrices



structured grid



unstructured grid

- A sparsity pattern is given by the non-zero entries of a matrix
- Typically, matrices arising in finite differences or finite elements are sparse.

Sparsity Pattern and CSR

- Storage of non-zero elements and structure using *compressed sparse row (CSR) format*.
- CSR format stores 3 vectors
  1. The first vector contains all non-zero entries. The data is sorted row-wise.
  2. The second vector contains the corresponding column indices.
  3. The third vector stores one entry for each row containing the starting position of the corresponding data in the first vector.
- CSR format is well suited for matrix vector products.

Example

Store the matrix

$$\begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & -1 & 2 & -1 & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{pmatrix}$$

in CSR-format

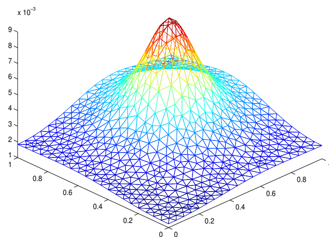
Start. points	1		3			6			9			12	
Column Ind.	1	2	1	2	3	2	3	4	3	4	5	4	5
Data	2	-1	-1	2	-1	-1	2	-1	-1	2	-1	-1	2

Example of a Finite Element Solution

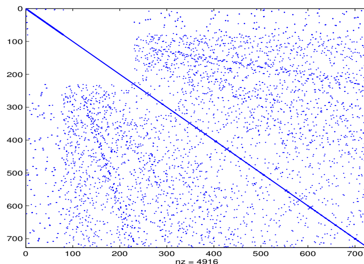
Consider the diffusion reaction problem

$$-\triangle u + u = f \text{ in } \Omega := (0,1)^2, \qquad \frac{\partial u}{\partial n} = 0 \text{ on } \partial\Omega$$

with $f(x,y) := \exp\left(100(-(x-0.6)^2 - (y-0.6)^2)\right)$



Solution



Sparsity Pattern

Outline

1 Poisson Problem and Finite Differences

2 First Steps in MPI

First Example - Hello World Program (I)

```
#include <stdio.h>
#include <string.h>
#include <mpi.h>

main(int argc, char** argv)
{
    int my_rank;      /* Rank of process */
    int p;            /* Number of processes */
    int source;       /* Rank of sender   */
    int dest;         /* Rank of receiver */
    int tag=50;       /* Tag for messages */
    char message[100]; /* Storage for the message */
    MPI_Status status; /* Return status for receive */
    int  namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Get_processor_name(processor_name,&namelen);
```

First Example - Hello World Program (II)

```
    if (my_rank!=0) {
         sprintf(message,"Greetings from process %d from %s",
           my_rank, processor_name);
         dest=0;
         MPI_Send(message,strlen(message)+1,
           MPI_CHAR, dest, tag, MPI_COMM_WORLD);
     }
     else
     {
      for (source=1; source <p ; source ++)
      {
          MPI_Recv(message,100,MPI_CHAR,source,
            tag,MPI_COMM_WORLD, &status);
          printf("%s\n",message);
      }
     }

     MPI_Finalize();
 }
```

First example

- We start the program with p processes. Then there exists processes with ranks $0, 1, \ldots, p-1$.
- Each process not equal to 0 sends a message containing the process number to process 0. Process 0 prints the message on the screen
- Output for 8 process:

```
Greetings from process 1 from eraping-ThinkPad-X60s
Greetings from process 2 from eraping-ThinkPad-X60s
Greetings from process 3 from eraping-ThinkPad-X60s
Greetings from process 4 from eraping-ThinkPad-X60s
Greetings from process 5 from eraping-ThinkPad-X60s
Greetings from process 6 from eraping-ThinkPad-X60s
Greetings from process 7 from eraping-ThinkPad-X60s
```

Structure of a MPI program

```
                    ...
#include <mpi.h>
                    ...
main(int argc, char* argv[]) {
                    ...
/* No MPI functions called before this */
MPI_Init(&argc, &argv);
                    ...
MPI_Finalize();
/* No MPI function called after this */
                    ...
} /* end main */
                    ...
```

Some Explanations

- The global structure is always the same.
- All MPI commands start with `MPI_`. Predefined constants in MPI are given in capital letters.
- Include header file for MPI

```
#include <mpi.h>
```

- Initialization of MPI

```
MPI_Init(&argc, &argv);
```

  This has to be the first MPI call in a program!
- Finalization of MPI

```
MPI_Finalize();
```

  This must be the last MPI call in the program.
- Almost all MPI commands return an error code using an integer value.

Communication Environment

	int MPI_Comm_rank (MPI_COMM comm, int *rank);
IN:	
comm	Communicator (handle)
OUT:	
rank	Number of the process in group comm

- The processes reads information about the parallel environment. For p processes each process will be provided with a rank between 0 and $p-1$.
- The first parameter is a communicator. A *communicator* is a group of processes, who are able to send messages to each other.
- The communicator `MPI_COMM_WORLD` is pre-defined. `MPI_COMM_WORLD` consists of all used processes.
- The second parameter is a pointer to an int. Here, the rank number of the process is returned. A value between 0 and $p-1$.

Communication Environment

int MPI_Comm_size(MPI_Comm comm, int *size)	
IN:	
comm	Communicator (Handle)
OUT:	
size	Number of processes in comm

- returns the number of processes `size` in the communicator group `comm`.
- Within this course we will just use `MPI_COMM_WORLD`. We will not build our own groups.

Communication Environment

	int MPI_Get_processor_name(char *name, int *resultlen)
IN:	
OUT:	
name	Name of the processor
resultlen	Length of the name of the processor

- returns the name and length of the processor name.
- The maximum length of the processor name is given by `MPI_MAX_PROCESSOR_NAME`.

Sending/ Receiving Messages

- Sending and receiving from messages is the heart of MPI.
- We start with the standard form of sending/receiving of messages. Alternative possibilities will be discussed later.
- Necessary information for sending/ receiving messages
  - The sender and receiver of the message must be known. Here, the rank within the communicator group is used.
  - message
  - data type of message
  - In order to distinguish messages, a `tag` is used. This is an integer between 0 and at least $2^{15} - 1$.

Sending messages

```
int MPI_Send( void* message, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

IN:	
message	Initial address of message
count	Length of message
datatype	MPI data type of each element of the message
dest	Rank number of the receiver
tag	Tag of the message
comm	Communicator
OUT:	

- The message is contained in an array starting at the address `message`.
- The next parameters `count` and `datatype` define the required storage demand of the message.

MPI data types

MPI data types are related to the corresponding *C* data types.

MPI data type	C data type
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Receiving Messages

```
int MPI_Recv( void* message, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
```

IN:	
count	maximum length of message
datatype	MPI datatype of the elements of the message
source	Rank number of the sender
tag	Tag of the message
comm	Communicator
OUT:	
message	Initial address of message
status	Information about received message

Receiving Messages

- The only new object is the status message. The data type of the status is MPI_Status.
- MPI_Status contains information about the received data.
- Using for instance MPI_Get_count returns the size of the message. The user does not need to know the size of the message in advance.
- `tag` does not need to be specified. One can use the wild-card MPI_ANY_TAG.
- The rank of the sender can be replaced by MPI_ANY_SOURCE .

Compiling and starting of parallel programs

- Compiling

  ```
  mpicc Example01.c -o Example01
  ```

  The name of executive can be determined with the help of the option $-o$.

- Execution with p processes

  ```
  mpirun -np p ./Example01
  ```

  The used CPUs can be determined with the help of `-machinefile file`. file contains the names of the processors.

Point-to-Point Communication

Definition

We call the communication between ONE sender and ONE receiver *point-to-point communication*.

Classification of point-to-point communication

- Blocking/ non-blocking communication
- Buffered/ non-buffered communication
- Synchronous/ asynchronous communication

Blocking/ non-blocking communication

- Most of the MPI point-to-point routines can be used in either blocking or non-blocking mode.
- A blocking send routine will only return after it is safe to modify the application buffer (your send data) for reuse.
- A blocking receive only returns after the data has arrived and is ready for use by the program.
- Non-blocking send and receive will return almost immediately. They do not wait for any communication events to complete.
- Non-blocking communications are primarily used to overlap computation with communication and exploit possible performance gains.
- There are waiting routines to guarantee that the data is sent.

Buffered/ non-buffered communication

- In a buffered communication the data will be buffered.
- Then, the sender of a message has not to wait until the receiver has confirmed sending.
- The disadvantage is that the data must be copied twice. Memory problems for large messages can occur.
- The user can control an own address space. This space is called application buffer.

Synchronous/ asynchronous communication

- If there is no communication buffer, the communication has to be synchronous. The sender cannot send until the receiver of the message is ready to get the message.
- In synchronous mode sender and receiver can only proceed after sending an receiving of the message is complete.