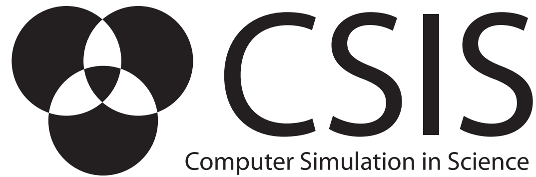


Lab Course II
Parallel Computing

Prof. Dr. Francesco Knechtli*

Dr. Roman Höllwieser†

Department of Physics, Bergische Universitt Wuppertal, Gaussstr. 20, D-42119 Wuppertal, Germany

January 18, 2018

Abstract

The goal of this lecture and tutorial series is to teach you state-of-the-art methods and tools for parallel computing and programming environments for CPU and GPU clusters. You can find a sample of programs and scripts discussed in the lecture or in the tutorial in `http://csis.uni-wuppertal.de/courses/lab217.html`.

Keywords: MPI, openMP, CUDA, Fox & Lanczos algorithm

*knechtli@physics.uni-wuppertal.de, office D.10.24
†hoellwieser@uni-wuppertal.de, office G.11.37

Contents

1 MPI: Greetings! 1
1.1 The Program . . . 1
1.2 Execution . . . 2
1.3 MPI . . . 4
1.3.1 General MPI Programs . . . 4
1.3.2 Finding out about the rest of the world . . . 5
1.3.3 Message: Data + Envelope . . . 5
1.4 Timing in MPI . . . 7

2 An application: Numerical Integration 8
2.1 The trapezoidal rule . . . 8
2.2 A serial program for the trapezoidal rule . . . 9
2.3 Parallelizing the trapezoidal rule . . . 10
2.4 I/O on parallel systems . . . 12

3 Collective Communication 15
3.1 Tree-structured communication . . . 15
3.2 Broadcast . . . 16
3.3 Reduce . . . 18
3.4 Safety, buffering and synchronization . . . 20
3.5 Dot product . . . 22
3.6 Matrix times Vector . . . 27
3.7 Gather and Scatter . . . 28
3.8 Allreduce . . . 31
3.9 Allgather . . . 33
3.10 Application of matrix $\times$ vector to matrix $\times$ matrix . . . 33
3.11 Circular shift of `local_B` . . . 35

4 Fox's algorithm for parallel matrix multiplication 36
4.1 Matrix multiplication . . . 36
4.2 Fox's algorithm . . . 36
4.3 Parallel Fox's algorithm (outline) . . . 39
4.4 Topologies . . . 40
4.5 `MPI_Cart_sub` . . . 44
4.6 Implementation of Fox's algorithm . . . 50

5 Strong/weak scaling, Amdahl's law 51
5.1 Amdahl's law . . . 52
5.2 Efficiency . . . 52
5.3 Overhead . . . 53
5.4 Scalability . . . 53

6 The Lanczos-algorithm 54
6.1 Strategy . . . 54
6.2 General Procedure . . . 56
6.3 Eigenvalues of T . . . 57
6.4 Error Estimates . . . 59

7 Shared-Memory Parallel Programming with OpenMP 61
7.1 False Sharing and Padding . . . 63
7.2 An OpenMP Trapezoidal Rule Implementation . . . 66
7.3 Scope of variables and the `reduction` clause . . . 67
7.4 The `parallel for` directive & thread safety . . . 68
7.5 OpenMP Accelerator Support for GPUs . . . 70

8 Hybrid Programming with MPI & OpenMP 71
8.1 Hybridization or "mixed-mode" programming . . . 72
8.2 Thread Safety, Processor Affinity & MPI . . . 73
8.3 Designing Hybrid Applications . . . 75

9 GPU Parallel Programming with CUDA 78
9.1 The Device - Graphics Processing Units . . . 79
9.1.1 Thread Hierarchy . . . 79
9.1.2 Memory Management . . . 81
9.1.3 Synchronization, within and between Blocks . . . 86
9.2 Hardware Requirements and Compilation . . . 88
9.3 Hello World! for CUDA - the real thing! . . . 89
9.4 Examples . . . 90
9.4.1 Finding Cumulative Sums . . . 90
9.4.2 Calculate Row Sums . . . 91
9.4.3 Finding Prime Numbers . . . 93
9.5 GPU Accelerated Lanczos Algorithm with Applications . . . 97

10 Makefile Example 98

11 A note on Monte Carlo simulations of a scalar field 99
11.1 The model . . . 99
11.2 Statistical simulations . . . 101
11.3 Markov-chain . . . 103
11.4 Local updates . . . 106
11.4.1 Simple Metropolis update . . . 107
11.5 Hybrid overrelaxation updates . . . 107
11.5.1 Heatbath update . . . 108
11.5.2 Metropolis reflection update . . . 109
11.5.3 Hybrid overrelaxation . . . 110
11.6 Equipartition . . . 110
11.7 Autocorrelation, statistical errors . . . 111

12 Parallelizing the Poisson equation 114
12.1 Poisson equation matrices . . . 114
12.2 Jacobi method . . . 117
12.3 Gauss-Seidel and Successive Over-Relaxation . . . 118
12.4 Conjugate Gradient method . . . 120
12.5 The Assignment . . . 121

References 123

1 MPI: Greetings!

Reference: Peter S. Pacheco: *Parallel Programming with MPI* (PPMPI) [1]

1.1 The Program

An MPI (or parallel) program runs on p processes[1] with rank $0, 1, 2, \ldots, p-1$. Each process other than 0 sends a message to process 0, which prints the messages received.

⇒ example program **greetings.c**, see also chapter 3, pp. 41ff in [1]
`http://csis.uni-wuppertal.de/courses/lab2/greetings.c` © [1]

```
1  /* greetings.c -- greetings program
2   *
3   * Send a message from all processes with rank != 0 to
4   *     process 0 who prints the messages received.
5   *
6   * Input: none.
7   * Output: contents of messages received by process 0.
8   *
9   * See Chapter 3, pp. 41 & ff in PPMPI.
10  */
11 #include <stdio.h>
12 #include <string.h>
13 #include "mpi.h"
14
15 main(int argc, char* argv[]) {
16     int         my_rank;       /* rank of process      */
17     int         p;             /* number of processes  */
18     int         source;        /* rank of sender       */
19     int         dest;          /* rank of receiver     */
20     int         tag = 0;       /* tag for messages     */
21     char        message[100];  /* storage for message  */
22     MPI_Status  status;        /* return status for    */
23                                /* receive              */
24
```

[1] A process is an instance of a program or a subprogram that is executing more or less autonomously on a physical processor.

```
25      /* Start up MPI */
26      MPI_Init(&argc, &argv);
27
28      /* Find out process rank  */
29      MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
30
31      /* Find out number of processes */
32      MPI_Comm_size(MPI_COMM_WORLD, &p);
33
34      if (my_rank != 0) {
35          /* Create message */
36          sprintf(message, "Greetings from process %d!",
37              my_rank);
38          dest = 0;
39          /* Use strlen+1 so that '\0' gets transmitted */
40          MPI_Send(message, strlen(message)+1, MPI_CHAR,
41              dest, tag, MPI_COMM_WORLD);
42      } else { /* my_rank == 0 */
43          for (source = 1; source < p; source++) {
44              MPI_Recv(message, 100, MPI_CHAR, source, tag,
45                  MPI_COMM_WORLD, &status);
46              printf("%s\n", message);
47          }
48      }
49
50      /* Shut down MPI */
51      MPI_Finalize();
52  } /* main */
```

1.2 Execution

- login to frontend called *stromboli*: `ssh -X stromboli`
- compile on stromboli:

  `/cluster/mpi/openmpi/1.6.5-gcc4.8.2/bin/mpicc greetings.c`
- submit a job on stromboli: see script:

  `http://csis.uni-wuppertal.de/courses/lab2/submit_script.sh`

```
#!/bin/bash
#
#SBATCH --nodes=2
#SBATCH --ntasks=16
#SBATCH --exclusive
#SBATCH --partition=NODE2008

MPIDIR=/cluster/mpi/openmpi/1.6.5-gcc4.8.2/
GCCDIR=/cluster/gcc/4.8.2/
export PATH=$PATH:${MPIDIR}/bin:${GCCDIR}/bin
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:${MPIDIR}\
/lib:${GCCDIR}/lib:${GCCDIR}/lib64

mpirun <executable> <input_file>
```

→ `sbatch submit_script.sh` → `squeue` → `scancel`

- on the local machine: `mpicc greetings.c; mpirun -np 16 ./a.out`

NEVER RUN INTERACTIVE JOBS!!!

⇒ Share the output when run on $p = 8$ processes!
We assume that one process runs on each processor (locking mode).
This is what happens when we run a parallel program:

1. The user issues a directive to the operating system that has the effect of placing a copy of the executable program on each processor.
2. Each processor begins execution of its copy of the executable.
3. Different processes can execute different statements by branching within the program based on their process ranks.

Point 3 is a form of MIMD (Multiple-Instruction Multiple-Data) programming called Single-Program Multiple-Data (SPMD), *i.e.*, write only one program and differentiate it using branching statements like

```
if (my_rank != 0)
    ⋮
else
    ⋮
```

1.3 MPI

A parallel C program consists of conventional C statements and preprocessor directives. MPI is a library of definitions and functions that can be used in C (or Fortran,...) programs. `MPI = Message-Passing Interface`

1.3.1 General MPI Programs

Every MPI program must include the preprocessor directive `#include "mpi.h"`, including the declarations and definitions necessary for compiling an MPI program.

MPI identifiers: `MPI_`

- constants: *e.g.* `MPI_CHAR`
- functions: *e.g.* `MPI_Init`

"Skeleton" of an MPI program:

```
    ⋮
#include "mpi.h"
    ⋮
main(int argc, char* argv[]){
    ⋮
MPI_Init(&argc, &argv); // ← before calling MPI functions
    ⋮
MPI_Finalize(); // ← when finished using MPI
    ⋮
}
```

with

- `argc`: # of arguments
- `argv`: array of pointers to arguments
- arguments (character strings) provided when invoking the program
  - `argv[0]` = command itself
  - ⋮
  - `argv[argc-1]`

1.3.2 Finding out about the rest of the world

In order to find out the rank of the process

```
int MPI_Comm_rank(
     MPI_Comm   comm,       // ← input
     int*       my_rank)    // ← output
```

communicator = collection of processes that can send messages to each other, *e.g.* `MPI_COMM_WORLD`, which consists of all processes running when program execution begins

In order to find out how many processes are involved

```
int MPI_Comm_size(
     MPI_Comm   comm,                   // ← input
     int*       number_of_processes)    // ← output
```

1.3.3 Message: Data + Envelope

The actual message passing is accomplished by the two functions

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

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

blocking: the process remains idle until the message has been sent or copied to the system-buffer (`MPI_Send`) resp. becomes available (`MPI_Recv`).

`void*`	:	generic pointer, can be (array of) char, float,...
`message`	:	contents of the mesage
`count`	:	size of the mesage
`datatype`	:	MPI type - contains `count` values, each having MPI-type `datatype` - predefined MPI types, most correspond to C types, *e.g.*, `MPI_CHAR`, `MPI_INT`, `MPI_FLOAT`,...see Table 3.1 in [1] - There must be sufficient storage allocated in the call to `MPI_Recv` to receive the message, otherwise overflow error occurs.
`dest/source`	:	rank of the receiving (`MPI_Send`) / sending (`MPI_Recv`) processes, `source` can be a wildcard: `MPI_ANY_SOURCE`
`tag`	:	message type used to distinguish messages, an integer in the range 0...32767 guaranteed by MPI (only) `MPI_Recv` can use a wildcard: `MPI_ANY_TAG`
`comm`	:	communicator, must be equal in `MPI_Send` and `MPI_Recv` $\Rightarrow$ no wildcard
`status`	:	information on the data received, it references a structure `struct` with at least 3 members: `status`→`MPI_SOURCE` (rank of the source process) `status`→`MPI_TAG` (tag of the message) `status`→`MPI_ERROR` (error code)

In order to determine the size of the message received:

```
int MPI_Get_count(
     MPI_Status*    status,          // ← input
     MPI_Datatype   datatype,        // ← input
     int*           count_pointer)   // ← output
```

return values of `MPI_Send` and `MPI_Recv` are integers = error codes; however default behavior of MPI: abort execution when error occurs $\Rightarrow$ ignore return values

General philosophy: message = data + envelope, the latter contains

1. the rank of the receiver (destination)
2. the rank of the source (sender)
3. a tag
4. a communicator

Comment about C: '\0' = null character is automatically appended to strings to designate their end.

1.4 Timing in MPI

```
double   start, finish
MPI_Barrier(comm);
start = MPI_Wtime();   /* wall-clock time */

    ⋮

/* code being timed */

    ⋮

MPI_Barrier(comm);
finish = MPI_Wtime();
if(my_rank==0)
   printf("Elapsed time = %e seconds \n", finish-start);
```

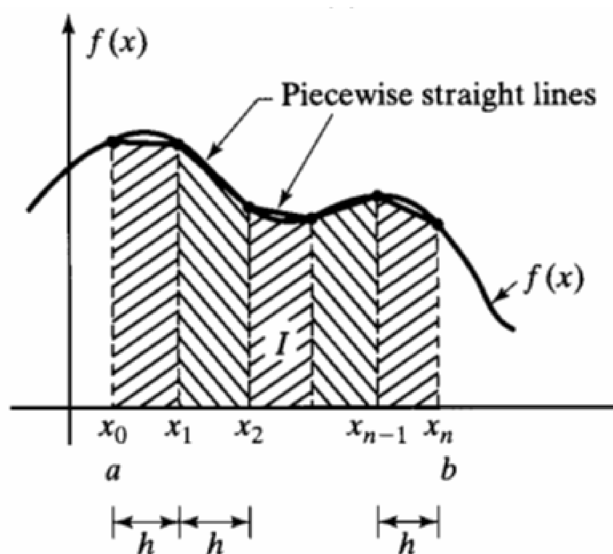
`int MPI_Barrier(MPI_Comm comm)` ← input
each process in `comm` blocks until every process in `comm` calls the function

`double MPI_Wtime(void)`
returns the number of seconds elapsed in the individual processes since some point in the past → sandwich the code between two calls

`double MPI_Wtick(void)`
if `MPI_Wtime` is incremented every μsec, returns 10^{-6} (time resolution)

2 An application: Numerical Integration

2.1 The trapezoidal rule



©http://www.unistudyguides.com/wiki/File:Trapezoidal_Rule_Graph.PNG

$$x_i = a + i \cdot h \qquad i = 0, 1, \ldots, n \qquad n = (b-a)/h$$

$$\begin{aligned}\int_a^b f(x)dx &\approx \frac{1}{2}[f(x_0)+f(x_1)]h + \frac{1}{2}[f(x_1)+f(x_2)]h + \ldots + \frac{1}{2}[f(x_{n-1})+f(x_n)]h \\ &= [\frac{f(x_0)}{2} + f(x_1) + f(x_2) + \ldots + f(x_{n-1}) + \frac{f(x_n)}{2}]h\end{aligned}$$

error: one trapezoid $[x_i, x_{i+1}]$: Taylor expansion

$$f(x) = \underbrace{f(x_i) + (x-x_i)f'(x_i)} + \underbrace{\frac{1}{2}(x-x_i)^2 f''(x_i)} + \ldots$$

$$\int_{x_i}^{x_{i+1}} f(x)dx \quad : \quad \text{exact with trapezoids} \quad \frac{1}{6}(x_{i+1}-x_i)^3 f''(x_i) = O(h^3)$$

all trapezoids: $n \cdot O(h^3) = O(h^2)$.

2.2 A serial program for the trapezoidal rule

http://csis.uni-wuppertal.de/courses/lab2/serial.c © [1]

```
1  /* serial.c -- serial trapezoidal rule
2   *
3   * Calculate definite integral using trapezoidal rule.
4   * The function f(x) is hardwired.
5   * Input: a, b, n.
6   * Output: estimate of integral from a to b of f(x)
7   *     using n trapezoids.
8   *
9   * See Chapter 4, pp. 53 & ff. in PPMPI.
10  */
11
12 #include <stdio.h>
13
14 main() {
15     float  integral;   /* Store result in integral   */
16     float  a, b;       /* Left and right endpoints   */
17     int    n;          /* Number of trapezoids       */
18     float  h;          /* Trapezoid base width       */
19     float  x;
20     int    i;
21
22     float f(float x);  /* Function we're integrating */
23
24     printf("Enter a, b, and n\n");
25     scanf("%f %f %d", &a, &b, &n);
26
27     h = (b-a)/n;
28     integral = (f(a) + f(b))/2.0;
29     x = a;
30     for (i = 1; i <= n-1; i++) {
31         x = x + h;
32         integral = integral + f(x);
33     }
34     integral = integral*h;
35
```

```
36     printf("With n = %d trapezoids, our estimate\n",n);
37     printf("of the integral from %f to %f = %f\n",
38         a, b, integral);
39 } /* main */
40
41 float f(float x) {
42     float return_val;
43     /* Calculate f(x).
44      * Store calculation in return_val. */
45
46     return_val = x*x;
47     return return_val;
48 } /* f */
```

use local machine to compile (also mpicc available):

```
gcc serial.c && ./a.out
```

$$\int_{-1}^{1} x^2 dx = \frac{x^3}{3}\bigg|_{-1}^{1} = \frac{2}{3}$$

try various n, error $\propto \frac{1}{n^2}$

2.3 Parallelizing the trapezoidal rule

Idea:

1. assign a subinterval of [a,b] to each process
2. each process estimates the integral of f over the subinterval
3. the processes' local calculations are added

ad.1. assignment of subintervals:

there are p processes, assume n evenly divisible by p

process	subinterval
0	$[a, a + \frac{n}{p}h]$
1	$[a + \frac{n}{p}h, a + 2\frac{n}{p}h]$
$\vdots$	
i	$[a + i\frac{n}{p}h, a + (i+1)\frac{n}{p}h]$
$\vdots$	
$p-1$	$[a + (p-1)\frac{n}{p}h, b]$

$\Rightarrow$ each process needs to know:

- $p \rightarrow$`MPI_Comm_size`
- rank $i \rightarrow$`MPI_Comm_rank`
- interval ends $[a, b]$
- number n of trapezoids

for now, 'hardwire' latter two values (input by user $\rightarrow$ see later)

ad.2. no communication, trivial parallelization of the work

ad.3. strategy:

each process sends its result to process 0, process 0 does the addition

global variables:
variables whose contents have significance on all processes, *e.g.*, a, b, n

local variables:
variables whose contents only have significance on individual processes, *e.g.*, local interval ends `local_a, local_b`, local number of trapezoids `local_n`

Do not use the same variable both as local and global!

2.4 I/O on parallel systems

Many parallel systems allow all processors

- to read from standard input (the keyboard)
- to write to standard output (the terminal screen)

Question: is it reasonable to have multiple processes reading data from a single terminal and simultaneously write data to the terminal screen?

Problems:
- do all processes get the data?
- in which order are the data written?
- ...

There is not (yet) a consensus in the parallel computing world...

Here: we assume that only process 0 can do I/O

⇒ process 0 has to send the input data to the other processes

1st version: short I/O function that uses `MPI_Send` and `MPI_Recv`:
`http://csis.uni-wuppertal.de/courses/lab2/get_data1.pdf` © [1]

```
1  /* Function Get_data
2   * Reads in the user input a, b, and n.
3   * Input parameters:
4   *      1.  int my_rank:  rank of current process.
5   *      2.  int p:  number of processes.
6   * Output parameters:
7   *      1.  float* a_ptr:  pointer to left endpoint a.
8   *      2.  float* b_ptr:  pointer to right endpoint b.
9   *      3.  int* n_ptr:  pointer to number of trapezoids.
10  * Algorithm:
11  *      1.  Process 0 prompts user for input and
12  *          reads in the values.
13  *      2.  Process 0 sends input values to other
14  *          processes.
15  */
```

```
16 void Get_data(
17          float*  a_ptr    /* out */,
18          float*  b_ptr    /* out */,
19          int*    n_ptr    /* out */,
20          int     my_rank  /* in  */,
21          int     p        /* in  */) {
22
23     int source = 0;  /* All local variables used by */
24     int dest;        /* MPI_Send and MPI_Recv       */
25     int tag;
26     MPI_Status status;
27
28     if (my_rank == 0){
29         printf("Enter a, b, and n\n");
30         scanf("%f %f %d", a_ptr, b_ptr, n_ptr);
31         for (dest = 1; dest < p; dest++){
32             tag = 0;
33             MPI_Send(a_ptr, 1, MPI_FLOAT, dest, tag,
34                 MPI_COMM_WORLD);
35             tag = 1;
36             MPI_Send(b_ptr, 1, MPI_FLOAT, dest, tag,
37                 MPI_COMM_WORLD);
38             tag = 2;
39             MPI_Send(n_ptr, 1, MPI_INT, dest, tag,
40                 MPI_COMM_WORLD);
41         }
42     } else {
43         tag = 0;
44         MPI_Recv(a_ptr, 1, MPI_FLOAT, source, tag,
45             MPI_COMM_WORLD, &status);
46         tag = 1;
47         MPI_Recv(b_ptr, 1, MPI_FLOAT, source, tag,
48             MPI_COMM_WORLD, &status);
49         tag = 2;
50         MPI_Recv(n_ptr, 1, MPI_INT, source, tag,
51                 MPI_COMM_WORLD, &status);
52     }
53 } /* Get_data */
```

Remark: How to provide input argument from standard input:

```
mpicc get_data.c
```

local: `mpirun -np 2 ./a.out < input.d` or

in `sbatch_script: executable < input.d`

`input.d`: file containing a, b, n

Alternative: process 0 opens an input file and reads its contents

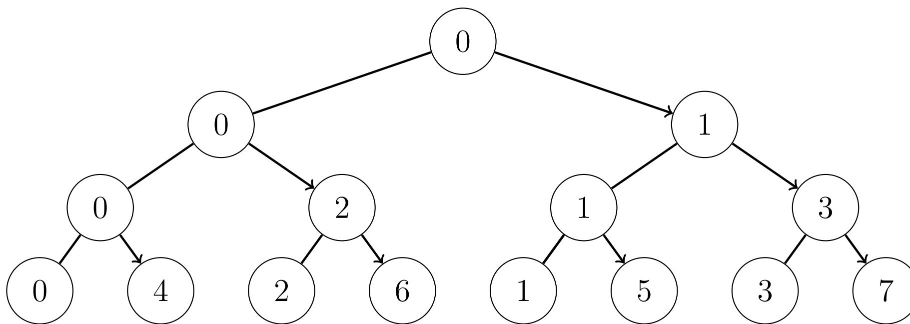
3 Collective Communication

Inefficiencies from the point of view of parallelization in the trapezoidal rule program:

- process 0 only collects the input data (see `Get_data`) and sends it to process 1, while processes $2, \ldots (p-1)$ have to wait
- similarly at the end when process 0 adds up the integral pieces from processes $1, \ldots, (p-1)$

3.1 Tree-structured communication

$p = 8$:



Reduction of the input distribution loop from $p-1$ stages to $\lceil \log_2(p) \rceil$ ($\lceil x \rceil$...ceiling of x, smallest whole number greater than or equal to x)

$p = 1024$: $\lceil \log_2(1024) \rceil = 10 \Rightarrow$ factor 100 reduction of communication time!

Implementation of tree-structured communication:

loop over `stage`$= 0, 1, \ldots, \lceil \log_2(p) \rceil - 1$

* process `rank`$= 0, 1, \ldots, 2^{\texttt{stage}} - 1$ send to `rank`$+2^{\texttt{stage}}$

* process `rank`$= 2^{\texttt{stage}}, 2^{\texttt{stage}}+1, \ldots, 2^{\texttt{stage}+1}-1$ receives from `rank`$-2^{\texttt{stage}}$

In order to find the optimal tree structure we need to know about the topology of our system. Let MPI do it for us...

3.2 Broadcast

collective communication = communication that involves all the processes in a communicator

broadcast = a single process sends the same data to each process in the communicator

```
int MPI_Bcast(
     void*          message,    // ← in/out
     int            count,      // ← input
     MPI_Datatype   datatype,   // ← input
     int            root,       // ← input
     MPI_Comm       comm)       // ← input
```

A copy of "message" on the process with rank `root` is sent to each process in the communication `comm`.

`MPI_Bcast` has to be called by all the processes in `comm`, using the same structure on all processes.

no tag is needed.

in/out `message` is 'in' on process with rank `root` and 'out' on the others.

2nd version of `Get_data`:

```
1  /* Function Get_data2
2   *
3   * Reads in the user input a, b, and n.
4   *
5    * Input parameters:
6   *       1.  int my_rank:  rank of current process.
7   *       2.  int p:  number of processes.
8   *
9   * Output parameters:
10  *       1.  float* a_ptr:  pointer to left endpoint a.
11  *       2.  float* b_ptr:  pointer to right endpoint b.
12  *       3.  int* n_ptr:  pointer to number of trapezoids.
```

```
13   *
14   * Algorithm:
15   *      1.  Process 0 prompts user for input and
16   *          reads in the values.
17   *      2.  Process 0 sends input values to other
18   *          processes using three calls to MPI_Bcast.
19   */
20 void Get_data2(
21          float*  a_ptr    /* out */,
22          float*  b_ptr    /* out */,
23          int*    n_ptr    /* out */,
24          int     my_rank  /* in  */) {
25
26     if (my_rank == 0) {
27         printf("Enter a, b, and n\n");
28         scanf("%f %f %d", a_ptr, b_ptr, n_ptr);
29     }
30     MPI_Bcast(a_ptr, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
31
32     MPI_Bcast(b_ptr, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
33
34     MPI_Bcast(n_ptr, 1, MPI_INT, 0, MPI_COMM_WORLD);
35
36 } /* Get_data2 */
```

Example:

time	process `A` (`root`)	process `B`	process `C`
1	`MPI_Bcast &x`	local work	local work
2	`MPI_Bcast &y`	local work	local work
3	local work	`MPI_Bcast &y`	`MPI_Bcast &x`
4	local work	`MPI_Bcast &x`	`MPI_Bcast &y`
	x=5	x=? (10)	x=? (5)
	y=10	y=? (5)	y=? (10)

In early days of parallel programming, broadcasts and all other collective communications were points of synchronization: a `root` process broadcast

command was completed only when every process received the data.

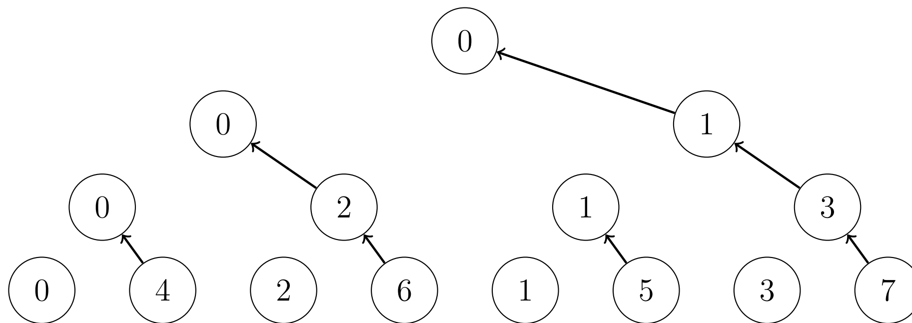
On current systems, if the system has buffering, `A` (`root`) can complete two broadcasts before `B` and `C` begin their calls.

But: in terms of data transmitted collective communications behave as if they were synchronous operations: the first `MPI_Bcast` on `B` matches the first `MPI_Bcast` on `A`, etc.

3.3 Reduce

trapezoidal rule program: process 0 collects the integral pieces from all the other processes ⇒ work is not equally distributed...

idea to do better: reverse the arrows in the tree diagram of 3.1:



1. a. 4 sends to 0; 5 sends to 1; 6 sends to 2; 7 sends to 3.
   b. 0 adds its integral to that of 4; 1 adds its integral to that of 5; etc.
2. a. 2 sends to 0; 3 sends to 1.
   b. 0 adds; 1 adds.
3. a. 1 sends to 0.
   b. 0 adds.

Again, in order to optimize we need to know about the topology of the system. Let MPI do that for us...

```
int MPI_Reduce(
     void*          operand,    // ← input
     void*          result,     // ← output
     int            count,      // ← input
     MPI_Datatype   datatype,   // ← input
     MPI_Op         operator,   // ← input
     int            root,       // ← input
     MPI_Comm       comm)       // ← input
```

`operand` and `result` refer to `count` memory locations with type `datatype`. `result` has meaning only on process `root` but each process has to supply an argument.

`MPI_Reduce` combines the operands stored in `operand`[2] using `operator` and stores the result in `*result` on process with rank `root`. Must be called by all the processes in `comm`, using the same structure on all processes.

class of collective communications called reduction operations, all the processes in a communication contribute data that is combined using a binary operation: add, max, min, logical and, etc.

Predefined reduction operators in MPI © [1]

Operation Name	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and location of maximum
MPI_MINLOC	Minimum and location of minimum

[2] in case `operand` is a vector of elements, `operator` acts element by element

Possible solutions to our example:

1. `MPI_Reduce(&integral,&total,1,MPI_FLOAT,\`
   `MPI_SUM,0,MPI_COMM_WORLD)` ✓

2. `MPI_Reduce(&integral,&integral,...)` ✗

2. is called aliasing of arguments, but it is illegal to alias out or in/out arguments in any MPI function, because MPI might be forced to provide large temporary buffers.

Question: `MPI_Reduce` can take vectors as operand and result. What happens if, *e.g.*, the operator `MPI_Sum` is used?
⇒ The reduce operation, in this case a sum, is done element by element of the vector:

node	0	1	⋯	p-1		result
	loc_a	loc_a	⋯	loc_a	→	∑loc_a
	loc_b	loc_b	⋯	loc_b	→	∑loc_b
	⋮	⋮	⋱	⋮	⋮	⋮
	loc_z	loc_z	⋯	loc_z	→	∑loc_z

3.4 Safety, buffering and synchronization

Example:

time-step	process A	process B
1	`MPI_Send` to B	`MPI_Send` to A
2	`MPI_Recv` from B	`MPI_Recv` from A

If a system has no buffering, execution uses synchronous mode: A cannot send the data until it knows that B is ready to receive them; the same applies to B ⇒ deadlock

If there is buffering of the message, the system has memory available (on the network card or on the RAM) to store the message until the receiver is ready to receive.

But it is unsafe to assume buffering; deadlock can nevertheless occur if the system buffer is not large enough ⇒ change the program above, so that no

deadlock can occur:

time-step	process A	process B
1	MPI_Send to B	MPI_Recv from A
2	MPI_Recv from B	MPI_Send to A

see exercise 2 (even/odd processes)

blocking communications: can safely modify/use data after MPI_Send/Recv

MPI: MPI_Send and MPI_Recv are **blocking** functions: they do not return until the arguments to the functions can be safely modified, this means:

- MPI_Send: message has been sent or message has been copied to system-buffer on the network card or on the RAM, message can be modified afterwards
- MPI_Recv: message has been received, stored in the local buffer and can be used afterwards

alternative: non-blocking communications (MPI_Isend/Irecv), they only start the operation (copying data out of the send buffer, copying data into the local buffer), I = immediate, the system returns immediately after the call, MPI_Wait to complete communications.

MPI continued: alternatives are **non-blocking** communication functions: they only **start** the operation:

- MPI_Isend: the message returns immediately after the call; the system has been informed that it can start copying data out of the send buffer
- MPI_Irecv: the system has been informed that it can start copying data into the buffer

Syntax is very similar to MPI_Send and MPI_Recv, in addition the I-versions have a **request** parameter to complete non-blocking operations:

- MPI_Wait (request, status): blocks until the operation identified by request is completed:
  send: message has been sent or buffered by the system
  receive: message has been copied into receive buffer

Question: What happens if one process makes multiple calls to `MPI_Send` with the same destination and the **same** tag?

⇒ Nothing. The system uses the "first in, first out" principle in dealing with messages and assign an "internal tag" to them.

Example:

What happens, if:

process 0:

`MPI_Send` to 1: a, tag=0

`MPI_Send` to 1: b, tag=1

process 1:

`MPI_Recv` from 0: a, tag=1

`MPI_Recv` from 0: b, tag=0

Answer:

process 0: a=-1, b=1

process 1: a=1, b=-1

⚠ program above deadlocks if system has no buffering!

⇒ tag matters, see `get_data_wrongtag.c`

3.5 Dot product

$$\vec{x} \cdot \vec{y} = x_0 y_0 + x_1 y_1 + \ldots x_{n-1} y_{n-1}$$

p processes, n evenly divisible by p, $\overline{n} = n/p$

assign $\overline{n}$ components of the vectors $\vec{x}, \vec{y}$ to each process

block distribution of the data

process	components
0	$x_0, x_1, \ldots, x_{\overline{n}-1}$
1	$x_{\overline{n}}, x_{\overline{n}+1}, \ldots, x_{2\overline{n}-1}$
$\vdots$	$\vdots$
k	$x_{k\overline{n}}, x_{k\overline{n}+1}, \ldots, x_{(k+1)\overline{n}-1}$
$\vdots$	$\vdots$
p-1	$x_{(p-1)\overline{n}}, x_{(p-1)\overline{n}+1}, \ldots, x_{n-1}$

...and similarly for $\vec{y}$

example uses a static allocation of memory for vectors (`MAX_LOCAL_ORDER`)

can do better using for example in main program:

```
float *local_x;
 ⋮  n_bar=...
local_x=malloc(n_bar *sizeof(float));
```

`http://csis.uni-wuppertal.de/courses/lab2/parallel_dot.c` © [1]

```
1  /* parallel_dot.c -- compute dot product of a vector
2   *     distributed among the processes.
3   *     Block distribution of the vectors.
4   *
5   * Input:
6   *     n: global order of vectors
7   *     x, y:  the vectors
8   *
9   * Output:
10  *     the dot product of x and y.
11  *
12  * Note: Arrays containing vectors are statically
13  *     allocated. n, the global order of the vectors,
14  *     is divisible by p, the number of processes.
15  *
```

```
16   * See Chap 5, pp. 75 & ff in PPMPI.
17   */
18
19  #include <stdio.h>
20  #include "mpi.h"
21  #define MAX_LOCAL_ORDER 100
22
23  main(int argc, char* argv[]) {
24      float   local_x[MAX_LOCAL_ORDER];
25      float   local_y[MAX_LOCAL_ORDER];
26      int     n;
27      int     n_bar;   /* = n/p */
28      float   dot;
29      int     p;
30      int     my_rank;
31
32      void Read_vector(char* prompt, float local_v[],
33               int n_bar, int p, int my_rank);
34      float Parallel_dot(float local_x[], float local_y[],
35               int n_bar);
36      MPI_Init(&argc, &argv);
37      MPI_Comm_size(MPI_COMM_WORLD, &p);
38      MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
39      if (my_rank == 0) {
40          printf("Enter the order of the vectors\n");
41          scanf("%d", &n);
42      }
43      MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
44      n_bar = n/p;
45
46      Read_vector("the first vector", local_x, n_bar,
47          p, my_rank);
48      Read_vector("the second vector", local_y, n_bar,
49          p, my_rank);
50
51      dot = Parallel_dot(local_x, local_y, n_bar);
52
53      if (my_rank == 0)
```

```
54          printf("The dot product is %f\n", dot);
55
56      MPI_Finalize();
57  }  /* main */
58
59  void Read_vector(
60           char*   prompt      /* in  */,
61           float   local_v[]   /* out */,
62           int     n_bar       /* in  */,
63           int     p           /* in  */,
64           int     my_rank     /* in  */) {
65      int i, q;
66      float temp[MAX_LOCAL_ORDER];
67      MPI_Status status;
68      if (my_rank == 0) {
69          printf("Enter %s\n", prompt);
70          for (i = 0; i < n_bar; i++)
71              scanf("%f", &local_v[i]);
72          for (q = 1; q < p; q++) {
73              for (i = 0; i < n_bar; i++)
74                  scanf("%f", &temp[i]);
75              MPI_Send(temp, n_bar, MPI_FLOAT, q, 0,
76                  MPI_COMM_WORLD);
77          }
78      } else {
79          MPI_Recv(local_v, n_bar, MPI_FLOAT, 0, 0,
80              MPI_COMM_WORLD,
81              &status);
82      }
83  }  /* Read_vector */
84
85
86  float Serial_dot(
87            float  x[]  /* in */,
88            float  y[]  /* in */,
89            int    n    /* in */) {
90      int    i;
91      float  sum = 0.0;
```

```
 92     for (i = 0; i < n; i++)
 93         sum = sum + x[i]*y[i];
 94     return sum;
 95 } /* Serial_dot */
 96
 97
 98 float Parallel_dot(
 99           float  local_x[]  /* in */,
100           float  local_y[]  /* in */,
101           int    n_bar      /* in */) {
102
103     float  local_dot;
104     float  dot = 0.0;
105     float  Serial_dot(float x[], float y[], int m);
106
107     local_dot = Serial_dot(local_x, local_y, n_bar);
108     MPI_Reduce(&local_dot, &dot, 1, MPI_FLOAT,
109         MPI_SUM, 0, MPI_COMM_WORLD);
110     return dot;
111 } /* Parallel_dot */
```

on stromboli: `/cluster/mpi/openmpi/1.6.5-gcc4.8.2/bin/mpicc -o parallel_dot parallel_dot.c`

```
sbatch submit_script.sh
```

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks=2
#SBATCH --exclusive
#SBATCH --partition=NODE2008

MPIDIR=/cluster/mpi/openmpi/1.6.5-gcc4.8.2/
GCCDIR=/cluster/gcc/4.8.2/
export PATH=$PATH:${MPIDIR}/bin:${GCCDIR}/bin
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:${MPIDIR}/lib:\
                       ${GCCDIR}/lib:${GCCDIR}/lib64

mpirun ./parallel_dot < input.d
```

- `input.d` includes input parameters
- run with 1 and 2 tasks
- exact result is 4674,02
- float → relative roundoff error 10^{-7}

3.6 Matrix times Vector

$$
\begin{aligned}
A &= (a_{ij}) \qquad i = 0, 1, \ldots, m-1; \quad j = 0, 1, \ldots, n-1; \quad m \times n - \text{matrix} \\
x &= (x_0, x_1, \ldots, x_{n-1})^T \qquad n - \text{dim vector} \\
y &= Ax \qquad \text{matrix-vector product} \rightarrow m - \text{dim vector} \\
y_i &= a_{i0}x_0 + a_{i1}x_1 + \ldots + a_{i,n-1}x_{n-1} \qquad i = 0, 1, \ldots, m-1
\end{aligned}
$$

⇒ block-row (or panel) distribution:

example: $p = 4, m = 8, n = 4$; `local_m`$:= m/p$; `local_n`$:= n/p$

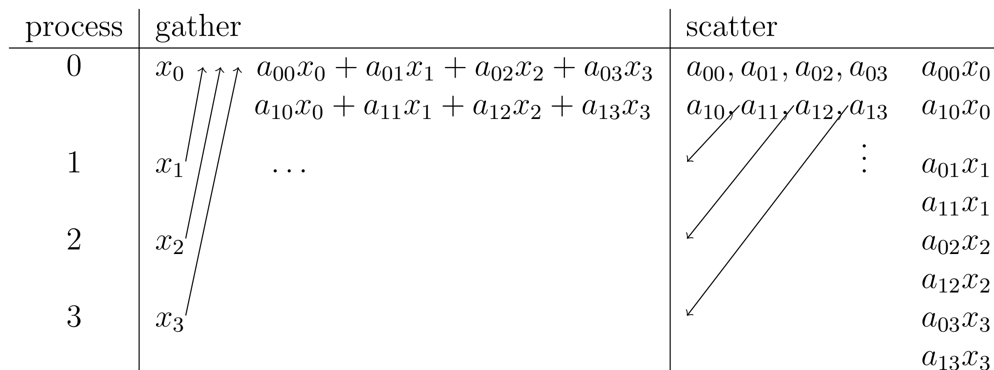
process	elements of A	elements of x, y
0	$a_{00}, a_{01}, a_{02}, a_{03}$	x_0
	$a_{10}, a_{11}, a_{12}, a_{13}$	y_0, y_1
1	$a_{20}, a_{21}, a_{22}, a_{23}$	x_1
	$a_{30}, a_{31}, a_{32}, a_{33}$	y_2, y_3
2	$a_{40}, a_{41}, a_{42}, a_{43}$	x_2
	$a_{50}, a_{51}, a_{52}, a_{53}$	y_4, y_5
3	$a_{60}, a_{61}, a_{62}, a_{63}$	x_3
	$a_{70}, a_{71}, a_{72}, a_{73}$	y_6, y_7

communication problem:

each element of y is a dot product of the corresponding row of A with **all** the elements of x

solution:

either gather all of x onto each process or scatter each row of A across the processes



3.7 Gather and Scatter

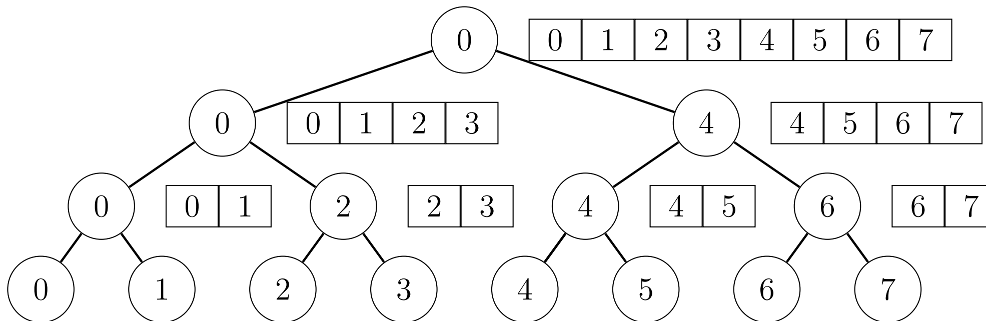
In routine `Read_vector`, process 0 reads in the vector components and distributes them using a loop over $q = 1, \ldots, p-1$ and `MPI_Send` to process `q`. Can we do better using a collective communication?

Yes! → `MPI_Scatter`, see 2nd version of `Read_vector` © [1]

```
1  #include <stdio.h>
2  #include "mpi.h"
3
4  #define MAX_ORDER 100
5
6  void Read_vector(
7           char*   prompt     /* in  */,
8           float   local_x[]  /* out */,
9           int     local_n    /* in  */,
10          int     my_rank    /* in  */,
11          int     p          /* in  */) {
12
13      int    i;
14      float  temp[MAX_ORDER];
15
16      if (my_rank == 0) {
17          printf("%s\n", prompt);
18          for (i = 0; i < p*local_n; i++)
19              scanf("%f", &temp[i]);
20      }
```

```
21        MPI_Scatter(temp, local_n, MPI_FLOAT, local_x, local_n,
22              MPI_FLOAT, 0, MPI_COMM_WORLD);
23
24  }  /* Read_vector */
```

Scatter as a collective communication:




```
int MPI_Scatter(
     void*         send_data,    // ← input
     int           send_count,   // ← input
     MPI_Datatype  send_type,    // ← input
     void*         recv_data,    // ← output
     int           recv_count,   // ← input
     MPI_Datatype  recv_type,    // ← input
     int           root,         // ← input
     MPI_Comm      comm)         // ← input
```

- latter two lines must be the same on all processes in `comm`
- `send_data/count/type` significant only on process `root`
- `send_count` usually the same as `recv_count`, as well as `*_type`
- on process `root`: data referenced by `send_data` are split into p segments, consisting of `send_count` elements of type `send_type`
- the first segment is sent to process `0`, the second to process `1`, etc.
- on each process `recv_count` elements of type `recv_type` referenced by `recv_data` are received

Remark: collective communications do not need tags, since they are executed by all processes in the same order as they appear in the program (in terms of data transmitted they are synchronous operations)

What about the reverse operation (cf. matrix times vector): a vector is split into p segments on p processes, process `0` wants to print it all?

→ see `Print_vector` © [1] using `MPI_Gather`

```
1  #include <stdio.h>
2  #include "mpi.h"
3
4  #define MAX_ORDER 100
5
6  void Print_vector(
7           char*   title       /* in */,
8           float   local_y[]   /* in */,
9           int     local_m     /* in */,
10          int     my_rank     /* in */,
11          int     p           /* in */) {
12
13      int     i;
14      float   temp[MAX_ORDER];
15
16      MPI_Gather(local_y, local_m, MPI_FLOAT, temp, local_m,
17                 MPI_FLOAT, 0, MPI_COMM_WORLD);
18
19      if (my_rank == 0) {
20          printf("%s\n", title);
21          for (i = 0; i < p*local_m; i++)
22              printf("%4.1f ", temp[i]);
23          printf("\n");
24      }
25
26  }  /* Print_vector */
```

```
int MPI_Gather(
    void*          send_data,    // ← input
    int            send_count,   // ← input
    MPI_Datatype   send_type,    // ← input
    void*          recv_data,    // ← output
    int            recv_count,   // ← input
    MPI_Datatype   recv_type,    // ← input
    int            root,         // ← input
    MPI_Comm       comm)         // ← input
```

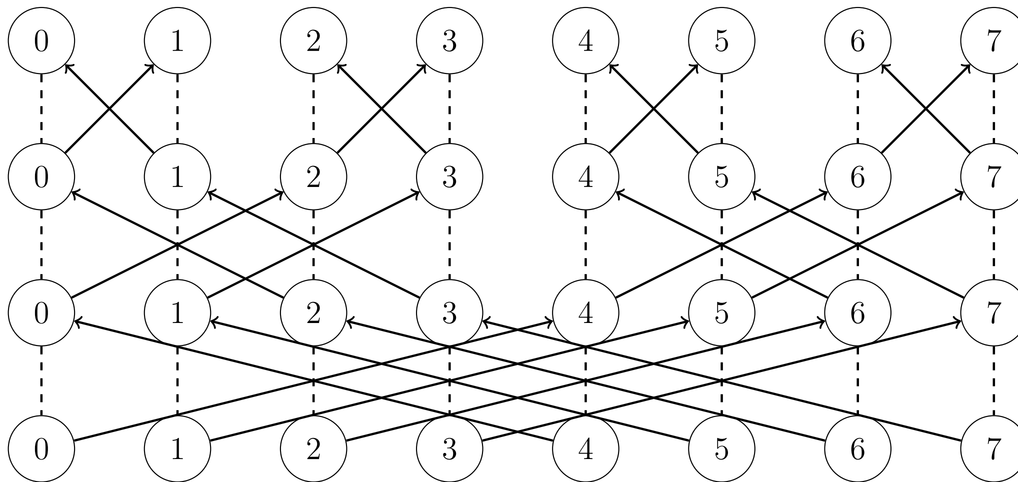
- latter two lines must be the same on all processes in `comm`
- `recv_data/count/type` significant only on process `root`
- `send_count` usually the same as `recv_count`, as well as `*_type`
- data referenced by `send_data`, consisting of `send_count` elements of type `send_type`, on each process in `comm` are collected and stored in process rank order as p segments of `recv_count` elements of type `recv_type` referenced by `recv_data` on process `root`

3.8 Allreduce

For `MPI_Reduce` with `root=0`, cf. `Parallel_dot()` program, only process 0 returns the value of the dot product; other processes return 0.

If we need that each process knows the result we could call `MPI_Bcast` afterwards, or, more efficient:

→ butterfly communication structure, example $p = 8$:



1. a. process 0 and 4 exchange local results; 1 and 5, 2 and 6, 3 and 7
   b. each process adds...
2. a. process 0 and 2 exchange local results; 1 and 3, 4 and 6, 5 and 7
   b. each process adds...
3. a. process 0 and 1 exchange local results; 2 and 3, 4 and 5, 6 and 7
   b. each process adds...

effect: tree-structured `reduce` rooted at all the processes simultaneously (add vertical lines joining same rank)

```
int MPI_Allreduce(
     void*          operand,   // ← input
     void*          result,    // ← output
     int            count,     // ← input
     MPI_Datatype   datatype,  // ← input
     MPI_Op         operator,  // ← input
     MPI_Comm       comm)      // ← input
```

only difference with resect to `MPI_Reduce`: `result` has meaning on all process, no need for argument `root`

3.9 Allgather

Scattering rows of A is not so convenient for $Ax = y$

process	product
0	$a_{00}x_0, a_{10}x_0$
1	$a_{01}x_1, a_{11}x_1$
2	$a_{02}x_2, a_{12}x_2$
3	$a_{03}x_3, a_{13}x_3$

$\rightarrow$ need to call `MPI_Reduce` to finish dot product.

better to use gather (or `MPI_Bcast`), but, we need to gather x onto each process...a loop over `root` $= 0, 1, \ldots, p-1$ is not so efficient $\rightarrow$ use a butterfly communication scheme to simultaneously gather all of x onto each process

```
int MPI_Allgather(
     void*          send_data,    // ← input
     int            send_count,   // ← input
     MPI_Datatype   send_type,    // ← input
     void*          recv_data,    // ← output
     int            recv_count,   // ← input
     MPI_Datatype   recv_type,    // ← input
     MPI_Comm       comm)         // ← input
```

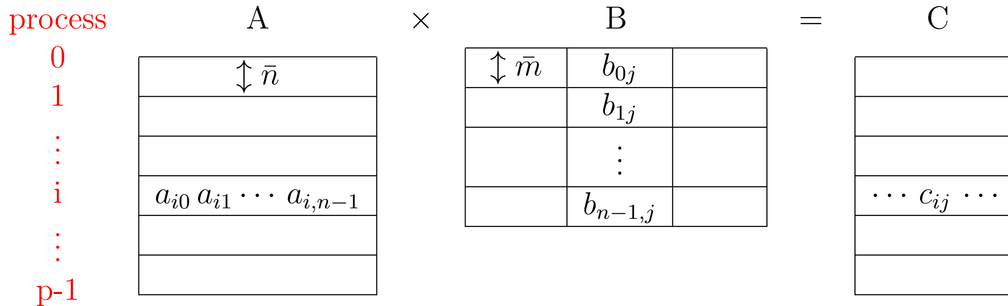
now, `recv_data/count/type` are significant on each process!

3.10 Application of matrix $\times$ vector to matrix $\times$ matrix

$$
\begin{aligned}
A &= (a_{ik}) \quad i = 0, 1, \ldots, m-1; \quad k = 0, 1, \ldots, n-1; \quad m \times n - \text{matrix} \\
B &= (b_{kj}) \quad k = 0, 1, \ldots, n-1; \quad j = 0, 1, \ldots, l-1; \quad n \times l - \text{matrix} \\
C &= A \cdot B \quad \text{matrix-times-matrix} \rightarrow m \times l - \text{matrix} \\
c_{ij} &= a_{i0}b_{0j} + a_{i1}b_{1j} + \ldots + a_{i,n-1}b_{n-1,j} = \sum_{k=0}^{n-1} a_{ik}b_{kj} \\
&\quad \text{for} \quad i = 0, 1, \ldots, m-1 \quad \text{and} \quad j = 0, 1, \ldots, l-1
\end{aligned}
$$

c_{ij} is the dot product of the ith row of A with the jth column of B

⇒ block-row distribution: assume m and n are evenly divisible by the number of processes p: $\overline{m} = m/p, \overline{n} = n/p$



for each column of B{
  `Allgather(column);`
  dot product of `my_rows` of A with `column`;
}

Alternative to Allgather of each column of B: circular shift

step:

1 build pieces of scalar product with local pieces of matrices
on process 0: $c_{ij} = a_{i0}b_{0j} + \ldots + a_{i,\overline{n}-1}b_{\overline{n}-1,j}$
on process 1: $c_{ij} = a_{i\overline{n}}b_{\overline{n}j} + \ldots + a_{i,2\overline{n}-1}b_{2\overline{n}-1,j}$
⋮
circular shift of rows of B: 1 process up...

2 add pieces of scalar product with new local pieces of matrices
on process 0: $c_{ij} = c_{ij} + a_{i\overline{n}}b_{\overline{n}j} + \ldots + a_{i,2\overline{n}-1}b_{2\overline{n}-1,j}$
⋮
circular shift of rows of B: 1 process up...

⋮

p (=$n/\overline{n}$) ... rows $n-\overline{n}$ until $n-1$ of B are on process 0; scalar products are complete; circular shift of rows of B (1 process up) restores B

3.11 Circular shift of `local_B`

use point-to-point communication function:

```
int MPI_Sendrecv_replace(
     void*          buffer,     // ← input/output
     int            count,      // ← input
     MPI_Datatype   datatype,   // ← input
     int            dest,       // ← input
     int            send_tag,   // ← input
     int            source,     // ← input
     int            recv_tag,   // ← input
     MPI_Comm       comm,       // ← input
     MPI_Status*    status)     // ← output
```

- the process in `comm` executing this function:
  - sends with tag `send_tag` content of `buffer` to the process in `comm` with rank `dest`
  - receives with tag `recv_tag` in `buffer` data sent from the process in `comm` with rank `source`
- the processes involved in the send and receive do not need to be distinct
- the process `dest` can receive `buffer` with a call to `MPI_Recv`
- the process `source` can send `buffer` with a call to `MPI_Send`
- `MPI_Sendrecv_replace` uses the same buffer for the send and receive, whereas `MPI_Sendrecv` uses two different buffers.

MPI does not allow aliasing of output variables, so we need a special MPI function `MPI_Sendrecv_replace` which allows to use the same buffer

4 Fox's algorithm for parallel matrix multiplication

4.1 Matrix multiplication

$$
\begin{aligned}
A &= (a_{ij}) \quad B = (b_{ij}) \quad i,j = 0,1,\ldots,n-1; \quad \text{are square } n \times n - \text{matrices} \\
C &= A \cdot B \qquad c_{ij} = a_{i0}b_{0j} + a_{i1}b_{1j} + \ldots + a_{i,n-1}b_{n-1,j} = \sum_{k=0}^{n-1} a_{ik}b_{kj}
\end{aligned}
$$

c_{ij} is the dot (scalar) product of the ith row of A with the jth column of B

Alternative to block-row distribution: processes are mapped onto a grid, assigning submatrices to each process, example: $p = 4, n = 4$

proc 0 $a_{00}\ a_{01}$ $a_{10}\ a_{11}$	proc 1 $a_{02}\ a_{03}$ $a_{12}\ a_{13}$
proc 2 $a_{20}\ a_{21}$ $a_{30}\ a_{31}$	proc 3 $a_{22}\ a_{23}$ $a_{32}\ a_{33}$

checkerboard distribution

4.2 Fox's algorithm

a_{ij}, b_{ij} and c_{ij} are assigned to process $i \cdot n + j \hat{=} (i,j)$, we assume $p = n^2$

checkerboard mapping example: $p = 9, n = 3 \Rightarrow 3 \times 3$ process grid

stage	a				b		
stage 0	$a_{00} \longrightarrow$	$c_{00} = a_{00}b_{00}$	$c_{01} = a_{00}b_{01}$	$c_{02} = a_{00}b_{02}$	b_{00}	b_{01}	b_{02}
	$\leftarrow a_{11} \rightarrow$	$c_{10} = a_{11}b_{10}$	$c_{11} = a_{11}b_{11}$	$c_{12} = a_{11}b_{12}$	b_{10}	b_{11}	b_{12}
	$\longleftarrow a_{22}$	$c_{20} = a_{22}b_{20}$	$c_{21} = a_{22}b_{21}$	$c_{22} = a_{22}b_{22}$	b_{20}	b_{21}	b_{22}
stage 1	$\leftarrow a_{01} \rightarrow$	$c_{00} \mathrel{+}= a_{01}b_{10}$	$c_{01} \mathrel{+}= a_{01}b_{11}$	$c_{02} \mathrel{+}= a_{01}b_{12}$	b_{10}	b_{11}	b_{12}
	$\longleftarrow a_{12}$	$c_{10} \mathrel{+}= a_{12}b_{20}$	$c_{11} \mathrel{+}= a_{12}b_{21}$	$c_{12} \mathrel{+}= a_{12}b_{22}$	b_{20}	b_{21}	b_{22}
	$a_{20} \longrightarrow$	$c_{20} \mathrel{+}= a_{20}b_{00}$	$c_{21} \mathrel{+}= a_{20}b_{01}$	$c_{22} \mathrel{+}= a_{20}b_{02}$	b_{00}	b_{01}	b_{02}
stage 2	$\longleftarrow a_{02}$	$c_{00} \mathrel{+}= a_{02}b_{20}$	$c_{01} \mathrel{+}= a_{02}b_{21}$	$c_{02} \mathrel{+}= a_{02}b_{22}$	b_{20}	b_{21}	b_{22}
	$a_{10} \longrightarrow$	$c_{10} \mathrel{+}= a_{10}b_{00}$	$c_{11} \mathrel{+}= a_{10}b_{01}$	$c_{12} \mathrel{+}= a_{10}b_{02}$	b_{00}	b_{01}	b_{02}
	$\leftarrow a_{21} \rightarrow$	$c_{20} \mathrel{+}= a_{21}b_{10}$	$c_{21} \mathrel{+}= a_{21}b_{11}$	$c_{22} \mathrel{+}= a_{21}b_{12}$	b_{10}	b_{11}	b_{12}

- stage 0 on process (i, j) : $c_{ij} = a_{ii}b_{ij}$
  - broadcast a_{ii} across the i-th row of processes
  - do the local multiplication with b_{ij}
  - "shift" the elements of B up one process row, the elements in the top row are shifted to bottom row (circular shift)
- stage 1 on process (i, j) : $c_{ij} = c_{ij} + a_{i,i+1}b_{i+1,j}$
  in the last row: $i + 1 \rightarrow (i + 1)\text{mod } n$
  - broadcast $a_{i,i+1}$ across the i-th row of processes
  - do the local multiplication, after the circular shift in stage 0, now the local element of B is $b_{i+1,j}$
  - circular shift of the elements of B up one process row
- ...
- stage k on process (i, j) : $c_{ij} = c_{ij} + a_{i,\overline{k}}b_{\overline{k},j}$ with $\overline{k} = (i + k)\text{mod}n$
- ...
- after stage $k = n - 1$, full multiplication on process (i, j):
  $\Rightarrow c_{ij} = a_{ii}b_{ij} + a_{i,i+1}b_{i+1,j} + \ldots + a_{i,n-1}b_{n-1,j} + a_{i0}b_{0j} + \ldots + a_{i,i-1}b_{i-1,j}$

Comments:

- it is not obvious that Fox's algorithm is superior to basic parallel matrix multiplication of 3.10
- it is unlikely that we have $p = n^2$ processors even for relatively small (100×100) matrices $\rightarrow$ how can we modify the algorithm for $p < n^2$? $\rightarrow$ store submatrices rather than matrix elements
- one natural way: square grid of processes: q = the number or process rows = number of process columns = $\sqrt{p}$ divides n evenly $\Rightarrow \overline{n} = n/\sqrt{p}$
- each process is assigned a $\overline{n} \times \overline{n}$ submatrix of A, B and $C \Rightarrow$ submatrices can be multiplied together
- define A_{ij} to be the $\overline{n} \times \overline{n}$ submatrix of A whose first entry is $a_{i\overline{n},j\overline{n}}$

example: $p = n = 4, \overline{n} = n/\sqrt{p} = 2$

process 0	process 1
$A_{00} = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix}$	$A_{01} = \begin{pmatrix} a_{02} & a_{03} \\ a_{12} & a_{13} \end{pmatrix}$
process 2	process 3
$A_{10} = \begin{pmatrix} a_{20} & a_{21} \\ a_{30} & a_{31} \end{pmatrix}$	$A_{11} = \begin{pmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{pmatrix}$

and similarly for B_{ij} and C_{ij}

A_{ij}, B_{ij} and C_{ij} are assigned to process $(i, j), i, j = 0, 1, \ldots, q-1$, where $q = \sqrt{p} \Rightarrow q \times q$ process' grid: rank $= iq + j$.

$C_{ij} = A_{ii}B_{ij} + A_{i,i+1}B_{i+1,j} + \ldots + A_{i,q-1}B_{q-1,j} + A_{i0}B_{0j} + \ldots + A_{i,i-1}B_{i-1,j}$

$\Rightarrow$ Fox's algorithm with q stages $k = 0, 1, \ldots, q-1$:
- $\overline{k} = (i + k) \bmod q$
- broadcast $A_{i\overline{k}}$
- compute $C_{ij} = C_{ij} + A_{i\overline{k}}B_{\overline{k}j}$
- circular shift of B_{ij} up one process row

proof: multiply out each submatrix product, *e.g.*, on process $(0, 0)$:

$$\begin{aligned}
&\begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix}\begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix} = \begin{pmatrix} a_{00}b_{00} + a_{01}b_{10} & \cdots \\ \cdots & \cdots \end{pmatrix} \\
+&\begin{pmatrix} a_{02} & a_{03} \\ a_{12} & a_{13} \end{pmatrix}\begin{pmatrix} b_{20} & b_{21} \\ b_{30} & b_{31} \end{pmatrix} = \begin{pmatrix} a_{02}b_{20} + a_{03}b_{30} & \cdots \\ \cdots & \cdots \end{pmatrix} \\
&\vdots \\
=&\begin{pmatrix} a_{00}b_{00} + a_{01}b_{10} + a_{02}b_{20} + a_{03}b_{30} & \cdots \\ \cdots & \cdots \end{pmatrix}
\end{aligned}$$

$\rightarrow$ computes the correct value of c_{ij}

4.3 Parallel Fox's algorithm (outline)

```
/* my process row = i, my process column = j */
q = sqrt(p);
dest = ((i+q-1) mod q,j);
source = ((i+1) mod q,j);

for (stage=0; stage<q; stage++){
   k_bar = (i + stage) mod q;
   Broadcast A[i,k_bar] across process row i;
   /* C was initialized to zero */
   C[i,j] = C[i,j] + A[i,k_bar]*B[k_bar,j];
   Send B[k_bar,j] to dest;
   Receive B[(k_bar+1) mod q,j] from source;
}
```

→ need to broadcast $A_{i\bar{k}}$ across the ith row before multiplication, and

→ shift the elements of $B_{\bar{k}j}$ up one row after multiplication (circular shift)

⇒ We need to create multiple communicators: one for each row and one for each column of the grid of processes: `row` and `column communicators`

These are intra-communicators = collection of processes that can send message to each other **and** engage in collective communications

intra-communicator = group + context:

group = ordered collection of q processes, each is assigned a unique `rank` $0, 1, \ldots, q-1$

context = system-defined `tag` (integer) that uniquely identifies a communicator and insures that messages are received correctly

A message can be sent and received only if communicator used by the sending process is the same as the one used by the receiving process. To ensure this, the system checks that the contexts are the same.

example of how to implement an intra communicator:
$p = 9$ processes viewed as a 3×3 grid of processes:

0	1	2
3	4	5
6	7	8

create a "second row communicator" `group`:

`group[0]=3, group[1]=4, group[2]=5`, with $0, 1, 2$ ranks in new group and $3, 4, 5$ ranks in old group (`MPI_COMM_WORLD`)

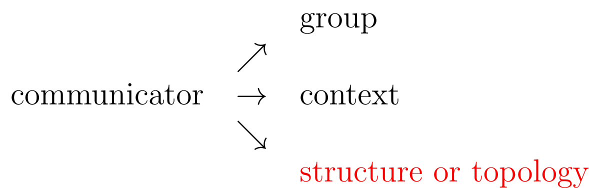
MPI:

`MPI_Comm_group`	:	returns the group underlying a communicator
`MPI_Group_incl`	:	creates a new group from a list of processes in an existing group
`MPI_Comm_create`	:	creates a new communicator (associates a context to a new group

When the new communicator is created, the processes involved "negotiate" the choice of a context (integer) = system-defined tag

→ in communication functions just the context can be sent

→ each process keeps a list of available contexts



4.4 Topologies

communicator = group + context:

topology: additional information = cached information or attribute : **virtual** structure of the processes belonging to a group, there may be no simple relation with the actual underlying physical structure

- Cartesian or grid topology = special case of

- Graph topology

grid topologies important for applications → special set of MPI functions to deal with them

Fox's algorithm:
processes in `MPI_COMM_WORLD` → coordinates of a square grid
each row and each column of the grid → communicators

Grid structure:

1. the number of dimensions (here: 2)
2. the size of each dimension (here: q rows and q columns)
3. periodicity of each dimension (here: first = row dimension: periodicity is unimportant; second = column dimension: has to be periodic for the circular shifts of the submatrices in the columns)
4. possibility to reorder the processes in `MPI_COMM_WORLD` to optimize the mapping of the grid processes to the underlying physical processes

example for reorder = 1:

suppose that the physical topology of the processes is a 3×3 grid and the process ranks in `MPI_COMM_WORLD` are:
→ renumber the processes to improve performance

physical grid

3	1	5
0	7	2
6	8	4

```
MPI_Comm          grid_comm;
int               dim_sizes[2];
int               wrap_around[2];
int               reorder=1;
# of rows         # of columns
dim_sizes[0] =    dim_sizes[1] = q;
wrap_around[0]  = wrap_around[1] = 1;
```

```
MPI_Cart_create(MPI_COMM_WORLD,2,dim_sizes,wrap_around,reorder,
      &grid_comm);
```

Communicator `grid_comm`, containing all processes in `MPI_COMM_WORLD`, is created, possibly reordered and organized in a two-dimensional Cartesian

coordinate system. In order for a process to determine its coordinates:

```
int  coordinates[2];        coordinates[0] = row coord.
int  my_grid_rank;          coordinates[1] = column coord.
```

```
MPI_Comm_rank(grid_comm, &my_grid_rank);

MPI_Cart_coords(grid_comm, my_grid_rank, 2, coordinates);
```

we set reorder=1 ⇒ need to determine `my_grid_rank`

and its inverse function:

```
MPI_Cart_rank(grid_comm, coordinates, &grid_rank);
```

The processes in `grid_comm` are ranked in row-major order. In our 2D case, the first row consists of processes $0, 1, \ldots, q-1 =$ `dim_sizes[1]` -1, the second row of `dim_sizes[1],dim_sizes[1]+1,...,2*dim_sizes[1]-1`, etc...

`http://csis.uni-wuppertal.de/courses/lab2/mpi_cart_test.c`

2D example: `dim_sizes[0] = 2` ↔ # of rows
`dim_sizes[1] = 3` ↔ # of columns

`rank` (in grid)	(	`coordinates[0]`	,	`coordinates[1]`	)
0	(	0	,	0	)
1	(	0	,	1	)
2	(	0	,	2	)
3	(	1	,	0	)
4	(	1	,	1	)
5	(	1	,	2	)

`rank = coordinates[0] * dim_sizes[1] + coordinates[1]`

```
|    ------------------>   1
|   (0,0)    (0,1)    (0,2)
|     x        x        x
v   (1,0)    (1,1)    (1,2)
0     x        x        x
```

```
 /(0,0)  (0,1)  (0,2)
|   x      x      x
|  (1,0)  (1,1)  (1,2)
|   x      x      x
 \(2,0)   ...
```

`http://csis.uni-wuppertal.de/courses/lab2/mpi_cart_test2.c`

periodic boundary conditions: `wrap_around[0]=1`

⇒ `coordinate[0] + dim_sizes[0] = coordinate[0]`

i.e., (2,0) corresponds to (0,0), (3,0) to (1,0), (2,1) to (0,1), etc...

if we set `wrap_around[0]=0` program does not work (range of coordinates)

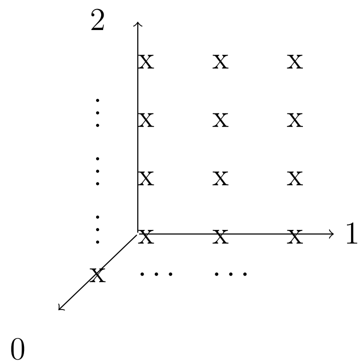
`http://csis.uni-wuppertal.de/courses/lab2/mpi_cart_test.c`

3D example: `dim_sizes[0] = 2`
`dim_sizes[1] = 3`
`dim_sizes[2] = 4`

`rank` (in grid)	(	`coord[0]`	,	`coord[1]`	,	`coord[2]`	)
0	(	0	,	0	,	0	)
1	(	0	,	0	,	1	)
2	(	0	,	0	,	2	)
3	(	0	,	0	,	3	)
4	(	0	,	1	,	0	)
5	(	0	,	1	,	1	)
6	(	0	,	1	,	2	)
7	(	0	,	1	,	3	)
8	(	0	,	2	,	0	)
9	(	0	,	2	,	1	)
10	(	0	,	2	,	2	)
11	(	0	,	2	,	3	)
12	(	1	,	0	,	0	)
13	(	1	,	0	,	1	)
⋮							

ranks 0-3, 4-7, 8-11,... are `z_comm` groups (communicators over z coordinate, keeping x and y coordinates fixed)

ranks 0-11 and 12-23 are the yz_comm groups (communicators at fixed x coordinate)



4.5 MPI_Cart_sub

We can partition a grid into grids of lover dimension. For example, we can create a communicator for each row:

```
    int       free_coords[2];
    MPI_Comm  row_comm;

free_coords[0] = 0;   /* row coordinate is fixed */
free_coords[1] = 1;   /* column coordinate is free */

MPI_Cart_sub(grid_comm,free_coords,&row_comm)
```

q new row communicators are created, in each process the new communicator (for the row that contains that process) is returned in row_comm.

similarly, we can create a communicator for each column:

```
    int       free_coords[2];
    MPI_Comm  col_comm;

free_coords[0] = 1;   /* row coordinate is free */
free_coords[1] = 0;   /* column coordinate is fixed */
```

`MPI_Cart_sub(grid_comm,free_coords,&col_comm)`

`free_coords` is an array of booleans: 0 (false): fixed; 1 (true): free (to vary)

`MPI_Cart_sub` is a collective communication and has to be called by all processes in `grid_comm`

if `grid_comm` has dimension $d_0 \times d_1 \times \ldots \times d_{n-1}$ and `free_coords[i]=0` for $i = i_0, i_1, \ldots, i_{k-1}$ and 1 otherwise, then `MPI_Cart_sub` creates $d_{i_0} \cdot d_{i_1} \cdot \ldots \cdot d_{i_{k-1}}$ new communicators

⇒ example program `mpi_cart_test.c` to setup 2D/3D Cartesian grids:

`http://csis.uni-wuppertal.de/courses/lab2/mpi_cart_test.c`

```
1  /*
2   * Example program to show the use of the MPI functions
3   * to create and use a cartesian grid of processes.
4   *
5   * Use:
6   *   if the chosen number of processes is 6, then it will
7   *      demonstrate the properties of a 2D (2x3) grid
8   *
9   *   if the chosen number of processes is 24, then it will
10  *      demonstrate the properties of a 3D (2x3x4) grid
11  *
12  *   else: it prints an error message
13  */
14
15 #include <stdio.h>
16 #include <stdlib.h>
17 #include <string.h>
18 #include "mpi.h"
19
20 /* FUNCTION TO SETUP THE GRID */
21 void Setup_2D_grid()
22 {
```

```
23     int old_rank;
24     int dimensions[2] = {2,3};   /* assuming p = 6 */
25     int wrap_around[2];
26     int coordinates[2];
27     int free_coords[2];
28
29     int nrows, ncols;
30
31     MPI_Comm cart_comm, row_comm, col_comm;
32     int my_cart_rank;
33
34     /* set up global grid information */
35     MPI_Comm_rank(MPI_COMM_WORLD, &old_rank);
36
37
38     /* circular shift in first and second dimension */
39     wrap_around[0] = 1;
40     wrap_around[1] = 1;
41
42     MPI_Cart_create(MPI_COMM_WORLD, 2, dimensions, wrap_around,\
43             1, &cart_comm);
44     MPI_Comm_rank(cart_comm, &my_cart_rank);
45
46     /* get process coordinates in grid communicator */
47     MPI_Cart_coords(cart_comm, my_cart_rank, 2, coordinates);
48
49     /* set up row communicator */
50     free_coords[0] = 0;
51     free_coords[1] = 1;
52     MPI_Cart_sub(cart_comm, free_coords, &row_comm);
53
54     /* set up column communicator */
55     free_coords[0] = 1;
56     free_coords[1] = 0;
57     MPI_Cart_sub(cart_comm, free_coords, &col_comm);
58
59     MPI_Comm_size(row_comm, &ncols);
60     MPI_Comm_size(col_comm, &nrows);
```

```
61
62     if( old_rank == 0 )
63     {
64       printf("\n2-dimensional cartesian grid,\
65           with dimensions[2] = {2,3}\n");
66       printf("nr of processes in a row: %d\n", ncols);
67       printf("nr of processes in a column: %d\n\n\n(see source\
68           code for further details)\n", nrows);
69     }
70
71     /* print grid info */
72     //printf("old rank =%2d,\tCart. rank=%2d,\tcoords=(%2d,%2d)\
73         \n",old_rank,my_cart_rank,coordinates[0],coordinates[1]);
74   }
75
76
77   /* FUNCTION TO SETUP THE GRID */
78   void Setup_3D_grid()
79   {
80     int old_rank;
81     int  dimensions[3] = {2,3,4};    /* assuming p = 24  */
82     int wrap_around[3];
83     int coordinates[3];
84     int free_coords[3];
85
86     int x_size, y_size, z_size, xy_size, xz_size, yz_size;
87
88     MPI_Comm cart_comm, x_comm, y_comm, z_comm, xy_comm,\
89         xz_comm, yz_comm;
90     int my_cart_rank;
91
92     /* set up global grid information */
93     MPI_Comm_rank(MPI_COMM_WORLD, &old_rank);
94
95     /* circular shift in all dimensions */
96     wrap_around[0] = 1;
97     wrap_around[1] = 1;
98     wrap_around[2] = 1;
```

```
 99
100    MPI_Cart_create(MPI_COMM_WORLD, 3, dimensions,\
101          wrap_around, 1, &cart_comm);
102
103    MPI_Comm_rank(cart_comm, &my_cart_rank);
104
105    /* get process coordinates in grid communicator */
106    MPI_Cart_coords(cart_comm, my_cart_rank, 3,  \
107          coordinates);
108
109    /* set up communicator at fixed X coordinate,\
110         i.e., on the YZ-plane */
111    free_coords[0] = 0;
112    free_coords[1] = 1;
113    free_coords[2] = 1;
114    MPI_Cart_sub(cart_comm, free_coords, &yz_comm);
115
116    /* set up communicator at fixed Y coordinate,\
117         i.e., on the XZ-plane */
118    free_coords[0] = 1;
119    free_coords[1] = 0;
120    free_coords[2] = 1;
121    MPI_Cart_sub(cart_comm, free_coords, &xz_comm);
122
123    /* set up communicator at fixed Z coordinate,\
124         i.e., on the XY-plane */
125    free_coords[0] = 1;
126    free_coords[1] = 1;
127    free_coords[2] = 0;
128    MPI_Cart_sub(cart_comm, free_coords, &xy_comm);
129
130    /* set up communicator over the X coordinate,\
131         i.e., keeping fixed Y and Z coords */
132    free_coords[0] = 1;
133    free_coords[1] = 0;
134    free_coords[2] = 0;
135    MPI_Cart_sub(cart_comm, free_coords, &x_comm);
136
```

```
137   /* set up communicator over the Y coordinate ,\
138       i.e., keeping fixed X and Z coords */
139   free_coords[0] = 0;
140   free_coords[1] = 1;
141   free_coords[2] = 0;
142   MPI_Cart_sub(cart_comm, free_coords, &y_comm);
143
144   /* set up communicator over the Z coordinate ,\
145       i.e., keeping fixed X and Y coords */
146   free_coords[0] = 0;
147   free_coords[1] = 0;
148   free_coords[2] = 1;
149   MPI_Cart_sub(cart_comm, free_coords, &z_comm);
150
151   /* get sizes of all communicators to print them */
152   MPI_Comm_size(x_comm, &x_size);
153   MPI_Comm_size(y_comm, &y_size);
154   MPI_Comm_size(z_comm, &z_size);
155   MPI_Comm_size(xy_comm, &xy_size);
156   MPI_Comm_size(xz_comm, &xz_size);
157   MPI_Comm_size(yz_comm, &yz_size);
158
159   if( old_rank == 0 )
160   {
161     printf("3D cart. grid, with dimensions[3]={2,3,4}\n");
162     printf("nr of processes in x_comm: %d\n",  x_size);
163     printf("nr of processes in y_comm: %d\n",  y_size);
164     printf("nr of processes in z_comm: %d\n",  z_size);
165     printf("nr of processes in xy_comm: %d\n",xy_size);
166     printf("nr of processes in xz_comm: %d\n",xz_size);
167     printf("nr of processes in yz_comm: %d\n\n\n(see \
168         source code for further details)\n", yz_size);
169   }
170   /* print grid info */
171   //printf("old rank = %2d,\tCart. rank = %2d,\tcoords \
172       = (%2d,%2d, %2d)\n", old_rank, my_cart_rank, \
173         coordinates[0], coordinates[1], coordinates[2]);
174 }
```

```
175
176  /****************    MAIN    ****************/
177 int main(int argc, char* argv[])
178 {
179   int p, r;
180   MPI_Init(&argc, &argv);
181   MPI_Comm_size(MPI_COMM_WORLD, &p);
182   MPI_Comm_rank(MPI_COMM_WORLD, &r);
183   if(p == 6) { Setup_2D_grid(); }
184   else if(p == 24) { Setup_3D_grid(); }
185   else { if(r == 0) {printf("this program only works with 6 or\
186      24 processes!!!\n"); } }
187   MPI_Finalize();
188   return 0;
189 }
```

4.6 Implementation of Fox's algorithm

Set up the grid of processes...define a structure for the grid:

```
typedef struct{
     int          p,          // # of processes
     MPI_Comm     comm,       // grid communicator
     MPI_Comm     row_comm,   // communicator for my row
     MPI_Comm     col_comm,   // communicator for my column
     int          q,          // order of grid
     int          my_row,     // my row's coordinate
     int          my_col,     // my column's coordinate
     int          my_rank,    // my rank in the grid
     } GRID_INFO_T
```

The structure is created by a routine

```
void Setup_grid(GRID_INFO_T* grid);
```

5 Strong/weak scaling, Amdahl's law

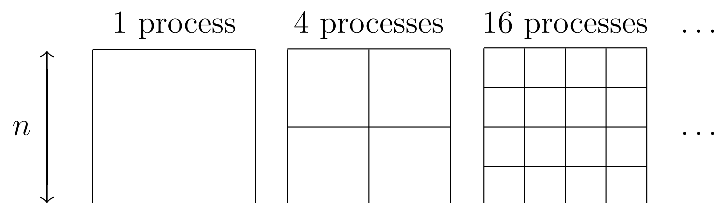
problem size = n, for example the dimension of the matrix in Fox's algorithm or the number of trapezoids in numerical integration

study the **scaling** of the program performance while increasing the number of processes p in two ways:

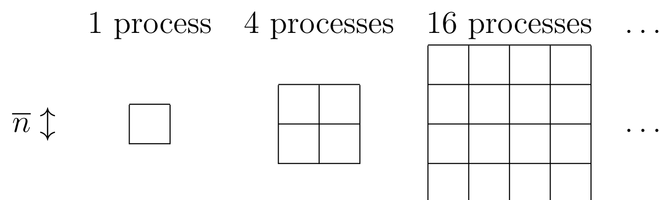
1. problem size stays the same: **strong scaling**
2. problem size increases proportionally to p: **weak scaling**

example: matrix dimension in Fox's algorithm:

1. **strong scaling**: the global problem size n is constant



2. **weak scaling**: the local problem size $\overline{n} = n/q$ is constant



speedup of a parallel program:

T_σ = time to solve the problem on a single process

T_π = time to solve the problem using p processes

speedup $S = \dfrac{T_\sigma}{T_\pi} = \dfrac{T_\sigma(n)}{T_\pi(n,p)}$

5.1 Amdahl's law

Suppose that the problem contains a fraction r of statements that are perfectly parallelizable, $0 \le r \le 1$: This fraction takes rT_σ/p execution time on p processes. The remaining part of the program is inherently serial and takes $(1-r)T_\sigma$ execution time, regardless how large p is (for example reading input):

$$\Rightarrow \text{speedup } S = \frac{T_\sigma}{T_\pi} = \frac{T_\sigma}{\frac{rT_\sigma}{p} + (1-r)T_\sigma} = \frac{1}{\frac{r}{p} + (1-r)}$$

$$\Rightarrow S \text{ increases when } p \text{ increases and } S(p) \xrightarrow{p\to\infty} \frac{1}{1-r}$$

The speedup is bounded by $S_{max} = (1-r)^{-1} \ldots$ if $r = 0.5$, maximum speedup is $S_{max} = 2$; for $r = 0.75$, $S_{max} = 4$ and for $r = 0.99$, $S_{max} = 100$.

Amdahl's law limits the speedup for strong scaling, where the problem size and hence r is fixed. However, in weak scaling, $r(n)$ is likely to increase with the number of processes and therefore larger speedups can be attained.

5.2 Efficiency

$W_\sigma(n)$ = amount of work done by the serial program = $T_\sigma(n)$

$W_\pi(n,p)$ = amount of work done by the parallel program

= sum of the amounts of work done by each process

$= \sum_{q=0}^{p-1} W_q(n,p) = \sum_{q=0}^{p-1} T_\pi(n,p) = p \cdot T_\pi(n,p)$

↖ includes idle times!

$$\text{efficiency } E(n,p) = \frac{W_\sigma(n)}{W_\pi(n,p)} = \frac{T_\sigma(n)}{pT_\pi(n,p)} = \frac{S}{p}$$

$$\text{using Amdahl's law: } E(n,p) = \frac{1}{p(1-r)+r}$$

if $r = 1 : E = 1$ (linear speedup, $S = p$, program is 100% parallelizable)

if $r < 1 : E \to 0$ as $p \to \infty$

5.3 Overhead

$$\begin{aligned} T_0(n,p) &= \text{amount of work done by the parallel program that is} \\ & \text{not done by the serial program} \\ &= W_\pi(n,p) - W_\sigma(n,p) = p \cdot T_\pi(n,p) - T_\sigma(n) \end{aligned}$$

sources of overhead:
1. communication
2. idle time
3. extra computation

$$\Rightarrow E(n,p) = \frac{T_\sigma}{T_0(n,p) + T_\sigma(n)} = \frac{1}{T_0(n,p)/T_\sigma(n) + 1} \Rightarrow \text{as } p \text{ is increased,}$$

the efficiency is solely determined by the overhead function $T_0(n,p)$

Example: trapezoidal rule with n the number of trapezoids

$T_\sigma(n) = k_1 \cdot n$ for some constant k_1

$T_\pi(n,p) = k_1 \frac{n}{p} + k_2 \log_2(p) \quad (+ k_3 = 0)$

time to execute ↑ `MPI_reduce` ↖ negligible

$$\Rightarrow T_0(n,p) = k_2 p \log_2(p) \Rightarrow E = \frac{1}{\frac{k_2 p \log_2(p)}{k_1 n} + 1} = \frac{k_1 n}{k_2 p \log_2(p) + k_1 n}$$

5.4 Scalability

A program is scalable, if, as the number of processes p is increased, there is a rate of increase of the problem size n such that the efficiency is constant.

Example: trapezoidal rule, constant efficiency if $n \propto p \log_2(p)$

$$E = \text{const.} \Leftrightarrow n = \frac{E}{1-E} \frac{k_2}{k_1} p \log_2(p)$$

In general, as p is increased, we can maintain a constant efficiency by increasing the problem size n so that T_0/T_σ remains constant.

6 The Lanczos-algorithm

Lanczos' algorithm is an iterative construction of an orthonormal basis, *i.e.*, $\{v_0, v_1, ...\}$, starting from a random vector v_0 using a tridiagonal matrix T from a symmetric (or hermitian, if complex) $(N \times N)$-matrix A, *i.e.*, $A = A^\dagger = (A^T)^*$. The algorithm is based on numerical linear algebra methods which make use of the so called *Krylov space*, a linear space whose basis vectors are given by successive applications of A to the start vector v_0, *i.e.*,

$$\{v_0, Av_0, A^2 v_0, ...A^{N-1} v_0\}.$$

These methods are highly efficient and widely used in solving linear systems involving *sparse* matrices (that is matrices which are mostly populated by zeros) because in this case the application of A to a vector v can be implemented in a very efficient way.

`http://ieeexplore.ieee.org/document/5763440` provides an interesting implementation of a GPU accelerated version of the Lanczos algorithm with applications, which will be discussed briefly in Section 9.5.

6.1 Strategy

1. We start with a random vector v_0 with norm $\|v_0\| = \sqrt{v_0^\dagger v_0} = 1$, recall

   - scalar product: $v^\dagger w = \sum_{i=0}^{N-1} (v_i)^* w_i$
   - property: $v^\dagger B w = (w^\dagger B^\dagger v)^*$
   - orthogonality: $v \perp w \Leftrightarrow v^\dagger w = 0$

2. We compute the product Av_0 and decompose it as $Av_0 = \alpha_0 v_0 + r_1$ by projecting it onto v_0, hence $r_1 \perp v_0$

   - if $r_1 = 0 : Av_0 = \alpha_0 v_0$, v_0 happened to be an eigenvector of A with eigenvalue α_0 and we restart our iterative procedure with $\tilde{v}_0 \perp v_0$
   - if $r_1 \neq 0$: $r_1 = \beta_0 v_1, \|v_1\| = 1$, hence, we get

$$\begin{aligned} 0 &\stackrel{!}{=} v_0^\dagger r_1 = v_0^\dagger (Av_0 - \alpha_0 v_0) = v_0^\dagger A v_0 - \alpha_0 \Rightarrow \alpha_0 = v_0^\dagger A v_0 \\ &\Rightarrow r_1 = \beta_0 v_1 = Av_0 - \alpha_0 v_0 = Av_0 - (v_0^\dagger A v_0) v_0 \\ &\Rightarrow \beta_0 = \|Av_0 - \alpha_0 v_0\| \in \mathbb{R}, \quad v_1 = \frac{1}{\beta_0}(Av_0 - \alpha_0 v_0) \end{aligned}$$

Furthermore, it follows that

$$\beta_0 = v_1^\dagger(\beta_0 v_1) = v_1^\dagger A v_0 - \alpha_0 \underbrace{v_1^\dagger v_0}_{=0} \Rightarrow \beta_0 = v_1^\dagger A v_0.$$

3. In the next step we project the product Av_1 onto our new basis $\{v_0, v_1\}$, exactly as above, that is

$$Av_1 = \alpha_1 v_1 + \gamma_0 v_0 + r_2, \quad r_2 = \beta_1 v_2 \perp v_0, v_1$$

Imposing orthogonality of the v_i's, we have

$$\begin{aligned}
\alpha_1 &= v_1^\dagger A v_1 \qquad [0 \stackrel{!}{=} v_1^\dagger r_2 = v_1^\dagger A v_1 - \alpha_1 - \gamma_0 \underbrace{v_1^\dagger v_0}_{=0}] \\
\gamma_0 &= v_0^\dagger A v_1 \stackrel{A=A^\dagger}{=} (v_1^\dagger A v_0)^\dagger = \beta_0 \;\; [0 \stackrel{!}{=} v_0^\dagger r_2 = v_0^\dagger A v_1 - \alpha_1 \underbrace{v_0^\dagger v_1}_{=0} - \gamma_0] \\
&\Rightarrow \;\; \beta_1 = \|Av_1 - \alpha_1 v_1 - \beta_0 v_0\| = v_2^\dagger A v_1 \\
&\Rightarrow \;\; v_2 = \frac{1}{\beta_1}(Av_1 - \alpha_1 v_1 - \beta_0 v_0)
\end{aligned}$$

4. To construct v_3 we proceed in the same spirit

$$\begin{aligned}
Av_2 &= \alpha_2 v_2 + \gamma_1 v_1 + \delta_0 v_0 + r_3, \quad r_3 = \beta_2 v_3 \perp v_0, v_1, v_2 \\
\alpha_1 &= v_2^\dagger A v_2 \qquad [0 \stackrel{!}{=} v_2^\dagger r_3] \\
\gamma_1 &= v_1^\dagger A v_2 \stackrel{A=A^\dagger}{=} \beta_1 \qquad [0 \stackrel{!}{=} v_1^\dagger r_3] \\
\delta_0 &= v_0^\dagger A v_2 = (v_2^\dagger A v_0)^\dagger = (v_2^\dagger(\alpha_0 v_0 + \beta_0 v_1))^\dagger = 0 \qquad [0 \stackrel{!}{=} v_0^\dagger r_3] \\
&\Rightarrow \;\; \beta_2 = \|Av_2 - \alpha_2 v_2 - \beta_1 v_1\| = v_3^\dagger A v_2 \\
&\Rightarrow \;\; v_2 = \frac{1}{\beta_2}(Av_2 - \alpha_2 v_2 - \beta_1 v_1)
\end{aligned}$$

In order to construct a new vector v_{k+1} we need only the *two* preceding vectors v_k, v_{k-1}, the others need not even be kept in memory if we are only interested in the components $(\alpha_0, \beta_0, \alpha_1, \ldots)$ of the tridiagonal matrix T.

In summary, we start with a vector v_0, with $\|v_0\| = 1$, and construct

$$\begin{aligned}
Av_0 &= \alpha_0 v_0 + \beta_0 v_1 \quad \text{where} \quad \alpha_0 = v_0^\dagger A v_0 \\
\text{if } \beta_0 &= \|Av_0 - \alpha_0 v_0\| == 0 : \text{exit; else } v_1 = \frac{1}{\beta_0}(Av_0 - \alpha_0 v_0)
\end{aligned}$$

6.2 General Procedure

Having v_0 and v_1, we can iterate the construction of a new vector v_{k+1}:

$$\begin{aligned}
Av_k &= \beta_k v_{k+1} + \alpha_k v_k + \beta_{k-1} v_{k-1} \\
\alpha_k &= v_k^\dagger A v_k \\
\beta_{k-1} &= v_{k-1}^\dagger A v_k \\
\beta_k &= \| Av_k - \alpha_k v_k - \beta_{k-1} v_{k-1} \| = v_{k+1}^\dagger A v_k \\
v_{k+1} &= \frac{1}{\beta_k}(Av_k - \alpha_k v_k - \beta_{k-1} v_{k-1})
\end{aligned}$$

or in terms of the tridiagonal $(N \times N)$-matrix T, the representation of A in the Lanczos basis $\{v_i\}$:

$$T_{ij} = v_i^\dagger A v_j, \qquad i, j = 0, 1, \ldots, N-1$$

or $A \cdot V = V \cdot T$ with

$$V = (\; v_0 \; v_1 \; \ldots \; v_{N-1} \;) \text{ (columns = vectors } v_i)$$

and

$$T = \begin{pmatrix}
\alpha_0 & \beta_0 & & & & & \vdots & \\
\beta_0 & \alpha_1 & \beta_1 & & & \cdots & 0 & \cdots \\
 & \beta_1 & \alpha_2 & \ddots & & & \vdots & \\
 & & \beta_2 & \ddots & \beta_{i-1} & & & \\
 & & & \ddots & \alpha_i & \ddots & & \\
 & \vdots & & & \beta_i & \ddots & \beta_{N-3} & \\
\cdots & 0 & \cdots & & & \ddots & \alpha_{N-2} & \beta_{N-1} \\
 & \vdots & & & & & \beta_{N-2} & \alpha_{N-1}
\end{pmatrix}$$

Hence, T has the α_i on the diagonal and the β_i on the first off-diagonals.

Evaluating $A \cdot V = V \cdot T$, we get, *e.g.*,

0th column: $Av_0 = \alpha_0 v_0 + \beta_0 v_1$

ith column: $Av_i = \beta_{i-1} v_{i-1} + \alpha_i v_i + \beta_i v_{i+1}$

6.3 Eigenvalues of T

The Lanczos iteration yields a sequence of tridiagonal matrices of increasing size:

$$
\begin{aligned}
T^1 &= \alpha_0 && \text{eigenvalue: } \alpha_0 = \theta_1^{(1)} \\
T^2 &= \begin{pmatrix} \alpha_0 & \beta_0 \\ \beta_0 & \alpha_1 \end{pmatrix} && \text{eigenvalues: } \theta_2^{(2)} \le \theta_1^{(2)} \\
T^3 &= \begin{pmatrix} \alpha_0 & \beta_0 & 0 \\ \beta_0 & \alpha_1 & \beta_1 \\ 0 & \beta_1 & \alpha_2 \end{pmatrix} && \text{eigenvalues: } \theta_3^{(3)} \le \theta_2^{(3)} \le \theta_1^{(3)} \\
T^4 &= \begin{pmatrix} \alpha_0 & \beta_0 & 0 & 0 \\ \beta_0 & \alpha_1 & \beta_1 & 0 \\ 0 & \beta_1 & \alpha_2 & \beta_2 \\ 0 & 0 & \beta_2 & \alpha_3 \end{pmatrix} && \text{eigenvalues: } \theta_4^{(4)} \le \theta_3^{(4)} \le \theta_2^{(4)} \le \theta_1^{(4)} \\
&\vdots
\end{aligned}
$$

The spectra (sets of eigenvalues) of $T^{(k)}$ are interlaced:

$$
\begin{array}{ccccccccccc}
 & & & & \theta_1^{(1)} & & & & & & \\
 & & & \theta_2^{(2)} & & \theta_2^{(2)} & & & & & \theta_2^{(2)} \le \theta_1^{(1)} \le \theta_1^{(2)} \\
 & & \theta_3^{(3)} & & \theta_2^{(3)} & & \theta_1^{(3)} & & & & \\
 & \iddots & & & \vdots & & & \ddots & & \text{...} & \text{fast convergence} \\
\lambda_N & \lambda_{N-1} & \cdots & & \cdots & & \cdots & \lambda_2 & \lambda_1 & &
\end{array}
$$

The eigenvalues λ_i of A resp. T are relatively easy to calculate using the "QL method" for tridiagonal matrices:

- use Householder reflections to construct $T = T_1 = Q_1 L_1$, where L_1 has only elements on diagonal and sub-(1st layer off-)diagonal, $Q_1^T Q_1 = \mathbb{1}$
- define a sequence:

  $$T = T_1 \to T_2 = Q_1^T T_1 Q_1 = L_1 Q_1 \to \ldots \to T_s = Q_{s-1}^T T_{s-1} \to \ldots$$

  after each iteration T_s is symmetric and tridiagonal;
  theorem: $s \to \infty : T_\infty$ is diagonal

- if T is a tridiagonal, symmetric $(N \times N)$-matrix and **none** of the elements in the sub- and super-diagonal are zero (*i.e.*, $T_{i,i+1} \neq 0$ for $i = 1, \ldots, N-1$), then **none** of its eigenvalues are degenerate

  Proof: if all $t_{i,i+1} \neq 0$

  $\Rightarrow$ the first $n-1$ columns of $T - \lambda\mathbb{1}$ are linearly independent
  $\Rightarrow$ $\texttt{rank}(T - \lambda\mathbb{1}) = \texttt{dim}(\Im(T - \lambda\mathbb{1})) \geq n-1$
  since $\texttt{dim}(\texttt{Ker}(T - \lambda\mathbb{1})) - \texttt{rank}(T - \lambda\mathbb{1}) = n$
  $\texttt{dim}(\texttt{Ker}(T - \lambda\mathbb{1}))$ # of linearly independent eigenvectors for λ
  $\Rightarrow$ degeneracy of $\lambda = 1$

  Remarks: suppose A is an $m \times n$-matrix: $A \in \mathbb{R}^{m \times n}$

  - `range` of A: $\texttt{ran}(A) = \{y \in \mathbb{R}^m : y = Ax \text{for some} x \in \mathbb{R}^n\}$
  - `null space` of A: $\texttt{null}(A) = \{x \in \mathbb{R}^m n Ax = 0\}$
  - if $A = [a_1, a_2, \ldots, a_n]$ is columns partitioning:
    $\Rightarrow \texttt{ran}(A) = \texttt{span}\{a_1, a_2, \ldots, a_n\}$
  - `rank` of A: $\texttt{rank}(A) = \texttt{dim}(\texttt{ran}(A))$
  - $\texttt{dim}(\texttt{null}(A)) + \texttt{rank}(A) = n$

- suppose that during the Lanczos iteration $\beta_{n-1} = 0$ for some n:

$$T = \left(\begin{array}{cccc|ccc} \alpha_0 & \beta_0 & & & & & \\ \beta_0 & \alpha_1 & \ddots & & & & \\ & \ddots & \ddots & \beta_{n-2} & & & \\ & & \beta_{n-2} & \alpha_{n-1} & 0 & & \\ \hline & & & 0 & \alpha_n & & \cdots \\ & & & & & \ddots & \\ & & & & \vdots & & \end{array} \right)$$

  eigenvalues of the upper, left block $(= T^{(n)})$ are **exact** eigenvalues of A; these blocks appear necessarily if A has **degenerate** eigenvalues (degenerate eigenvalues appear only once per block)

6.4 Error Estimates

Theorem: A is a normal ($[A, A^\dagger] = 0 \Leftrightarrow A^\dagger A = AA^\dagger \Rightarrow A$ has a complete set of orthonormal eigenvectors) $N \times N$-matrix, $\sigma \in \mathbb{C}, v \in \mathbb{C}^N$ and $||v|| = 1$. If $\delta = ||(A - \sigma)v||$, then A has (at least) an eigenvalue λ_i with $|\lambda_i - \sigma| \leq \delta$.

Proof: $Au_i = \lambda_i u_i, \quad i = 1, \ldots, N$

spectral representation: $A = \sum_i u_i \lambda_i u_i^\dagger$

$\Rightarrow A - \sigma = \sum_i u_i(\lambda_i - \sigma)u_i^\dagger$

$\Rightarrow (A - \sigma)v = \sum_i u_i(\lambda_i - \sigma)(u_i^\dagger v)$

$\Rightarrow \delta^2 = \sum_i |\lambda_i - \sigma|^2 |u_i^\dagger v|^2 \qquad (*)$

if we assume that $|\lambda_i - \sigma| > \delta \ \forall i$, then it follows that

$$\sum_i |\lambda_i - \sigma|^2 |u_i^\dagger v|^2 > \delta^2 \underbrace{\sum_i |u_i^\dagger v|^2}_{||v||^2 = 1 \text{ (computed in } u_i \text{ basis)}} = \delta^2, \text{ which contradicts } (*) \ \triangle$$

Application of the theorem to Lanczos-algorithm:

Let $\theta_i^{(n)}$ be an eigenvalue of $T^{(n)}$: $T^{(n)} h_i^{(n)} = \theta_i^{(n)} h_i^{(n)}$, $||h_i^{(n)}|| = 1$

In the Lanczos basis we have then: $(A - \underbrace{\theta_i^{(n)}}_{\sigma}) \underbrace{h_i^{(n)}}_{v} = \beta_{n-1}(h_i^{(n)})_{n-1} v_n$

$\Rightarrow \delta_n^{(n)} = |\beta_{n-1}||(h_i^{(n)})_{n-1}| = |\beta_{n-1}||v_{n-1}^\dagger h_i^{(n)}|$

$\Rightarrow A$ has (at least) one eigenvalue in the interval $\theta_i^{(n)} \pm \delta_i^{(n)}$.

These error estimates can be computed with low computational cost during the Lanczos iteration: we need to diagonalize $T^{(n)}$.

The extremal eigenvalues of $T^{(n)}$ are surprisingly good approximations to the extremal eigenvalues of $A \rightarrow$ Kaniel-Paige convergence theory (see Gene H. Golub, Charles F. Van Loan, "Matrix Computations", section 9.1.4)

Let $\lambda_1 \geq \ldots \geq \lambda_N$ be the eigenvalues of A with orthonormal eigenvectors $z_1, \ldots, z_N$. Let $\theta_1^{(n)} \geq \ldots \geq \theta_n^{(n)}$ be the eigenvalues of $T^{(n)}$.

Theorem:

$$\lambda_1 \geq \theta_1^{(n)} \geq \lambda_1 - \frac{(\lambda_1 - \lambda_N) \tan {\phi_1}^2}{[c_{n-1}(1 + 2\delta_1)]^2} \tag{1}$$

where $\cos\phi_1 = |v_0^\dagger z_1|, \delta_1 = \frac{\lambda_1 - \lambda_2}{\lambda_2 - \lambda_N}$ and $c_{n-1}(x)$ is the Chebyshev polynomial of degree $n-1$.

Chebyshev polynomials: $c_k(z) = 2zc_{k-1}(z) - c_{k-2}(z)^-, c_0 = 1, c_1 = z;$

$|c_k(z)| \leq 1$ if $z \in [-1, 1], c_k$ grow very rapidly outside $[-1, 1]$

Theorem:

$$\lambda_N \leq \theta_n^{(n)} \leq \lambda_N + \frac{(\lambda_1 - \lambda_N) \tan {\phi_N}^2}{[c_{n-1}(1 + 2\delta_N)]^2}$$

where $\cos\phi_N = |v_{N-1}^\dagger z_N|$ and $\delta_N = \frac{\lambda_{N-1} - \lambda_N}{\lambda_1 - \lambda_{N-1}}$.

Comparison of $\theta_1^{(n)}$ with power method: $v = A^{n-1}v_0, \gamma_1^{(n)} = \frac{v^\dagger A v}{v^\dagger v}$

Theorem:

$$\lambda_1 \geq \gamma_1^{(n)} \geq \lambda_1 - (\lambda_1 - \lambda_N) \tan {\phi_1}^2 \left(\frac{\lambda_2}{\lambda_1}\right)^{2(n-1)} \tag{2}$$

Compare lower bounds for λ_1: Lanczos is superior

$$L_{n-1} = \frac{1}{[c_{n-1}\left(2\frac{\lambda_1}{\lambda_2} - 1\right)]^2} \geq \frac{1}{[c_{n-1}\left(1 + 2\frac{\lambda_1 - \lambda_2}{\lambda_2 - \lambda_N}\right)]^2} \quad (1) \quad \text{vs.}$$

$$R_{n-1} = \left(\frac{\lambda_2}{\lambda_1}\right)^{2(n-1)} \quad (2)$$

example: $\dfrac{\lambda_1}{\lambda_2} = 1.5; \dfrac{L_{n-1}}{R_{n-1}} \approx 10^{-2}\ (n = 5), 10^{-6}\ (n = 10), 10^{-15}\ (n = 20)$

7 Shared-Memory Parallel Programming with OpenMP

- shared-memory system ⇒ all the cores can access all memory locations
- OpenMP..."multiprocessing", a directives-based shared-memory API
- an instance of a program running on a processor is called a **thread** (vs. a process in MPI)
- needs compiler support, preprocessor instructions `pragma`s, in C or C^{++} `pragma`s start with: `# pragma`, for OpenMP: `# pragma omp`

⇒ example program `omp_hello.c`, using OpenMP © [1]

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <omp.h>
4
5  void Hello(void); /* Thread function */
6
7  int main(int argc, char* argv[]) {
8  /* Get number of threads from command line */
9    int thread_count = strtol(argv[1], NULL, 10);
10
11 # pragma omp parallel num_threads(thread_count)
12
13   Hello();
14
15   return 0;
16 } /* main */
17
18 void Hello(void) {
19   int my_rank = omp_get_thread_num();
20   int thread_count = omp_get_num_threads();
21
22   printf("Hello from thread %d of %d\n", my_rank,
23                                          thread_count);
24 } /*Hello*/
```

The `parallel` **directive** (after `# pragma omp`) specifies that the structured block of code that follows should be executed by multiple threads.

The number of threads can be specified by the **clause** `num_threads` (in our case `thread_count`), otherwise it's defined by the run-time system.

The **team** of threads, the **master** (original) and `thread_count`-1 **slaves** will call `Hello()` and then hit an **implicit barrier**, until all threads complete.

Then the slave threads terminate and the original or master thread continues.

OpenMP functions `omp_get_thread_num` and `omp_get_num_threads` get the rank or id of a thread $(0, 1, ...,$ `thread_count` $- 1)$ and the total number of threads in the team.

compile on stromboli: `gcc -g -Wall -fopenmp -o omp_hello omp_hello.c`

run on stromboli: `./omp_hello 4`, but DON'T RUN INTERACTIVE JOBS!!!

⇒ submit on stromboli: `sbatch submit_script.sh`:

```
#!/bin/bash
./omp_hello 4 > output
```

example output:

```
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4
Hello from thread 1 of 4
```

If the compiler does not support OpenMP, it will just ignore the `parallel` directive. However, to avoid errors from including `omp.h` and the calls to OpenMP functions, we can check whether the preprocessor macro `_OPENMP` is defined:

```
#ifdef _OPENMP
# include <omp.h>
#endif
```

7.1 False Sharing and Padding

Shared memory computers are everywhere...most laptops and servers have multicore (multiprocessor) CPUs, but there are two cases:

- Symmetric Multi-Processor (SMP): a shared address space with equal-time access for each processor; OS treats every processor the same way
- Non-Uniform Memory Access multiprocessor (NUMA): different memory regions have different access costs (”near” and ”far” memory)

...any multiprocessor CPU with a cache is a NUMA system...cache hierarchy means different processors have different costs to access different address ranges...treating the system as an SMP will not result in proper scaling

- OpenMP is a multi-threading, shared address model; threads communicate by sharing variables
- OS scheduler decides when to run which threads...interleaved fairness
- unintended sharing of data causes race conditions: when the program outcome changes as the threads are scheduled differently
- to control race conditions use synchronization to protect data conflicts, but synchronization is expensive so:
- change how data is accessed to minimize the need for synchronization

e.g., promoting scalars to an array to support creation of an SPMD program, the array elements are contiguous in memory and hence share cache lines

if independent data elements happen to sit on the same cache line, each update will cause the cache lines to slosh back and forth between threads...this is called **false sharing**

Solution: **Pad** arrays so elements you use are on distinct cache lines
⇒ see example: estimating π via unit circle/square area ratio

Padding arrays requires deep knowledge of the cache architecture; systems have different sized cache lines ⇒ software performance may fall apart

⇒ there has got to be a better way to deal with false sharing...synchronization

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <omp.h>
4  #include <time.h>
5  #include <gsl/gsl_rng.h>
6
7  #define MAXTHREADS 4
8  #define PADDING 16
9
10 int inside_unit_circle(double x, double y){
11   return (x*x+y*y<1);
12 }
13
14 int main(int argc, char * argv[]){
15     if(argc!=3){ printf("Correct program usage: %s num_of_\
16         threads_requested num_of_random_picks\n", argv[0]);
17         exit(-1);
18     }
19     int nthd_req = atoi(argv[1]);
20     int npicks  = atoi(argv[2]);
21     if(nthd_req > MAXTHREADS){
22         printf("You can request at most %d threads\n",\
23          MAXTHREADS);
24         exit(-1);
25     }
26     omp_set_num_threads(nthd_req);
27     double inside[MAXTHREADS][PADDING];
28     double outside[MAXTHREADS][PADDING];
29     for(int i=0; i<MAXTHREADS; i++){
30                         inside[i]=0;
31                         outside[i]=0;
32         }
33
34     double t1=omp_get_wtime();
35     #pragma omp parallel
36     {
37         int tid  = omp_get_thread_num();
38         int nthd = omp_get_num_threads();
```

```
39          if(tid==0){
40              printf("Requested %d threads, got %d\n",\
41                  nthd_req,nthd);
42              if(npicks % nthd != 0){
43                  printf("Number of random picks not\
44                      divisible by number of threads\n");
45                  exit(-1);
46              }
47          }
48          const gsl_rng_type * T;
49          gsl_rng * r;
50          gsl_rng_env_setup();
51          T = gsl_rng_default;
52          r = gsl_rng_alloc (T);
53          long seed = abs(((time(NULL)*181)*((tid-83)\
54              *359))%104729);
55          gsl_rng_set(r, seed);
56          for(int i=0; i<npicks/nthd; i++){
57              double x = gsl_rng_uniform(r)*2.0-1.0;
58              double y = gsl_rng_uniform(r)*2.0-1.0;
59
60              if(inside_unit_circle(x,y)){
61                  inside[tid][0] += 1.0;
62              }else{
63                  outside[tid][0] += 1.0;
64              }
65          }
66      }
67      printf("The parallel section took %f seconds to\
68          execute\n", omp_get_wtime()-t1);
69      double out = 0; double in=0;
70      for(int i=0; i<MAXTHREADS; i++){
71          out += outside[i][0];
72          in  += inside[i][0];
73      }
74      printf("Our estimate of pi is %f\n",4.*in/(in+out));
75  }
```

7.2 An OpenMP Trapezoidal Rule Implementation

We can have individual threads compute the areas of individual trapezoids and add them to a shared variable, *e.g.*:

```
global_result += my_result;
```

However, this is called a **race condition**: multiple threads are attempting to access a shared resource, at least one of the accesses is an update, and the accesses can result in an error.

The code that causes the race condition is called a **critical section** and we can use the `critical` directive

```
# pragma omp critical
  global_result += my_result;
```

to make sure that no other thread can start executing this code until the first thread has finished (**mutual exclusion**).

Example code to estimate $\int_1^4 x^2 dx = 21$ using the trapezoidal rule:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <omp.h>
4
5 int main(int argc, char* argv[]) {
6 /* Get number of threads from command line */
7   int thread_count = strtol(argv[1], NULL, 10);
8   double a = 1;
9   double b = 4;
10  double result = 0;
11 # pragma omp parallel num_threads(thread_count)
12     trap(a,b,&result);
13   printf("With %d trapezoids the estimate of the integral
14     of x^2 between %f and %f is %f\n", thread_count, a,
15     b, result);
16   return 0;
17 } /* main */
```

```
18
19 void trap(double a, double b, double* result) {
20 /* Thread function */
21   int n = omp_get_num_threads();
22   int m = omp_get_thread_num();
23   double h = (b-a)/n;
24   double x = a+m*h;
25
26 # pragma omp critical
27     *result += h*x*x;
28
29 } /* trap */
```

⇒ `gcc -fopenmp omp_trap.c && ./a.out 999` gives:

```
With 999 trapezoids the estimate of the integral x^2 between 1
and 4 is 20.977482.
```

7.3 Scope of variables and the `reduction` clause

In OpenMP, the **scope** of a variable refers to the set of threads that can access the variable in a `parallel` block. A variable that can be accessed by all the threads in the team has **shared** scope (global), while a variable that can only be accessed by a single thread has **private** scope (local). OpenMP provides clauses to modify the `default` scope of a variable: *e.g.*, `private(vars)`, `shared(vars)`,...; but there is also a more elegant way:

Suppose the subroutine `trap(n)` returns the local trapezoid contribution from thread `n`, then we could rewrite the above code

```
# pragma omp critical
  global_result += trap(n);
```

this would result in `trap(n)` being executed only one thread at a time; alternatively we can add the `reduction` clause to the `parallel` directive:

```
# pragma omp parallel num_threads(thread_count) \
      reduction(+: global_result)
  global_result += trap(n);
```

The code `reduction(<operator>: <variable list>)` specifies that in our case `global_result` is a `reduction` **variable** and the plus sign ("+") indicates that the `reduction` **operator** is addition (could be any `C` operator: $+, *, -, \&, |, \hat{}, \&\&, ||$). Effectively, OpenMP creates a private variable for each thread, and the run-time system stores each thread's result in this private variable. OpenMP also creates a critical section and the values stored in the private variables are added in this critical section.

7.4 The `parallel for` directive & thread safety

Like the `parallel` directive, the `parallel for` directive forks a team of threads to execute the following structured block, which has to be a `for` loop in **canonical form**, *i.e.*, with a definite number of iterations.

For example, `while` or `do-while`, as well as `for` loops including a `break` or `return` statement can not be parallelized (the only exception is an `exit`).

⇒ Hello World example using the `parallel for` directive:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <omp.h>
4
5  int i,my_rank;
6
7  int main(int argc, char* argv[]) {
8
9  /* Get number of threads from command line */
10   int thread_count = strtol(argv[1], NULL, 10);
11
12 # pragma omp parallel for num_threads(thread_count)
13   for ( i = 0; i < 5; i++ ) {
14     my_rank = omp_get_thread_num();
15     printf("Hello from thread %d processing for loop \
16                index i=%d\n", my_rank, i);
17   }
18   return 0;
19 } /* main */
```

$\Rightarrow$ ./a.out 4 produces for example:

```
Hello from thread 1 processing for loop index i=2
Hello from thread 1 processing for loop index i=3
Hello from thread 2 processing for loop index i=4
Hello from thread 0 processing for loop index i=0
Hello from thread 0 processing for loop index i=1
```

The output is not sorted and OpenMP decides how many threads are needed.

Further, loops in which the results of one or more iterations depend on other (prior) iterations cannot, in general, be correctly parallelized by OpenMP. For example, the parallelized code for the Fibonacci series

```
fibo[0] = fibo[1] = 1;
# pragma omp parallel for num threads(thread count)
  for (i = 2; i < n; i++)
    fibo[i] = fibo[i-1] + fibo[i-2];
```

will not necessarily give the correct result (1 1 2 3 5 8 13 21 34 55) because of a so-called **data** or **loop-carried dependence**.

A block of code is **thread-safe**, if it can be simultaneously executed by multiple threads without causing problems.

A third mechanism for ensuring mutual exclusion in critical sections (besides the critical directive and the reduction clause), is the atomic directive, to be used only when the critical section has the form x <op>= <expression>, x++, ++x, x--, or --x. It is designed to exploit special hardware (load-modify-store) instructions; much faster than an ordinary critical section.

There is no guarantee of **fairness** in mutual exclusion constructs (it is possible that a thread can be blocked forever in waiting for access to a critical section), and it can be dangerous to "nest" mutual exclusion constructs, which might cause a **deadlock**.

Finally, modern microprocessor architectures provide **cache memory** to reduce access times, a perfect playground for OpenMP, however, typical architectures have special hardware to insure that the caches on the different chips are **coherent** to avoid **false sharing**, which might reduce performance.

7.5 OpenMP Accelerator Support for GPUs

OpenMP V4.0/V4.5 include/extend accelerator support for GPUs

The execution model is host-centric:

The host (CPU) offloads target regions (code and data) to the target

The target devices can be a GPU, DSP, coprocessor, etc.

Insert directives to the code block that is offloaded to the device, clauses control data movement between the host device and the target device:

```
# pragma omp target
# pragma omp teams
# pragma omp distribute parallel for
  { for (i=0; i<N; i++)
    z[i] = a*x[i] + y[i]; }
```

- the `target` pragma/directive encloses the code block that is executed on the target device, this model allows the target region executed on the host device
- the `teams` construct creates a league of thread teams to exploit extra level of parallelism on some hardware, *e.g.*, NVIDIA GPUs (thread blocks in CUDA)
- work can be distributed among the teams using the `distribute` construct, the iteration of the for loop is chunked and distributed among the teams
- the `distribute parallel for` construct exploits the parallelism among the teams and within each team

The spec also provides data mapping clauses to control data movement between the host device and the target device

For more details on GPU programming see section 9.

8 Hybrid Programming with MPI & OpenMP

The best from both worlds:

MPI:

- provides a familiar and explicit means to use message passing on distributed memory clusters
- has implementations on many architectures and topologies, inter-node communication relatively easy
- specializes in packing and sending complex data structures over the network with efficient inter-node scatters and reductions
- requires that program state synchronization must be handled explicitly due to the nature of distributed memory
- is the standard for distributed memory communications
- data goes to the process

OpenMP:

- allows for high performance, and relatively straightforward, intra-node communication (threading), which is a shared memory paradigm
- provides an interface for the concurrent utilization of shared memory SMP systems, which is much more efficient that using message passing
- facilitates relatively easy threaded programming
- does not incur the overhead of message passing, since communication among threads and program state synchronization are implicit on each SMP node
- is supported by most major compilers (Intel, IBM, gcc, etc.)
- the process goes to the data

8.1 Hybridization or "mixed-mode" programming

Hybridization is the use of inherently different models of programming in a complementary manner that takes advantage of the good points of each, in order to achieve some benefit not possible otherwise.

Hybridization of MPI and OpenMP:

- facilitates cooperative shared memory (OpenMP) programming across clustered SMP nodes
- MPI facilitates communication among SMP nodes, including the efficient packing and sending of complex data structures
- OpenMP manages the workload on each SMP node
- MPI and OpenMP are used in tandem to manage the overall concurrency of the application

3 ways of hybridization:

- introducing MPI into OpenMP:
  - scale a shared memory OpenMP application for use on multiple SMP nodes in a cluster
  - can help applications scale across multiple SMP nodes
  - entails the rethinking of most of the implicit parallelism
- introducing OpenMP into MPI:
  - reduce an MPI application's sensitivity to becoming communication bound
  - make more efficient use of the shared memory on SMP nodes, thus mitigating the need for explicit intra-node communication
  - straightforward, similar to introducing OpenMP into serial code
- introducing MPI and OpenMP:
  - designing a parallel program from scratch to maximize utilization of a distributed memory machine consisting of SMP nodes
  - will allow designing the right balance between shared memory computations and the associated message passing overhead

8.2 Thread Safety, Processor Affinity & MPI

MPI implementations are not required to be **thread-safe**, therefore MPI calls made within OpenMP threads must be inside of a `critical` section (or a `master` or `single` section, making sure that the MPI function is only called by a single, *e.g.*, the master thread).

This also means, that in a model that has all threads making MPI calls, one should use a thread safe implementation of MPI - the MPI-2 standard actually addresses the issue of thread safety

A common thread-safe execution scenario is given by

1. a single MPI process is launched on each SMP node in the cluster
2. each process spawns N threads on each SMP node
3. at some global sync point, the master thread on MPI process 0 communicates with the master thread on all other nodes
4. the threads belonging to each process continue until another sync point or completion

Example MPI/OpenMP program `ompi.c`:

```
1 #include <stdio.h>
2 #include <omp.h>
3 #include "mpi.h"
4 #define _NUM_THREADS 4
5
6 /* Each MPI process spawns a distinct OpenMP
7  * master thread; so limit the number of MPI
8  * processes to one per node
9  */
10 int main (int argc, char *argv[]) {
11   int q,r,s,t;
12
13   /* set number of threads to spawn */
14   omp_set_num_threads(_NUM_THREADS);
15
```

```
16    /* initialize MPI stuff */
17    MPI_Init(&argc, &argv);
18    MPI_Comm_rank(MPI_COMM_WORLD,&r);
19    MPI_Comm_size(MPI_COMM_WORLD,&s);
20
21    /* the following is a trivial parallel OpenMP
22     * executed by each MPI process */
23   # pragma omp parallel
24      {
25        q = omp_get_thread_num();
26        t = omp_get_num_threads();
27        printf("This is thread %d of %d on process %d of\
28          %d.\n", q, t, r, s);
29      }
30
31    /* finalize MPI */
32    MPI_Finalize();
33    return 0;
34  }
```

compile with `mpicc -fopenmp ompi.c` and `sbatch submit_script.sh`:

```
#!/bin/bash
#SBATCH --nodes=4
#SBATCH --ntasks=4
#SBATCH --exclusive
#SBATCH --partition=NODE2008

MPIDIR=/cluster/mpi/openmpi/1.6.5-gcc4.8.2/
GCCDIR=/cluster/gcc/4.8.2/
export PATH=$PATH:${MPIDIR}/bin:${GCCDIR}/bin
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:${MPIDIR}/lib:\
  ${GCCDIR}/lib:${GCCDIR}/lib64

export VIADEV_USE_AFFINITY=0 # affinity control for mvapich1
export MV2_ENABLE_AFFINITY=0 # affinity control for mvapich2
export OMP_NUM_THREADS=8     # should use omp_set_num_threads

mpirun a.out > out
```

results in output:

```
This is thread 0 of 4 on process 0 of 4.
This is thread 3 of 4 on process 0 of 4.
This is thread 2 of 4 on process 0 of 4.
This is thread 1 of 4 on process 0 of 4.
This is thread 0 of 4 on process 1 of 4.
This is thread 1 of 4 on process 1 of 4.
This is thread 3 of 4 on process 1 of 4.
This is thread 2 of 4 on process 1 of 4.
This is thread 0 of 4 on process 2 of 4.
This is thread 3 of 4 on process 2 of 4.
This is thread 2 of 4 on process 2 of 4.
This is thread 3 of 4 on process 2 of 4.
This is thread 1 of 4 on process 3 of 4.
This is thread 0 of 4 on process 3 of 4.
This is thread 3 of 4 on process 3 of 4.
This is thread 2 of 4 on process 3 of 4.
```

It is recommended that the number of threads is set internally using the OpenMP function `omp_set_num_threads(_NUM_THREADS)`, because the environment variable `OMP_NUM_THREAD` is considered non-portable.

Processor "affinity" is becoming increasingly important in multi-core and virtualized SMP environments. The `AFFINITY` variables must be set accordingly to make sure that the threads on each compute node are not bound to a single core - this would devastate performance. This is accounted for with the MVAPICH implementation of MPI used at stromboli.

8.3 Designing Hybrid Applications

great care should be taken to find the right balance of MPI cummunication and OpenMP "work":

- it is the shared memory parts that do the work; MPI is used to simply keep everyone on the same page
- the ratio of communication among nodes to time spent computing on each SMP node should be minimized in order to maximize scaling

- the shared memory computations on each node should utilize as many threads as possible during the computation parts
- MPI is most efficient at communicating a small number of larger data structures; therefore, many small messages will introduce a communication overhead unnecessarily

3 communication concepts:

1. `root` MPI process controls all communications
    - most straightforward paradigm
    - maps one MPI process to one SMP node
    - each MPI process spawns a fixed number of shared memory threads
    - communication among MPI processes is handled by the main MPI process only, at fixed predetermined intervals
    - allows for tight control of all communications

    ```
    // do only if master thread, else wait
    # pragma omp master {
      if (0 == my_rank)
        // some MPI_ call as ROOT process
      else
        // some MPI_ call as non-ROOT process
    }
    // end of omp master
    ```

2. `master` OpenMP thread controls all communications
    - each MPI process uses its own OpenMP master thread (1 per SMP node) to communicate
    - allows for more asynchronous communications
    - not nearly as rigid as concept 1
    - more care needs to be taken to ensure efficient communications
    - the flexibility may yield efficiencies elsewhere

```
// do only if master thread, else wait
# pragma omp master
{
  // some MPI_ call as an MPI process
}
// end of omp master
```

3. all OpenMP threads may use MPI calls
   - this is by far the most flexible communication scheme
   - enables true distributed behavior similar to that which is possible using pure MPI
   - the greatest risk of inefficiencies are contained using this approach
   - great care must be taken in explicitly accounting for which thread of which MPI process is in communication
   - requires a addressing scheme that denotes the tuple of which MPI processes participating in communication and which thread of the MPI process is involved, *e.g.*, `<my_rank,omp_thread_id>`
   - neither MPI nor OpenMP have built-in facilities for tracking this
   - critical sections may have to be utilized for some level of control and correctness since MPI implementations are not assured to be thread safe!

```
// each thread makes a call; can utilize
// critical sections for some control
# pragma omp critical
{
  // some MPI_ call as an MPI process
}
```

Designing a hybrid application from scratch is ideal, and allows one to best balance the strengths of both MPI and OpenMP to create an optimal performing and scaling application.

For further reference see [2, 3].

9 GPU Parallel Programming with CUDA

Graphics processing has always been a highly parallelized task using optimized devices (GPUs). Recently, GPU programming models (APIs) have been developed to combine the power of CPUs and GPUs to accelerate science applications. Examples include **OpenCL**, **OpenACC** and **OpenMP** (see Section 7.5), all cross-platform, cross-vendor APIs using shared memory, supporting `C/C++/Fortran/`etc. This section introduces **CUDA**, the pioneering API developed by NVIDIA and specific to NVIDIA hardware, mainly supporting `C/C++`. It often provides faster execution than other APIs and 3rd-party support for numerical libraries like `cuBLAS` or `cuFFT`.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <cuda.h>
4
5  __global__ void mykernel(void) {}
6
7  int main(void) {
8      mykernel<<<1,1>>>();
9      printf("Hello World!\n");
10     return 0;
11 }
```

A CUDA program consists of code to be run on the **host**, *i.e.*, the CPU, and code to run on the **device**, *i.e.*, the GPU. `nvcc` separates source code into host and device components, device functions (*e.g.*, `mykernel()`) are processed by the NVIDIA compiler, host functions (*e.g.*, `main()`) are processed by standard host compiler (*e.g.*, `gcc`).

The CUDA C/C++ keyword `__global__` indicates a **kernel**, a function that is called by the host to execute on the device. Triple angle brackets mark a call from host code to device code, also called a **kernel launch**, indicating the number of threads or the size of the parallel program.

Threads in an application are grouped into **blocks**, the entirety of blocks is called the **grid** of that application. In the example above, `mykernel()` only initializes a one-dimensional grid, *i.e.*, one block and one thread, but so far nothing is executed by the device - its architecture is discussed next.

9.1 The Device - Graphics Processing Units

A GPU consists of a large set of **streaming multiprocessors** (SMs), consisting of a number of **streaming processors** (SPs), *i.e.*, individual cores. Each SM is essentially a multicore machine in its own right, hence the GPU is a "multi-multiprocessor" machine.

The cores run threads, as ordinary cores, but threads in an SM run in lockstep, to be explained below. Two threads located in different SMs cannot synchronize with each other in the barrier sense. Though this sounds like a negative at first, it is actually a great advantage, as the independence of threads in separate SMs means that the hardware can run faster. So, if the CUDA application programmer can write his/her algorithm so as to have certain independent chunks, those chunks can be assigned to different SMs.

9.1.1 Thread Hierarchy

GPU operation is highly threaded using **SIMT** (single instruction, multiple thread) architecture. The hardware will partition threads into blocks and assign an entire block to a single SM, though several blocks can run in the same SM. The hardware will then divide a block into **warps**, 32 threads to a warp[3]. Knowing that the hardware works this way, the programmer controls the block size and the number of blocks, and in general writes the code to take advantage of how the hardware works.

The central point is that *all the threads in a warp run the code in lockstep*. During the machine instruction fetch cycle, the same instruction will be fetched for all of the threads in the warp. Then in the execution cycle, each thread will either execute that particular instruction or execute nothing. The execute-nothing case occurs in the case of branching - the problem of thread divergence:

Consider what happens with `if/then/else` code. If some threads in a warp take the `then` branch and others go in the `else` direction, they cannot operate in lockstep. That means that some threads must wait while others execute.

[3] Note that word size is 32 bits, thus for instance floating-point operations in hardware were originally in single precision only. Newer devices are capable of double precision, *e.g.*, by declaring `float2 x; // 64 bits`.

This renders the code at that point serial rather than parallel, a situation called **thread divergence**. On the other hand, threads in the same block but in different warps can diverge with no problem.

Following the hardware, threads in CUDA software follow a hierarchy:

- Each block has its own ID within the grid, consisting of an 'x and a y coordinate. Likewise each thread has x, y and z coordinates within whichever block it belongs to.

- Just as an ordinary CPU thread needs to be able to sense its ID, *e.g.*, by calling `omp_get_thread_num()` in OpenMP, CUDA threads need to do the same. A CUDA thread can access its block ID via the built-in variables `blockIdx.x` and `blockIdx.y`, and can access its thread ID within its block via `threadIdx.x`, `threadIdx.y` and `threadIdx.z`.

- CUDA extends C syntax to allow specifying the grid and block sizes, using structs of type `dim3`, *i.e.* via variables **gridDim** and **blockDim**, with member variables for the various dimensions, *e.g.*, **blockDim.x** for the size of the X dimension for the number of threads per block.

  ```
  dim3 dimGrid(n,1);
  dim3 dimBlock(1,1,1);
  mykernel<<<dimGrid,dimBlock>>>();
  ```

  Here the grid is specified to consist of n ($n \times 1$) blocks, and each block consists of just one ($1 \times 1 \times 1$) thread.

- The "coordinates" of a block within the grid, and of a thread within a block, are merely abstractions. If for instance one is programming computation of heat flow across a two-dimensional slab, the programmer may find it clearer to use two-dimensional IDs for the threads. *But this does not correspond to any physical arrangement in the hardware.*

- Each block in your code is assigned to some SM. It will be tied to that SM during the entire execution of your kernel, though of course it will not constantly be running during that time. If there are more blocks than can be accommodated by all the SMs, then some blocks will need to wait for assignment; when a block finishes, that block's resources, e.g. shared memory, can now be assigned to a waiting block.

- The GPU has a limit on the number of threads that can run on a single block, typically 512, and on the total number of threads running on an SM, 786. If a block contains fewer than 32 threads, only part of the processing power of the SM it's running on will be used. So block size should normally be at least 32. Moreover, for the same reason, block size should ideally be a multiple of 32.

- If the code makes used of shared memory, larger block size may be the better. On the other hand, the larger the block size, the longer the time it will take for barrier synchronization. However, to make use the full power of the GPU, with its many SMs, thus implying a need to use at least as many blocks as there are SMs, which may require smaller blocks.

- Moreover, due to the need for latency hiding in memory access, it is favorable to have lots of warps, so that some will run while others are doing memory access. Two threads doing unrelated work, or the same work but with many `if/elses`, would cause a lot of thread divergence if they were in the same block. A commonly-cited rule of thumb is to have between 128 and 256 threads per block.

9.1.2 Memory Management

Host and device memory are separate entities:

- Device pointers point to GPU memory
  - May be passed to/from host code
  - May not be dereferenced in host code
- Host pointers point to CPU memory
  - May be passed to/from device code
  - May not be dereferenced in device code

There are various types of device memory:

- **Shared memory**: All the *threads in an SM* share this memory, and use it to *communicate among themselves*, just as is the case with threads in CPUs. Access is *very fast*, as this memory is on-chip. It is declared

inside the kernel, or in the kernel call (details below). Shared memory is divided into 16 or 32 banks, in a low-order interleaved manner: words with consecutive addresses are stored in consecutive banks, mod the number of banks, *i.e.*, wrapping back to 0 when hitting the last bank. The best access to shared memory arises when the accesses are to different banks. An exception occurs in *broadcast*, *i.e.*, if all threads in the block wish to read from the same word in the same bank, the word will be sent to all the requestors simultaneously without conflict. However, if only some threads try to read the same word, there may or may not be a conflict, as the hardware chooses a bank for broadcast in some unspecified way.

The biggest performance issue with shared memory is its size, as little as 16K per SM in many GPU cards. This is divided up among the blocks on a given SM, *e.g.*, if we have 4 blocks running on an SM, each one can only use 16K/4 = 4K bytes of shared memory, and the data stored in it are valid only for the life of the currently-executing kernel. Also, shared memory *cannot be accessed by the host*. Note that the term *shared* only refers to the fact that it is shared among threads in the same block.

- **Global memory**: This is *shared by all the threads in an entire application*, and is persistent across kernel calls, throughout the life of the application, i.e. until the program running on the host exits. Global memory is organized into six or eight **partitions**, depending on the GPU model, of 256 bytes each, hence it is usually much larger than shared memory. It is *accessible from the host* and pointers to global memory can be declared outside the kernel.

  On the other hand, global memory is off-chip and *very slow*, taking hundreds of clock cycles per access instead of just a few. This can be ameliorated by exploiting *latency hiding*, *i.e.*, if a warp has requested a global memory access that will take a long time, the hardware will schedule another warp to run while the first is waiting for the memory access to complete; or via hardware actions called *coalescing*, *i.e.*, if the hardware sees that the threads currently accessing global memory are accessing consecutive words, the hardware can execute the memory requests in groups of up to 32 words at a time. This works because the memory is low-order interleaved, and is true for both reads and writes.

The newer GPUs go even further, coalescing much more general access patterns, not just to consecutive words. The programmer may be able to take advantage of coalescing, by a judicious choice of algorithms and/or by inserting padding into arrays.

- **Registers:** Each SM has a set of registers, much more numerous than in a CPU. Access to them is very fast, said to be slightly faster than to shared memory. The compiler normally stores the local variables for a device function in registers, but there are exceptions. An array won't be placed in registers if the array is too large, or if it has variable index values, since registers are not indexable by the hardware.

- **Local memory:** This is physically part of global memory, but is an area within that memory that is allocated by the compiler for a given thread. As such, it is slow, and accessible only by that thread. The compiler allocates this memory for local variables in a device function if the compiler cannot store them in registers. This is called **register spill**.

- **Constant memory:** As the name implies, it's read-only from the device (read/write by the host), for storing values that will not be changed by device code. It is off-chip, thus potentially slow, but has a cache on the chip. At present, the size is 64K. One designates this memory with `__constant__`, as a global variable in the source file. One sets its contents from the host via `cudaMemcpyToSymbol()`.

- **Texture:** This is similar to constant memory, in the sense that it is read-only and cached. The difference is that the caching is two-dimensional. The elements **a[i][j]** and **a[i+1][j]** are far from each other in the global memory, but since they are "close" in a two-dimensional sense, they may reside in the same cache line.

The key implication is that shared memory is used essentially as a programmer-managed cache. Data will start out in global memory, but if a variable is to be accessed multiple times by the GPU code, it is better to copy it to shared memory, and then access the copy instead of the original. If the variable is changed and is to be eventually transmitted back to the host, it has to be copied back to global memory.

CUDA provides functions for handling device memory similar to the C equivalents `malloc()`, `free()`, and `memcpy()`, *i.e.*, `cudaMalloc()`, `cudaFree()` and `cudaMemcpy()`. Copying data between host and device can be a major bottleneck. One way to ameliorate this is to use `cudaMallocHost()` instead of `malloc()` when allocating memory on the host. This sets up page-locked memory, meaning that it cannot be swapped out by the OS' virtual memory system. This allows the use of DMA hardware to do the memory copy, said to make `cudaMemcpy()` twice as fast.

Typically each thread deals with its own portion of the shared data, however, all the threads in a block can read/write any element in shared memory. Shared memory consistency is sequential within a thread, but **relaxed** among threads in a block: A write by one thread is not guaranteed to be visible to the others in a block until `__syncthreads()` is called. On the other hand, writes by a thread *will* be visible to that same thread in subsequent reads without calling `__syncthreads()`. Among the implications of this is that if each thread writes only to portions of shared memory that are not read by other threads in the block, then `__syncthreads()` need not be called. It is also possible to allocate shared memory in the kernel call, along with the block and thread configuration. Here is a simple example:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <cuda.h>
4
5  // example: illustrates kernel-allocated shared memory;
6  // does nothing useful, just copying an array from host
7  // to device global, then to device shared, doubling it
8  // there, then copying back to device global then host.
9
10 __global__ void doubleit(int *dv, int n)
11 {  extern __shared__ int sv[];
12    int me = threadIdx.x;
13    // threads share in copying dv to sv, with each
14    // thread copying one element
15    sv[me] = dv[me];
16    sv[me] = 2 * sv[me];
17    dv[me] = sv[me];
```

```
18 }
19
20 int main(int argc, char **argv)
21 {
22     int n = atoi(argv[1]);  // number of matrix rows/cols
23     int *hv, // host array
24         *dv; // device array
25     int vsize = n * sizeof(int);  // size of array in bytes
26     // allocate space for host array
27     hv = (int *) malloc(vsize);
28     // fill test array with consecutive integers
29     int t = 0,i;
30     for (i = 0; i < n; i++)
31        hv[i] = t++;
32     // allocate space for device array
33     cudaMalloc((void **)&dv,vsize);
34     // copy host array to device array
35     cudaMemcpy(dv,hv,vsize,cudaMemcpyHostToDevice);
36     // set up parameters for threads structure
37     dim3 dimGrid(1,1);
38     dim3 dimBlock(n,1,1); // all n threads in the same block
39     // invoke the kernel; vsize...amount of shared memory
40     doubleit<<<dimGrid,dimBlock,vsize>>>(dv,n);
41     // wait for kernel to finish
42     cudaThreadSynchronize();
43     // copy row array from device to host
44     cudaMemcpy(hv,dv,vsize,cudaMemcpyDeviceToHost);
45     // check results
46     if (n < 10) for(int i=0; i<n; i++) printf("%d\n",hv[i]);
47     // clean up
48     free(hv);
49     cudaFree(dv);
50 }
```

The variable `sv` is kernel allocated, it's declared in `extern __shared__ int sv[];` but actually allocated during `doubleit<<<dimGrid,dimBlock,vsize>>>(dv,n);` in that third argument of the kernel invocation within the chevrons, `vsize`.

Note that one can only directly declare one region of space in this manner. This has two implications:

- Two **__device__** functions, each declared an **extern __shared__** array like this, will occupy the same place in memory!
- Within one **__device__** function, two **extern __shared__** arrays can only share the space via subarrays, e.g.:

  ```
  [fontsize=\relsize{-2}]
  int *x = &sv[120];
  ```

  would set up **x** as a subarray of **sv** above, starting at element 120.

One can also set up shared arrays of fixed length in the same code by declaring them before the variable-length one. In the example above, the array `sv` is syntactically local to the function `doubleit()`, but is shared by all invocations of that function in the block, thus acting "global" to them in a sense. But the point is that it is not accessible from within *other* functions running in that block. In order to achieve the latter situation, a shared array can be declared outside any function.

9.1.3 Synchronization, within and between Blocks

As mentioned earlier, a barrier for the threads in the same block is available by calling `__syncthreads()`. Note carefully that if one thread writes a variable to shared memory and another then reads that variable, one must call this function (from both threads) in order to get the latest value. Keep in mind that within a block, different warps will run at different times, making synchronization vital.

Remember too, that threads across blocks cannot sync with each other in this manner. There are, though, several `atomic` operations—read/modify/write actions that a thread can execute without **pre-emption**, *i.e.*, without interruption—available on both global and shared memory. For example, `atomicAdd()` performs a fetch-and-add operation:

```
atomicAdd(address of integer variable,inc);
```

where `address of integer variable` is the address of the (device) variable to add to, and `inc` is the amount to be added. The return value of the function is the value originally at that address before the operation.

There are also `atomicExch()` (exchange the two operands), `atomicCAS()` (if the first operand equals the second, replace the first operand by the third), `atomicMin()`, `atomicMax()`, `atomicAnd()`, `atomicOr()`, and so on.

Though a barrier could in principle be constructed from the atomic operations, its overhead would be quite high. In earlier models that was near a microsecond, and though that problem has been ameliorated in more recent models, implementing a barrier in this manner would not be not much faster than attaining interblock synchronization by returning to the host and calling `cudaThreadSynchronize()` there. Recall that the latter *is* a possible way to implement a barrier, since global memory stays intact in between kernel calls, but again, it would be slow.

So, what if synchronization is really needed? This is the case, for instance, for iterative algorithms, where all threads must wait at the end of each iteration. For small problems, maybe using just one block can provide satisfactory performance. This might need larger granularity, *i.e.*, more work assigned to each thread, and using just one block means that only one SM is in use, thus only a fraction of the potential power of the machine. Using multiple blocks, though, the only feasible option for synchronization is to rely on returns to the host, where synchronization occurs via `cudaThreadSynchronize()`, causing the situation outlined in the discussion of **constant memory** above.

Finally, CUDA looks like C, it feels like C, and for the most part, it *is* C. But in many ways, it's quite different from C:

- No access to the C library (the library consists of host machine language, after all). There are special versions of math functions, however, e.g. **__sin()**.
- No stack. Functions are essentially inlined, rather than their calls being handled by pushes onto a stack.
- No pointers to functions.

9.2 Hardware Requirements and Compilation

There is a list of suitable NVIDIA video cards at `http://www.nvidia.com/object/cuda_gpus.html`; see also the Wikipedia entry, `http://en.wikipedia.org/wiki/CUDA#Supported_GPUs`.

Compiling `x.cu`, is achieved via `nvcc -g -G x.cu`. The **-g -G** options are for setting up debugging, the first for host code, the second for device code. One may also need to specify `-I/your_CUDA_include_path`, to pick up the file `cuda.h`.

The executable can be run as usual, locally via `./a.out` or in a submission script on a cluster.

One may need to take special action to set the library path properly, *e.g.*, on Linux machines, the environment variable `LD_LIBRARY_PATH` to include the CUDA library.

The following code can be used to determine the limits, *e.g.*, maximum number of threads, of the device 0, assuming there is only one device. The return value of `cudaGetDeviceProperties()` is a complex C struct whose components are listed at `http://docs.nvidia.com/cuda/`.

```
1  #include<cuda.h>
2  #include <stdio.h>
3
4  int main()
5  {
6     cudaDeviceProp Props;
7     cudaGetDeviceProperties( &Props,0);
8
9     printf("shared mem: %d)\n", Props.sharedMemPerBlock);
10    printf("max threads/block: %d\n",Props.maxThreadsPerBlock);
11    printf("max blocks: %d\n",Props.maxGridSize[0]);
12    printf("total Const mem: %d\n",Props.totalConstMem);
13 }
```

9.3 Hello World! for CUDA - the real thing!

`http://computer-graphics.se/hello-world-for-cuda.html` provides an example of the *Hello World!* program by Ingemar Ragnemalm, making use of a kernel adding array elements to produce the string "World!".

```
1  // This is the REAL "hello world" for CUDA!
2  // It takes the string "Hello ", prints it, then passes
3  // it to CUDA with an array of offsets. Then the offsets
4  // are added in parallel to produce the string "World!"
5  // By Ingemar Ragnemalm 2010
6
7  #include <stdio.h>
8
9  const int N = 16;
10 const int blocksize = 16;
11
12 __global__
13 void hello(char *a, int *b)
14 {
15         a[threadIdx.x] += b[threadIdx.x];
16 }
17
18 int main()
19 {
20         char a[N] = "Hello \0\0\0\0\0\0";
21         int b[N] = {15,10,6,0,-11,1,0,0,0,0,0,0,0,0,0,0};
22         char *ad;
23         int *bd;
24         const int csize = N*sizeof(char);
25         const int isize = N*sizeof(int);
26
27         printf("%s", a);
28         cudaMalloc( (void**)&ad, csize );
29         cudaMalloc( (void**)&bd, isize );
30         cudaMemcpy(ad,a,csize,cudaMemcpyHostToDevice);
31         cudaMemcpy(bd,b,isize,cudaMemcpyHostToDevice);
32         dim3 dimBlock( blocksize, 1 );
```

```
33          dim3 dimGrid( 1, 1 );
34          hello<<<dimGrid,dimBlock>>>(ad, bd);
35          cudaMemcpy(a,ad,csize,cudaMemcpyDeviceToHost);
36          cudaFree( ad );
37          cudaFree( bd );
38          printf("%s\n", a);
39          return EXIT_SUCCESS;
40  }
```

9.4 Examples

The following examples are taken from `http://heather.cs.ucdavis.edu/~matloff/158/PLN/ParProcBook.pdf`.

9.4.1 Finding Cumulative Sums

The following code computes cumulative sums for the special case of a single block. For instance, if the original array is (3,1,2,0,3,0,1,2), then it is changed to (3,4,6,6,9,9,10,12). The general plan is for each thread to operate on one chunk of the array. A thread will find cumulative sums for its chunk, and then adjust them based on the high values of the chunks that precede it. In the above example, for instance, using 4 threads, the threads will first produce (3,4), (2,2), (3,3) and (1,3). Since thread 0 found a cumulative sum of 4 in the end, thread 1 adds 4 to each element of (2,2), yielding (6,6). Thread 1 had found a cumulative sum of 2 in the end, which together with the 4 found by thread 0 makes 6. Thus thread 2 must add 6 to each of its elements, *i.e.* add 6 to (3,3), yielding (9,9). The case of thread 3 is similar.

```
1  // for this simple illustration, it is assumed that the
2  // code runs in just one block, and that the number of
3  // threads evenly divides n.
4  // some improvements that could be made:
5  //   1. change to multiple blocks, to try to use all SMs
6  //   2. possibly use shared memory
7  //   3. have each thread work on staggered elements of dx,
8  //      rather than on contiguous ones, to get more
9  //      efficient bank access
```

```
10 #include <cuda.h>
11 #include <stdio.h>
12
13 __global__ void cumulker(int *dx, int n)
14 {
15    int me = threadIdx.x;
16    int csize  = n / blockDim.x;
17    int start = me * csize;
18    int i,j,base;
19    for (i = 1; i < csize; i++) {
20       j = start + i;
21       dx[j] = dx[j-1] + dx[j];
22    }
23    __syncthreads();
24    if (me > 0) {
25       base = 0;
26       for (j = 0; j < me; j++)
27          base += dx[(j+1)*csize-1];
28    }
29    __syncthreads();
30    if (me > 0) {
31       for (i = start; i < start + csize; i++)
32          dx[i] += base;
33    }
34 }
```

9.4.2 Calculate Row Sums

In the following program, each thread will handle one row of a n x n matrix, stored in one-dimensional form in row-major order, so the loop

```
for (int k = 0; k < n; k++)
   sum += m[rownum*n+k];
```

will traverse the n elements of row number `rownum=blockIdx.x`, and compute their sum. That sum is then placed in the proper element of the output array: `rs[rownum] = sum;`

```
1  // CUDA example: finds row sums of an integer matrix m
2  // find1elt() finds the rowsum of one row of the n x n
3  // matrix m, storing the result in the corresponding
4  // position in the rowsum array rs; matrix
5  // stored as 1-dimensional, row-major order
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <cuda.h>
9
10 __global__ void find1elt(int *m, int *rs, int n)
11 {
12    // this thread will handle row # rownum
13    int rownum = blockIdx.x;
14    int sum = 0;
15    for (int k = 0; k < n; k++)
16       sum += m[rownum*n+k];
17    rs[rownum] = sum;
18 }
19
20 int main(int argc, char **argv)
21 {
22     // number of matrix rows/cols
23     int n = atoi(argv[1]);
24     int *hm, // host matrix
25         *dm, // device matrix
26         *hrs, // host rowsums
27         *drs; // device rowsums
28     // size of matrix in bytes
29     int msize = n * n * sizeof(int);
30     // allocate space for host matrix
31     hm = (int *) malloc(msize);
32     // as a test, fill matrix with consecutive integers
33     int t = 0,i,j;
34     for (i = 0; i < n; i++) {
35        for (j = 0; j < n; j++) {
36           hm[i*n+j] = t++;
37        }
38     }
```

```
39      // allocate space for device matrix
40      cudaMalloc((void **)&dm, msize);
41      // copy host matrix to device matrix
42      cudaMemcpy(dm,hm, msize,cudaMemcpyHostToDevice);
43      // allocate host, device rowsum arrays
44      int rssize = n * sizeof(int);
45      hrs = (int *) malloc(rssize);
46      cudaMalloc((void **)&drs,rssize);
47      // set up parameters for threads structure
48      dim3 dimGrid(n,1);  // n blocks
49      dim3 dimBlock(1,1,1);  // 1 thread per block
50      // invoke the kernel
51      find1elt<<<dimGrid,dimBlock>>>(dm,drs,n);
52      // wait for kernel to finish
53      cudaThreadSynchronize();
54      // copy row vector from device to host
55      cudaMemcpy(hrs,drs,rssize,cudaMemcpyDeviceToHost);
56      // check results
57      if (n < 10) for(int i=0; i<n; i++)
58          printf("%d\n",hrs[i]);
59      // clean up
60      free(hm);
61      cudaFree(dm);
62      free(hrs);
63      cudaFree(drs);
64  }
```

9.4.3 Finding Prime Numbers

The code below finds all the prime numbers from 2 to **n**. This code has been designed with some thought as to memory speed and thread divergence. The code uses the classical Sieve of Erathosthenes, "crossing out" multiples of 2, 3, 5, 7 and so on to get rid of all the composite numbers. Using just two threads, say A and B, thread A deals with only some multiples of 19 and B handles the others for 19. Then they both handle their own portions of multiples of 23, and so on. The thinking here is that the second version will be more amenable to lockstep execution, thus causing less thread divergence. Thus, each thread handles a chunk of multiples of the given prime.

In order to enhance memory performance, this code uses device shared memory. All the "crossing out" is done in the shared memory array `sprimes`, and then when we are all done, that is copied to the device global memory array `dprimes`, which is in turn copies to host memory. Note that the amount of shared memory here is determined dynamically.

```
1  // CUDA example: illustration of shared memory alloca-
2  // tion at run time; finds primes using classical Sieve
3  // of Erathosthenes: make list of numbers 2 to n, then
4  // cross out all multiples of 2 (but not 2 itself), then
5  // all multiples of 3, etc.; whatever is left over is
6  // prime; in our array, 1 will mean "not crossed out"
7  // and 0 will mean "crossed out"
8  // IMPORTANT NOTE: uses shared memory, in a single block,
9  // without rotating parts of array in and out of shared
10 // memory; thus limited to n <= 4000 for 16K shared memory
11 #include <stdio.h>
12 #include <stdlib.h>
13 #include <cuda.h>
14 // initialize sprimes, 1s for the odds, 0s for the evens;
15 // see sieve(), for the nature of the arguments
16 __device__ void initsp(int *sprimes,int n,int nth,int me)
17 {
18    int chunk,startsetsp,endsetsp,val,i;
19    sprimes[2] = 1;
20    // determine sprimes chunk for this thread to init
21    chunk = (n-1) / nth;
22    startsetsp = 2 + me*chunk;
23    if (me < nth-1) endsetsp = startsetsp + chunk - 1;
24    else endsetsp = n;
25    // now do the init
26    val = startsetsp % 2;
27    for (i = startsetsp; i <= endsetsp; i++) {
28       sprimes[i] = val;
29       val = 1 - val;
30    }
31    // make sure sprimes up to date for all
32    __syncthreads();
```

```
33 }
34 // copy sprimes back to device global memory; see sieve()
35 // for the nature of the arguments
36 __device__ void cpytoglb(int *dprimes,int *sprimes,int n,\\
37   int nth,int me)
38 {
39    int startcpy,endcpy,chunk,i;
40    chunk = (n-1) / nth;
41    startcpy = 2 + me*chunk;
42    if (me < nth-1) endcpy = startcpy + chunk - 1;
43    else endcpy = n;
44    for (i=startcpy;i<=endcpy;i++) dprimes[i]=sprimes[i];
45    __syncthreads();
46 }
47 // finds primes from 2 to n, storing the information
48 // in dprimes, with dprimes[i] being 1 if i is prime, 0 if
49 // composite; nth is the number of threads
50 __global__ void sieve(int *dprimes, int n, int nth)
51 {
52    extern __shared__ int sprimes[];
53    int me = threadIdx.x;
54    int nth1 = nth - 1;
55    // initialize sprimes array, 1s for odds, 0 for evens
56    initsp(sprimes,n,nth,me);
57    // "cross out" multiples of various numbers m, with each
58    // thread doing a chunk of m's; always check first to
59    // determine whether m has already been found to be
60    // composite; finish when m*m > n
61    int maxmult,m,startmult,endmult,chunk,i;
62    for (m = 3; m*m <= n; m++) {
63       if (sprimes[m] != 0) {
64          // find largest multiple of m that is <= n
65          maxmult = n / m;
66          // now partition 2,3,...,maxmult among the threads
67          chunk = (maxmult - 1) / nth;
68          startmult = 2 + me*chunk;
69          if (me < nth1) endmult = startmult + chunk - 1;
70          else endmult = maxmult;
```

```
71         }
72         // OK, cross out my chunk
73         for (i=startmult;i<=endmult;i++) sprimes[i*m]=0;
74      }
75      __syncthreads();
76      // copy back to device global memory for return to host
77      cpytoglb(dprimes,sprimes,n,nth,me);
78  }
79  int main(int argc, char **argv)
80  {
81     int n = atoi(argv[1]), // will find primes among 1,...,n
82         nth = atoi(argv[2]);  // number of threads
83     int *hprimes,  // host primes list
84         *dprimes;  // device primes list
85     // size of primes lists in bytes
86     int psize = (n+1) * sizeof(int);
87     // allocate space for host list
88     hprimes = (int *) malloc(psize);
89     // allocate space for device list
90     cudaMalloc((void **)&dprimes,psize);
91     dim3 dimGrid(1,1);
92     dim3 dimBlock(nth,1,1);
93     // invoke the kernel, and allocate shared memory
94     sieve<<<dimGrid,dimBlock,psize>>>(dprimes,n,nth);
95     // check whether we asked for too much shared memory
96     cudaError_t err = cudaGetLastError();
97     if(err != cudaSuccess) printf("%s\n",cudaGetErrorString(err));
98     // wait for kernel to finish
99     cudaThreadSynchronize();
100    // copy list from device to host
101    cudaMemcpy(hprimes,dprimes,psize,cudaMemcpyDeviceToHost);
102    // check results
103    if (n <= 1000) for(int i=2; i<=n; i++)
104       if (hprimes[i] == 1) printf("%d\n",i);
105    // clean up
106    free(hprimes);
107    cudaFree(dprimes);
108 }
```

9.5 GPU Accelerated Lanczos Algorithm with Applications

In this paper, `http://ieeexplore.ieee.org/document/5763440`, an implementation of Lanczos Algorithm (see Section 6) on GPU is presented using the CUDA programming model and applied to two important problems: graph bisection using spectral methods, and image segmentation.

In mathematics, the graph partition problem is defined on data represented in the form of a graph $G = (V, E)$, with V vertices and E edges, such that it is possible to partition G into smaller components with specific properties. A bisection of a graph is a bipartition of its vertex set in which the number of vertices in the two parts differ by at most 1, and its size is the number of edges which go across the two parts. In computer vision, image segmentation is the process of partitioning a digital image into multiple segments (sets of pixels, also known as super-pixels). The goal of segmentation is to simplify and/or change the representation of an image into something that is more meaningful and easier to analyze.

Comparison experiments were run on the following systems:

- **CPU**: An Intel Core i7 920, with 8 MB cache, 4 GB RAM and a 4.8 GT/s Quick path interface, maximum memory bandwidth of 25 GB/s.
- **GPU**: A Tesla C1060 which is one quarter of a Tesla S1070 computing system with 4 GB memory and 102 GB/s memory bandwidth, attached to a Intel Core i7 CPU, running CUDA Toolkit/SDK version 2.2.

For the experiments on the graph bisection using spectral methods the graphs from the Walshaw benchmark, which is a popular graph partitioning archive, were taken. The GPU implementation of spectral bisection performs better when compared to both an Intel Math Kernel Library implementation and a Matlab implementation. It shows a speedup up to 97.3 times over Matlab Implementation and 2.89 times over the Intel Math Kernel Library implementation on a Intel Core i7 920 Processor, which is a quad-core CPU.

Similarly, the image segmentation implementation achieves a speed up of 3.27 compared to a multicore CPU based implementation using Intel Math Kernel Library and OpenMP.

10 Makefile Example

Compiling the source code files can be tiring, especially when you have to include several source files and type the compiling command every time you need to compile. Makefiles are the solution to simplify this task.

Makefiles are special format files that help build and manage the projects automatically. For example, let?s assume we have the following source files `main.c, hello.c, factorial.c` and `functions.h`. The latter contains all the function declarations and should be referenced in all the `*.c` files via

```
#include "functions.h"
```

In this example we have only four files and we know the sequence of the function calls. However, for a large project where we have thousands of source code files, it becomes difficult to maintain the binary builds. Moreover, you notice that you usually only work on a small section of the program (such as a single function), and much of the remaining program is unchanged and does not have to be recompiled.

This is an example of the Makefile for compiling the hello program.

```
1  # Define required macros here
2  SHELL = /bin/sh
3
4  OBJS =  main.o factorial.o hello.o
5  CFLAG = -Wall -g
6  CC = /cluster/mpi/openmpi/1.6.5-gcc4.8.2/bin/mpicc
7  INCLUDE =
8  LIBS = -lm
9
10 hello:${OBJ}
11     ${CC} ${CFLAGS} ${INCLUDES} -o $@ ${OBJS} ${LIBS}
12
13 clean:
14     -rm -f *.o core *.core
```

Now you can build your program hello using the "make". If you will issue a command 'make clean' then it removes all the object files and core files in the current directory.

Here is a nice tutorial for how to build your own Makefile `https://www.tutorialspoint.com/makefile/makefile_quick_guide.htm`

11 A note on Monte Carlo simulations of a scalar field

In statistical mechanics of systems in thermal equilibrium one is interested in the computation of ensemble averages. In the case discussed here, an ensemble average corresponds to the integration over fields $\Phi = \{\Phi_x\}$ which are defined on a lattice (or grid) of points x. This integration consists of $\mathrm{O}(10^6)$ or more integrals. Its solution will be estimated by means of a statistical simulation method called Monte Carlo simulation.

11.1 The model

Consider a d-dimensional lattice Λ with lattice spacing a. The points on the lattice have coordinates which are integer multiples of the lattice spacing:

$$x = (x_0, x_1, \ldots, x_{d-1}) = (n_0, n_1, \ldots, n_{d-1})a\,. \tag{3}$$

The lattice is Euclidean. Then $|x-y|^2 = \sum_{\mu=0}^{d-1}(x_\mu - y_\mu)^2$ is the distance squared between the points x and y. Displacements on the lattice are defined in terms of unit vectors $\hat{\mu}$ as

$$x + a\hat{\mu} = (x_0, \ldots, x_\mu + a, \ldots, x_{d-1})\,. \tag{4}$$

A complex scalar field $\Phi = \{\Phi_x\}$ assigns to each point x of the lattice a variable $\Phi_x \in \mathbb{C}$. Fig. 1 shows a two-dimensional example.

We will consider a finite box of size $L_0 \times L_1 \times \ldots \times L_{d-1}$. Then the integer coordinates are in the range

$$n_\mu = 0, 1, \ldots, \frac{L_\mu}{a} - 1\,, \quad \mu = 0, 1, \ldots, d-1\,. \tag{5}$$

We have to specify boundary conditions. We choose periodic boundary conditions defined by

$$x + L_\mu\hat{\mu} = x\,, \quad \mu = 0, 1, \ldots, d-1\,. \tag{6}$$

They correspond to the addition of the integer coordinates n_μ modulo L_μ/a and they are illustrated in Fig. 1.

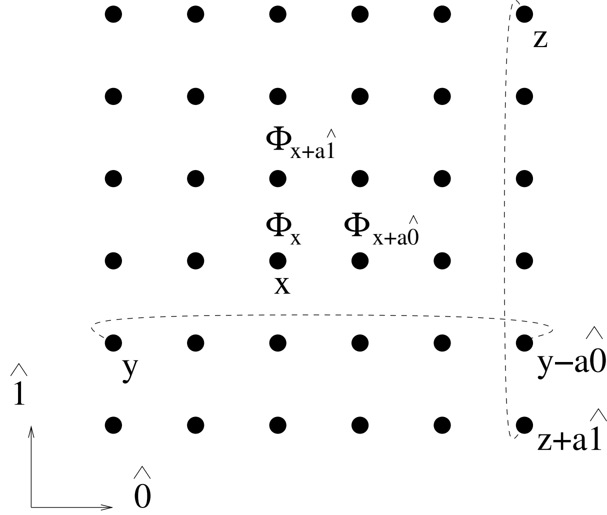


Figure 1: Example of a two-dimensional lattice with a complex field Φ.

A complex scalar field is a model for a ferromagnet, where the field values Φ_x describe the spin (or the magnetic moment associated with the spin). The action or energy function of such a system is given by

$$S[\Phi, H] = \sum_{x\in\Lambda} \Bigg[|\Phi_x|^2 + \lambda \left(|\Phi_x|^2 - 1\right)^2 - \kappa \sum_{\mu=0}^{d-1} \left(\Phi_x^* \Phi_{x+a\hat{\mu}} + \Phi_{x+a\hat{\mu}}^* \Phi_x\right) - \left(H_x^* \Phi_x + \Phi_x^* H_x\right) \Bigg] . \quad (7)$$

Here the * means complex conjugation. The field $H = \{H_x\}$ is a space-dependent external magnetic field with $H_x \in \mathbb{C}$. The real parameter κ determines the coupling strength between nearest-neighbor spins. The real and positive parameter $\lambda > 0$ is a local weight. The canonical ensemble is defined by the partition function

$$Z[H] = \int \mathcal{D}[\Phi] \; \mathrm{e}^{-S[\Phi,H]} , \quad (8)$$

where $\mathcal{D}[\Phi] = \prod_{x\in\Lambda} \mathrm{dRe}\Phi_x \; \mathrm{dIm}\Phi_x$. The system has the following limiting cases depending on the value of λ:

- $\lambda = 0$: S is a quadratic form of the spins Φ_x. The model is defined only when the parameter κ satisfies the condition $2d|\kappa| < 1$. It guarantees

that for real constant fields $\Phi_x = c$ or purely imaginary constant fields $\Phi_x = ic$ the quadratic form is positive. This implies that the integral in Eq. (8) is finite. For $\lambda = 0$ the system can be solved exactly and it is called the *Gaussian model*.

- $\lambda = \infty$: this forces $|\Phi_x| = 1$, i.e. the spins are restricted to the unit circle. This is called the *XY-model*.

The general case $\lambda > 0$ interpolates between these two limiting cases.

The action Eq. (7) can be equivalently written using a real notation. We define $\Phi_x = \phi_{x,1} + i\phi_{x,2}$ where $\phi_{x,1} \in \mathbb{R}$ and $\phi_{x,2} \in \mathbb{R}$ are the real and imaginary parts of Φ_x respectively. Similarly, $H_x = h_{x,1} + ih_{x,2}$ with $h_{x,i} \in \mathbb{R}$, $i = 1, 2$. We construct the two-dimensional real vectors

$$\phi_x = \begin{pmatrix} \phi_{x,1} \\ \phi_{x,2} \end{pmatrix} \text{ and } h_x = \begin{pmatrix} h_{x,1} \\ h_{x,2} \end{pmatrix}, \tag{9}$$

in terms of which the action Eq. (7) becomes

$$S[\phi, h] = \sum_{x\in\Lambda} \left[\phi_x^T \phi_x + \lambda \left(\phi_x^T \phi_x - 1 \right)^2 - 2\kappa \sum_{\mu=0}^{d-1} \phi_x^T \phi_{x+a\hat{\mu}} - 2h_x^T \phi_x \right]. \tag{10}$$

The notation $h_x^T \phi_x = \sum_{i=1}^{2} h_{x,i}\phi_{x,i}$ denotes the scalar product. The partition function is

$$Z[h] = \int \mathcal{D}[\phi] \; \mathrm{e}^{-S[\phi,h]}, \quad \mathcal{D}[\Phi] = \prod_{x\in\Lambda} \underbrace{\mathrm{d}\phi_{x,1}\,\mathrm{d}\phi_{x,2}}_{\mathrm{d}^2\phi_x}. \tag{11}$$

In the following we will use the summation notations $\sum_x \equiv \sum_{x\in\Lambda}$ and $\sum_\mu \equiv \sum_{\mu=0}^{d-1}$.

11.2 Statistical simulations

An observable is a function $A[\Phi]$ of the field Φ. Examples are the energy $A[\Phi] = S[\Phi, H]$ and the magnetization $A[\Phi] = m = \frac{1}{V}\sum_x \Phi_x$, where V denotes the number of lattice points. Another example is $A[\Phi] = \sum_x |\Phi_x|^2$ etc. . The *ensemble average* or *expectation value* of an observable A is defined by

$$\langle A \rangle_H = \frac{1}{Z[H]} \int \mathcal{D}[\Phi] \; A[\Phi] \mathrm{e}^{-S[\Phi,H]}. \tag{12}$$

The factor $e^{-S[\Phi,H]}$ in the integrand is called Boltzmann weight. Typically it is a function peaked around a small subset of the fields whose contribution dominates the ensemble average. The goal of a stochastic simulation is to interpret the Boltzmann weight (normalized by the partition function) as a probability distribution and to construct a sample of the space of field configurations (or phase space) $\{\Phi^{(1)}, \Phi^{(2)}, \ldots\}$ where the frequency of appearance of a given field configuration Φ in the sample is proportional to $e^{-S[\Phi,H]}$. This is called *importance sampling*. The ensemble average can be approximated by the sample average

$$\langle A \rangle_H \approx \frac{1}{N} \sum_{n=1}^{N} A[\Phi^{(n)}] \,, \tag{13}$$

where N is the number of field configurations in the sample. Later we will discuss the statistical error of Eq. (13).

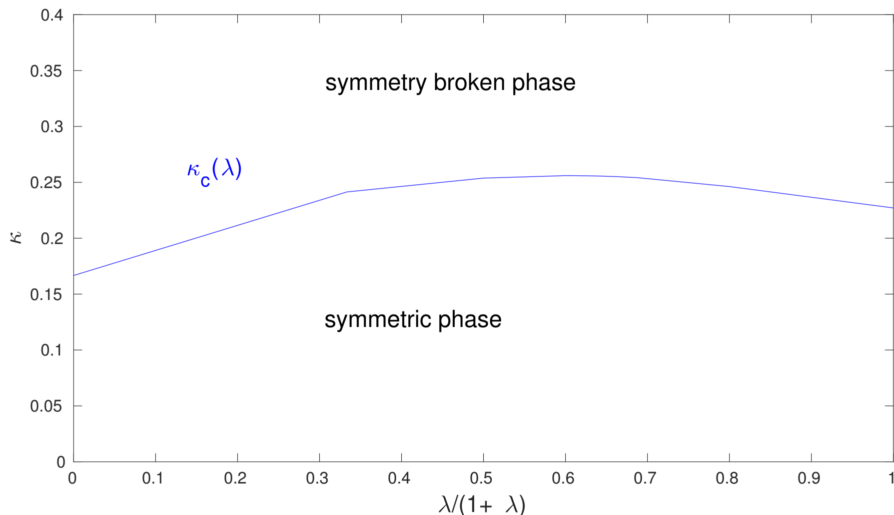


Figure 2: The phase diagram of the three-dimensional two-component ϕ^4 model.

The phase diagram of the model in the plane of the couplings λ and κ for $d > 1$ dimensions looks similar to the one shown in Fig. 2. Fig. 2 shows the data for the three-dimensional model from Ref. [5]. The external magnetic field is set to zero. There are two phases, the symmetric phase and the symmetry broken phase which are separated by a line $\kappa = \kappa_c(\lambda)$. The symmetry

in question is defined by global $O(2)$ rotations of the components of ϕ, under which the action Eq. (10) is invariant. In the symmetric phase for $\kappa < \kappa_c(\lambda)$ the magnetization vanishes: $\langle m \rangle = 0$. In the symmetry broken phase for $\kappa > \kappa_c(\lambda)$ the magnetization is nonzero: $\langle m \rangle \neq 0$. The magnetization can be defined using the external field h as $m = \lim_{h \to 0} \langle \phi_{0,2} \rangle_h$ [4]. A nonzero magnetization means that the $O(2)$ symmetry is spontaneously broken since the magnetization is not $O(2)$ invariant. The line $\kappa = \kappa_c(\lambda)$ is a line of second order phase transitions. For $0 < \lambda \leq \infty$ these phase transitions are in the universality class of the XY-model [5]. $\lambda = 0$ is the Gaussian model.

11.3 Markov-chain

References for this section are [6–8]. A direct construction of field configurations distributed according to the probability density

$$\Pi[\Phi] \quad = \quad \frac{1}{Z[H]} \mathrm{e}^{-S[\Phi,H]} \tag{14}$$

is usually not possible. This is due to the many variables which are coupled together and, as a consequence, to the fact that we do not know the normalization $Z[H]$. Therefore a different algorithm is used which is based on a *Markov chain*

$$\Phi^{(0)} \longrightarrow \Phi^{(1)} \longrightarrow \Phi^{(2)} \longrightarrow \ldots \tag{15}$$

and does not require the knowledge of $Z[H]$. The transition $\Phi^{(i)} \longrightarrow \Phi^{(i+1)}$ occurs with a *transition probability function* $T(\Phi, \Phi')$ which is defined by the following properties

$$\begin{aligned}
T(\Phi, \Phi') &\geq 0 \qquad \text{(P1)} \\
\int \mathcal{D}[\Phi'] \; T(\Phi, \Phi') &= 1 \qquad \text{(P2)} \\
\int \mathcal{D}[\Phi] \; \Pi[\Phi] T(\Phi, \Phi') &= \Pi[\Phi'] \qquad \text{(P3)} \\
\exists n \text{ such that } T^n(\Phi, \Phi') &> 0 \forall \Phi \,,\ \Phi' . \text{ Moreover } T(\Phi, \Phi) > 0 \,. \qquad \text{(P4)}
\end{aligned}$$

Properties P1 and P2 state that $T(\Phi, \Phi')$ is a normalized probability density for Φ' (for any given Φ).

Property P3 is called *stability*. Stability guarantees that, if the Markov chain reaches the *equilibrium* distribution $\Pi[\Phi]$, consequent updates will preserve this property. One says that the Markov chain has reached equilibrium. Notice that the normalization $Z[H]$ drops out in the stability equation (P3). In practical applications, a condition called *detailed balance* is often used, which is sufficient but not necessary for stability

$$\Pi[\Phi]T(\Phi,\Phi') \;=\; \Pi[\Phi']T(\Phi',\Phi) \quad \forall\Phi\,,\ \Phi'\,. \tag{16}$$

The proof that detailed balance implies stability (but not vice versa) is

$$\begin{aligned}\int\mathcal{D}[\Phi]\;\;\Pi[\Phi]T(\Phi,\Phi') &= \int\mathcal{D}[\Phi]\;\;\Pi[\Phi']T(\Phi',\Phi)\\ &= \Pi[\Phi']\int\mathcal{D}[\Phi]\;\;T(\Phi',\Phi)\\ &= \Pi[\Phi']\,. \end{aligned}\tag{17}$$

In the first equality we used detailed balance and in the third property P2.

Property P4 guarantees that the whole phase space can be reached in the Markov chain. It is called *ergodicity* and states that for each pair of field configurations Φ and Φ' there is a finite number n of applications of the transition function such that the transition $\Phi \longrightarrow \Phi'$ occurs with a nonzero probability. The many-step transition function is defined by convolution, e.g.

$$T^2(\Phi,\Phi') \;=\; \int\mathcal{D}[\Phi'']\;\;T(\Phi,\Phi'')T(\Phi'',\Phi')\,. \tag{18}$$

The property $T(\Phi,\Phi) > 0$ is called *aperiodicity* and guarantees that the Markov chain does not get trapped in cycles.

The properties of the Markov chain guarantee that starting from an initial ensemble of fields distributed according to an arbitrary density $\Pi_0[\Phi]$, equilibrium will be reached, i.e.

$$\lim_{k\to\infty}\int\mathcal{D}[\Phi]\;\;\Pi_0[\Phi]T^k(\Phi,\Phi') \;=\; \Pi[\Phi']\,. \tag{19}$$

This is a consequence of the theorem of Perron–Frobenius. We can think of $T(\Phi,\Phi')$ as a matrix in phase space and the field configurations Φ and Φ' correspond to the matrix indices. The matrix is normalized (the row sums are one due to P2) and we can assume that all its elements are strictly positive

(due to ergodicity this can be always achieved by considering a many-step transition function). The theorem of Perron–Frobenius states that such a matrix has an eigenvalue equal to 1 which is non-degenerate and all other eigenvalues have modulus strictly smaller than 1. The stability condition $\Pi T = \Pi$ (in matrix notation) means that the left eigenvector of the eigenvalue 1 is the equilibrium distribution $\Pi[\Phi]$.

Another consequence of the theorem of Perron–Frobenius is that in the Markov chain the relaxation of the field ensemble towards equilibrium is governed by the second largest eigenvalue of the transition function matrix. In order to illustrate this point consider the following toy-model. A system with two states has the transition function

$$T = \begin{pmatrix} 1-\kappa_1 & \kappa_1 \\ \kappa_2 & 1-\kappa_2 \end{pmatrix} \text{ with } 0 < \kappa_1, \kappa_2 < 1\,, \tag{20}$$

where $T_{ij} \equiv T(i,j)$ is the probability for the system to hop to state j if it is currently in state i $(i,j = 1,2)$. The eigenvalues of T are 1 and $\lambda = 1-\kappa_1-\kappa_2$ and indeed $|\lambda| < 1$. $\Pi = \frac{1}{\kappa_1+\kappa_2}(\kappa_2 \;\; \kappa_1)$ is the left eigenvector with eigenvalue 1. It is the equilibrium probability density, according to which state 1 is realized with probability $\frac{\kappa_2}{\kappa_1+\kappa_2}$ and state 2 is realized with probability $\frac{\kappa_1}{\kappa_1+\kappa_2}$. Using the decomposition $T = SDS^{-1}$, where D is a diagonal matrix with the eigenvalues 1 and λ on the diagonal, we can compute

$$T^n = SD^nS^{-1} = \frac{1}{\kappa_1+\kappa_2}\begin{pmatrix} \kappa_2 & \kappa_1 \\ \kappa_2 & \kappa_1 \end{pmatrix} + \frac{\lambda^n}{\kappa_1+\kappa_2}\begin{pmatrix} \kappa_1 & -\kappa_1 \\ -\kappa_2 & \kappa_2 \end{pmatrix} . \tag{21}$$

The first term in Eq. (21) is the projection operator onto the eingenvector corresponding to the equilibrium density. The second determines the rate of approach to equilibrium, since $\lim_{n\to\infty} \lambda^n = 0$ and represents the memory of the Markov chain. We define the *exponential autocorrelation time*

$$\tau_{\mathrm{exp}} = \frac{-1}{\ln(\lambda)} \simeq \frac{1}{1-\lambda} \tag{22}$$

in terms of which $\lambda^n = \mathrm{e}^{-n/\tau_{\mathrm{exp}}}$. States along the Markov chain are correlated. The correlations die off exponentially as the number of steps n grows at a rate determined by τ_{exp}. In general, consider two different Markov chain algorithms which have the same equilibrium density. The better algorithm will have the smaller second largest eigenvalue λ, $|\lambda| < 1$. A perfect "heat bath" alogorithm has all eigenvalues of the transition matrix equal to zero except the eigenvalue 1 (independent sampling with no autocorrelations) but it can be constructed only in special cases.

11.4 Local updates

It is not easy to construct a transition function for a totally new field configuration Φ' which fulfills properties P1 to P4. Instead we can define a transition function for changing a single field variable Φ_x and then repeat this change sweeping through all the lattice points x. The change of a single variable is called a local update.

If all field variables Φ_y for $y \neq x$ are kept fixed, the equilibrium probability density for the field variable Φ_x is given by

$$\pi(\Phi_x) \quad \sim \quad \mathrm{e}^{B_x^* \Phi_x + B_x \Phi_x^* - |\Phi_x|^2 - \lambda \left(|\Phi_x|^2 - 1\right)^2} , \tag{23}$$

where

$$B_x \quad = \quad H_x + \kappa \sum_{\mu} (\Phi_{x+a\hat{\mu}} + \Phi_{x-a\hat{\mu}}) \tag{24}$$

is called the "local magnetic field". In terms of the real components, see Eq. (9), the probability Eq. (23) can be written as

$$\pi(\phi_x) \quad \sim \quad \mathrm{e}^{-\left[(\phi_x - b_x)^T (\phi_x - b_x) + \lambda (\phi_x^T \phi_x - 1)^2\right]} , \tag{25}$$

where $b_x^T = (b_1 = \mathrm{Re} B_x \,,\ b_2 = \mathrm{Im} B_x)$.

A local update of Φ_x can be constructed with the *Metropolis algorithm.* A symmetric proposal for a change $\Phi_x \to \Phi'_x$ is made according to the probability $t_{\mathrm{trial}}(\Phi_x, \Phi'_x)$. Symmetric means $t_{\mathrm{trial}}(\Phi_x, \Phi'_x) = t_{\mathrm{trial}}(\Phi'_x, \Phi_x)$. The proposal is accepted with the probability

$$p_{\mathrm{acpt}}(\Phi_x, \Phi'_x) \quad = \quad \min \left\{ 1, \frac{\pi(\Phi'_x)}{\pi(\Phi_x)} \right\} . \tag{26}$$

The transition probability function for this update is

$$t_{\mathrm{metro}}(\Phi_x, \Phi'_x) \quad = \quad t_{\mathrm{trial}}(\Phi_x, \Phi'_x) p_{\mathrm{acpt}}(\Phi_x, \Phi'_x) + [1 - A(\Phi_x)] \delta(\Phi_x - \Phi'_x) , \tag{27}$$

where

$$A(\Phi_x) \quad = \quad \int \mathrm{dRe}\Phi'_x \, \mathrm{dIm}\Phi'_x \;\; t_{\mathrm{trial}}(\Phi_x, \Phi'_x) p_{\mathrm{acpt}}(\Phi_x, \Phi'_x) . \tag{28}$$

$A(\Phi_x)$ is the probability that the local variable Φ_x is set to a new value and is called the *acceptance*. If the proposal is rejected, we keep the old variable $\Phi'_x = \Phi_x$. The rejection happens with a probability $[1 - A(\Phi_x)]$. The formula Eq. (28) is derived from the normalization condition

$$\int \mathrm{dRe}\Phi'_x \, \mathrm{dIm}\Phi'_x \;\; t_{\mathrm{metro}}(\Phi_x, \Phi'_x) \;\; = \;\; 1 \, . \tag{29}$$

The Metropolis update fulfills detailed balance. The proof goes as follows:

$$\begin{aligned} \pi(\Phi_x) t_{\mathrm{metro}}(\Phi_x, \Phi'_x) \;\; &= \;\; t_{\mathrm{trial}}(\Phi_x, \Phi'_x)\pi(\Phi_x) p_{\mathrm{acpt}}(\Phi_x, \Phi'_x) + \\ & \quad\;\; \pi(\Phi_x)[1 - A(\Phi_x)]\delta(\Phi_x - \Phi'_x) \\ &= \;\; t_{\mathrm{trial}}(\Phi_x, \Phi'_x) \min\{\pi(\Phi_x), \pi(\Phi'_x)\} + \\ & \quad\;\; \pi(\Phi_x)[1 - A(\Phi_x)]\delta(\Phi_x - \Phi'_x) \, . \end{aligned} \tag{30}$$

The expression Eq. (30) is manifestly invariant under exchange of Φ_x and Φ'_x (due to the symmetry of the proposal).

11.4.1 Simple Metropolis update

The proposal

$$\Phi'_x \;\; = \;\; \Phi_x + r_1 + i r_2 \, , \tag{31}$$

where r_1 and r_2 are real random numbers uniformly distributed in the interval $[-\delta, \delta]$, is symmetric. In combination with the acceptance-rejection step Eq. (26) it defines a valid Metropolis algorithm.

11.5 Hybrid overrelaxation updates

We use the notation $\phi \equiv \phi_x$ and $b \equiv b_x$. We write Eq. (25) as

$$\pi(\phi) \;\; \sim \;\; \mathrm{e}^{-V(\phi)} \, , \quad V(\phi) = (\phi - b)^T(\phi - b) + \lambda(\phi^T\phi - 1)^2 \, . \tag{32}$$

We follow [9] and use the identity

$$V(\phi) \;\; = \;\; \alpha(\phi - \alpha^{-1}b)^T(\phi - \alpha^{-1}b) + \lambda(\phi^T\phi - v_\alpha^2)^2 - c_\alpha \, , \tag{33}$$

where α is a free parameter and

$$v_\alpha^2 = 1 + \frac{\alpha - 1}{2\lambda} \, , \qquad c_\alpha = \lambda(v_\alpha^4 - 1) + (\alpha^{-1} - 1) b^T b \, . \tag{34}$$

We define two types of local updates.

11.5.1 Heatbath update

1. Draw a new value ϕ' according to the trial probability density
$$p_{\rm trial}(\phi') \;=\; \frac{\alpha}{\pi}{\rm e}^{-\alpha(\phi'-\alpha^{-1}b)^T(\phi'-\alpha^{-1}b)}\,. \tag{35}$$
In practice this is done by generating a Gaussian two-component vector R with probability density
$$p(R) \;=\; \frac{1}{\pi}{\rm e}^{-R^TR} \tag{36}$$
and then by setting
$$\phi' \;=\; \frac{1}{\sqrt{\alpha}}R+\alpha^{-1}b\,. \tag{37}$$
This follows from the identification $R=\sqrt{\alpha}(\phi'-\alpha^{-1}b)$.

2. Accept the new value ϕ' with probability
$$p_{\rm acpt}(\phi') \;=\; {\rm e}^{-\lambda(\phi'^T\phi'-v_\alpha^2)^2}\,. \tag{38}$$

3. Repeat the trial Eq. (37) until it is accepted.

The acceptance probability is
$$\begin{aligned} A(\alpha) &= \int d^2\phi' \;\; p_{\rm trial}(\phi')p_{\rm acpt}(\phi') \\ &= \frac{\alpha}{\pi}\int d^2\phi' \;\; {\rm e}^{-\alpha(\phi'-\alpha^{-1}b)^T(\phi'-\alpha^{-1}b)}{\rm e}^{-\lambda(\phi'^T\phi'-v_\alpha^2)^2}\,. \end{aligned} \tag{39}$$
The maximization of the acceptance with respect to α means solving the equation $\frac{{\rm d}A}{{\rm d}\alpha}=0$. Differentiating the expression Eq. (39) with respect to α and requiring that the integrand vanishes leads to the cubic equation
$$\alpha^3-(1-2\lambda)\alpha^2-2\lambda\alpha \;=\; 2\lambda b^Tb\,. \tag{40}$$
The acceptance is maximized by taking $\alpha=\alpha_{\rm opt}=$ positive root of Eq. (40). One can show that $\forall\lambda>0$, $A(\alpha_{\rm opt})\to\frac{1}{\sqrt{3}}\simeq 0.577\ldots$ as $b^Tb\to\infty$. An approximate solution $\alpha_{\rm opt}^{\rm approx}$ is defined as follows. We introduce
$$\alpha_0 \;=\; \frac{1}{2}-\lambda+\left[\left(\frac{1}{2}-\lambda\right)^2+2\lambda\right]^{1/2}, \tag{41}$$

which is the solution of Eq. (40) for $b^T b = 0$. Then we define

$$\alpha_{\text{opt}}^{\text{approx}} = h_0 + \left[h_1 + h_2 b^T b\right]^{1/2} \tag{42}$$

with

$$h_0 = \alpha_0 - \frac{\alpha_0^2 + 2\lambda}{6\alpha_0 + 4\lambda - 2}, \tag{43}$$

$$h_1 = \left(\frac{\alpha_0^2 + 2\lambda}{6\alpha_0 + 4\lambda - 2}\right)^2, \tag{44}$$

$$h_2 = \frac{4\lambda}{6\alpha_0 + 4\lambda - 2}. \tag{45}$$

The derivation of the approximate solution starts by setting $\alpha_{\text{opt}} = \alpha_0 + \delta$ and expanding $f(\alpha_0 + \delta) = 0$ in a Taylor series to second order in δ, where $f(\alpha) = \alpha^3 - (1 - 2\lambda)\alpha^2 - 2\lambda\alpha - 2\lambda b^T b$.

The total probability to generate a new value ϕ' is

$$\begin{aligned} P(\phi') &= p_{\text{trial}}(\phi')p_{\text{acpt}}(\phi') + \\ &\quad (1 - A)p_{\text{trial}}(\phi')p_{\text{acpt}}(\phi') + \\ &\quad (1 - A)^2 p_{\text{trial}}(\phi')p_{\text{acpt}}(\phi') + \ldots \\ &= \sum_{k=0}^{\infty}(1 - A)^k p_{\text{trial}}(\phi')p_{\text{acpt}}(\phi') \\ &= \frac{p_{\text{trial}}(\phi')p_{\text{acpt}}(\phi')}{A}. \end{aligned} \tag{46}$$

This shows that $P(\phi) \sim p_{\text{trial}}(\phi)p_{\text{acpt}}(\phi) \sim \pi(\phi)$ corresponds to the normalized equilibrium probabilty since $\int \mathrm{d}^2\phi \;\; P(\phi) = 1$.

11.5.2 Metropolis reflection update

1. Propose the change

$$\phi \;\; \to \;\; \phi' = 2\alpha^{-1}b - \phi. \tag{47}$$

   This proposal is symmetric since $\phi = 2\alpha^{-1}b - \phi'$. It leaves the term $\alpha(\phi - \alpha^{-1}b)^T(\phi - \alpha^{-1}b)$ in Eq. (33) and the integration measure $\mathrm{d}^2\phi$ invariant.

2. Metropolis acceptance-rejection step: accept the proposal with probability

$$p_{\rm metro} = \min\left\{1, \frac{\pi(\phi')}{\pi(\phi)}\right\} = \min\left\{1, \frac{e^{-\lambda(\phi'^T\phi'-v_\alpha^2)^2}}{e^{-\lambda(\phi^T\phi-v_\alpha^2)^2}}\right\} \quad (48)$$

$$= \min\left\{1, e^{\lambda(\phi^T\phi-\phi'^T\phi')(\phi^T\phi+\phi'^T\phi'-2v_\alpha^2)}\right\} \quad (49)$$

11.5.3 Hybrid overrelaxation

A hybrid overrelaxation cycle consists of one sweep through all the lattice points using the heatbath update followed by a number $n_{\rm ref}$ of sweeps using the Metropolis reflection update. In particular the reflection updates help reducing the autocorrelations as it was shown in [9].

Exercise: prove detailed balance for the heatbath update and for the Metropolis reflection update.

11.6 Equipartition

A useful relation to check the correctness of the Markov chain simulation is (the external field is set to zero)

$$2 = -4\kappa\sum_\mu \left\langle\phi_x^T\phi_{x+a\hat{\mu}}\right\rangle - 2(2\lambda-1)\left\langle\phi_x^T\phi_x\right\rangle + 4\lambda\left\langle(\phi_x^T\phi_x)^2\right\rangle . \quad (50)$$

The left-hand side of Eq. (50) is the number of degrees of freedom per lattice point, which is two because the field has two independent real components. Translation invariance has been assumed to hold and x is an arbitrary point on the lattice. The starting point for the proof of Eq. (50) is the partition function Eq. (11) for $h = 0$ (in the following we drop the argument h). We rescale the field variables $\phi_x = \alpha\tilde{\phi}_x\ \forall x$ by a parameter α

$$Z = \int \mathcal{D}[\tilde{\phi}]\ \alpha^{2V} e^{-S[\alpha\tilde{\phi}]}$$

and compute the derivative of Z with respect to α

$$\begin{aligned}\frac{\mathrm{d}Z}{\mathrm{d}\alpha} &= \int \mathcal{D}[\tilde{\phi}] \;\; \alpha^{2V} \left\{ \frac{2V}{\alpha} + \sum_x \left[-2\alpha \tilde{\phi}_x^T \tilde{\phi}_x - 4\alpha\lambda \left(\alpha^2 \tilde{\phi}_x^T \tilde{\phi}_x - 1 \right) \tilde{\phi}_x^T \tilde{\phi}_x + \right.\right. \\ &\quad \left.\left. + 4\alpha\kappa \sum_\mu \tilde{\phi}_x^T \tilde{\phi}_{x+a\hat{\mu}} \right] \right\} \mathrm{e}^{-S[\alpha\tilde{\phi}]} \,.\end{aligned}$$

At this point we can change the integration variables back to ϕ

$$\begin{aligned}\frac{\mathrm{d}Z}{\mathrm{d}\alpha} &= \int \mathcal{D}[\phi] \;\; \left\{ \frac{2V}{\alpha} + \sum_x \left[-\frac{2}{\alpha} \phi_x^T \phi_x - \frac{4\lambda}{\alpha} \left(\phi_x^T \phi_x - 1 \right) \phi_x^T \phi_x + \right.\right. \\ &\quad \left.\left. + \frac{4\kappa}{\alpha} \sum_\mu \phi_x^T \phi_{x+a\hat{\mu}} \right] \right\} \mathrm{e}^{-S[\phi]} \,.\end{aligned}$$

Since Z is actually independent of α, the derivative $\frac{\mathrm{d}Z}{\mathrm{d}\alpha}$ has to vanish $\forall\alpha$. We evaluate the expression $0 = \frac{1}{Z} \left.\frac{\mathrm{d}Z}{\mathrm{d}\alpha}\right|_{\alpha=1}$ and obtain

$$0 = 2V + \sum_x \left[4\kappa \sum_\mu \left\langle \phi_x^T \phi_{x+a\hat{\mu}} \right\rangle - 2(1-2\lambda) \left\langle \phi_x^T \phi_x \right\rangle - 4\lambda \left\langle \left(\phi_x^T \phi_x \right)^2 \right\rangle \right] . \tag{51}$$

From this relation we obtain Eq. (50).

11.7 Autocorrelation, statistical errors

In order to simplify the notation we set the external field to zero and drop this argument without loss of generality. We assume that the Markov chain has reached equilibrium after *thermalization* (see below) and it produces a finite ensemble of N field configurations $\Phi^{(1)}$, $\Phi^{(2)}$, ..., $\Phi^{(N)}$ with the correct probability density given by Eq. (14). We denote the value of an observable A evaluated on the configuration $\Phi^{(i)}$ by $a_i = A[\Phi^{(i)}]$. The Monte Carlo estimate of the expectation value Eq. (12) is $\langle A \rangle \simeq \frac{1}{N} \sum_{i=1}^N a_i$. The error squared of this estimate can be approximated by the average quadratic

deviation of the mean values which is given by (we follow [6])

$$\begin{aligned}\sigma(N,A)^2 &= \left\langle \left(\frac{1}{N}\sum_{i=1}^{N} a_i - \langle A\rangle \right)^2 \right\rangle_{\text{MC}} \\ &= \frac{1}{N^2}\sum_{i,j=1}^{N} \underbrace{\langle (a_i - \langle A\rangle)(a_j - \langle A\rangle)\rangle_{\text{MC}}}_{\Gamma_A(i,j)} \qquad (52)\end{aligned}$$

Here the expectation value $\langle\cdots\rangle_{\text{MC}}$ denotes an average over infinitely many Markov chains of length N whereas $\langle\cdots\rangle$ means the statistical average according to Eq. (12). The function $\Gamma_A(i,j)$ is called *autocorrelation function.* It depends only on the distance between the measurements i and j and not on their distance from the beginning of the chain, so we write $\Gamma_A(i-j)$. The relation $\Gamma_A(i-j) = \Gamma_A(j-i)$ holds as it is obvious from the definition of Γ_A. Introducing the *integrated autocorrelation time* defined by

$$\tau_{\text{int,A}} = \frac{1}{2}\sum_{t=-\infty}^{\infty} \frac{\Gamma_A(t)}{\Gamma_A(0)}\,. \qquad (53)$$

the error squared in Eq. (52) can be expressed by

$$\sigma(N,A)^2 = \frac{\text{var}(A)}{N/(2\tau_{\text{int,A}})}\,. \qquad (54)$$

The *variance* is defined by

$$\text{var}(A) = \left\langle (A - \langle A\rangle)^2 \right\rangle = \Gamma_A(0)\,. \qquad (55)$$

The quantities $\text{var}(A)$, $\Gamma_A(t)$ and $\tau_{\text{int,A}}$ entering Eq. (54) can be estimated from the Monte Carlo simulation as follows:

$$\text{var}(A) \simeq \frac{1}{N}\sum_{i=1}^{N}\left(a_i - \frac{1}{N}\sum_{j=1}^{N} a_j\right)^2 , \qquad (56)$$

$$\Gamma_A(t) \simeq \frac{1}{N-t}\sum_{i=t+1}^{N}\left(a_i - \frac{1}{N}\sum_{j=1}^{N} a_j\right)\left(a_{i-t} - \frac{1}{N}\sum_{k=1}^{N} a_k\right) , \qquad (57)$$

$$\tau_{\text{int,A}} \simeq 0.5 + \sum_{t=1}^{W} \frac{\Gamma_A(t)}{\Gamma_A(0)}\,. \qquad (58)$$

Typically the autocorrelation function decays exponentially like $\Gamma_A(t) \sim \exp(-|t|/\tau)$ for some Monte Carlo time scale τ as $|t| \to \infty$. Notice that the units of Monte Carlo time refer to the number of updates in the Markov chain between evaluations (called "measurements") of the observable A. The summation over t in Eq. (58) actually extends to $t = \infty$. Practically the sum is performed up to a window $t \leq W$ where the exponentially decaying summand becomes zero within its statistical noise (extending the sum further would result in summing up noise).

The proof of Eq. (54) goes as follows. The starting point is Eq. (52). If the estimates a_i were independent then

$$\langle (a_i - \langle A\rangle)(a_j - \langle A\rangle)\rangle_{\mathrm{MC}} = \begin{cases} \langle (a_i - \langle A\rangle)\rangle_{\mathrm{MC}} \langle (a_j - \langle A\rangle)\rangle_{\mathrm{MC}} = 0 & i \neq j \\ \langle (a_i - \langle A\rangle)^2\rangle_{\mathrm{MC}} = \mathrm{var}(A) & i = j \end{cases},$$

where we used the property $\langle a_i\rangle_{\mathrm{MC}} = \langle A\rangle$. It follows that for independent estimates $\Gamma_A(i-j) = \Gamma_A(0)\delta_{ij} = \mathrm{var}(A)\delta_{ij}$ and $\sigma(N,A)^2 = \mathrm{var}(A)/N$. But in general $\Gamma_A(t) > 0$ for $t \neq 0$ and the error becomes larger. We use the approximation

$$\sum_{i,j=1}^{N} \frac{\Gamma_A(i-j)}{\Gamma_A(0)} = N \sum_{t=-N+1}^{N-1} (1 - |t|/N) \frac{\Gamma_A(t)}{\Gamma_A(0)} \simeq N \sum_{t=-\infty}^{\infty} \frac{\Gamma_A(t)}{\Gamma_A(0)} \,. \tag{59}$$

The approximation is justified assuming $N \gg \tau$. The last expression in Eq. (59) equals $2N\tau_{\mathrm{int,A}}$ (cf. the definition Eq. (53)). Finally

$$\sigma(N,A)^2 = \frac{\mathrm{var}(A)}{N^2} \sum_{i,j=1}^{N} \frac{\Gamma_A(i-j)}{\Gamma_A(0)} = \frac{\mathrm{var}(A)}{N} 2\tau_{\mathrm{int,A}} \,. \tag{60}$$

In principle the integrated autocorrelation times can be quite different depending on the observable A. They are dynamical quantities which depend on the eigenvalues and the eigenvectors of the transition function T defining the Markov chain. Typically $\tau_{\mathrm{int,A}}$ is of the order of the exponential autocorrelation time τ_{exp} defined in Eq. (22). As a practical rule the thermalization should consist of 20 to 100 times τ_{max} updates, where $\tau_{\mathrm{max}} \sim \tau_{\mathrm{exp}}$ is the largest autocorrelation time.

A MATLAB program `autocorr.m` to compute the error Eq. (54) can be found on `http://csis.uni-wuppertal.de/courses/lab216.html`.

For further reading on statistical error analysis we refer to [6, 8, 10, 11].

12 Parallelizing the Poisson equation

This section gives an introduction to set up an iterative method to solve partial differential equations (PDEs) with MPI. In the lab course project you develop a simple parallel algorithm to solve Poisson's equation on the unit square (2D), using Dirichlet boundary conditions. For more details see Numerical Recipes in C [12].

12.1 Poisson equation matrices

In two dimensions, the Poisson equation can be written as

$$-\triangle u = -\nabla^2 u = -\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = f(x,y) \tag{61}$$

where u is the unknown function and $f(x,y)$ is a given function. The simplest discretization of this equation is based on the finite difference expression for a second derivative

$$\frac{d^2 u}{dx^2} := u_{xx} = \frac{u(x+\Delta x) - 2u(x) + u(x-\Delta x)}{\Delta x^2} + O(\Delta x)$$

Define the computational domain (*e.g.*, the unit square $[0,1] \times [0,1]$) as the following set of points:

$$D = \{(x,y) | 0 < x, y < 1\}$$

The set of points on the boundary of D (the "edges" of the square) is denoted ∂D. Given functions $f(x,y)$ and $g(x,y)$, the PDE-problem can then be formulated in the following way: Find (approximate) $u(x,y)$ such that:

$$\begin{aligned} -u_{xx}(x,y) - u_{yy}(x,y) &= f(x,y), \qquad (x,y) \in D \\ u(x,y) &= g(x,y), \qquad (x,y) \in \partial D \end{aligned}$$

The domain D is discretized using equidistant meshpoints (see also Fig. 3)

$$(x_j, y_k), j = 0, 1, \ldots, n+1, k = 0, 1, \ldots, n+1$$

with spacing $h = 1/(n+1)$, where:

$$0 = x_0 < x_1 < \ldots < x_n < x_{n+1} = 1, 0 = y_0 < y_1 < \ldots < y_n < y_{n+1} = 1,$$

and $x_j = jh,\ j = 0, 1, \ldots, n+1,\ y_k = kh,\ k = 0, 1, \ldots, n+1.$

Let $u_{j,k}$ be the approximation of $u(x_j, y_k)$, the standard second order approximations of the second derivatives:

$$u_{xx}(x_j, y_k) \approx \frac{u_{j-1,k} - 2u_{j,k} + u_{j+1,k}}{h^2},\ u_{yy}(x_j, y_k) \approx \frac{u_{j,k-1} - 2u_{j,k} + u_{j,k+1}}{h^2},$$

giving a system of n^2 linear equations in n^2 unknowns $(u_{j,k}, 1 \le j, k \le n)$:

$$\frac{4u_{j,k} - u_{j-1,k} - u_{j+1,k} - u_{j,k-1} - u_{j,k+1}}{h^2} = f(x_j, y_k), \quad 1 \le j, k \le n \tag{62}$$

and the boundary values (if j or k equals 0 or $n+1$) are given by

$$u_{j,k} = g(x_j, y_k), \quad j \vee k = 0 \vee n+1.$$

The corresponding linear system can be written as a matrix problem $Au = b$, where we wish to find a solution u, given that A is a matrix capturing the differentiation operator, and u and b are column vectors, the latter corresponding to any source or boundary terms.

This can be achieved by choosing a linear ordering of the unknowns $u_{i,j}$, *e.g.*, a natural column/row ordering. Here is an example of the linear system of the Poisson equation (61) for $n = 4$, using the notation $f_{j,k} = f(x_j, x_k)$ and $g_{j,k} = g(x_j, x_k)$ for source and boundary terms:

$$\left[\begin{array}{cccc|cccc|cccc|cccc}
4 & -1 & & & -1 & & & & & & & & & & & \\
-1 & 4 & -1 & & & -1 & & & & & & & & & & \\
 & -1 & 4 & -1 & & & -1 & & & & & & & & & \\
 & & -1 & 4 & & & & -1 & & & & & & & & \\
\hline
-1 & & & & 4 & -1 & & & -1 & & & & & & & \\
 & -1 & & & -1 & 4 & -1 & & & -1 & & & & & & \\
 & & -1 & & & -1 & 4 & -1 & & & -1 & & & & & \\
 & & & -1 & & & -1 & 4 & & & & -1 & & & & \\
\hline
 & & & & -1 & & & & 4 & -1 & & & -1 & & & \\
 & & & & & -1 & & & -1 & 4 & -1 & & & -1 & & \\
 & & & & & & -1 & & & -1 & 4 & -1 & & & -1 & \\
 & & & & & & & -1 & & & -1 & 4 & & & & -1 \\
\hline
 & & & & & & & & -1 & & & & 4 & -1 & & \\
 & & & & & & & & & -1 & & & -1 & 4 & -1 & \\
 & & & & & & & & & & -1 & & & -1 & 4 & -1 \\
 & & & & & & & & & & & -1 & & & -1 & 4
\end{array}\right]
\begin{bmatrix} u_{11} \\ u_{21} \\ u_{31} \\ u_{41} \\ u_{12} \\ u_{22} \\ u_{32} \\ u_{42} \\ u_{13} \\ u_{23} \\ u_{33} \\ u_{43} \\ u_{14} \\ u_{24} \\ u_{34} \\ u_{44} \end{bmatrix}
= h^2 \begin{bmatrix} f_{11}+g_{10}+g_{01} \\ f_{21}+g_{20} \\ f_{31}+g_{30} \\ f_{41}+g_{40}+g_{51} \\ f_{12}+g_{02} \\ f_{22} \\ f_{32} \\ f_{42}+g_{52} \\ f_{13}+g_{03} \\ f_{23} \\ f_{33} \\ f_{43}+g_{53} \\ f_{14}+g_{15}+g_{04} \\ f_{24}+g_{25} \\ f_{34}+g_{35} \\ f_{44}+g_{45}+g_{54} \end{bmatrix}$$

Theoretically, this problem could be solved on a computer by any of the standard methods for dealing with matrices. However, the real challenge for PDEs is that frequently, the dimensionality of the problem can be enormous. For example, for a two dimensional PDE problem, a 100×100 grid would be a perfectly reasonable size to consider. Thus u would be a vector with 10^4 elements, and A would be a matrix with 10^8 elements. Even allocating memory for such a large matrix may be problematic. Direct approaches, such as the explicit construction of A^{-1}, are impractical.

The key to making progress is to note that in general, the matrix A is extremely sparse, *i.e.*, all empty entries in the matrix above are equal to zero, since the linear relationships usually only relate nearby mesh-points. We therefore seek methods which do not require ever explicitly specifying all the elements of A, but exploit its special structure directly. Many of these methods are iterative - we start with a guess u_k, and apply a process that yields a closer solution u_{k+1}.

Typically, these iterative methods are based on a splitting of A. This is a decomposition $A = M - K$, where M is non-singular. Any splitting creates a possible iterative process, we can write

$$\begin{aligned} Au &= b \\ (M-K)u &= b \\ Mu &= Ku + b \\ u &= M^{-1}Ku + M^{-1}b \end{aligned}$$

and hence a possible iteration is $u_{k+1} = M^{-1}Ku_k + M^{-1}b$.

Of course, there is no guarantee that an arbitrary splitting will result in an iterative method which converges. To study convergence, we must look at the properties of the matrix $R = M^{-1}K$. For convergence analysis, it is helpful to introduce the spectral radius

$$\rho(R) = \max_j(|\lambda_j|)$$

where the λ_j are the eigenvalues of R. It can be shown that an iterative scheme converges if and only if $\rho(R) < 1$. The time to gain k extra digits of accuracy, *i.e.*, reduce the error by a factor $10^{-k} = \rho^q$ is approximately

$$q = -k/\log_{10}\rho(R).$$

12.2 Jacobi method

Jacobi's algorithm is a very simple iterative method, to solve a linear system. The main advantage of Jacobi's method is its simplicity and the ease by which it can be parallelized. The big drawback is its slow convergence, which becomes slower with increasing n.

One iteration of the standard Jacobi method can be written like follows, by introducing temporary quantities, $\tilde{u}_{j,k}$, $1 \leq j, k \leq n$, derived from (62), which are computed using two loops over j and k

$$\tilde{u}_{j,k} = \frac{u_{j-1,k} + u_{j+1,k} + u_{j,k-1} + u_{j,k+1} + h^2 f(x_j, y_k)}{4}, \quad 1 \leq j, k \leq n,$$

where the g-values are used for the boundary, as mentioned above ($u_{j,k} = g_{j,k}$ for $j \vee k = 0 \vee n+1$). In the image below, internal nodes are marked by circles and boundary nodes by small squares. To form the $\tilde{u}_{j,k}$-value marked by **a**, $h^2 f(x_j, y_k)$ and an average of the four u-neighbors is used. The **b**-node uses a boundary value and the **c**-node uses two boundary values.

To find the corresponding matrix form, write $A = D - L - U$ where D is diagonal, L is lower-triangular, and U is upper-triangular. Then the above iteration can be written as

$$\tilde{u} = D^{-1}(L+U)u + D^{-1}b.$$

The convergence properties are then set by the matrix $R_J = D^{-1}(L+U)$, *i.e.*, the convergence rate or spectral radius $\rho_J = \rho(R_J)$.

When all $\tilde{u}_{j,k}$-values have been computed the difference between the two successive approximations to the solution is given by:

$$\delta = \max |\tilde{u}_{j,k} - u_{j,k}|, \; 1 \leq j, k \leq n$$

$u_{j,k}$ is replaced by the temporary values $\tilde{u}_{j,k}$ for $1 \leq j, k \leq n$ and the Jacobi-step is repeated until $\delta < \tau$, with τ the intended tolerance of the solution.

The algorithm can be parallelized easily, because each $u_{i,j}$ may be updated independently. We can use the so called domain decomposition, *i.e.*, dividing the domain D, into N subdomains. Assuming p CPUs, the unit square is divided in $N = p$ smaller squares (or rectangles). The number of mesh-points n^2 should be chosen as to be evenly divisible by N, see the example in Fig.3 on the right for $n = 8$ and $N = 4$. The CPUs can compute the $u_{j,k}$ values corresponding to the circles, in parallel. In order to compute

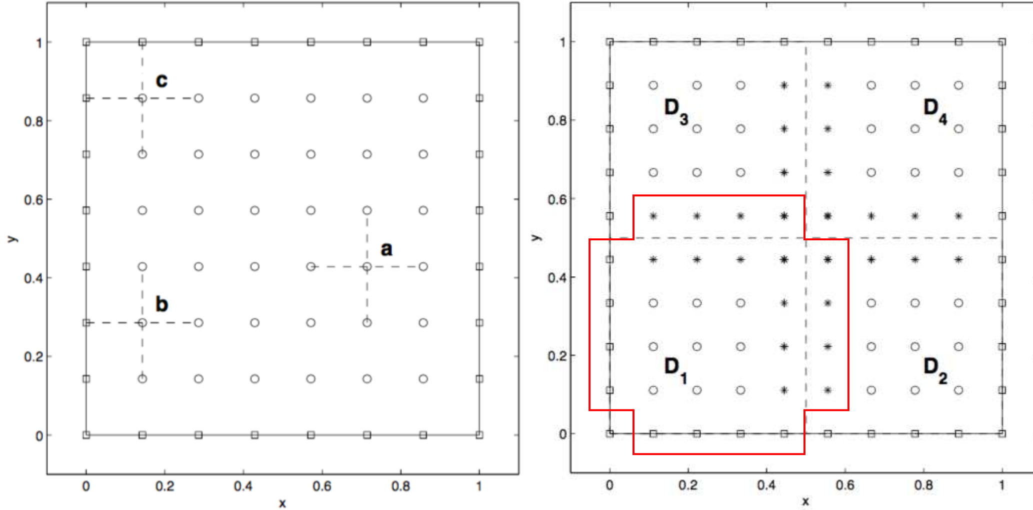


Figure 3: (left) sample mesh for $n = 6$ with n^2 internal nodes (squares) and $4(n+1)$ boundary nodes (squares). (right) sample mesh for $n = 8$, divided into four subdomains D_{1-4} with $4(n-1)$ subdomain boundary nodes (*). The red line surrounds a necessary data structure for a local subdomain.

the *-values, CPUs must first have communicated and exchanged some data with neighboring domains. To make this efficient, we want to minimize the number of matrix-elements that are communicated, and when we have to communicate we want to send as much data as possible in one go. When $n >> p$, so that each processor owns a large number n^2/p of mesh-points, the amount of data communicated, will be relatively small.

12.3 Gauss-Seidel and Successive Over-Relaxation

A modification of Jacobi's method in order to simplify the algorithm and to make it slightly less inefficient, is to overwrite the $u_{j,k}$ with $\tilde{u}_{j,k}$ as soon as it has been used to update the δ-value. So, the latest $u_{j,k}$ available will be used at all times. Note that it is sufficient to store $\tilde{u}_{j,k}$ (and δ) as scalar variables and not as arrays in this case. The iteration can be written as

$$\text{for } i = 1 \text{ to } n^2: \quad \tilde{u}_i = \frac{1}{a_{ii}}(b_i - \sum_{j=1}^{i-1} a_{ij}\tilde{u}_j - \sum_{j=i+1}^{n^2} a_{ij}u_j),$$

where a_{ij} are the elements of A, with $a_{ii} = 4$ in our case, and u_i, $\tilde{u}_i$ and b_i are the elements of the column vectors u, $\tilde{u}$ and b, respectively. From the

algorithm above, we can write down the corresponding matrix splitting for the Gauss-Seidel method as $\tilde{u} = (D-L)^{-1}Uu + (D-L)^{-1}b$.

When parallelizing the algorithm, we have to be careful about the ordering of the individual updates, *i.e.*, the loop over i, because of the dependence to previously updated neighbors. The idea is to set up an even/odd (checkerboard) grid by splitting the mesh into even and odd mesh-points whenever the mesh-point indices $(i+j)$ are even/odd. Note, that all even mesh-points have only odd neighbors and vice versa, hence we can implement the Gauss-Seidel algorithm in two parallel steps, one for updating all even points, and a second one for all odd points. Using domain decomposition as before, we have to update all even points and send the results to the corresponding neighbor domains, before updating the odd points, sending their results and start all over if necessary. It turns out that one iteration of the Gauss-Seidel method is equivalent to two Jacobi steps, *i.e.*, $\rho_{GS}(n) = \rho_J(n)^2$, with $\rho_{GS} = \rho(R_{GS})$ and $R_{GS} = (D-L)^{-1}U$. Note however, the complexity is the same: we still need $O(n^2)$ iterations.

Successive Over-Relaxation (SOR) is a refinement to the Gauss-Seidel algorithm, and depends on the even/odd ordering for its speedup. At each stage in the Gauss-Seidel algorithm, a value u_i is updated to a new one $\tilde{u}_i$, which we can think of as displacing u_i by an amount $\Delta u = \tilde{u}_i - u_i$. The idea behind over-relaxation is that if the correction Δu is a good direction in which to move (update) u_i to make it a better solution, one should move even further in that direction, by $\omega \Delta u$. The method converges for $0 < \omega < 2$, but typically $\omega > 1$ (over-relaxation). The iteration can be written as:

$$\text{for } i = 1 \text{ to } n^2: \quad \tilde{u}_i = (1-\omega)u_i + \frac{\omega}{a_{ii}}(b_i - \sum_{j=1}^{i-1} a_{ij}\tilde{u}_j - \sum_{j=i+1}^{n^2} a_{ij}u_j),$$

the corresponding matrix form is

$$\tilde{u} = (D+\omega L)^{-1}[(1-\omega)D - \omega U]u + (D+\omega L)^{-1}\omega b,$$

and therefore $R_{SOR} = (D+\omega L)^{-1}[(1-\omega)D - \omega U]$. It can be shown, that the optimal choice for

$$\omega = \frac{2}{1+\sqrt{1-\rho_{SOR}^2}} \quad \text{with} \quad \rho_{SOR} = \rho(R_{SOR}) \approx 1 - \frac{4\pi}{n+1},$$

and the order of computation decreases to $O(n)$ per grid point compared to $O(n^2)$ for Jacobi and Gauss-Seidel.

12.4 Conjugate Gradient method

The CG method is suitable for solving any linear system $Ax = b$, where the coefficient matrix A is both symmetricand positive definite.

The algorithm is based on the *Krylov space* introduced in section 6, a linear space whose basis vectors are given by successive applications of A to the start vector x_0, *i.e.*, $\{x_0, Ax_0, A^2x_0, ...A^{N-1}x_0\}$. One very important point to remember about the conjugate gradient method (and other Krylov methods) is that only the matrix-vector product is required. You can write and debug the method using ordinary matrix arithmetic and then apply it using other storage methods, such as the compact matrix storage.

The conjugate gradient algorithm can be described in the following way. Start with an initial guess vector x_0 for $A^{-1}b$ (*e.g.*, $x_0 = 0$), calculate the first residual vector $r_0 = b - Ax_0$ (for $x_0 = 0 \Rightarrow r_0 = b!$), which is to be minimized and gives the initial "search direction" $p_0 = r_0$.

Then iterate while $\rho_k = r_k^T r_k > \epsilon$

$$
\begin{aligned}
s &= Ap_k \\
\alpha &= \frac{\rho_k}{p_k^T s} \\
x_{k+1} &= x_k + \alpha p_k \\
r_{k+1} &= r_k - \alpha s \\
\rho_k + 1 &= r_{k+1}^T r_{k+1} \\
p_{k+1} &= r_{k+1} + \frac{\rho_{k+1}}{\rho_k} p_k
\end{aligned}
$$

During one iteration, the vectors x_k, r_k and p_k can be immediately replaced by x_{k+1}, r_{k+1} and p_{k+1} respectively, in order to save memory. You have to keep ρ_k for the next iteration, hence, use a temporary ρ_{k+1} as well as vector s and scalar α to save computation steps.

At each step the solution x is improved by searching for a better solution in the direction p yielding an improved solution $x + \alpha p$. This direction p is called a gradient because we are in fact doing gradient descent on a certain measure of the error (namely $\sqrt{r^T A^{-1} r}$). The directions p_i and p_j from steps $i \neq j$ of the algorithm are called conjugate, or more precisely A-conjugate, because they satisfy $p_i^T A p_j = 0$. One can also show that after i iterations x_i is the "optimal" solution among all possible linear combinations of the form $\alpha_0 x + \alpha_1 Ax + \alpha_2 A^2 x + \alpha_3 A^3 x + \ldots + \alpha_i A^i x$. For most matrices, the majority of work is in the sparse matrix-vector multiplication $s = Ap$ in the

first step. For Poisson's equation, where we can think of p and s living on a square grid, this means computing $s(i,j) = 4p(i,j) - p(i-1,j) - p(i+1,j) - p(i,j-1) - p(i,j+1)$, which is nearly identical to the inner loop of Jacobi or GSOR in the way it is parallelized. The other operations in CG are easy to parallelize. The dot-products require local dot-products and then a global add using, say, a tree to form the sum in log(p) steps.

The rate of convergence of CG depends on a number κ called the condition number of A. It is defined as the ratio of the largest to the smallest eigenvalue of A (and so is always at least 1). A roughly equivalent quantity is $|A^{-1}||A|$, where the norm of a matrix is the magnitude of the largest entry. The larger the condition number, the slower the convergence. One can show that the number of CG iterations required to reduce the error by a constant $g < 1$ is proportional to the square root of κ. For Poissons's equation, κ is $O(n)$, so $\sqrt{n}$ iterations are needed. This means GSOR and CG take about the same number of steps, but CG is applicable to a much larger class of problems.

12.5 The Assignment

The project consists of three main parts and one bonus exercise:

- Setup of the "mesh", with a variable (even) number n for $(n+2)^2$ elements and variable (efficient) domain decomposition using p processors. Implementation of Jacobi's method to solve the linear system and check scaling with n and domain decomposition. (30 points)

- Implementation of Gauss-Seidel algorithm with successive over-relaxation to solve the linear system using even/odd checkerboard grid and check scaling with n and p. (30 points)

- Implementation of a conjugate gradient algorithm using the domain decomposition from earlier for the Jacobi step and scaling comparison to other methods. (30 points)

- Bonus: Make the GSOR (or CG) version of your code three dimensional, *i.e.*, us a 3D grid and use $f(x,y)$ in the $z=0$ and $g(x,y)$ in the $z=n+1$ planes as boundary conditions, trivial (zero) boundary values in x and y directions and no additional source. The easiest way is to just extend the 2D domains in z direction without further decomposition. You may also just do the whole assignment in 3D! (30 points)

Test Example:

In order to test your program use the source and boundary functions

$$f(x,y) = 2((1+x)\sin(x+y) - \cos(x+y)) \quad \text{and} \quad g(x,y) = (1+x)\sin(x+y),$$

then $u(x,y) = (1+x)\sin(x+y)$ solves the Poison equation as you can check by computing the derivatives:

$$u_{xx} = 2\cos(x+y) - (1+x)\sin(x+y) \text{ and } u_{yy} = -(1+x)\sin(x+y) \text{ so}$$

$$-u_{xx} - u_{yy} = 2((1+x)\sin(x+y) - cos(x+y)) = f(x,y),$$

and u satisfies the boundary condition, as well, since $g = u$. You should use this example to debug your program, but your program should be written so that it can cope with any reasonable functions f and g. Compute the discrete source and sink values once and store them efficiently (domains!).

Hints:

1. If you have no previous experience of this kind of PDE-problem, think through the serial case first. You may also write a Hybrid code, *i.e.* use OpenMP threads on the nodes, but the domain decomposition has to be implemented using MPI.

2. Parallel programming is harder than ordinary serial programming and message passing is usually harder then using threads. So, it is important to think about algorithm, data structures and communication before coding. You will save time by making a plan.

3. You should use dynamic memory allocation (since the master will be reading the value of n). Which process (rank) will take care of which subdomain? Extend the local lattice (array) by the number of inner/outer boundary points on each side, *i.e.* $4\sqrt{n^2/p}$ additional points.

4. Processes must exchange *-data. When should the communication take place? It is inconvenient to have it in the update loop. `MPI_Sendrecv` may be a suitable communication routine.

5. You must think about how to terminate the iterations. Different processes will get different δ-values, and your program will not work if just one process, say, ends the iteration.

References

[1] Peter S. Pacheco, *Parallel Programming with MPI*, Morgan Kaufmann Publishers, Inc. (1997), `http://docenti.unicam.it/tmp/3002.pdf`.

[2] L. Smith and M. Bull, *Development of mixed mode MPI / OpenMP applications*, Scientific Programming, vol. 9, no. 2-3, pp. 83-98 (2001) `downloads.hindawi.com/journals/sp/2001/450503.pdf`.

[3] B. Estrade, *Hybrid Programming with MPI and OpenMP*, `https://www.cct.lsu.edu/~estrabd/intro-hybrid-mpi-openmp.pdf`.

[4] M. Lüscher and P. Weisz, *Scaling Laws and Triviality Bounds in the Lattice ϕ^4 Theory. 3. N Component Model*, Nucl. Phys. B **318** (1989) 705. `http://inspirehep.net/record/265873/files/desy88-146.kek.pdf`

[5] M. Hasenbusch and T. Török, *High precision Monte Carlo study of the 3-D XY universality class*, J. Phys. A **32** (1999) 6361, `https://arxiv.org/abs/cond-mat/9904408`.

[6] U. Wolff, *Computational Physics II*, `http://www.physik.hu-berlin.de/com/teachingandseminars`.

[7] B. Bunk, *Computational Physics III* `http://www-com.physik.hu-berlin.de/~bunk/cp3/`.

[8] F. Knechtli, M. Günther and M. Peardon, *Lattice Quantum Chromodynamics: Practical Essentials* `doi:10.1007/978-94-024-0999-4`.

[9] B. Bunk, *Monte Carlo methods and results for the electro-weak phase transition*, Nucl. Phys. Proc. Suppl. **42** (1995) 566.

[10] I. Montvay and G. Münster, *Quantum fields on a lattice*, (1994).

[11] U. Wolff [ALPHA Collaboration], *Monte Carlo errors with less errors*, Comp.Phys.Comm. **156** (2004) 143, Erratum: [Comp.Phys.Comm. **176** (2007) 383] `https://arxiv.org/abs/hep-lat/0306017`.

[12] B. P. Fannery, S. Teukolsky, W. H. Press, and W. T. Vetterling, *Numerical Recipes in C - The Art of Scientific Computing*, Cambridge Univ. Pr. (1988) `https://e-maxx.ru/bookz/files/numerical_recipes.pdf`