EM Graduate course: Practical Introduction to Parallel Programming

[1]Dynaflow Research Group

[2]Delft Institute of Applied Mathematics

Erik Jan Lingen[1] (erikjan.lingen@dynaflow.com)

Matthias Möller[2] (m.moller@tudelft.nl)

March 20-24, 2023

Version March 16, 2023

Part I

Introduction to parallel programming

Outline

Parallel computers

A parallel computer is a set of processors that are able to work cooperatively to solve a computational problem.

Parallel computers provide expandable processing power, memory capacity and I/O bandwidth.

Shared vs distributed

Parallel computers can be roughly divided into two classes, based on their memory architecture:

- shared-memory computers;
- distributed-memory computers.

Shared-memory computers

Processors share the same memory.

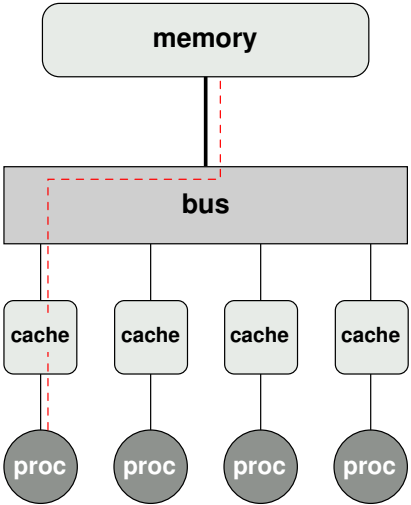
Each processor can access all memory addresses.

Inter-process communication via memory.

Memory access time increases with number of processors.

Bus-based design limits the maximum number of processors.

Shared-memory computers



Distributed-memory computers

Each processor has its own local memory.

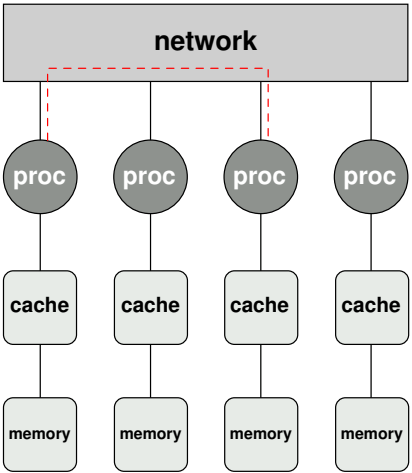
A processor cannot access the memory of another processor.

Inter-process communication by sending packets over the network.

Communication performance measured in terms of *latency* and *bandwidth* (more about this later).

Number of processors can be very large.

Distributed-memory computers



Virtually shared

Each processor has its own local memory.

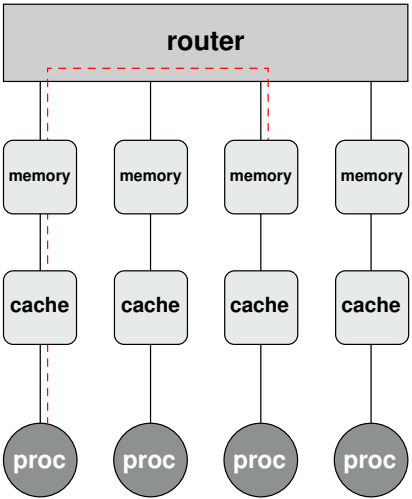
A processor can directly access the memory of another processor through a routing network.

Identical to a shared-memory computer from a logical point of view.

Memory access time depends on the distance between the processor and the memory module: Non-Uniform Memory Access (NUMA).

Number of processors can be large.

Virtually shared



Hybrid: shared & distributed

Most current parallel computers have a hybrid memory architecture, combining shared and distributed memory.

The hybrid memory architecture is a logical consequence of combining multiple processor *cores* in a single CPU.

A parallel computer with a hybrid architecture is composed of a collection of *nodes*, each containing a few CPUs that share memory.

The nodes are connected to each other by a network.

Hybrid: shared & distributed

Hybrid parallel computers enable the use of hybrid programming models to achieve optimal performance.

Hybrid computers typically result in a non-regular performance profile. That is, the runtime of a parallel program tends to be an irregular function of the number of processor cores that are used.

The runtime will change differently when an additional core within a node is used than when another core in another node is used.

Accelerators

Some types of computations can be performed (very) much faster on special hardware. Many large parallel computers therefore include so-called *accelerators* that can be viewed as co-processors.

A common accelerator is a graphics processor or GPU. Modern GPUs are very efficient when processing large sequences of similar operations on different data.

More flexible accelerators are field-programmable gate arrays or FPGAs. These can be viewed as processors with a programmable instruction set.

Accelerators

It is possible, and sometimes useful, to run only part of a program on an accelerator.

However, the interaction between the CPU and the accelerator can become a bottleneck. There is therefore a trend to run complete programs on the accelerator only.

A drawback of using accelerators is that it requires a program to be written specifically for an accelerator.

Several attempts to create uniform programming models for accelerators are being worked on.

Terminology

Modern parallel computers are composed of different units on different levels.

For instance, a parallel computer may be composed of multiple cabinets, containing multiple nodes, containig multiple CPUs, containing multiple processor cores.

A processor core itself may even be composed of multiple compute units named hyper threads or something similar.

In the remainder of this presentation we will use the term *processor* to indicate the basic parallel compute unit.

Depending on the hardware being used, a processor could be a processor core, a hyper thread or maybe a compute core in a GPU.

Examples of parallel computers

Parallel computers used to be exotic and quite rare.

One of the first well-known parallel computers is the Cray X-MP from 1983. It contained up to four CPUs with vector processors in a shared memory configuration.

It has the same processing power as a modern phone.

Examples of parallel computers

Nowadays, parallel computers can be found anywhere; your mobile phone most probably has more than four cores and a powerful GPU.

High-end phones even have special accelerators for executing AI algorithms.

Typical notebook or desktop

A typical notebook or desktop computer has from four to 32 processing cores connected to shared memory.

On a macro level it is a shared-memory machine, but on a micro level it is a virtually shared-memory machine because of the hierarchical cache memory structure.

It often contains one or more accelerators in the form of GPUs.

Supercomputers

The top-end supercomputers are all distributed-memory machines with several million computing cores.

Many supercomputers combine general-purpose processors with some kind of accelerator often in the form of GPUs.

Example: the Summit contains about 2.4 million processing cores and 28 thousand accelerators. It can perform about 144 Peta flops.

The INGInious HPC server

`https://inginious-hpc.ewi.tudelft.nl`

The INGInious high-performance computing server runs on a shared-memory machine with one AMD EPYC 7662 64-core processor and 128GB total memory.

The current configuration allows to run up to 4 student submissions in parallel with up to 64 processes and 32GB memory consumption per job. We advise you to test your parallel programs with 1-16 processes/threads and request more processes/threads only for well-tested programs to test the scalability of your implementation.

Submissions that take longer than 90 seconds to execute are stopped by the system.

Performance characteristics

The overall performance of a parallel computer depends on:

- the speed of the processors/accelerators;
- the speed of the memory system;
- the speed of the network.

The network is only relevant for distributed-memory machines.

Speed of the processors

Obviously, faster and/or more processors increase the performance of a parallel computer.

Heat dissipation and power consumption pose practical limits on the number of processors and their speed.

The speed and number of the processors should match the speed of the memory system and the network.

For instance, if the processors are too fast, they will spend a lot of time waiting. This is one of the biggest problems in practice.

Speed of the memory system

The memory system must be fast enough to supply the processors with the required data at a rate that those data can be processed.

The more processors, the faster the memory system must be; the total *memory bandwidth* must scale with the number of processors.

In a distributed-memory machine this scaling is more or less automatic as each node comes with its own memory modules.

More cores on a node require a larger memory bandwidth per node.

Speed of the network

Except for the most trivial parallel programs, data need to be exchanged between processors.

In a distributed-memory machine this means that data must be exchanged over the network connecting the nodes.

The slower the network, the longer a processor must wait for the data it needs.

As the number of processors increases, more data typically needs to be exchanged over the network; the *network bandwidth* must scale with the number of processors.

Shared vs distributed machines

In shared-memory machines all data exchanges implicitly go through the shared memory. The speed of the memory system is therefore a crucial factor in the overall performance.

In distributed-memory machines the performance of the network is the most important factor affecting the overall performance.

In the ideal case every processor can send data directly to every other processor. However, this requires p^2 network connections which is not scalable.

Most distributed memory machines therefore involve a network with a grid or tree topology.

Parallel speedup

The *parallel speedup* is an important measure of the performance of a parallel program.

The speedup S is defined as:

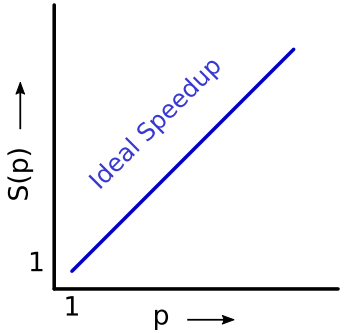
$$S(p) = \frac{T(1)}{T(p)}$$

with $T(p)$ the execution time of the program on p processors and $T(1)$ the execution time on one processor.

Parallel speedup

The speedup tells how well the program can take advantage of an increasing number of processors.

In the ideal case the speedup is equal to p. That is, doubling the number of processors will halve the execution time of the program.



For most parallel programs the speedup will be less than ideal because of the finite speed of the memory system and the network.

Parallel speedup

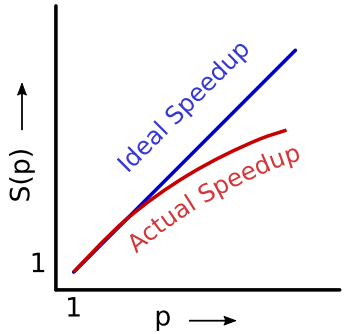
Note that it is more fair to take $T(1)$ as the execution time of the most efficient serial implementation of the program.

This is not necessarily equal to the execution time of the parallel implementation of the program on one processor because an efficient parallel algorithm is not always an efficient serial algorithm.

As it might not be trivial to implement the most efficient serial algorithm, one often takes $T(1)$ as the execution time of the parallel implementation on one processor.

Parallel speedup

In practice, the parallel speedup tends to increase less than p because an increasing amount of time is required to coordinate the execution of multiple processors.



Typical causes of a lower-than-ideal speedup are: load imbalance, communication overhead and serial sections.

Load imbalance

Load imbalance arises when the total amount of work is not divided evenly over the processors.

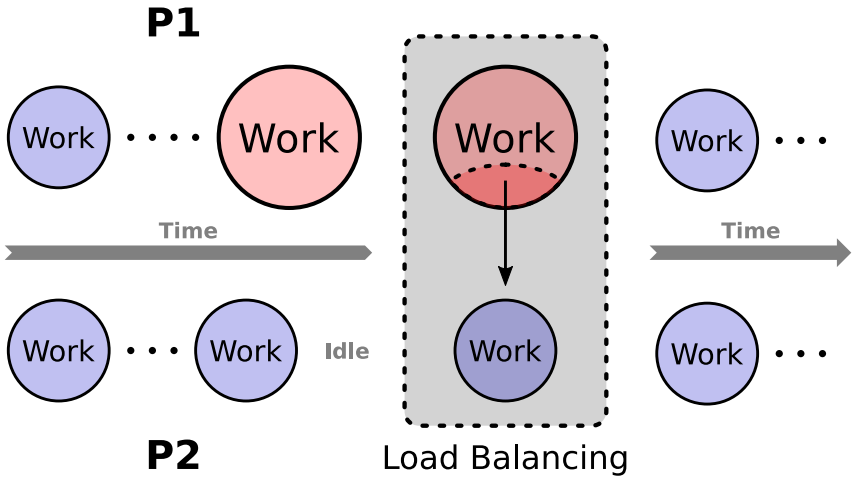
That means that some processors have more work to do than others; the ones with less work have to wait for the ones with more work.

Avoiding load imbalance is not too difficult if the workload per processor is known in advance.

It becomes more difficult when the workload varies dynamically in an unpredictable way. In this case it might be necessary to periodically re-distribute the work over the processors.

This is called *dynamic load balancing*.

Load balancing



Communication overhead

In any non-trivial parallel program, processors need to exchange data.

Typically, sections of independent computations are interleaved by sections of inter-processor data exchanges in which processors send and receive packets of data.

Sending/receiving a data packet involves time during which no useful work can be done. This so-called *communication overhead* is not present in a serial program.

Communication overhead

The time T_x that is required to transmit a package of size n is approximately given by:

$$T_x(n) = T_0 + \frac{n}{B}$$

where T_0 is the *latency* and B the *bandwidth* of the network or memory system.

The latency is the time required to start a data exchange while the bandwidth is equal to the number of data units (bytes/bits) that can be transmitted in each time unit (second).

Latency

The latency is determined by signal propagation speeds and overhead in one or more software layers (OS, libraries, applications).

The latency often depends on the load of the system. The more processors are using the network or memory system, the higher the latency due to collisions.

If a physical wire or data lane is in use by one processor, another processor must wait.

If the latency is high, then good parallel speedup can only be obtained by keeping the number of data exchanges low.

If possible, one should aggregate multiple smaller data exchanges into one larger exchange.

Bandwidth

The bandwidth depends on the number of data lanes, their physical nature (light-based or electron-based), and the clock frequency of the network or memory system.

Ideally, the total bandwidth of the system scales linearly with the number of processors. If that is not the case, then it will be more difficult to achieve a good parallel speedup.

If the bandwidth is low, one should try to reduce the total data exchange volume. Sometimes this is possible by repeating the same computations on multiple processors.

Serial sections and Amdahl's law

It might not be feasible to distribute an entire computation over multiple processors; some parts of the computation are performed on a single processor only.

These parts are called *serial sections*, and their combined, relative size impose an upper limit on the speedup that can be obtained.

Suppose that the serial runtime $T(1)$ of a parallel program is given by:

$$T(1) = T_s + T_p$$

where T_s is the time spent in the serial sections, and T_p is the time spent in the parallel sections.

Amdahl's law

In the ideal case, the parallel execution time is then given by:

$$T(p) = T_s + \frac{T_p}{p}$$

Thus, the best possible speedup is given by:

$$S(p) = \frac{1+f}{1+\frac{f}{p}} \qquad f = \frac{T_p}{T_s}$$

This is called *Amdahl's law*.

Amdahl's law

Amdahl's law imposes an upper limit on the speedup:

$$S_{\text{max}} = 1 + f = 1 + \frac{T_p}{T_s}$$

For instance, if 10% of the execution time is spent in serial sections, then the program will be at most 11 times faster on a parallel computer.

To make a program truly scalable it may not have any serial sections.

Exercise: parallel speedup

Analyze the theoretical parallel performance of Conway's game of life.

Conway's game of life describes the evolution of a $m \times n$ grid in which each cell has an associated value that is either zero or one.

The value of a cell in the next time step depends on the values of the neighboring cells in the current time step.

The neighbors of a cell (i, j) are the cells $(i \pm 1, j \pm 1)$.

The initial state of the grid completely determines how the grid evolves in time.

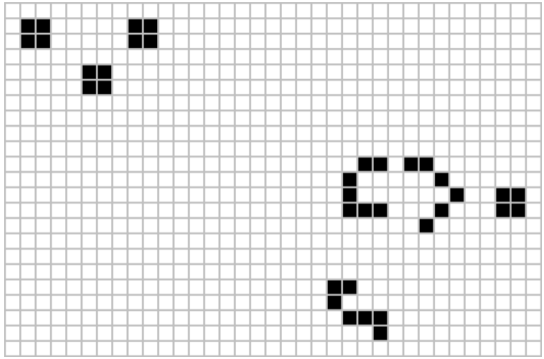
Transition rules

The value of a cell transitions according to the following rules:

1. a non-zero cell becomes zero if it has less than two non-zero neighbors;
2. a non-zero cell remains non zero if it has two or three non-zero neighbors;
3. a non-zero cell becomes zero if it has more than three non-zero neighbors;
4. a zero cell becomes non zero if it has exactly three non-zero neighbors.

Pseudo implementation

```
int  a[M][N], b[M][N];
int  i, j, k;

/* Initialize the matrix a ... */

while ( true )
{
  for ( i = 0; i < M; i++ )
  {
    for ( j = 0; j < N; j++ )
    {
      k = sum ( a[i±1][j±] );

      if ( a[i][j] )
        if  ( k < 2 || k > 3 )
          b[i][j] = 0;
        else
          b[i][j] = 1;
      else if ( k == 3 )
        b[i][j] = 1;
      else
        b[i][j] = a[i][j];
    }
  }
  a = b;
}
```



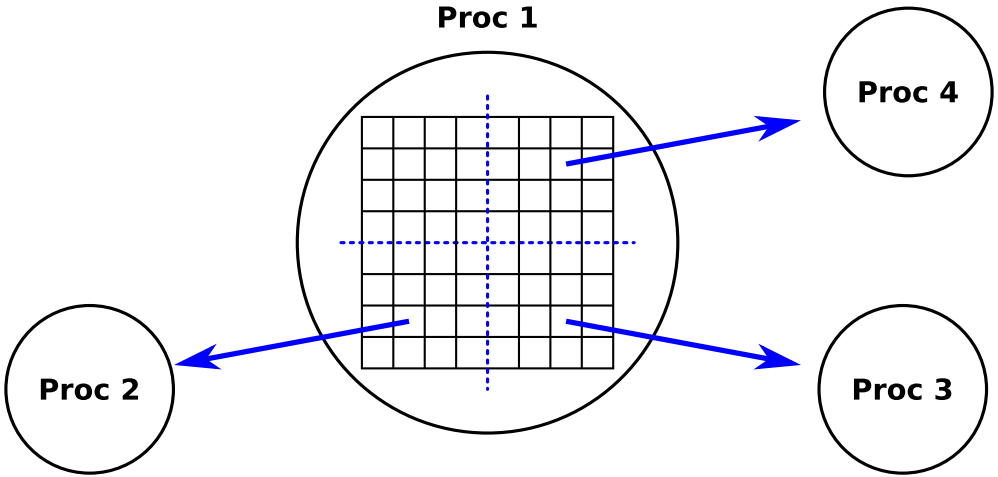
Parallel implementation

To execute game of life in parallel, one processor initializes the grid and partitions it into p rectangular sub-grids.

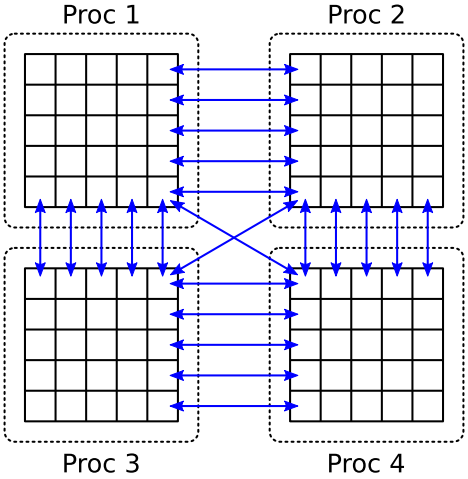
It then sends those sub-grids to all processors, including itself. After that, each processor handles the evolution of its own sub-grid.

In each time step the processors exchange the values at the edges of their sub-grids with their neighboring processors.

Initialization phase



Computation phase



Performance analysis

Derive an expression for the parallel speed up of the program.

Assumptions:

- the network has latency T_0 and bandwidth B;
- one cell requires time T_i to initialize;
- one cell requires time T_e to process;
- the total number of time steps equals k.

Parallel programming models

A programming model provides an abstract view of a computer, typically involving a central processing unit, memory, a program counter and storage.

Using a programming model avoids becoming entangled in nitty gritty details like the exact layout of the CPU and its connections.

It also makes a program portable across different hardware platforms.

The programming model is often implicitly part of the programming language being used.

Parallel programming models

A parallel programming model essentially describes how multiple processors can be coordinated.

Such a model enables one to describe a parallel program on a high level without having to explicitly manage the execution of multiple processors and their interactions.

The three most commonly used programming models are the:

- *message-passing programming model*;
- *shared-memory programming model*;
- *data-parallel programming model*.

Message-passing model

In the message-passing programming model, a program consists of concurrent *tasks*, each one encapsulating its own sequential code and data.

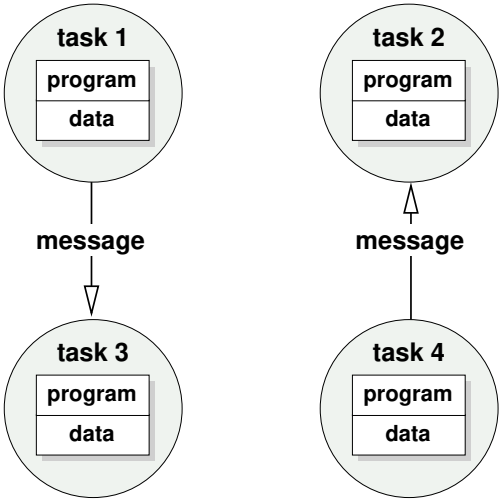
Tasks are identified by a unique number or *rank*.

Tasks interact by sending and receiving data packets or *messages*.

Programmers must explicitly describe which tasks communicate with each other and when they communicate with each other.

The message-passing programming model is supported by all types of parallel computers, including shared-memory computers.

Message-passing model



Shared-memory model

In the shared-memory programming model, a program also consists of concurrent tasks that execute their own sequential code.

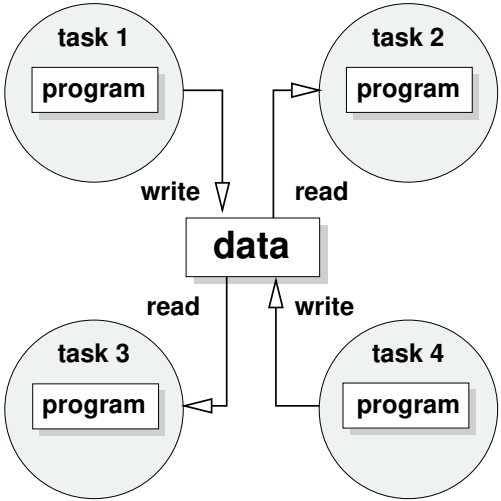
In contrast to the message passing model, tasks share a common memory address space.

Tasks can read/write data from/to memory asynchronously. Access to shared data is controlled by mutual exclusion primitives.

The programmer does not need to explicitly specify task interactions.

The shared-memory programming model is supported by shared-memory computers, but only by a few distributed-memory computers.

Shared-memory model



Data-parallel model

In the data-parallel programming model, a program consists of a single task that executes the same instructions on different data in parallel.

That is, the program consists of one stream of instructions that are applied to different parts of the data set.

In general, it is not possible to apply different instructions to different parts of the data set concurrently.

The processor and the runtime environment control how the operations are scheduled and how the data are divided between different processing units.

The data-parallel model is often used when programming for accelerators or GPUs.

SIMD/SPMD/MPMD

A parallel programming model is loosely coupled with a parallel execution model that describes how a parallel program is actually executed.

Common execution models are:

SIMD Single Instruction, Multiple Data;

SPMD Single Program, Multiple Data;

MPMD Multiple Programs, Multiple Data;

SIMD

The SIMD execution model is typically associated with data-parallel programs.

That is, all processors execute the same instructions that are applied to different data. The processors operate synchronously; at any given time, all processors execute the same instruction.

Modern processor cores support SIMD operations at the machine code level. These operations, sometimes called vector operations, can be used to process multiple words at the same time.

For instance, a vector addition can be used to sum multiple (floating point) numbers at the same time.

SPMD

The SPMD execution model involves a single program that is executed by multiple processors concurrently. Each processor processes a different data set.

The processors operate asynchronously in that they can execute different instructions at any given time. This means that one processor can be executing a different part of the program than another processor.

An SPMD program typically contains points at which the execution of the processors is synchronized.

MPMD

The MPMD execution model involves multiple programs that are executed by multiple processors concurrently.

A classic example is the manager-worker pattern in which one processor assumes a manager role and the other processors assume a worker role. The manager processor typically farms out work to the other processors and collects the results afterwards.

The manager processor executes a different program than the worker processors. The worker processors execute the same program for different data sets.

The MPMD execution model is less common than the SPMD model as the former is less scalable.

Exercise: performance analysis

Analyze the real-world performance of the parallel implementation of the game of life (based on MPI).

Create a new account at

`https://inginious-hpc.ewi.tudelft.nl`

and activate it by clicking on the link that you will receive via the activation email.

Sign in and register for the course

'`Practical Introduction to Parallel Programming`'

The password is '`PIPP2023`'.

Go to course overview and click on *[Introduction] Demo 1 : Game of Life*.

Command-line arguments

The example program can be configured through the following command-line arguments:

`--dims` *m*x*n* the dimensions of the grid;

`--steps` *k* the number of time steps;

`--debug` *l* the debug level (0 : none, 1 : info, 2 : all).

First run a small problem instance on a single processor with debug output, e.g., `"--dims","15x15","--steps","3","--debug","2"`.

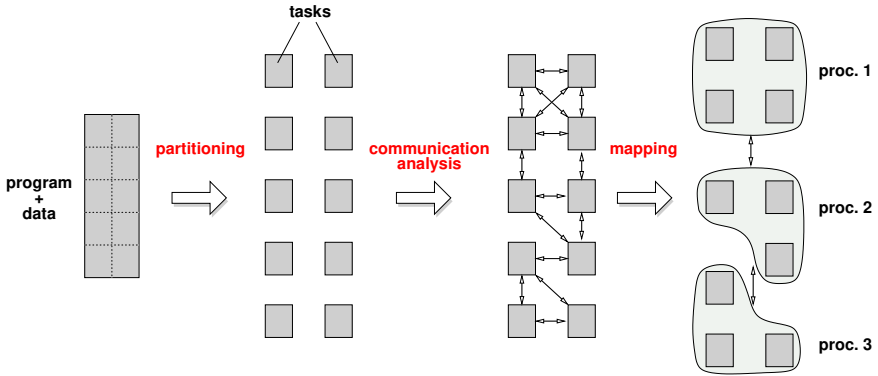
Next, disable debug output and select a larger problem size that you run on 1, 2, 4, 8, 16, 32, and 64 cores, e.g., `"--dims","1000x1000","--steps","200","--debug","0"`.

Parallel program design

The design of a parallel program can be divided into three stages:

1. *partitioning*: the computation and the data are decomposed into concurrent *tasks*.
2. *communication analysis*: the interactions between the tasks are identified.
3. *mapping*: the tasks are assigned to the available processors.

Parallel program design



Partitioning

During the partitioning stage the computations and the associated data are decomposed into concurrent tasks.

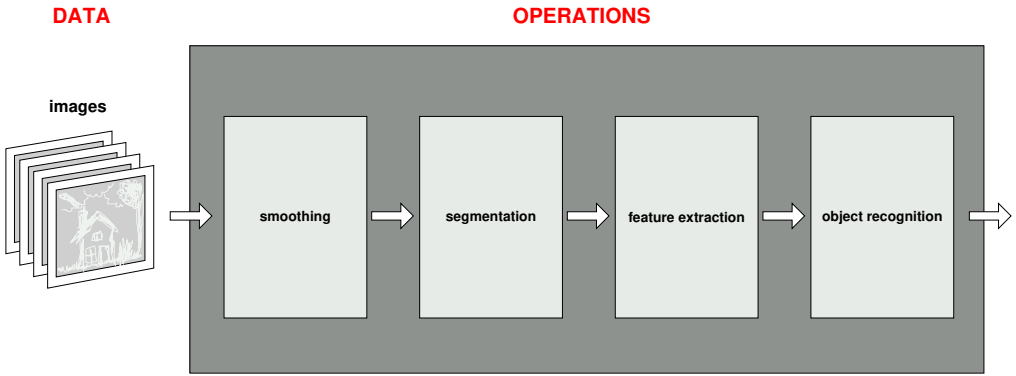
The goal is to maximize the number of tasks in order to increase the parallel scalability of the program.

Two common partitioning techniques:

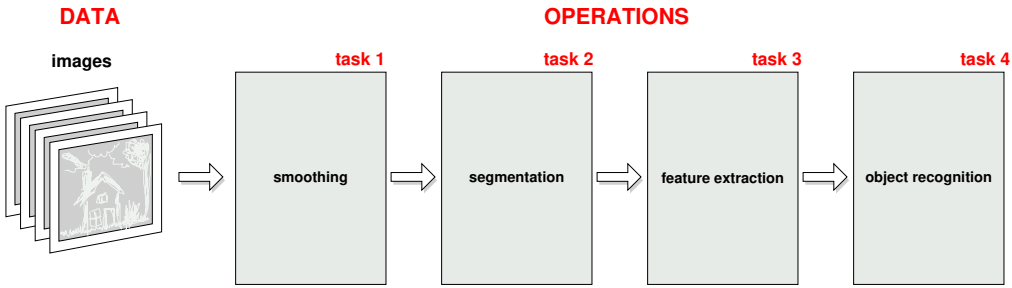
- *functional decomposition*: the computation is partitioned into groups of logically related operations; each group is assigned to a task.
- *domain decomposition*: the data are partitioned into multiple units. Each unit is assigned to a task.

Partitioning example

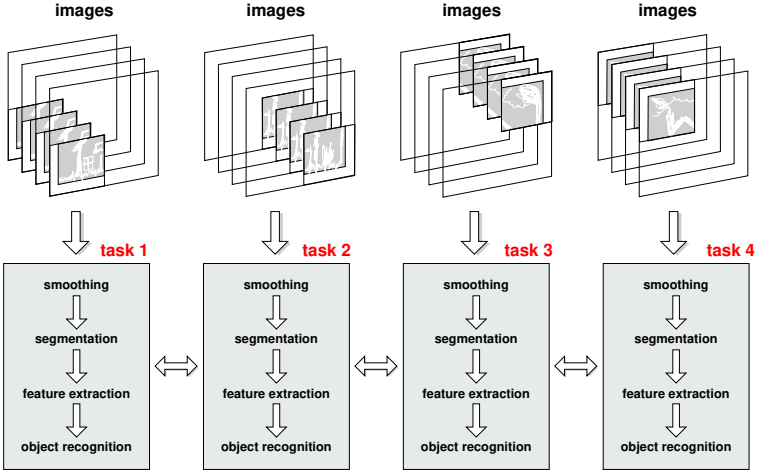
Example: transformation of a series of images.



Functional decomposition



Domain decomposition



Partitioning checklist

- *Does your partition define more tasks than processors?* If not, you have little flexibility in subsequent design stages.
- *Does your partition avoid redundant computation and storage requirements?* If not, the resulting algorithm may not be scalable to deal with large problems.
- *Are tasks of comparable size?* If not, it will be harder to allocate each processor equal amounts of work.
- *Does the number of tasks scale with problem size?* If not, your parallel program may not be able to solve larger problems when more processors are available.

Communication analysis

Tasks are intended to execute concurrently but cannot, in general, execute independently; the computation to be performed in one task will typically require data associated with another task.

The goal of the communication analysis stage is to identify the communication patterns between the tasks and define the communication procedures and data structures.

Communication patterns

Common communication patterns:

Structured Communication patterns form a tree or grid.

Unstructured Communication patterns form an arbitrary graph.

Static Tasks always communicate with the same neighbor tasks.

Dynamic Neighbor tasks may vary during the computation.

Unstructured communication patterns complicate the mapping stage as sophisticated algorithms may be required to determine a mapping strategy that balances the workload between processors and minimizes communication requirements

Communication checklist

- *Do all tasks perform about the same number of communication operations?* Unbalanced communication requirements suggest a non-scalable design.
- *Does each task communicate only with a small number of neighbors?* If not, communication overhead may limit the scalability of the program.
- *Are communication operations able to proceed concurrently?* If not, your algorithm is likely to be inefficient and non-scalable.
- *Do the communication patterns contain circular dependencies?* If so, the program may get stuck in a deadlock situation.

Mapping

The goal of the mapping stage is to map tasks to the available processors in such a way that each processor requires approximately the same time to complete its tasks, and that the communication overhead is as small as possible

Three mapping strategies:

- place tasks that are able to execute concurrently on different processors to increase parallelism;
- place tasks that are able to execute concurrently on the same processor to hide latencies;
- place tasks that communicate frequently with each other on the same processor or near to each other.

Mapping considerations

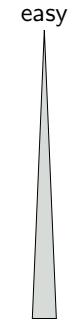
Clearly, the three strategies will sometimes conflict, in which case our design will involve trade-offs.

In addition, resource limitations may restrict the number of tasks that can be placed on a single processor.

Increasing the task granularity decreases communication costs because of a decreasing communication size and a decreasing number of messages.

This can be done by reducing the number of tasks and by replicating computation.

Mapping problem classification



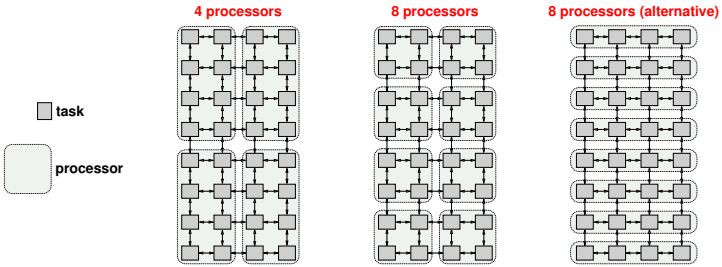
Structured data with structured communication
An efficient mapping is usually straightforward.

Unstructured data with unstructured communication
An efficient mapping strategy may not be obvious. Use static load balancing algorithms.

Dynamic computation and communication
The number of tasks or the amount of computation or communication per task changes dynamically. Static mapping does not suffice; use dynamic load balancing algorithms.

Mapping example

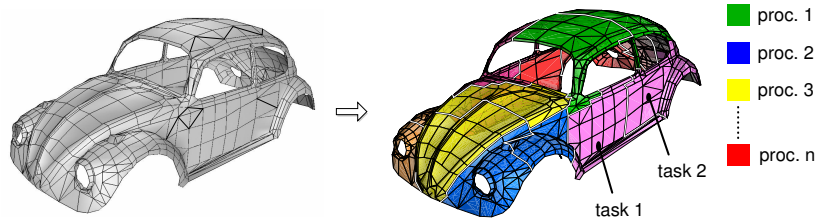
Structured data with structured communication.



Map tasks at the beginning of the calculation in a way that minimizes interprocessor communication.

Mapping example

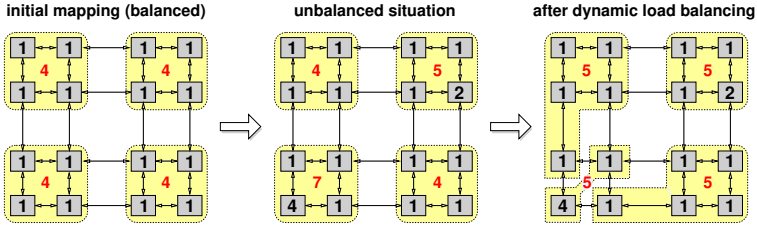
Unstructured data with unstructured communication.



Map tasks at the beginning of the calculation using *static load balancing* algorithms.

Mapping example

Dynamic computation and communication.



Use *dynamic load balancing* algorithms to periodically re-map the tasks to the processors.

Parallel program design recipe

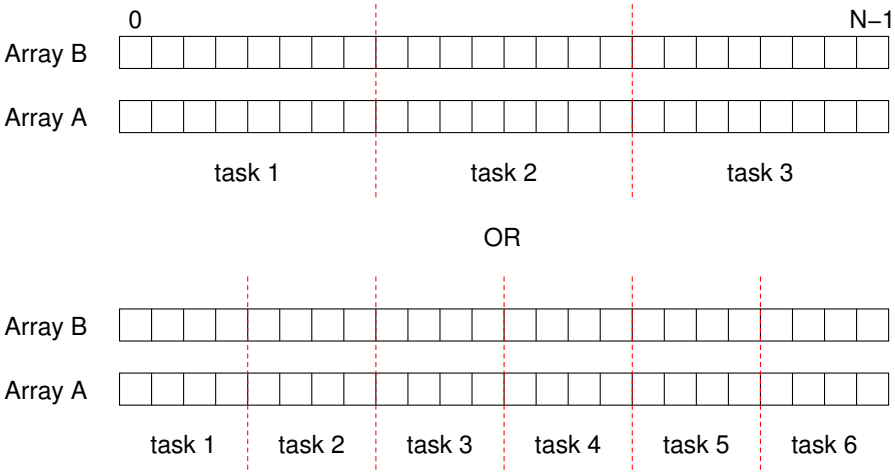
1. Select a suitable programming model such as the message-passing model or the shared-memory model.
2. Partition the computation into tasks using functional decomposition or domain decomposition.
3. Identify the communication patterns.
4. Map the tasks to the available processors.

Parallel program design example

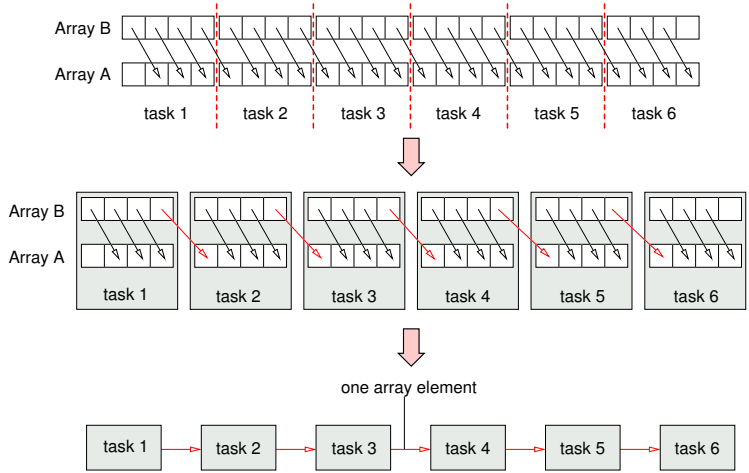
Sequential algorithm: array shift.

```
for ( i = 1; i < N; i++ )
{
  a[i] = b[i - 1];
}
```

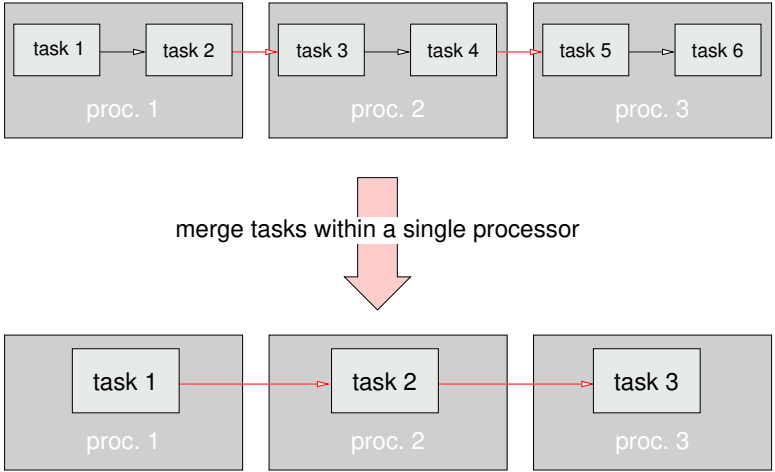
Partitioning: domain decomposition



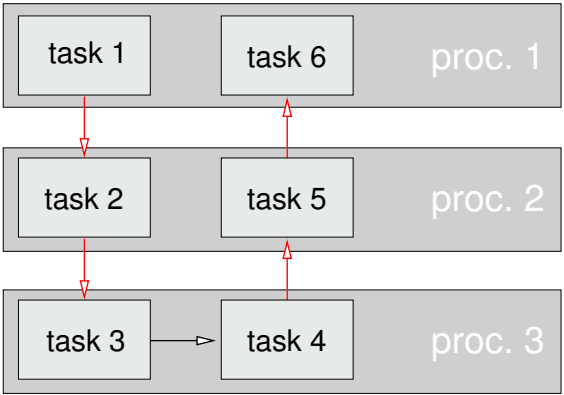
Communication analysis



Mapping


task 1
task 2
task 3
task 4
task 5
task 6
proc. 1
proc. 2
proc. 3
merge tasks within a single processor
task 1
task 2
task 3
proc. 1
proc. 2
proc. 3

Mapping (alternative)



Bad idea!

Parallel program on three processors

Processor 1

```
Send b[7] to right neighbor

for ( i = 1; i < 8; i++ )
{
  a[i] = b[i-1];
}
```

Processor 2

```
Receive b[7] from left neighbor
Send b[15] to right neighbor

for ( i = 8; i < 16; i++ )
{
  a[i] = b[i-1];
}
```

Processor 3

```
Receive b[15] from left neighbor

for ( i = 16; i < 24; i++ )
{
  a[i] = b[i-1];
}
```

Loop within each processor is actually from 0 to 8 (except in processor 1, which starts at 1).

Actual parallel program

Processor i

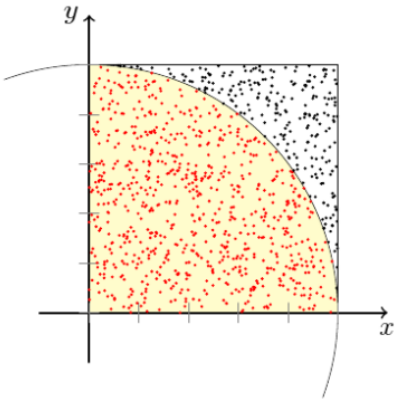
```
size = N / num_procs;

Receive b[-1] from left neighbor
Send b[size-1] to right neighbor

for ( i = 0; i < size; i++ )
{
  a[i] = b[i-1];
}
```

Note: the first and last processors perform dummy receive and send operation, respectively.

Exercise: parallel computation of π

The area of a circle of radius R is πR^2, the area of a square with side length $2R$ is $4R^2$. If the circle is inscribed inside the square then the ratio of both areas is $\pi/4$.

Monte Carlo method: if we pick N points at random inside the square and count the number of points that are inside the circle (M) then we can compute the approximation $\pi \approx 4M/N$.



Open '[Introduction] Exercise 2 : Compute PI' and rewrite the sequential algorithm as a parallel algorithm using the functions provided by the minimpi.h file

Exercise: parallel computation of π

Functions in *minimpi.h*

```
/* Initialization of parallel communication */
void init()

/* Finalization of parallel communication */
void finalize()

/* Get the total number of processors involved */
int get_total_proc_number()

/* Get the number of the current processor */
int get_my_proc_number()
```

Exercise: parallel computation of π

Functions in *minimpi.h*

```
/* Send an integer to processor dest */
void send_int ( int value, int dest );
```

`value`	IN	Integer to be sent.
`dest`	IN	Processor number to which the integer is sent.

The destination processor must call `recv_int`.

Exercise: parallel computation of π

Functions in *minimpi.h*

```
/* Receive an integer from processor source */
void recv_int ( int* value, int source );
```

`value`	OUT	Pointer to the integer to be received.
`source`	IN	Processor from which the message is received.

The source processor must call `send_int`.

Exercise: parallel computation of π

Send and receive operations must match:

```
int  i;

if ( get_my_proc_number() == 0 )
{
  recv_int ( &i, 1 );
}
else
{
  send_int ( i, 0 );
}
```

Part II

MPI

Outline

*M*essage *P*assing *I*nterface

MPI is a formal standard defining the *interface* of a message passing library that can be used to implement parallel programs based on the message passing programming model.

In other words, MPI is a collection of functions (procedures) and data types that can be used to exchange data between processes.

MPI hides the actual hardware details involved in inter-process communication from the programmer.

MPI enables portability on the source code level.

That is, MPI programs can be ported to different kinds of parallel computers without having to modify the source code; the code only needs to be re-compiled.

In some cases re-compilation is not even necessary.

Before MPI was created, each vendor of parallel computers (including SGI, Cray, IBM) had its own message passing library. Parallel programs were generally not portable between different parallel computers.

What is included in MPI?

MPI includes functions for:

- point-to-point communication;
- collective communication;
- remote memory access;
- communication context management;
- (dynamic) process management;
- parallel I/O.

Topics explained in this course

The complete MPI standard is much too extensive to be covered here completely. This course is therefore limited to:

- point-to-point communication;
- collective communication;
- communication context management.

These topics cover a wide range of parallel programs. The other topics are relevant mostly for more specialized parallel programs.

Language bindings

MPI provides formal language bindings for C and Fortran 90. Here we will use the C bindings exclusively.

Previous versions of MPI also provided C++ bindings. However, these have been deprecated in MPI version 2.2 and have been removed in version 3.0.

Note that informal bindings exist for other programming languages, including C++ (through Boost) and Python.

MPI versions

The MPI standard is determined by the Message Passing Interface Forum in which many academic and private organizations participate.

Version 1.0 of the MPI standard was released in 1994 and covered point-to-point communication, collective communication and context management.

Version 2.0 was released in 1997 and added dynamic process management, remote memory access and parallel I/O.

MPI versions

Version 3.0 was released in 2012 and essentially provides extensions to the operations defined in the earlier standards. This version also removed the C++ bindings.

Version 3.1 was released in 2015 and contains only minor improvements.

Version 4.0 was released in June 2021 and adds better support for hybrid programming models, 'big counts', events, fault-tolerant computing, among other improvements.

Backward compatibility

The MPI versions are backward compatible as far as the C bindings are concerned.

That is, a program using the C bindings of MPI 1 can be compiled with MPI 2 and MPI 3.

Note that this is only valid for source-code compatibility. Different MPI versions are generally not binary compatible.

MPI implementations

MPI only defines a programming interface; you need an actual implementation in order to compile and run an MPI-based program.

There are about a dozen different MPI implementations available, many of which are aimed at specific hardware.

Two widely used, and generic, implementations are MPICH and OpenMPI. Both are open source which is useful when running into a bug related to a new MPI feature.

MPICH

MPICH is (one of) the oldest MPI implementations. In fact, it was created to test concepts that later became part of the first MPI standard.

It is maintained by the Argonne National Laboratory in collaboration with universities and private companies, including Intel, IBM and Microsoft.

It supports single, multi-core workstations and clusters running Linux and OS X. It implements version 3 of the MPI standard.

It is used as the basis of special MPI implementations targeting specific platforms (including Windows) and parallel computers.

It is licensed under a BSD-like license (use it however you want).

OpenMPI

OpenMPI is younger than MPICH and came into being by the merger of four other MPI implementations.

It is maintained by a consortium of academic and private institutions, some of which also participate in MPICH.

OpenMPI supports more or less the same systems as MPICH (Linux and OS X clusters), although there are some differences in the supported network types.

OpenMPI tends to be a bit behind MPICH in supporting the latest MPI features.

It is also licensed under a BSD-like license.

Which implementation to use?

On a single, multi-core workstation both MPICH and OpenMPI are fine and work more or less the same.

On a (small) cluster the choice between MPICH and OpenMPI depends on the type of the network. It is always a good idea to run a couple of benchmarks to test which implementation is better for you.

Large-scale parallel computers typically come with MPI implementations (possibly based on MPICH or OpenMPI) that have been tuned for the particular architecture on which the computers are based. Using the system-provided MPI implementation typically results in the best performance.

Processes

An MPI program consists of multiple, concurrent processes, executing their own program. Typically all processes execute the same program.

A process has its own, private memory address space and communicates with other processes through calls to MPI functions. A process can not directly access the memory of another process.

The processes are identified by an integer number ranging from 0 to the number of processes minus one. This number is called the *rank* of the process.

Process management

The MPI standard is mainly concerned with the communication between processes. It does not specify how the initial processes are created and how they are bound to processors.

An MPI implementation typically provides an external program for starting an MPI program with a given number of processors. This program is often named `mpirun`.

Here is an example that shows how to start an MPI program with eight processes:

```
mpirun -np 8 program
```

Note that many MPI implementations allow one to start more processes than there are processors available.

Local and non-local functions

The functions defined by the MPI standard can be divided into local and non-local functions.

A functions is called *local* if its completion depends only on the executing process and not on the state of other processes.

A function is called *non local* if its completion may require the execution of some MPI function on another process. That is, a non-local function may involve communication with another process.

In the following slides the labels `[local]` and `[nonlocal]` will be used to indicate whether an MPI function is local or not.

Handles

MPI manages system memory that is used for storing internal representations of various MPI objects. These objects are *opaque*; their exact definitions are hidden from an MPI program.

MPI provides indirect access to these objects through *handles*.

For instance, when creating a persistent communication request (more about this later), MPI returns a handle associated with that request. This handle can be used to perform operations involving the request.

In C the handles are essentially pointers. Different types are provided for handles associated with different object categories.

Handles support the assignment and (in)equality operators.

C language bindings

The C language bindings are defined in the header file `<mpi.h>`. This file must be included by source files using MPI functions and/or data types.

The names of all MPI entities have the prefix `MPI_`.

Names of functions start with an upper case character, followed by lower case characters. The same convention is used for data type names. For instance: `MPI_Init`.

Names of constant values consist of all upper case characters and underscores. Example: `MPI_DOUBLE`.

C language bindings

An MPI program should not use the `MPI_` prefix for the names of its own entities. This is to avoid name collisions when a new MPI version is released.

Most MPI functions return an integer error code that indicates whether an error occurred during the execution of the function.

When no error has occurred, the constant value `MPI_SUCCESS` is returned. Otherwise, an implementation-dependent error code is returned.

Function parameters

Some MPI functions declare parameters of type `void*` to indicate that they can be called with arguments of different types. Example:

```
MPI_Send ( const void* buffer, ... );

void example ()
{
  int    ibuf[4];
  float  fbuf[4];

  MPI_Send ( ibuf, ... );
  MPI_Send ( fbuf, ... );
}
```

Function parameters

In the following slides the parameters of MPI functions will be marked as follows:

IN – the function only uses the parameter
OUT – the function updates the parameter
INOUT – the function both uses and updates the parameter

Parameters marked as OUT and INOUT are declared as pointers.

Parameters marked as IN can be declared as const-qualified pointers (if they represent arrays) or as non-pointer types.

Basics of an MPI program

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

int main ( int argc, char** argv )
{
  /* No MPI calls allowed yet. */

  MPI_Init      ( &argc, &argv );
  printf        ( "Hello there!\n" );
  MPI_Finalize  ();

  /* No MPI calls allowed anymore. */

  return 0;
}
```

Initialization

```
int MPI_Init ( int* argc, char*** argv );  [nonlocal]
```

argc	INOUT	Pointer to number of command-line arguments, or NULL.
argv	INOUT	Pointer to command-line arguments, or NULL.

The function MPI_Init initializes the internal state of the MPI library. It must be called at the beginning of an MPI program to avoid problems with process creation. It may only be called once.

The arguments passed to MPI_Init are typically pointers to the parameters of the main function.

No MPI function may be called before MPI_Init, with one exception.

Initialization

```
int MPI_Initialized ( int* flag );  [local]
```

`flag`	OUT	Pointer to an integer that will be set to a non-zero value if MPI has been initialized. Otherwise, it will be set to zero.

The function `MPI_Initialized` can be used to check whether MPI has been initialized. It is mainly useful for developers of libraries using MPI.

`MPI_Initialized` is the only function that may be called before `MPI_Init`.

Finalization

```
int MPI_Finalize ( void );  [nonlocal]
```

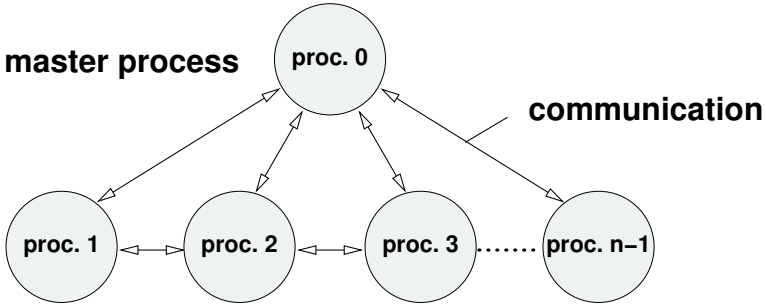
The function `MPI_Finalize` must be called at the end of an MPI program. It makes sure that all outstanding communication operations are completed properly.

If you forget to call `MPI_Finalize`, your program will most probably not finish properly.

No MPI function (except `MPI_Initialized`) may be called after `MPI_Finalize` has been called.

Differentiation between processes

Suppose that we want a communication scheme like this:



Differentiation between processes

This requires that:

- the master process can identify itself;
- the master process knows how many other processes there are;
- the other processes know their rank (process number) so that they can identify their neighboring processes (process 2 has neighbors 1 and 3).

The next slide shows an MPI program that implements this communication scheme.

Differentiation between processes

```
#include <mpi.h>

int main ( int argc, char** argv )
{
  int  my_rank, proc_count;

  MPI_Init      ( &argc, &argv )
  MPI_Comm_size ( MPI_COMM_WORLD, &proc_count );
  MPI_Comm_rank ( MPI_COMM_WORLD, &my_rank );

  if ( my_rank == 0 )
    /* Communicate with processes 1, 2, ... proc_count - 1. */
  else
    /* Communicate with processes 0, my_rank - 1 and my_rank + 1. */

  MPI_Finalize ();

  return 0;
}
```

Getting the current process rank

```
int MPI_Comm_rank ( MPI_Comm comm, int* rank ); [local]
```

`comm`	IN	Handle associated with a communicator.
`rank`	OUT	Pointer to an integer that will be set to the rank of the current process within the given communicator.

The function `MPI_Comm_rank` returns the rank of the current process within a given *communicator*.

A communicator can be viewed as a kind of virtual parallel computer that comprises a number of processes plus a communication network. More about this later.

An MPI program starts with the `MPI_COMM_WORLD` communicator that contains all (initial) processes.

Getting the number of processes

```
int MPI_Comm_size ( MPI_Comm comm, int* size ); [local]
```

`comm`	IN	Handle associated with a communicator.
`size`	OUT	Pointer to an integer that will be set to the number of processes within the given communicator.

The function `MPI_Comm_size` returns the number of processes within a given communicator.

When called with the `MPI_COMM_WORLD` communicator, this function returns the total number of (initial) processes.

Exercise: hello world

Implement a small MPI program that prints the total number of processes and the rank of the current process.

A skeleton implementation of the program is given in task *[MPI] Exercise 1: Hello World*.

If you want to run your program on your local computer outside the web-based INGInious environment use `make` to build the executable program and `mpirun` to start the executable:

```
mpirun -np N 01-hello-world-mpi
```

Communication

MPI provides a collection of functions for exchanging data between processes.

These functions can be divided into two groups:

1. *point-to-point* communication functions;
2. *collective* communication functions.

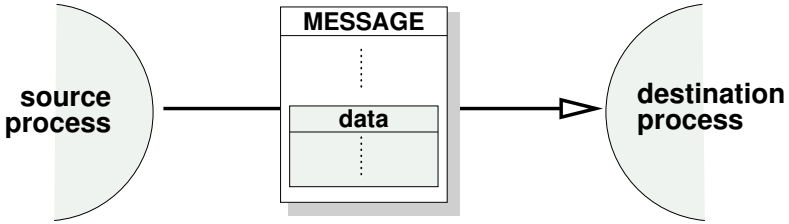
The first group deals with exchanging data between pairs of processes.

The second deals with exchanging data between multiple processes.

Point-to-point communication

Point-to-point communication involves pairs of processes. One process sends data and the other process receives those data.

The sending process is called the *source* and the receiving process is called the *destination*. The data to be sent/received are referred to as the *message*.



Point-to-point example

```
#include <mpi.h>

int main ( int argc, char** argv )
{
  MPI_Status  status;
  int         my_rank, value = 1;

  MPI_Init      ( &argc, &argv );
  MPI_Comm_rank ( MPI_COMM_WORLD, &my_rank );

  if ( my_rank == 0 )
    MPI_Send ( &value, 1, MPI_INT, 1, 0, MPI_COMM_WORLD );
  else
    MPI_Recv ( &value, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );

  MPI_Finalize ();
  return 0;
}
```

Point-to-point example

The example program only works when started with two processes.

Note that a send operation on the source process must be matched by a receive operation on the destination process.

A program will “hang” if this is not the case.

In other words, any pair of send/receive operations must be balanced: for each send there must be a matching receive in the program.

Sending a message

```
int MPI_Send ( const void* buf, int count, MPI_Datatype type,
               int dest, int tag, MPI_Comm comm );  [nonlocal]
```

`buf`	IN	A pointer to the data to be sent.
`count`	IN	The number of data elements to be sent.
`type`	IN	The type of the data elements.
`dest`	IN	The rank of the destination process.
`tag`	IN	The tag associated with the message.
`comm`	IN	The communicator to be used.

The function `MPI_Send` sends zero or more data elements to the specified destination process.

The destination process must execute a matching receive operation.

Sending a message

The `buf` parameter can point to a single value or an array containing multiple values. The number of values, called elements, is specified by the `count` parameter.

The `type` parameter specifies the type of the elements to be sent. It can be one of the following constants:

MPI Constant	C Datatype
MPI_CHAR	char
MPI_SHORT	short
MPI_INT	int
MPI_LONG	long
MPI_FLOAT	float
MPI_DOUBLE	double

Sending a message

The MPI type that is specified must match the actual type of the data elements to be sent.
The compiler does not check this!

```
void example ()
{
  int  buffer[3];

  MPI_Send ( buffer, 3, MPI_DOUBLE, ... );  /* Oops, wrong type. */
}
```

It is possible to define new MPI data types but that is not explained here.

Sending a message

The `tag` parameter of the `MPI_Send` function can be used to associate an user-defined identifier (an arbitrary non-zero integer) with a message.

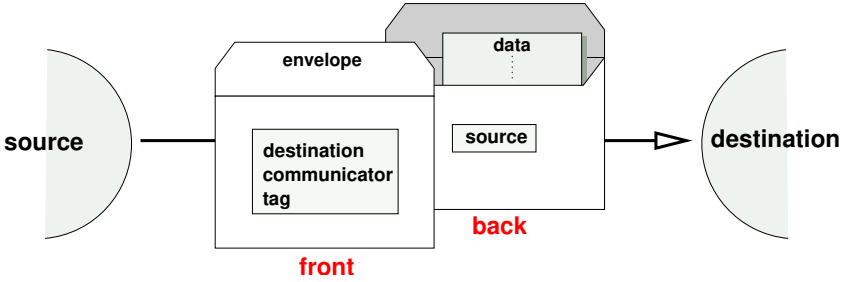
The destination process can use the tag to receive messages in a particular order that is not necessarily the same as the order in which the messages were sent.

You do not need to use the tag; simply set it to zero (or some other value) in all send and receive operations if you want to receive messages in the order that they were sent.

More about message tags in later slides.

Message envelope

MPI extracts the data from the buffer and creates an *envelope* that provides the communication system and the receiver with the following information:



The envelope and the data together form the message.

Message envelope

An MPI message envelope can be compared to an envelope containing a (paper) letter:

- the *source* on the back of the envelope is like the return address of the letter;
- the *destination* is like the street number of the adressee;
- the *communicator* is equivalent with the street and city name;
- the *tag* is equivalent with a logo on or color of the envelope containing the letter.

Receiving a message

```
int MPI_Recv ( void* buf, int count, MPI_Datatype type,
               int source, int tag,
               MPI_Comm comm, MPI_Status* stat );  [nonlocal]
```

`buf`	OUT	A pointer to an array or value in which the data are to be received.
`count`	IN	The size of `buf` (number of elements).
`type`	IN	The type of the data elements.
`source`	IN	The rank of the source process.
`tag`	IN	The tag associated with the message.
`comm`	IN	The communicator to be used.
`stat`	OUT	A pointer to a status object.

The function `MPI_Recv` receives zero or more data elements from the specified source process. The source process must execute a matching send operation.

Receiving a message

The `buf` parameter can point to a single value or an array. The size of the buffer is specified by the `count` parameter.

Note that the buffer (and also the `count` parameter) may be larger than the actual number of data elements that have been received. This can be useful if the receiving process does not know the exact number of elements to be received beforehand.

The `source` parameter specifies the rank of the source process. It can be set to the special value `MPI_ANY_SOURCE` to indicate that the message may be received from any process.

Receiving a message

The `tag` parameter can be used to indicate that only messages with a matching tag value are to be received; messages with a non-mathcing tag will be ignored.

The special value `MPI_ANY_TAG` can be used as a kind of wildcard value; it will match messages with any tag.

The `stat` parameter must point to a `MPI_Status` object. This opaque object is filled with information from the message envelope:

- the rank of the source process;
- the tag provided by the source process;
- and the number of data elements received.

Status objects

The source rank and the message tag are stored in the fields `MPI_SOURCE` and `MPI_TAG`, respectively, of an `MPI_Status` object.

The function `MPI_Get_count` can be used to determine how many data elements have actually been received. This function only needs to be called if the receiving process does not know the exact number of data elements that have been sent.

If the status object is not used, one may specify the special value `MPI_STATUS_IGNORE` for the `stat` parameter. Example:

```
MPI_Recv ( buf, 4, MPI_INT, 0, 0,
           MPI_COMM_WORLD, MPI_STATUS_IGNORE );
```

Example use of status objects

The following slide shows part of an MPI program that makes use of status objects.

The program implements a client-server model in which one process waits for data from another process, performs some kind of (unspecified) transformation on the data, and sends it back to the source process.

The code fragment shown is executed by the server process.

Example use of status objects

```
void run_server ()
{
  const int   max_len = 1024;
  double      data[max_len];
  int         count;
  MPI_Status  status;

  while ( 1 )
  {
    MPI_Recv      ( data, max_len, MPI_DOUBLE, MPI_ANY_SOURCE,
                    MPI_ANY_TAG,   MPI_COMM_WORLD, &status );
    MPI_Get_count ( &status, MPI_DOUBLE, &count );
    transform     ( data, count );
    MPI_Send      ( data, count, MPI_DOUBLE, status.MPI_SOURCE,
                    status.MPI_TAG, MPI_COMM_WORLD );
  }
}
```

Message ordering

MPI guarantees that two or more messages sent from a single source process to a single destination process arrive in the order that they have been sent.

In other words, messages sent between a pair of processes do not overtake each other.

No such guarantee is provided for messages that have been sent from different source processes to the same destination process; those message can arrive in any order.

Message ordering

Source Process

```
int  dest = 0;
int  tag1 = 1, tag2 = 2;

MPI_Send ( ..., dest, tag1,
           ... );
MPI_Send ( ..., dest, tag2,
           ... );
```

Destination Process

```
int  source = 1;

MPI_Recv ( ..., source, MPI_ANY_TAG,
           ... );
MPI_Recv ( ..., source, MPI_ANY_TAG,
           ... );
```

Here the destination process will first receive the message with `tag1` and then the message with `tag2`.

Message ordering

Source Process 1

```
int  dest = 0, tag = 0;

MPI_Send ( ..., dest, tag, ... );
```

Source Process 2

```
int  dest = 0, tag = 1;

MPI_Send ( ..., dest, tag, ... );
```

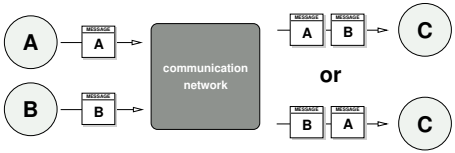
Destination Process

```
MPI_Recv ( ..., MPI_ANY_SOURCE, MPI_ANY_TAG, ... );
MPI_Recv ( ..., MPI_ANY_SOURCE, MPI_ANY_TAG, ... );
```

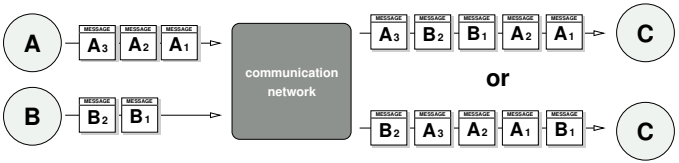
Here the destination process might receive the two messages in any order. In fact, the order may vary from run to run.

Message ordering

Messages from different processes can arrive in any order:



Messages from the same process do not overtake each other:



Blocking communication operations

The function `MPI_Send` will – in principle – not return until the destination process has called `MPI_Recv`, and the other way around.

These functions are said to perform *blocking* communication operations.

A program may end up in a *deadlock* situation if calls to `MPI_Send` and `MPI_Recv` are not ordered correctly.

This is shown in the example program on the next slide. The program is meant to be run with two processes.

Deadlock example

```
int main ( int argc, char** argv )
{
  int  my_rank, other_rank;

  MPI_Init      ( &argc, &argv );
  MPI_Comm_rank ( MPI_COMM_WORLD, &my_rank );

  if ( my_rank == 0 )
    other_rank = 1;
  else
    other_rank = 0;

  MPI_Send ( ..., other_rank, ... );
  MPI_Recv ( ..., other_rank, ... );

  MPI_Finalize ();
  return 0;
}
```

Deadlock example

The problem is that both processes execute the send/recv operations in the same order.

The solution is to swap the send/recv operations on one process so that each send operation is matched by a receive operation on the other process.

This is shown in the modified example on the next slide.

Deadlock-free example

```
if ( my_rank == 0 )
{
  other_rank = 1;

  MPI_Send ( ..., other_rank, ... );
  MPI_Recv ( ..., other_rank, ... );
}
else
{
  other_rank = 0;

  MPI_Recv ( ..., other_rank, ... );
  MPI_Send ( ..., other_rank, ... );
}
```

Exercise: pass data around

Implement an MPI program in which a single integer is passed around all processes.

That is, the process with rank zero assigns the initial value to the integer. It then passes the integer to the process with rank one.

This process, in turn, passes the integer to the process with rank two.

This continues until the process with rank $(p-1)$, with p the number of processes. This last process passes the integer back to the process with rank zero.

Finally, the process with rank zero prints the integer to the terminal and the program ends.

Test the program with one and more processes.

Hints

Use *[MPI] Exercise 2 : Pass Data (blocking point-to-point communication)* as the basis of your implementation.

Use the functions `MPI_Send` and `MPI_Recv` to pass the integer from one process to the next.

Use the function `MPI_Comm_rank` to determine from which process the integer is to be received, and to which process the integer is to be sent.

Use the function `MPI_Comm_size` to determine the total number of processes.

Non-blocking point-to-point communication

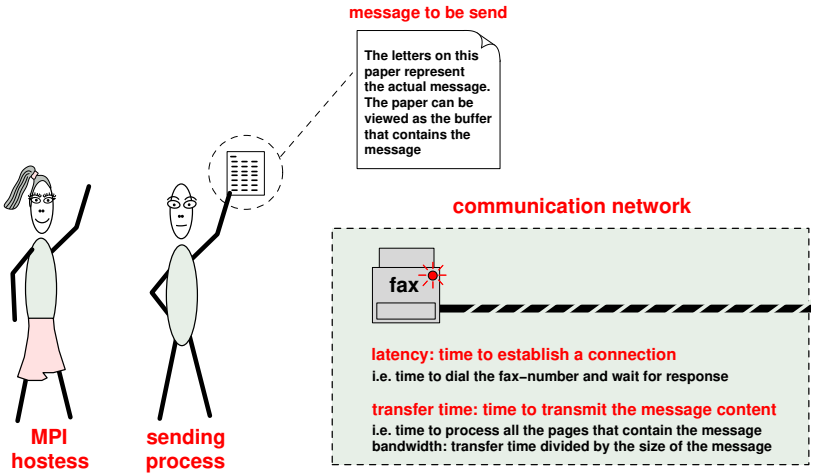
The point-to-point communication functions considered so far are blocking in nature: they will not return until the message exchange has been completed.

Blocking operations are simple to understand but can lead to deadlock.

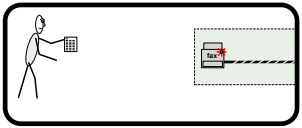
MPI also provides *non-blocking* point-to-point communication procedures that return immediately. You must check later whether the message exchange has been completed.

Non-blocking operations are more difficult to understand but can avoid deadlock and increase performance.

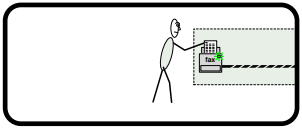
Communication metaphor



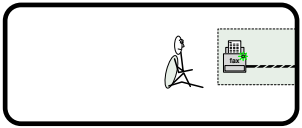
Blocking send illustration



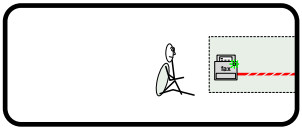
post send

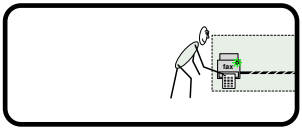


wait until paper is processed

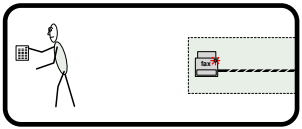


wait until paper is processed



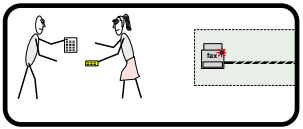


return to work



Happy end

Non-blocking send illustration

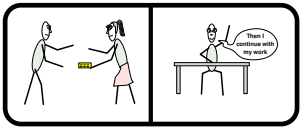


post send, immediate return

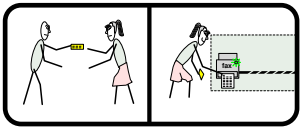
sender can continue with work

ask if ready

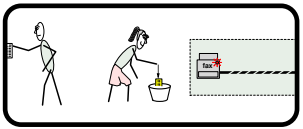
continue with work



ask if ready



retreive paper (buffer)



Happy end

Non-blocking example

```
void nonblocking ()
{
  int          my_rank, other_rank;
  MPI_Request  req;

  MPI_Comm_rank ( MPI_COMM_WORLD, &my_rank );

  if ( my_rank == 0 )
    other_rank = 1;
  else
    other_rank = 0;

  MPI_Isend ( ..., other_rank, ..., &req );
  MPI_Recv  ( ..., other_rank, ... );
  MPI_Wait  ( &req, MPI_STATUS_IGNORE );
}
```

Note: no deadlock because the function `MPI_Isend` returns immediately.

Non-blocking send

```
int MPI_Isend ( const void* buf, int count, MPI_Datatype type,
                int dest, int tag,
                MPI_Comm comm, MPI_Request* req );  [local]
```

`buf`	IN	A pointer to the data to be sent.
`count`	IN	The number of data elements to be sent.
`type`	IN	The type of the data elements.
`dest`	IN	The rank of the destination process.
`tag`	IN	The tag associated with the message.
`comm`	IN	The communicator to be used.
`req`	OUT	A pointer to a request handle.

The function `MPI_Isend` starts a non-blocking send operation. It returns immediately.

The destination process must execute a matching receive operation.

Non-blocking send

The function `MPI_Isend` is similar to the function `MPI_Send`. There are two differences:

- the function `MPI_Isend` does not wait for the data to have been sent;
- the function `MPI_Isend` returns a *request handle* that can be used to check the status of the send operation.

The buffer specified in a call to `MPI_Isend` must exist until MPI has signalled that the operation has been completed. This is the responsibility of the programmer!

Premature free of buffer

```
void example ()
{
  int          count  = 100;
  double*      buffer = malloc ( count * sizeof(double) );
  MPI_Request  req;

  MPI_Isend ( buffer, count, MPI_DOUBLE, 0, 0,
              MPI_COMM_WORLD, &req );

  free ( buffer ); /* Error: buffer still in use by MPI. */

  MPI_Request_wait ( &req, MPI_STATUS_IGNORE );
}
```

Correct free of buffer

```
void example ()
{
  int          count  = 100;
  double*      buffer = malloc ( count * sizeof(double) );
  MPI_Request  req;

  MPI_Isend ( buffer, count, MPI_DOUBLE, 0, 0,
              MPI_COMM_WORLD, &req );

  MPI_Request_wait ( &req, MPI_STATUS_IGNORE );

  free ( buffer ); /* OK: buffer no longer in use. */
}
```

Non-blocking receive

```
int MPI_Irecv ( void* buf, int count, MPI_Datatype type,
                int source, int tag,
                MPI_Comm comm, MPI_Request* req );  [local]
```

`buf`	OUT	A pointer to an array or value in which the data are to be received.
`count`	IN	The size of `buf` (number of elements).
`type`	IN	The type of the data elements.
`source`	IN	The rank of the source process.
`tag`	IN	The tag associated with the message.
`comm`	IN	The communicator to be used.
`req`	OUT	A pointer to a request handle.

The function `MPI_Irecv` starts a non-blocking receive operation. It returns immediately.

The source process must execute a matching send operation.

Non-blocking receive

The function `MPI_Irecv` is similar to the function `MPI_Recv`. Again, there are two differences:

- the function `MPI_Irecv` does not wait for the data to have been received;
- the function `MPI_Irecv` returns a *request handle* that can be used to check the status of the send operation.

The buffer specified in a call to `MPI_Irecv` must exist until MPI has signalled that the operation has been completed.

Matching send/receive operations

A non-blocking send operation executed by a source process must be matched by a blocking or non-blocking receive operation executed by the destination process.

Conversely, a non-blocking receive operation must be matched by a blocking or non-blocking send operation.

If the above conditions are not satisfied, a program will end up in a deadlock situation, even when it uses non-blocking operations.

Non-blocking example 1

Process 0

```
int          other = 1;
MPI_Request  req;

MPI_Isend ( ..., other, ...,
            &req );
MPI_Recv  ( ..., other, ... );
MPI_Wait  ( &req, ... );
```

Process 1

```
int          other = 0;
MPI_Request  req;

MPI_Isend ( ..., other, ...,
            &req );
MPI_Recv  ( ..., other, ... );
MPI_Wait  ( &req, ... );
```

`MPI_Isend` matches `MPI_Recv`; no deadlock.

Non-blocking example 2

Process 0

```
int          other = 1;
MPI_Request  req;

MPI_Irecv ( ..., other, ...,
            &req );
MPI_Send  ( ..., other, ... );
MPI_Wait  ( &req, ... );
```

Process 1

```
int          other = 0;
MPI_Request  req;

MPI_Irecv ( ..., other, ...,
            &req );
MPI_Send  ( ..., other, ... );
MPI_Wait  ( &req, ... );
```

`MPI_Irecv` matches `MPI_Send`; no deadlock.

Non-blocking example 3

Process 0

```
int          other = 1;
MPI_Request  req[2];

MPI_Irecv   ( ..., other, ...,
              &req[0] );
MPI_Isend   ( ..., other, ...,
              &req[1] );
MPI_Waitall ( 2, req, ... );
```

Process 1

```
int          other = 0;
MPI_Request  req[2];

MPI_Irecv   ( ..., other, ...,
              &req[0] );
MPI_Isend   ( ..., other, ...,
              &req[1] );
MPI_Waitall ( 2, req, ... );
```

`MPI_Irecv` matches `MPI_Isend`; no deadlock.

Non-blocking example 4

Process 0

```
int          other = 1;
MPI_Request  req;

MPI_Send  ( ..., other, ... );
MPI_Irecv ( ..., other, ...,
            &req );
MPI_Wait  ( &req, ... );
```

Process 1

```
int          other = 0;
MPI_Request  req;

MPI_Send  ( ..., other, ... );
MPI_Irecv ( ..., other, ...,
            &req );
MPI_Wait  ( &req, ... );
```

Deadlock in `MPI_Send`.

Request handles

The functions `MPI_Isend` and `MPI_Irecv` return a handle to an opaque *communication request* object in their last parameter.

Request objects are allocated and managed by MPI. Programs obtains handles to request objects, not the objects themselves.

A program can query the status of a request object and the corresponding communication operation through various MPI functions; see next slides.

The special value `MPI_REQUEST_NULL` is used to indicate an invalid request handle. MPI functions that deallocate request objects set their handle to this value.

Checking request handles

The following functions can be used to check one or more request handles:

`MPI_Test`	Checks whether one non-blocking operation has completed.
`MPI_Wait`	Waits until one non-blocking operation has completed.
`MPI_Waitall`	Waits until multiple non-blocking operations have completed.

MPI actually provides more, similar functions, but these are not considered here.

Testing a single request

```
int MPI_Test ( MPI_Request *req,
               int *flag, MPI_Status *stat );  [local]
```

`req`	INOUT	Pointer to a request handle.
`flag`	OUT	Pointer to an integer flag that is set to the result of the test.
`stat`	OUT	Pointer to a status object.

The function `MPI_Test` checks whether the given request has completed. If so, the request object is deallocated and the handle pointed to by `req` is set to `MPI_REQUEST_NULL`.

Otherwise the request handle is not modified.

Testing a single request

If the request has completed, then the `flag` parameter is set to a non-zero value; otherwise it is set to zero.

The status object pointed to by the `stat` parameter is filled with information about the message if the request has completed. Otherwise it is untouched.

Example:

```
MPI_Request  req;
int          flag = 0;

MPI_Isend ( ..., &req );

while ( ! flag )
  MPI_Test ( &req, &flag, MPI_STATUS_IGNORE );
```

Waiting for a single request

```
int MPI_Wait ( MPI_Request *req, MPI_Status *stat );  [local]
```

`req`	INOUT	Pointer to a request handle.
`stat`	OUT	Pointer to a status object.

The function `MPI_Wait` waits until the given communication request has completed. It also deallocates the request object and sets the handle pointed to by `req` to `MPI_REQUEST_NULL`.

The status object pointed to by the `stat` parameter is filled with information about the message. This parameter may be set to `MPI_STATUS_IGNORE`.

Waiting for a single request

Example use of `MPI_Wait`:

```
MPI_Request  req;
MPI_Status   stat;

MPI_Irecv ( ..., &req );

/* Do something else ... */

MPI_Wait ( &req, &stat );
```

Waiting for multiple requests

```
int MPI_Waitall ( int count, MPI_Request *req,
                  MPI_Status *stat );  [local]
```

`count`	IN	Number of requests.
`req`	INOUT	Array of request handles.
`stat`	OUT	Array of status objects.

The function `MPI_Waitall` waits until multiple communication requests have completed. It deallocates the associated request objects and sets the handles stored in the `req` array to `MPI_REQUEST_NULL`.

The status objects in the `stat` array are filled with information about the messages. This parameter may be set to `MPI_STATUSES_IGNORE`.

Waiting for multiple requests

Example use of `MPI_Waitall`:

```
MPI_Request  req[2];
int          my_rank;

MPI_Comm_rank ( MPI_COMM_WORLD, &my_rank );
MPI_Irecv     ( ..., my_rank - 1, ..., &req[0] );
MPI_Irecv     ( ..., my_rank + 1, ..., &req[1] );

/* Do something else ... */

MPI_Waitall   ( 2, req, MPI_STATUSES_IGNORE );
```

Performance tip

When exchanging data using non-blocking communication operations, try to execute the receive operations as early as possible.

This reduces the communication overhead as MPI will be able to copy the send data directly into the receive buffer.

Otherwise the send data is first stored in a queue on the receiving process, and is later copied into the receive buffer.

This is illustrated in the next slide.

Performance tip

```
MPI_Request  send_req[2];
MPI_Request  recv_req[2];
double*      send_buf[2];
double*      recv_buf[2];

/* Initialize the buffers ... */

MPI_Irecv   ( recv_buf[0], ..., &recv_req[0] );
MPI_Irecv   ( recv_buf[1], ..., &recv_req[1] );

/* Do computations not involving receive data ... */

MPI_Isend   ( send_buf[0], ..., &send_req[0] );
MPI_Isend   ( send_buf[1], ..., &send_req[1] );

MPI_Waitall ( 2, recv_req, MPI_STATUSES_IGNORE );

/* Use receive data ... */

MPI_Waitall ( 2, send_req, MPI_STATUSES_IGNORE );
```

Exercise: pass data around (2)

Modify the MPI program from previous exercise so that it uses non-blocking communication operations instead of blocking operations. You can use *[MPI] Exercise 3 : Pass Data (non-blocking point-to-point communication)* as a skeleton for your program.

Does the use of non-blocking operations bring any benefits in this case?

Exercise: circular array shift

Use non-blocking point-to-point communication operations to implement a parallel, circular array shift.

Here is the serial implementation of the algorithm:

```
const int  N = 100;
double     a[N];
double     t;
int        i;

t = a[N - 1];

for ( i = N - 1; i >= 1; i-- )
  a[i] = a[i - 1];

a[0] = t;
```

Exercise: circular array shift

In the parallel implementation of the program each process stores a sub-section of the array a. The processes need to exchange data in order to process the first and last element in their sub-array.

[MPI] Exercise 4 : Circular Shift contains an initial implementation of the parallel program. Edit this file to complete the implementation.

Try to start the non-blocking receive operations as soon as possible.

The original array and the shifted array are printed by the program. Try to print the array elements in the correct order. That is, the processes should print their sub-section one after another.

Collective communication

In contrast to point-to-point communication, *collective communication* involves *all* processes associated with a communicator.

Example collective communication operations:

- all processes determine the maximum value of a distributed set of numbers;
- one process sends a message to all other processes;
- all processes wait until a specific point in a program has been reached.

Collective communication

A collective communication function must be called by all processes in a communicator. A program will end up in a deadlock situation if one or more processes do not participate in a collective communication operation.

Example:

```
int  my_rank;

MPI_Comm_rank ( MPI_COMM_WORLD, &my_rank );
MPI_Barrier   ( MPI_COMM_WORLD );   /* OK */

if ( my_rank != 0 )
  MPI_Barrier ( MPI_COMM_WORLD );   /* Deadlock */
```

More about `MPI_Barrier` later.

Collective communication

A collective communication function does not have a message tag parameter; collective communication function calls are matched by the order in which they are executed.

In particular, point-to-point communication operations do not interfere with collective communication operations.

Example:

```
int          other_rank = ...;
MPI_Request  req;

MPI_Isend   ( ..., other_rank, ..., &req );
MPI_Barrier ( MPI_COMM_WORLD );
MPI_Wait    ( &req, MPI_STATUSES_IGNORE );
```

Collective communication categories

	Synchronization
Barrier	Synchronize all processes in a communicator.
	Global reduction
Reduce	Combine a distributed set of numbers into a single number using different types of operations (max, min, sum, etc.).
	Data transfer
Broadcast	Send data from one process to all other processes.
Gather	Collect data from all processes on one process.
Scatter	Send different data from one process to all other processes.

Synchronization

```
int MPI_Barrier ( MPI_Comm comm );  [nonlocal]
```

comm	IN	The communicator associated with the processes to by synchronized.

The function `MPI_Barrier` waits until all processes in the given communicator have called this function.

This function can be a handy debugging aid as it makes it easier to understand the flow of execution in a parallel program.

Most (well-designed) parallel programs do not need to call this function.

Global reduction

The global reduction functions in MPI combine a distributed data set into a single number using various types of reduction operations, such as min, max and sum.

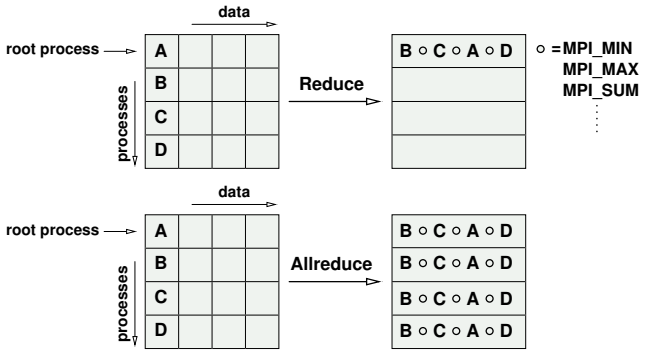
There are two reduction functions: `MPI_Reduce` and `MPI_Allreduce`.

The main difference between these two is that the first only makes the result available on a single process. The second makes the result available on all processes.

`MPI_Allreduce` is similar to `MPI_Reduce` followed by a broadcast (more about that later).

Global reduction

Both `MPI_Reduce` and `MPI_Allreduce` can apply a reduction operation on multiple values stored in an array; the reduction operation is applied element-wise across all processes.



Global reduction example

The following example shows how to compute the maximum value across all processes. The maximum value is only made available on the process with rank `root`.

```
const int  root  = 0;
double     value = ...;
double     result;
int        my_rank;

MPI_Comm_rank ( MPI_COMM_WORLD, &my_rank );

MPI_Reduce ( &value, &result, 1, MPI_DOUBLE, MPI_MAX, root,
             MPI_COMM_WORLD );

if ( my_rank == root )
  printf ( "The maximum is: %g.\n", result );
```

Global reduction example

Here is the same program (simplified) executed with two processes.

Process 0

```
const int  root  = 0;
double     value = 1.0;
double     result;

MPI_Reduce ( &value, &result, ...,
             MPI_MAX, root, ... );

/* The result is 2.0. */
```

Process 1

```
const int  root  = 0;
double     value = 2.0;
double     result;

MPI_Reduce ( &value, &result, ...,
             MPI_MAX, root, ... );

/* The result is undefined. */
```

Global reduction example

Here is the same program, but with `MPI_Allreduce` instead of `MPI_Reduce`.

Process 0

```
double  value = 1.0;
double  result;

MPI_Allreduce ( &value, &result,
                ..., MPI_MAX, ... );

/* The result is 2.0. */
```

Process 1

```
double  value = 2.0;
double  result;

MPI_Allreduce ( &value, &result,
                ..., MPI_MAX, ... );

/* The result is 2.0. */
```

Note that no root process has to be specified anymore, as the result is made available on all processes.

Global reduction

```
int MPI_Reduce ( const void* send_buf, void* recv_buf, int count,
                 MPI_Datatype type, MPI_Op op, int root,
                 MPI_Comm comm );  [nonlocal]
```

`send_buf`	IN	A pointer to the send buffer containing the data to be reduced.
`recv_buf`	OUT	A pointer to the receive buffer that is filled with the result(s).
`count`	IN	The number of data elements in `send_buf` and `recv_buf`.
`type`	IN	The type of the elements stored in the send/receive buffer.
`op`	IN	The reduction operation to be performed.
`root`	IN	The rank of the process where the results are to be made available.
`comm`	IN	The communicator to be used.

Global reduction

The parameters `send_buf` and `recv_buf` may point to a single variable or to an array with multiple elements. In the latter case, the reduction is performed for each array element independently.

Example:

Process 0

```
const int  root  = 0;
const int  sbuf[2] = { 1, 2 };
int        rbuf[2];

MPI_Reduce ( sbuf, rbuf, 2, ...,
             MPI_MAX, root, ... );

/* rbuf[0] equals 2, and
   rbuf[1] equals 4. */
```

Process 1

```
const int  root  = 0;
const int  sbuf[2] = { 2, 4 };
int        rbuf[2];

MPI_Reduce ( sbuf, rbuf, 2, ...,
             MPI_MAX, root, ... );

/* Contents of rbuf are undefined. */
```

Global reduction

The parameter op determines what kind of reduction operation is to be performed. It can have the following values:

MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product

MPI provides more reduction operations, but these are not discussed here. The operations listed above are the most commonly used ones.

Global reduction

```
int MPI_Allreduce ( const void* send_buf, void* recv_buf,
                    int count, MPI_Datatype type,
                    MPI_Op op, MPI_Comm comm );  [nonlocal]
```

`send_buf`	IN	A pointer to the send buffer containing the data to be reduced.
`recv_buf`	OUT	A pointer to the receive buffer that is filled with the result(s).
`count`	IN	The number of data elements in `send_buf` and `recv_buf`.
`type`	IN	The type of the elements stored in the send/receive buffer.
`op`	IN	The reduction operation to be performed.
`comm`	IN	The communicator to be used.

Global reduction

The function `MPI_Allreduce` is similar to `MPI_Reduce`, except that the former makes the results available on all processes. The `root` parameter is therefore not present.

If the results are only needed on one process, then it is more efficient to use `MPI_Reduce`.

Performance aspects

Use of the functions `MPI_Reduce` or `MPI_Allreduce` has a negative impact on the parallel scalability of a program.

The main reason is that the execution time of these functions increases with the number of processors. To be precise, their time complexity is $O(\log(p))$ with p the number of processes.

One should therefore try to avoid performing global reduction operations, although that is certainly not always possible.

At the very least one should try to minimize the number of calls to `MPI_Reduce` and `MPI_Allreduce` possibly by performing multiple reduction operations with one function call.

Performance aspects

To illustrate the last point, suppose that one wants to compute both the maximum and minimum of a set of numbers distributed across the processes.

Here is how to do that efficiently with one function call:

```
const int  value = ...;
const int  sbuf[2] = { value, -value };
int        rbuf[2];
int        minval, maxval;

MPI_Allreduce ( sbuf, rbuf, 2, MPI_INT,
                MPI_MAX, MPI_COMM_WORLD );

maxval =  rbuf[0];
minval = -rbuf[1];
```

Performance aspects

The previous performance-related remarks also apply to (most) other collective communication operations.

Avoiding collective communication operations will help increase the parallel performance of a parallel program.

If you can not avoid them, however, then try to use the collective communication functions provided by MPI instead of rolling your own. Sometimes MPI can take advantage of special hardware features to limit the performance impact of collective operations.

Also try to aggregate multiple operations in one function call to limit the latency-related overhead.

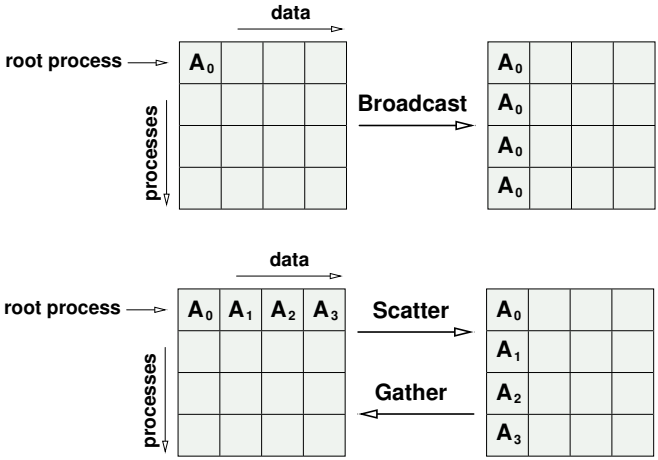
Data transfer

There are (approximately) three types of collective operations that transfer data between all processes associated with a communicator:

Broadcast	Send data from one process to all other processes.
Gather	Collect different data from all processes on one process.
Scatter	Send different data from one process to all other processes.

MPI provides a few more collective data transfer operations, but those are not considered here; most programs can do without them.

Data transfer



Broadcast

```
int MPI_Bcast ( void* buffer, int count, MPI_Datatype type,
                int root, MPI_Comm comm );  [nonlocal]
```

`buffer`	INOUT	Send/receive buffer (see below).
`count`	IN	Number of elements in the buffer.
`type`	IN	The type of the elements in the buffer.
`root`	IN	The rank of the broadcasting process.
`comm`	IN	The communicator.

The function `MPI_Bcast` sends `count` data elements from the process with rank `root` to all other processes.

Broadcast

On the process with rank `root`, the `buffer` parameter must contain the data to be broadcast.

On the other processes, the `buffer` parameter will contain those data when the function returns.

The example on the next slide shows how to broadcast two integers from the process with rank 0 to all other processes.

Broadcast example

```
const int  root = 0;
int        my_rank;
int        buffer[2];

MPI_Comm_rank ( MPI_COMM_WORLD, &my_rank );

if ( my_rank == root )
{
  buffer[0] = 1;
  buffer[1] = 2;
}

MPI_Bcast ( buffer, 2, MPI_INT, root, MPI_COMM_WORLD );
```

At the end of the program, the array `buffer` contains the values 1 and 2 on all processes.

Gather

```
int MPI_Gather ( const void* send_buf, int send_count,
                 MPI_Datatype send_type,
                 void* recv_buf, int recv_count,
                 MPI_Datatype recv_type, int root,
                 MPI_Comm comm );  [nonlocal]
```

send_buf	IN	Send buffer containing data to be sent.
send_count	IN	The number of data elements in the send buffer.
send_type	IN	The type of the elements in the send buffer.
recv_buf	OUT	Receive buffer.
recv_type	IN	The type of the elements in the receive buffer.
recv_count	IN	The number of data elements to be received from each process.
root	IN	The rank of the process to receive the data.
comm	IN	The communicator to be used.

Gather

The function `MPI_Gather` collects data from all processes on the process with rank `root`.

Each process, including `root`, must send the same number of data elements stored in `send_buf`; the parameters `send_count` and `send_type` must be the same on all processes.

The process with rank `root` receives the data elements in the buffer `recv_buf` in process rank order. That is, the first `send_count` elements contain the data from rank 0; the next `send_count` elements contain the data from rank 1; etc.

Gather

The parameter `recv_count` must be equal to the number of data elements to be received from each processor. In most cases it is equal to `send_count`.

There are exceptions possible, but these are not explained here.

The parameter `recv_type` is typically equal to `send_type`.

The parameters `recv_buf`, `recv_count` and `recv_type` are only significant on the process with rank `root`.

The next slide shows an example in which all processes send two integers to the process with rank 0.

Gather example

```
const int  root     = 0;
int*       recv_buf = NULL;
int        send_buf[2];
int        my_rank;
int        proc_count;

MPI_Comm_rank ( MPI_COMM_WORLD, &my_rank );
MPI_Comm_size ( MPI_COMM_WORLD, &proc_count );

send_buf[0] = my_rank + 1;
send_buf[1] = my_rank + 2;

if ( my_rank == root )
  recv_buf = malloc ( proc_count * 2 * sizeof(int) );

MPI_Gather ( send_buf, 2, MPI_INT, recv_buf, 2, MPI_INT,
             root, MPI_COMM_WORLD );
```

Scatter

```
int MPI_Scatter ( const void* send_buf, int send_count,
                  MPI_Datatype send_type,
                  void* recv_buf, int recv_count,
                  MPI_Datatype recv_type, int root,
                  MPI_Comm comm );  [nonlocal]
```

send_buf	IN	Send buffer containing data to be sent.
send_count	IN	The number of data elements in the send buffer.
send_type	IN	The type of the elements in the send buffer.
recv_buf	OUT	Receive buffer.
recv_type	IN	The type of the elements in the receive buffer.
recv_count	IN	The number of data elements to be received from each process.
root	IN	The rank of the process to send the data.
comm	IN	The communicator to be used.

Scatter

The function `MPI_Scatter` performs the reverse operation performed by `MPI_Gather`. That is, it sends data from the process with rank `root` to all other processes.

It is similar to `MPI_Bcast`, except that it sends different data to each process.

Each process, including `root`, must receive the same number of data elements that are stored in `recv_buf`; the parameters `recv_count` and `recv_type` must be the same on all processes.

The process with rank `root` sends the data elements in the buffer `send_buf` in process rank order. That is, the first `send_count` elements contain the data for rank 0; the next `send_count` elements contain the data for rank 1; etc.

Scatter

The parameter `send_count` will be equal to `recv_count` in most cases. Likewise, `send_type` is typically equal to `recv_count`. There are exceptions possible, but these are not explained here.

The parameters `send_buf`, `send_count` and `send_type` are only significant on the process with rank `root`.

The next slide shows an example in which process 0 sends two integers to all other processes.

Scatter example

```
const int  root     = 0;
int*       send_buf = NULL;
int        recv_buf[2];
int        i, my_rank, proc_count;

MPI_Comm_rank ( MPI_COMM_WORLD, &my_rank );
MPI_Comm_size ( MPI_COMM_WORLD, &proc_count );

if ( my_rank == root )
{
  send_buf = malloc ( proc_count * 2 * sizeof(int) );

  for ( i = 0; i < 2 * proc_count; i++ )
    send_buf[i] = i;
}

MPI_Scatter ( send_buf, 2, MPI_INT, recv_buf, 2, MPI_INT,
              root, MPI_COMM_WORLD );
```

Scatter example

The previous example can also be implemented with point-to-point operations. This is shown in the next slide.

Note that the use of `MPI_Scatter` is more efficient as – at the very least – it reduces the function-call overhead.

This also applies to `MPI_Gather`.

Scatter example

```
const int  root = 0;
const int  tag  = 0;
int        send_buf[2];
int        recv_buf[2];
int        i, my_rank, proc_count;

MPI_Comm_rank ( MPI_COMM_WORLD, &my_rank );
MPI_Comm_size ( MPI_COMM_WORLD, &proc_count );

if ( my_rank == root )
  for ( i = 0; i < proc_count; i++ ) {
    send_buf[0] = 2 * i + 0;
    send_buf[1] = 2 * i + 1;

    MPI_Send ( send_buf, 2, MPI_INT, i, tag, MPI_COMM_WORLD );
  }
else
  MPI_Recv ( recv_buf, 2, MPI_INT, root, tag,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE );
```

Exercise: Gram-Schmidt algorithm

Implement a parallel program that executes the Gram-Schmidt algorithm to make the columns of an $N \times N$ matrix $\boldsymbol{A}$ orthonormal through a series of dot products and vector subtractions.

The next slide shows the Gram-Schmidt algorithm in pseudo code. Note that $\boldsymbol{A}_j$ denotes the j-th column of the matrix $\boldsymbol{A}$.

The two slides after that show a serial implementation of the Gram-Schmidt algorithm.

Gram-Schmidt algorithm

do $j = 1, N$

do $k = 1, j-1$

$c_k := \boldsymbol{A}_k^T \cdot \boldsymbol{A}_j$

end do

$\boldsymbol{A}_j := \boldsymbol{A}_j - \sum_{k=1}^{j-1} c_k \cdot \boldsymbol{A}_k$

$\boldsymbol{A}_j := \frac{\boldsymbol{A}_j}{|\boldsymbol{A}_j|}$

end do

Serial Gram-Schmidt implementation (1)

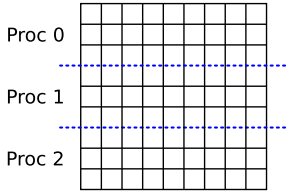
```
double  a[N][N];
double  c[N];
double  t;
int     i, j, k;

for ( j = 0; j < N; j++ )
{
  for ( k = 0; k < j; k++ )
  {
    c[k] = 0.0;

    /* Compute the dot products with the previous columns. */

    for ( i = 0; i < N; i++ )
      c[k] += a[i][k] * a[i][j];
  }

  /* Continued on the next slide ... */
```

Serial Gram-Schmidt implementation (2)

```
  /* ... continued from the previous slide. */

  for ( k = 0; k < j; k++ )
    for ( i = 0; i < N; i++ )
      a[i][j] -= c[k] * a[i][k];

  /* Normalize this column. */

  for ( i = 0, t = 0.0; i < N; i++ )
    t += a[i][j] * a[i][j];

  t = 1.0 / sqrt ( t );

  for ( i = 0; i < N; i++ )
    a[i][j] *= t;
}
```

Parallel Gram-Schmidt implementation

In the parallel implementation of the Gram-Schmidt algorithm, the matrix $\boldsymbol{A}$ is distributed row-wise over the processors:



This means that each process stores $\frac{N}{p}$ rows, with p the number of processes.

Hints

Use the sequential implementation of the Gram-Schmidt algorithm given in task *[MPI] Exercise 6 : Gram-Schmidt Algorithm* as the basis of your parallel version.

Use the functions `alloc_matrix` and `free_matrix` to create and destroy a matrix with arbitrary dimensions.

Use a global reduction operation to implement a parallel dot product.

What performance do you get?

Exercise: random numbers

Implement a parallel program that generates a stream of uniform random numbers within the range $[0, 1]$ within each process.

The program should stop if it has found ten numbers that are smaller than $\epsilon = 1 \cdot 10^{-7}$.

When the program has found ten numbers (or more), the master process (with rank zero) should collect all found numbers and print them to the terminal.

Exercise: hints

Use *[MPI] Exercise 6 : Random Number Generator* as the starting point for your program. This file implements a sequential version of the program.

Use the provided function `next_random` to generate the stream of random numbers. This function will set a different random seed for each process. Without this, no speedup is possible. Why not?

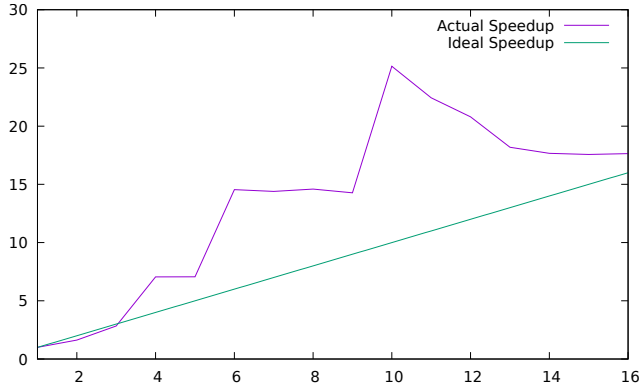
Use collective MPI functions to determine when the program should stop and to collect the results on the master.

Measure the parallel speedup of your program.

What can you do to improve the performance?

Exercise: obtained speedup

This is the speedup obtained with one particular implementation of the program on a 12-core machine.



Communicators

A *communicator* can be viewed as a virtual network through which a group of processes can communicate with each other.

Processes within a communicator are ordered and are identified by their rank within that communicator.

Processes may participate in multiple communicators in which they have a different rank.

When MPI is initialized it creates the communicator `MPI_COMM_WORLD` that contains all initial processes.

Communicators

Because a communicator defines its own virtual network, communication operations within different communicators do not interfere with each other.

Consider this code:

```
MPI_Comm  comm1, comm2;

/* Initialize comm1 and comm2 ... */

MPI_Recv ( ..., comm1, ... );
MPI_Recv ( ..., comm2, ... );
```

The first receive operation only matches a send operation within the communicator `comm1`. Likewise, the second receive operation only matches a send operation within `comm2`.

Why use communicators?

Why use other communicators than `MPI_COMM_WORLD`?

- *Modularity:* interference between communications performed in different program units or libraries can be avoided by using private communicators.
- *Efficiency:* when not all processes need to participate in a collective operation, it is more efficient to create a communicator including only the relevant processes.

Communicators and libraries

Communicators are more or less essential when developing general-purpose libraries that use MPI internally to execute algorithms in parallel.

To illustrate this, consider the following code:

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

void example ()
{
  MPI_Request  req;

  MPI_Irecv     ( ..., &req );
  matlib_invert ( ... );        /* Library call. */
  MPI_Wait      ( &req, ... );
}
```

Communicators and libraries

If the library function `matlib_invert` uses MPI to send messages between processes, then the `MPI_Irecv` call could interfere with the library.

That is, the `MPI_Irecv` call could be matched by an `MPI_Send` call performed in the function `matlib_invert`, instead of an `MPI_Send` call performed by the program itself.

This would certainly not lead to a happy end ...

Tags as communicators

The `tag` associated with a message can be used to simulate communicators.

That is, by using distinct tags in different program units/libraries, one can avoid interference between communication operations.

However, this only works for point-to-point communication and for situations in which one has control over the tags used in all point-to-point operations.

It does not work when using an external library that is developed by an independent third party.

Creating a communicator

MPI provides various functions for creating and managing communicators. Two of those are considered here:

`MPI_Comm_dup` – Duplicates a given communicator. The new communicator contains the same processes as the original.

`MPI_Comm_split` – Creates multiple communicators containing different sub-sets of processes in the original communicator.

Both functions use an existing communicator to create new communicators. The existing communicator can be `MPI_COMM_WORLD` or a communicator created with one of these functions.

Duplicating a communicator

```
int MPI_Comm_dup ( MPI_Comm src, MPI_Comm* copy );  [nonlocal]
```

src	IN	The communicator to be duplicated.
copy	OUT	A pointer to a communicator handle that will be set to the duplicated communicator.

The function MPI_Comm_dup creates a copy of an existing communicator.

All processes within the existing communicator must call this function. They all get a handle to the new communicator.

This function is typically used to create a private communicator within a program unit or library.

Duplicating a communicator

The processes have the same rank in the original and the new communicator.

```
MPI_Comm  comm;
int       rank1, rank2;

MPI_Comm_dup  ( MPI_COMM_WORLD, &comm );
MPI_Comm_rank ( MPI_COMM_WORLD, &rank1 );
MPI_Comm_rank ( comm,           &rank2 );

printf ( "My rank in MPI_COMM_WORLD is %d.\n", rank1 );
printf ( "My rank in comm           is %d.\n", rank2 );
```

The above code will print the same rank twice for each process.

Splitting a communicator

```
int MPI_Comm_split ( MPI_Comm src, int color, int key,
                     MPI_Comm* copy );  [nonlocal]
```

`src`	IN	The communicator to be split.
`color`	IN	A non-negative integer indicating how to split the original communicator.
`key`	IN	An integer that indicates how the processes are to be numbered within each sub-group.
`copy`	OUT	A pointer to a communicator handle that will be set to the new communicator for this process.

The function `MPI_Comm_split` partitions the processes within the source communicator into disjoint sub-groups.

Each sub-group gets its own communicator that is returned in the `comm` parameter.

Splitting a communicator

The parameter `color` controls how the processes in the source communicator are to be split into sub-groups.

That is, all processes specifying the same value for the `color` parameter end up in the same sub-group.

The `key` parameter controls how the processes are numbered in their sub-groups.

That is, the processes are ranked in the order defined by the `key` parameter, with ties broken by their rank in the source communicator.

The next slide shows how to divide the processes in `MPI_COMM_WORLD` into two sub-groups.

Splitting a communicator example

```
MPI_Comm  comm;
int       proc_count;
int       rank1, rank2;
int       color, key;

MPI_Comm_size ( MPI_COMM_WORLD, &proc_count );
MPI_Comm_rank ( MPI_COMM_WORLD, &rank1 );

if ( rank1 < (proc_count / 2) )
  color = 0;
else
  color = 1;

key = rank1;

MPI_Comm_split ( MPI_COMM_WORLD, color, key, &comm );
MPI_Comm_rank  ( comm, &rank2 );

printf ( "rank1 = %d, rank2 = %d\n", rank1, rank2 );
```

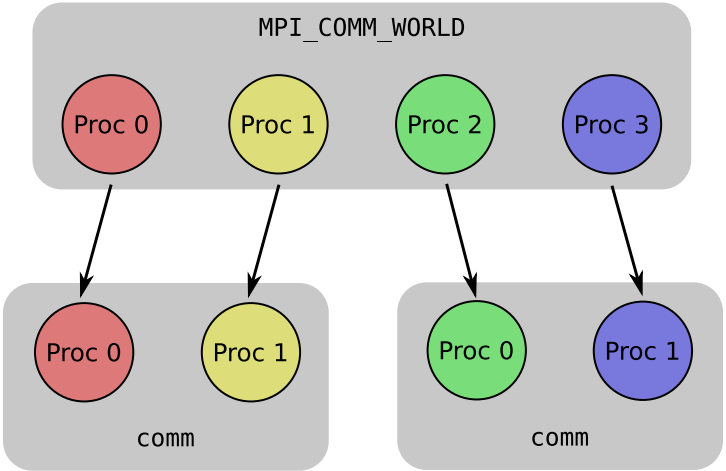
Splitting a communicator example

When the program is run with 4 processes, the output will be:

```
rank1 = 0, rank2 = 0
rank1 = 1, rank2 = 1
rank1 = 2, rank2 = 0
rank1 = 3, rank2 = 1
```

This is illustrated in a graphical way on the next slide.

Splitting a communicator example



Splitting a communicator example

The next slide shows an alternative implementation of the previous example.

In this case the odd-numbered processes are put in one sub-group, and the even-numbered processes are put in the other group.

As in the previous example, the ranks of the processes in `MPI_COMM_WORLD` are used to order the processes in the new communicators.

Splitting a communicator example

```
MPI_Comm  comm;
int       proc_count;
int       rank1, rank2;
int       color, key;

MPI_Comm_size ( MPI_COMM_WORLD, &proc_count );
MPI_Comm_rank ( MPI_COMM_WORLD, &rank1 );

if ( (rank1 % 2) == 0 )
  color = 0;
else
  color = 1;

key = rank1;

MPI_Comm_split ( MPI_COMM_WORLD, color, key, &comm );
MPI_Comm_rank  ( comm, &rank2 );

printf ( "rank1 = %d, rank2 = %d\n", rank1, rank2 );
```

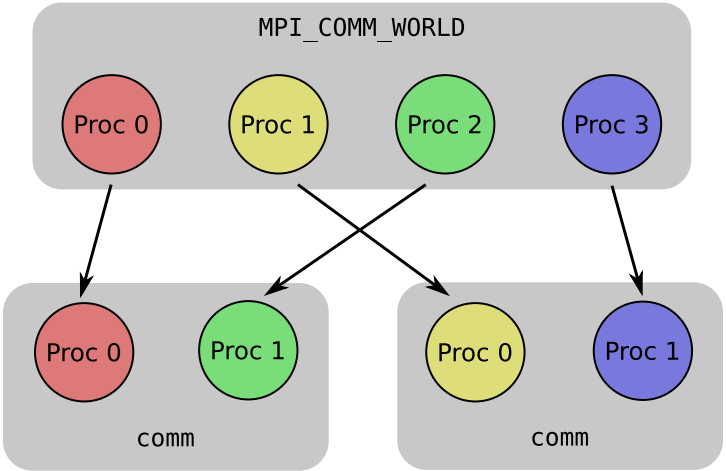
Splitting a communicator example

When the program is run with 4 processes, the output will be:

```
rank1 = 0, rank2 = 0
rank1 = 1, rank2 = 0
rank1 = 2, rank2 = 1
rank1 = 3, rank2 = 1
```

This is illustrated in a graphical way on the next slide.

Splitting a communicator example



Deleting a communicator

```
int MPI_Comm_free ( MPI_Comm* comm );  [nonlocal]
```

comm	INOUT	The communicator to be deleted.

The function `MPI_Comm_free` deletes a communicator that has been obtained with `MPI_Comm_dup` or `MPI_Comm_split`.

A call to one of these two functions should always be matched by a call to `MPI_Comm_free` in order to avoid resource leaks.

Deleting a communicator

The function `MPI_Comm_free` does not immediately delete the specified communicator; any ongoing (communication) operations involving the communicator will be completed before the communicator is deleted.

The following code is therefore valid:

```
MPI_Comm     comm;
MPI_Request  req;

MPI_Comm_dup  ( MPI_COMM_WORLD, &comm );
MPI_Irecv     ( ..., comm, &req );
MPI_Comm_free ( &comm );
MPI_Wait      ( &req, MPI_STATUS_IGNORE );
```

Exercise: competing groups of processes

Implement a game in which two groups of processes compete with each other by drawing random numbers.

The game proceeds in N rounds in which the two groups of processes generate a random number. The group with the largest number wins that round.

To generate a number per group, all processes in that group generate a random number independently. The largest number across all processes in the group is compared with the random number generated in a similar way by the other group.

The process with rank zero (within `MPI_COMM_WORLD`) should keep track of the number of wins for each group.

Exercise: hints

Use *[MPI] Exercise 7 : Game* as the starting point for your program.

Use the provided function `next_random` to generate the stream of random numbers.

Test the program with 2, 3, 4, 5, 6, 7 and 8 processes. Do the results match your expectations?

Note that this exercise is not so much about performance but about correctness.

Part III

OpenMP

Outline

What is OpenMP?

Open Multi-Processing

OpenMP is an application programming interface (API) for *multi-platform shared memory multiprocessing*.

It consists of a set of *compiler directives*, *library routines*, and *environment variables* that influence run-time behaviour.

OpenMP is managed by a group of major computer hardware and software vendors who define the *OpenMP specifications*. Compiler vendors are free to (partly) implement them in their products.

See `http://www.openmp.org`.

OpenMP specifications and implementations

	OpenMP specifications								
Compiler	5.1	5.0	4.5	4.0	3.1	3.0	2.5	2.0	1.0
GCC C/C++	—	(10)	6.1	4.9.1	4.7.0	4.4.0	4.2.0	?	?
Clang	—	11	(3.8)	(3.8)	3.8	—	—		
Intel C/C++	(2021.1)	(2021.1)	17	16	12				

There is no standard way to detect the OpenMP standard supported by the compiler. You can pass `-DOPENMPx` flags and switch between different implementations inside the code via

```
#ifdef OPENMP45
  /* Special implementation using OpenMP 4.5 features */
#else
  /* Generic implementation not using OpenMP 4.5 features */
#endif
```

Some newer OpenMP features make it possible to express an algorithm in a more elegant and/or more efficient way.

Why use OpenMP?

It is very easy to parallelise existing code incrementally.

It allows you to keep serial and parallel code side by side.

It enables both coarse- and fine-grained parallelisation.

It is implemented in most C, C++ and Fortran compilers.

It is platform independent; OpenMP runtime environment ensures that programs run on different parallel computers without modifications

It can be used on CPUs and GPUs/accelerators.

It can be used as the 'X' in hybrid MPI-X parallelisation.

What is included in OpenMP?

Directives for parallelisation within a single address space

Loop-parallelisation (Loop nesting since OpenMP 3.0)

Asynchronous task parallelisation (since OpenMP 3.0)

Places and thread affinity control (since OpenMP 4.0)

Vectorisation by SIMD-directives (since OpenMP 4.0)

Error handling (since OpenMP 4.0)

Offloading support for GPUs (since OpenMP 4.0)

Taskloop construct and doacross parallelism (since OpenMP 4.5)

What is not included in OpenMP?

Explicit communication between processes via messages

Fork-join model restricted to a single address space

Control on implementation of the runtime system; no direct control on thread creation/destruction (implementation-dependent thread-pool)

Support for proper parallel I/O

Debugging facilities

Semantics

Fork-join model

OpenMP programs follow the fork-join model starting with the sequential execution of instructions by *the master thread*

FORK: When a *parallel region* construct is encountered, the master thread spawns a *team of threads* to execute the program statements enclosed in the parallel region in parallel among the team threads

JOIN: Once the statements inside the parallel region are completed, all team threads synchronize and terminate leaving only the master thread

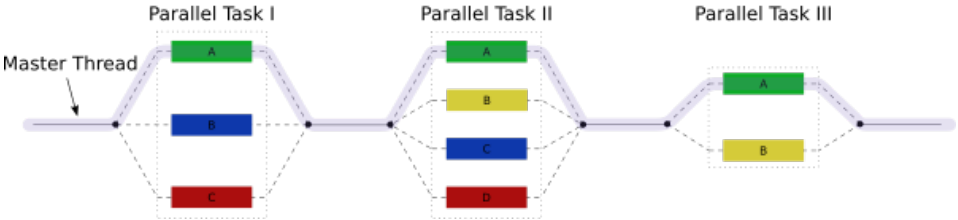
At each fork, a different number of worker threads can be spawned

Threads exist within a single process and cease to exist outside

Fork-join model

Sequential program executed by the master thread

OpenMP program executed partly in parallel



Semantics

Threads

Within an OpenMP program threads are identified by an integer number ranging from 0 to the number of threads minus one.

The master thread has thread number 0.

The thread number is returned by the run-time function

```
int tid=omp_get_thread_num();
```

Vendors of OpenMP compilers implement their own mechanism to create/destroy threads physically. One can therefore not assume that the same physical threads are active in two consecutive parallel regions

Semantics

Threads

OpenMP provides mechanisms to specify the number of threads to be allocated when forking from the master thread into a parallel region

- environment variable

```
set OMP_NUM_THREADS=8; progname
```

- run-time function (overwrites environment variable)

```
omp_set_num_threads(8);
```

The number of threads typically matches the number of processors/ cores. The actual number depends on the type of application

Threads

Threads can execute the same instructions (*work sharing*)

Thread 0

```
for (i=0; i<n/2; i++)
  a[i] = a[i]+1;
```

Thread 1

```
for (i=n/2+1; i<n; i++)
  a[i] = a[i]+1;
```

Threads can execute different instructions (*task parallelism*)

Thread 0

```
for (i=0; i<n; i++)
  a[i] = a[i]+1;
```

Thread 1

```
for (i=0; i<n; i++)
  b[i] = b[i]-1;
```

C/C++ language bindings

All OpenMP run-time functions have the `omp_` prefix

OpenMP functions are defined in the file `omp.h`, which must be included in any program that makes OpenMP calls

OpenMP parallelism is specified through the use of *compiler directives*

```
#pragma omp parallel
```

The compiler must be explicitly instructed to interpret these directives

```
gcc -fopenmp progname.c -o progname
```

The preprocessor macro `_OPENMP` is defined when the compiler/ preprocessor is invoked with OpenMP-support enabled

The basics of an OpenMP program

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

int main() {
  /* Serial code executed by master thread            */

  #pragma omp parallel
  {
    /* Parallel region executed by all threads        */
    /* Other OpenMP directives/run-time library calls */
  } /* All threads synchronize and join master thread */

  /* Serial code executed by master thread            */
}
```

Exercise: hello world

Implement a small OpenMP program that prints the total number of threads and the rank of the current thread.

A skeleton implementation of the program is given in task *[OpenMP] Exercise 1 : Hello World*.

Make sure that the program works in serial and parallel mode

Exercise: hints

Note that in serial mode the header file `omp.h` is not included so that `omp_get_thread_num()` and `omp_get_num_threads()` are not available.

Variables can be defined to be

- shared between all threads

```
#pragma omp parallel shared(variable)
```

- private to each thread

```
#pragma omp parallel private(variable)
```

Solution 1: hello world

```
int main() {
#ifdef _OPENMP
  int nthreads, tid;

  /* Start a parallel region with thread-private data */
  #pragma omp parallel private(nthreads, tid)
  { tid = omp_get_thread_num();
    printf("Thread number = %d\n", tid);

    if (tid == 0) { /* Master thread only */
       nthreads = omp_get_num_threads();
       printf("Number of threads = %d\n", nthreads);
    }
  }
#else
  printf("OpenMP disabled.\n");
#endif
}
```

Solution 2: hello world

Create preprocessor macros for the OpenMP runtime functions

```
#include<stdio.h>
#ifdef _OPENMP
#include<omp.h>
#else
#define omp_get_num_threads() 1
#define omp_get_max_threads() 1
#define omp_get_thread_num()  0
#endif
```

All `#pragma` statements are interpreted as comments if the code is compiled without OpenMP and therefore they are ignored.

Solution 2: hello world

```
int main() {
  int nthreads, tid;

  /* Start a parallel region with thread-private data */
  #pragma omp parallel private(nthreads, tid)
  { tid = omp_get_thread_num();
    printf("Thread number = %d\n", tid);

    /* Perform next tasks on master thread only */
    if (tid == 0) {
       nthreads = omp_get_num_threads();
       printf("Number of threads = %d\n", nthreads);
    }
  }
}
```

Parallel regions: private vs. shared variables

Parallel regions are processed by all threads in the team. Threads can have *private copies* of variables or access to a *single global* variable:

```
/* Define and initialize variables */
int global=0, local=0;

/* Create parallel region with 4 threads */
omp_set_num_threads(4);
#pragma omp parallel region shared(global) private(local)
{
  /* Each thread has its own private copy of variable 'local' */
  printf("Local: %d\n", local++);

  /* All threads have read/write access to variable 'global'  */
  printf("Global: %d\n", global++);
}
```

Parallel regions: default behaviour of variables

Compiler vendors are free to choose the default behaviour of variables if you do neither specify shared nor private treatment.

The default behaviour of variables can be specified further

- Treat all variables that are not specified explicitly as globally shared

```
#pragma omp parallel default(shared)
```

- Force the user to specify the treatment for all variables explicitly

```
#pragma omp parallel default(none)
```

Some compilers also support `#pragma omp parallel default(private)` but this is not specified in the OpenMP specification for C language.

Parallel regions: (first)private

Thread private copies of variables are not initialized by the value that the variable has before the parallel region

```
int local=7;
#pragma omp parallel private(local)
printf("My thread number + offset: %d\n", local+omp_get_thread_num());
/* This will print 0, 1, ... */
```

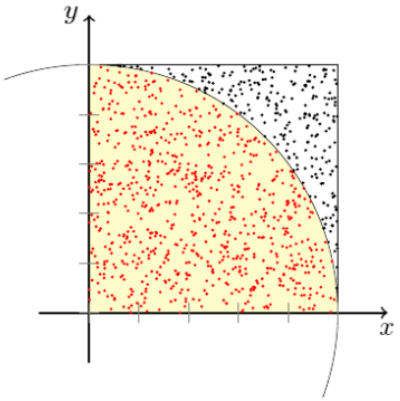
Thread private copies of variables are initialized by the value that the variable has before the parallel region by the *firstprivate* clause

```
int local=7;
#pragma omp parallel firstprivate(local)
printf("My thread number + offset: %d\n", local+omp_get_thread_num());
/* This will print 7, 8, ... */
```

Example: parallel computation of π

The area of a circle of radius R is πR^2, the area of a square with side length $2R$ is $4R^2$. If the circle is inscribed inside the square then the ratio of both areas is $\pi/4$.

Monte Carlo method: if we pick N points at random inside the square and count the number of points that are inside the circle (M) then we can compute the approximation $\pi \approx 4M/N$.



Open [OpenMP] Exercise 2 : Compute PI and rewrite the sequential algorithm as a parallel algorithm using parallel regions.

Example: hints

You can retrieve an upper bound of the numbers of threads that will be created in a parallel regions using `omp_get_max_threads()`. This even works outside any parallel region.

Use an array of size `omp_get_max_threads()` to pass data from the different threads to the master thread.

Use `#pragma omp parallel default(none)` to force yourself to specify the behaviour of all variables (private/shared) explicitly.

Parallel for

For-loops can be very easily transformed into parallel for-loops

```
int a[n];

#pragma omp parallel shared(a) /* Parallel region   */
#pragma omp for                /* For-loop */
for (int i=0; i<n; i++)
{
  a[i]=i;
}
```

The OpenMP runtime system takes care that each thread works on a separate part of the globally accessible shared array `a[0:n-1]`.

Do not change the upper bound of the loop counter to `n/nprocs`.

Parallel for

The pragmas for parallel regions and for-loops can be combined

```
int a[n];

/* Parallel for-loop */
#pragma omp parallel for shared(a)
for (int i=0; i<n; i++)
{
  a[i]=i;
}
```

Note that in the above code a team of threads is spawned only to process the for-loop in parallel and all threads except for the master thread are killed afterwards.

Parallel for

Keep in mind that at the end of a parallel region all threads in the team except for the master thread are killed. Spawning threads over and over again for each new parallel region can become costly.

```
/* Spawn team of threads and */
/* perform parallel for-loop /*/
#pragma omp parallel for shared(a)
for (int i=0; i<n; i++)
{
  a[i]=i;
} /* Kill team of threads */

/* Spawn team of threads and */
/* perform parallel for-loop */
#pragma omp parallel for shared(b)
for (int i=0; i<n; i++)
{
  b[i]=i;
} /* Kill team of threads */
```

Parallel for

It can make sense to keep separate pragmas for parallel regions and for-loops to prevent repeated spawning and killing of threads.

```
/* Spawn team of threads */
#pragma omp parallel shared(a,b)
{
  /* Parallel for-loop */
  #pragma omp for
  for (int i=0; i<n; i++)
  {
    a[i]=i;
  }
  /* Parallel for-loop */
  #pragma omp for
  for (int i=0; i<n; i++)
  {
    b[i]=i;
  }
} /* Kill team of threads */
```

Parallel for: ordered

If it is essential that the parallel for-loop is to be executed like a sequential loop you can use the `ordered` directive:

```
int a[n];
a[0]=1;
a[1]=1;

/* Compute Fibonacci sequence */
#pragma omp parallel for shared(a) ordered
for (int i=2; i<n; i++)
{
  a[i] = a[i-1] + a[i-2];
}
```

However, the `ordered` directive might cause truly sequential execution. It is therefore not advisable to use it; try to change your algorithm first.

Parallel for: schedule

The OpenMP runtime system knows different strategies to divide the workload and distribute the work between threads.

A strategy can be chosen using the `schedule(<strategy>)` directive.

```
int a[n];
#pragma omp parallel for shared(a) schedule(<strategy>)
for (int i=0; i<n; i++)
  a[i]=i;
```

If no strategy is chosen by you then the OpenMP compiler and/or runtime system is free to choose one. This behaviour can be made explicit by specifying `schedule(auto)`.

Parallel for: schedule(static)

`schedule(static, [chunk_size])` divides the iteration into chunks of size `chunk_size` and assigns them to the threads in the team in a round-robin fashion

```
int a[100];

omp_set_thread_num(4);
#pragma omp parallel for shared(a) schedule(static, 30)
for (int i=0; i<100; i++)
  a[i]=i;
```

Thread 0 processes entries 0-29, thread 1 processes entries 30-59, thread 2 processes entries 60-89, and thread 3 processes entries 90-99.

Parallel for: schedule(static)

One can specify a chunk size so that there are more subtasks than threads in the team

```
int a[100];

omp_set_thread_num(4);
#pragma omp parallel for shared(a) schedule(static,20)
for (int i=0; i<100; i++)
  a[i]=i;
```

All threads will first work on a chunk of size 20. Once the first thread has finished execution it will process the last chunk of size 20, while all other threads are waiting (implicit barrier!).

Parallel for: schedule(static)

When no `chunk_size` is specified, the iteration is divided into chunks that are approximately equal in size, and at most one chunk is distributed to each thread.

```
int a[100];

omp_set_thread_num(4);
#pragma omp parallel for shared(a) schedule(static)
for (int i=0; i<100; i++)
  a[i]=i;
```

All threads are working on a chunk of size 25.

Parallel for: schedule(static)

When no `chunk_size` is specified, the iteration is divided into chunks that are approximately equal in size, and at most one chunk is distributed to each thread.

```
int a[100];

omp_set_thread_num(3);
#pragma omp parallel for shared(a) schedule(static)
for (int i=0; i<100; i++)
  a[i]=i;
```

The OpenMP compiler can decide to have thread 0 and thread 1 working on chunks of size 33, and thread 2 working on a chunk of size 34.

Parallel for: schedule(dynamic)

`schedule(dynamic,[chunk_size])` divides the iteration into chunks of size `chunk_size` and distributes them to threads dynamically. Once a thread has finished its chunk it requests another chunk until all chunks have been processed.

```
int a[100];

omp_set_thread_num(4);
#pragma omp parallel for shared(a) schedule(dynamic,10)
for (int i=0; i<100; i++)
  a[i]=i;
```

Each chunk contains `chunk_size` iterations, except for the chunk that contains the sequentially last iteration, which may have fewer iterations.

When no `chunk_size` is specified, it defaults to 1.

Parallel for: schedule(dynamic)

Dynamic scheduling is useful if the size of a chunk hardly correlates with the time needed to process it.

```
int a[100];

omp_set_thread_num(4);
#pragma omp parallel for shared(a) schedule(dynamic,1)
for (int i=0; i<100; i++)
  do_costly_procedure_with_varying_computing_time(a[i], i);
```

For instance, the procedure might perform different operations on `a[i]` that require between milliseconds and minutes to finish depending on the value of `i`.

Parallel for: schedule(guided)

When `schedule(guided,[chunk_size])` is used threads request chunks dynamically until all chunks are processed.

For a `chunk_size` of 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads in the team, decreasing to 1.

or a `chunk_size` of k, the size of each chunk is determined in the same way but no chunk contains fewer than k iterations (except for the sequentially last one).

Parallel for: schedule(guided)

```
int a[1234];

omp_set_thread_num(5);
#pragma omp parallel for shared(a) schedule(guided,1)
for (int i=0; i<1234; i++)
  a[i]=i;
```

First, all five threads are processing chunks of size $\lfloor 1234/5 \rfloor = 246$. The remaining $1234 - 5 \times 246 = 4$ iterations are split into chunks of size 1 and processed by thread 0 to thread 3.

`schedule(guided,4)` prevents the splitting of the remainder into chunks of size 1 so that the last 4 iterations are processed by thread 0.

Parallel for: implicit barrier

By default, the OpenMP runtime system forces all threads to wait at the end of a for-loop until all threads have finished execution.

This behaviour is called an *implicit barrier*. Obviously, some threads are idle while others are still executing; the slowest thread determines the overall computing time.

Many OpenMP constructs (like `section`, `workshare`, etc.) have an implicit barrier to ensure that execution only continues once the previous task has been completed by all threads in the team.

At the end of each parallel section all threads wait until they get killed.

Parallel for: nowait

If multiple independent for-loops are executed one after the other in the *same parallel region* one can instruct the OpenMP runtime system to *not wait* but continue immediately by using the `nowait` directive

```
int a[n], b[n];

#pragma omp parallel shared(a,b)
{
  #pragma omp for nowait  /* All threads start at the same time   */
  for (int i=0; i<n; i++)
  {
    a[i]=i;
  }               /*  Once a thread has finished the above loop   */
  #pragma omp for /*  it executes the second loop without waiting */
  for (int i=0; i<n; i++)
  {
    b[i]=i;
  } /* implicit barrier */
}
```

Parallel for: nowait

However, `nowait` is dangerous if the second loop depends on data that is produced in the first loop. In this case one has to guarantee that data from the first loop is available before it is used in the second loop.

```
int a[n], b[n];

#pragma omp parallel shared(a,b)
{
  #pragma omp for
  for (int i=0; i<n; i++)
  {
    a[i]=i;
  } /* implicit barrier */

  #pragma omp for
  for (int i=0; i<n; i++)
  {
    b[i]=a[i]; /* okay since first loop has implicit barrier */
  }
}
```

Parallel for: nowait

However, `nowait` is dangerous if the second loop depends on data that is produced in the first loop. In this case one has to guarantee that data from the first loop is available before it is used in the second loop.

```
int a[n], b[n];

#pragma omp parallel shared(a,b)
{
  #pragma omp for nowait
  for (int i=0; i<n; i++)
  {
    a[i]=i;
  } /* NO implicit barrier */

  #pragma omp for
  for (int i=0; i<n; i++)
  {
    b[i]=a[i]; /* this can go wrong if a[i] has not been processed */
  } /* implicit barrier */
}
```

Parallel for: nowait

`schedule(static)` ensures that each thread processes *exactly the same* set of chunks in multiple consecutive parallel for-loops.

```
int a[n], b[n];

#pragma omp parallel shared(a,b)
{
  #pragma omp for nowait schedule(static)
  for (int i=0; i<n; i++)
  {
    a[i]=i;
  }  /* NO implicit barrier */

  #pragma omp for schedule(static)
  for (int i=0; i<n; i++)
  {
    b[i]=a[i]; /* okay since the thread that processes this chunk has  */
               /* also processed the corresponding chunk in first loop */
  }
}
```

Exercise: Parallel axpy

Write a parallel program that computes

$$y := \alpha x + y$$

for two vectors $x, y \in \mathbb{R}^N$ and a scalar coefficient $\alpha \in \mathbb{R}$.

Test your program for different scheduling strategies and chunk sizes, and different values for N.

You can start from the sequential implementation in *[OpenMP] Exercise 3: AXPY Operation*.

Parallel for: collapse

Multiple nested for-loops can be collapsed into one large parallel for-loop by using the `collapse` directive.

```
int a[n][m];

#omp parallel for shared(a) collapse(2)
for (int i=0; i<n; i++)
  for (int j=0; j<m; j++)
  {
    a[i][j] = i+j;
  }
```

The collapsed loops will be treated as a single parallel for-loop from 0 to `n*m`. This makes it much easier to divide the overall work into chunks and assign them to threads.

Parallel for: lastprivate

`firstprivate(i)` declares `i` as private variable and initializes its thread-private copy by the value of `i` *before* the parallel region.

`lastprivate(i)` declares `i` as private variable and updates its value by the value of the thread-private copy *after* the parallel region.

```
  int a=10;
  int n=10;

  omp_set_num_threads (3);
#pragma omp parallel for firstprivate(a) lastprivate(a) \
                                         schedule(static,3)
  for (int i=0; i<n; i++)
    a++;

 printf("%d\n", a); /* What is the answer? */
```

Parallel reduction

The `reduction` directive combines multiple private copies from all threads in a team into a single number using various types of reduction operations, such as min, max and `+`, `-`, `*`, `/`, `&`, `|`, `^`, `&&`, `||`.

```
int a[n], b[n];
long int sum;

#pragma omp parallel for shared(a,b)
for (int i=0; i<n; i++)
  a[i] = b[i] = i;

#pragma omp parallel for shared(a,b) reduction(+:sum)
for (int i=0; i < n; i++)
  sum = sum + (a[i] * b[i]);

printf("Sum = %d\n", sum);
```

Exercise: Parallel inner product

Write a parallel program that computes the inner product

$$\mathbf{x}^\top \mathbf{y} = \sum_{i=1}^{N} x_i y_i$$

for two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^N$.

Test your program for different scheduling strategies and chunk sizes, and different values for N.

You can start from the sequential implementation in *[OpenMP] Exercise 4 : Inner Product*.

Exercise: Parallel dyadic product

Write a parallel program that computes the dyadic product

$$\mathrm{x}\mathrm{y}^\top = \begin{bmatrix} x_1 y_1 & x_1 y_2 & \dots & x_1 y_N \\ x_2 y_1 & x_2 y_2 & \dots & x_2 y_N \\ \vdots & \vdots & \ddots & \vdots \\ x_N y_1 & x_N y_2 & \dots & x_N y_N \end{bmatrix}$$

for two vectors $\mathrm{x}, \mathrm{y} \in \mathbb{R}^N$.

Test your program for different scheduling strategies and chunk sizes, and different values for N.

You can start from the sequential implementation in *[OpenMP] Exercise 5 : Dyadic Product*.

Influencing the number of threads

OpenMP allows to explicitly define the number of threads that should be used in a *parallel region* by using the directive `num_threads(n)`. Note that this has to be done at the beginning of a parallel region

```
int n = 1000;
long int sum = 0;
#pragma omp parallel shared(sum) num_threads(4)
#pragma omp for reduction(+:sum)
for (int i=0; i<n; i++)
  sum++;
```

What is the value of `sum` after the parallel region?

Conditional parallelization

OpenMP allows to enable/disable parallelization based on conditions.

```
#pragma omp parallel for reduction(+:sum) if (n>100)
for (int i=0; i<n; i++)
  sum++;
```

In practice, the overhead caused by spawning and killing threads typically only pays off if the size of the task to be done in parallel is sufficiently large. A common approach is as follows

```
#define OMP_MIN_SIZE = 1000

#pragma omp parallel for reduction(+:sum) if (n>OMP_MIN_SIZE)
for (int i=0; i<n; i++)
  sum++;
```

Custom reduction rules

In OpenMP it is very easy to `declare reduction` rules that realize more complex reduction expressions and/or perform custom reduction rules for user-defined structures. The input and output variables are called `omp_in` and `omp_out`, respectively.

```
int n=1000;
long int sum=0;

#pragma omp declare reduction(mysum : \
                              int   : \
                              omp_out = omp_out + omp_in)
#pragma omp parallel for reduction(mysum:sum)
for (int i=0; i<n; i++)
  sum++;
```

This yields `sum = 1000`.

Custom reduction rules

The initial value of the reduction can be specified explicitly by defining the thread-private variable `omp_priv`

```
int n=1000;
long int sum=0;

#pragma omp declare reduction(mysum : \
                              int   : \
                              omp_out = omp_out + omp_in) \
                              initializer (omp_priv = 42)
#pragma omp parallel for reduction(mysum:sum)
for (int i=0; i<n; i++)
  sum++;
```

This yields `sum = 1000 + OMP_NUM_THREADS * 42`.

Custom reduction rules

Attributes/functions of the user-defined structure can be used in the custom reduction rule. Note that multiple assignments in the custom reduction rule need to be separated by ',' and **not** ';'

```
int n=1000;
struct point { int a; int b; } sum;

#pragma omp declare reduction(mysum : struct point : \
                              omp_out.a = omp_out.a + omp_in.a, \
                              omp_out.b = omp_out.b + omp_in.b)
#pragma omp parallel for reduction(mysum:sum)
for (int i=0; i<n; i++) {
  sum.a++;
  sum.b++;
}
```

This yields `sum.a = 1000` and `sum.b = 1000`.

Custom reduction rules

Custom reduction rules are very powerful especially in combination with C++ meta-programming techniques.

It is, for instance, possible to declare custom reduction rules together with the implementation of a structure/class

```
template <typename T>
struct point {
  T a,b;
  point() { a=0; b=0; }
};
#pragma omp declare reduction(pointsum : point : \
                              omp_out.a = omp_out.a + omp_in.a, \
                              omp_out.b = omp_out.b + omp_in.b)
                              initializer (omp_priv = point())
```

and use the struct/class as if it was an intrinsic data type.

Barriers

OpenMP adds *implicit barriers* at the end of each parallel region, parallel for, etc. Threads can be explicitly instructed to *not wait* and proceed with the next parallel for.

It is also possible to explicitly instruct threads to *wait* at a particular location of the code by using an explicit `barrier`

```
#pragma parallel shared(a,b,c)
{
  /* Initialize a */
  /* Initialize b */
  #pragma omp barrier /* All threads in the team will wait here */

  /* Compute c = a+b */
}
```

Atomic operations

Due to the shared-memory nature of OpenMP, all threads in the team have the possibility to read/write from/to the same memory location.

This concurrent memory access can cause the following problems:

- If two threads try to update the same memory location (a++) at the same time one update (+1) gets lost

```
int n=1000;
int a=0;

#pragma omp parallel for shared(a)
for (int i=0; i<n; i++)
  if (i%10 == 0)
    a++;
```

Atomic operations

Due to the shared-memory nature of OpenMP, all threads in the team have the possibility to read/write from/to the same memory location.

This concurrent memory access can cause the following problems:

- If `thread 0` tries to read from a memory location which is updated by `thread 1` at the same time then `thread 0` reads corrupted data.

```
int n=1000;
struct point { int a; int b; } p;
#pragma omp parallel for shared(p)
for (int i=0; i<n; i++)
  if (i%10 == 0) {
    int t = p.a;
    p.a   = p.b;
    p.b   = t;
  }
```

Atomic operations

The `atomic` construct ensures that a specific memory location is accessed *atomically*, that is, by exactly one thread at a time

```
int n=1000;
int a=0;
#pragma omp parallel for shared(a)
for (int i=0; i<n; i++)
{
  #pragma omp atomic
  if (i%10 == 0) a++;
}
```

Be careful with using `atomic` since it will add extra code that *locks* access to the specified memory location and removes the lock afterwards. Therefore, carefully analyse your algorithm and only use `atomic` if it is *really* needed.

Atomic operations

OpenMP allows fine-grained control over `atomic` memory access

- `read`: allows that multiple threads read from the same memory location at the same time but that no other thread modifies it

```
#pragma omp atomic read
v = a[i];
```

- `write`: prevents multiple threads from writing to the same memory location at the same time without reading from it before

```
#pragma omp atomic write
a[i] = v;
```

- `update`: prevents multiple threads from updating the same memory location at the same time

```
#pragma omp atomic update
a[i]++; /* a[i] = a[i]+1 */
```

Atomic operations

As the name says, `atomic` only ensures *atomic memory access to an individual memory location*. It therefore cannot handle situation in which a sequence of operations has to be applied without interference by other threads to ensure data consistency

```
int n=1000;
struct point { int a; int b; } p;
#pragma omp parallel for shared(p)
for (int i=0; i<n; i++)
  if (i%10 == 0)
  {
    #pragma omp atomic
    int t = p.a;
    #pragma omp atomic /* p.a can be overwritten by another thread */
    p.a   = p.b;
    #pragma omp atomic
    p.b   = t;
  }
```

Critical sections

A code block that, as a whole, must be executed by only one thread at a time has to be marked as a *critical section*

```
int n=1000;
struct point { int a; int b; } p;
#pragma omp parallel for shared(p)
for (int i=0; i<n; i++)
  if (i%10 == 0)
  {
    #pragma omp critical
    {
      int t = p.a;
      p.a   = p.b;
      p.b   = t;
    }
  }
```

Atomic vs. critical sections

Concerning computational overhead, `atomic` is more lightweight than `critical section`.

- `atomic` restricts accesses to the same memory location but it allows multiple threads to perform the same operation on different memory locations
- `critical section` allows only one thread at a time to perform the instructions of an entire code block even if they would apply the operations to different memory locations

Named critical sections

OpenMP allows to distinguish between multiple critical sections that can be performed simultaneously by giving them different names

```
int n=1000;
struct point { int a; int b; } p10;
struct point { int a; int b; } p20;
#pragma omp parallel for shared(p10,p20)
for (int i=0; i<n; i++)
  if (i%10 == 0)
  {
    #pragma omp critical point10
    { int t = p10.a; p10.a = p10.b; p10.b = t; }
  }
  if (i%20 == 0)
  {
    #pragma omp critical point20
    { int t = p20.a; p20.a = p20.b; p20.b = t; }
  }
```

Exercise: Conditional counting

Write a program that generates a sequence of random numbers between 0 and N and counts how many times the integer part of that number equals $0, 1, \ldots, N$. This process is known as generating a *histogram*.

Implement two version, one with `atomic` the other with `critical section`, and compare the performance of both implementations for different N and a large number of sampling points.

You can start from *[OpenMP] Exercise 6 : Histogram*.

Differentiation between threads

It is possible to distinguish between different threads making use of the runtime function `omp_get_thread_num()`. For instance, one could define three different tasks

```
if (omp_get_thread_num() == 0)
{
  /* Task for the master thread */
}
else if (omp_get_thread_num() == 1)
{
  /* Task for thread no. 1 */
}
else if (omp_get_thread_num() == 2)
{
  /* Task for thread no. 2 */
}
```

However, this contradicts the OpenMP paradigm that the same code should work in sequential and parallel mode.

Parallel sections

Sections are a better way to assign different tasks to different threads

```
#pragma omp sections
{
  #pragma omp section
  /* Task for a single threads */

  #pragma omp section
  /* Task for another single threads */

  #pragma omp section
  /* Task for yet another single threads */
}
```

Each block of code is executed once by one of the threads in the team. However, it is not clear that the master thread will execute the first block, etc.

Parallel sections

The `#pragma omp sections` construct accepts `private`, `firstprivate`, `lastprivate`, `reduction`, and `nowait`

```
int a=1;
int sum;

#pragma omp sections firstprivate(a) reduction(+:sum) nowait
{
  #pragma omp section
  sum = ++a;

  #pragma omp section
  sum = 2*a;
}
```

Master/single section

The *single* construct ensures that a code block is executed by a single thread, which is not necessarily the master thread

```
#pragma omp single
{
  printf("Total number of threads is %d\n", omp_get_num_threads());
}
```

The *master* construct ensures that this is the master thread

```
#pragma omp master
{
  printf("Total number of threads is %d\n", omp_get_num_threads());
}
```

`single` and `master` accept `private`, `firstprivate`, and `nowait`.

Exercise: Circular array shift

Implement a parallel, circular array shift.

Here is the serial implementation of the algorithm:

```
const int  N = 100;
double     a[N];
double     t;
int        i;

t = a[N - 1];

for ( i = N - 1; i >= 1; i-- )
  a[i] = a[i - 1];

a[0] = t;
```

You can start from *[OpenMP] Exercise 7 : Circular Array Shift*.

Calling functions within parallel regions

OpenMP allows to call functions from parallel regions

```
void hello() { printf("Hello from thread %d\n", omp_get_thread_num()); }

int main() {
  #pragma omp parallel num_threads(4)
  hello();
}
```

This will result in

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

Calling functions within parallel regions

OpenMP allows to call *parallelized* functions from parallel regions

```
void hello() {
  #pragma omp parallel num_threads(4)
  printf("Hello from thread %d\n", omp_get_thread_num());
}

int main() {
  #pragma omp parallel num_threads(4)
  hello();
}
```

This will, however, result (at least with the lecturer's compiler) in

```
Hello from thread 0
Hello from thread 0
Hello from thread 0
Hello from thread 0
```

Calling functions within parallel regions

By default, OpenMP does not enable nested parallelization so that the parallel region in the `main` function is executed by four different threads but each call to the `hello` function only uses the master thread.

Since OpenMP 2.5 *nested parallelization* can be enabled

- via environment variable `OMP_NESTED=TRUE`
- via runtime function `omp_set_nested(1)`

Calling functions within parallel regions

```
void hello() {
  #pragma omp parallel num_threads(2)
  printf("Hello from thread %d\n", omp_get_thread_num());
}

int main() {
  omp_set_nested(1);
  #pragma omp parallel num_threads(4)
  hello();
}
```

What will be the output?

What will be the output if you swap the numbers 2 and 4.

Calling functions within parallel regions

If you want to write a function that can be called from a parallel region make sure that the function is *thread-safe*. That is, the function can be safely called multiple times in parallel. Typical examples for not thread-safe functions are those which store data internally

```
int increase_by_counter(int i)
{
  static int counter=0;
  counter++;
  return i+counter;
}
```

You can simply render the above function thread-safe by using `atomic`

Nested parallelization

Use nested parallelization to improve the efficiency of nested for-loops. The simplest approach is to `collapse` multiple for-loops. This even works with nested parallelization turned off.

```
int N=1000;
double A[N][N], x[N], y[N];

#pragma omp parallel for collapse(2)
for (int i=0; i<N; i++)
{
  for (int j=0; j<N; j++)
  {
    y[i] += A[i][j]*x[j]
  }
}
```

Nested parallel regions

It is also possible to explicitly create two nested parallel for-loops.

```
int N=1000;
double A[N][N], x[N], y[N];

omp_set_nested(1);
#pragma omp parallel for num_threads(2)
for (int i=0; i<N; i++)
{
  double sum = 0.0;
  #pragma omp parallel for num_threads(2) reduction(+:sum)
  for (int j=0; j<N; j++)
  {
    sum += A[i][j]*x[j]
  }
  y[i] = sum;
}
```

Nested parallel regions

Be careful with hand-written nested parallelization and test your implementation extensively. The overhead of nested parallelization (spawn/kill threads) can significantly increase the computing time.

IMHO, nested parallel for-loops have poor performance in most cases unless the outer for-loop has only very few iterations and the inner one has many. However, even in this case the `collapse` clause should be more efficient than hand-written nested parallel regions.

If a simple `collapse` clause is not applicable or does not perform well then fine-grained parallelization using explicit creation of `tasks` a better way towards nested parallelization.

SIMD

Modern CPUs consist of multiple compute cores (*parallelization*) and each core is able to perform SIMD operations (*vectorization*).

Since OpenMP 4.0 the `simd` construct exists to explicitly tell the compiler to *vectorize for-loops* (if the compiler does not auto-vectorize)

```
int N=1000;
double x[N], y[N], a=2.0;

#pragma omp simd
for (int i=0; i<N; i++)
  y[i] = a*x[i] + y[i];
```

Example for vector length 4

```
y[0:3]  = a*x[0:3]  + y[0:3];
y[4:7]  = a*x[4:7]  + y[4:7];
y[8:11] = a*x[8:11] + y[8:11];
        ...
```

The above *vectorized for-loop* will be executed by a *single thread using SIMD operations* even if the computer has multiple cores.

Note that the `simd` construct does not accept `shared(variable)`.

Parallel for SIMD

The combination of `parallel` and `simd` tells the compiler to *parallelize* the execution of the loop and to use *vectorization* within each thread.

```
int N=1000;
double x[N], y[N], a=2.0;

#pragma omp parallel for simd
for (int i=0; i<N; i++)
  y[i] = a*x[i] + y[i];
```

```
tid0: y[0:3]   = a*x[0:3]   + y[0:3];
      y[4:7]   = a*x[4:7]   + y[4:7];
      y[8:11]  = a*x[8:11]  + y[8:11];
                ...
tid1: y[N/2:N/2+4] = a*x[N/2:N/2+4]
                   +   y[N/2:N/2+4];
                  ...
```

In this case it is typically better to leave the decision on the scheduling strategy to the compiler unless you really know what you are doing.

Parallel for & SIMD

Another typical programming pattern that combines `parallel for` and `simd` is to parallelize the outer loop and to vectorize the inner one[1]

```
int N=1000;
double x[N][N], y[N][N];

#pragma omp parallel for
for (int i=0; i<N; i++)
{
  #pragma omp simd safelen(18)
  for (int j=18; j<N-18; j++) {
    x[i][j] = x[i][j-18] + sinf(y[i][j]);
    y[i][j] = y[i][j+18] + cosf(x[i][j]);
  }
}
```

[1] Example from: "Explicit Vector Programming with OpenMP 4.0 SIMD Extensions" X. Tian, B.R. de Supinski, Primeur Magazine 2014-11-10.

SIMD safelen

Without the `simd` construct the compiler would not be able to perform auto-vectorization of the inner loop because:

- the loop construct cannot easily be extended to the j-loop, since it would violate the construct's semantics
- the j-loop contains a lexically backward loop-carried dependency that prohibits vectorization

However, the code can be vectorized for any given vector length for array y and for vectors shorter than 18 elements for array x.

This extra information is necessary for vectorization and has to be provided by the user using the `safelen` clause.

SIMD safelen

In the following example, there is a loop-carried lexically backward dependency. Chunks of size 4 or less can be vectorized but the sequence of chunks has to be processed sequentially.

```
#pragma omp simd safelen(4)
for (k=5; k<N; k++)
{
  a[k] = a[k-4] + b[k];
}
```

```
a[5] = a[1] +  b[5];
a[6] = a[2] +  b[6];
a[7] = a[3] +  b[7];
a[8] = a[4] +  b[8];

a[9] = a[5] +  b[9];
a[10] = a[6] + b[10];
a[11] = a[7] + b[11];
a[12] = a[8] + b[12];
```

Note that even if the hardware SIMD length is larger than 4 the algorithm only allows to process chunks of size 4 in vectorized way.

Declare SIMD

Traditionally functions in C/C++ accept scalar arguments and return scalar values, which is a bottleneck in vectorizing a loop that involves such functions. Since OpenMP 4.0 it is possible to instruct the compiler to generate specific vector variants of the scalar function.

```
#pragma omp declare simd
int inc(int x)
{
  return x+1;
}
```

A vector variant of this function is then called from a vectorized loop

```
#pragma omp simd
for (int i=0; i<N; i++)
  y[i] = inc(x[i]);
```

Declare SIMD

The `simdlen` clause tells the compiler to SIMDize a function for a predefined vector length. If this clause is missing then the compiler automatically determines the optimal SIMD length from the hardware.

```
#pragma omp declare simd simdlen(4)
int inc(int x)
{
  return x+1;
}
```

The specified vector length should be a multiple of the hardware SIMD length. Otherwise the potential of the hardware is not fully exploited.

Recent version of the `gcc` compiler provide detailed information about vectorization if the `-fopt-info-vec` option is used.

Declare SIMD

Another implementation of the `inc` function is as follows

```
#pragma omp declare simd uniform(x) linear(k:1)
int inc(int *x, int k)
{
  return x[k]+1;
}
```

The `uniform(x)` clause defines that x is an invariant value for the complete chunk. Here, it gives the base address of the array `x`.

The `linear(k:1)` clause defines that index k increases linearly by the stride value 1 within the chunk.

Declare SIMD

Another implementation of the `inc` function is as follows

```
#pragma omp declare simd uniform(x) linear(k:1)
int inc(int *x, int k)
{
  return x[k]+1;
}
```

Performance considerations:

The `uniform` clause directs the compiler to generate code that passes the parameter's value (or address if it is a pointer) via a *scalar register* instead of a vector register.

The `linear` clause for a scalar parameter (or variable) directs the compiler to generate code that loads/stores its value using a *scalar register* instead of a vector register.

Declare SIMD

Another implementation of the `inc` function is as follows

```
#pragma omp declare simd uniform(x) linear(k:1)
int inc(int *x, int k)
{
  return x[k]+1;
}
```

Performance considerations:

In this example, the compiler generates *linear unit-stride memory load/store instructions* to obtain performance gains

SIMD reduction

The `simd` construct can also be combined with a `reduction` clause.

```
int N=1000;
double x[N], y=0;

#pragma omp simd reduction(+:y)
for (int i=0; i<N; i++)
  y += x[i];
```

In this case, the reduction loop is vectorized using SIMD instructions.

Exercise: Matrix-matrix multiplication

Write a parallel implementation of the matrix-matrix-multiplication

$$\boldsymbol{A} \in \mathbb{R}^{N \times M}, \boldsymbol{B} \in \mathbb{R}^{M \times O} \; : \quad \boldsymbol{C} = A \cdot B \in \mathbb{R}^{N \times O}$$

where each entry of matrix $\boldsymbol{C}$ is computed from

$$c_{ik} = \sum_{j=1}^{M} a_{ij} \cdot b_{jk}$$

You can start from the sequential implementation in task *[OpenMP] Exercise 8 : Matrix-Matrix Multiplication*.

Tasks

Fine-grained task-level parallelization is available since OpenMP 3.0.

The *OpenMP tasking* concept allows each thread to generate new tasks once it encounters a `task` construct.

The OpenMP runtime system controls the actual creation and execution of the task. Execution can either be immediate or delayed.

Completion of a task can be enforced through task synchronization.

Tasks are not threads! *Threads* are like workers in a factory and *tasks* are the jobs they have to do. If you generate a new task this means that the next free worker will do the job.

Tasks

What does the following example code print using 2 threads?

```
#pragma omp parallel
{ /* Begin of parallel region */
  #pragma omp single
  { /* Begin of single */
    printf( "A " );

    #pragma omp task
    { printf( "car "); }

    #pragma omp task
    { printf( "race "); }

    #pragma omp taskwait
    printf( "is fun to watch ");
  } /* End of single - implicit barrier */
} /* End of parallel region - implicit barrier */
printf("\n");
```

Taskwait

The `taskwait` construct forces the OpenMP runtime to *wait on the completion of child tasks of the current task*.

In the example, the 'current task' is the *single thread* that executes the `single` region. It will wait until the two child tasks completed printing.

What happens if each of the two tasks generates descendent child tasks?

Taskgroup

The `taskgroup` construct forces the OpenMP runtime to *wait on the completion of child tasks of the current task and their descendent tasks*.

```
#pragma omp parallel
{ /* Begin of parallel region */
  #pragma omp single
  { /* Begin of single */
    printf( "A " );

    #pragma omp taskgroup
    { /* Begin of taskgroup */
      #pragma omp task
      { printf( "car "); }

      #pragma omp task
      { printf( "race "); }

    } /* End of taskgroup - implicit barrier */
    printf( "is fun to watch ");
  } /* End of single - implicit barrier */
} /* End of parallel region - implicit barrier */
printf("\n");
```

Example: Conditional counting

Write a program that generates random numbers between 0 and 1 and counts how many times the random number is smaller than 0.3. Once the program has found 10 random numbers smaller than 0.3 it should return and print the random numbers found.

Hint: Write a function that generates a single random number and stores it into a global array if the random number is smaller than 0.3. The function should generate a new task if less than 10 random numbers have been found so far.

You can start from *[OpenMP] Exercise 9 : Random Number Generator*.

Taskgroup

Loop over tasks in OpenMP 4.0

```
#pragma omp taskgroup
{
  for (int tmp = 0; tmp < 32; tmp++)
#pragma omp task
    for (long l = tmp * 32; l < tmp * 32 + 32; l++)
      do_something (l);
} /* wait for completion of all tasks */
```

This creates 32 tied tasks in a new taskgroup and assigns a fixed number of 32 iterations of `do_something()` to each created task.

Taskloop

Loop over tasks in OpenMP 4.5

```
#pragma omp taskloop num_tasks (32)
  for (long l = 0; l < 1024; l++)
    do_something (l);
/* wait for completion of all tasks */
```

This creates 32 tied tasks in a new taskgroup and lets each task perform one or more iterations of the entire loop.

Implicit taskgroup ensures awaiting of completion of all tasks.

Exercise: Binary search

Implement a program that uses tasks to execute a binary search algorithm in parallel.

Use the sequential implementation of the program as the basis of your program. The sequential program is implemented in task *[OpenMP] Exercise 10 : Binary Search*.

Print the number of the thread that has found the value in the tree.

Hints

The binary tree is *not* sorted; a value can be located in the left or right sub-tree of a node. You must search both sub-trees.

Do not forget to indicate which variables are private and which are shared.

Can you verify that the search operation is executed in parallel?

What happens if you vary the maximum number of threads through the environment variable `OMP_NUM_THREADS`?

Part IV

Threads

Outline

Introduction

Threads enable one to implement parallel programs that are based on the shared memory model.

Like MPI, they provide a programmer with much freedom and control when implementing a parallel program.

On the other hand, threads require a programmer to handle data exchanges and execution synchronisation of on a fairly low level.

With threads you can implement the same parallel algorithms that you can implement with OpenMP (and much more), but it generally requires more effort.

What is a thread?

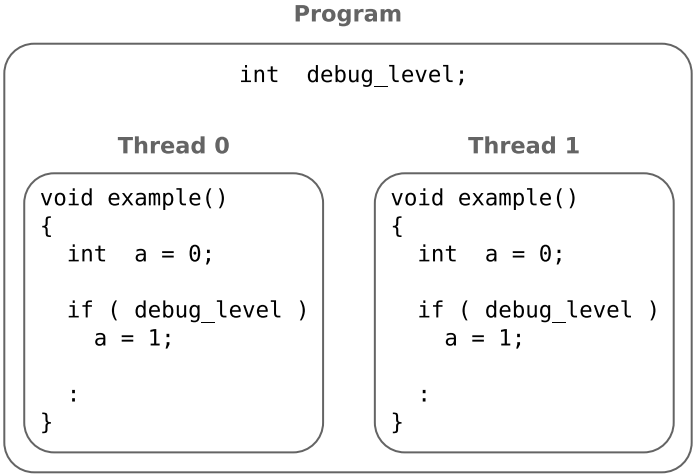
A thread can be viewed as a light-weight process.

That is, a thread executes its own stream of instructions and has its own *stack* on which all local function variables are allocated.

In contrast to a process, all global variables are shared between the threads.

In fact, a thread can access all memory allocated to the program, including variables on the stack of another thread if it knows their memory addresses.

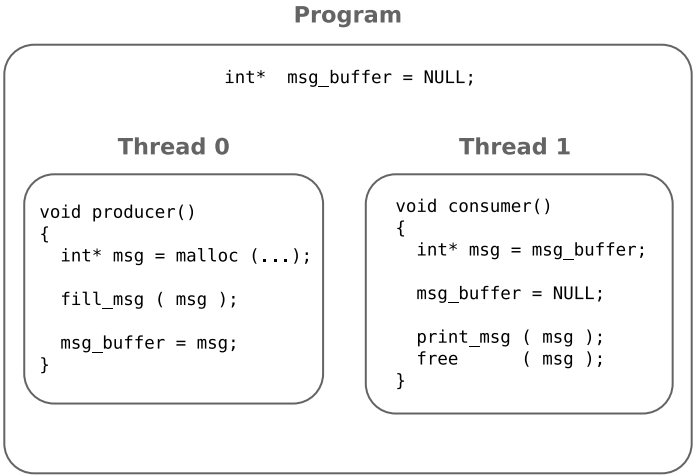
What is a thread?



Each thread has its own variable `a`, but they share the global variable `debug_level`.

Exchanging data between threads

Threads can exchange data by writing to and reading from the same memory location(s).



Exchanging data between threads

That looks easy! Too easy in fact ...

Unfortunately there is no such thing as a free lunch.

How do you make sure that the consumer thread reads the data after the producer thread has made those data available?

The solution is to use a so-called *mutex* and *condition* variable to synchronise the execution of the two threads; see the next slide.

Mutexes and condition variables are essential thread synchronisation primitives. More about those later.

Exchanging data between threads

Program

```
int*         msg_buffer = NULL;
condition_t  msg_cond;
mutex_t      msg_mutex;
```

Thread 0

```
void producer()
{
  int* msg = malloc (...);

  fill_msg ( msg );
  lock     ( msg_mutex );

  msg_buffer = msg;

  signal   ( msg_cond );
  unlock   ( msg_mutex );
}
```

Thread 1

```
void consumer()
{
  int* msg = NULL;

  lock ( msg_mutex );

  if ( ! msg_buffer )
    wait ( msg_cond );

  msg = msg_buffer;
  msg_buffer = NULL;

  unlock    ( msg_mutex );
  print_msg ( msg );
  free      ( msg );
}
```

Managing threads

Threads are typically managed by the operating system kernel.

That is, the operating system provides a collection of system calls for creating and destroying threads, and for synchronising the execution between threads.

Unfortunately, there is no standard library – at least not for C – for managing threads. This makes it more difficult to write portable programs that use threads.

Fortunately, all operating systems provide similar concepts for managing threads. This means that porting a program from one operating system to another is rather straightforward.

The POSIX threads library

On Linux and macOS you can use the POSIX threads library to manage threads. This is a system-level library with a well-defined application programming interface.

Windows does not implement the POSIX threads API natively, but there are various libraries that implement the POSIX threads API on top of the native Windows threads API.

Using such a compatibility library can simplify porting a program from one platform to another.

The POSIX threads library

This course focuses on the POSIX threads library.

Most (or all) concepts that are explained are certainly not limited to the POSIX thread library. When using another thread library the details may be different, but the general principles will be the same.

In fact, all concepts are applicable to other programming languages, like Java, too. This is not surprising because each programming language must use the same system-level services at the lowest level.

In C you can use those system-level services directly; there are no abstraction layers if you do not want those.

Working with threads

To use the POSIX thread library you need to include the header file `<pthread.h>` in your source code.

This header file provides declarations of functions, data types, constants and macros that can be used to start, end, and synchronise threads.

All names exported by the POSIX thread library start with the prefix `pthread_` or, in the case of macro names, `PTHREAD_`.

You must link your program with the `pthread` library.

Error codes

All POSIX thread functions (with a few exceptions) return an integer error code to indicate success or failure.

By convention, a zero return value indicates success and a non-zero return value indicates failure. The specific return value indicates what kind of error occurred.

A robust program should check the return values and take action when an error occurs.

The examples shown in these slides will ignore the return values for the sake of brevity.

Thread IDs

Each POSIX thread has an identifier of the type `pthread_t` which is an integral type.

In contrast to MPI and OpenMP, threads are *not* numbered in a linear way; the thread identifier can be an arbitrary number.

Each thread within a process has a unique thread identifier.

A thread can obtain its own thread identifier by calling the function `pthread_self`.

This is one of the few POSIX thread functions that does not return an error code.

Main thread

When a process starts, it consist of one thread that is called the *main thread*. This is the thread that starts the execution of the `main` function.

The main thread can create sub-threads or *child threads*. The main thread is said to be the *parent thread* of its child threads.

The child threads can recursively create child threads. That is, a child thread can be the parent of its own child threads.

Starting and ending threads

The functions `pthread_create` and `pthread_join` can be used to create a thread and wait for a thread to finish, respectively.

Here is an example:

```
pthread_t  tid;

pthread_create ( &tid, NULL, thread_func, NULL );
pthread_join   (  tid, NULL );
```

Note that the function `thread_func` (not shown here) is the starting point of the thread.

Starting threads

```
int pthread_create ( pthread_t* tid, const pthread_attr_t* attr,
                     void *(*func) (void*), void* arg );
```

`tid`	OUT	The identifier of the new thread.
`attr`	IN	A pointer to an attribute object indicating how the thread is to be created, or `NULL`.
`func`	IN	A pointer to the thread entry point.
`arg`	IN	A pointer to data to be passed to the thread entry point.

The function `pthread_create` creates a new thread that will start executing the function specified in the `func` parameter.

The identifier of the new thread will be returned in the variable pointer to the `tid` parameter. The calling thread can use the identifier to interact with the thread later.

Starting threads

The parameter `attr` can be used to pass a so-called *attribute* specifying various details associated with the creation of threads.

This can be useful for advanced application; it will be ignored here. We will pass a `NULL` pointer to indicate that defaults settings are to be used.

Initial data for the thread can be passed by means of the `arg` parameter that may point to an object of an arbitrary type. This parameter will be passed as an argument to the thread function `func`. More about this later.

The next slide shows an example involving the `pthread_create` functions.

Starting threads example

```
void* thread_func ( void* arg )
{
  pthread_t  self = pthread_self ();

  printf ( "I am thread %ld\n", (long) self );

  return NULL;
}

int main ()
{
  pthread_t  tid;

  pthread_create ( &tid, NULL, thread_func, NULL );

  return 0;
}
```

Ending threads

A thread ceases to exist when it returns from its thread start function (or thread entry point).

The parent thread – the one that created the thread – normally has to wait until its child thread has ended.

The parent thread, for instance, needs to collect a result produced by the child thread.

The parent thread can call the function `pthread_join` to wait for the child thread to end.

Ending threads

```
int pthread_join ( pthread_t tid, void** retval );
```

`tid`	IN	The identifier of the thread to be joined.
`retval`	OUT	A pointer to a pointer to an optional return object, or `NULL`.

The function `pthread_join` waits until the thread identified by the parameter `tid` has ended.

If the parameter `retval` is not `NULL`, it will be set to the value that the child thread has returned from its thread function. More about this later.

Ending threads example

```
void* thread_func ( void* arg );

int main ()
{
  pthread_t  tid;

  pthread_create ( &tid, NULL, thread_func, NULL );
  pthread_join   ( tid, NULL );

  return 0;
}
```

When the parent thread does not call `pthread_join`, it can not know when the child thread has ended.

Ending threads example

```
void* thread_func ( void* arg )
{
  /* Do important computation here ... */

  return NULL;
}

int main ()
{
  pthread_t  tid;

  pthread_create ( &tid, NULL, thread_func, NULL );

  return 0;

  /* Oops, child thread killed before it has completed its
     computation. */
}
```

Ending threads example

When the main thread exits, all its child threads will be terminated.

To be precise, when the main thread exits, the process is terminated. All threads that are part of the process are terminated too.

You must use the function `pthread_join` if your want to ensure that a child thread ends before its parent thread.

Or you must roll your own synchronisation mechanism with mutexes and condition variables. More about those later.

Exercise: hello world

Implement a program that spawns eight threads, each of which prints its identifier three times.

Use the function `sleep` to pause one second after each printed line:

```
for ( i = 0; i < 3; i++ ) {
  printf ( "I am thread ...\n" );
  sleep  ( 1 );
}
```

Use *[Threads] Exercise 1 : Hello World* as the starting point of your program.

Passing data to threads

The thread start function must define a `void` pointer parameter that can be used to pass data from a parent thread to a child thread.

Example:

```
void* thread_func ( void* arg )
{
  printf ( "Number passed: %d.\n", *((int*) arg) );
  return NULL;
}

int main ()
{
  int        num = 10;
  pthread_t  tid;

  pthread_create ( &tid, NULL, thread_func, &num );
  pthread_join   (  tid, NULL );
  return 0;
}
```

Passing data to threads

The parent thread must make sure that the data exist for as long as the child thread may use it.

The next two slides show how (not) to pass data to a child thread.

How not to pass data to a thread

```
pthread_t spawn_thread ()
{
  int        num = 10;
  pthread_t  tid;

  pthread_create ( &tid, NULL, thread_func, &num );
  return tid;
}

int main ()
{
  pthread_t  tid = spawn_thread ();

  pthread_join ( tid, NULL );
  return 0;
}
```

What is wrong here?

How to pass data to a thread correctly

```
void* thread_func ( void* arg )
{
  int*  num = (int*) arg;

  printf ( "Number passed: %d.\n", *num );
  free   ( num );
  return NULL;
}

pthread_t spawn_thread ()
{
  int*       num = malloc ( sizeof(int) );
  pthread_t  tid;

  *num = 10;

  pthread_create ( &tid, NULL, thread_func, num );
  return tid;
}
```

Retrieving data from threads

The function `pthread_join` can return data from a thread through its second parameter.

If this parameter is not `NULL`, it will be set to the pointer returned from the thread start function.

You must make sure that the object that is pointed to exists until it has been used by the parent thread.

The next two slides show how (not) to retrieve data from a child thread.

How to retrieve data from a thread

```
void* thread_func ( void* arg )
{
  int*  result = malloc ( sizeof(int) );

  *result = 42;
  return result;
}

int main ()
{
  void*      result;
  pthread_t  tid;

  pthread_create ( &tid, NULL, thread_func, NULL );
  pthread_join   (  tid, &result );
  printf         ( "Result is %d.\n", *((int*) result) );
  free           ( result );
  return 0;
}
```

How not to retrieve data from a thread

```
void* thread_func ( void* arg )
{
  int  result = 42;

  return &result;
}

int main ()
{
  void*      result;
  pthread_t  tid;

  pthread_create ( &tid, NULL, thread_func, NULL );
  pthread_join   (  tid, &result );
  printf         ( "Result is %d.\n", *((int*) result) );
  return 0;
}
```

What is wrong here?

Exercise: thread rank

Implement a program in which the main threads spawns a number of child threads.

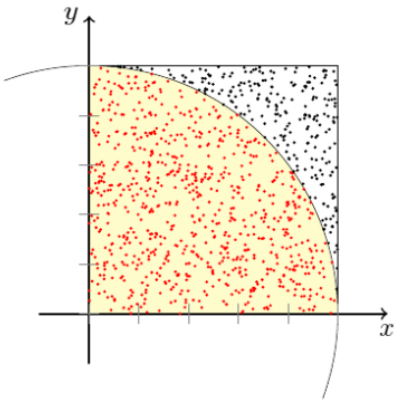
The main thread should pass an integer rank – ranging from zero to the number of threads minus one – to each child thread.

Each child thread should print its rank.

Use *[Threads] Exercise 2 : Rank* as starting point of your program.

Exercise: parallel computation of π

Compute an approximate value of π in parallel by means of the Monte Carlo method: pick N points at random inside a unit square and count the number of points M that are inside a unit circle; $\pi \approx 4M/N$.



Use *[Threads] Exercise 3 : Compute PI* as the starting point of your program.

Hints

Use the `conf_t` data type to obtain the number of iterations N and the number of threads from the command-line arguments passed to the program.

Call the function `parse_args` to parse the command line arguments.

Pass the `conf` object to each thread so that it knows how many iterations it should execute.

Use `malloc` to allocate an array of thread identifiers in the main thread.

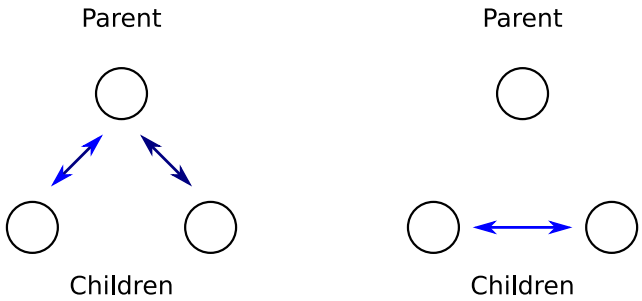
Test your program with the command-line arguments:

```
--niter 40000 --nthreads 4
```

Synchronisation primitives

The previous slides have shown how to pass data between threads in a vertical way. That is, between parent and child threads.

Most non-trivial programs also need to pass data between threads in a horizontal way. That is, between sibling threads.



Synchronisation primitives

Horizontal data exchanges require synchronisation between threads to:

- avoid erroneous results;
- ensure the consistency of data structures;
- order events in time.

There are two kinds of *synchronisation primitives* with which one can handle (almost) all situations: *mutexes* and *condition variables*.

If necessary one can build more sophisticated synchronisation objects by combining mutexes and condition variables.

Mutexes

A mutex is an object of type `pthread_mutex_t` that provides *mutual exclusive* access to data.

A mutex can have two states: locked and unlocked. It is unlocked by default.

Only one thread can lock a mutex at any time.

If two threads try to lock a mutex at the same time, only one will succeed. The other thread will wait until the mutex is unlocked.

Race conditions

Mutexes are primarily used to avoid so-called *race conditions* in which two threads try to make changes to the same memory location(s) at the same time.

This can lead to erroneous results and/or inconsistent data structures.

To illustrate this, consider a program in which two threads generate two sequences of uniform random numbers. Whenever they find a number smaller than ϵ they increment a shared counter.

Race condition example

Program

```
int  count = 0;
```

Thread 0

```
void* thread_func(void*)
{
  double x;

  while ( 1 )
  {
    x = next_random ();

    if ( x < EPS )
      count++;
  }
}
```

Thread 1

```
void* thread_func(void*)
{
  double x;

  while ( 1 )
  {
    x = next_random ();

    if ( x < EPS )
      count++;
  }
}
```

Race condition example

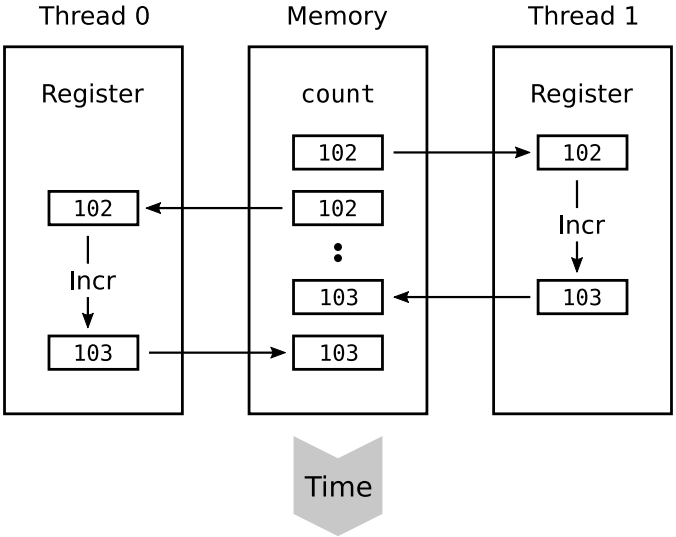
When two threads try to update the global variable `count` at (almost) the same time, the result will be incorrect.

This is what happens:

1. both threads fetch the value of `count` from memory and store it in a CPU register;
2. both threads increment the value in the CPU register;
3. both threads store the new value from the register into the memory location associated with `count`.

Outcome: one increment has been missed.

Race condition example



Race condition example

The solution is to use a mutex:

```
pthread_mutex_t  mutex;
int              count = 0;

void* thread_func ( void* )
{
    :

  if ( x < EPS )
  {
    pthread_mutex_lock   ( &mutex );
    count++;
    pthread_mutex_unlock ( &mutex );
  }

  return NULL;
}
```

Using mutexes

In contrast to MPI, mutexes – and condition variables too – are *not* managed by the POSIX thread library.

You must manage mutex variables yourself by defining mutex variables or by embedding them in your structures.

When a mutex is instantiated, it has an undefined state. You must call the function `pthread_mutex_init` to initialise the mutex; this must be done before it is used.

The functions `pthread_mutex_lock` and `pthread_mutex_unlock` can be used to lock and unlock a mutex.

The function `pthread_mutex_destroy` must be called to destroy a mutex when it is no longer needed.

Initialising a mutex

```
int pthread_mutex_init ( pthread_mutex_t* mutex,
                         const pthread_mutexattr_t* attr );
```

`mutex`	OUT	A pointer to the mutex to be initialised.
`attr`	IN	A pointer to mutex attributes or `NULL`.

The function `pthread_mutex_init` initialises a mutex. It will assign initial values to the fields that are part of the `pthread_mutex_t` type.

It may also allocate additional data structures that are associated with the mutex.

Initialising a mutex

The parameter `attr` can be used to specify various properties of the mutex. When `NULL`, default values are used. The `attr` parameter will be ignored here.

The function `pthread_mutex_init` must be called *before* a mutex is used.

```
void bad_idea ()
{
  pthread_mutex_t  mutex;

  pthread_mutex_lock ( &mutex );  /* Error: not yet initialised. */
}
```

Destroying a mutex

```
int pthread_mutex_destroy ( pthread_mutex_t* mutex );
```

`mutex`	INOUT	A pointer to the mutex to be destroyed.

The function `pthread_mutex_destroy` destroys a mutex. It must be called when a mutex is no longer used.

This function may deallocate any data structures that are associated with the mutex.

In well-formed program, calls to `pthread_mutex_init` are always balanced by calls to `pthread_mutex_destroy`.

Mutex initialisation example

```
pthread_mutex_t  mutex;
int              count = 0;

void* thread_func ( void* );  /* Not shown here */

int main ()
{
  pthread_mutex_init ( &mutex, NULL );

  /* Spawn threads using the mutex ... */
  /* Join threads ... */

  pthread_mutex_destroy ( &mutex );

  return 0;
}
```

Locking a mutex

```
int pthread_mutex_lock ( pthread_mutex_t* mutex );
```

`mutex`	INOUT	A pointer to the mutex to be locked.

The function `pthread_mutex_lock` locks a mutex. Only one thread can lock a given mutex at a time.

A thread will be blocked if it calls this function while the mutex has been locked.

When this function returns, the current thread has locked the mutex.

Unlocking a mutex

```
int pthread_mutex_unlock ( pthread_mutex_t* mutex );
```

`mutex`	INOUT	A pointer to the mutex to be unlocked.

The function `pthread_mutex_unlock` unlocks a mutex. The calling thread must have locked the mutex with a call to `pthread_mutex_lock`.

In a well-formed program each call to `pthread_mutex_lock` is balanced by a call to `pthread_mutex_unlock`.

Recursive locking of a mutex

A mutex can *not* be locked recursively by the same thread.

That is, a thread will block forever if it calls `pthread_mutex_lock` for a second time without calling `pthread_mutex_unlock` in between.

```
pthread_mutex_t  mutex;

void* thread_func ( void* arg )
{
  pthread_mutex_lock ( &mutex );
  pthread_mutex_lock ( &mutex );  /* Oops, deadlock */

  return NULL;
}
```

Mutex scope

A mutex can provide mutual exclusive access to a single variable, multiple variables or even larger data structures.

The data that are *protected* by a mutex are said to form the *scope* of the mutex.

In C there is no formal way to specify the scope of a mutex in the source code; the only way to determine the scope of a mutex is to inspect all code that can be executed when the mutex is locked.

Mutex scope

To make your code more readable you should indicate as clear as possible what the scope is of a mutex.

Ways to achieve this:

- declare the mutex right next to the variables it protects;
- link the name of mutex and the variables it protects;
- combine the mutex and the variables it protects in a single `struct`;
- define functions for accessing the data protected by the mutex;
- indicate the mutex scope by means of comments.

How to indicate the scope of a mutex

Declare the mutex next to the variables it protects:

```
pthread_mutex_t  mutex;
int              error;
double           result;
```

Link the name of the mutex to the names of the variables it protects:

```
pthread_mutex_t  solver_mutex;
int              solver_error;
double           solver_result;
```

How to indicate the scope of a mutex

Define functions for accessing the data protected by the mutex:

```
void   set_error  ( int err );
int    get_error  ();
void   set_result ( double res );
double get_result ();
```

Indicate the mutex scope by means of comments:

```
pthread_mutex_t  mutex;  /* This mutex protects the variables */
int              error;  /* "error" and "result".             */
double           result;
```

Mutex granularity and lock contention

A mutex with a large scope is said to be a *coarse-grain* mutex.

A mutex with a small scope, on the other hand, is said to be a *fine-grain* mutex.

The granularity of the mutexes in a program have a large effect on the parallel scalability of the program.

Coarse-grain mutexes tend to result in *lock contention* when threads are regularly waiting on a mutex.

Coarse-grain mutexes

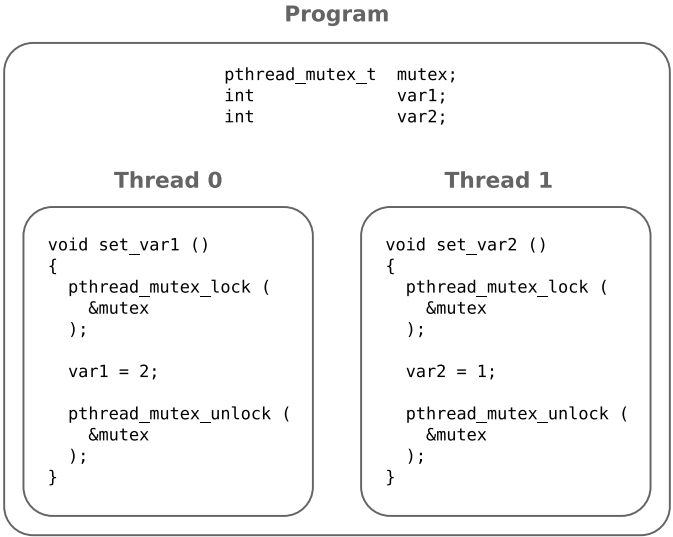
To illustrate this, consider a program in which all shared data are protected by a single mutex.

Whenever a thread needs to access one shared variable it must lock that mutex.

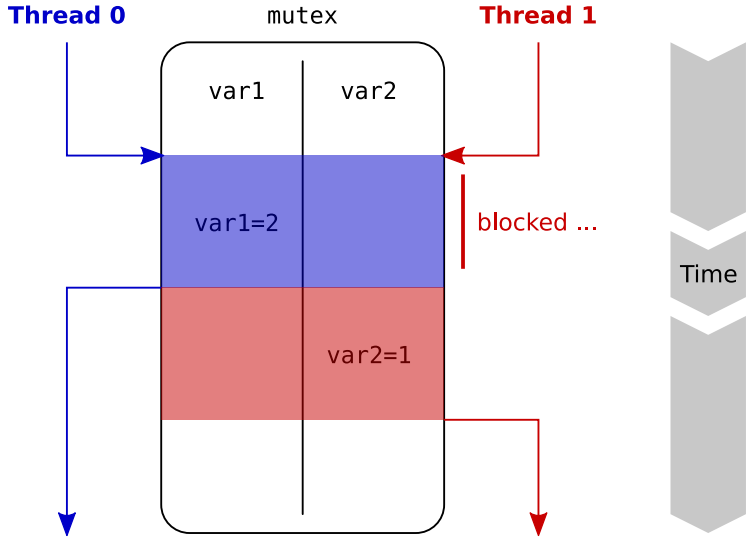
This will block all other threads that need to access a shared variable, even when those variables are not related to each other.

In other words, even when threads need to access unrelated, shared variables, they frequently have to wait; the coarse-grain mutex becomes a contended resource.

Coarse-grain mutex example



Coarse-grain mutex example

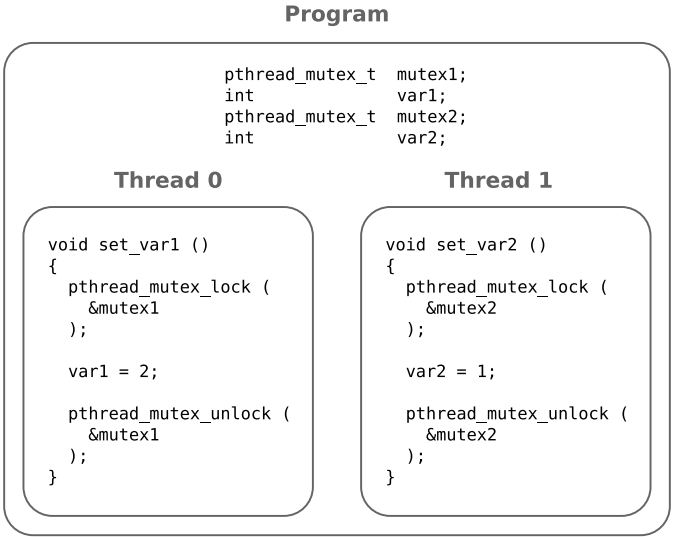


Fine-grain mutexes

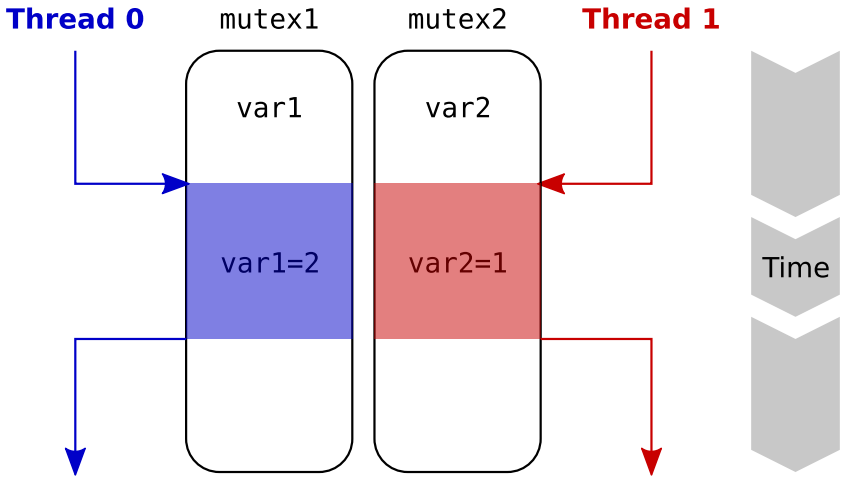
Lock contention can often be reduced by breaking a single coarse-grain mutex into multiple fine-grain mutexes that protect different (unrelated) variables.

In this way different threads can access different shared variables without blocking each other.

Fine-grain mutex example



Fine-grain mutex example



Fine-grain vs coarse-grain mutexes

There is trade off between fine-grain and coarse-grain mutexes.

Fine-grain mutexes can result in better performance, but make the code harder to understand. They also increase the possibilities for introducing subtle bugs involving race conditions.

Advice: use as coarse-grain mutexes as possible as long as lock-contention does not have a significant impact on performance.

Sharing mutexes

A mutex is useful only if it is shared between multiple threads.

One way to achieve this is to declare a mutex as a global variable, along with the other global variables that are protected by the mutex.

This is a simple approach, but one needs to be careful to make clear which mutex protects which variables.

Sharing mutexes through structures

Another approach is to combine the mutex and the variables it protects in one `struct`.

This has several advantages:

- it is clear from the code which variables are protected by which mutex;
- the `struct` can be allocated dynamically or on the stack, avoiding global variables;
- multiple instances of the `struct` can be created.

Sharing mutexes example (1/2)

```
typedef struct shared_count {
  pthread_mutex_t  mutex;
  int              count;
} shared_count_t;

void* thread_func ( void* arg )
{
  shared_count_t*  count = (shared_count_t*) arg;

  pthread_mutex_lock   ( &count->mutex );
  count->count++;
  pthread_mutex_unlock ( &count->mutex );

  return NULL
}

/* Continued on next slide ... */
```

Sharing mutexes example (2/2)

```
int main ()
{
  shared_count_t  count;
  pthread_t       threads[4];
  int             i;

  count.count = 0;
  pthread_mutex_init ( &count.mutex, NULL );

  for ( i = 0; i < 4; i++ )
    pthread_create ( &threads[i], NULL, thread_func, &count );

  for ( i = 0; i < 4; i++ )
    pthread_join   (  threads[i], NULL );

  printf ( "Final count: %d.\n", count.count );
  pthread_mutex_destroy ( &count.mutex );

  return 0;
}
```

Exercise: parallel computation of π

Modify the program that computes π in parallel from the previous exercise so that it uses a shared variable to keep track of the number of points within a unit circle. Use *[Threads] Exercise 4 : Compute PI*.

That is, instead of defining a counter per thread, define one shared counter.

Use a mutex to avoid race conditions.

Is this new implementation more efficient than the previous one?

Condition variables: introduction

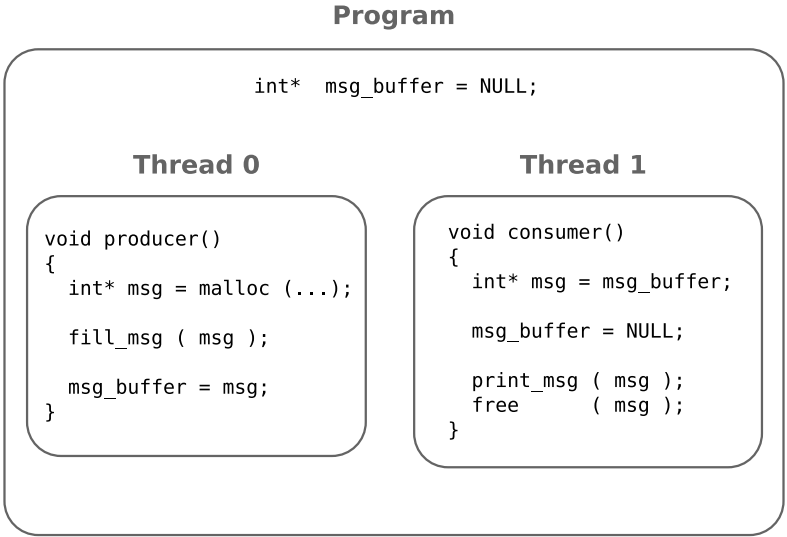
In any non-trivial (parallel) program there are dependencies between data and/or operations.

That is, an operation requires input data that are the result of another operations.

When two dependent operations are executed by different threads, one thread will have to signal the other thread that the required input data are available.

Let's go back to a previous example.

Condition variables: introduction



Condition variables: introduction

How to make sure that the "consumer" thread only prints the message after the "producer" thread has created that message?

One approach would be to use a mutex to avoid race conditions and continuously poll for a new message.

This is shown on the next two slides.

Wait by polling (1/2)

```
pthread_mutex_t  msg_mutex;
int*             msg_buffer = NULL;

void producer ()
{
  int*  msg = malloc ( 10 * sizeof(int) );

  fill_msg             ( msg );
  pthread_mutex_lock   ( &msg_mutex );

  msg_buffer = msg;

  pthread_mutex_unlock ( &msg_mutex );
}

/* Continued on next slide ... */
```

Wait by polling (2/2)

```
void consumer ()
{
  int*  msg = NULL;

  while ( ! msg )
  {
    pthread_mutex_lock   ( &msg_mutex );

    msg        = msg_buffer;
    msg_buffer = NULL;

    pthread_mutex_unlock ( &msg_buffer );
  }

  print ( msg );
  free  ( msg );
}
```

Condition variables: introduction

This works, but it has two drawbacks:

- it results in lock contentions because the consumer thread is likely to hold a mutex lock when the producer thread is trying to lock the mutex;
- it keeps one processor core busy while the consumer thread is polling a message.

The POSIX thread library provides a better solution in the form of *condition variables*.

Condition variables: introduction

A condition variable, which has type `pthread_cond_t`, enables a thread to "sleep" until it is signalled by another thread that some particular condition has been fulfilled.

This is more efficient than polling because it lowers lock contention and it enables the processor core to do something useful while a thread is waiting.

The next two slides show how to implement the producer-consumer example with a condition variable.

Wait by using a condition variable (1/2)

```
pthread_mutex_t  msg_mutex;
pthread_cond_t   msg_cond;
int*             msg_buffer = NULL;

void producer ()
{
  int*  msg = malloc ( 10 * sizeof(int) );

  fill_msg             ( msg );
  pthread_mutex_lock   ( &msg_mutex );

  msg_buffer = msg;

  pthread_cond_signal  ( &msg_cond );
  pthread_mutex_unlock ( &msg_mutex );
}

/* Continued on next slide ... */
```

Wait by using a condition variable (2/2)

```
void consumer ()
{
  int*  msg = NULL;

  pthread_mutex_lock ( &msg_mutex );

  while ( ! msg_buffer )
    pthread_cond_wait ( &msg_cond, &msg_mutex );

  msg        = msg_buffer;
  msg_buffer = NULL;

  pthread_mutex_unlock ( &msg_mutex );

  print ( msg );
  free  ( msg );
}
```

Condition variables: introduction

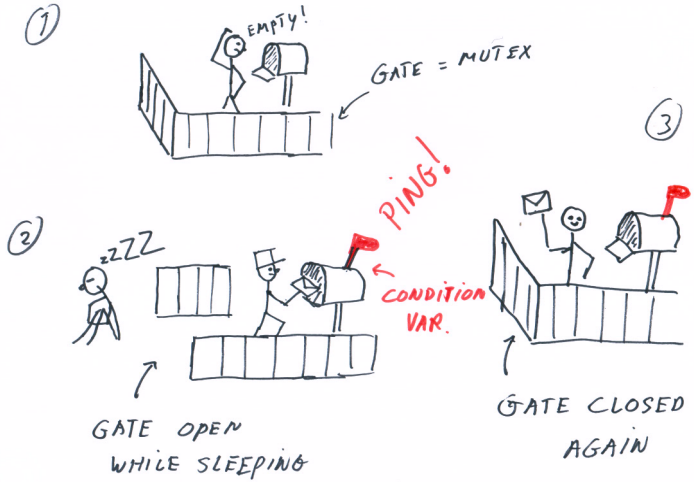
As shown in the example, a condition variable itself has no explicit state; you can only use it to let a thread wait until it is signalled.

You must use some other variable – or multiple variables – to keep track of the condition that is to be monitored.

In the previous example the pointer `msg_buffer` stores the condition that is monitored: a non-`NULL` pointer signals a new message.

Typically you have one condition variable for each condition that needs to be signalled between threads.

Condition variables: cartoon



Condition variables and mutexes

A condition variable must always be paired with a mutex protecting the variable that stores the actual condition. In the previous example this is the variable `msg_buffer`.

The mutex may also protect additional variables that are shared between threads.

A condition variable may only be signalled and waited on when its associated mutex has been locked.

It is your responsibility to make sure that this is the case, and that the signalling and waiting thread have locked the same mutex.

In short: whenever you define a condition variable, you must define an associated mutex.

Using condition variables

To use a condition variable, you must define a variable of type `pthread_cond_t`. It makes sense to define a condition variable right next to its associated mutex.

You can use similar names to indicate that they are related:

```
pthread_mutex_t  msg_mutex;
pthread_cond_t   msg_cond;
```

Or embed them in the same `struct`:

```
struct shared_data
{
   pthread_mutex_t  mutex;
   pthread_cond_t   cond;
};
```

Using condition variables

When a condition variable is instantiated, it has an undefined state. You must call the function `pthread_cond_init` to initialise the condition variable; this must be done before it is used.

The functions `pthread_cond_wait` and `pthread_cond_signal` can be used to wait on a condition variable and to wake up a waiting thread.

The function `pthread_cond_destroy` must be called to destroy a condition when it is no longer needed.

Initialising a condition variable

```
int pthread_cond_init ( pthread_cond_t* cond,
                        const pthread_condattr_t* attr );
```

`cond`	OUT	A pointer to the condition variable to be initialised.
`attr`	IN	A pointer to condition attributes or `NULL`.

The function `pthread_cond_init` initialises a condition variable. It will assign initial values to the fields that are part of the `pthread_cond_t` type.

It may also allocate additional data structures that are associated with the condition variable.

Initialising a condition variable

The parameter `attr` can be used to specify various properties of the condition variable. When `NULL`, default values are used. The `attr` parameter will be ignored here.

Your program is ill-formed if you forget to initialise a condition variable before you use it.

This is a fancy way of saying that bad things will happen if you make a mistake.

Destroying a condition variable

```
int pthread_cond_destroy ( pthread_cond_t* cond );
```

`cond`	INOUT	A pointer to the condition variable to be destroyed.

The function `pthread_cond_destroy` destroys a condition variable. It must be called when a condition variable is no longer used.

This function may deallocate any data structures that are associated with the condition variable.

In well-formed program, calls to `pthread_cond_init` are always balanced by calls to `pthread_cond_destroy`.

Condition variable initialisation example

```
pthread_mutex_t  msg_mutex;
pthread_cond_t   msg_cond;
int*             msg_buffer = NULL;

void* thread_func ( void* );  /* Not shown here */

int main ()
{
  pthread_mutex_init ( &msg_mutex, NULL );
  pthread_cond_init  ( &msg_cond,  NULL );

  /* Spawn threads using the mutex and condition variable ... */
  /* Join threads ... */

  pthread_mutex_destroy ( &msg_mutex );
  pthread_cond_destroy  ( &msg_cond );

  return 0;
}
```

Signalling a condition variable

```
int pthread_cond_signal ( pthread_cond_t* cond );
```

`cond`	IN	A pointer to the condition variable to be signalled.

The function `pthread_cond_signal` wakes up *at least one* thread that is waiting on the specified condition variable.

It does nothing if no thread is waiting on the condition variable.

Typically, the thread calling `pthread_cond_signal` will have locked the mutex associated with the condition variable, but that is not required. Be careful not to introduce subtle race conditions, however!

Signalling example

This is the typical use of `pthread_cond_signal`:

```
void producer ()
{
  int*  msg = new_message ();

  pthread_mutex_lock   ( &msg_mutex );

  msg_buffer = msg;

  pthread_cond_signal  ( &msg_cond );
  pthread_mutex_unlock ( &msg_mutex );
}
```

Signalling example

This implementation is also valid:

```
void producer ()
{
  int*  msg = new_message ();

  pthread_mutex_lock   ( &msg_mutex );

  msg_buffer = msg;

  pthread_mutex_unlock ( &msg_mutex );
  pthread_cond_signal  ( &msg_cond );
}
```

You must make sure that a waiting thread will never miss a signal.

Waiting on a condition variable

```
int pthread_cond_wait ( pthread_cond_t*  cond,
                        pthread_mutex_t* mutex );
```

`cond`	IN	A pointer to the condition variable on which to wait.
`mutex`	INOUT	A pointer to the mutex associated with the condition variable.

The function `pthread_cond_wait` will block the calling thread until it is woken up by another thread that calls `pthread_cond_signal` on the same condition variable.

When calling this function, the mutex pointer to the parameter `mutex` must have been locked by the calling thread.

When this function returns the mutex it will still be locked by the calling thread; it must be unlocked with a call to `pthread_mutex_unlock`.

Waiting example

```
void consumer ()
{
  int*  msg = NULL;

  pthread_mutex_lock  ( &msg_mutex );

  if ( ! msg_buffer )
    pthread_cond_wait ( &msg_cond, &msg_mutex );

  msg        = msg_buffer;
  msg_buffer = NULL;

  pthread_mutex_unlock ( &msg_mutex );
}
```

Atomic unlock

A call to the function `pthread_cond_wait` will *atomically* unlock the mutex before the calling thread is suspended.

That is, the POSIX thread library guarantees that the calling thread will not miss a signal while the mutex is unlocked.

The mutex must be unlocked while the thread is suspended so that another thread is able to lock the mutex and signal the waiting thread.

When the suspended thread is woken up, it will lock the mutex before it returns from the function `pthread_cond_wait`.

Atomic unlock

Consider this sequence of events:

1. The consumer thread locks the mutex and notices that there is no message.
2. The producer thread tries to lock the mutex but is blocked.
3. The consumer thread waits and unlocks the mutex.
4. The producer thread locks the mutex and signals the consumer thread.

Atomic unlock

If the wait and unlock would not be atomic, then there would be a small window in which the producer thread would be able to signal the condition variable before the consumer thread is suspended.

The consumer thread would miss the wakeup signal and remain suspended forever.

Note here that a signal is not persistent; it will only have effect if a thread is waiting on a condition variable when `pthread_cond_signal` is called.

If no thread is waiting at that moment, the function call will have no effect at all; the condition variable will not "remember" that it has been signalled.

Spurious wakeup

A thread waiting on a condition variable might be woken up while it was not signalled by another thread.

This is called *spurious wakeup* and has to do with low-level signal handling within the operating system.

To make your code robust, you should check for a condition in a loop instead of in an if-statement.

Spurious wakeup

This is not safe when a spurious wakeup occurs:

```
if ( ! msg_buffer )
  pthread_cond_wait ( &msg_cond, &msg_mutex );
```

This is a more robust way to wait on a condition variable:

```
while ( ! msg_buffer )
  pthread_cond_wait ( &msg_cond, &msg_mutex );
```

Exercise: FIFO between threads

Implement a program in which two threads pass data through a FIFO (First In, First Out) buffer.

That is, one thread – the producer – reads characters from the standard input and puts those characters in the FIFO buffer.

The other thread – the consumer – reads characters from the FIFO buffer and writes them to the standard output.

The producer and the consumer threads should end when they encounter the end of the input stream.

Hints

The main thread initialises the FIFO buffer and spawns both the producer and the consumer thread. It then waits until those threads have finished executing.

Use *[Threads] Exercise 5 : FIFO* as the starting point of your program.

Use the function `getc` to read the next character from the standard input, and use the function `putc` to write a character to the standard output.

Hints

Define a `struct fifo_t` that represents the FIFO buffer. It must contain a mutex, a condition variable and space for storing at least one character. It must also contain one or two integer variables indicating whether the buffer is full or empty.

To simplify your code, you can use a buffer of one character.

If you want to make your code more elegant you can define the functions `fifo_init` and `fifo_destroy` that initialise and destroy a buffer, respectively.

The main thread should create an instance of the FIFO buffer and pass its address to both child threads.

Producer thread

This is essentially what the producer thread should do:

```
void* producer ( void* arg )
{
  fifo_t*  fifo = (fifo_t*) arg;
  int      ch   = getc ( stdin );

  while ( 1 )
  {
    /* Wait until buffer is empty ... */
    /* Store the character ch in the buffer ... */

    if ( ch == EOF )
      break;

    ch = getc ( stdin );
  }

  return NULL;
}
```

Consumer thread

This is essentially what the consumer thread should do:

```
void* consumer ( void* arg )
{
  fifo_t*  fifo = (fifo_t*) arg;
  int      ch;

  while ( 1 )
  {
    /* Wait until there is data in the buffer ... */
    /* Read the character ch from the buffer ... */

    if ( ch == EOF )
      break;

    putc ( ch, stdout );
  }

  return NULL;
}
```

Broadcasting a condition

The function `pthread_cond_signal` wakes at least one thread that is waiting on a condition variable.

That is, the POSIX thread library only guarantees that one thread will wake up, although more than one *may* be woken up.

If multiple threads are waiting on the same condition variable, then you must ensure that the other threads are woken up. If you do not do that you will have a (possible) deadlock situation.

Broadcasting a condition

If you use `pthread_cond_signal` then a woken thread must call `pthread_cond_signal` again to wake up the next thread.

The next slide shows an example in which `nthreads` call a function `barrier_wait` that will block the calling thread until it has been called by the last thread.

The last thread wakes up one waiting thread, which, in turn wakes up the next thread. This continues until all threads have been woken up.

Broadcasting a condition example

```
pthread_mutex_t  mutex;
pthread_cond_t   cond;
int              nthreads;

void barrier_wait ()
{
  pthread_mutex_lock ( &mutex );

  if ( (--nthreads) > 0 )
  {
    /* Wait for the last thread. */
    pthread_cond_wait ( &cond, &mutex );
  }

  /* Wake up the next thread. */
  pthread_cond_signal  ( &cond );
  pthread_mutex_unlock ( &mutex );
}
```

Broadcasting a condition

While this works, it is not the most efficient solution because all the calls to `pthread_cond_signal` involve a non-trivial amount of overhead.

Fortunately, the POSIX library provides a better solution:

```
int pthread_cond_broadcast ( pthread_cond_t* cond );
```

`cond`	IN	A pointer to the condition variable to be signalled.

This function will wake up *all* threads waiting on a condition variable.

Using this function the previous example can be implemented more efficiently as shown on the next slide.

Broadcasting a condition example

```
pthread_mutex_t  mutex;
pthread_cond_t   cond;
int              nthreads;

void barrier_wait ()
{
  pthread_mutex_lock ( &mutex );

  if ( (--nthreads) == 0 )
    /* This is the last thread; wake up all other threads. */
    pthread_cond_broadcast ( &cond );
  else
    /* Wait for the last thread. */
    pthread_cond_wait ( &cond, &mutex );

  pthread_mutex_unlock ( &mutex );
}
```

Exercise: barrier object

Implement a `struct barrier_t` that can be used to synchronise the execution of a group of threads.

That is, this data type should enable multiple threads to wait until they all have reached a certain point in the execution of a parallel program.

You must implement the following functions: `barrier_init`, `barrier_destroy`, and `barrier_wait`.

These are described on the next slides.

Initialising a barrier object

```
void barrier_init ( barrier_t* bar, int nt );
```

`bar`	INOUT	A pointer to the barrier object to be initialised.
`nt`	IN	The number of threads that will participate in a barrier operation.

The function `barrier_init` initialises the members of the `struct barrier_t`; more about this later.

It must be called before a barrier operation can be performed.

The parameter `nt` specifies the number of threads that will participate in a barrier operation. This will be the number of threads that should call the function `barrier_wait`.

Destroying a barrier object

```
void barrier_destroy ( barrier_t* bar );
```

`bar`	INOUT	A pointer to the barrier object to be destroyed.

The function `barrier_destroy` destroys a barrier object.

It must be called when a barrier object is no longer needed. Each call to `barrier_init` must be paired with a call to `barrier_destroy`.

Performing a barrier operation

```
void barrier_wait ( barrier_t* bar );
```

`bar`	IN	A pointer to a barrier object.

The function `barrier_wait` blocks all calling threads until the last thread in the group has entered this function.

To be precise, this function blocks until it has been called by the number of threads indicated by the `nt` parameter of the `barrier_init` function.

Hints

The `struct barrier_t` must contain at least a mutex, a condition variable and a counter that keeps track of the number of threads that have called the function `barrier_wait`.

You need more members; that is for you to figure out.

Use *[Threads] Exercise 6 : Barrier* as the starting point of your program.

Test the program by spawning multiple threads and by performing *at least four* barrier operations in succession.

Exercise: reduction operation

Implement a `struct reduction_t` that can be used to perform a global reduction operation – like a global sum – across multiple threads.

That is, this data type should enable multiple threads to provide a local value (of type `double`) and obtain the global reduction of all local values.

You must implement the following functions: `reduction_init`, `reduction_destroy`, `reduction_sum`.

These are described on the next slides.

Initialising a reduction variable

```
void reduction_init ( reduction_t* red, int nt );
```

`red`	INOUT	A pointer to the reduction variable to be initialised.
`nt`	IN	The number of threads that will participate in a reduction operation.

The function `reduction_init` initialises a reduction variable. That is, it initialises the members of the `struct reduction_t`; more about this later.

It must be called before a reduction operation can be performed.

The parameter `nt` specifies the number of threads that will participate in a reduction operation. This will be the number of threads calling the function `reduction_sum`.

Destroying a reduction variable

```
void reduction_destroy ( reduction_t* red );
```

`red`	INOUT	A pointer to the reduction variable to be destroyed.

The function `reduction_destroy` destroys a reduction variable.

It must be called when a reduction variable is no longer needed. Each call to `reduction_init` must be paired with a call to `reduction_destroy`.

Performing a reduction operation

```
double reduction_sum ( reduction_t* red, double val );
```

`red`	IN	A pointer to a reduction variable.
`val`	IN	The local value to be summed.

The function `reduction_sum` performs a global sum across all threads.

The parameter `val` is the local value to be added to the global sum. The return value is the global sum.

This function waits until it has been called by the number of threads indicated by the `nt` parameter of the `reduction_init` function.

Hints

The `struct reduction_t` must contain at least a mutex, a condition variable and a `double` storing the result of the reduction operation (global sum in this case).

You need more members; that is for you to figure out.

Use *[Threads] Exercise 7 : Reduction* as the starting point of your program.

Test the program by spawning multiple threads and by performing *at least four* global reduction operations in succession. Check whether the computed results are correct.

Hints

There is at least a 99% chance you will not get it right at first.

That is normal; the instructors did not get it right too when they first implemented a global reduction operation.

But you might be a genius ... :-)

Exercise: Gram-Schmidt algorithm

Implement another parallel program that executes the Gram-Schmidt algorithm. This time use the POSIX threads library instead of MPI.

The program should make the columns of an $N \times N$ matrix $\boldsymbol{A}$ orthonormal through a series of dot products and vector subtractions.

Use the `struct reduction_t` from the previous exercise to execute the dot products in parallel with multiple concurrent threads.

Exercise: Gram-Schmidt algorithm

The next slide shows the Gram-Schmidt algorithm again in pseudo code. Note that $\boldsymbol{A}_j$ denotes the j-th column of the matrix $\boldsymbol{A}$.

The two slides after that show a serial implementation of the Gram-Schmidt algorithm.

Gram-Schmidt algorithm

do $j = 1, N$

do $k = 1, j-1$

$c_k := \boldsymbol{A}_k^T \cdot \boldsymbol{A}_j$

end do

$\boldsymbol{A}_j := \boldsymbol{A}_j - \sum_{k=1}^{j-1} c_k \cdot \boldsymbol{A}_k$

$\boldsymbol{A}_j := \frac{\boldsymbol{A}_j}{|\boldsymbol{A}_j|}$

end do

Serial Gram-Schmidt implementation (1)

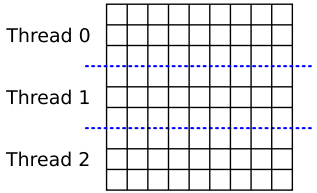
```
double  a[N][N];
double  c[N];
double  t;
int     i, j, k;

for ( j = 0; j < N; j++ )
{
  for ( k = 0; k < j; k++ )
  {
    c[k] = 0.0;

    /* Compute the dot products with the previous columns. */

    for ( i = 0; i < N; i++ )
      c[k] += a[i][k] * a[i][j];
  }

  /* Continued on the next slide ... */
```

Serial Gram-Schmidt implementation (2)

```
  /* ... continued from the previous slide. */

  for ( k = 0; k < j; k++ )
    for ( i = 0; i < N; i++ )
      a[i][j] -= c[k] * a[i][k];

  /* Normalize this column. */

  for ( i = 0, t = 0.0; i < N; i++ )
    t += a[i][j] * a[i][j];

  t = 1.0 / sqrt ( t );

  for ( i = 0; i < N; i++ )
    a[i][j] *= t;
}
```

Parallel Gram-Schmidt implementation

In the parallel implementation of the Gram-Schmidt algorithm, the matrix $\boldsymbol{A}$ is distributed row-wise over the threads:



This means that each threads handles $\frac{N}{p}$ rows, with p the number of threads.

Hints

Use *[Threads] Exercise 8 : Gram-Schmidt Algorithm* as the starting point of your implementation.

Use the functions `matrix_alloc` and `matrix_free` to create and destroy a matrix with arbitrary dimensions.

Define a `struct data_t` that is to be passed to each thread. It should contain at least the matrix size, a pointer to the matrix, a pointer to a shared reduction variable, and the first and last row indices to be processed by a thread.

You must define a `data_t` instance for each thread!

What performance do you get? How could you improve the performance?

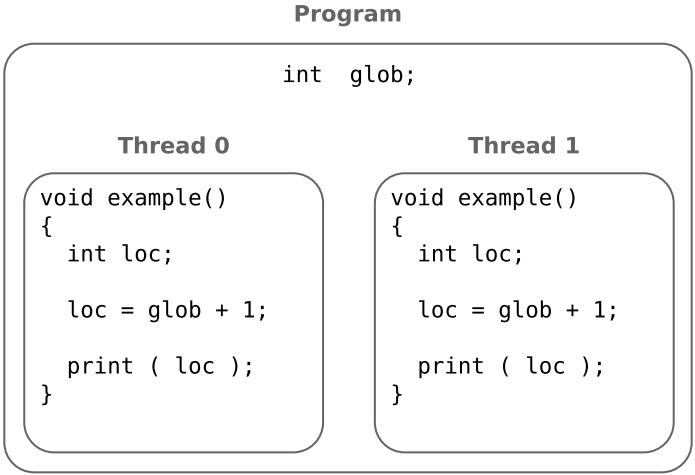
Thread-local storage

Recall that variables, or data in general, can be shared between threads or private to a thread.

Global variables are shared and local variables are private by default.

Each thread has its own copy of a local or private variable; the variable can be accessed without race conditions.

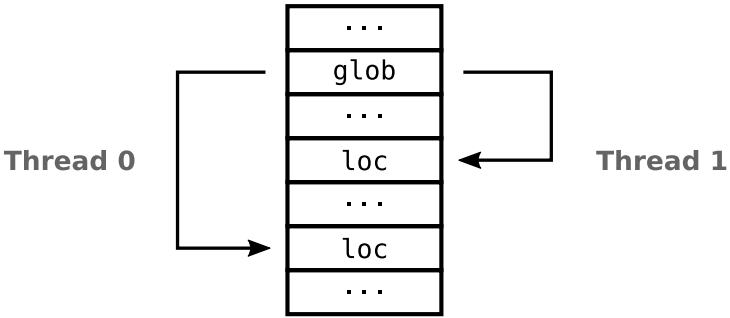
Shared vs local variables



Each thread has its own variable `loc`, but they share the variable `glob`.

Shared vs local variables

This is what happens in memory:



Global private variables

So far, so good.

But how to define a global variable that is private to each thread?

That is, how to define a global variable that has a different value for each thread?

Classical example: `errno`.

The errno variable

The errno variable is a global integer that signals error conditions.

It is typically set by standard library functions that do not return an integer error code.

Many math functions use the errno variable to signal an invalid mathematical operation.

The errno variable

```
#include <math.h>
#include <errno.h>

void example ()
{
  double  x;

  errno = 0;
  x     = sqrt ( -1.0 );

  if ( errno )
  {
    printf ( "Math error!\n" );
  }
}
```

The errno variable

This works fine in a sequential program.

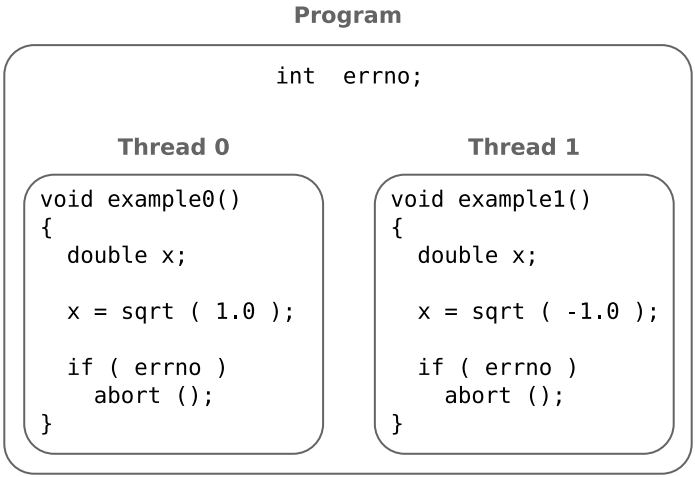
However, what happens of two threads call functions that set the errno variable?

This would result in race conditions if errno would be an ordinary global integer variable.

In particular, one thread might detect an error condition caused by another thread. Or the error condition could be missed altogether.

This is illustrated in the next slide.

Race condition with errno



Thread-local `errno` variables

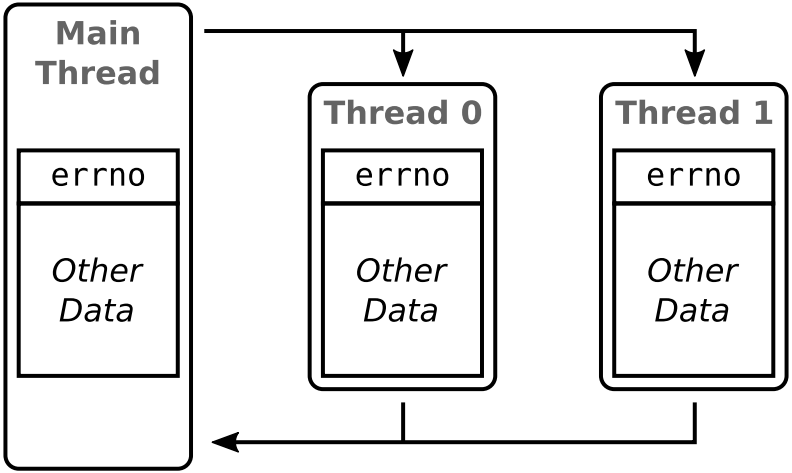
Fortunately, the standard library has foreseen this problem.

The `errno` variable is a global variable but each thread has its *own, private* copy.

The copies are created and destroyed automatically when a thread starts and ends, respectively.

This is achieved by means of *thread-local storage*.

Thread-local `errno` variables



Declaring thread-local variables

The POSIX thread library provides functions for defining and manipulating thread-local variables.

These include: `pthread_key_create`, `pthread_key_delete`, `pthread_setspecific` and `pthread_getspecific`.

While these functions are very flexible, they are a bit cumbersome to use for simple thread-local variables such as `errno`.

Declaring thread-local variables

Fortunately, the C11 and C++11 standards provide a much more simple way to define thread-local variables.

Simply specify `thread_local` before the variable declaration.

This is a macro in C (defined in `<threads.h>`) and a keyword in C++.

This only works for global variables!

Local variables are private by default so it makes no sense to use `thread_local` for local (function) variables.

Thread-local variable example

```
#include <threads.h>

thread_local int  ierror = 0;

void assert_even ( int n )
{
  if ( n % 2 )
  {
    ierror = 1;
  }
}
```

Thread-local variable example [cont]

```
void* thread_func ( void* )
{
  pthread_t  self = pthread_self ();

  assert_even ( (int) self );

  if ( ierror )
  {
    printf ( "Thread ID not even!\n" );
  }

  return NULL;
}
```

Thread-local example explained

The variable `ierror` is a global variable so that it can be accessed from different functions. It also retains its value across function calls.

However, unlike a normal global variable, each thread has its own, private copy.

Different threads can not access each others copy, unless the address of the private copy is passed from one thread to another.

The thread-local copy of the variable `ierror` is destroyed when the current threads exits.

Exercise: thread-safe random number generator

Implement a thread-safe pseudo random number generator using thread-local storage.

Each thread should generate at least ten million random numbers. Run your program with at least four threads.

Verify in a statistical way that all threads generate the same sequence of pseudo random numbers.

Random number generator example

Here is a simple pseudo random number generator:

```
unsigned long  rand_state = 1;

int calc_random ()
{
  rand_state = rand_state * 1103515245 + 12345;

  return ((unsigned) (rand_state / 65536) % 32768);
}
```

Hints

The variable `rand_state` can not be a local variable; its value must be preserved between calls to the function `calc_random`.

Take a small number of samples (ten, say) to verity that all threads have generated the same sequence.

Use *[Threads] Exercise 9 : Thread-safe Random Number Generator* as the starting point of your program.

What happens when you do not use thread-local storage?

Threads in C++

The C++11 standard has added classes to the standard C++ library that make it possible to write portable C++ programs that make use of threads, mutexes and condition variables.

These classes include:

- `std::thread`;
- `std::mutex`;
- `std::condition_variable`;
- and `std::atomic`.

There are more classes and functions, some of them introduced by later C++ standards.

The `std::thread` class

The `std::thread` class represents a thread. It essentially bundles the POSIX thread functions `pthread_create` and `pthread_join`.

When you instantiate a `std::thread` object you can pass the thread start function to the constructor. This will create a child thread of the current thread.

Call the member function `join` to join the child thread.

This is illustrated in the next slide.

Creating a std::thread instance

```
#include <thread>
#include <iostream>

void start_func ()
{
  std::cout << "Hello! I am thread " << std::this_thread::get_id()
            << std::endl;
}

int main ()
{
  std::thread  thread ( start_func );

  std::cout << "Created thread " << thread.get_id() << std::endl;
  thread.join ();
  return 0;
}
```

The thread start function

The thread start function does not need to have a specific signature. You can use any function with an arbitrary parameter list.

You can even use a member function as the thread start function.

Note that the return value from the thread start function will be ignored.

The thread start function should not throw exceptions; this will terminate the program.

Thread start function example

```
struct Task
{
  int  data = 0;
  void run () { data = 1; }
};

void example ()
{
  Task         task;
  std::thread  thread ( &Task::run, &task );

  thread.join ();

  std::cout << "Data = " << task.data << std::endl;
}
```

Joining a thread

The `join` member function must be called before the destructor of a `std::thread` object is invoked. Otherwise your program will be terminated.

```
void example ()
{
  std::thread  thread ( start_func );

  // Oops, thread is destroyed without join().
}
```

The `std::mutex` class

The `std::mutex` class represents a mutex. It essentially bundles the POSIX thread functions `pthread_mutex_init`, `pthread_mutex_destroy`, `pthread_mutex_lock` and `pthread_mutex_unlock`.

A `std::mutex` object can not be copied. It can be embedded in other objects like other classes.

A `std::mutex` may not be locked when its destructor is called.

Example use of the `std::mutex` class

```
#include <mutex>

struct Data
{
  std::mutex  mutex;
  int         count = 0;
};

void incr_count ( Data& data )
{
  data.mutex.lock   ();
  data.count++;
  data.mutex.unlock ();
}
```

Wrong use of the `std::mutex` class

```
void example ()
{
  std::mutex  mutex;

  mutex.lock ();

  throw "Oops, mutex destroyed while locked.";

  mutex.unlock ();
}
```

The `std::unique_lock` class

The `lock` and `unlock` members are usually not called directly to avoid unbalanced lock/unlock operations.

Use the `std::unique_lock` class to lock/unlock mutexes instead. The constructor locks a mutex and the destructor unlocks that mutex.

The `std::unique_lock` class guarantees balanced lock/unlock calls.

Example use of the std::unique_lock class

```
#include <mutex>

struct Data
{
  std::mutex  mutex;
  int         count = 0;
};

void incr_count ( Data& data )
{
  std::unique_lock<std::mutex>  lock ( data.mutex );

  if ( ++data.count > 100000 )
  {
    throw "Overflow"  // Fine, the destructor unlocks the mutex.
  }
}
```

The std::condition_variable class

The std::condition_variable class represents a condition variable. It essentially bundles the POSIX thread functions pthread_cond_init, pthread_cond_destroy, pthread_cond_wait and pthread_cond_signal.

A std::condition_variable must be used together with a mutex or a std::unique_lock to be precise.

Use the member function wait to wait for a condition and use the notify_one function to signal a waiting thread.

This is illustrated in the next slides.

Example use of the std::condition_variable class

```
#include <condition_variable>

struct FIFO
{
  std::mutex               mutex;
  std::condition_variable  cond;
  int*                     message = nullptr;
};

void producer ( FIFO& fifo )
{
  int*  msg = make_message ();
  {
    std::unique_lock<std::mutex>  lock ( fifo.mutex );

    fifo.message = msg;
    fifo.cond.notify_one ();
  }
}
```

Example use of the std::condition_variable class

```
void consumer ( FIFO& fifo )
{
  int*  msg = nullptr;

  {
    std::unique_lock<std::mutex>  lock ( fifo.mutex );

    while ( ! fifo.message ) // Handle spurious wakeups.
    {
      fifo.cond.wait ( lock );
    }

    std::swap ( msg, fifo.message );
  }

  delete [] msg;
}
```

The `std::atomic` class

Use the `std::atomic` class to perform atomic operations on objects of fundamental (integral) type. It is a class template; the template parameter specifies the underlying object type.

Use the member functions `store` and `load` member functions to atomically write and read values.

Overloaded operators can be used to perform the usual numerical operations.

Other member functions can be used to perform atomic swap and atomic compare-and-swap operations. These are useful for implementing *lockless* parallel algorithms.

Example use of the `std::atomic` class

```
#include <atomic>

struct Data
{
  std::atomic<int>  count;
};

void incr_count ( Data& data )
{
  data.count++;  // Atomically increment the counter.

  std::cout << "Count = " << data.count.load() << std::endl;
}
```

Part V

Final Exercise

Exercise: parallel sparse matrix-vector multiplication

This self-study exercise involves writing a program that executes a series of parallel, sparse matrix-vector products.

The program reads a sparse matrix $\boldsymbol{A}$ from a file and stores it in a Compressed Sparse Row (CSR) format.

After that it performs the following series of matrix-vector products:

$$\boldsymbol{x}_i = \boldsymbol{A}\,\boldsymbol{x}_{i-1}$$

The first vector $\boldsymbol{x}_0$ is generated randomly.

Exercise: parallel sparse matrix-vector multiplication

Use *[Project] Parallel sparse matrix-vector multiplication* as the basis of your parallel program.

This source files implements the sequential version of the program. It defines a `struct sparse_matrix_t` that represents a sparse matrix.

The program reads a sparse matrix from a file named `matrix.dat`.

You should (try to) implement three parallel versions of the program: one with MPI; one with OpenMP and one with threads.

The CSR format

The CSR format stores the non-zero values in a sparse matrix row wise in three arrays: a `value` array; an `index` array; and an `offset` array.

The `value` array stores all non-zero values, one row after another.

That is, the first entries store the non-zero values on the first row. The entries after that store the non-zero values on the second row. And so forth.

The length of the `value` array is equal to the number of non-zero value in the matrix.

The CSR format

The `index` array stores the column index of each non-zero value.

That is, the i-th entry in the `index` array stores the column index corresponding with the i-th entry in the `value` array.

The length of the `index` array is equal to the length of the `value` array.

The `offset` array stores the positions in the `value` and `index` array where each row starts.

That is, the i-th entry in the `offset` array is the position of the first non-zero value on the i-th row.

The CSR format

The length of the `offset` array is equal to the number of rows plus one. The last entry in the `offset` array equals the total number of non-zero values.

The number of non-zero values on the *i*-th row is equal to:

```
k = offset[i + 1] - offset[i];
```

Note that the first entry in the `offset` array is zero by definition.

The CSR format

Here is an example of a sparse matrix that is stored in the CSR format:

$$\boldsymbol{A} = \begin{bmatrix} 1 & 0 & 0 & 0 & 4 \\ 0 & 2 & 1 & 0 & 0 \\ 1 & 0 & 3 & 1 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 2 & 0 & 1 \end{bmatrix}$$

$$\texttt{value} = \begin{bmatrix} 1 & 4 & 2 & 1 & 1 & 3 & 1 & 2 & 2 & 1 \end{bmatrix}$$
$$\texttt{index} = \begin{bmatrix} 0 & 4 & 1 & 2 & 0 & 2 & 3 & 3 & 2 & 4 \end{bmatrix}$$
$$\texttt{offset} = \begin{bmatrix} 0 & 2 & 4 & 7 & 8 & 10 \end{bmatrix}$$

Sparse matrix-vector product

The following code fragment computes the product of a sparse matrix with an array named `rhs`. The result is stored in an array named `lhs`.

```
for ( i = 0; i < nrows; i++ )
{
  t = 0.0;

  for ( j = offset[i]; j < offset[i + 1]; j++ )
  {
    t += value[j] * rhs[index[j]];
  }

  lhs[i] = t;
}
```

Hints

Partition the matrix row wise and assign each batch of rows to a different thread or process.

Read the matrix on the master thread/process.

Start with the parallel implementation based on OpenMP as this is the most straightforward.

Compare the parallel performance of each implementation. Is there a significant difference?