合肥學院
HEFEI UNIVERSITY

Distributed Computing
Lesson 22: MPI

Thomas Weise · 汤卫思

tweise@hfuu.edu.cn · http://www.it-weise.de

Hefei University, South Campus 2
Faculty of Computer Science and Technology
Institute of Applied Optimization
230601 Shushan District, Hefei, Anhui, China
Econ. & Tech. Devel. Zone, Jinxiu Dadao 99

合肥学院 南艳湖校区/南2区
计算机科学与技术系
应用优化研究所
中国 安徽省 合肥市 蜀山区 230601
经济技术开发区 锦绣大道99号

Outline

website

Overview

- Discuss the requirements of scientific and engineering computing
- Consider the algorithm and the hardware perspective
- Get to know MPI as an example framework for using cluster computing
- Learn about the basic components and data structures in MPI
- Apply it by yourself in a homework

Introduction

- Research and Engineering often require enormous amonts of computing power

Introduction

- Research and Engineering often require enormous amonts of computing power
- Many problems require large-scale computations or simulations

Introduction

- Research and Engineering often require enormous amonts of computing power
- Many problems require large-scale computations or simulations, e.g.,
  - weather forecast

Introduction

- Research and Engineering often require enormous amonts of computing power
- Many problems require large-scale computations or simulations, e.g.,
  - weather forecast,

Introduction

- Research and Engineering often require enormous amonts of computing power
- Many problems require large-scale computations or simulations, e.g.,
  - weather forecast,
  - weather survey: National Oceanic and Atmospheric Administration (NOAA) has more than 20PB of data and processes 80TB/day [1, 2]

Introduction

- Research and Engineering often require enormous amonts of computing power
- Many problems require large-scale computations or simulations, e.g.,
  - weather forecast,
  - weather survey: National Oceanic and Atmospheric Administration (NOAA) has more than 20PB of data and processes 80TB/day [1, 2],
  - CERN's LHC produces about 15PB per year

Introduction

- Research and Engineering often require enormous amonts of computing power
- Many problems require large-scale computations or simulations, e.g.,
  - weather forecast,
  - weather survey: National Oceanic and Atmospheric Administration (NOAA) has more than 20PB of data and processes 80TB/day [1, 2],
  - CERN's LHC produces about 15PB per year,
  - heat flow simulations for engines

Introduction

- Research and Engineering often require enormous amonts of computing power
- Many problems require large-scale computations or simulations, e.g.,
  - weather forecast,
  - weather survey: National Oceanic and Atmospheric Administration (NOAA) has more than 20PB of data and processes 80TB/day [1, 2],
  - CERN's LHC produces about 15PB per year,
  - heat flow simulations for engines,
  - Data mining [3, 4]

Introduction

- Research and Engineering often require enormous amonts of computing power
- Many problems require large-scale computations or simulations, e.g.,
  - weather forecast,
  - weather survey: National Oceanic and Atmospheric Administration (NOAA) has more than 20PB of data and processes 80TB/day [1, 2],
  - CERN's LHC produces about 15PB per year,
  - heat flow simulations for engines,
  - Data mining [3, 4], and
  - solving optimization prolems [5–8]

Segments System Share (November 2011)

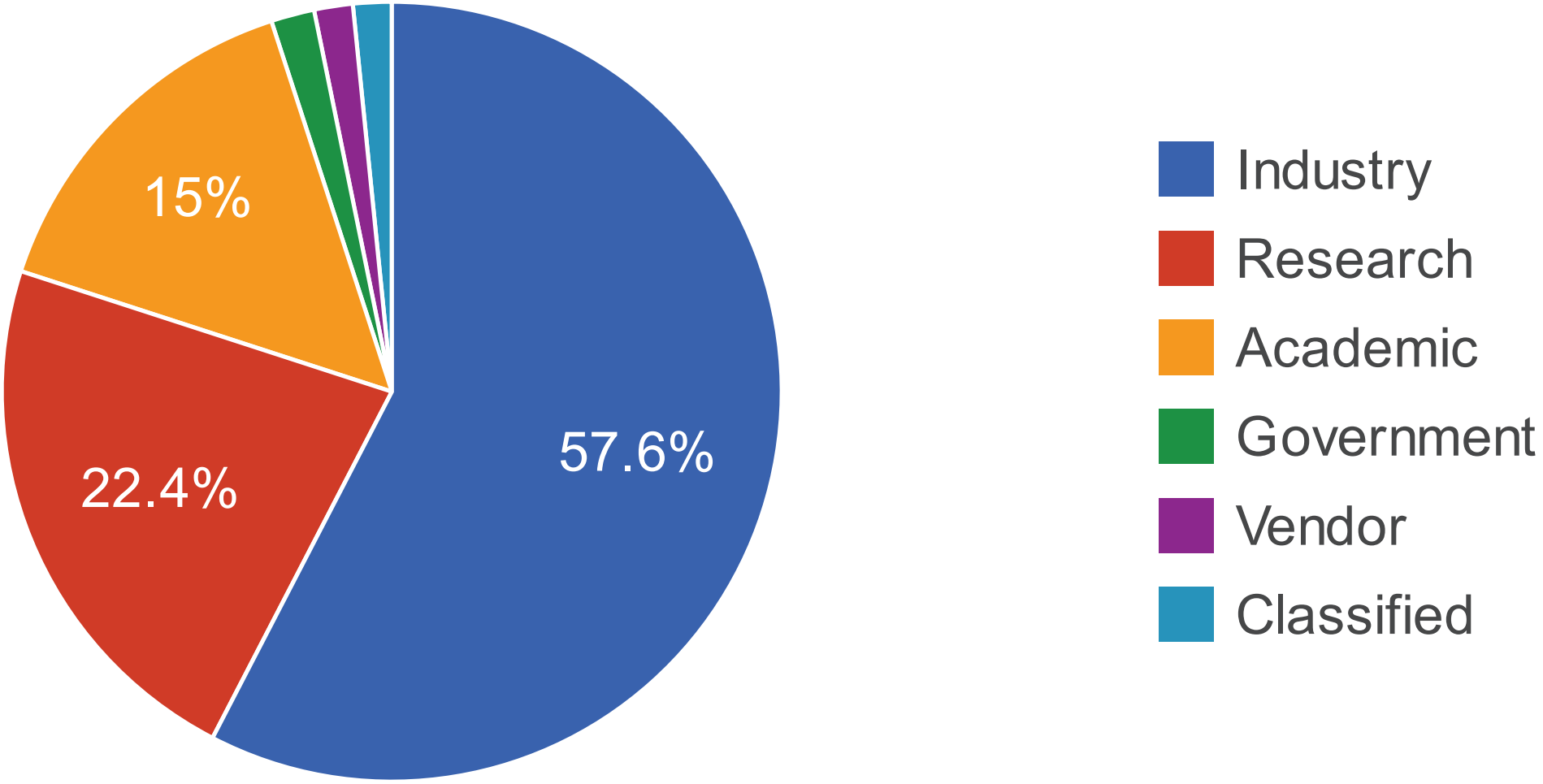


image source: [9]

Application Area System Share (November 2011)

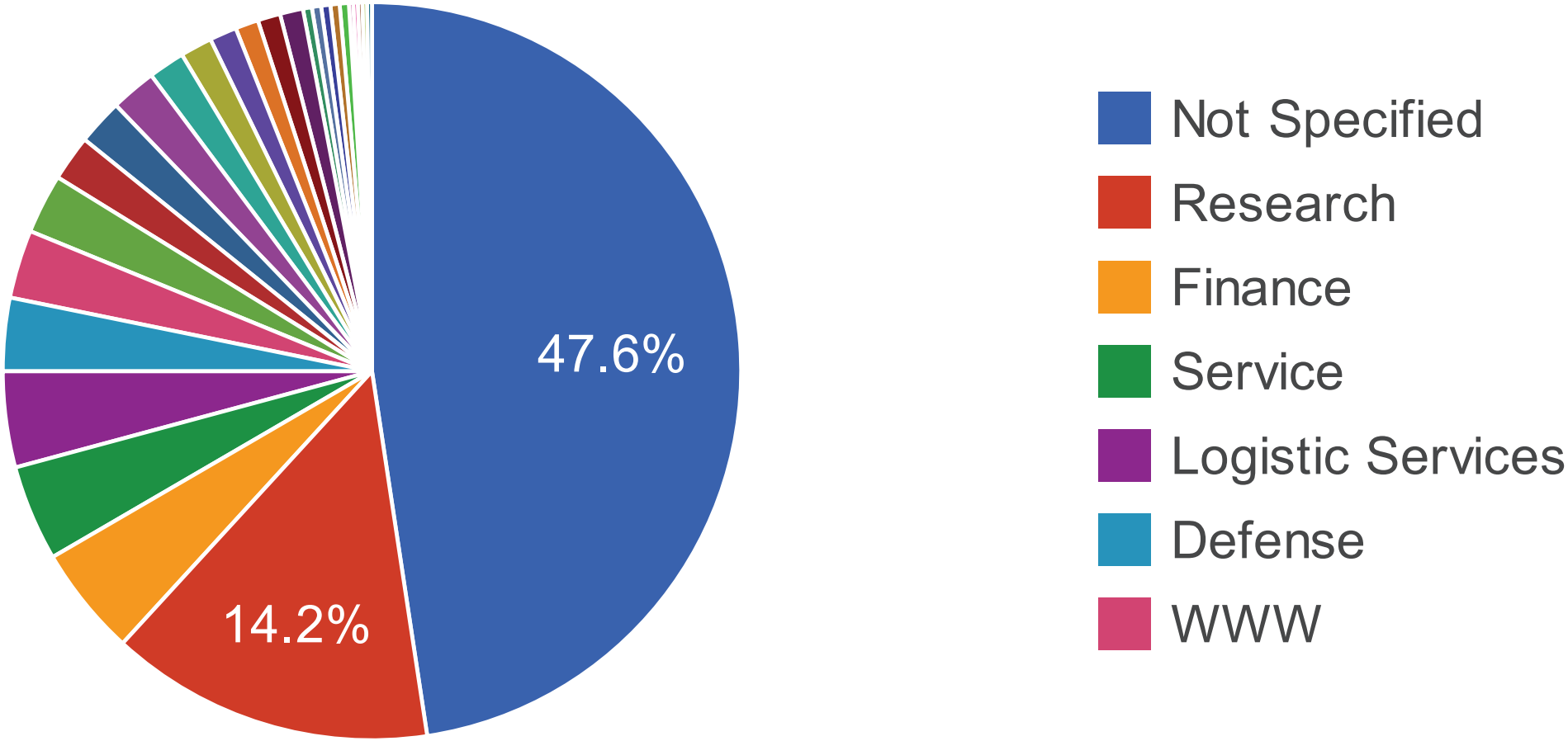


image source: [9]

TOP 500 Super Computers

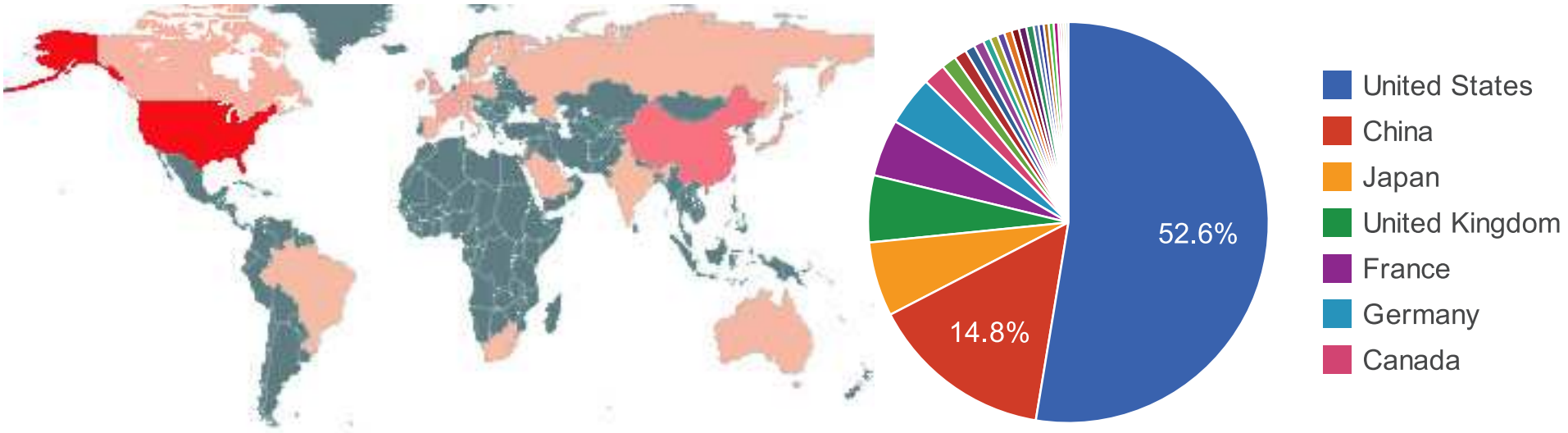


image source: [9]

TOP 500 Super Computers

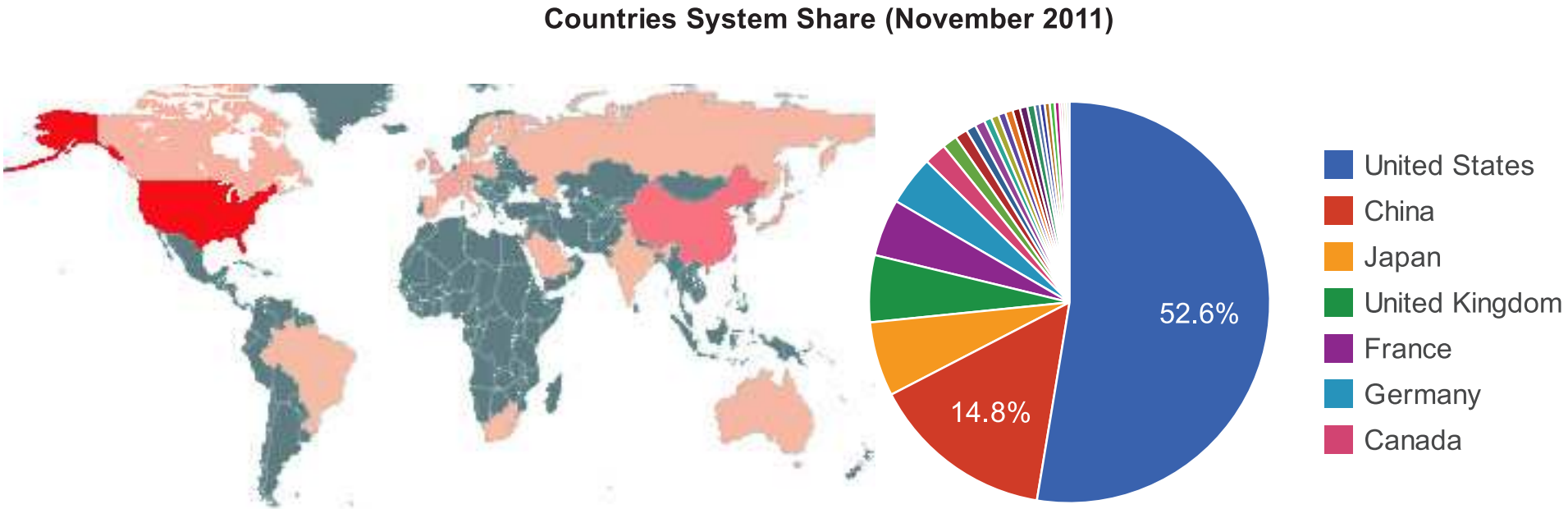


image source: [9]

- Tianhe-1A

TOP 500 Super Computers

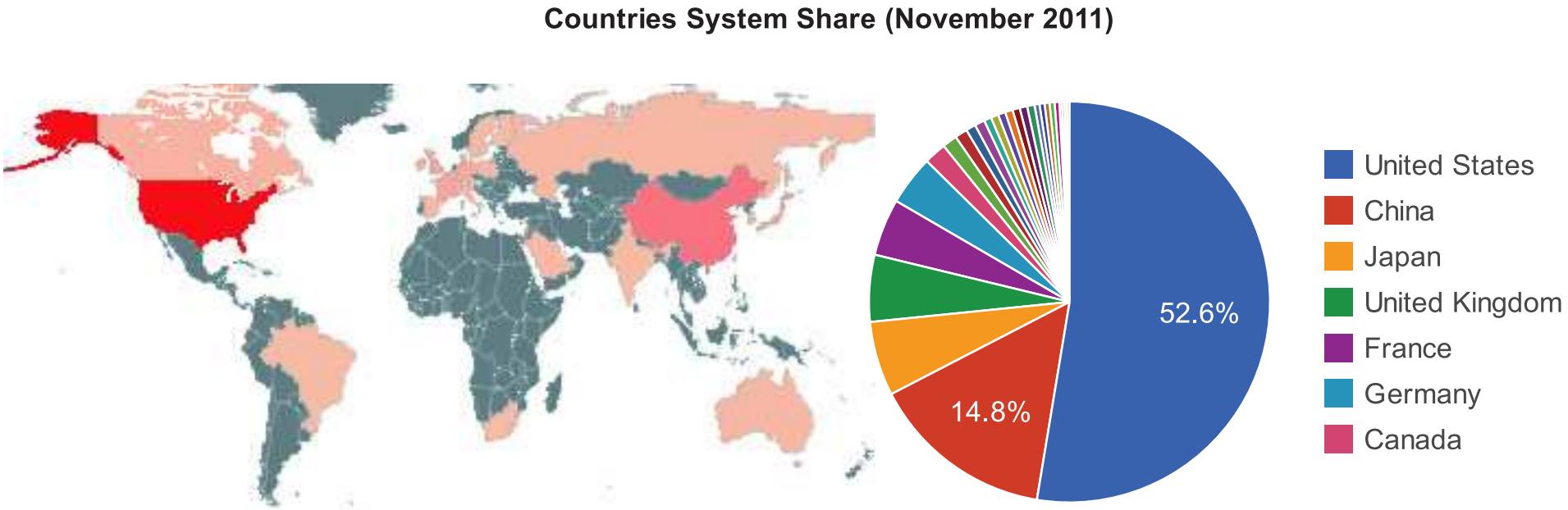


image source: [9]

- Tianhe-1A: #2 in top 500 super computers 2011 [9], at the NUDT in Tianjin, Xeon X5670 6C 2.93 GHz, NVIDIA 2050

TOP 500 Super Computers

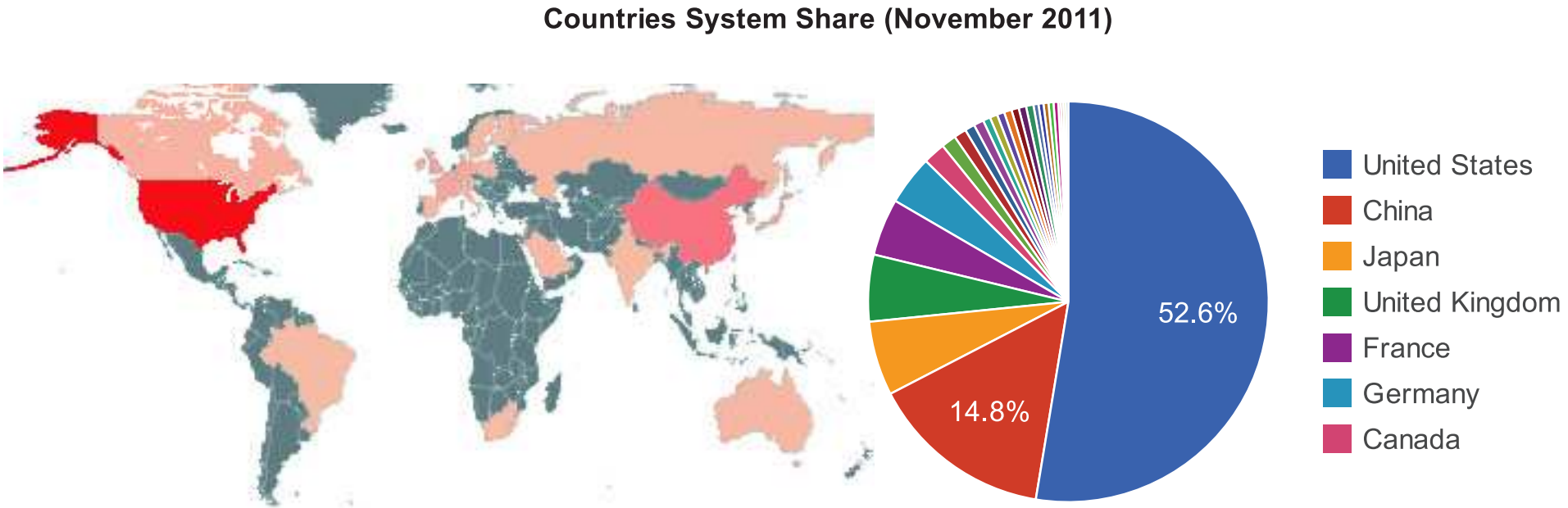


image source: [9]

- Tianhe-1A: #2 in top 500 super computers 2011 [9], at the NUDT in Tianjin, Xeon X5670 6C 2.93 GHz, NVIDIA 2050, 186 368 cores

TOP 500 Super Computers

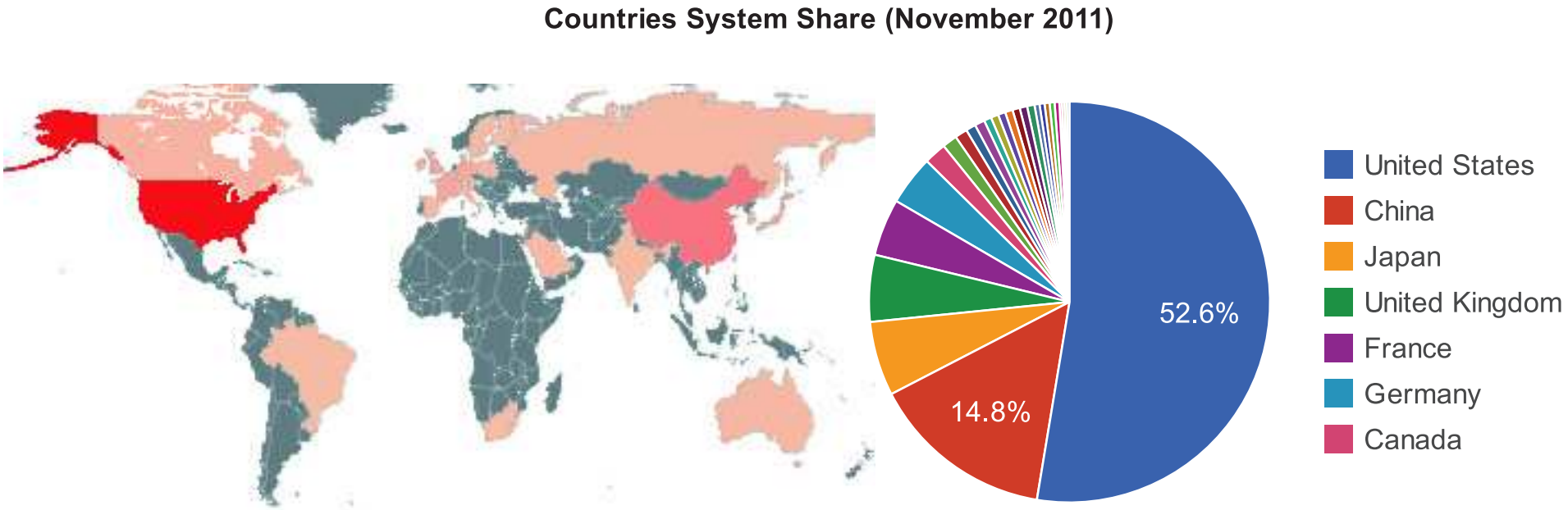


image source: [9]

- Tianhe-1A: #2 in top 500 super computers 2011 [9], at the NUDT in Tianjin, Xeon X5670 6C 2.93 GHz, NVIDIA 2050, 186 368 cores, 2 566 000 GFlops/s in Linpack

TOP 500 Super Computers

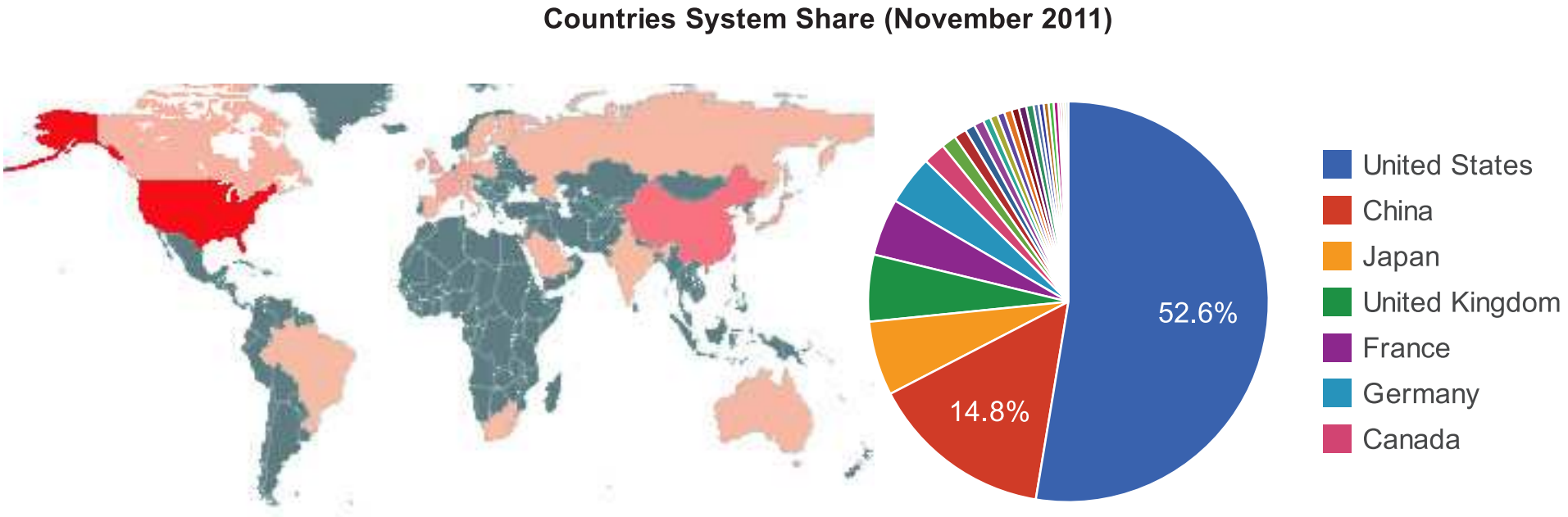


image source: [9]

- Tianhe-1A: #2 in top 500 super computers 2011 [9]
- Nebulae

TOP 500 Super Computers

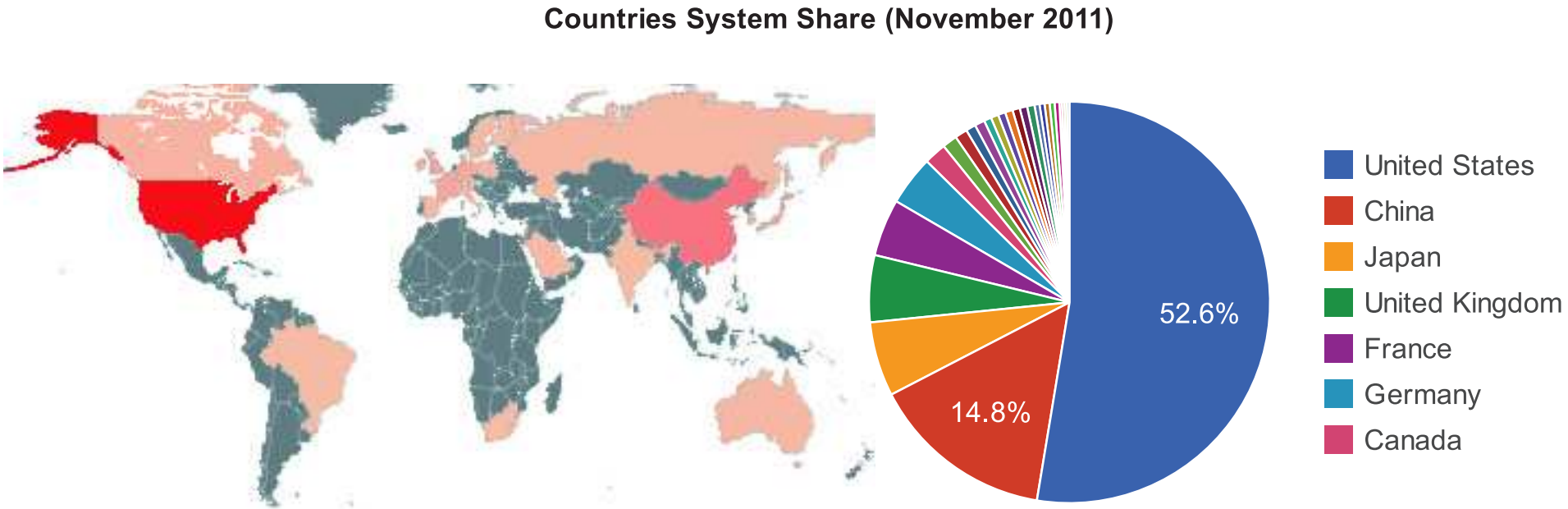


image source: [9]

- Tianhe-1A: #2 in top 500 super computers 2011 [9]
- Nebulae: #4 in top 500 super computers 2011 [9], at the National Supercomputing Centre in Shenzhen (NSCS), Xeon X5650 6C 2.66GHz, Infiniband QDR, NVIDIA 2050

TOP 500 Super Computers

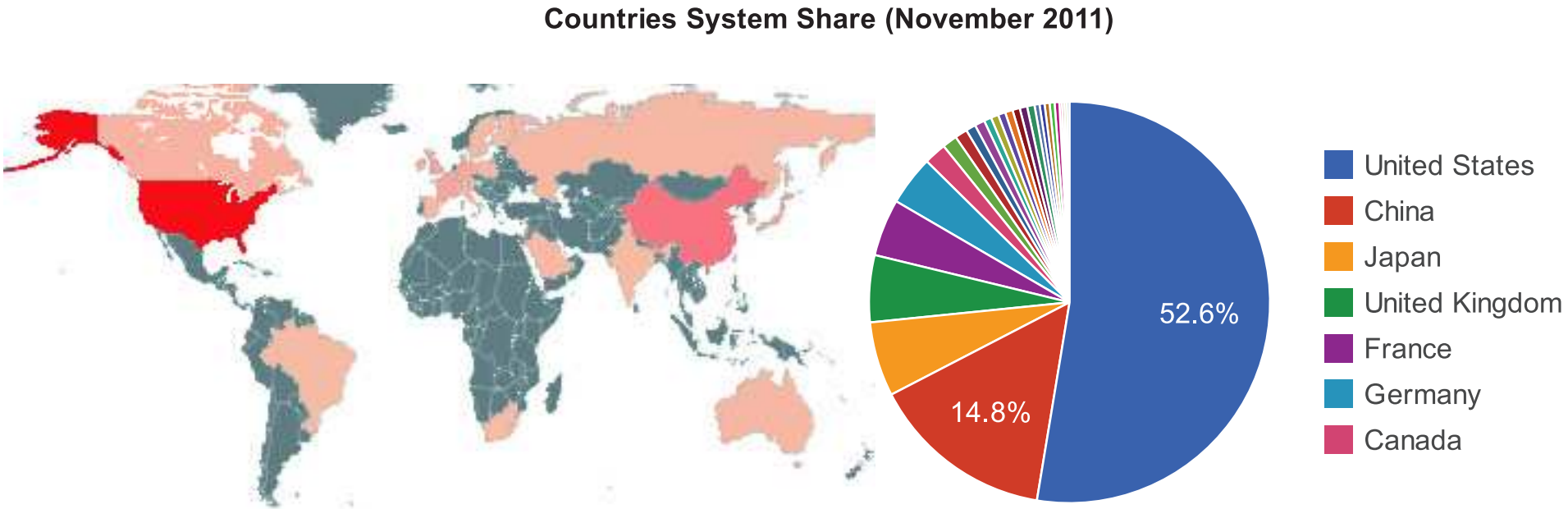


image source: [9]

- Tianhe-1A: #2 in top 500 super computers 2011 [9]
- Nebulae: #4 in top 500 super computers 2011 [9], at the National Supercomputing Centre in Shenzhen (NSCS), Xeon X5650 6C 2.66GHz, Infiniband QDR, NVIDIA 2050, 120 640 cores

TOP 500 Super Computers

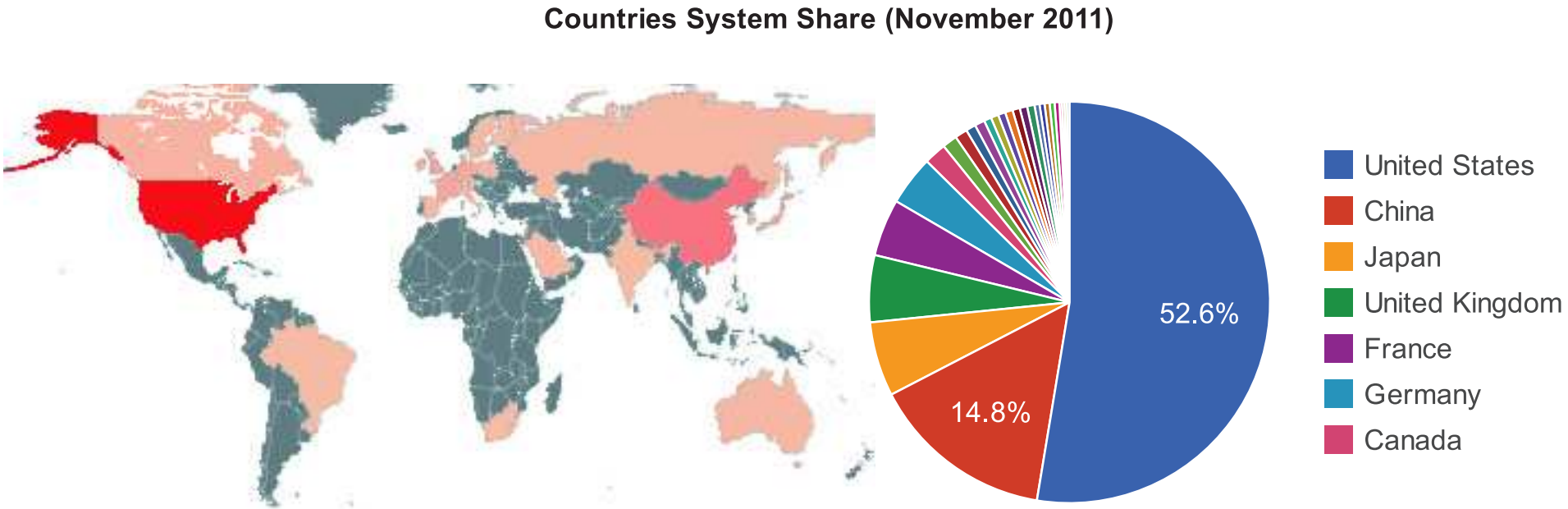


image source: [9]

- Tianhe-1A: #2 in top 500 super computers 2011 [9]
- Nebulae: #4 in top 500 super computers 2011 [9], at the National Supercomputing Centre in Shenzhen (NSCS), Xeon X5650 6C 2.66GHz, Infiniband QDR, NVIDIA 2050, 120 640 cores, 1 271 000 GFlops/s in Linpack

TOP 500 Super Computers

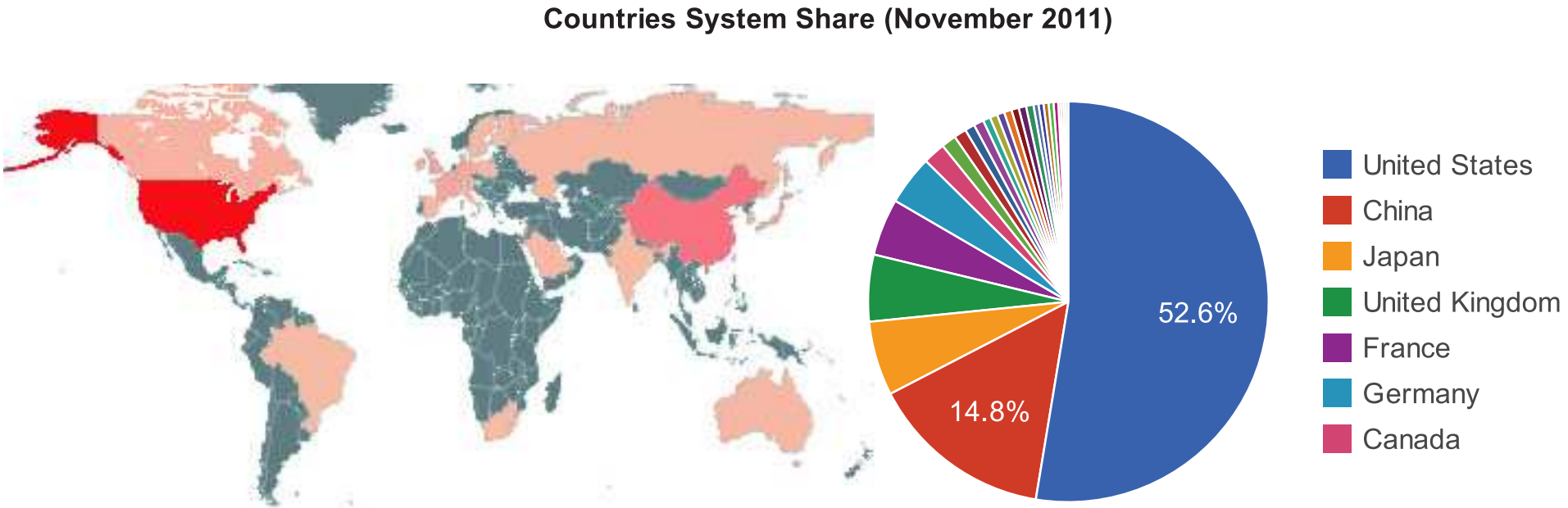


image source: [9]

- Tianhe-1A: #2 in top 500 super computers 2011 [9]
- Tianhe-2

TOP 500 Super Computers

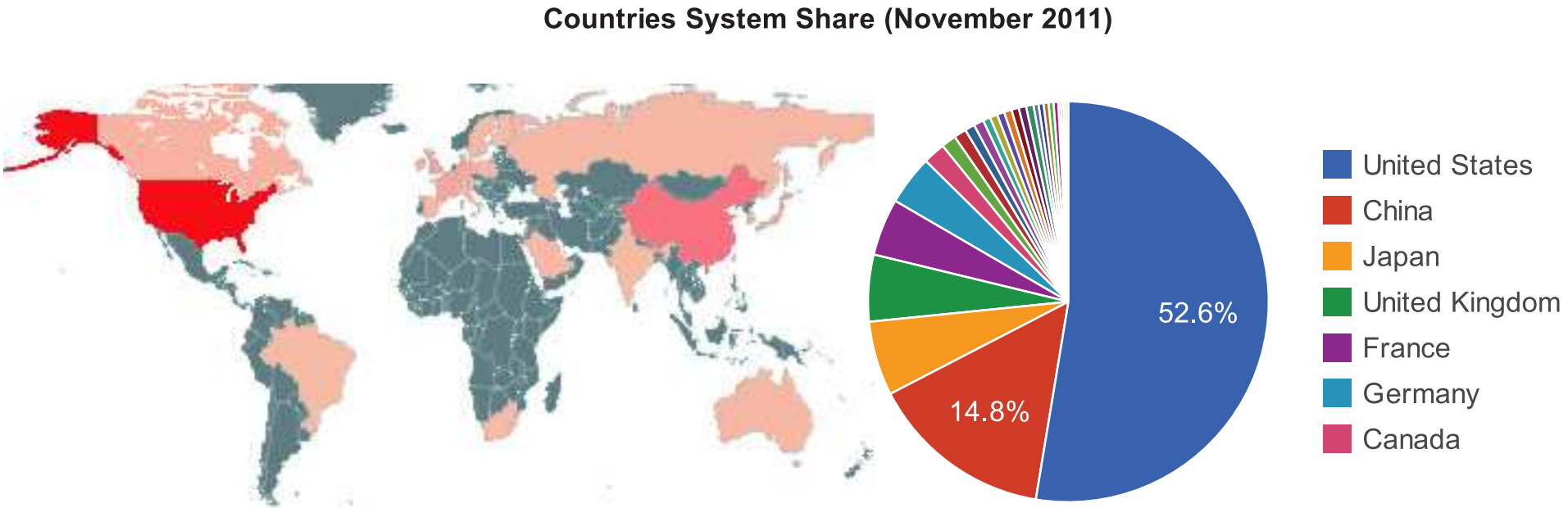


image source: [9]

- Tianhe-1A: #2 in top 500 super computers 2011 [9]
- Tianhe-2: fastest super computer in 2015, fully Intel-based

TOP 500 Super Computers

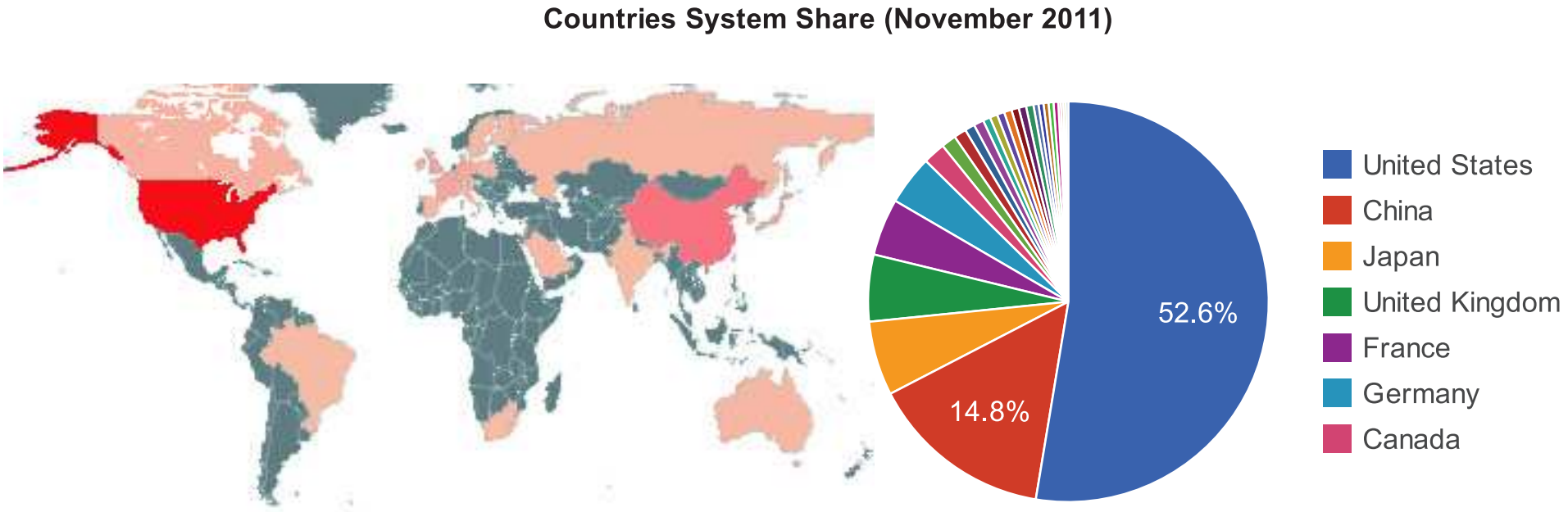


image source: [9]

- Tianhe-1A: #2 in top 500 super computers 2011 [9]
- Tianhe-2: fastest super computer in 2015, fully Intel-based USA moved super computing center + NUDT + Tianhe center on 'Denial List'[10] . . . . . now wants to build faster super computer using Intel tech[11]

Divide & Conquer

- For large-scale problems, a single thread of execution may be slow

Divide & Conquer

- For large-scale problems, a single thread of execution may be too slow...

Divide & Conquer

- For large-scale problems, a single thread of execution may be too slow...

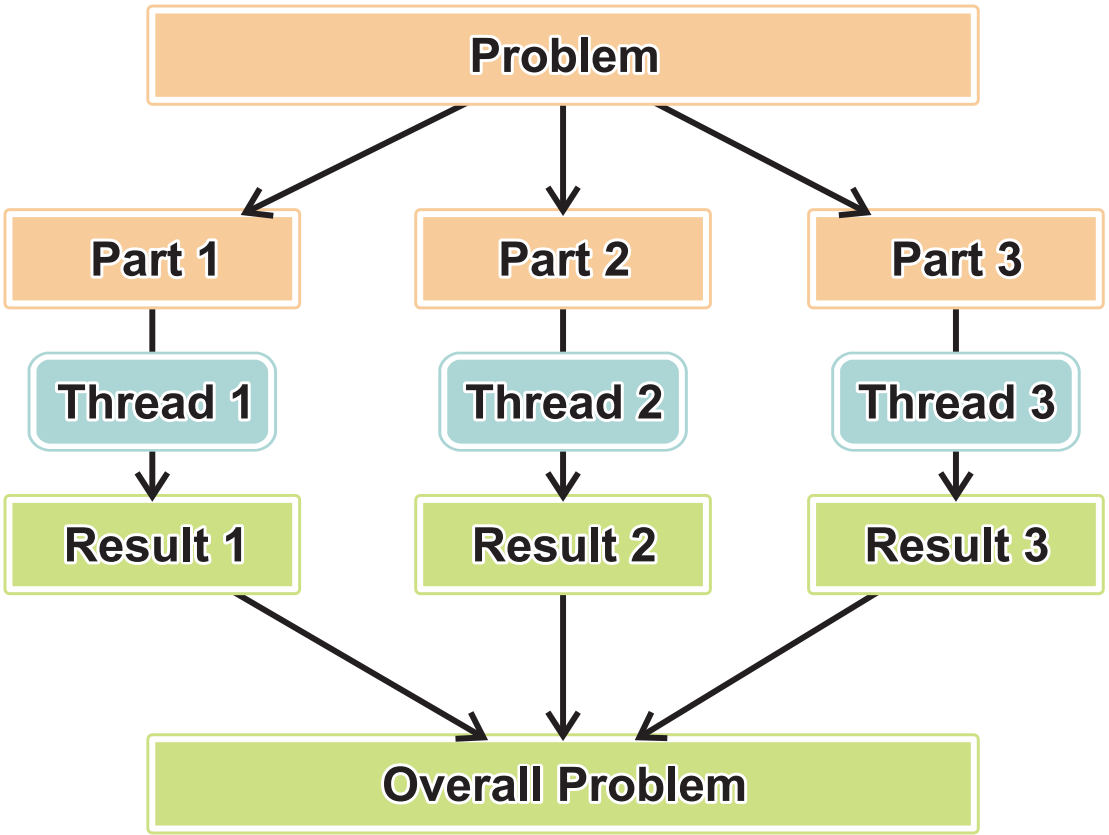


image source: [12]

Partition = Divide

Conquer

Combine

Scalability

- One argument for distributed computing is scalability

Scalability

- One argument for distributed computing is scalability, i.e.,
  more computers = more computing power = more work can be done at the same time

Scalability

- One argument for distributed computing is scalability, i.e., more computers = more computing power = more work can be done at the same time
- Let's say we have a large-scale simulation of the air over China for predicting the weather.

Scalability

- One argument for distributed computing is scalability, i.e., more computers = more computing power = more work can be done at the same time
- Let's say we have a large-scale simulation of the air over China for predicting the weather.
- Let's make it faster!

Scalability

- One argument for distributed computing is scalability, i.e., more computers = more computing power = more work can be done at the same time
- Let's say we have a large-scale simulation of the air over China for predicting the weather.
- Let's make it faster!
- But how fast can we get at most?

Scalability

- One argument for distributed computing is scalability, i.e., more computers = more computing power = more work can be done at the same time
- Let's say we have a large-scale simulation of the air over China for predicting the weather.
- Let's make it faster!
- But how fast can we get at most?
- Does using p computers mean we can solve a task in $\frac{1}{p}$ of the time on a single computer?

No.

- Of course not.

No.

- Of course not.
- We need communication

No.

- Of course not.
- We need communication: Data must be sent to each computer and the computers must send back their results [13].

No.

- Of course not.
- We need communication: Data must be sent to each computer and the computers must send back their results [13]. This takes time.

No.

- Of course not.
- We need communication: Data must be sent to each computer and the computers must send back their results [13]. This takes time.
- Some things cannot be parallelized.

No.

- Of course not.
- We need communication: Data must be sent to each computer and the computers must send back their results [13]. This takes time.
- Some things cannot be parallelized. E.g. loading of files, initialization of variables...

No.

- Of course not.
- We need communication: Data must be sent to each computer and the computers must send back their results [13]. This takes time.
- Some things cannot be parallelized. E.g. loading of files, initialization of variables...
- Thus, even with $p \to \infty$ processors, we cannot finish a task in 0 time...

No.

- Of course not.
- We need communication: Data must be sent to each computer and the computers must send back their results [13]. This takes time.
- Some things cannot be parallelized. E.g. loading of files, initialization of variables...
- Thus, even with $p \to \infty$ processors, we cannot finish a task in 0 time...
- This is true for any distributed application...

No.

- Of course not.
- We need communication: Data must be sent to each computer and the computers must send back their results [13]. This takes time.
- Some things cannot be parallelized. E.g. loading of files, initialization of variables...
- Thus, even with $p \to \infty$ processors, we cannot finish a task in 0 time...
- This is true for any distributed application...
- But how much faster can we go, if we include the time needed for communication into our considerations?

Limits of Scalability

- Speed-up S means: we can be S times faster!

Limits of Scalability

- Speed-up S means: we can be S times faster!

$$S \tag{1}$$

S . . . the speed-up

Limits of Scalability

- Speed-up S means: we can be S times faster!

$$S = \frac{T_{seq}}{T_{par}} \tag{1}$$

S ... the speed-up
T_{seq} ... runtime when sequential

- Speed-up S means: we can be S times faster!

$$S = \frac{T_{seq}}{T_{par}} = \frac{T_{seq}}{T_{seq}\left[\alpha + c_p * \beta + \frac{1-\alpha-\beta}{p}\right]} \tag{1}$$

S ... the speed-up
T_{seq} ... runtime when sequential
α ... fraction of sequential instructions, e.g., for start-up
c_p ... value depending on communication model
T_{par} ... runtime when parallelized
β ... fraction of instructions needed for communication
p ... number of nodes/processors

- Speed-up S means: we can be S times faster!

$$S = \frac{T_{seq}}{T_{par}} = \frac{1}{\alpha + c_p * \beta + \frac{1-\alpha-\beta}{p}} \tag{1}$$

S... the speed-up
T_{seq}... runtime when sequential
α... fraction of sequential instructions, e.g., for start-up
c_p... value depending on communication model
T_{par}... runtime when parallelized
β... fraction of instructions needed for communication
p... number of nodes/processors

Limits of Scalability

- Speed-up S means: we can be S times faster!

$$S = \frac{T_{seq}}{T_{par}} = \frac{1}{\alpha + c_p * \beta + \frac{1-\alpha-\beta}{p}} \quad (1)$$

part of the program that cannot be parallelized, e.g., initialization, reading of data from a file, writing output to file, etc.

S

T_{seq}

α

tions, e.g., for start-up

c_p . . . value depending on comm

tion model

Limits of Scalability

- Speed-up S means: we can be S times faster!

$$S = \frac{T_{seq}}{T_{par}} = \frac{1}{\alpha + c_p * \beta + \frac{1-\alpha-\beta}{p}} \quad (1)$$

part of the program that cannot be parallelized, e.g., initialization, reading of data from a file, writing output to file, etc.

part of the program that performs the communication: tell all nodes what to do; receive the results from the nodes

S

T_{seq}

α

tions, e.g., for start-up

c_p . . . value depending on comm tion model

ors

Limits of Scalability

- Speed-up S means: we can be S times faster!

$$S = \frac{T_{seq}}{T_{par}} = \frac{1}{\alpha + c_p * \beta + \frac{1-\alpha-\beta}{p}} \quad (1)$$

part of the program that cannot be parallelized, e.g., initialization, reading of data from a file, writing output to file, etc.

part of the program that performs the communication: tell all nodes what to do; receive the results from the nodes

the rest: stuff that can be parallelized

S

T_{seq}

α

tions, e.g., for start-up

c_p . . . value depending on comm tion model

- Speed-up S means: we can be S times faster!

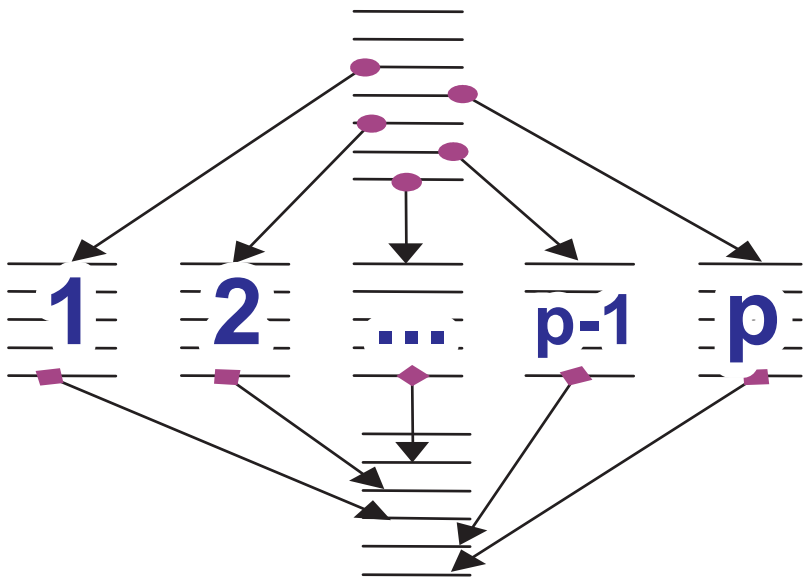
$$S = \frac{T_{seq}}{T_{par}} = \frac{1}{\alpha + c_p * \beta + \frac{1-\alpha-\beta}{p}} \tag{1}$$

S... the speed-up
T_{seq}... runtime when sequential
α... fraction of sequential instructions, e.g., for start-up
c_p... value depending on communication model
T_{par}... runtime when parallelized
β... fraction of instructions needed for communication
p... number of nodes/processors

- Speed-up S means: we can be S times faster!

$$S = \frac{T_{seq}}{T_{par}} = \frac{1}{\alpha + c_p * \beta + \frac{1-\alpha-\beta}{p}} \tag{1}$$

S... the speed-up
T_{seq}... runtime when sequential
α... fraction of sequential instructions, e.g., for start-up
c_p... value depending on communication model

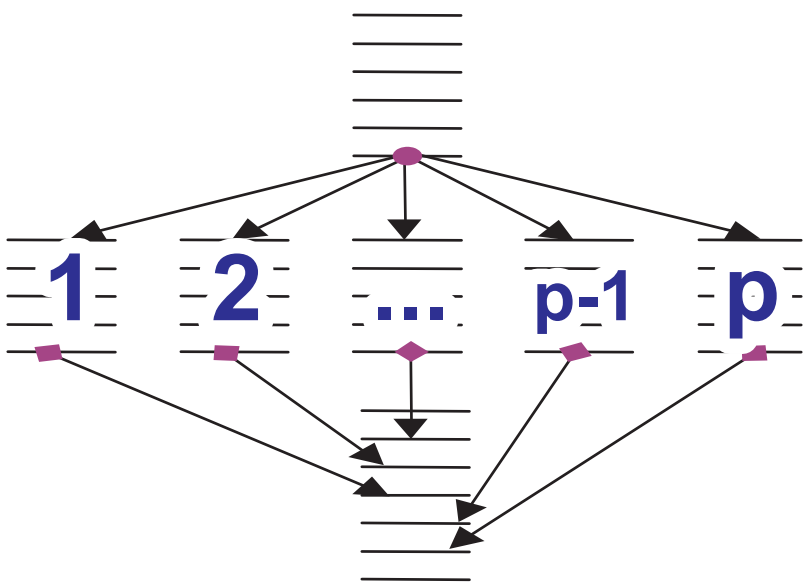


unicast: $c_p = 2p$

- Speed-up S means: we can be S times faster!

$$S = \frac{T_{seq}}{T_{par}} = \frac{1}{\alpha + c_p * \beta + \frac{1-\alpha-\beta}{p}} \tag{1}$$

S ... the speed-up
T_{seq} ... runtime when sequential
α ... fraction of sequential instructions, e.g., for start-up
c_p ... value depending on communication model



multicast: $c_p = 1 + p$

Amdahl's Law

$$S = \frac{1}{\alpha + c_p * \beta + \frac{1-\alpha-\beta}{p}} \tag{2}$$

(5)

α... fraction of sequential instructions
β... fraction of instructions needed for communication
c_p... value depending on communication model
p... number of nodes/processors

Amdahl's Law

$$S = \frac{1}{\alpha + c_p * \beta + \frac{1-\alpha-\beta}{p}} \tag{2}$$

$$\lim_{\beta \to 0} S = \frac{1}{\alpha + \frac{1-\alpha}{p}} \quad \text{(classical Amdahl's Law [14])} \quad \text{ignore communication, } \beta \to 0 \tag{3}$$

(5)

α... fraction of sequential instructions
β... fraction of instructions needed for communication
c_p... value depending on communication model
p... number of nodes/processors

Amdahl's Law

$$S = \frac{1}{\alpha + c_p * \beta + \frac{1-\alpha-\beta}{p}} \tag{2}$$

$$\lim_{\beta \to 0} S = \frac{1}{\alpha + \frac{1-\alpha}{p}} \quad \text{(classical Amdahl's Law [14])} \quad \text{ignore communication, } \beta \to 0 \tag{3}$$

$$\beta = 0 \Rightarrow \lim_{p \to \infty} S = \frac{1}{\alpha} \quad \text{no communication} \wedge \infty \text{ processors} \tag{4}$$

(5)

α... fraction of sequential instructions
β... fraction of instructions needed for communication
c_p... value depending on communication model
p... number of nodes/processors

Amdahl's Law

$$S = \frac{1}{\alpha + c_p * \beta + \frac{1-\alpha-\beta}{p}} \tag{2}$$

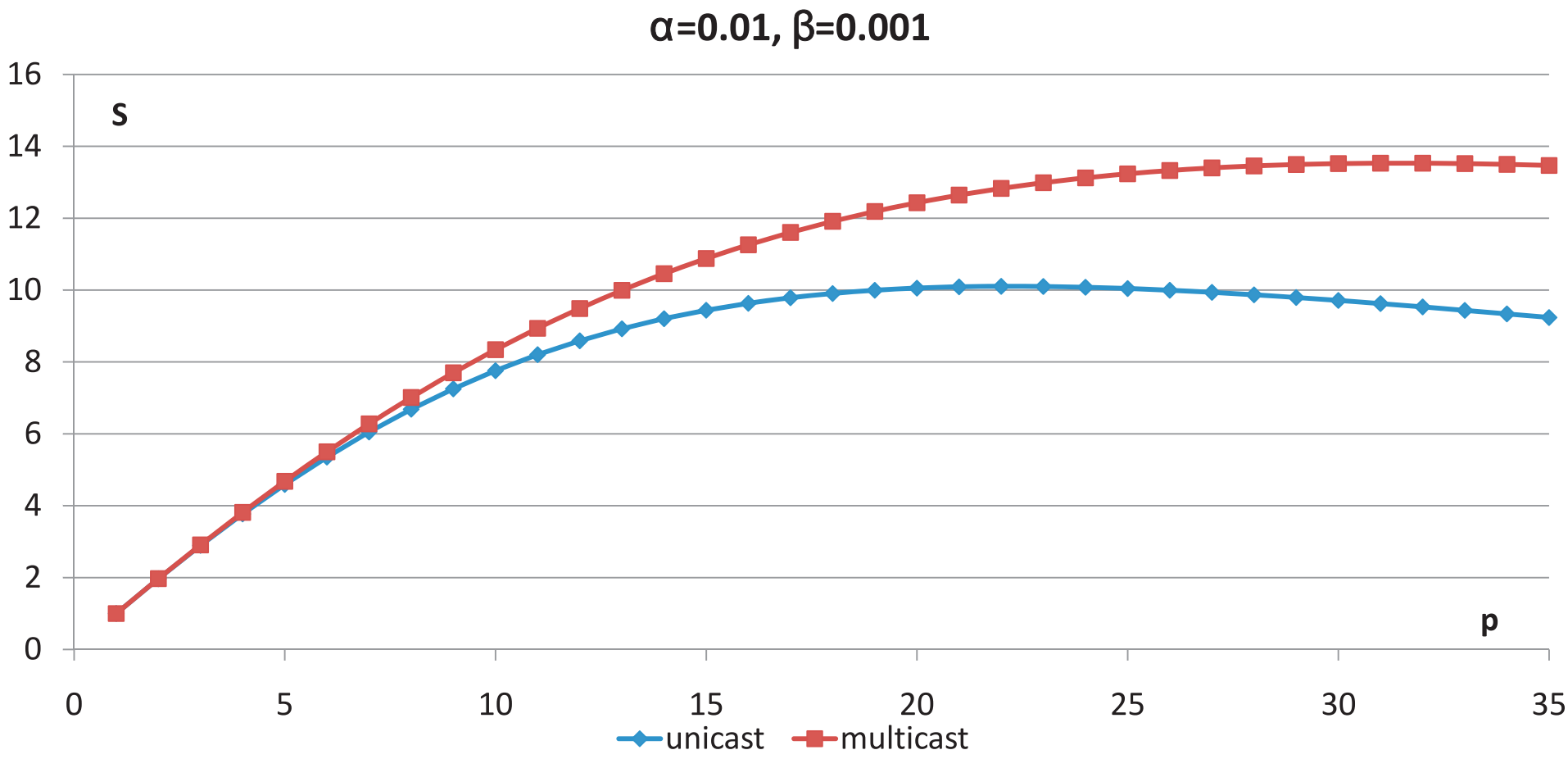
$$\lim_{\beta \to 0} S = \frac{1}{\alpha + \frac{1-\alpha}{p}} \quad \text{(classical Amdahl's Law [14])} \quad \text{ignore communication, } \beta \to 0 \tag{3}$$

$$\beta = 0 \Rightarrow \lim_{p \to \infty} S = \frac{1}{\alpha} \quad \text{no communication} \wedge \infty \text{ processors} \tag{4}$$

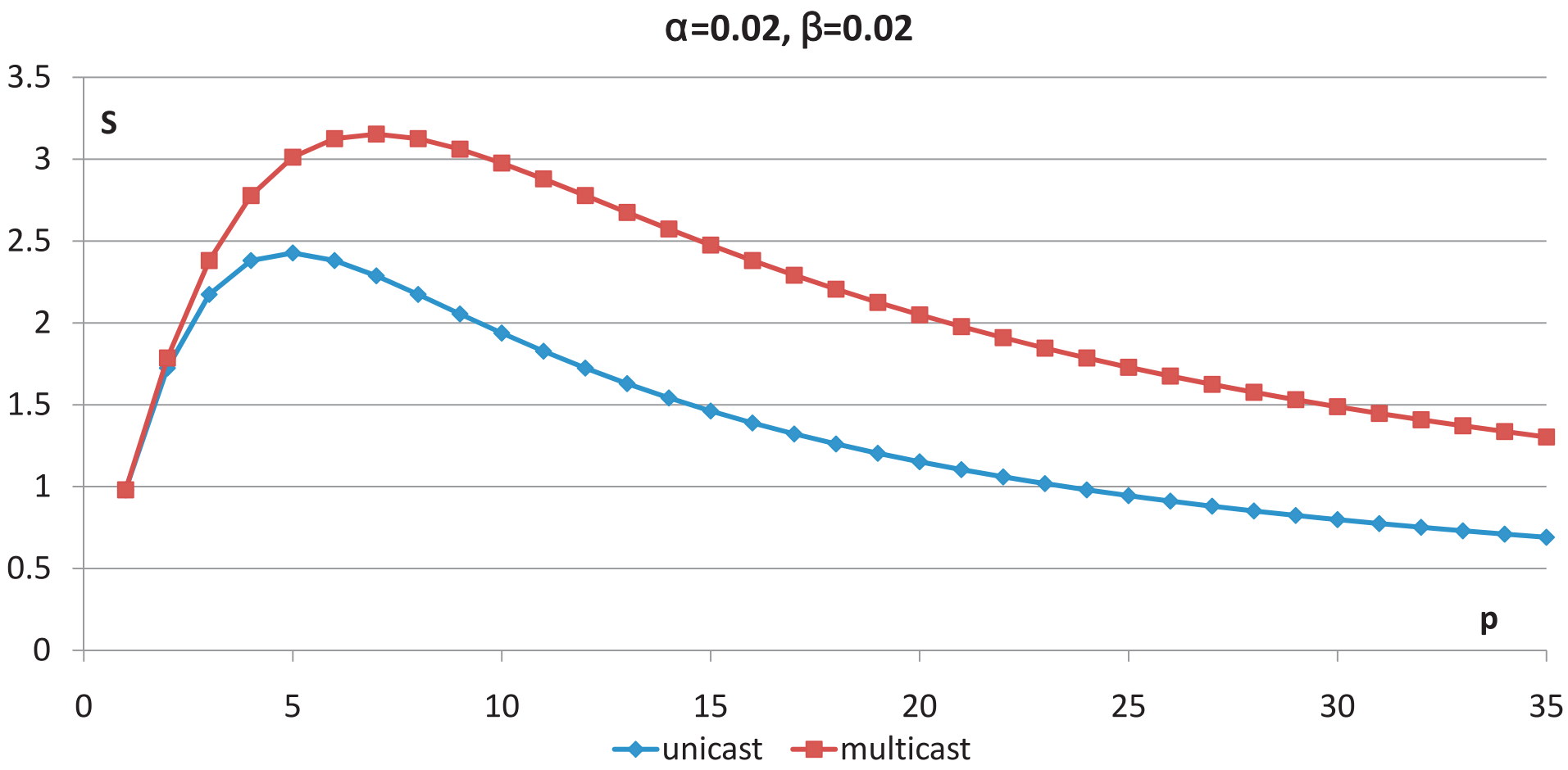
$$\beta \neq 0 \Rightarrow \lim_{p \to \infty} S = 0 \quad \text{communication} \wedge \infty \text{ processors} \tag{5}$$

α... fraction of sequential instructions
β... fraction of instructions needed for communication
c_p... value depending on communication model
p... number of nodes/processors

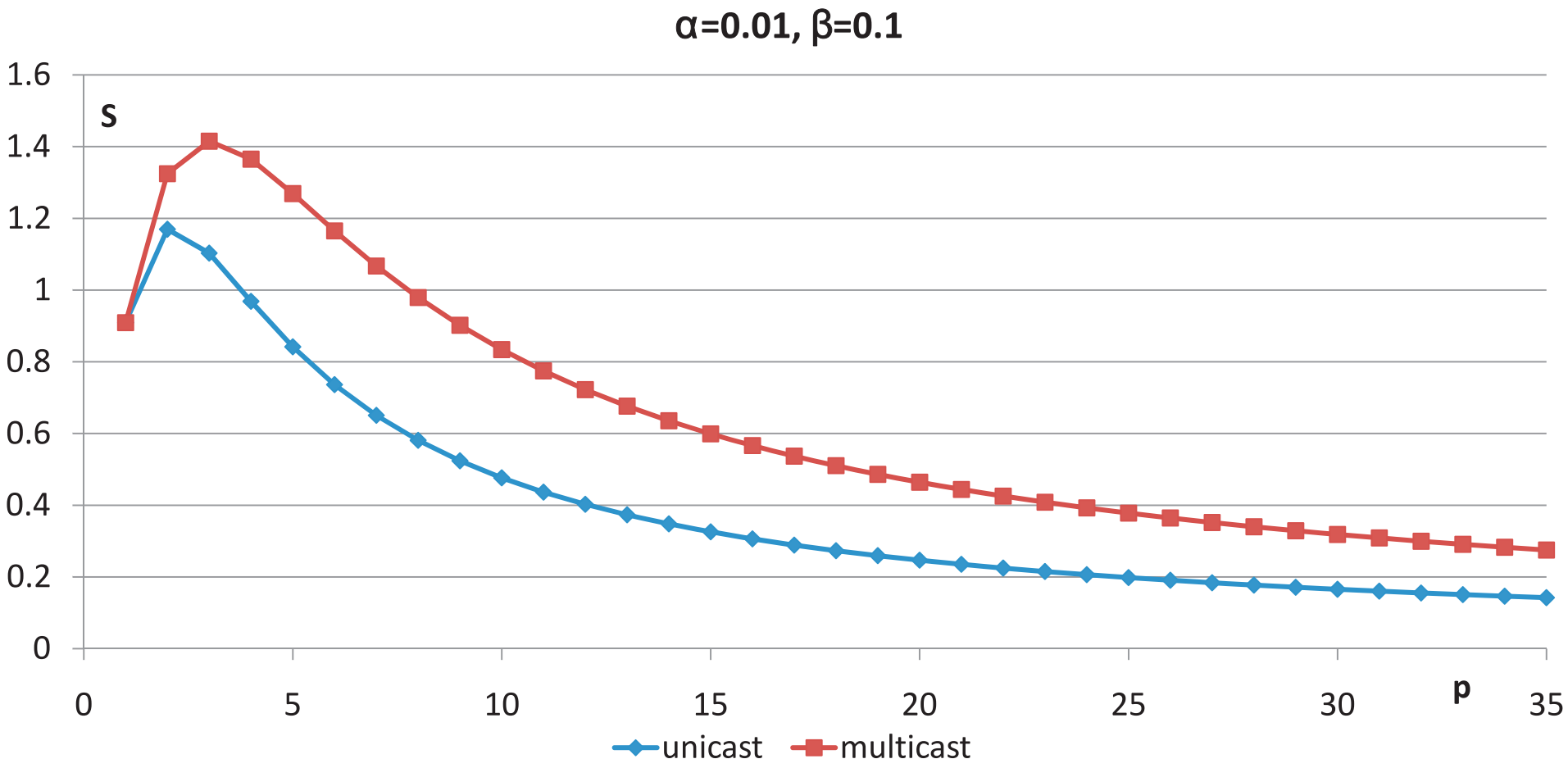
Amdahl's Law


α=0.01, β=0.001
S
p
0
2
4
6
8
10
12
14
16
0
5
10
15
20
25
30
35
unicast
multicast

Amdahl's Law


α=0.02, β=0.02
S
p
3.5
3
2.5
2
1.5
1
0.5
0
0
5
10
15
20
25
30
35
unicast
multicast

Amdahl's Law


α=0.01, β=0.1
S
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
0
5
10
15
20
25
30
35
p
unicast
multicast

Amdahl's Law: Consequences

- Speedup can be high but is always limited

Amdahl's Law: Consequences

- Speedup can be high but is always limited
- Two lessons valid for every parallel or distributed application

Amdahl's Law: Consequences

- Speedup can be high but is always limited
- Two lessons valid for every parallel or distributed application:
  - Try to communicate as little as possible

Amdahl's Law: Consequences

- Speedup can be high but is always limited
- Two lessons valid for every parallel or distributed application:
  - Try to communicate as little as possible
  - Try to minimize the fraction of sequential code and increase fraction of the code that can run in parallel

Algorithm Perspective

- There are two basic use cases for parallelization [15]

Algorithm Perspective

- There are two basic use cases for parallelization [15]
  1. A set of unrelated jobs is handed to several different threads, each one carrying out one distinct job

Algorithm Perspective

- There are two basic use cases for parallelization [15]
  1. A set of unrelated jobs is handed to several different threads, each one carrying out one distinct job
     Example: n different experiments or simulations with a certain algorithm

Algorithm Perspective

- There are two basic use cases for parallelization [15]
  1. A set of unrelated jobs is handed to several different threads, each one carrying out one distinct job
     Example: n different experiments or simulations with a certain algorithm
  2. Each job can somehow be broken into pieces which can be solved *cooperatively* by different computers

Algorithm Perspective

- There are two basic use cases for parallelization [15]
  1. A set of unrelated jobs is handed to several different threads, each one carrying out one distinct job
     Example: n different experiments or simulations with a certain algorithm
  2. Each job can somehow be broken into pieces which can be solved *cooperatively* by different computers
     Example: Evolutionary Algorithm with a population distributed over several threads/processors/computers [6, 16, 17]

Algorithm Perspective

- Especially the latter scenario is interesting for us here (the other is trivial).

Algorithm Perspective

- Especially the latter scenario is interesting for us here (the other is trivial).
- We can distinguish four kinds of problems [18]

Algorithm Perspective

- Especially the latter scenario is interesting for us here (the other is trivial).
- We can distinguish four kinds of problems [18]:
  1. Parallel Problems
  2. Regular Problems
  3. Irregular Problems

Algorithm Perspective

- Especially the latter scenario is interesting for us here (the other is trivial).
- We can distinguish four kinds of problems [18]:
  1. Parallel Problems
  2. Regular Problems
  3. Irregular Problems
  4. Any combination of the above: division into parts of the above types

Parallel Problems

- The problem can be broken down into parts

Parallel Problems

- The problem can be broken down into parts
- The parts are independent from each other (similar to the first case in the introduction)

Parallel Problems

- The problem can be broken down into parts
- The parts are independent from each other (similar to the first case in the introduction)
- Communication only needed for sending the parts to different workstations and finally gathering the results

Parallel Problems

- The problem can be broken down into parts
- The parts are independent from each other (similar to the first case in the introduction)
- Communication only needed for sending the parts to different workstations and finally gathering the results
- Almost linear speed-up can be expected (Amdahl's Law [14])

Parallel Problems

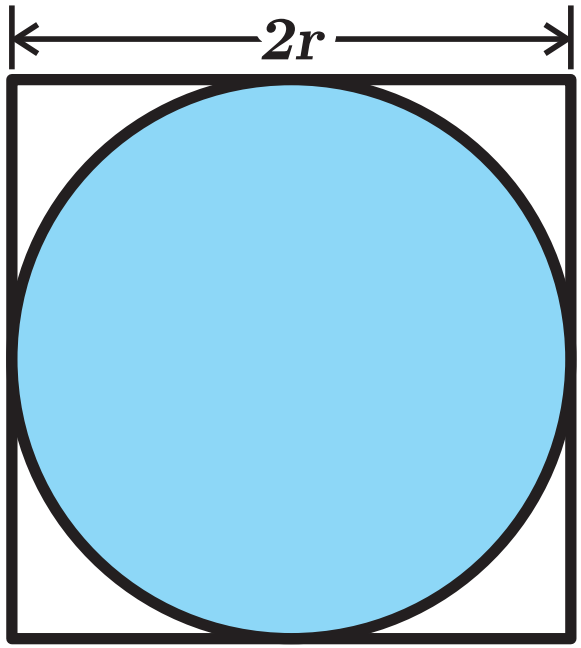
- The problem can be broken down into parts
- The parts are independent from each other (similar to the first case in the introduction)
- Communication only needed for sending the parts to different workstations and finally gathering the results
- Almost linear speed-up can be expected (Amdahl's Law [14])
- This is the ideal case!

Parallel Problems

- The problem can be broken down into parts
- The parts are independent from each other (similar to the first case in the introduction)
- Communication only needed for sending the parts to different workstations and finally gathering the results
- Almost linear speed-up can be expected (Amdahl's Law [14])
- This is the ideal case!
- *Examples:* simple matrix-vector products, rendering of fractals ($\rightarrow$ see your homework)

Estimation of π

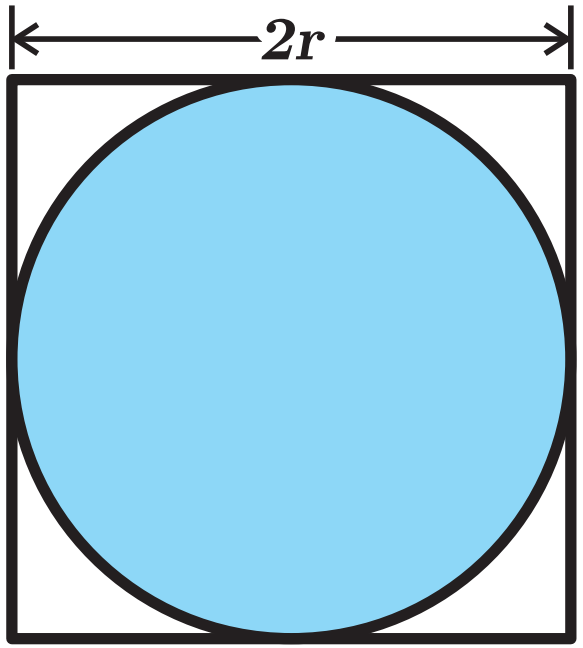
- Simple way to estimate the value of π [18, 19]



Estimation of π

- Simple way to estimate the value of π [18, 19]

$$A_s = (2r)^2 = 4r^2 \tag{8}$$

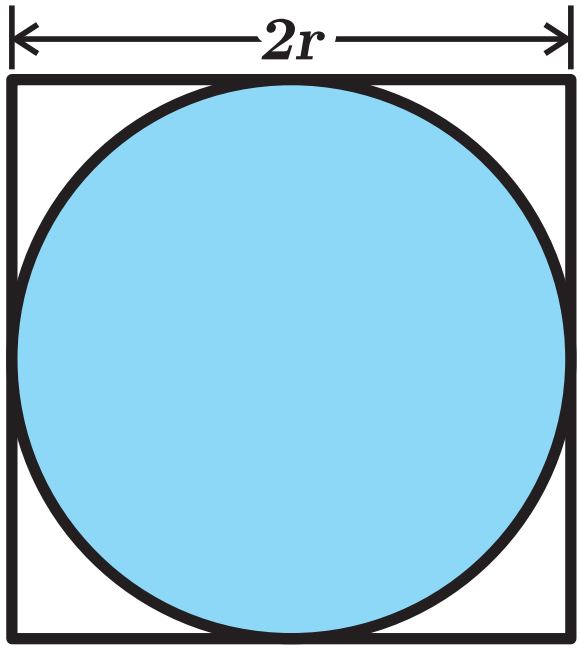


Estimation of π

- Simple way to estimate the value of π [18, 19]

$$A_s = (2r)^2 = 4r^2 \quad (6)$$

$$A_c = \pi r^2 \quad (8)$$



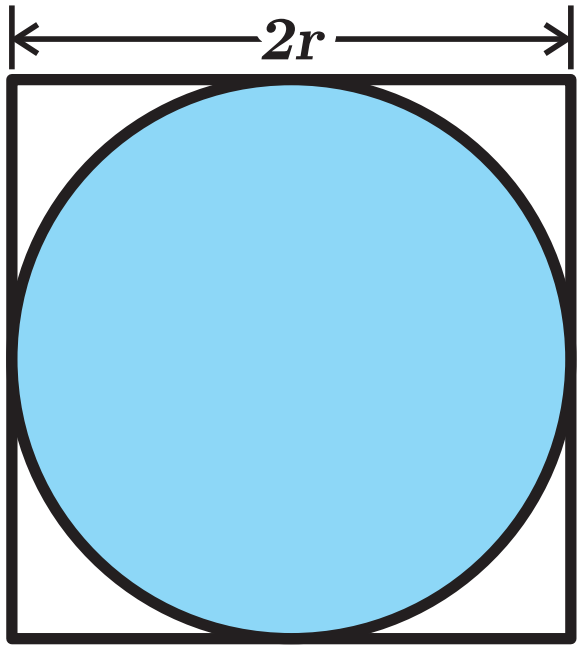
Estimation of π

- Simple way to estimate the value of π [18, 19]

$$A_s = (2r)^2 = 4r^2 \quad (6)$$

$$A_c = \pi r^2 \quad (7)$$

$$\pi = 4\frac{A_c}{A_s} = 4 * \left(\frac{\pi r^2}{4r^2}\right) \quad (8)$$



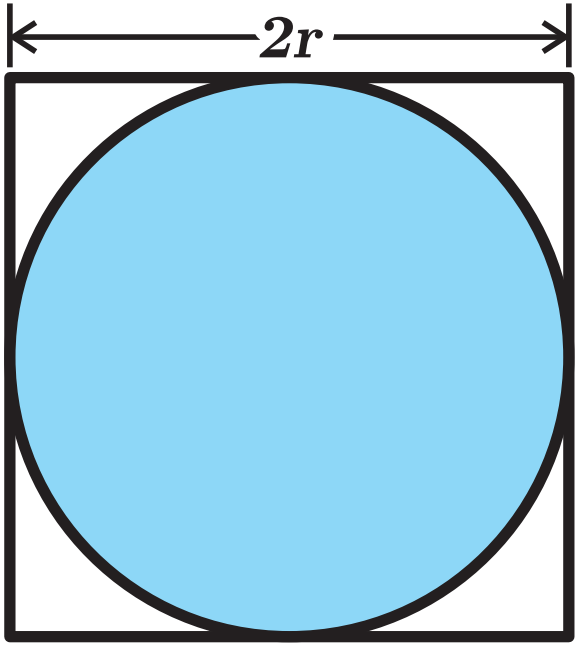
Estimation of π

- Simple way to estimate the value of π [18, 19]

$$A_s = (2r)^2 = 4r^2 \quad (6)$$

$$A_c = \pi r^2 \quad (7)$$

$$\pi = 4\frac{A_c}{A_s} = 4 * \left(\frac{\pi r^2}{4r^2}\right) \quad (8)$$



- Randomly generate n points in a square

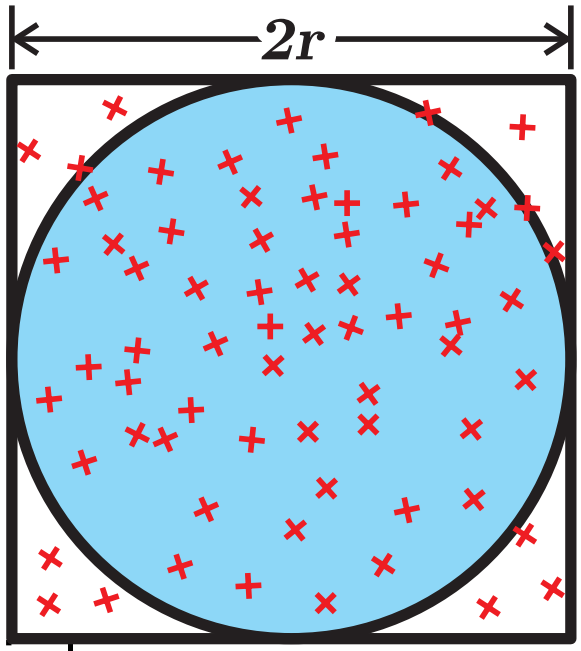
Estimation of π

- Simple way to estimate the value of π [18, 19]

$$A_s = (2r)^2 = 4r^2 \quad (6)$$

$$A_c = \pi r^2 \quad (7)$$

$$\pi = 4\frac{A_c}{A_s} = 4 * \left(\frac{\pi r^2}{4r^2}\right) \quad (8)$$



- Randomly generate n points in a square
- Count the number c of points falling into the circle

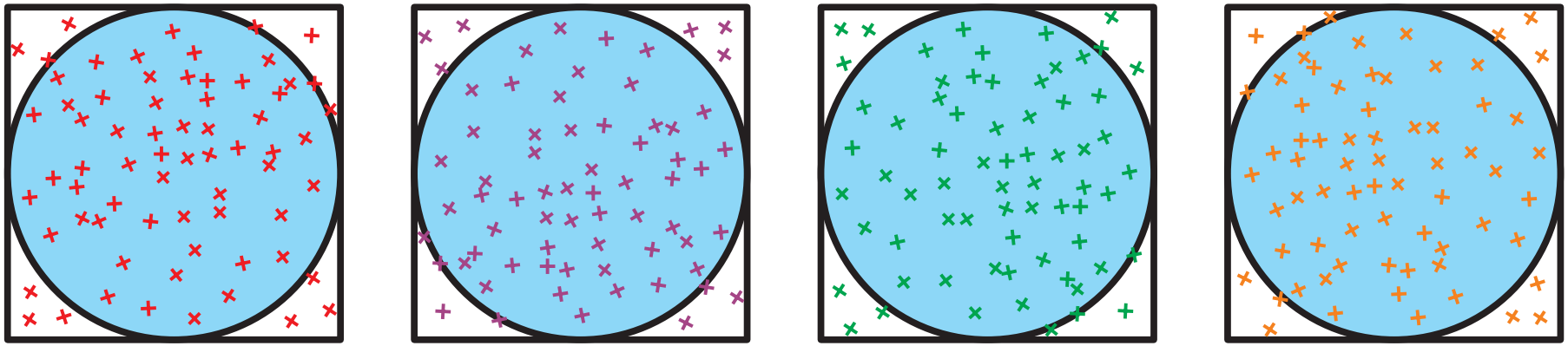
$$\pi \approx \frac{4c}{n} \quad (9)$$

Parallel Estimation of π

- Ideal for parallelization and distribution [18, 19]

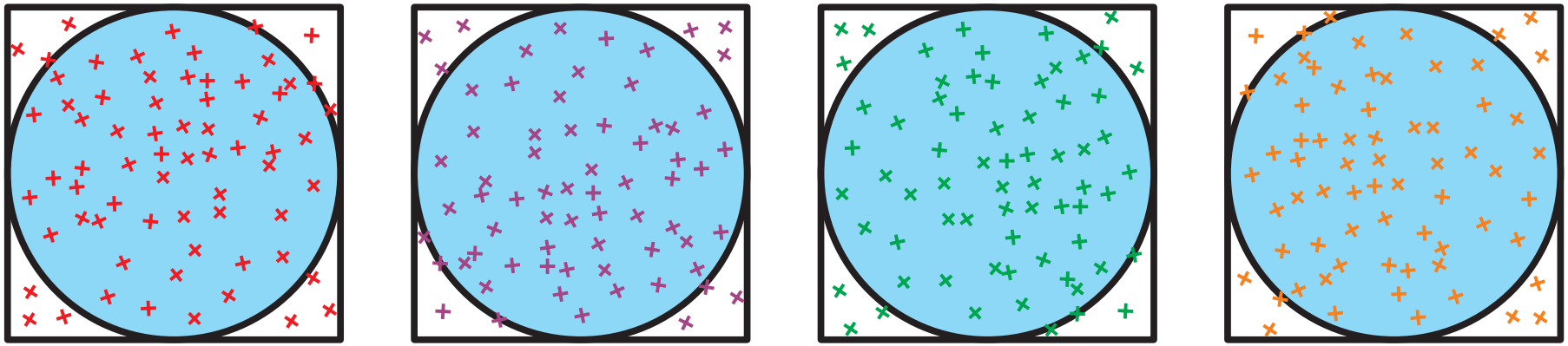
Parallel Estimation of π

- Ideal for parallelization and distribution [18, 19]:
  - Let p threads each create $\frac{n}{p}$ random points...



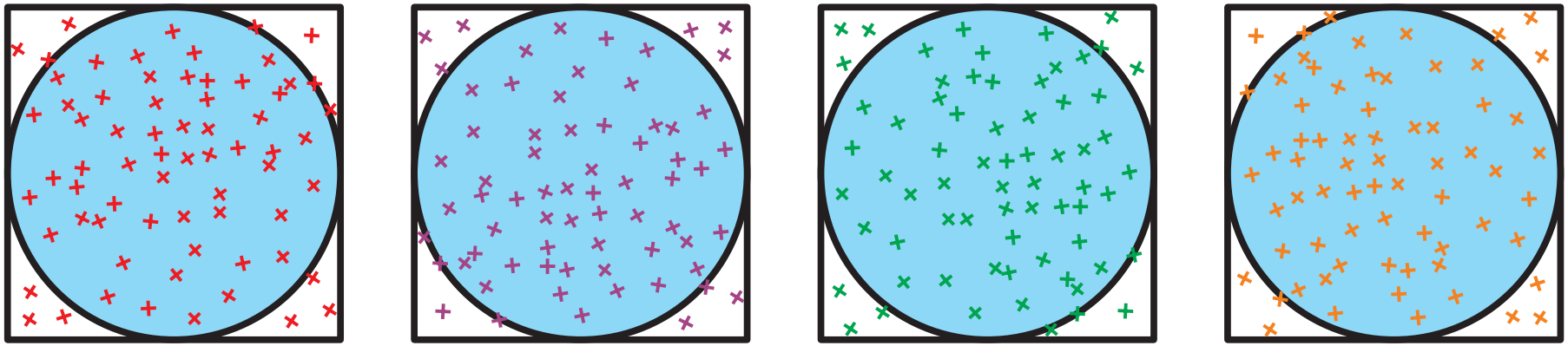
Parallel Estimation of π

- Ideal for parallelization and distribution [18, 19]:
  - Let p threads each create $\frac{n}{p}$ random points...
  - ...and combine the results.



Parallel Estimation of π

- Ideal for parallelization and distribution [18, 19]:
  - Let p threads each create $\frac{n}{p}$ random points...
  - ...and combine the results.
- No communication between workers necessary!



Listing: Server program estimating π (PiServer.java).

```
import java.io.ByteArrayInputStream;    import java.io.ByteArrayOutputStream;   import
    java.io.DataInputStream;
import java.io.DataOutputStream;        import java.io.OutputStream;            import
    java.net.DatagramPacket;
import java.net.DatagramSocket;         import java.net.InetAddress;

public class PiServer {
  public static final void main(final String[] args) {
    DatagramSocket        server;   DatagramPacket    p, answer;
    ByteArrayInputStream  bis;      DataInputStream   dis;
    byte[]                data;     String            s;
    long                  n, c;     double            d;

    n=0;c=0;//try to approximate PI
    try {
      server = new DatagramSocket(9992);//create server socket
      data   = new byte[16];//create package: 2* 8 byte long ints must fit

      for (;;) {//forever
        p = new DatagramPacket(data, data.length);//create new package
        server.receive(p); //wait for and receive incoming data

        bis = new ByteArrayInputStream(data, 0, p.getLength());//wrap data into stream
        dis = new DataInputStream(bis);//unmarshall data

        n += dis.readLong(); //update total number of random points sampled from unit square
        c += dis.readLong(); //update number of these points that fell into the unit circle
        d  = ((4.0 * c) / n);//approximate PI
        System.out.println(d + "␣" + (d - Math.PI)); //print approximation and error
      }
    } catch (Throwable t) {
      t.printStackTrace();
    }
  }
}
```

Listing: Client/Slave program for doing the work when computing π (PiClient.java).

```
import java.io.ByteArrayInputStream;   import java.io.ByteArrayOutputStream;   import
    java.io.DataInputStream;   import java.io.DataOutputStream;
import java.net.DatagramPacket;        import java.net.DatagramSocket;         import
    java.net.InetAddress;

public class PiClient { //the worker part of the example for approximating the number of pi
  public static final void main(final String[] args) {
    DatagramSocket  client;         InetAddress           ia;
    ByteArrayOutputStream bos;      DataOutputStream      dos;
    DatagramPacket        p;        byte[]                data;
    long                  c, n;     double                x, y;

    c = 0;//work: approximate fraction of points in unit square which are in unit circle
    for(n = 1; n <= 100000000; n++) {//create a lot of random points in [0, 1)
      x = Math.random(); //x-coordinate of point
      y = Math.random(); //y-coordinate of point
      if(Math.sqrt((x*x) + (y*y)) <= 1d){ //is the point inside the unit circle?
        c++; //count
        }
      }

    try {
      ia = InetAddress.getByName("localhost");//get local host address
      client = new DatagramSocket(); //create UDP/datagram socket

      bos = new ByteArrayOutputStream();//create buffered output stream
      dos = new DataOutputStream(bos);//marshall computed data
      dos.writeLong(n);//store the number of generated points in unit square
      dos.writeLong(c);//store the number of points in unit circle
      dos.close();//close and flush
      data = bos.toByteArray();//get data

      p = new DatagramPacket(data, data.length, ia, 9992);//create data package
      client.send(p);//send the package to the server

      client.close();//close connection
    } catch (Throwable t) {
      t.printStackTrace();
    }
  }
}
```

Matrix-Vector Product

$$\vec{M} \times \vec{v} = \vec{r} \tag{11}$$

Matrix-Vector Product

$$\vec{M} \times \vec{v} = \vec{r} \tag{10}$$

$$\begin{pmatrix} m_{1,1} & m_{1,2} & m_{1,3} & m_{1,4} \\ m_{2,1} & m_{2,2} & m_{2,3} & m_{2,4} \\ m_{3,1} & m_{3,2} & m_{3,3} & m_{3,4} \\ m_{4,1} & m_{4,2} & m_{4,3} & m_{4,4} \end{pmatrix} \times \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{pmatrix} = \begin{pmatrix} r_1 \\ r_2 \\ r_3 \\ r_4 \end{pmatrix} \tag{11}$$

Matrix-Vector Product

$$\vec{M} \times \vec{v} = \vec{r} \tag{10}$$

$$\begin{pmatrix} m_{1,1} & m_{1,2} & m_{1,3} & m_{1,4} \\ m_{2,1} & m_{2,2} & m_{2,3} & m_{2,4} \\ m_{3,1} & m_{3,2} & m_{3,3} & m_{3,4} \\ m_{4,1} & m_{4,2} & m_{4,3} & m_{4,4} \end{pmatrix} \times \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{pmatrix} = \begin{pmatrix} m_{1,1}v_1 + m_{1,2}v_2 + m_{1,3}v_3 + m_{1,4}v_4 \\ m_{2,1}v_1 + m_{2,2}v_2 + m_{2,3}v_3 + m_{2,4}v_4 \\ m_{3,1}v_1 + m_{3,2}v_2 + m_{3,3}v_3 + m_{3,4}v_4 \\ m_{4,1}v_1 + m_{4,2}v_2 + m_{4,3}v_3 + m_{4,4}v_4 \end{pmatrix} \tag{11}$$

Matrix-Vector Product

$$\vec{M} \times \vec{v} = \vec{r} \tag{10}$$

$$\begin{pmatrix} m_{1,1} & m_{1,2} & m_{1,3} & m_{1,4} \\ m_{2,1} & m_{2,2} & m_{2,3} & m_{2,4} \\ m_{3,1} & m_{3,2} & m_{3,3} & m_{3,4} \\ m_{4,1} & m_{4,2} & m_{4,3} & m_{4,4} \end{pmatrix} \times \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{pmatrix} = \begin{pmatrix} m_{1,1}v_1 + m_{1,2}v_2 + m_{1,3}v_3 + m_{1,4}v_4 \\ m_{2,1}v_1 + m_{2,2}v_2 + m_{2,3}v_3 + m_{2,4}v_4 \\ m_{3,1}v_1 + m_{3,2}v_2 + m_{3,3}v_3 + m_{3,4}v_4 \\ m_{4,1}v_1 + m_{4,2}v_2 + m_{4,3}v_3 + m_{4,4}v_4 \end{pmatrix} \tag{11}$$

- Ideal for parallelization and distribution [18]

Matrix-Vector Product

$$\vec{M} \times \vec{v} = \vec{r} \tag{10}$$

$$\begin{pmatrix} m_{1,1} & m_{1,2} & m_{1,3} & m_{1,4} \\ m_{2,1} & m_{2,2} & m_{2,3} & m_{2,4} \\ m_{3,1} & m_{3,2} & m_{3,3} & m_{3,4} \\ m_{4,1} & m_{4,2} & m_{4,3} & m_{4,4} \end{pmatrix} \times \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{pmatrix} = \begin{pmatrix} m_{1,1}v_1 + m_{1,2}v_2 + m_{1,3}v_3 + m_{1,4}v_4 \\ m_{2,1}v_1 + m_{2,2}v_2 + m_{2,3}v_3 + m_{2,4}v_4 \\ m_{3,1}v_1 + m_{3,2}v_2 + m_{3,3}v_3 + m_{3,4}v_4 \\ m_{4,1}v_1 + m_{4,2}v_2 + m_{4,3}v_3 + m_{4,4}v_4 \end{pmatrix} \tag{11}$$

- Ideal for parallelization and distribution [18]
  1. the *root* thread hands each row of matrix M and the whole vector v to one worker
  2. the worker computes the product of the row and v and

Matrix-Vector Product

$$\vec{M} \times \vec{v} = \vec{r} \tag{10}$$

$$\begin{pmatrix} m_{1,1} & m_{1,2} & m_{1,3} & m_{1,4} \\ m_{2,1} & m_{2,2} & m_{2,3} & m_{2,4} \\ m_{3,1} & m_{3,2} & m_{3,3} & m_{3,4} \\ m_{4,1} & m_{4,2} & m_{4,3} & m_{4,4} \end{pmatrix} \times \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{pmatrix} = \begin{pmatrix} m_{1,1}v_1 + m_{1,2}v_2 + m_{1,3}v_3 + m_{1,4}v_4 \\ m_{2,1}v_1 + m_{2,2}v_2 + m_{2,3}v_3 + m_{2,4}v_4 \\ m_{3,1}v_1 + m_{3,2}v_2 + m_{3,3}v_3 + m_{3,4}v_4 \\ m_{4,1}v_1 + m_{4,2}v_2 + m_{4,3}v_3 + m_{4,4}v_4 \end{pmatrix} \tag{11}$$

- Ideal for parallelization and distribution [18]
  1. the *root* thread hands each row of matrix M and the whole vector v to one worker
  2. the worker computes the product of the row and v and
  3. hands back the result to *root*

Matrix-Vector Product

$$\vec{M} \times \vec{v} = \vec{r} \tag{10}$$

$$\begin{pmatrix} m_{1,1} & m_{1,2} & m_{1,3} & m_{1,4} \\ m_{2,1} & m_{2,2} & m_{2,3} & m_{2,4} \\ m_{3,1} & m_{3,2} & m_{3,3} & m_{3,4} \\ m_{4,1} & m_{4,2} & m_{4,3} & m_{4,4} \end{pmatrix} \times \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{pmatrix} = \begin{pmatrix} m_{1,1}v_1 + m_{1,2}v_2 + m_{1,3}v_3 + m_{1,4}v_4 \\ m_{2,1}v_1 + m_{2,2}v_2 + m_{2,3}v_3 + m_{2,4}v_4 \\ m_{3,1}v_1 + m_{3,2}v_2 + m_{3,3}v_3 + m_{3,4}v_4 \\ m_{4,1}v_1 + m_{4,2}v_2 + m_{4,3}v_3 + m_{4,4}v_4 \end{pmatrix} \tag{11}$$

- Ideal for parallelization and distribution [18]
  1. the *root* thread hands each row of matrix M and the whole vector v to one worker
  2. the worker computes the product of the row and v and
  3. hands back the result to *root*
  4. *root* assembles the result vector $\vec{r}$

Regular Problems

- Same algorithm applied to all data

Regular Problems

- Same algorithm applied to all data
- Synchronous communication (or close to): each processor finishes its task at the same time

Regular Problems

- Same algorithm applied to all data
- Synchronous communication (or close to): each processor finishes its task at the same time
- Local (neighbour to neighbour) and collective (combine final results) communication

Regular Problems

- Same algorithm applied to all data
- Synchronous communication (or close to): each processor finishes its task at the same time
- Local (neighbour to neighbour) and collective (combine final results) communication
- Speed-up largely based on the computation to communication ratio $\left(\frac{1-\alpha-\beta}{p}\right.$ in Amdahl's Law [14]$\left.\right)$
  $\Longrightarrow$ if it is large: good speed-up

Regular Problems

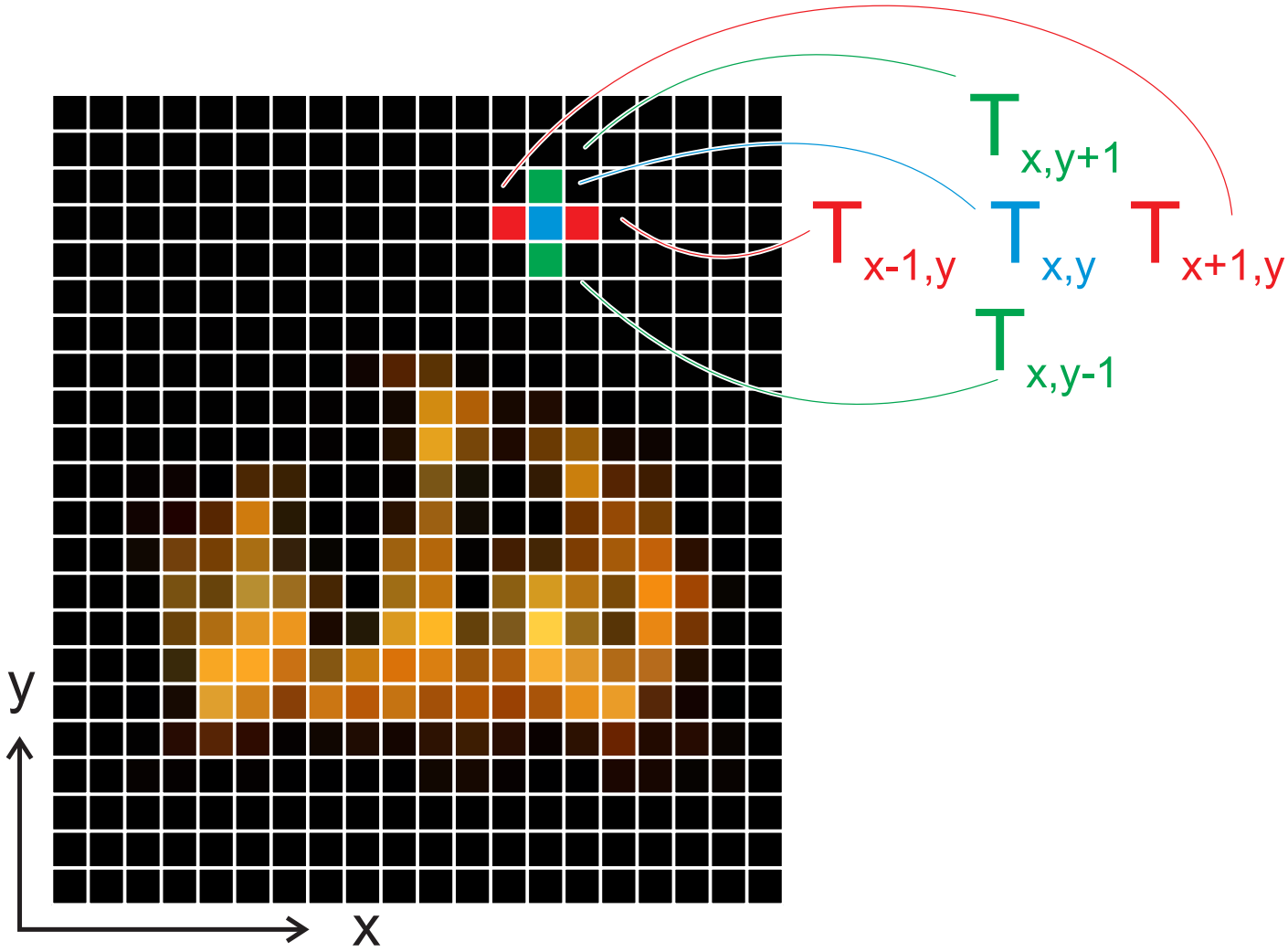
- Same algorithm applied to all data
- Synchronous communication (or close to): each processor finishes its task at the same time
- Local (neighbour to neighbour) and collective (combine final results) communication
- Speed-up largely based on the computation to communication ratio $\left(\frac{1-\alpha-\beta}{p}\right.$ in Amdahl's Law [14]$\left.\right)$
  $\Longrightarrow$ if it is large: good speed-up
- *Examples:* Parallel Evolutionary Algorithms, Finding low-energy molecule states in chemistry, Cellular Automata-based simulations, discrete time simulation of ion movements, multi-player games with large worlds

Heat Dissemination Simulation

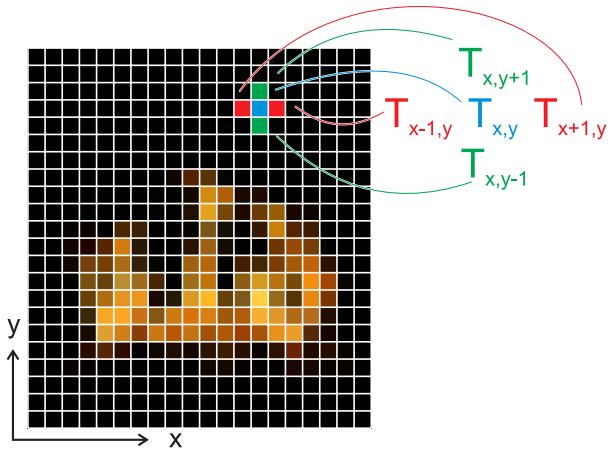
Heat Dissemination Simulation

- Heat equations describe temperature change over time based on a given initial situation and boundary conditions [19]

Heat Dissemination Simulation



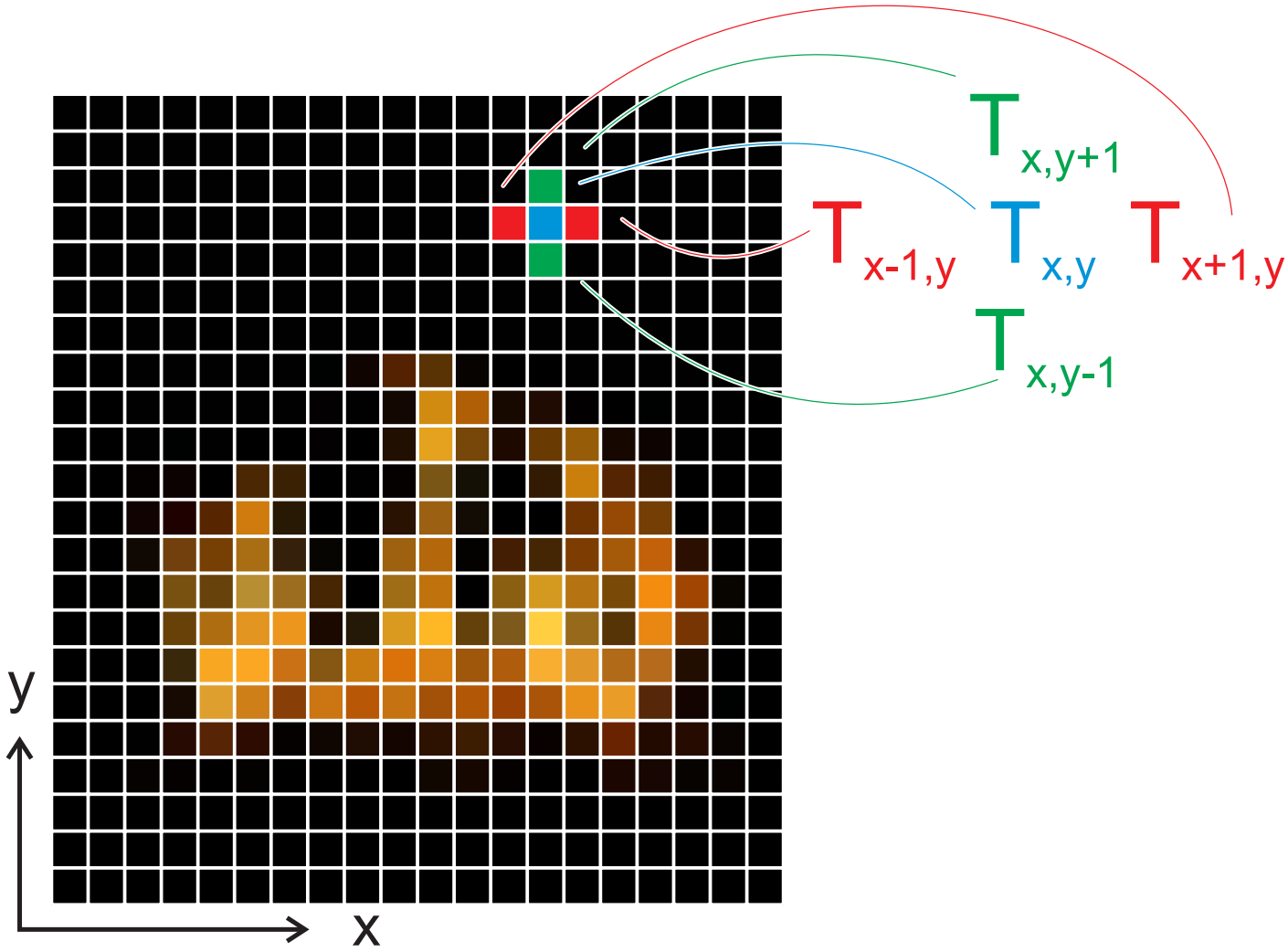
Heat Dissemination Simulation

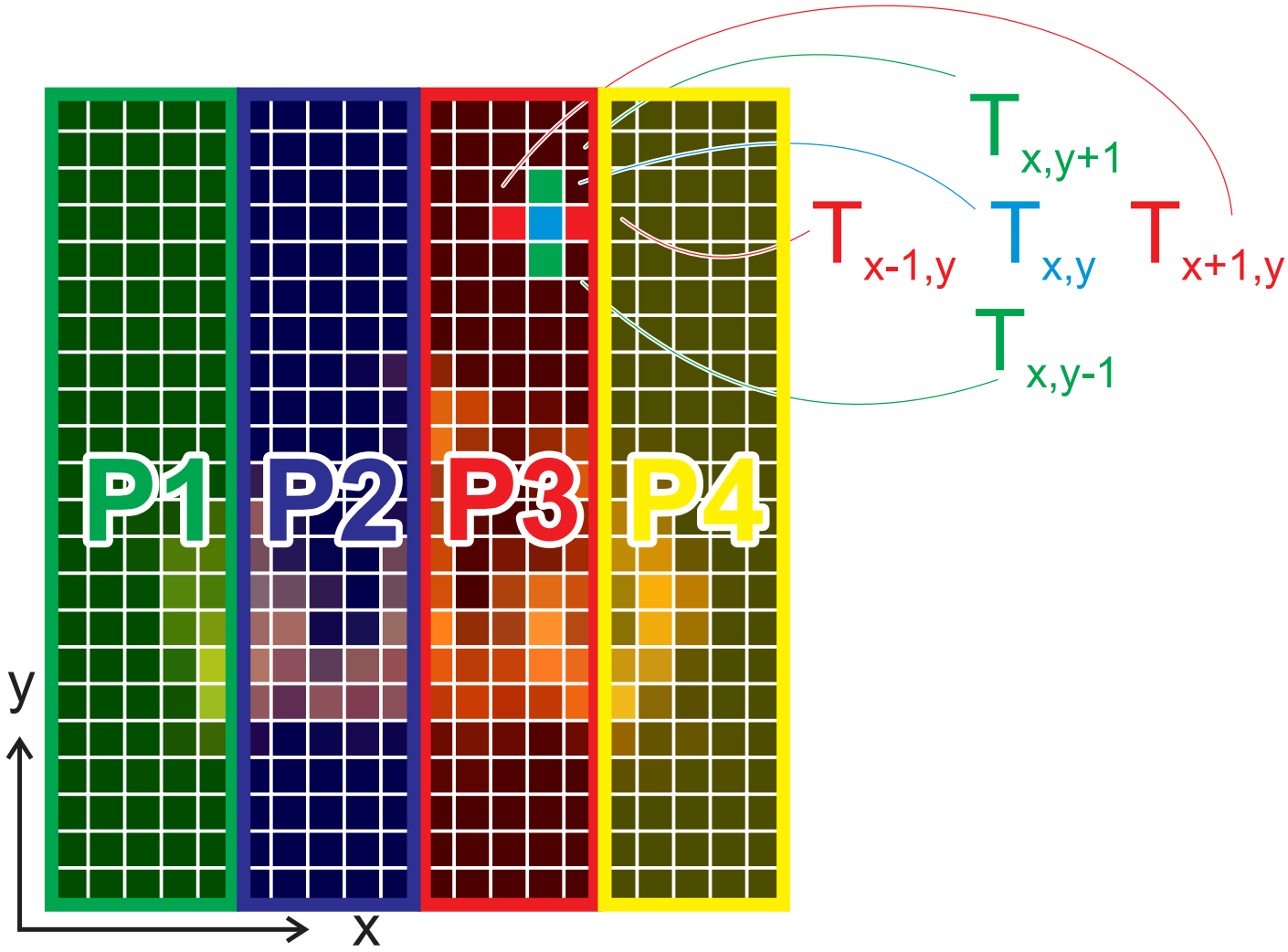


- Heat equations describe temperature change over time based on a given initial situation and boundary conditions [19]
- Finite differencing approximation, numerical, based on a rectangular grid

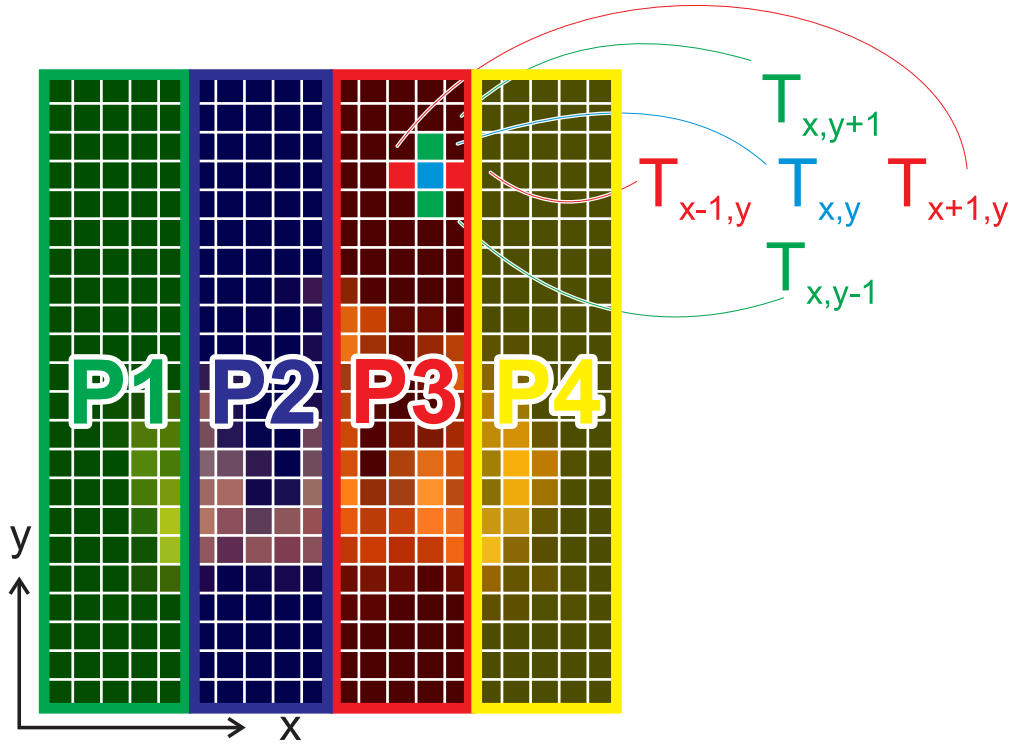
$$T_{x,y}(t+1) = T_{x,y}(t) + c_x\left(T_{x-1,y}(t) + T_{x+1,y}(t) - 2T_{x,y}(t)\right) + c_y\left(T_{x,y-1}(t) + T_{x,y+1}(t) - 2T_{x,y}(t)\right) \quad (12)$$

Distributed Heat Dissemination Simulation



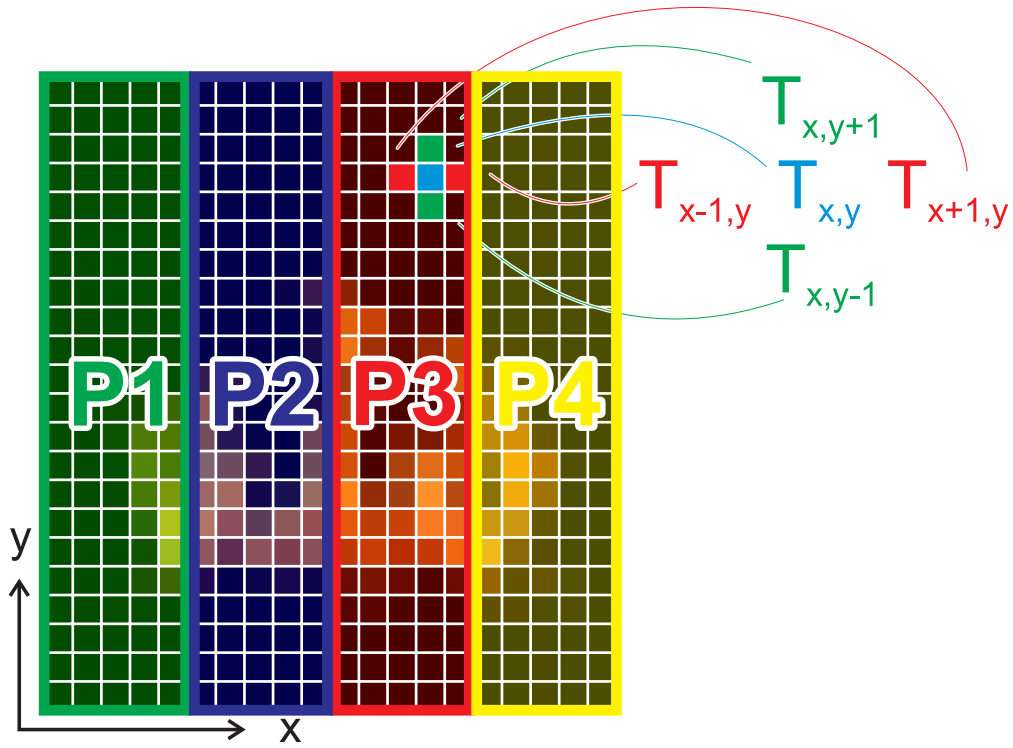

P1
P2
P3
P4
y
x
T x,y+1
T x-1,y
T x,y
T x+1,y
T x,y-1

Distributed Heat Dissemination Simulation



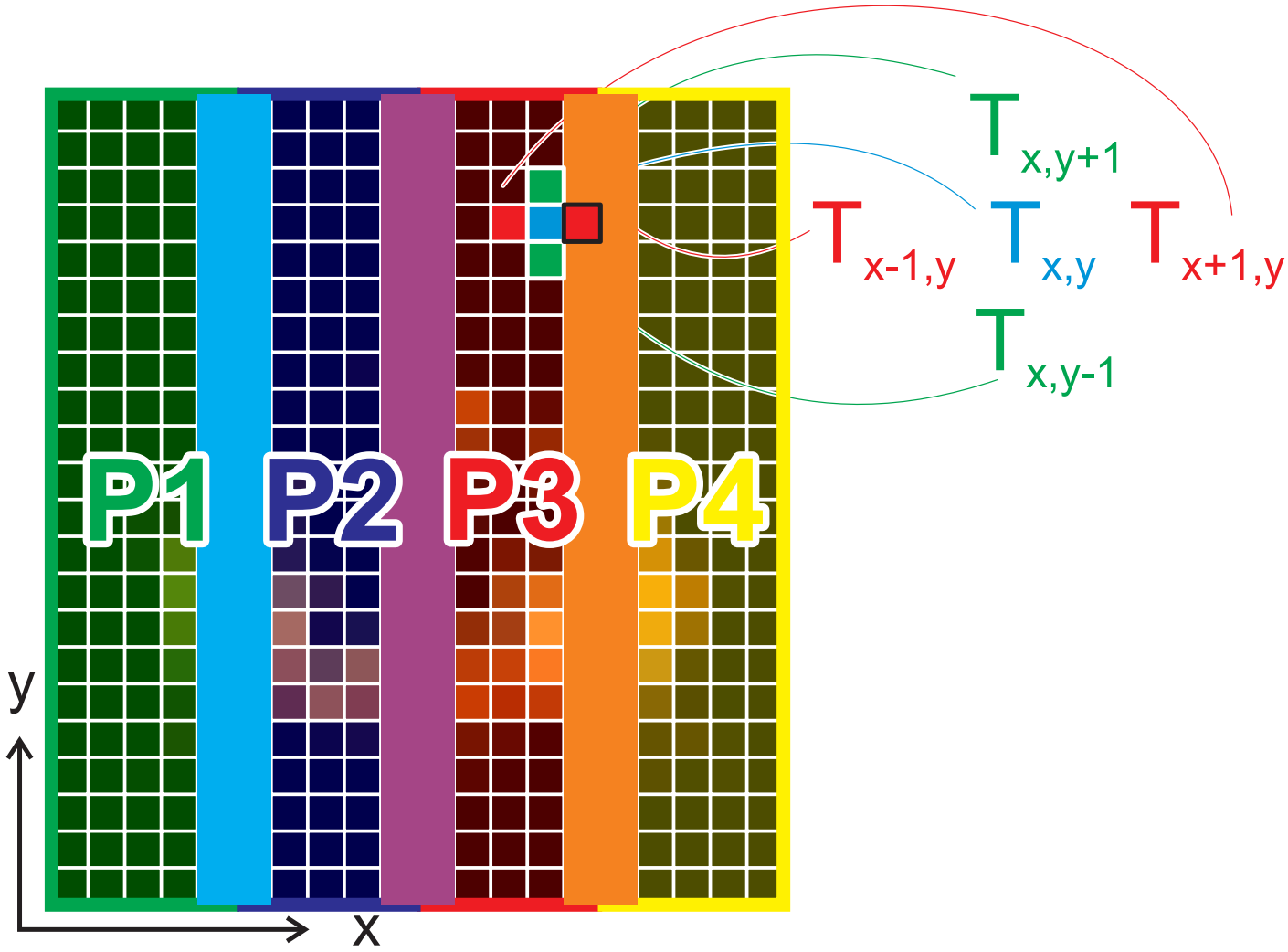
- Divide data into several pieces and simulate in parallel [19]

Distributed Heat Dissemination Simulation



- Divide data into several pieces and simulate in parallel [19]
- But: After *each* time step, exchange data on boundary between "neighboring" threads

Distributed Heat Dissemination Simulation



Irregular Problems

- Irregular algorithm which cannot be parallelized/distributed efficiently except with high communication overhead

Irregular Problems

- Irregular algorithm which cannot be parallelized/distributed efficiently except with high communication overhead
- Communication often asynchronous, complex, may require load balancing

Irregular Problems

- Irregular algorithm which cannot be parallelized/distributed efficiently except with high communication overhead
- Communication often asynchronous, complex, may require load balancing
- Often dynamic repartitioning of data between processors is required

Irregular Problems

- Irregular algorithm which cannot be parallelized/distributed efficiently except with high communication overhead
- Communication often asynchronous, complex, may require load balancing
- Often dynamic repartitioning of data between processors is required
- *Examples:* calculate Fibonacci Numbers [20] by using $F(n) = F(n-1) + F(n-2)$, multi-player games or simulations with strong interaction, such as car racing

What to do?

- If problem suitable for parallelization and computational resources are available, then parallelize!

What to do?

- If problem suitable for parallelization and computational resources are available, then parallelize!
- In many cases, simple parallelization (multiple threads) will be good enough

What to do?

- If problem suitable for parallelization and computational resources are available, then parallelize!
- In many cases, simple parallelization (multiple threads) will be good enough
- If problem is huge and can be parallelized, distribute!

What to do?

- If problem suitable for parallelization and computational resources are available, then parallelize!
- In many cases, simple parallelization (multiple threads) will be good enough
- If problem is huge and can be parallelized, distribute! (but remember Amdahl's Law [14])

What to do?

- If problem suitable for parallelization and computational resources are available, then parallelize!
- In many cases, simple parallelization (multiple threads) will be good enough
- If problem is huge and can be parallelized, distribute! (but remember Amdahl's Law [14])
- How to deal with communication?

What to do?

- If problem suitable for parallelization and computational resources are available, then parallelize!
- In many cases, simple parallelization (multiple threads) will be good enough
- If problem is huge and can be parallelized, distribute! (but remember Amdahl's Law [14])
- How to deal with communication?
  1. Use sockets?

What to do?

- If problem suitable for parallelization and computational resources are available, then parallelize!
- In many cases, simple parallelization (multiple threads) will be good enough
- If problem is huge and can be parallelized, distribute! (but remember Amdahl's Law [14])
- How to deal with communication?
  1. Use sockets? $\longrightarrow$ a bit complex and much work to do (e.g., marshalling data), what about interoperability, maintenance, ...

What to do?

- If problem suitable for parallelization and computational resources are available, then parallelize!
- In many cases, simple parallelization (multiple threads) will be good enough
- If problem is huge and can be parallelized, distribute! (but remember Amdahl's Law [14])
- How to deal with communication?
  1. Use sockets? $\longrightarrow$ a bit complex and much work to do (e.g., marshalling data), what about interoperability, maintenance, ...
  2. Use stuff such as RPC, CORBA, Web Services (i.e., business/corporate-focused frameworks)?

What to do?

- If problem suitable for parallelization and computational resources are available, then parallelize!
- In many cases, simple parallelization (multiple threads) will be good enough
- If problem is huge and can be parallelized, distribute! (but remember Amdahl's Law [14])
- How to deal with communication?
  1. Use sockets? $\longrightarrow$ a bit complex and much work to do (e.g., marshalling data), what about interoperability, maintenance, ...
  2. Use stuff such as RPC, CORBA, Web Services (i.e., business/corporate-focused frameworks)? $\longrightarrow$ not suitable for long computations, massive parallel computations, large data volumes, and high performance requirements

What to do?

- How to deal with stuff such as group communication, synchronization, termination detection?

What to do?

- How to deal with stuff such as group communication, synchronization, termination detection?
  1. Implement your own, specialized algorithms?

What to do?

- How to deal with stuff such as group communication, synchronization, termination detection?
  1. Implement your own, specialized algorithms? $\longrightarrow$ not always suitable, large codebase, maybe make errors

What to do?

- How to deal with stuff such as group communication, synchronization, termination detection?
  1. Implement your own, specialized algorithms? $\longrightarrow$ not always suitable, large codebase, maybe make errors
  2. Use several existing implementations?

What to do?

- How to deal with stuff such as group communication, synchronization, termination detection?
  1. Implement your own, specialized algorithms? $\longrightarrow$ not always suitable, large codebase, maybe make errors
  2. Use several existing implementations? $\longrightarrow$ software will become too hetergeneous, complicated, many libraries, hard to maintain

What to do?

- How to deal with stuff such as group communication, synchronization, termination detection?
  1. Implement your own, specialized algorithms? $\longrightarrow$ not always suitable, large codebase, maybe make errors
  2. Use several existing implementations? $\longrightarrow$ software will become too hetergeneous, complicated, many libraries, hard to maintain
- We want a uniform programming interface and implementations which provide the services we need.

Messare Passing Interface

- Messare Passing Interface (MPI) [21–24] is a standard [25, 26] for the message exchange and synchronization in parallel computations on distributed computing systems

Messare Passing Interface

- Messare Passing Interface (MPI) [21–24] is a standard [25, 26] for the message exchange and synchronization in parallel computations on distributed computing systems
- Developed since 1992, predecessors: PVM [27, 28], PARMACS [28], P4 [28], Chameleon, and Zipcode

Messare Passing Interface

- Messare Passing Interface (MPI) [21–24] is a standard [25, 26] for the message exchange and synchronization in parallel computations on distributed computing systems
- Developed since 1992, predecessors: PVM [27, 28], PARMACS [28], P4 [28], Chameleon, and Zipcode
- It provides a set of operations and their semantics, i.e., a programming interface

Messare Passing Interface

- Messare Passing Interface (MPI) [21–24] is a standard [25, 26] for the message exchange and synchronization in parallel computations on distributed computing systems
- Developed since 1992, predecessors: PVM [27, 28], PARMACS [28], P4 [28], Chameleon, and Zipcode
- It provides a set of operations and their semantics, i.e., a programming interface
- It does *not* define a specific protocol or implementation

Messare Passing Interface

- Messare Passing Interface (MPI) [21–24] is a standard [25, 26] for the message exchange and synchronization in parallel computations on distributed computing systems
- Developed since 1992, predecessors: PVM [27, 28], PARMACS [28], P4 [28], Chameleon, and Zipcode
- It provides a set of operations and their semantics, i.e., a programming interface
- It does *not* define a specific protocol or implementation
- All definitions are hardware-independent

Typical Structure

- A typical MPI application

Typical Structure

- A typical MPI application:
  - a set of communicating processes

Typical Structure

- A typical MPI application:
  - a set of communicating processes
  - started in parallel possibly on

Typical Structure

- A typical MPI application:
  - a set of communicating processes
  - started in parallel possibly on
    1. multiple different computers e.g., in a cluster

Typical Structure

- A typical MPI application:
  - a set of communicating processes
  - started in parallel possibly on
    1. multiple different computers e.g., in a cluster or
    2. dedicated parallel computers

Typical Structure

- A typical MPI application:
  - a set of communicating processes
  - started in parallel possibly on
    1. multiple different computers e.g., in a cluster or
    2. dedicated parallel computers
  - processes work together on one problem

Typical Structure

- A typical MPI application:
  - a set of communicating processes
  - started in parallel possibly on
    1. multiple different computers e.g., in a cluster or
    2. dedicated parallel computers
  - processes work together on one problem
  - processes use messages for information exchange

Typical Structure

- A typical MPI application:
  - a set of communicating processes
  - started in parallel possibly on
    1. multiple different computers e.g., in a cluster or
    2. dedicated parallel computers
  - processes work together on one problem
  - processes use messages for information exchange
  - Basic paradigm: message-based (no streams)

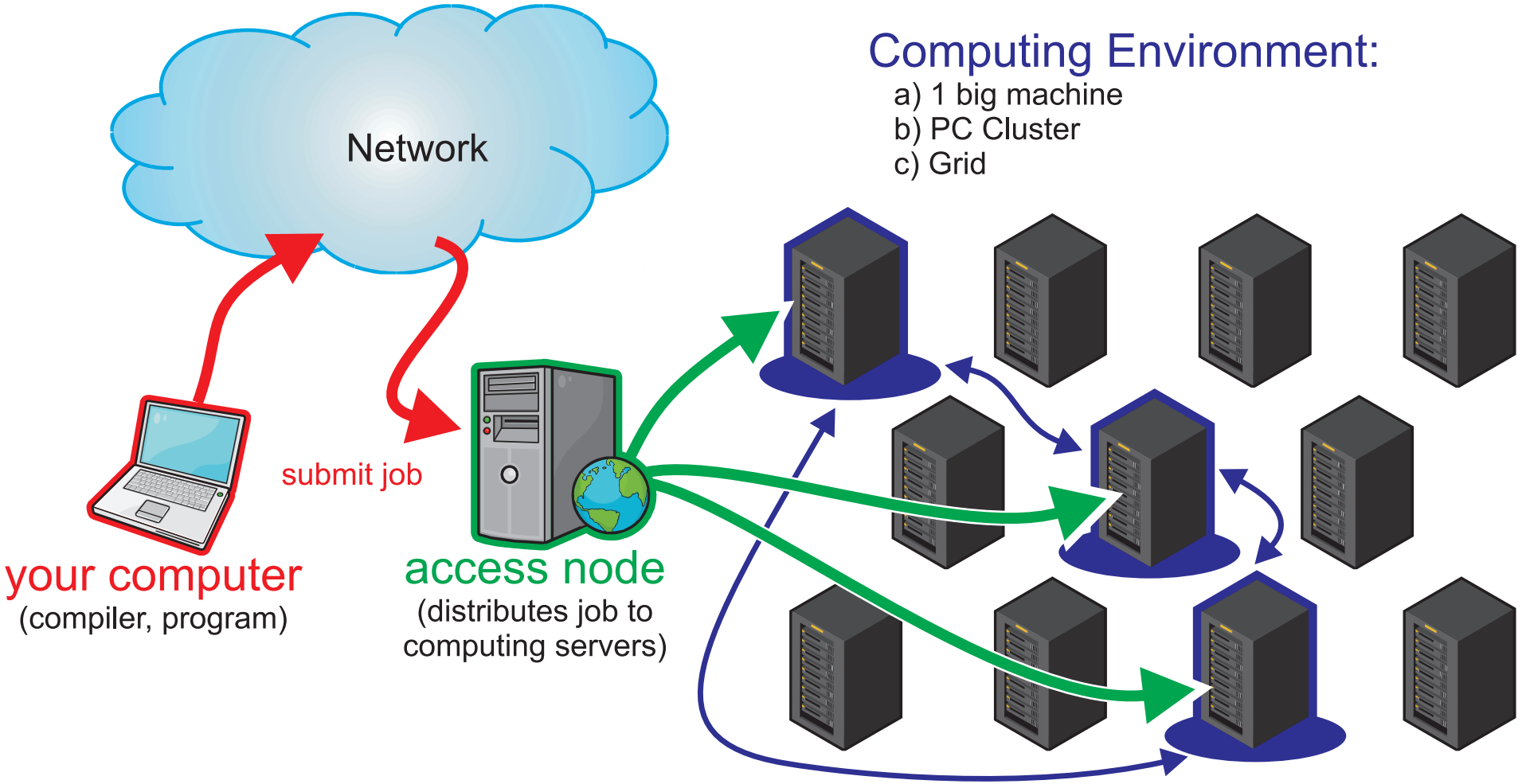
Typical Structure

- A typical MPI application:
  - a set of communicating processes
  - started in parallel possibly on
    1. multiple different computers e.g., in a cluster or
    2. dedicated parallel computers
  - processes work together on one problem
  - processes use messages for information exchange
  - Basic paradigms: message-based (no streams), *group communication*

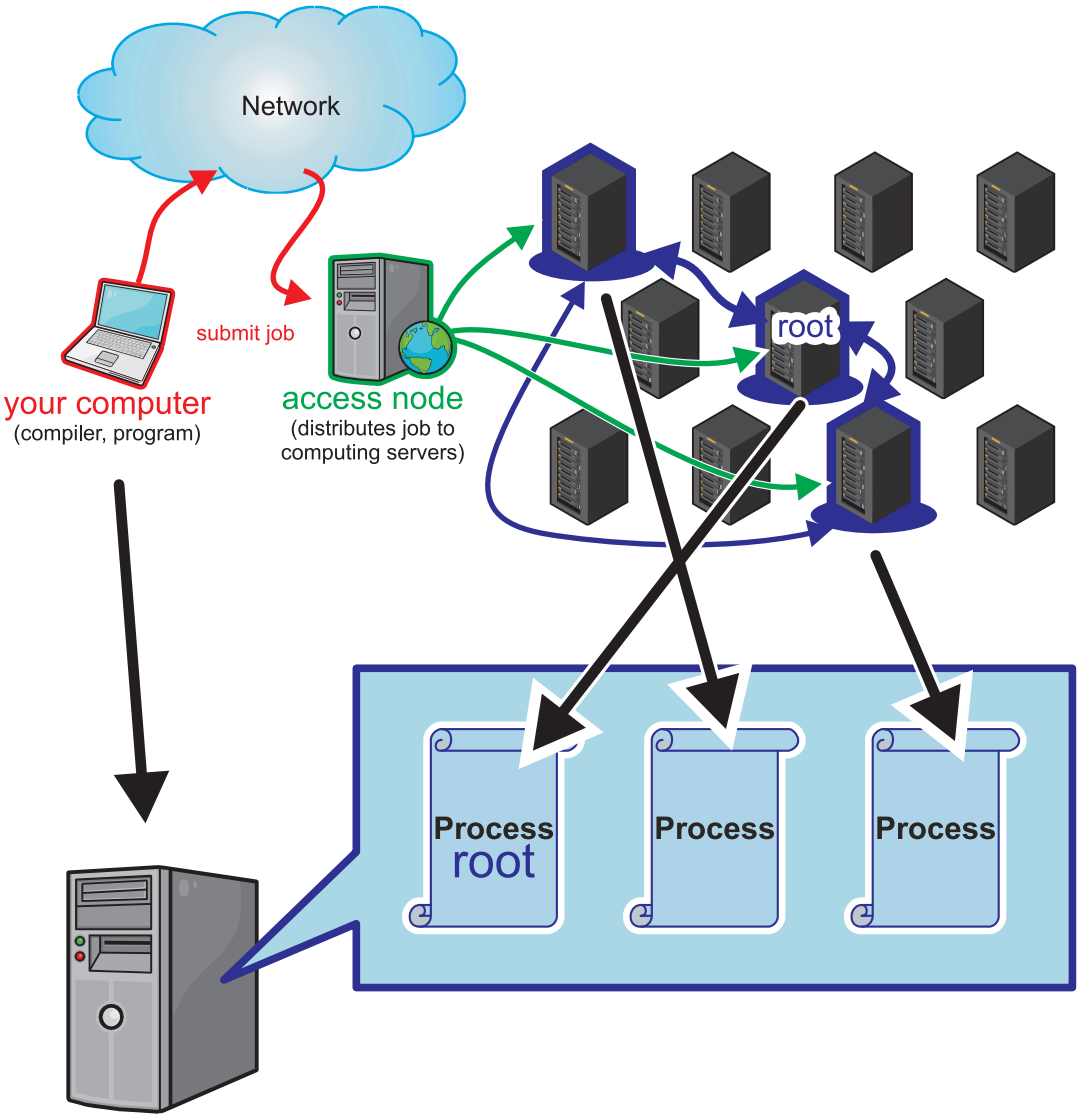
Typical Structure

- A typical MPI application:
  - a set of communicating processes
  - started in parallel possibly on
    1. multiple different computers e.g., in a cluster or
    2. dedicated parallel computers
  - processes work together on one problem
  - processes use messages for information exchange
  - Basic paradigms: message-based (no streams), *group communication*, *reliable* [26]

Typical Structure



. . . but in our experiments. . .



Standard

- The current version of the MPI standard is 2.2 [25]

Standard

- The current version of the MPI standard is 2.2 [25]
- MPI 1

Standard

- The current version of the MPI standard is 2.2 [25]
- MPI 1:
  - Point-to-Point Communication (Unicast)

Standard

- The current version of the MPI standard is 2.2 [25]
- MPI 1:
  - Point-to-Point Communication (Unicast)
  - Global communication (Broadcast)

Standard

- The current version of the MPI standard is 2.2 [25]
- MPI 1:
  - Point-to-Point Communication (Unicast)
  - Global communication (Broadcast)
  - Groups, Contexts, and Communicators

Standard

- The current version of the MPI standard is 2.2 [25]
- MPI 1:
  - Point-to-Point Communication (Unicast)
  - Global communication (Broadcast)
  - Groups, Contexts, and Communicators
  - Environment

Standard

- The current version of the MPI standard is 2.2 [25]
- MPI 1:
  - Point-to-Point Communication (Unicast)
  - Global communication (Broadcast)
  - Groups, Contexts, and Communicators
  - Environment
  - Profiling Interface

Standard

- The current version of the MPI standard is 2.2 [25]
- MPI 1:
  - Point-to-Point Communication (Unicast)
  - Global communication (Broadcast)
  - Groups, Contexts, and Communicators
  - Environment
  - Profiling Interface
  - Language binding for C and Fortran

Standard

- The current version of the MPI standard is 2.2 [25]
- MPI 1:
  - Point-to-Point Communication (Unicast)
  - Global communication (Broadcast)
  - Groups, Contexts, and Communicators
  - Environment
  - Profiling Interface
  - Language binding for C and Fortran
- MPI 2

Standard

- The current version of the MPI standard is 2.2 [25]
- MPI 1:
  - Point-to-Point Communication (Unicast)
  - Global communication (Broadcast)
  - Groups, Contexts, and Communicators
  - Environment
  - Profiling Interface
  - Language binding for C and Fortran
- MPI 2:
  - Parallel File IO

Standard

- The current version of the MPI standard is 2.2 [25]
- MPI 1:
  - Point-to-Point Communication (Unicast)
  - Global communication (Broadcast)
  - Groups, Contexts, and Communicators
  - Environment
  - Profiling Interface
  - Language binding for C and Fortran
- MPI 2:
  - Parallel File IO
  - Dynamic Process Management

Standard

- The current version of the MPI standard is 2.2 [25]
- MPI 1:
  - Point-to-Point Communication (Unicast)
  - Global communication (Broadcast)
  - Groups, Contexts, and Communicators
  - Environment
  - Profiling Interface
  - Language binding for C and Fortran
- MPI 2:
  - Parallel File IO
  - Dynamic Process Management
  - Access to memory of other processes

Standard

- The current version of the MPI standard is 2.2 [25]
- MPI 1:
  - Point-to-Point Communication (Unicast)
  - Global communication (Broadcast)
  - Groups, Contexts, and Communicators
  - Environment
  - Profiling Interface
  - Language binding for C and Fortran
- MPI 2:
  - Parallel File IO
  - Dynamic Process Management
  - Access to memory of other processes
  - Language Binding for C++ and Fortran

Standard

- The current version of the MPI standard is 2.2 [25]
- MPI 1:
  - Point-to-Point Communication (Unicast)
  - Global communication (Broadcast)
  - Groups, Contexts, and Communicators
  - Environment
  - Profiling Interface
  - Language binding for C and Fortran
- MPI 2:
  - Parallel File IO
  - Dynamic Process Management
  - Access to memory of other processes
  - Language Binding for C++ and Fortran
- More than 200 functions

Standard

- The current version of the MPI standard is 2.2 [25]
- MPI 1:
  - Point-to-Point Communication (Unicast)
  - Global communication (Broadcast)
  - Groups, Contexts, and Communicators
  - Environment
  - Profiling Interface
  - Language binding for C and Fortran
- MPI 2:
  - Parallel File IO
  - Dynamic Process Management
  - Access to memory of other processes
  - Language Binding for C++ and Fortran
- More than 200 functions, but we need only a few of them

Implementations

- C/C++/Fortran

Implementations

- C/C++/Fortran:
  1. MPICH [29, 30]

Implementations

- C/C++/Fortran:
  1. MPICH [29, 30]
  2. LAM/MPI [31]. not continued

Implementations

- C/C++/Fortran:
  1. MPICH [29, 30]
  2. LAM/MPI [31]. not continued, development now focussed on:
  3. Open MPI [32]

Implementations

- C/C++/Fortran:
  1. MPICH [29, 30]
  2. LAM/MPI [31]. not continued, development now focussed on:
  3. Open MPI [32]
  4. DeinoMPI [33]

Implementations

- C/C++/Fortran:
  1. MPICH [29, 30]
  2. LAM/MPI [31]. not continued, development now focussed on:
  3. Open MPI [32]
  4. DeinoMPI [33]
- C#: MPI.NET [34]

Implementations

- C/C++/Fortran:
  1. MPICH [29, 30]
  2. LAM/MPI [31]. not continued, development now focussed on:
  3. Open MPI [32]
  4. DeinoMPI [33]
- C#: MPI.NET [34]
- Python: pyMPI [35]

- As said, MPI defines many different functions

- As said, MPI defines many different functions
- Here we can only discuss a few

- As said, MPI defines many different functions
- Here we can only discuss a few
- We will focus on key aspects and on some simple examples

- As said, MPI defines many different functions
- Here we can only discuss a few
- We will focus on key aspects and on some simple examples
- For more information and examples, you can look into...

- As said, MPI defines many different functions
- Here we can only discuss a few
- We will focus on key aspects and on some simple examples
- For more information and examples, you can look into
  1. any of [21–26, 29, 30]
  2. Tutorials:
     - `http://www.lam-mpi.org/tutorials/`
     - `http://www.mcs.anl.gov/research/projects/mpi/tutorial/`
     - `http://www.mpitutorial.com/`

- As said, MPI defines many different functions
- Here we can only discuss a few
- We will focus on key aspects and on some simple examples
- For more information and examples, you can look into
  1. any of [21–26, 29, 30]
  2. Tutorials:
     - `http://www.lam-mpi.org/tutorials/`
     - `http://www.mcs.anl.gov/research/projects/mpi/tutorial/`
     - `http://www.mpitutorial.com/`
  3. Documentation:
     - `http://www.mpi-forum.org/docs/`
     - `http://www.mcs.anl.gov/research/projects/mpich2/documentation/`

Simple Program

Listing: Care bones of MPI program (bareBones.c).

```
#include <mpi.h> // import MPI header

int main(int argc, char **argv) {
  MPI_Init(&argc, &argv); // initialize MPI
  MPI_Finalize();         // shut down MPI
  return 0;
}
```

Simple Program

Listing: Care bones of MPI program (bareBones.c).

```
#include <mpi.h> // import MPI header

int main(int argc, char **argv) {
  MPI_Init(&argc, &argv); // initialize MPI
  MPI_Finalize();         // shut down MPI
  return 0;
}
```

- `MPI_Init` starts the MPI subsystem

Simple Program

Listing: Care bones of MPI program (bareBones.c).

```
#include <mpi.h> // import MPI header

int main(int argc, char **argv) {
  MPI_Init(&argc, &argv); // initialize MPI
  MPI_Finalize();         // shut down MPI
  return 0;
}
```

- `MPI_Init` starts the MPI subsystem
- `MPI_Finalize` shuts down the MPI subsystem

Simple Program

Listing: Care bones of MPI program (bareBones.c).

```
#include <mpi.h> // import MPI header

int main(int argc, char **argv) {
  MPI_Init(&argc, &argv); // initialize MPI
  MPI_Finalize();         // shut down MPI
  return 0;
}
```

- `MPI_Init` starts the MPI subsystem
- `MPI_Finalize` shuts down the MPI subsystem
- Similar to `WSAStartup` and `WSACleanup` in the sockets lecture

MPI_Init and MPI_Finalize

- `int MPI_Init(int *argc, char ***argv)` executes all actions which are necessary for communication later

MPI_Init and MPI_Finalize

- `int MPI_Init(int *argc, char ***argv)` executes all actions which are necessary for communication later, such as
  1. establishing connections

MPI_Init and MPI_Finalize

- `int MPI_Init(int *argc, char ***argv)` executes all actions which are necessary for communication later, such as
  1. establishing connections
  2. initialization of variables

MPI_Init and MPI_Finalize

- `int MPI_Init(int *argc, char ***argv)` executes all actions which are necessary for communication later, such as
  1. establishing connections
  2. initialization of variables
  3. explore the network

MPI_Init and MPI_Finalize

- `int MPI_Init(int *argc, char ***argv)` executes all actions which are necessary for communication later, such as
  1. establishing connections
  2. initialization of variables
  3. explore the network
  4. maybe initializing WinSock etc.
  5. ...

MPI_Init and MPI_Finalize

- `int MPI_Init(int *argc, char ***argv)` executes all actions which are necessary for communication later, such as
  1. establishing connections
  2. initialization of variables
  3. explore the network
  4. maybe initializing WinSock etc.
  5. ...
- `MPI_Finalize()` is the last MPI call in a program

MPI_Init and MPI_Finalize

- `int MPI_Init(int *argc, char ***argv)` executes all actions which are necessary for communication later, such as
  1. establishing connections
  2. initialization of variables
  3. explore the network
  4. maybe initializing WinSock etc.
  5. ...
- `MPI_Finalize()` is the last MPI call in a program
  - all communication must be finished before that

MPI_Init and MPI_Finalize

- `int MPI_Init(int *argc, char ***argv)` executes all actions which are necessary for communication later, such as
  1. establishing connections
  2. initialization of variables
  3. explore the network
  4. maybe initializing WinSock etc.
  5. ...
- `MPI_Finalize()` is the last MPI call in a program
  - all communication must be finished before that
- All MPI routines return an `int` with the result status, `MPI_SUCCESS` means everything went OK

Important Runtime Parameters

- MPI programs need information

Important Runtime Parameters

- MPI programs need information about
  1. "themselves"

Important Runtime Parameters

- MPI programs need information about
  1. "themselves" and
  2. the current system of processes

Important Runtime Parameters

- MPI programs need information about
  1. "themselves" and
  2. the current system of processes

Important Runtime Parameters

- MPI programs need information about
  1. "themselves" and
  2. the current system of processes
- How many processes are there?

Important Runtime Parameters

- MPI programs need information about
  1. "themselves" and
  2. the current system of processes
- How many processes are there?
  - `MPI_Comm_size(MPI_Comm *comm, int *size)`

Important Runtime Parameters

- MPI programs need information about
  1. "themselves" and
  2. the current system of processes
- How many processes are there?
  - `MPI_Comm_size(MPI_Comm *comm, int *size)`
- Which ID do I have?

Important Runtime Parameters

- MPI programs need information about
  1. "themselves" and
  2. the current system of processes
- How many processes are there?
  - `MPI_Comm_size(MPI_Comm *comm, int *size)`
- Which ID do I have?
  - `MPI_Comm_rank(MPI_Comm *comm, int *rank)`

Important Runtime Parameters

- MPI programs need information about
  1. "themselves" and
  2. the current system of processes
- How many processes are there?
  - `MPI_Comm_size(MPI_Comm *comm, int *size)`
- Which ID do I have?
  - `MPI_Comm_rank(MPI_Comm *comm, int *rank)`
  - `rank` $\in \{0 \ldots \texttt{size} - 1\}$

Communicators

- Basis for group communication

Communicators

- Basis for group communication:
  - communicators are special MPI constructs

Communicators

- Basis for group communication:
  - communicators are special MPI constructs
  - hold a subset of processes

Communicators

- Basis for group communication:
  - communicators are special MPI constructs that
  - hold a subset of processes
  - is passed as parameter for communication

Communicators

- Basis for group communication:
  - communicators are special MPI constructs that
  - hold a subset of processes and
  - is passed as parameter for communication
  - communicators can be created by MPI processes

Communicators

- Basis for group communication:
  - communicators are special MPI constructs that
  - hold a subset of processes and
  - is passed as parameter for communication
  - communicators can be created by MPI processes
- For now, we just use `MPI_COMM_WORLD`

Communicators

- Basis for group communication:
  - communicators are special MPI constructs that
  - hold a subset of processes and
  - is passed as parameter for communication
  - communicators can be created by MPI processes
- For now, we just use `MPI_COMM_WORLD`
- which contains all MPI processes, i.e., does broadcast

Listing: Extended MPI Program (basicInfo.c)

```c
#include <mpi.h>    // import MPI header
#include <stdio.h> // import for printf

int main(int argc, char **argv) {
  int size, rank;

  MPI_Init(&argc, &argv); // initialize MPI

  MPI_Comm_size(MPI_COMM_WORLD, &size); // get number of program instances
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); // get own ID/address

  // often, an MPI application has a master and some slaves
  // master distributes tasks and combine partial results to final results
  // slaves receive partial task, compute partial result, and send to master
  if(rank == 0) { // the instance with rank=0 is often chosen as master
    printf("Hi␣from␣Master\n");
  } else {        // the others are often slaves
    printf("Just␣Slave␣%d␣out␣of␣%d\n", rank, size);
  }

  MPI_Finalize(); // finalize = shut down MPI
  return 0;
}
```

Blocking Send: MPI_Send

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

- Performs a blocking send

Blocking Send: MPI_Send

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

- Performs a blocking send:
  - Will block until message has been copied to OS/network stack buffers

Blocking Send: MPI_Send

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

- Performs a blocking send:
  - Will block until message has been copied to OS/network stack buffers
  - *May* block until message has received at destination process

Blocking Send: MPI_Send

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

- Performs a blocking send:
  - Will block until message has been copied to OS/network stack buffers
  - *May* block until message has received at destination process
  - Buffer can be overwritten after function returns

Blocking Send: MPI_Send

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

- Performs a blocking send:
  - Will block until message has been copied to OS/network stack buffers
  - *May* block until message has received at destination process
  - Buffer can be overwritten after function returns
- Input Parameters
  - `buf` ... initial address of send buffer
  - `count` ... number of elements in send buffer
  - `datatype` ... datatype of each send buffer element
  - `dest` ... rank/id of destination process
  - `tag` ... message tag: which send belongs to which receive
  - `comm` dotsthe communicator to use

Blocking Receive: MPI_Recv

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

- Performs a blocking receive: Waits until a message has been received

Blocking Receive: MPI_Recv

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

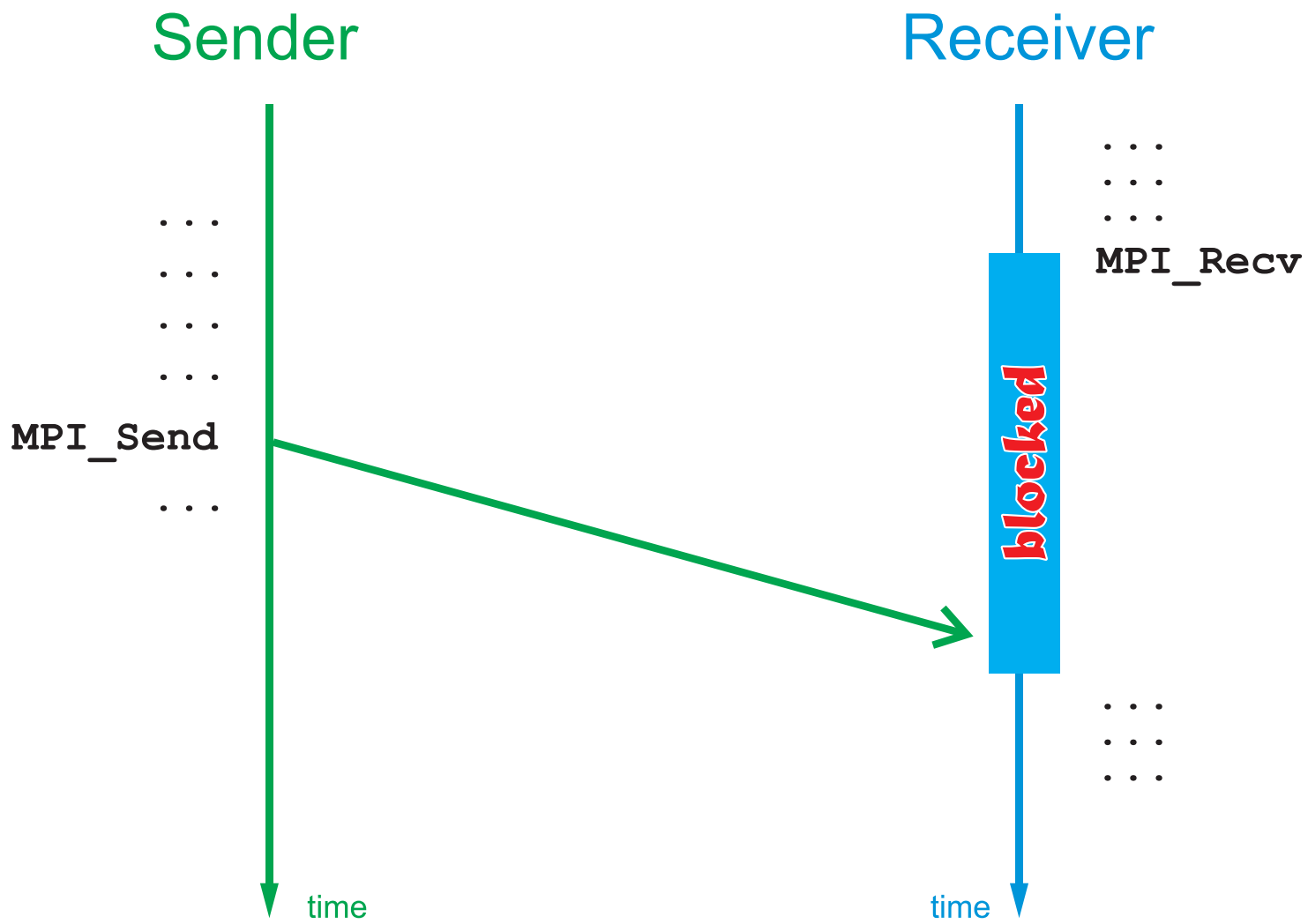
- Performs a blocking receive: Waits until a message has been received
- Input Parameters
  - `count` ... maximum number of elements in receive buffer
  - `datatype` ... datatype of each receive buffer element
  - `source` ... rank/id of source
  - `tag` ... message tag – must match to tag specified when sending
  - `comm` ... the communicator to use

Blocking Receive: MPI_Recv

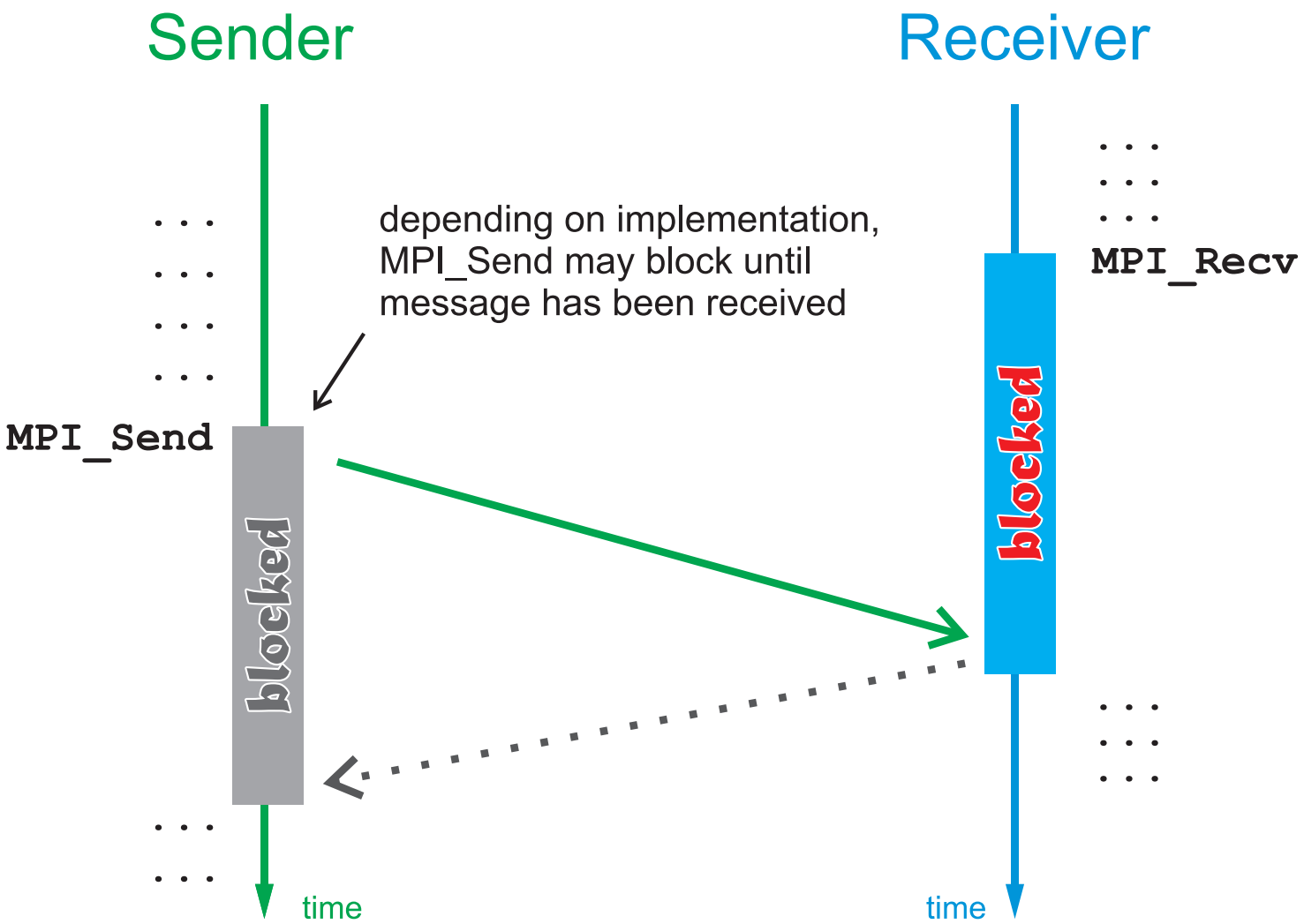
```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm,
I_Status *status)
```

- Performs a blocking receive: Waits until a message has been received
- Input Parameters
  - `count` ... maximum number of elements in receive buffer
  - `datatype` ... datatype of each receive buffer element
  - `source` ... rank/id of source
  - `tag` ... message tag – must match to tag specified when sending
  - `comm` ... the communicator to use
- Output Parameters
  - `buf` ... initial address of receive buffer
  - `status` ... status object

MPI_Send and MPI_Recv



MPI_Send and MPI_Recv



Synchronized Send: MPI_Ssend

```
int MPI_Ssend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

- Performs a blocking and synchronized send

Synchronized Send: MPI_Ssend

```
int MPI_Ssend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

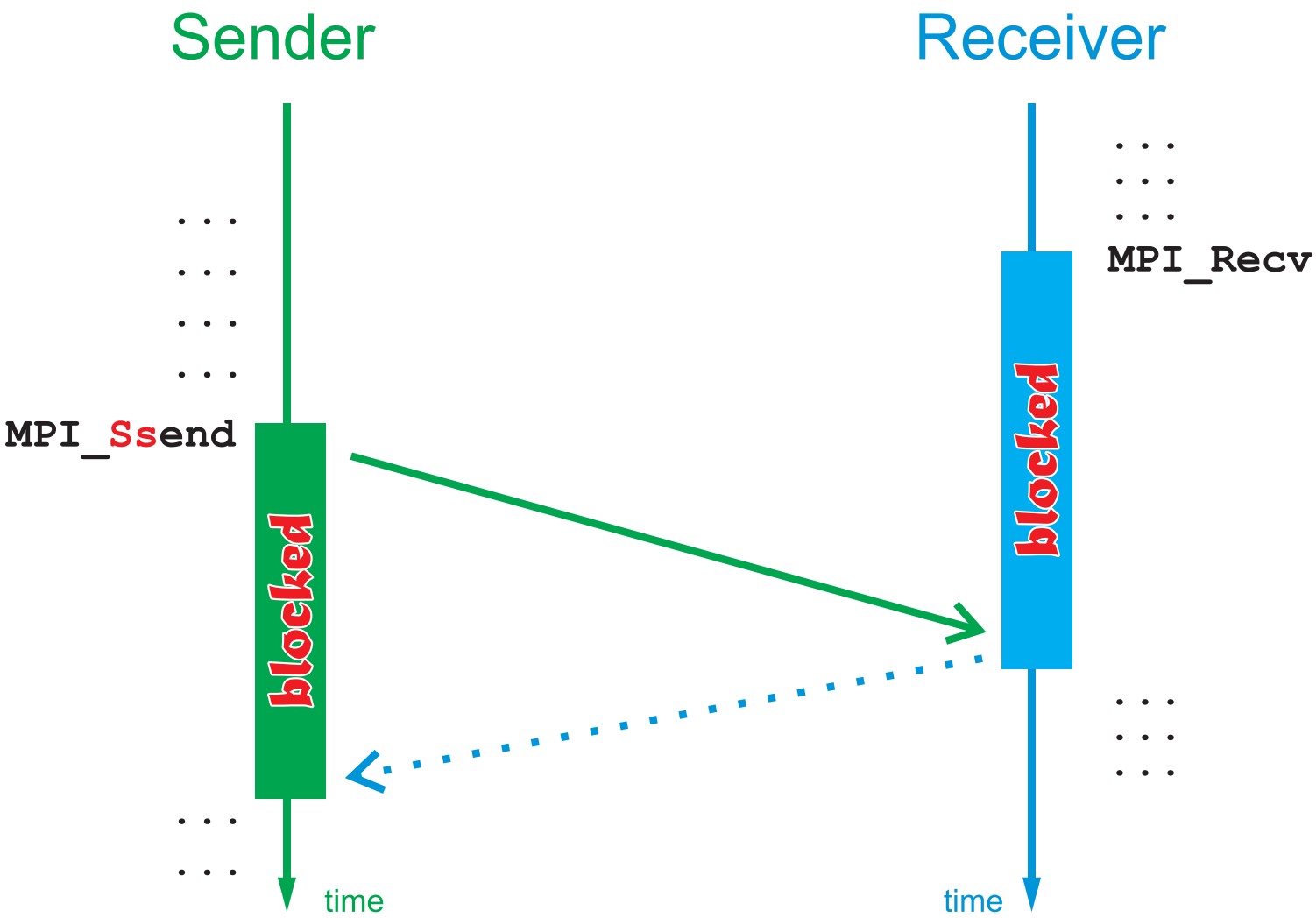
- Performs a blocking and synchronized send:
  - Will block until message has been copied to OS/network stack buffers

Synchronized Send: MPI_Ssend

```
int MPI_Ssend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

- Performs a blocking and synchronized send:
  - Will block until message has been copied to OS/network stack buffers
  - Will block until message has received at destination process

MPI_Ssend and MPI_Recv



Listing: A Point-to-Point communication example (simplePointToPoint1.c)

```c
#include <mpi.h>   // import MPI header
#include <stdio.h> // needed for printf

int main(int argc, char **argv) {
  int size, rank, s_msg, r_msg, next, prev;
  MPI_Status status;

  MPI_Init(&argc, &argv); // initialize MPI
  MPI_Comm_size(MPI_COMM_WORLD, &size); // get number of program instances
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); // get own ID/address

  next  = ((rank + 1) % size);  // next higher id, wrap from size-1 to 0
  prev  = ((rank + size - 1) % size); // next lower id, wrap from 0 to size-1
  s_msg = ((size * rank) + next); // the example message, just some number

  if((rank % 2) == 0) { // even rank: message to next, receive from prev
    MPI_Send(&s_msg, 1, MPI_INT, next, 42, MPI_COMM_WORLD);
    MPI_Recv(&r_msg, 1, MPI_INT, prev, 42, MPI_COMM_WORLD, &status);
  } else { // otherwise: receive from rev, send to next
    MPI_Recv(&r_msg, 1, MPI_INT, prev, 42, MPI_COMM_WORLD, &status);
    MPI_Send(&s_msg, 1, MPI_INT, next, 42, MPI_COMM_WORLD);
  }

  printf("id: %d, next: %d, prev: %d, send: %d, recv: %d\n", rank, next, prev, s_msg,
      r_msg);

  MPI_Finalize(); // shut down MPI
  return 0;
}
```

Listing: A Point-to-Point communication example (simplePointToPoint2.c)

```c
#include <mpi.h>     // import MPI header
#include <stdio.h>   // needed for printf
#include <string.h> // needed for strlen

int main(int argc, char *argv[]) {
  char message[20];  // char array big enough to hold message
  int rank;          // own rank
  MPI_Status status; // status variable

  MPI_Init(&argc, &argv); // initialize mpi
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); // get own rank

  if (rank == 0) { // if we have rank 0...
    strcpy(message, "Hello, there"); /// ...create and send message to rank 1
    MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, 42, MPI_COMM_WORLD);
    printf("sent: \"%s\"\n", message); // print the message that was sent
  } else { // if we are rank 1, receive message coming from rank 0
    MPI_Recv(message, 20, MPI_CHAR, 0, 42, MPI_COMM_WORLD, &status);
    printf("received: \"%s\"\n", message); // print message
  }

  MPI_Finalize(); // shut down MPI
  return 0;
}
```

Listing: Computing Pi with Point-to-Point communication example (piPointToPoint.c)

```c
#include <mpi.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

int main(int argc, char **argv) {
  int           i, size, rank;            double     x, y;
  long long int root[2], worker[2];       MPI_Status status;

  MPI_Init(&argc, &argv);                                 //initialize MPI
  MPI_Comm_size(MPI_COMM_WORLD, &size);                   //get the number of processes in the global communicator
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);                   //get the rank of this process within the global communicator
  root[0] = root[1] = worker[0] = worker[1] = 0LL;        //clear the data buffer

  if(rank == 0) {                                         //check if we are root
    for(i = size; (--i) > 0; ) {                          //receive data from the workers
      MPI_Recv(&worker[0], 2, MPI_LONG_LONG_INT, i, 42, MPI_COMM_WORLD, &status); //do receive
      root[0] += worker[0];                               //get the received sample size (number of points)
      root[1] += worker[1];                               //get the number of samples (points) inside the unit circle
      printf("worker␣%d␣sends␣estimate␣%G␣(based␣on␣%lld␣samples),␣total␣estimate␣now␣is␣%G␣(based␣on␣%lld␣samples).\n", i,
                  ((4.0 * worker[1]) / worker[0]), worker[0], ((4.0 * root[1]) / root[0]), root[0]);
      fflush(stdout);                                     //flush the standard out
    }
  } else {                                                //ok, we are a worker
    srand(time(NULL));
    for(worker[0] = 1; worker[0] < (rank * 100000000LL); worker[0]++) { //make 100 000 000 samples
      x = (rand() / ((double)RAND_MAX));                  //random x-coordinate in [0,1]
      y = (rand() / ((double)RAND_MAX));                  //random y-coordinate in [0,1]
      if( ((x*x) + (y*y)) <= 1.0 ) {                      //did it fall into the inner circle?
        worker[1]++;                                      //yes, it did - increase counter
      }
    }
    MPI_Send(&worker[0], 2, MPI_LONG_LONG_INT, 0, 42, MPI_COMM_WORLD); //send worker result synchronously
  }

  MPI_Finalize();                                         //finish the MPI stuff
  return 0;
}
```

Listing: Point-to-Point with error (deadlock.c)

```c
#include <mpi.h>    // import MPI header
#include <stdio.h>  // needed for printf
#include <string.h> // needed for strlen

int main(int argc, char **argv) {
  int         rank, size, prev, next;
  MPI_Status  status;
  char        messageIn[20], messageOut[20];

  MPI_Init(&argc, &argv); // initialize MPI
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); // get own rank/ID
  MPI_Comm_size(MPI_COMM_WORLD, &size); // get total number of processes

  prev = ((size + rank - 1) % size); // get rank of process to receive from, wrap at 0
  MPI_Recv(messageIn, 20, MPI_CHAR, prev, 0, MPI_COMM_WORLD, &status); // receive msg
  printf("Process␣%d␣received␣message␣%s␣from␣process␣%d.\n", rank, messageIn, prev);

  next = ((rank + 1) % size); // get rank of process to send message to
  strcpy(messageOut, "Important␣message!"); // construct message
  printf("Process␣%d␣is␣sending␣message␣%s␣to␣process␣%d.\n", rank, messageOut, next);
  MPI_Send(messageOut, 20, MPI_CHAR, next, 0, MPI_COMM_WORLD); // send message

  MPI_Finalize(); // shut down MPI
  return 0;
}
```

Non-blocking Operations

- Sometimes, we want to keep calculating while sending/receiving is going on: non-blocking operations

Non-blocking Operations

- Sometimes, we want to keep calculating while sending/receiving is going on: non-blocking operations

```
int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm
m, MPI_Request* request)
```

```
int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm
m, MPI_Request* request)
```

Non-blocking Operations

- Sometimes, we want to keep calculating while sending/receiving is going on: non-blocking operations

```
int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm
m, MPI_Request* request)
```

```
int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm
m, MPI_Request* request)
```

- Return immediately, i.e., do not wait until data is copied (will be done in background)
- `request` . . . address of a data structure for information about the operation

Non-blocking Operations

- Sometimes, we want to keep calculating while sending/receiving is going on: non-blocking operations

```
int MPI_Isend(...MPI_Request* request)
```

```
int MPI_Irecv(...MPI_Request* request)
```

- Return immediately, i.e., do not wait until data is copied (will be done in background)
- `request` ... address of a data structure for information about the operation

Non-blocking Operations

- If we start an *asynchronous* operation, like sending or receiving...

Non-blocking Operations

- If we start an *asynchronous* operation, like sending or receiving...
- ...how do we know when we can change the data (being sent) or use the data (being received) if the operation returns immediately?

Non-blocking Operations

- If we start an *asynchronous* operation, like sending or receiving...
- ...how do we know when we can change the data (being sent) or use the data (being received) if the operation returns immediately?

```
int MPI_Test(MPI_Request* request, int* flag, MPI_Status* status)
```

Non-blocking Operations

- If we start an *asynchronous* operation, like sending or receiving...
- ...how do we know when we can change the data (being sent) or use the data (being received) if the operation returns immediately?

```
int MPI_Test(MPI_Request* request, int* flag, ...)
```

- check operation status
- stores `flag=1` if operation is finished, `flag=0` if it is ongoing

Non-blocking Operations

- If we start an *asynchronous* operation, like sending or receiving...
- ...how do we know when we can change the data (being sent) or use the data (being received) if the operation returns immediately?

```
int MPI_Test(MPI_Request* request, int* flag, ...)
```

- check operation status
- stores `flag=1` if operation is finished, `flag=0` if it is ongoing

```
int MPI_Wait(MPI_Request* request, MPI_Status* status)
```

Non-blocking Operations

- If we start an *asynchronous* operation, like sending or receiving...
- ...how do we know when we can change the data (being sent) or use the data (being received) if the operation returns immediately?

```
int MPI_Test(MPI_Request* request, int* flag, ...)
```

- check operation status
- stores `flag=1` if operation is finished, `flag=0` if it is ongoing

```
int MPI_Wait(MPI_Request* request, MPI_Status* status)
```

- blocks until operation has finished

Non-blocking Operations

- If we start an *asynchronous* operation, like sending or receiving...
- ...how do we know when we can change the data (being sent) or use the data (being received) if the operation returns immediately?

```
int MPI_Test(MPI_Request* request, int* flag, ...)
```

- check operation status
- stores `flag=1` if operation is finished, `flag=0` if it is ongoing

```
int MPI_Wait(MPI_Request* request, MPI_Status* status)
```

- blocks until operation has finished

```
int MPI_Waitany(int count, MPI_Request array_of_requests[], int *index, MPI_Status *status)
```

Non-blocking Operations

- If we start an *asynchronous* operation, like sending or receiving...
- ...how do we know when we can change the data (being sent) or use the data (being received) if the operation returns immediately?

```
int MPI_Test(MPI_Request* request, int* flag, ...)
```

- check operation status
- stores `flag=1` if operation is finished, `flag=0` if it is ongoing

```
int MPI_Wait(MPI_Request* request, MPI_Status* status)
```

- blocks until operation has finished

```
int MPI_Waitany(int count, ..., int *index, ...)
```

- blocks until *one* of the `count` operations in `array_of_requests` has finished

Non-blocking Operations

- If we start an *asynchronous* operation, like sending or receiving...
- ...how do we know when we can change the data (being sent) or use the data (being received) if the operation returns immediately?

```
int MPI_Test(MPI_Request* request, int* flag, ...)
```

- check operation status
- stores `flag=1` if operation is finished, `flag=0` if it is ongoing

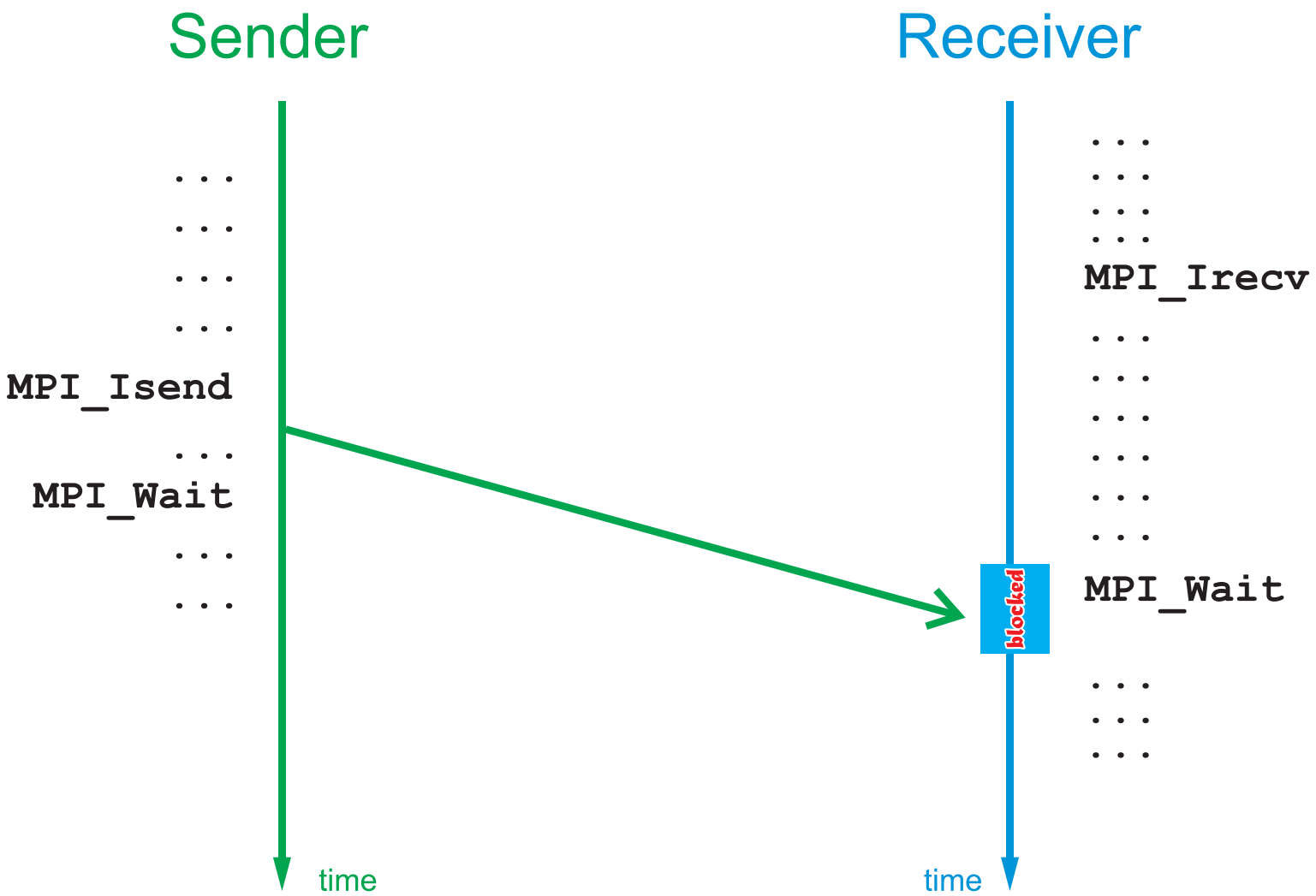
```
int MPI_Wait(MPI_Request* request, MPI_Status* status)
```

- blocks until operation has finished

```
int MPI_Waitany(int count, ..., int *index, ...)
```

- blocks until *one* of the `count` operations in `array_of_requests` has finished
- returns index of finished operation in `index`

MPI_Isend and MPI_Irecv



Listing: Non-blocking Point-to-Point communication [33] (nonBlockingPointToPoint.c)

```c
#include <mpi.h>    // import MPI header
#include <stdio.h>  // needed for printf

int main(int argc, char *argv[]) {
  int            rank, size, prev, next;
  char           receiveBuffer[30], sendBuffer[30];
  MPI_Request    receiveRequest, sendRequest;
  MPI_Status     status;

  MPI_Init(&argc,&argv); // initialize MPI
  MPI_Comm_size(MPI_COMM_WORLD, &size); // get own rank / ID
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); // get total number of processes

  next = ((rank + 1) % size); // get rank of process to receive from
  // _initiate_ receive operation, but do not wait for its completion
  MPI_Irecv(receiveBuffer, 30, MPI_CHAR, prev, 42, MPI_COMM_WORLD, &receiveRequest);

  prev = ((rank + size - 1) % size); // get rank of process to send to
  sprintf(sendBuffer, "Non-blocking from %d!", rank);
  // _initiate_ send operation, but do not wait for its completion
  MPI_Isend(sendBuffer, 30, MPI_CHAR, next, 42, MPI_COMM_WORLD, &sendRequest);

  MPI_Wait(&receiveRequest, &status); // wait for receive to complete
  printf("%d received \"%s\"\n", rank, receiveBuffer); // print received msg

  MPI_Wait(&sendRequest, &status); // wait for send to complete

  MPI_Finalize(); // shut down MPI
  return 0;
}
```

Listing: An asynchronous Pi computation (piNonBlockingPointToPoint.c)

```c
#include <mpi.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <stdio.h>

int main(int argc, char **argv) {
  int           i, j, size, rank;  double     x, y;
  long long int *data;             MPI_Status status;
  MPI_Request   *req;

  MPI_Init(&argc, &argv);                              //initialize MPI
  MPI_Comm_size(MPI_COMM_WORLD, &size);                //get the number of processes in the global communicator
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);                //get the rank of this process within the global communicator

  data = (long long int*)malloc(sizeof(long long int) * size * 2);//allocate data (ok, waste some memory in the workers)
  memset(data, 0, (sizeof(sizeof(long long int)) * 2 * size));    //clear the data buffer

  if(rank == 0) {                                      //check if we are root
    req = (MPI_Request*)malloc(sizeof(MPI_Request) * size); //allocate request list
    for(i = size; (--i) > 0; ) {                       //initiate receives from the workers
      MPI_Irecv(&data[2*i], 2, MPI_LONG_LONG_INT, i, 42, MPI_COMM_WORLD, &req[i]);
    }

    for(i = size-2; i >= 0; i--) {                     //for each unfulfilled receive request
      MPI_Waitany(size-1, &req[1], &j, &status);       //now wait until something has been received from any worker
      j++;
      data[0] += data[2*j];                            //get the received sample size (number of points)
      data[1] += data[2*j + 1];                        //get the number of samples (points) inside the unit circle
      printf("worker %d sends estimate %G (based on %lld samples), total estimate now is %G (based on %lld samples).\n", j,
                  ((4.0 * data[2*j + 1]) / data[2*j]), data[2*j], ((4.0 * data[1]) / data[0]), data[0]);
      fflush(stdout);                                  //flush the standard out
    }
  } else {                                             //ok, we are a worker
    srand(time(NULL));
    for(data[0] = 1; data[0] < (rank * 100000000LL); data[0]++) { //make 100 000 000 samples
      x = (rand() / ((double)RAND_MAX));               //random x-coordinate in [0,1]
      y = (rand() / ((double)RAND_MAX));               //random y-coordinate in [0,1]
      if( ((x*x) + (y*y)) <= 1.0 ) {                   //did it fall into the inner circle?
        data[1]++;                                     //yes, it did - increase counter
      }
    }
    MPI_Send(&data[0], 2, MPI_LONG_LONG_INT, 0, 42, MPI_COMM_WORLD); //send worker result synchronously
  }

  MPI_Finalize();                                      //finish the MPI stuff
  return 0;
}
```

So far...

- So far: data transfer using "classical point-to-point communication"

So far...

- So far: data transfer using "classical point-to-point communication"
- `MPI_Send`, `MPI_Ssend`, `MPI_Isend`, ... `MPI_Recv`, `MPI_Irecv`, ...

So far...

- So far: data transfer using "classical point-to-point communication"
- `MPI_Send`, `MPI_Ssend`, `MPI_Isend`, ... `MPI_Recv`, `MPI_Irecv`, ...
- Addressing using the "`rank`"

So far...

- So far: data transfer using "classical point-to-point communication"
- `MPI_Send`, `MPI_Ssend`, `MPI_Isend`, ... `MPI_Recv`, `MPI_Irecv`, ...
- Addressing using the "`rank`"
- blocking/non-blocking

So far...

- So far: data transfer using "classical point-to-point communication"
- `MPI_Send`, `MPI_Ssend`, `MPI_Isend`, ... `MPI_Recv`, `MPI_Irecv`, ...
- Addressing using the "`rank`"
- blocking/non-blocking
- synchronized/non-synchronized

Process Groups

- MPI supports process groups

Process Groups

- MPI supports process groups
  - Processes can be members of arbirary groups

Process Groups

- MPI supports process groups
  - Processes can be members of arbirary groups
  - For each group it is member of, a process has a specific rank (relative to that group)

Process Groups

- MPI supports process groups
  - Processes can be members of arbirary groups
  - For each group it is member of, a process has a specific rank (relative to that group)
- So far, we only used the pre-defined group (communicator) `MPI_COMM_WORLD` of all processes

Communicators and Process Groups

- Communicators and process groups are closely related

Communicators and Process Groups

- Communicators and process groups are closely related
- But: MPI-communicators and MPI-groups are different constructs!

Communicators and Process Groups

- Communicators and process groups are closely related
- But: MPI-communicators and MPI-groups are different constructs!
- A communicator always belongs to exactly one group

Communicators and Process Groups

- Communicators and process groups are closely related
- But: MPI-communicators and MPI-groups are different constructs!
- A communicator always belongs to exactly one group
- But: A group can associated with multiple communicators

Predefined Groups/Communicators

- At startup, there are two communicators

Predefined Groups/Communicators

- At startup, there are two communicators:
  1. `MPI_COMM_WORLD` . . . corresponds to all processes

Predefined Groups/Communicators

- At startup, there are two communicators:
  1. `MPI_COMM_WORLD` ... corresponds to all processes
  2. `MPI_COMM_SELF` ... corresponds to the calling process itself

Predefined Groups/Communicators

- At startup, there are two communicators:
  1. `MPI_COMM_WORLD` ... corresponds to all processes
  2. `MPI_COMM_SELF` ... corresponds to the calling process itself
- New process groups and communicators can be created at runtime with methods such as `MPI_Group_union`, `MPI_Group_intersection`, `MPI_Group_difference`, ...

Collective Communication

- All processes within a communicator can exchange information at the same time

Collective Communication

- All processes within a communicator can exchange information at the same time
  - There are different semantics for the information exchange

Collective Communication

- All processes within a communicator can exchange information at the same time
  - There are different semantics for the information exchange
  - Either all processes or pair-wise

Collective Communication

- All processes within a communicator can exchange information at the same time
  - There are different semantics for the information exchange
  - Either all processes or pair-wise
- Synchronization usually implicitly contained

Collective Communication

- All processes within a communicator can exchange information at the same time
  - There are different semantics for the information exchange
  - Either all processes or pair-wise
- Synchronization usually implicitly contained
- Every collective operation can also be expressed with `MPI_Send` / `MPI_Recv`

Broadcast

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
```

- Simplest way to do collective communication is broadcast

Broadcast

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
```

- Simplest way to do collective communication is broadcast
- Broadcast a message from the process with rank `root` to all other processes of the communicator

Broadcast

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
```

- Simplest way to do collective communication is broadcast
- Broadcast a message from the process with rank `root` to all other processes of the communicator
- Input/Output Parameter
  - `buffer` . . . starting address of the buffer used for input (at `root`) or output (other processes)

Broadcast

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
```

- Simplest way to do collective communication is broadcast
- Broadcast a message from the process with rank `root` to all other processes of the communicator
- Input/Output Parameter
  - `buffer` ... starting address of the buffer used for input (at `root`) or output (other processes)
- Input Parameters
  - `count` ... number of entries in buffer
  - `datatype` ... data type of buffer
  - `root` ... rank of broadcast root (must be the same for all processes calling this function)
  - `comm` ... the communicator

Broadcast

Listing: Broadcast (broadcast.c)

```
#include <mpi.h>   // import MPI header
#include <stdio.h> // import for printf

int main(int argc, char *argv[]) {
  char message[60]; // space allocated for the message
  int rank;         // variable for process id

  MPI_Init(&argc, &argv); // initialize MPI
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); // get own rank

  if (rank == 0) { // create message if process is "root" (rank = 0)
    sprintf(message, "Message from root (rank %d).", rank);
  }

  // broadcast: send message to all if rank==0, otherwise receive
  MPI_Bcast(message, 60, MPI_CHAR, 0, MPI_COMM_WORLD);
  printf("The message sent/received at node %d is \"%s\"\n", rank, message);

  MPI_Finalize(); // shutdown MPI
  return 0;
}
```

Scatter

```
int MPI_Scatter(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt,
atatype recvtype, int root, MPI_Comm comm)
```

- Divides an array of `sendcnt` elements into n pieces, where n is the number of processes in a communicator

Scatter

```
int MPI_Scatter(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt,
atatype recvtype, int root, MPI_Comm comm)
```

- Divides an array of `sendcnt` elements into n pieces, where n is the number of processes in a communicator
- If root: sends the pieces to each of the n processes (including itself)

Scatter

```
int MPI_Scatter(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt,
atatype recvtype, int root, MPI_Comm comm)
```

- Divides an array of `sendcnt` elements into n pieces, where n is the number of processes in a communicator
- If root: sends the pieces to each of the n processes (including itself)
- If not root: receive a data piece

Scatter

```
int MPI_Scatter(...
```

- Divides an array of `sendcnt` elements into n pieces, where n is the number of processes in a communicator
- If root: sends the pieces to each of the n processes (including itself)
- If not root: receive a data piece
- Input Parameters
  - `sendbuf` ... address of send buffer (only relevant at `root`)
  - `sendcount` ... number of elements sent to each process (only relevant at `root`)
  - `sendtype` ... data type of send buffer elements (only relevant at `root`)
  - `recvcount` ... number of elements in receive buffer
  - `recvtype` ... data type of receive buffer elements
  - `root` ... rank of sending process
  - `comm` ... the communicator to use

Scatter

```
int MPI_Scatter(...
```

- Divides an array of `sendcnt` elements into n pieces, where n is the number of processes in a communicator
- If root: sends the pieces to each of the n processes (including itself)
- If not root: receive a data piece
- Input Parameters
  - `sendbuf` ... address of send buffer (only relevant at `root`)
  - `sendcount` ... number of elements sent to each process (only relevant at `root`)
  - `sendtype` ... data type of send buffer elements (only relevant at `root`)
  - `recvcount` ... number of elements in receive buffer
  - `recvtype` ... data type of receive buffer elements
  - `root` ... rank of sending process
  - `comm` ... the communicator to use
- Output Parameter
  - `recvbuf` ... address of receive buffer

Gather

```
int MPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype
ype, int root, MPI_Comm comm)
```

- Complement to `MPI_Scatter`

Gather

```
ype, int MPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype
int root, MPI_Comm comm)
```

- Complement to `MPI_Scatter`
- Receives data in small arrays from all processes in a communicator

Gather

```
ype, int MPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype
int root, MPI_Comm comm)
```

- Complement to `MPI_Scatter`
- Receives data in small arrays from all processes in a communicator
- If root: combines all the data into one array (order like in `MPI_Gather`)

Gather

`int MPI_Gather(...`

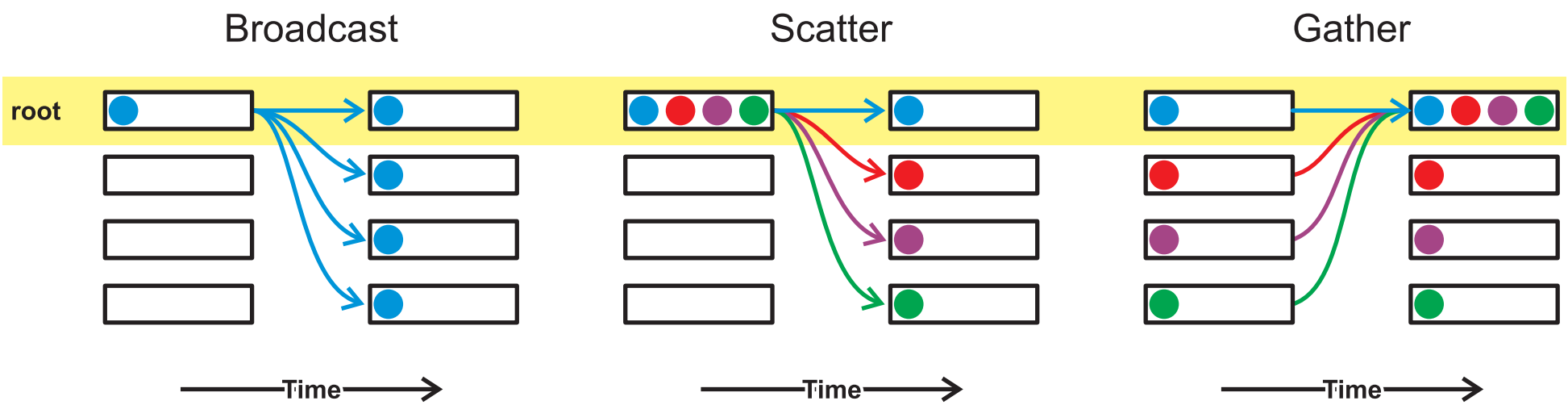
- Complement to `MPI_Scatter`
- Receives data in small arrays from all processes in a communicator
- If root: combines all the data into one array (order like in `MPI_Gather`)
- Input Parameters
  - `sendbuf` ... address of send buffer
  - `sendcount` ... number of elements in the send buffer
  - `sendtype` ... data type of send buffer elements
  - `recvcount` ... number of elements in receive buffer (only relevant at `root`)
  - `recvtype` ... data type of receive buffer elements (only relevant at `root`)
  - `root` ... rank of receiving process
  - `comm` ... the communicator to use

Gather

`int MPI_Gather(...`

- Complement to `MPI_Scatter`
- Receives data in small arrays from all processes in a communicator
- If root: combines all the data into one array (order like in `MPI_Gather`)
- Input Parameters
  - `sendbuf` ... address of send buffer
  - `sendcount` ... number of elements in the send buffer
  - `sendtype` ... data type of send buffer elements
  - `recvcount` ... number of elements in receive buffer (only relevant at `root`)
  - `recvtype` ... data type of receive buffer elements (only relevant at `root`)
  - `root` ... rank of receiving process
  - `comm` ... the communicator to use
- Output Parameter
  - `recvbuf` ... address of receive buffer (only relevant at `root`)

Broadcast, Scatter, & Gather



Listing: Gather/Scatter: bare bones (gatherScatterBareBones.c)

```c
#include <mpi.h>    // import MPI header
#include <stdio.h>  // needed for printf

#define DATA_SIZE 1024 // the data size

int main(int argc, char *argv[]) {
  int send[DATA_SIZE], recv[DATA_SIZE];
  int rank, size, count, root, res;
  MPI_Status status;

  MPI_Init(&argc, &argv); // initialize MPI
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); // get own rank/ID
  MPI_Comm_size(MPI_COMM_WORLD, &size); // get total number of processes

  if(rank == 0) { //If root: Generate data to be distributed.
  }

  //Send data to all nodes. here: an integer array of length "count".
  count = (DATA_SIZE / size); // each receive gets chunk of same size
  // scatter: if rank=0, send data (and get own share); otherwise: receive data
  MPI_Scatter(send, count, MPI_INT, recv, count, MPI_INT, 0, MPI_COMM_WORLD);

  // Each node processes its share of data and sends the result (here: int "res") to
      root.
  MPI_Gather(&res, 1, MPI_INT, recv, 1, MPI_INT, 0, MPI_COMM_WORLD);

  if(rank == 0) { //If root: process the received data.
  }

  MPI_Finalize(); // shut down MPI
  return 0;
}
```

Listing: Gather/Scatter: Count Primes (gatherScatterPrimes.c)

```c
#include <mpi.h>    // import MPI header
#include <stdio.h>  // needed for printf
#include <math.h>   // needed for sqrt

#define DATA_SIZE 1024 // let's count the primes among the first 1024 numbers

int main(int argc, char *argv[]) {
  int send[DATA_SIZE], recv[DATA_SIZE];
  int rank, size, count, root, res, i, j;
  MPI_Status status;

  MPI_Init(&argc, &argv); // initialize MPI
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); // get own rank/ID
  MPI_Comm_size(MPI_COMM_WORLD, &size); // get total number of processes

  if(rank == 0) { //generate data (i.e., the first DATA_SIZE natural numbers) if root
    for(i = DATA_SIZE; (--i)>=0; ) { send[i] = (i + 1); }
  }

  count = (DATA_SIZE / size); // divide the data among _all_ processes
  // scatter: if rank=0, send data (and get own share); otherwise: receive data
  MPI_Scatter(send, count, MPI_INT, recv, count, MPI_INT, 0, MPI_COMM_WORLD);

  // each node now processes its share of the numbers
  res = count; //here: count how many prime numbers are contained in the array
  for(i = count; (--i) >= 0; ) { //j: test all odd numbers 1<j<sqrt(j)|1
    for(j = ((int)(sqrt(recv[i]))|1); j>1; j -= 2) {
      if((recv[i] % j) == 0) { // if a number can be divided by j
        res--; // it cannot be a prime number, reduce number of primes
        break; } // break inner loop to test next number
    }
  }
  printf("Process␣%d␣discovered␣%d␣primes␣in␣the␣numbers␣from␣%d␣to␣%d.\n", rank, res, recv[0], recv[count-1]);

  // gather: all processes send data to root, only root receives data
  MPI_Gather(&res, 1, MPI_INT, recv, 1, MPI_INT, 0, MPI_COMM_WORLD);

  if(rank == 0) { //if root, process the received data
    res = 0;
    for(i = size; (--i) >= 0; ) { //add up the prime number counts
      res += recv[i];
    }
    printf("The␣total␣number␣of␣primes␣in␣the␣first␣%d␣natural␣numbers␣is␣%d.\n", (count*size), res);
  }

  MPI_Finalize();
  return 0;
}
```

Listing: Gather/Scatter: Pi (piGatherScatter.c)

```c
#include <mpi.h>    // import MPI header
#include <stdio.h>  // needed for printf
#include <stdlib.h> // for rand and RAND_MAX
#include <string.h> // for memset
#include <time.h>   // for srand(time(NULL));

int main(int argc, char **argv) {
  int           i, size, rank;     double     x, y;
  long long int *data, worker[2];  MPI_Status status;

  MPI_Init(&argc, &argv);                              //initialize MPI
  MPI_Comm_size(MPI_COMM_WORLD, &size);                //get the number of processes in the global communicator
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);                //get the rank of this process within the global communicator

  if(rank != 0) {
    worker[0] = worker[1] = 0LL;                       //the local worker array
    srand(time(NULL));
    for(worker[0] = 1; worker[0] < (rank * 100000000LL); worker[0]++) { //make 100 000 000 samples
      x = (rand() / ((double)RAND_MAX));               //random x-coordinate in [0,1]
      y = (rand() / ((double)RAND_MAX));               //random y-coordinate in [0,1]
      if( ((x*x) + (y*y)) <= 1.0 ) {                   //did it fall into the inner circle?
        worker[1]++;                                   //yes, it did - increase counter
      }
    }
  }

  data = (long long int*)malloc(sizeof(long long int) * size * 2); //allocate data (ok, waste some memory in the workers)
  memset(data, 0, (sizeof(data[0]) * 2 * size));       //clear the data buffer
  MPI_Gather(worker, 2, MPI_LONG_LONG_INT, data, 2, MPI_LONG_LONG_INT, 0, MPI_COMM_WORLD);  //gather results

  if(rank == 0) {                                      //root now evaluates results
    for(i = size; (--i) > 0; ) {                       //receive data from the workers
      data[0] += data[2*i];                            //get the received sample size (number of points)
      data[1] += data[2*i+1];                          //get the number of samples (points) inside the unit circle
      printf("worker %d sends estimate %G (based on %lld samples), total estimate now is %G (based on %lld samples).\n", i,
            ((4.0 * data[2*i + 1]) / data[2*i]), data[2*i], ((4.0 * data[1]) / data[0]), data[0]);
    }
  }
  MPI_Finalize();                                      //finish the MPI stuff
  return 0;
}
```

Reduce

```
int MPI_Reduce (void *sendbuf, void *recvbuf, int count, MPI_Datatype type, MPI_Op op, int root,
I_Comm comm)
```

- Similar to `MPI_Gather`

Reduce

```
int MPI_Reduce (void *sendbuf, void *recvbuf, int count, MPI_Datatype type, MPI_Op op, int root,
I_Comm comm)
```

- Similar to `MPI_Gather`, but
- Data is aggregated by applying a specific reduction operation `op`

Reduce

```
int MPI_Reduce (void *sendbuf, void *recvbuf, int count, MPI_Datatype type, MPI_Op op, int root,
I_Comm comm)
```

- Similar to `MPI_Gather`, but
- Data is aggregated by applying a specific reduction operation `op`
- Therefore, volume of data transmission is reduced (not all needs to be sent)

Reduce

```
int MPI_Reduce(...
```

- Similar to `MPI_Gather`, but
- Data is aggregated by applying a specific reduction operation `op`
- Therefore, volume of data transmission is reduced (not all needs to be sent)
- Values for `op`
  - `MPI_LAND` / `MPI_BAND` ... logical/bitwise *and*
  - `MPI_LOR` / `MPI_BOR` ... logical/bitwise *or*
  - `MPI_LXOR` / `MPI_BXOR` ... logical/bitwise *xor*
  - `MPI_MAX` ... compute the maximum
  - `MPI_MIN` ... compute the minimum
  - `MPI_SUM` ... compute the sum
  - `MPI_PROD` ... compute the product

Listing: Scatter/Reduce: Count Primes (reducePrimes.c)

```c
#include <mpi.h>    // import MPI header
#include <stdio.h>  // needed for printf
#include <math.h>   // needed for sqrt

#define DATA_SIZE 1024 // let's count the primes among the first 1024 numbers

int main(int argc, char *argv[]) {
  int send[DATA_SIZE], recv[DATA_SIZE];
  int rank, size, count, root, res, i, j;
  MPI_Status status;

  MPI_Init(&argc, &argv); // initialize MPI
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); // get own rank/ID
  MPI_Comm_size(MPI_COMM_WORLD, &size); // get total number of processes

  if(rank == 0) { //generate data (i.e., the first DATA_SIZE natural numbers) if root
    for(i = DATA_SIZE; (--i)>=0; ) { send[i] = (i + 1); }
  }

  count = (DATA_SIZE / size); // divide the data among _all_ processes
  // scatter: if rank=0, send data (and get own share); otherwise: receive data
  MPI_Scatter(send, count, MPI_INT, recv, count, MPI_INT, 0, MPI_COMM_WORLD);

  // each node now processes its share of the numbers
  res = count; //here: count how many prime numbers are contained in the array
  for(i = count; (--i) >= 0; ) { //j: test all odd numbers 1<j<sqrt(j)|1
    for(j = ((int)(sqrt(recv[i]))|1); j>1; j -= 2) {
      if((recv[i] % j) == 0) { // if a number can be divided by j
        res--; // it cannot be a prime number, reduce number of primes
        break; } // break inner loop to test next number
    }
  }
  printf("Process %d discovered %d primes in the numbers from %d to %d.\n", rank, res, recv[0], recv[count-1]);

  // reduce: each node takes results, applies operator MPI_SUM locally, sends result to root, where MPI_SUM is
  // applied again. (here: locally summing up does not matter, as only 1 number). The final result is returned.
  MPI_Reduce(&res, recv, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
  if(rank == 0) { //if root, print
    printf("The total number of primes in the first %d natural numbers is %d.\n", (count*size), recv[0]);
  }

  MPI_Finalize(); // shut down MPI
  return 0;
}
```

Ways to distribute

1. Can my problem be parallelized?

Ways to distribute

1. Can my problem be parallelized?
   - Is it parallel by nature or regular?

Ways to distribute

1. Can my problem be parallelized?
   - Is it parallel by nature or regular?
2. Which part of my program should I parallelize?

Ways to distribute

1. Can my problem be parallelized?
   - Is it parallel by nature or regular?
2. Which part of my program should I parallelize?
   - The stuff that takes the most time!

Ways to distribute

1. Can my problem be parallelized?
   - Is it parallel by nature or regular?
2. Which part of my program should I parallelize?
   - The stuff that takes the most time!
   - Test, trial, use profiler, ...

Ways to distribute

1. Can my problem be parallelized?
   - Is it parallel by nature or regular?
2. Which part of my program should I parallelize?
   - The stuff that takes the most time!
   - Test, trial, use profiler, ...
   - Find bottlenecks (e.g., I/O)

Ways to distribute

1. Can my problem be parallelized?
   - Is it parallel by nature or regular?
2. Which part of my program should I parallelize?
   - The stuff that takes the most time!
   - Test, trial, use profiler, ...
   - Find bottlenecks (e.g., I/O)
3. Two basic parallelization schemes

Ways to distribute

1. Can my problem be parallelized?
   - Is it parallel by nature or regular?
2. Which part of my program should I parallelize?
   - The stuff that takes the most time!
   - Test, trial, use profiler, ...
   - Find bottlenecks (e.g., I/O)
3. Two basic parallelization schemes
   1. data-based parallelization
   2. function-based parallelization

Data-based Partition

- Each worker processes a part of the data

Data-based Partition

- Each worker processes a part of the data

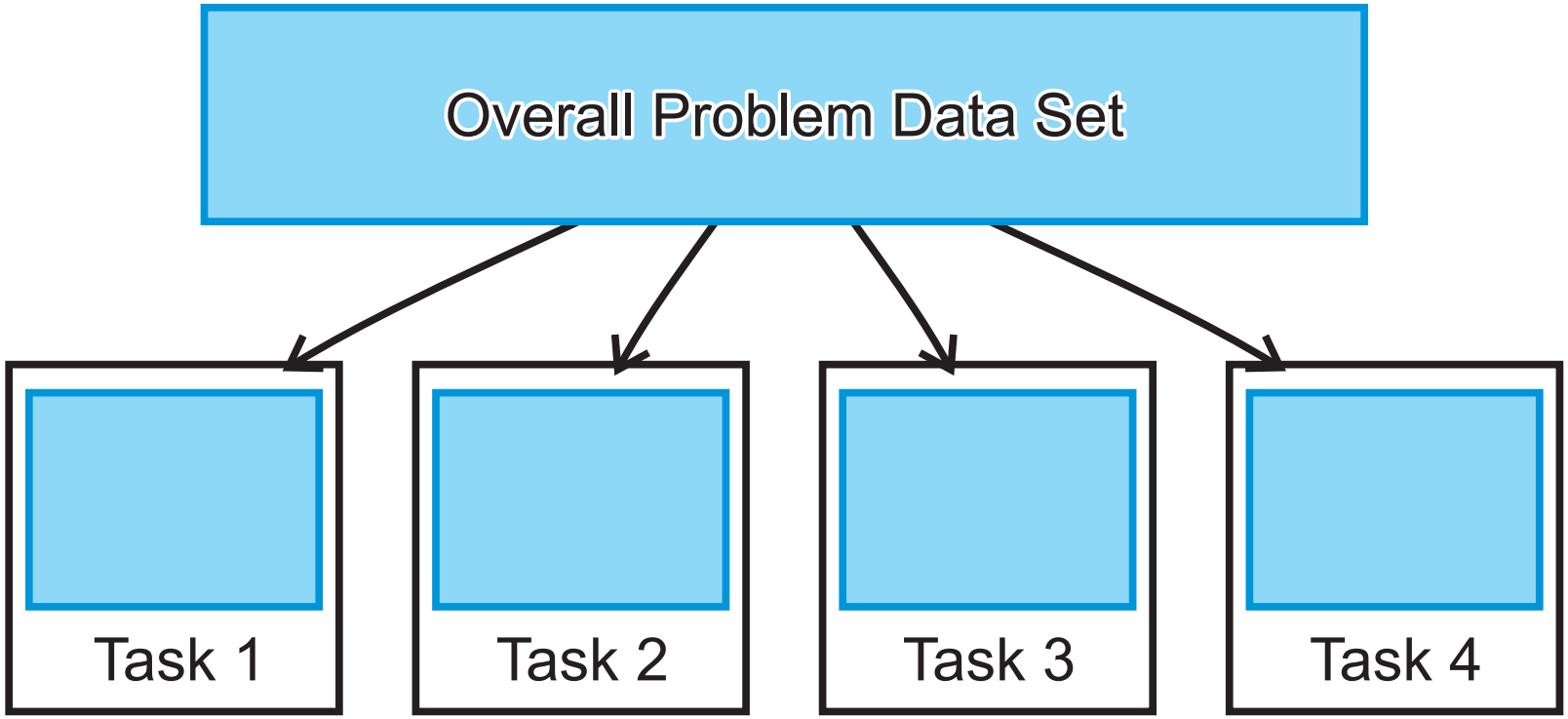
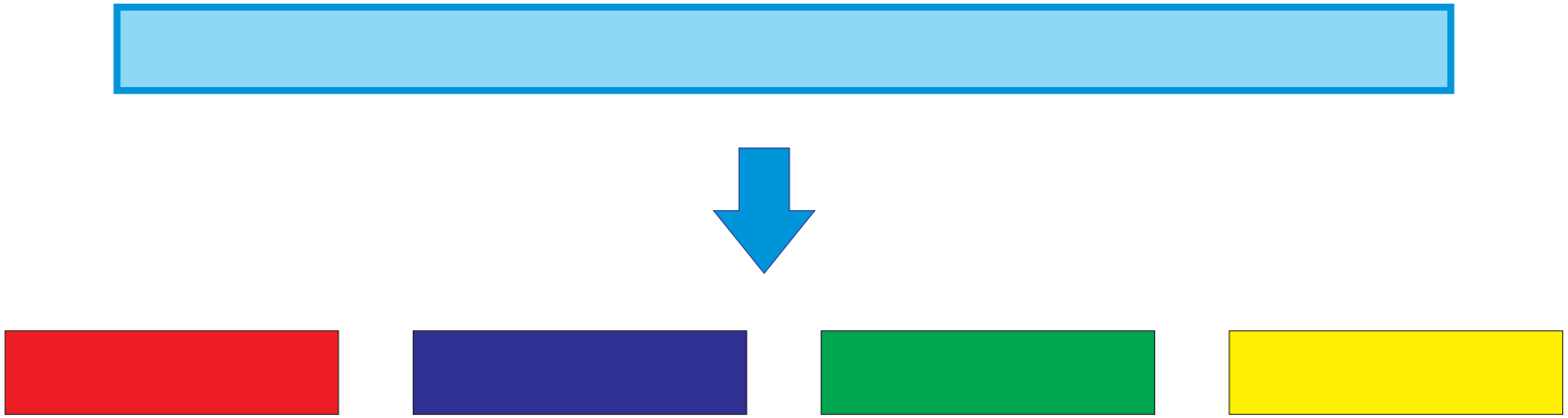
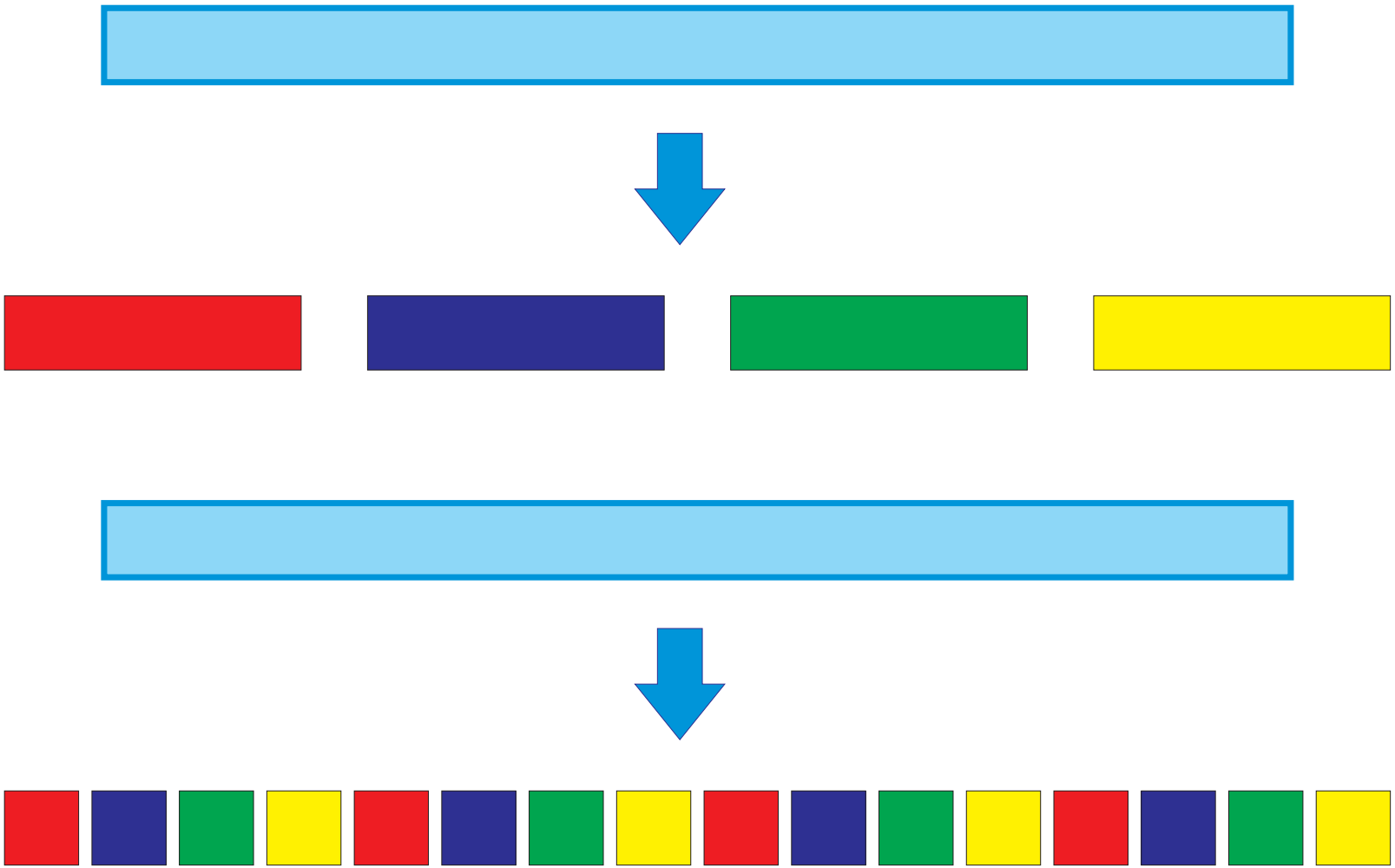


image source: [19]

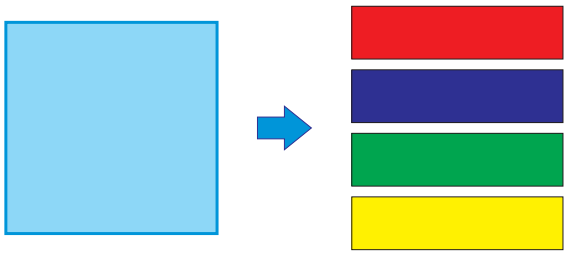
Data-based Partition: 1D bib@oldcitepB2012ITPC



Data-based Partition: 1D bib@oldcitepB2012ITPC

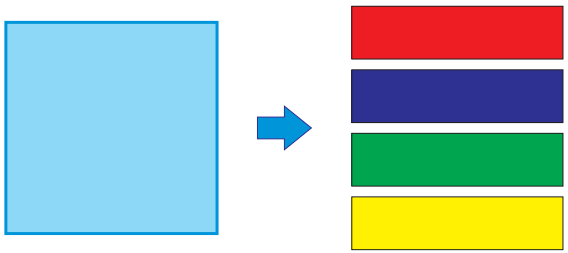


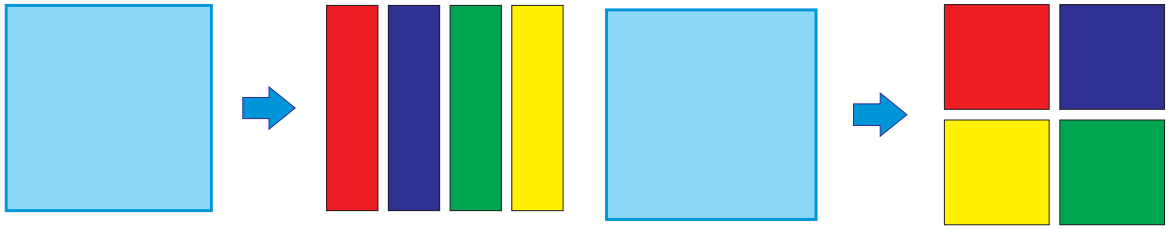
Data-based Partition: 2D bib@oldcitepB2012ITPC



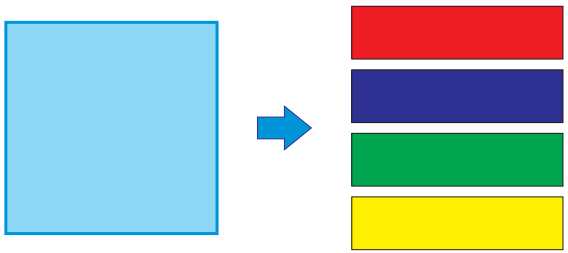
Data-based Partition: 2D bib@oldcitepB2012ITPC

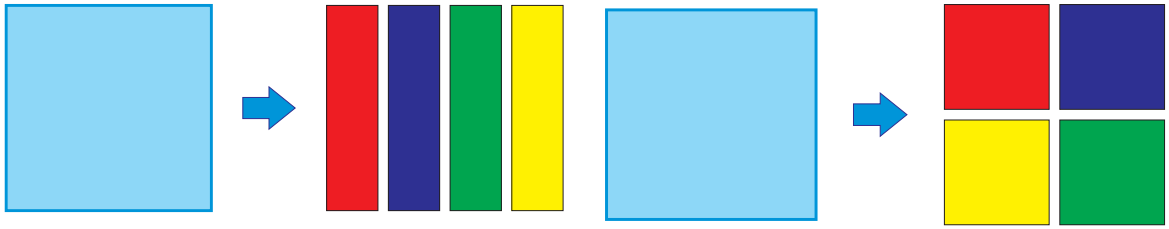
Data-based Partition: 2D bib@oldcitepB2012ITPC

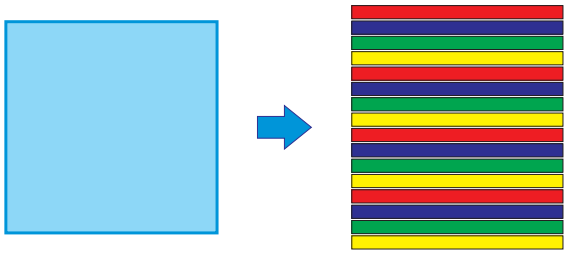




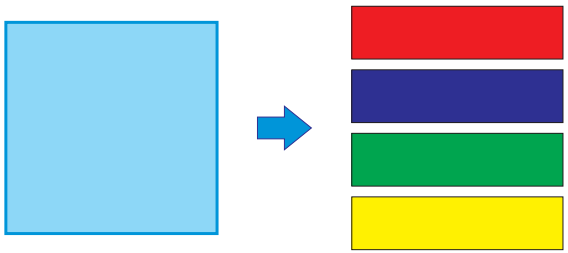
Data-based Partition: 2D bib@oldcitepB2012ITPC

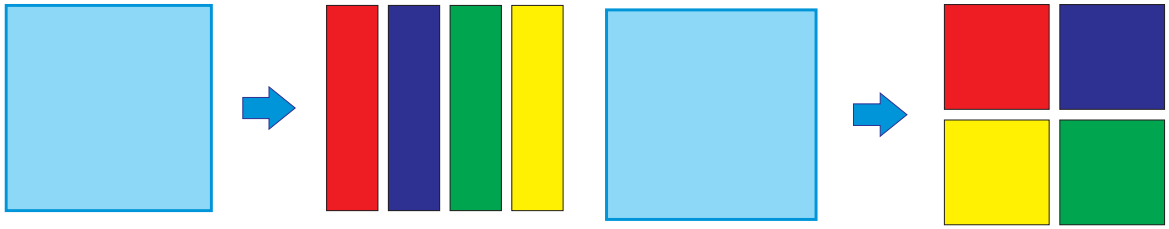


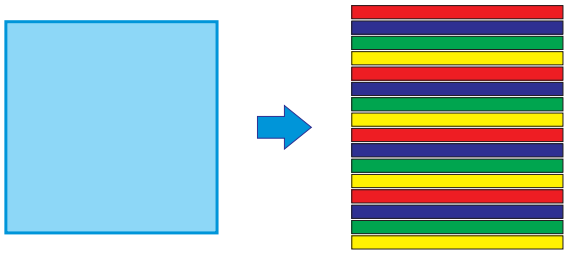


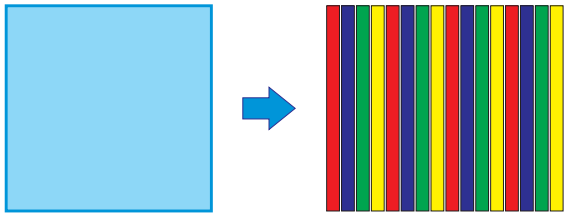


Data-based Partition: 2D bib@oldcitepB2012ITPC

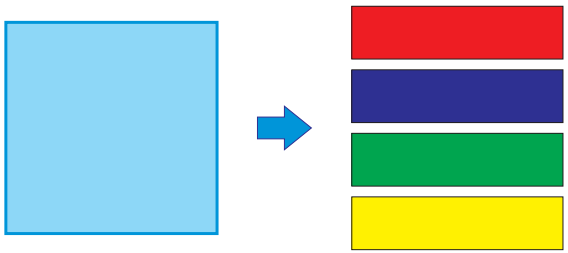


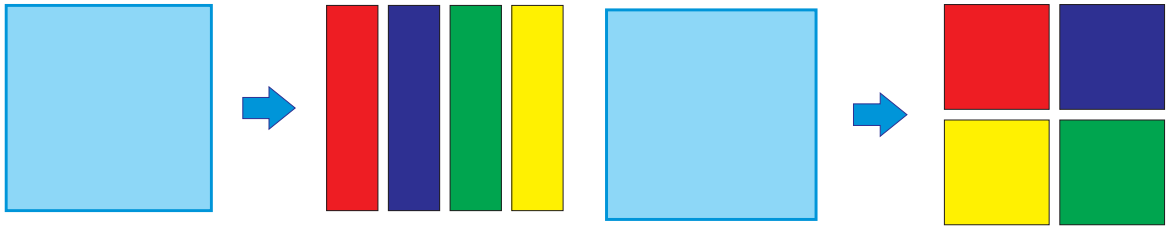


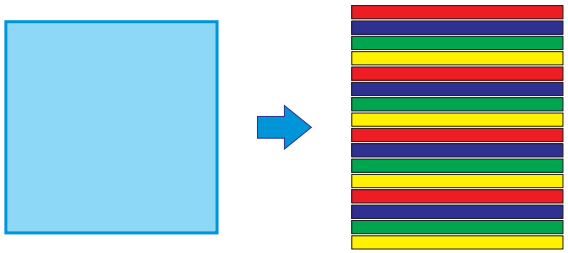


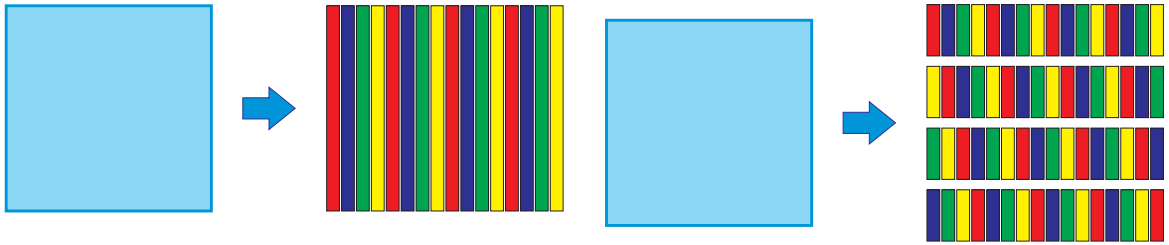


Data-based Partition: 2D bib@oldcitepB2012ITPC









Function-based Decomposition

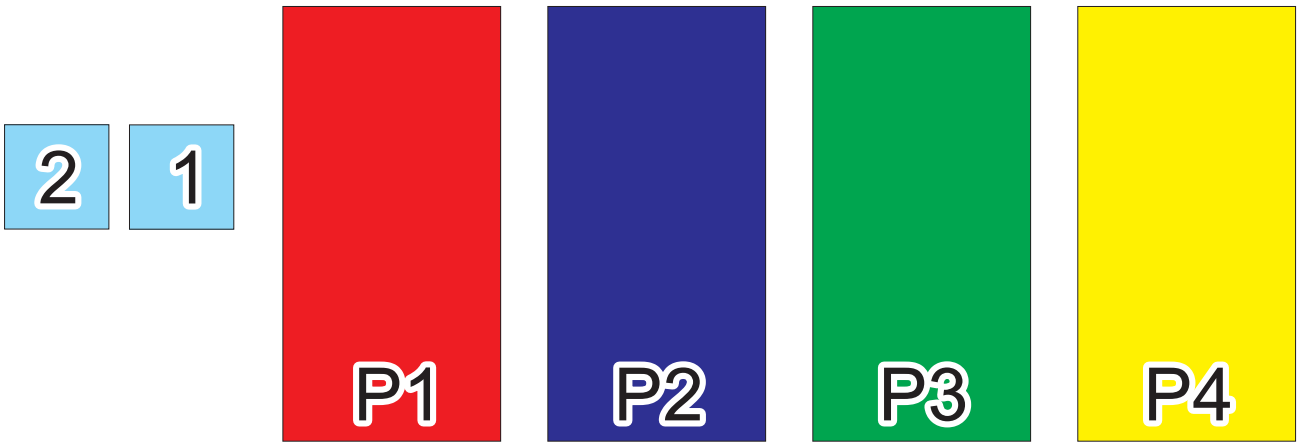
- Distribution based on functionality, instead of data

Function-based Decomposition

- Distribution based on functionality, instead of data
- Example: pipes and filters architectures

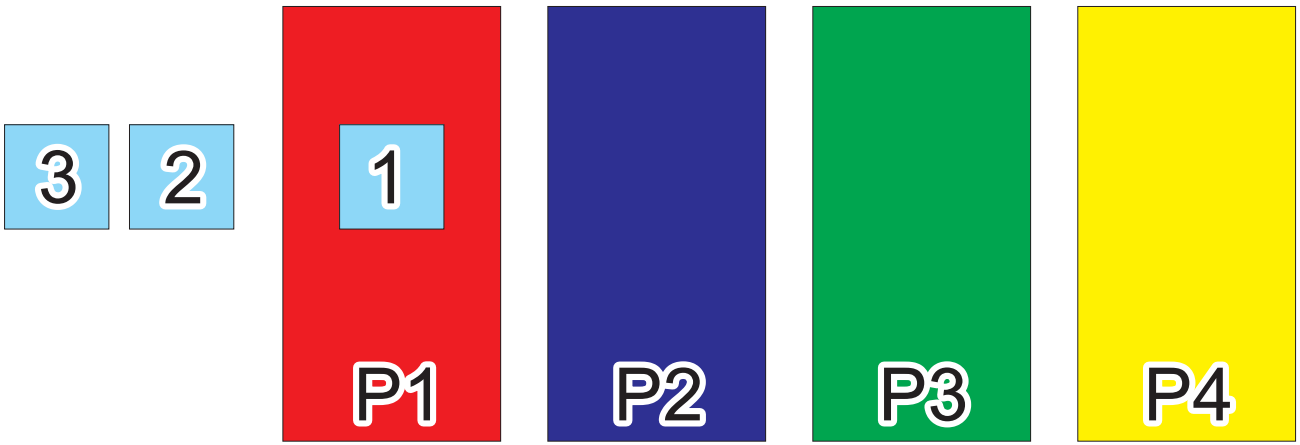
Function-based Decomposition

- Distribution based on functionality, instead of data
- Example: pipes and filters architectures



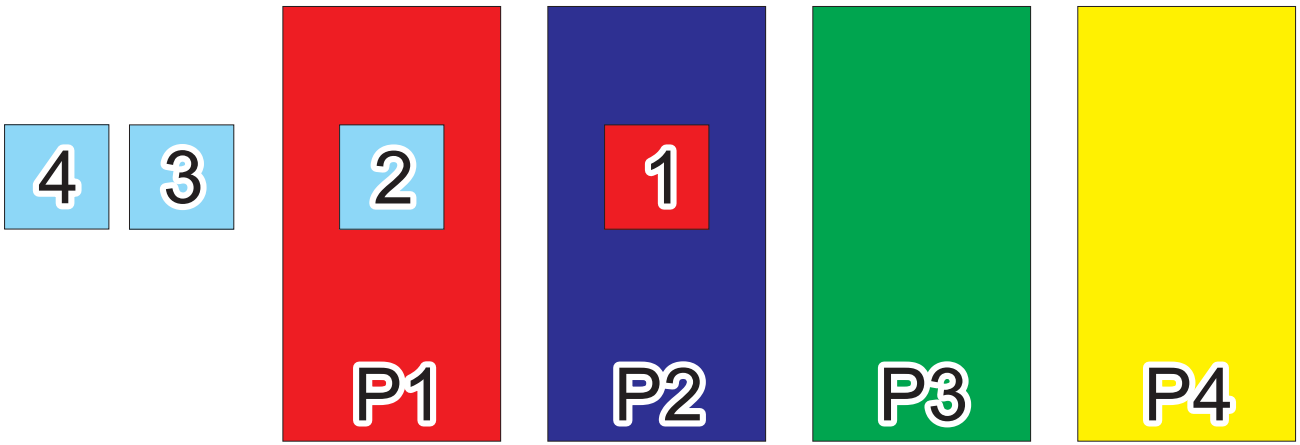
Function-based Decomposition

- Distribution based on functionality, instead of data
- Example: pipes and filters architectures



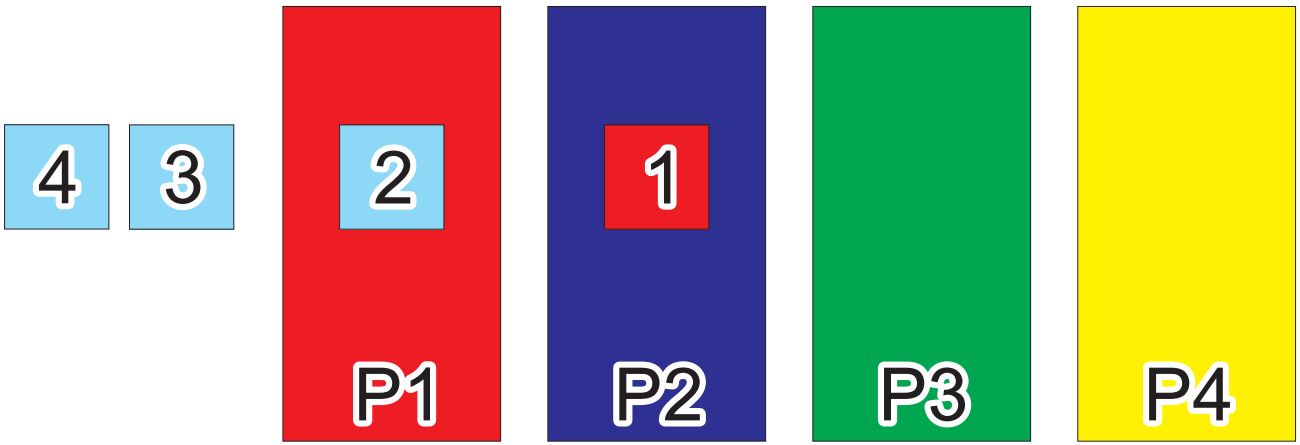
Function-based Decomposition

- Distribution based on functionality, instead of data
- Example: pipes and filters architectures



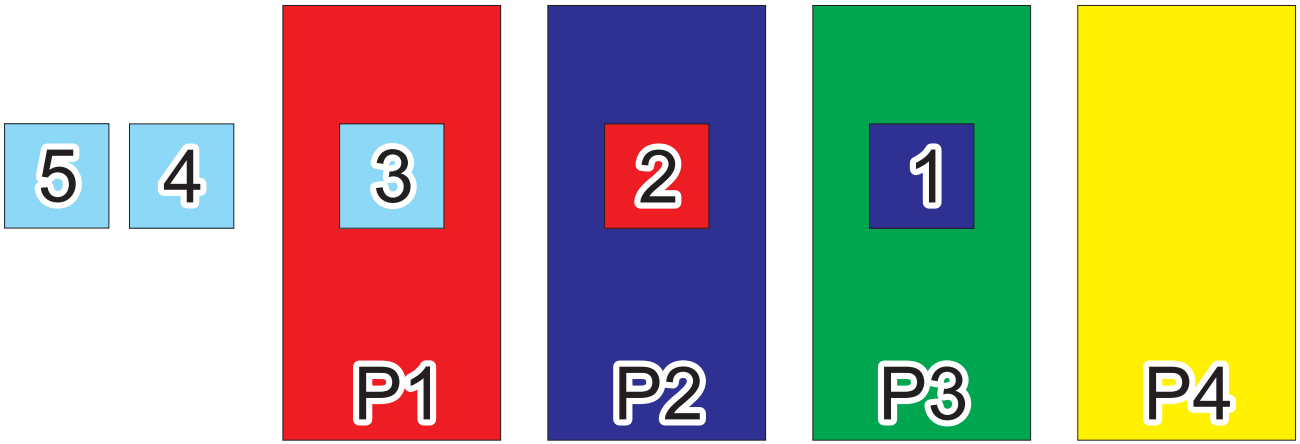
Function-based Decomposition

- Distribution based on functionality, instead of data
- Example: pipes and filters architectures



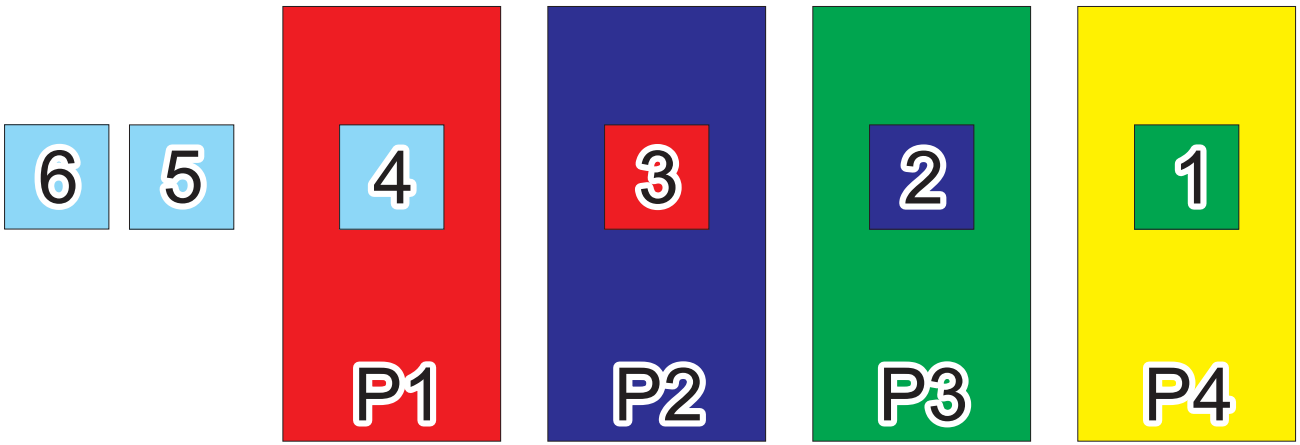
Function-based Decomposition

- Distribution based on functionality, instead of data
- Example: pipes and filters architectures



Function-based Decomposition

- Distribution based on functionality, instead of data
- Example: pipes and filters architectures

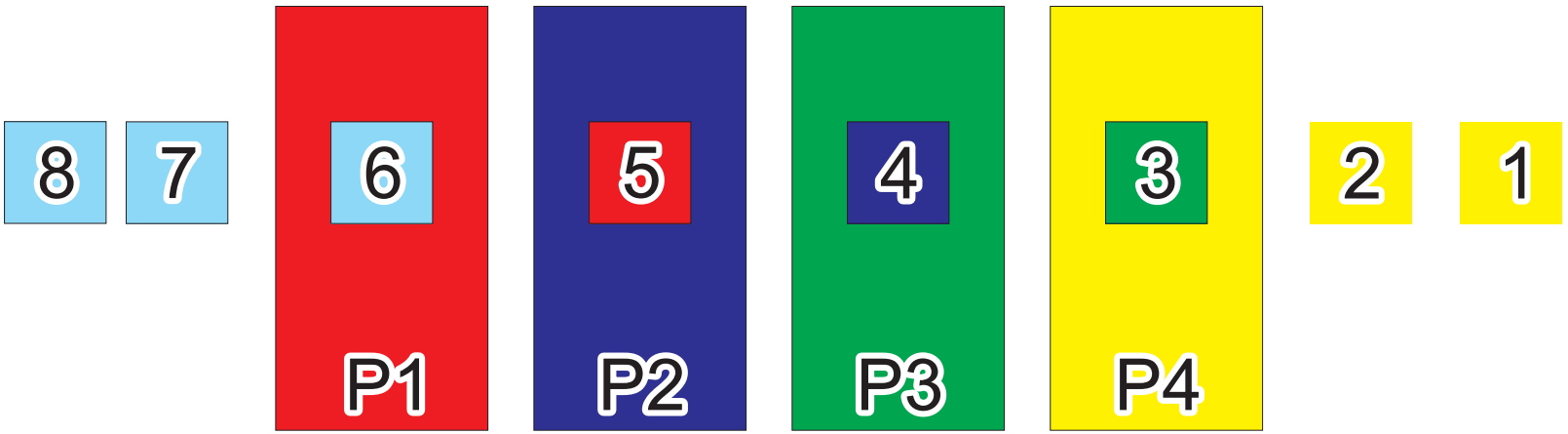


Function-based Decomposition

- Distribution based on functionality, instead of data
- Example: pipes and filters architectures

Function-based Decomposition

- Distribution based on functionality, instead of data
- Example: pipes and filters architectures



Good Distribution

- Utilize available resources as good as possible

Good Distribution

- Utilize available resources as good as possiblee.g.,
- No processor should be idle for a longer time

Good Distribution

- Utilize available resources as good as possiblee.g.,
- No processor should be idle for a longer time
- Waiting time caused by communication should be reduced

Good Distribution

- Utilize available resources as good as possiblee.g.,
- No processor should be idle for a longer time
- Waiting time caused by communication should be reduced
- Synchronization should be used as little as possible

Installation

- Programming language: C or C++
- For your operating system, you therefore need

Installation

- Programming language: C or C++
- For your operating system, you therefore need

Installation

- Programming language: C or C++
- For your operating system, you therefore need:
  1. C or C++ compiler

Installation

- Programming language: C or C++
- For your operating system, you therefore need:
  1. C or C++ compiler
  2. MPI Implementation/Framework

Windows

- Original choices for Windows

Windows

- Original choices for Windows:
  1. MinGW: Minimalist GNU for Windows [36]

Windows

- Original choices for Windows:
  1. MinGW: Minimalist GNU for Windows [36]
     - GCC, G++, Bourne Shell and that alike
     - Website: `http://www.mingw.org/`

Windows

- Original choices for Windows:
  1. MinGW: Minimalist GNU for Windows [36]
     - GCC, G++, Bourne Shell and that alike
     - Website: `http://www.mingw.org/`
  2. MPICH 2 [29]

Windows

- Original choices for Windows:
  1. MinGW: Minimalist GNU for Windows [36]
     - GCC, G++, Bourne Shell and that alike
     - Website: `http://www.mingw.org/`
  2. MPICH 2 [29]
     - High-performance and widely portable implementation of both MPI-1 and MPI-2
     - Website: `http://www.mcs.anl.gov/research/projects/mpich2/`

Windows

- Original choices for Windows:
  1. MinGW: Minimalist GNU for Windows [36]
     - GCC, G++, Bourne Shell and that alike
     - Website: `http://www.mingw.org/`
  2. MPICH 2 [29]
     - High-performance and widely portable implementation of both MPI-1 and MPI-2
     - Website: `http://www.mcs.anl.gov/research/projects/mpich2/`
- Reasons

Windows

- Original choices for Windows:
  1. MinGW: Minimalist GNU for Windows [36]
     - GCC, G++, Bourne Shell and that alike
     - Website: `http://www.mingw.org/`
  2. MPICH 2 [29]
     - High-performance and widely portable implementation of both MPI-1 and MPI-2
     - Website: `http://www.mcs.anl.gov/research/projects/mpich2/`
- Reasons:
  1. Available for both Linux and Windows

Windows

- Original choices for Windows:
  1. MinGW: Minimalist GNU for Windows [36]
     - GCC, G++, Bourne Shell and that alike
     - Website: `http://www.mingw.org/`
  2. MPICH 2 [29]
     - High-performance and widely portable implementation of both MPI-1 and MPI-2
     - Website: `http://www.mcs.anl.gov/research/projects/mpich2/`
- Reasons:
  1. Available for both Linux and Windows
  2. Relatively easy to use

Windows

- Original choices for Windows:
  1. MinGW: Minimalist GNU for Windows [36]
     - GCC, G++, Bourne Shell and that alike
     - Website: `http://www.mingw.org/`
  2. MPICH 2 [29]
     - High-performance and widely portable implementation of both MPI-1 and MPI-2
     - Website: `http://www.mcs.anl.gov/research/projects/mpich2/`
- Reasons:
  1. Available for both Linux and Windows
  2. Relatively easy to use
  3. MPICH even works with Visual Studio (but don't ask me how)

Windows

- Original choices for Windows:
  1. MinGW: Minimalist GNU for Windows [36]
     - GCC, G++, Bourne Shell and that alike
     - Website: `http://www.mingw.org/`
  2. MPICH 2 [29]
     - High-performance and widely portable implementation of both MPI-1 and MPI-2
     - Website: `http://www.mcs.anl.gov/research/projects/mpich2/`
- Reasons:
  1. Available for both Linux and Windows
  2. Relatively easy to use
  3. MPICH even works with Visual Studio (but don't ask me how)
  4. Stable technology for quite a few years [37]

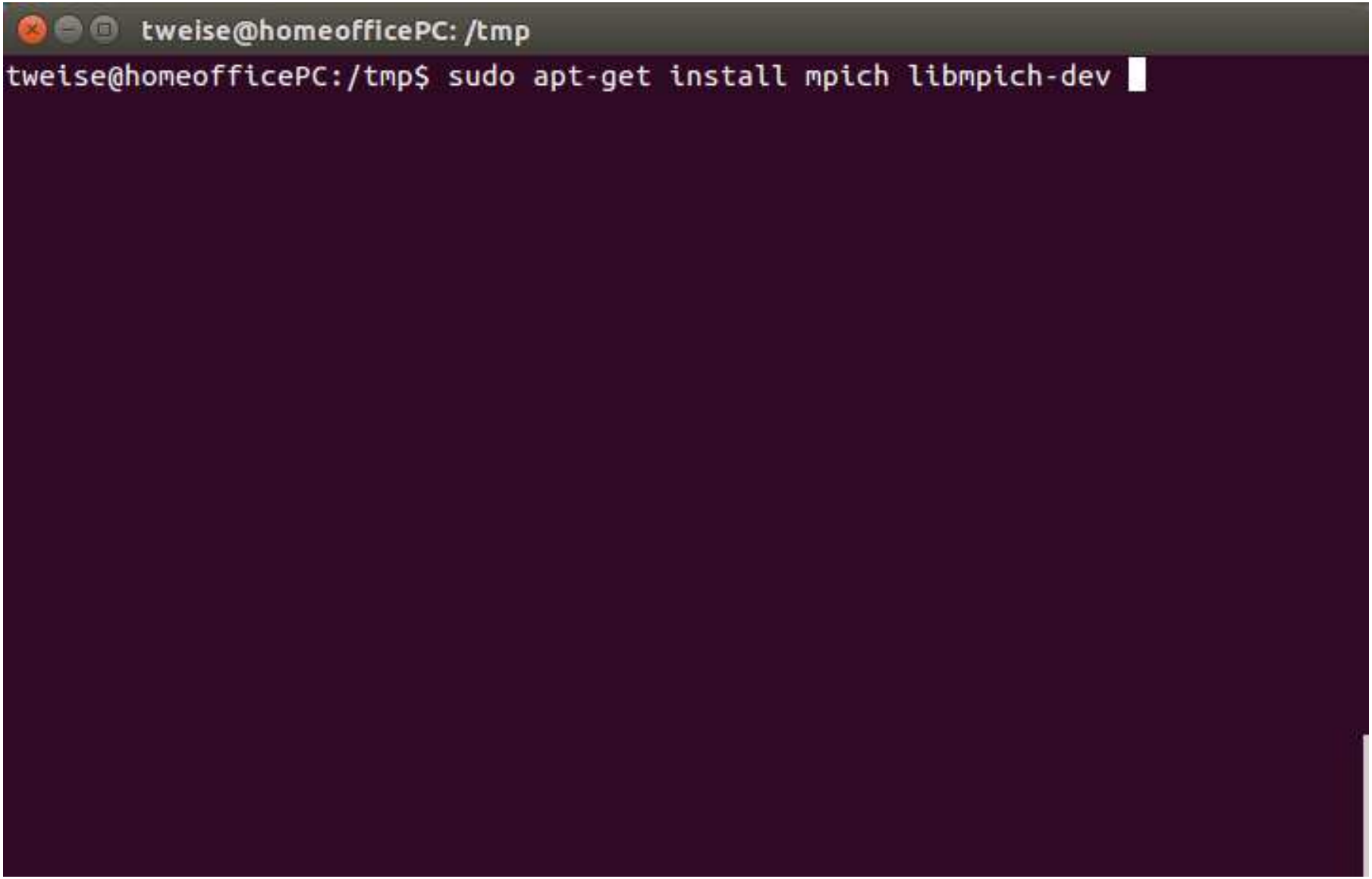
Windows

- Original choices for Windows:
  1. MinGW: Minimalist GNU for Windows [36]
     - GCC, G++, Bourne Shell and that alike
     - Website: `http://www.mingw.org/`
  2. MPICH 2 [29]
     - High-performance and widely portable implementation of both MPI-1 and MPI-2
     - Website: `http://www.mcs.anl.gov/research/projects/mpich2/`
- Reasons:
  1. Available for both Linux and Windows
  2. Relatively easy to use
  3. MPICH even works with Visual Studio (but don't ask me how)
  4. Stable technology for quite a few years [37]
  5. But this is no longer possible: MPICH is no longer available for Windows, I could not get the Microsoft MPI to work.

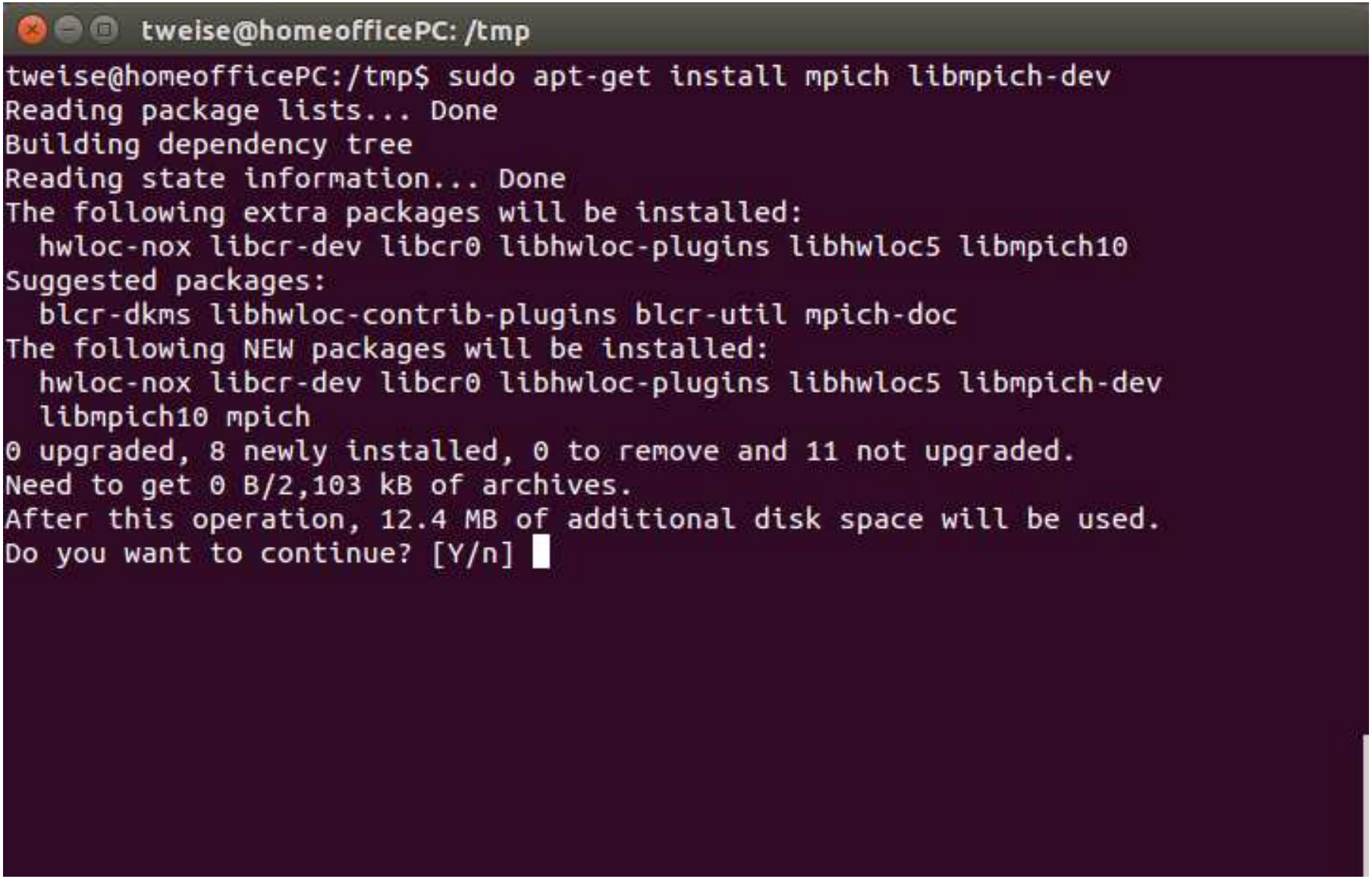
Installation: Linux

- `sudo apt-get install mpich libmpich-dev`

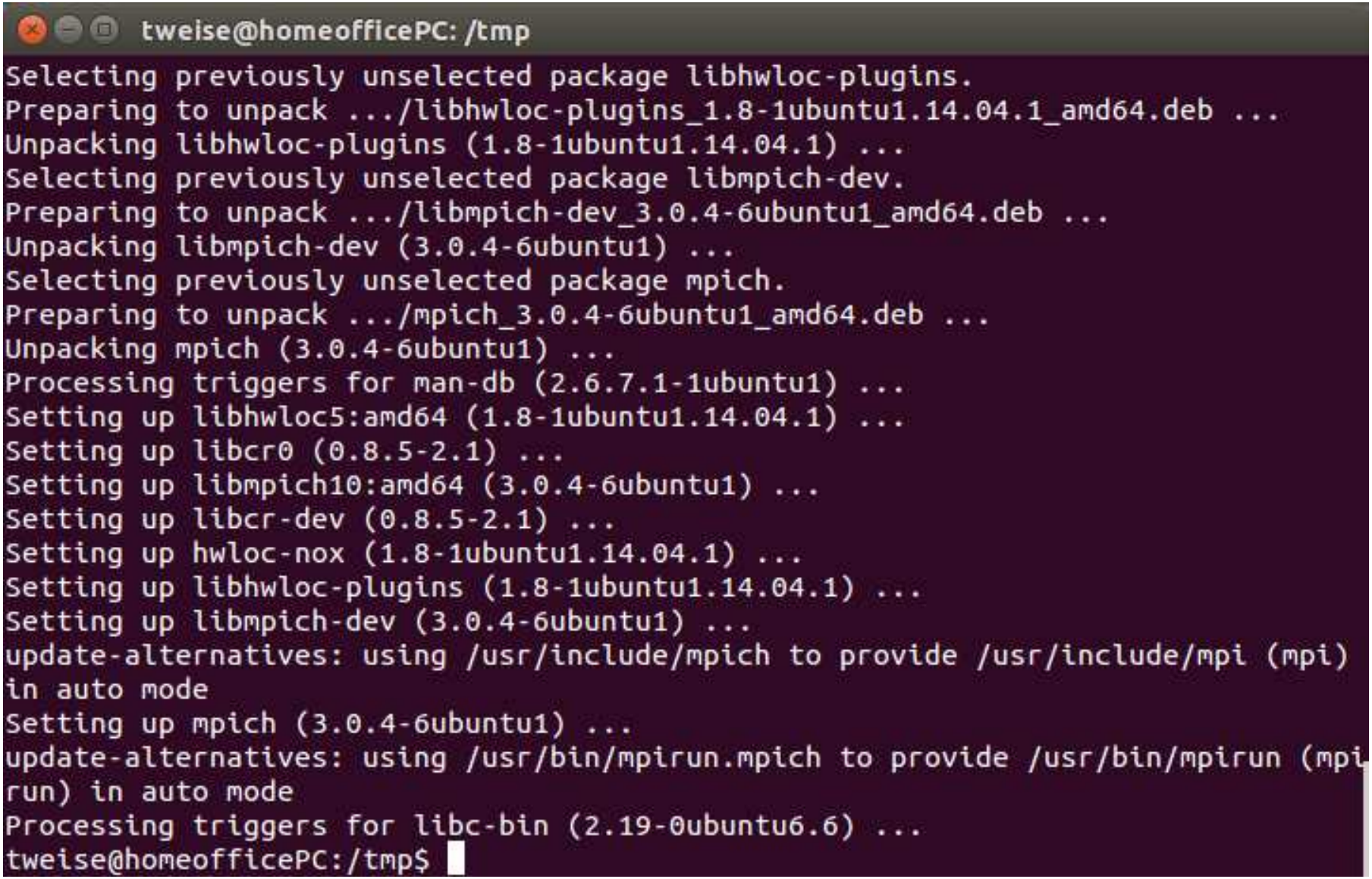
Installation: Linux



Installation: Linux



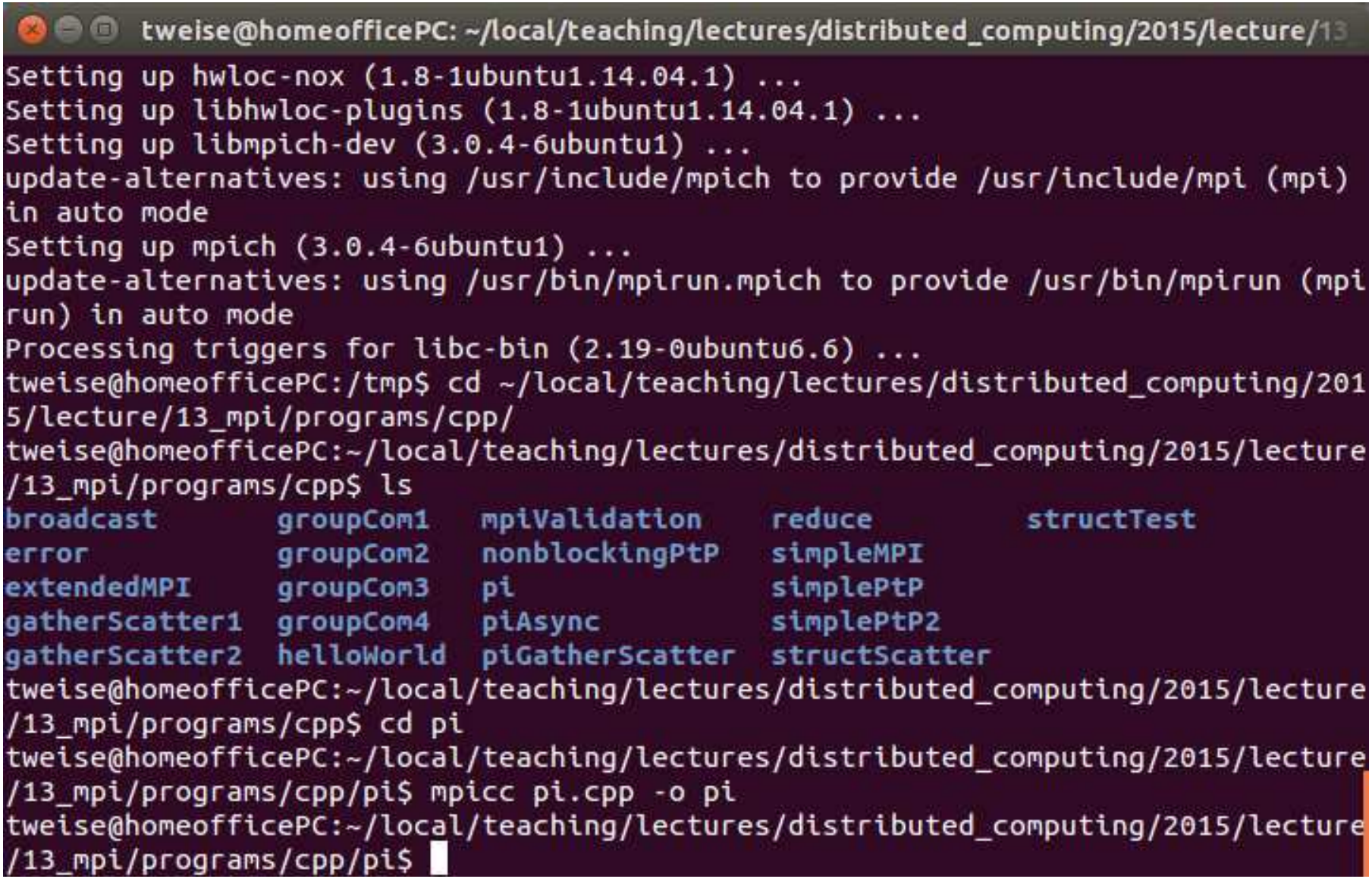
Installation: Linux



Installation: Linux

- `sudo apt-get install mpich libmpich-dev`
- `mpicc myprogram.cpp -o myprogram`

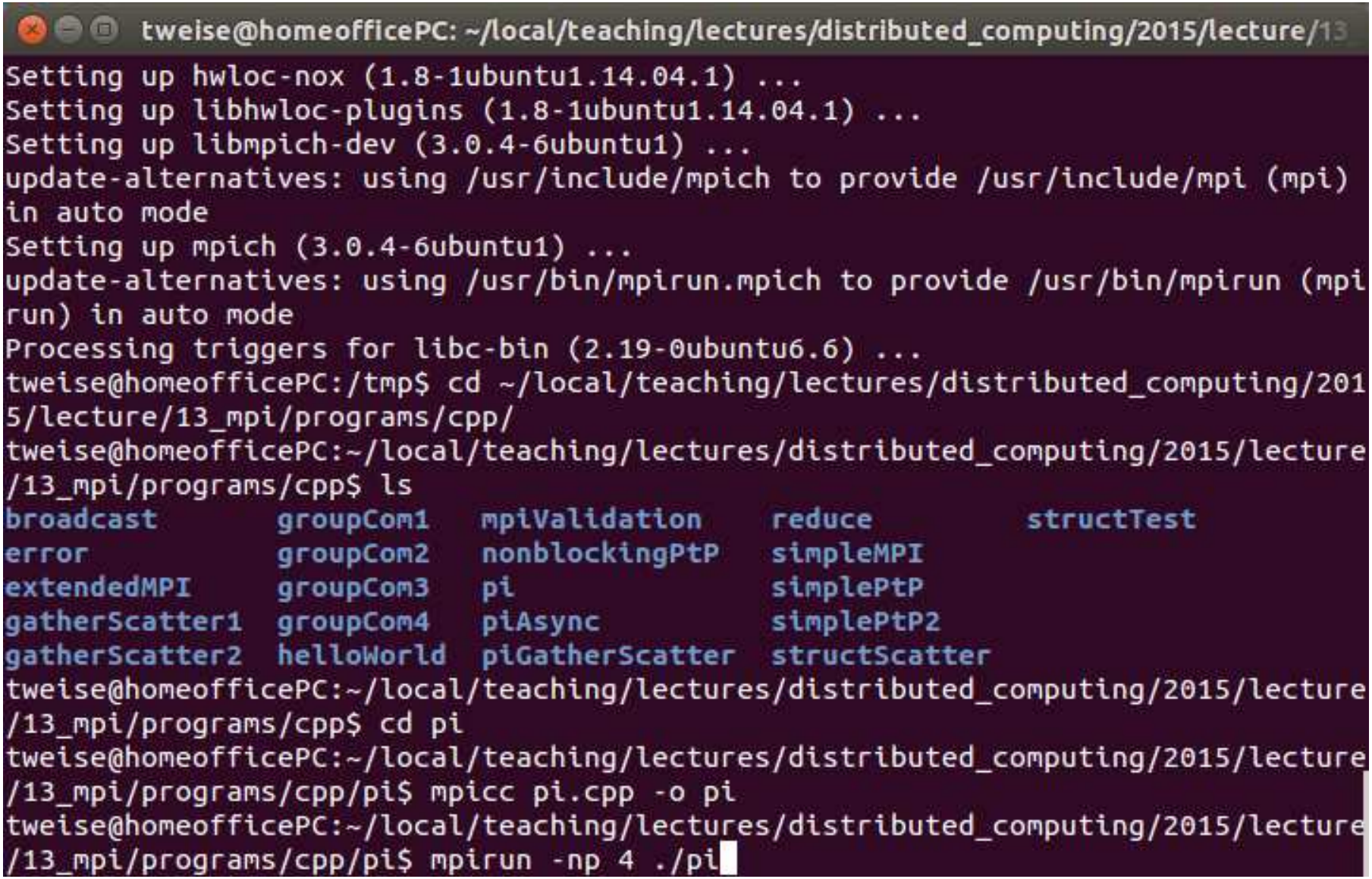
Installation: Linux



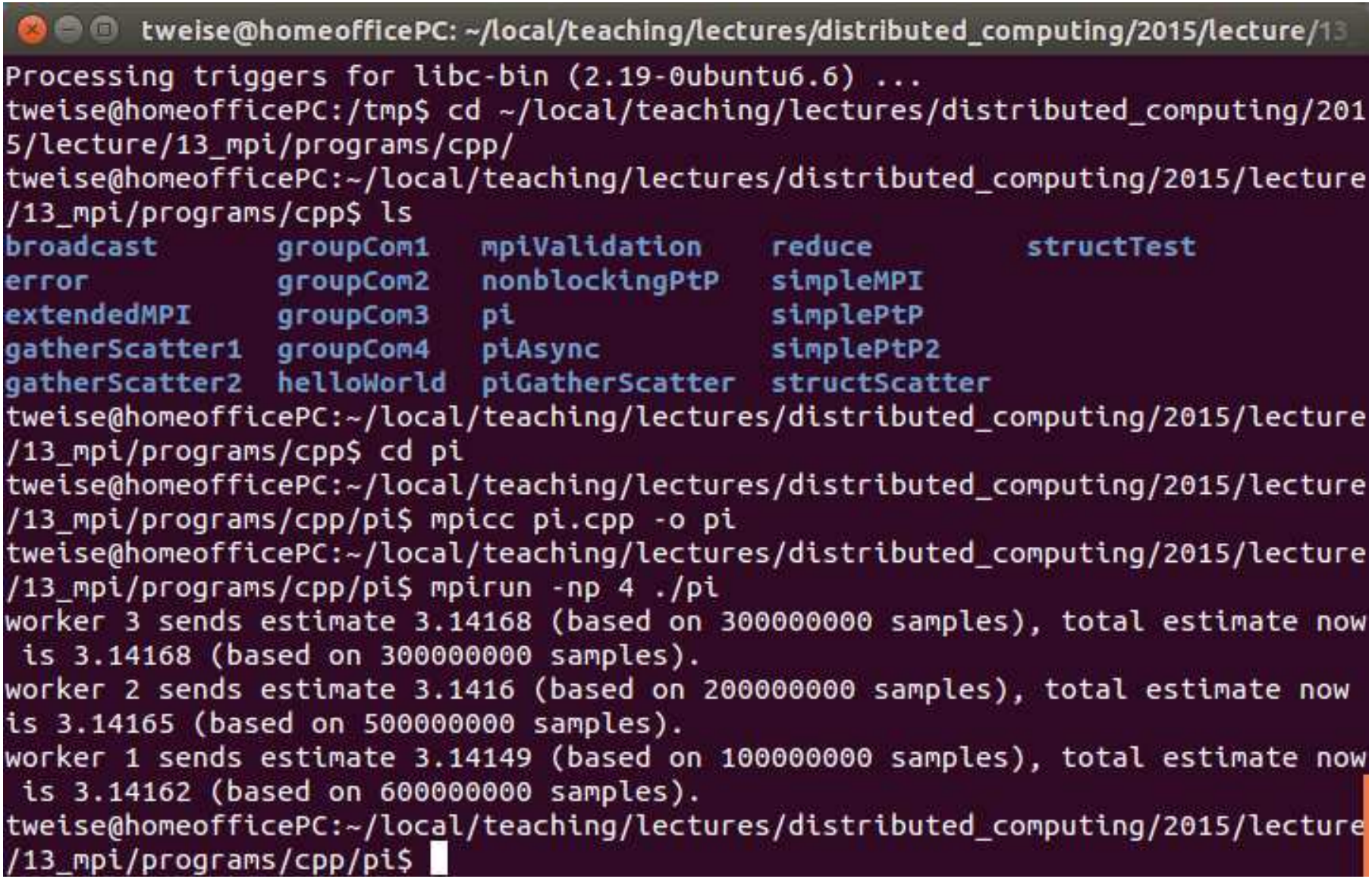
Installation: Linux

- `sudo apt-get install mpich libmpich-dev`
- `mpicc myprogram.cpp -o myprogram`
- `mpirun -np 4 ./myprogram`

Installation: Linux



Installation: Linux



谢谢

Thank you

Thomas Weise [汤卫思]
tweise@hfuu.edu.cn
http://www.it-weise.de

Hefei University, South Campus 2
Institute of Applied Optimization
Shushan District, Hefei, Anhui,
China

Caspar David Friedrich, “Der Wanderer über dem Nebelmeer”, 1818
http://en.wikipedia.org/wiki/Wanderer_above_the_Sea_of_Fog

Bibliography

Bibliography I

1. Caron Carlson. How noaa handles 80 tb of data a day. *FierceCIO – The Executive IT Management Briefing*, April 6, 2011. URL http://www.fiercecio.com/story/how-noaa-handles-80-tb-data-day/2011-04-06.
2. David A. Robinson, David C. Bader, Donald M. Burgess, Kenneth E. Eis, Sara J. Graves, Ernest G. Hildner III, Kenneth E. Kunkel, Mark A. Parsons, Mohan K. Ramamurthy, Deborah K. Smith, John R. G. Townshend, Paul D. Try, Steven J. Worley, Xubin Zeng, and Ian Kraucunas. Environmental data management at noaa: Archiving, stewardship, and access. 2007. URL http://dels.nas.edu/resources/static-assets/materials-based-on-reports/reports-in-brief/data_at_noaa_final.pdf.
3. Pu Wang, Thomas Weise, and Raymond Chiong. Novel evolutionary algorithms for supervised classification problems: An experimental study. *Evolutionary Intelligence*, 4(1):3–16, January 12, 2011. doi: 10.1007/s12065-010-0047-7. URL http://www.it-weise.de/documents/files/WWC2011NEAFSCPAES.pdf.
4. Thomas Weise and Raymond Chiong. Evolutionary data mining approaches for rule-based and tree-based classifiers. In Fuchun Sun, Yingxu Wang, Jianhua Lu, Bo Zhang, Witold Kinsner, and Lotfi A. Zadeh, editors, *Proceedings of the 9th IEEE International Conference on Cognitive Informatics (ICCI'10)*, pages 696–703, Beijing, China: Tsinghua University, 2010. Los Alamitos, CA, USA: IEEE Computer Society Press. doi: 10.1109/COGINF.2010.5599821. URL http://www.it-weise.de/documents/files/WC2010EDMAFRBATBC.pdf.
5. Thomas Weise, Raymond Chiong, and Ke Tang. Evolutionary optimization: Pitfalls and booby traps. *Journal of Computer Science and Technology (JCST)*, 27(5):907–936, September 2012. doi: 10.1007/s11390-012-1274-4. URL http://www.it-weise.de/documents/files/WCT2012EOPABT.pdf. Special Issue on Evolutionary Computation, edited by Xin Yao and Pietro S. Oliveto.
6. Thomas Weise. *Global Optimization Algorithms – Theory and Application*. Germany: it-weise.de (self-published), 2009. URL http://www.it-weise.de/projects/book.pdf.
7. Thomas Weise, Michael Zapf, Raymond Chiong, and Antonio Jesús Nebro Urbaneja. Why is optimization difficult? In Raymond Chiong, editor, *Nature-Inspired Algorithms for Optimisation*, volume 193/2009 of *Studies in Computational Intelligence*, chapter 1, pages 1–50. Berlin/Heidelberg: Springer-Verlag, 2009. doi: 10.1007/978-3-642-00267-0_1. URL http://www.it-weise.de/documents/files/WZCN2009WIOD.pdf.
8. Ke Tang, Xiaodong Li, Ponnuthurai Nagaratnam Suganthan, Zhenyu Yang, and Thomas Weise. Benchmark functions for the cec'2010 special session and competition on large-scale global optimization. Technical report, Hefei, Anhui, China: University of Science and Technology of China (USTC), School of Computer Science and Technology, Nature Inspired Computation and Applications Laboratory (NICAL), January 8, 2010. URL http://www.it-weise.de/documents/files/TLSYW2009BFFTCSSACOLSGO.pdf.

Bibliography II

9. Top500 statistics, March 2012. URL http://i.top500.org/stats.
10. Uncle sam shocks intel with a ban on xeon supercomputers in china. April 7, 2015. URL http://www.vrworld.com/2015/04/07/usa-shocks-intel-ban-on-china-xeon-supercomputers/.
11. U.s. department of energy selects intel to deliver nation's most powerful supercomputer at argonne national laboratory, April 9, 2015. URL http://newsroom.intel.com/community/intel_newsroom/blog/2015/04/09/us-department-of-energy-selects-intel-to-deliver-nations-most-powerful-supercomputer-at-argonne-national-
12. Jimmy Lin. *Cloud Computing Lecture – #1 What is Cloud Computing? (and an intro to parallel/distributed processing)*. College Park, MD, USA: University of Maryland, The iSchool, September 3, 2008. URL http://www.umiacs.umd.edu/~jimmylin/cloud-2008-Fall/Session1.ppt.
13. Winfried Kalfa. *Betriebssysteme*, volume 24 of *Informatik, Kybernetik, Rechentechnik*. Berlin, Germany: Akademie Verlag, 1988. ISBN 3055004779 and 9783055004773. URL http://books.google.de/books?id=Pm8mAAAACAAJ.
14. Gene M. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *American Federation of Information Processing Societies: Proceedings of the Spring Joint Computer Conference (AFIPS)*, pages 483–485, Atlantic City, NJ, USA, 1967. New York, NY, USA: Association for Computing Machinery (ACM) and London, New York: Academic Press. doi: 10.1145/1465482.1465560. URL http://www-inst.eecs.berkeley.edu/~n252/paper/Amdahl.pdf.
15. Günter Rudolph. Deployment scenarios of parallelized code in stochastic optimization. In Bogdan Filipič and Jurij Šilc, editors, *Proceedings of the Second International Conference on Bioinspired Optimization Methods and their Applications (BIOMA'06)*, Informacijska Družba (Information Society), pages 3–11, Ljubljana, Slovenia: Jožef Stefan International Postgraduate School, 2006. Ljubljana, Slovenia: Jožef Stefan Institute. URL http://bioma.ijs.si/conference/ProcBIOMA2006.pdf.
16. Thomas Weise and Kurt Geihs. Dgpf – an adaptable framework for distributed multi-objective search algorithms applied to the genetic programming of sensor networks. In Bogdan Filipič and Jurij Šilc, editors, *Proceedings of the Second International Conference on Bioinspired Optimization Methods and their Applications (BIOMA'06)*, Informacijska Družba (Information Society), pages 157–166, Ljubljana, Slovenia: Jožef Stefan International Postgraduate School, 2006. Ljubljana, Slovenia: Jožef Stefan Institute. URL http://www.it-weise.de/documents/files/WG2006DGPFc.pdf.
17. Enrique Alba Torres. *Parallel Metaheuristics: A New Class of Algorithms*, volume 47 of *Wiley Series on Parallel and Distributed Computing*. New York, NY, USA: John Wiley & Sons Ltd., 2005. ISBN 0471678066 and 9780471678069. URL http://books.google.de/books?id=Tjt4V_81MBIC.

Bibliography III

18. Dimitri Perrin. *Message Passing Interface (MPI)*. Dublin, Ireland: Dublin City University (DCU), School of Computing, November 2007. URL http://www.computing.dcu.ie/~mcrane/CA463/MPI%20lecture%20%282007-08%29%20-%20part%201.pdf.
19. Blaise Barney. Introduction to parallel computing, February 15, 2012. URL https://computing.llnl.gov/tutorials/parallel_comp/.
20. Leonardo Pisano Bigollo. *Liber Abaci (Book of Calculation)*. Rome, Italy, 1202.
21. George Karniadakis and Robert M. Kirby. *Parallel Scientific Computing in C++ and MPI: A Seamless Approach to Parallel Algorithms and their Implementation*. Cambridge, UK: Cambridge University Press, 2003. ISBN 0521817544 and 9780521817547. URL http://books.google.de/books?id=EnqgXI1AXAQC.
22. William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI – Portable Parallel Programming with the Message-Passing Interface*. Scientific and Engineering Computation. Cambridge, MA, USA: MIT Press, 2 edition, 1999. ISBN 0262571323 and 9780262571326. URL http://books.google.de/books?id=xpBZ0RyRb-oC.
23. William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2 – Advanced features of the Message-Passing Interface*. Scientific and Engineering Computation. Cambridge, MA, USA: MIT Press, 1999. URL http://books.google.de/books?id=X6rSjwEACAAJ.
24. Peter S. Pacheco. *Parallel Programming with MPI*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. ISBN 1558603395 and 9781558603394. URL http://books.google.de/books?id=GufgfWSHt28C.
25. USA: Message Passing Interface Forum Urbana, IL. *MPI: A Message-Passing Interface Standard*. Knoxville, TN, USA: University of Tennessee, version 2.2 edition, September 4, 2009. URL http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf.
26. Message passing interface forum (a complete issue on the mpi standard). *The International Journal of High Performance Computing Applications*, 8(3–4):167–414, September 1994. doi: 10.1177/109434209400800301.
27. G. Al Geist, Adam Beguelin, Jack J. Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy S. Sunderam. *PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing*. Cambridge, MA, USA: MIT Press, 1994. ISBN 0-262-57108-0. URL http://www.netlib.org/pvm3/book/pvm-book.html.
28. Jack J. Dongarra, G. Al Geist, Robert Manchek, and Vaidy S. Sunderam. Integrated pvm framework supports heterogeneous network computing. *Computers in Physics*, 7(2):166–175, April 1993. URL http://www.netlib.org/utk/papers/comp-phy7/comp-phy7.html.
29. *MPICH2*. Argonne, IL, USA: Argonne National Laboratory, Mathematics and Computer Science Division, 2012. URL http://www.mcs.anl.gov/research/projects/mpich2/.

Bibliography IV

30. Frederic P. Miller, Agnes F. Vandome, and McBrewster John. *MPICH*. Saarbrücken, Saarland, Germany: VDM Verlag Dr. Müller AG und Co. KG, 2010. ISBN 6130811276 and 9786130811273. URL http://books.google.de/books?id=hrBvcAAACAAJ.
31. *LAM/MPI Parallel Computing*. Bloomington, IN, USA: Indiana University, Bloomington Campus, 1996. URL http://www.lam-mpi.org/.
32. *Open MPI: Open Source High Performance Computing*. Bloomington, IN, USA: Indiana University, Bloomington Campus, 2004. URL http://www.open-mpi.org/.
33. David J. Ashton. *DeinoMPI: The Great and Terrible Implementation of MPI-2*. Sandy, UT, USA: Deino Software, 2009. URL http://mpi.deino.net/.
34. *MPI.NET: High-Performance C# Library for Message Passing*. Bloomington, IN, USA: Indiana University, Bloomington Campus, 2004. URL http://osl.iu.edu/research/mpi.net/.
35. *pyMPI: Putting the py in MPI*. Fairfax, VA, USA: SourceForge, 2004. URL http://pympi.sourceforge.net/.
36. *MinGW – Minimalist GNU for Windows*. Fairfax, VA, USA: SourceForge, 2012. URL http://www.mingw.org/.
37. Othmar Korn. Mpi under mingw – basic instructions for compiling mpi programs in windows using the free (as in speech, not beer) mingw (minimalist gnu for windows) c/c++ compiler, August 19, 2004. URL http://www.dehne.carleton.ca/teaching/programming-resources/mpi/mpi-under-mingw.