Parallel computations[1]

• *Parallel computation = use more than one processor to run one program.*

• *Goal is to **speed up the calculations**; i.e. to minimize the wall clock time used by the program.*
- There is some overhead caused by communication and synchronization of processors.
- Note that it is now the wall clock time we are measuring because some processors may be forced to be idle due to communication and synchronization.
- This idling is inherent to the computational task and not forced by (for example) the operation system scheduler.

• *By dividing the problem into smaller parts **bigger problems may be solved**.*

• *Desired attributes of a parallel algorithms and software*

- **Concurrency**: Ability to perform many actions simultaneously.

- **Scalability**: Performance as a function of the number of processors.

- **Locality**: Parallel computer: accessing local memory faster than remote memory.

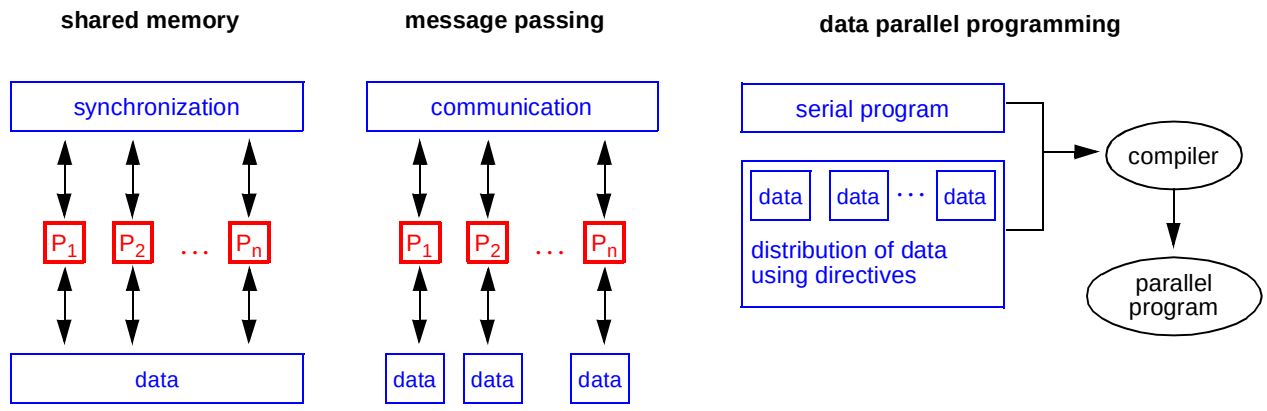
- **Modularity**: Decomposition of complex entities into simpler components.

• *Note that parallelization of a code is possible only if the algorithm is parallelizable.*
- Many efficient sequential algorithms are poorly parallizable.
- The programmer may have to change the algorithm.
- This may be a considerable programming effort.

1. Partly adapted from Ian Foster, Designing and Building Parallel Programs, Addison-Wesley, 1995

Parallel computations

• *There are many ways of classifying **parallel programming models**. One possibility is the following. P_i denotes a processing element (CPU).*



- Although there is close correspondence between the programming and machine models, the type of the machine need not be the same as the programming model.
  - For example: in an shared memory machine message passing programming can be used.

Parallel computations

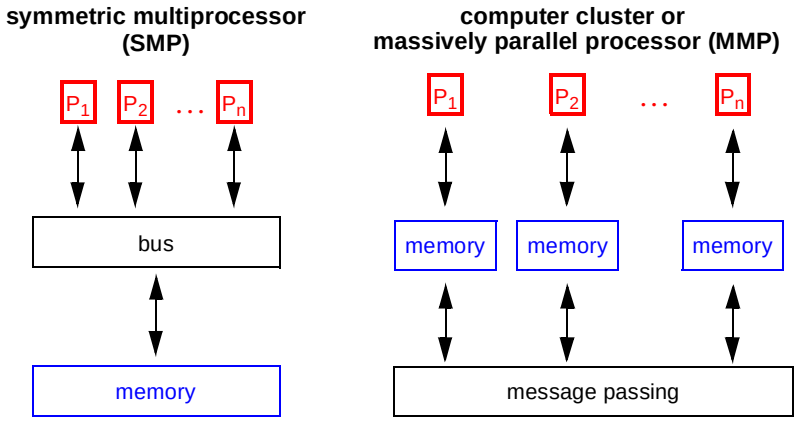
- Examples how programs using message passing and data parallel programming look like:

```
program hello
  integer :: nthreads,tid,omp_get_num_threads,omp_get_thread_num
  !$omp parallel private(tid)
  ! obtain and print thread id
  tid = omp_get_thread_num()
  print *, 'hello world from thread = ', tid
  if (tid==0) then
     nthreads = omp_get_num_threads()
     print *,'number of threads = ', nthreads
  end if
  !$omp end parallel
end program hello
```

```
program helloworld
   use mpi
   implicit none
   integer :: ierr,procnum,numprocs

   call mpi_init(ierr)
   call mpi_comm_rank(mpi_comm_world,procnum,ierr)
   call mpi_comm_size(mpi_comm_world,numprocs,ierr)
   print *,'hello world! from processor ',procnum,' out of ',numprocs
   call mpi_finalize(ierr)

   stop
end program helloworld


 mpi> ~/mpich_ifort/bin/mpif90 mpi_helloworld.f90
 mpi> ~/mpich_ifort/bin/mpirun  -machinefile machine.dat -np 3 a.out
  Hello world! From processor           0  out of           3
  Hello world! From processor           2  out of           3
  Hello world! From processor           1  out of           3
```


Parallel computations

• *On the other hand parallel computers can be classified in various ways. One possibility is given on the right*



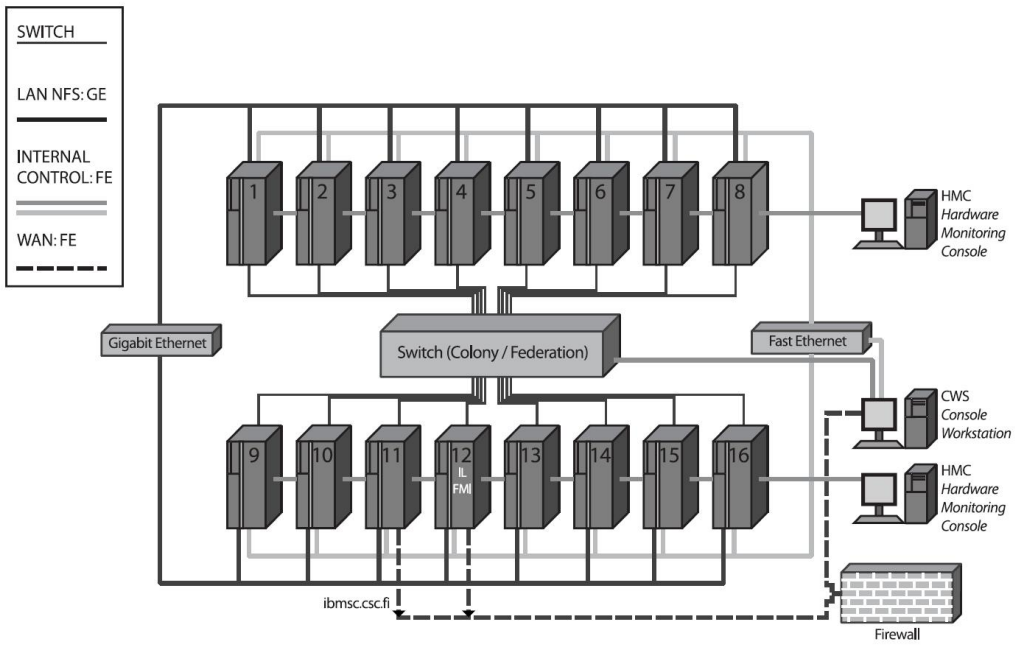
• ***Shared memory***
  1) *doesn't need message passing*
  2) *is usually the fastest*

• ***Message passing***
  1) *Processors only see their own memory.*
  2) *Data from the memory of another processor must be explicitly requested.*
  3) *Distribution of data and work load is the responsibility of the programmer.*
  4) *Today the message passing library is **MPI** (Message Passing Interface)*

• ***Data parallel programming***
  1) *Processors perform identical operations to differents elements of a data structure.*
  2) *Fine-grained parallelization.*
  3) *Programming tools: **OpenMP***
     - *Sequential Fortran with compiler directives*
     - *Directives tell the compiler how data is distributed between processors.*
     - *Programming easy.*
     - *Only for problems with regular data structures.*

Parallel computations

• *Often a modern parallel computer is a combination of these. For example the IBM SC computer of CSC[1]:*



Each node (1-16) contains 32 POWER4 CPUs and 32 or 63 GB of memory.

The latency of the switch is 20 microseconds and performance 2x500 MB/s/port.

1. IBMSC User's Guide, Raimo Uusvuori and Tiina Kupila-Rantala (Eds.), CSC Scientific Computing Ltd.

Parallel computations

• *When parallelizing a computational task one has to make many choices:*

- Coarse or fine grained parallelization
  How small or large parts are used to divide data and tasks?

- How is data distributed between processors?

- How is computation divided between processors?

- Synchronization
  Do the processors need to be synchronized?
  If they need how is it done in the most efficient way so that idling is minimized?

- Load balancing
  How is the computational burden divided between processors so that they all get the same amount of computations?

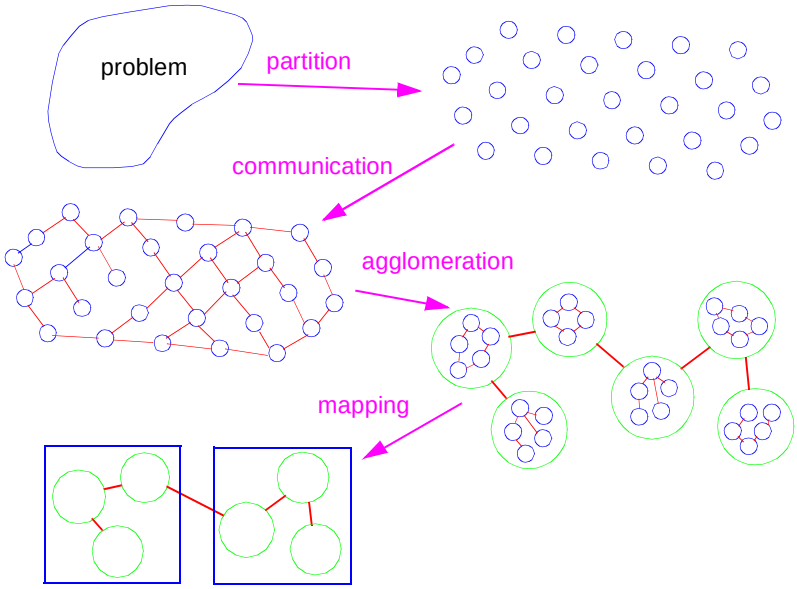
Parallel computations

• *Steps in building a parallel program*

1) **Partition** the problem into smaller parts (modularity).

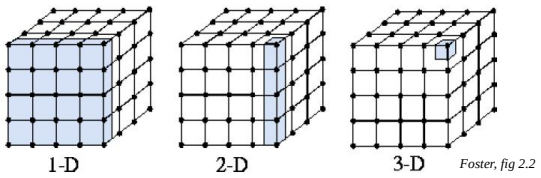
- Divide data or tasks.
- Results is a fine-grained decomposition.
- Checklist:
  1) Does the partition result in oders of magnitude more parts than the computer has processors? If not, then there is only little flexibility in the next stages.
  2) Can we avoid unnecessary computations and memory usage by the partitioning?
  3) Are the task of the same size?
  4) Can the number of tasks be increased when the size of the problem grows?

- In many parallel problems the data is first partitioned. This is called **domain decomposition**.



Parallel computations

- A simple example of this is parallel atomistic simulation:
  Each processor handles atoms that have coordinates in predetermined limits.

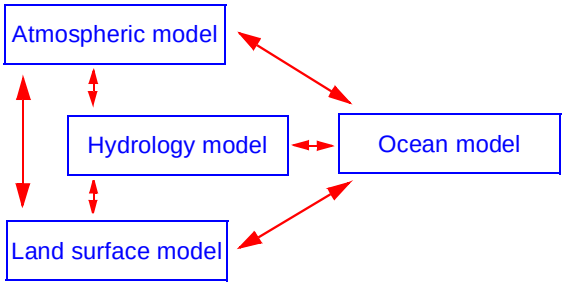


Foster, fig 2.2

- In **functional decomposition** the computations are partitioned.

  - After this the data requirements are determined.

  Example: climate model



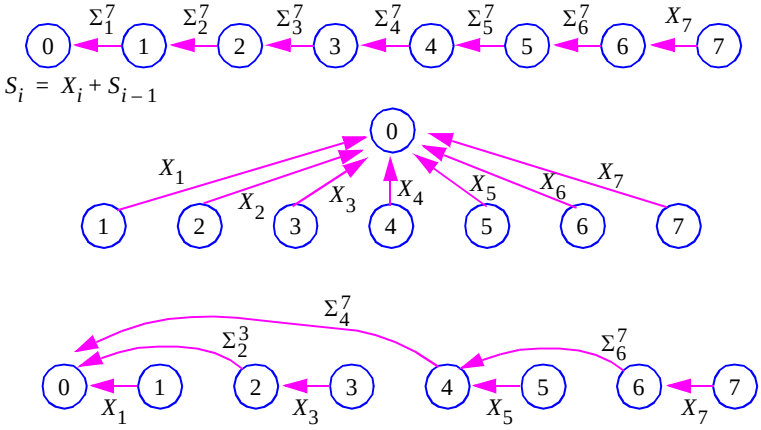
Parallel computations

2) The **communication** needed between the partial tasks is determined and appropriate algorithms are defined.

- Local or global?
  - Tasks communicate only between their neighbors.
  - Task communicate with many other tasks (reduction operations).

Summation of numbers:

$$S = \sum_{i=0}^{7} X_i$$

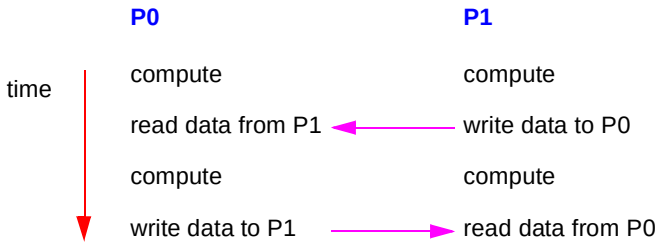


- Structured or unstructured?
  - Communication partners in a regular structure: a grid or a tree.
  - Irregular: for example FEM calculations of an irregular object.

Parallel computations

- Static or dynamic?
  - The same communication partners during the whole run.

- Synchronous or asynchronous?
  - Synchronous: consumers and producers cooperate in data transfer.



  - Asynchronous: data producers not able to determine whether consumers require more data.

- Checklist:
  1) Does each task perform the same amount of communication?
  2) Does each task communicate with only a small number of its neighbors?
  3) Can the communication proceed concurrently?
  4) Can the computation proceed concurrently?

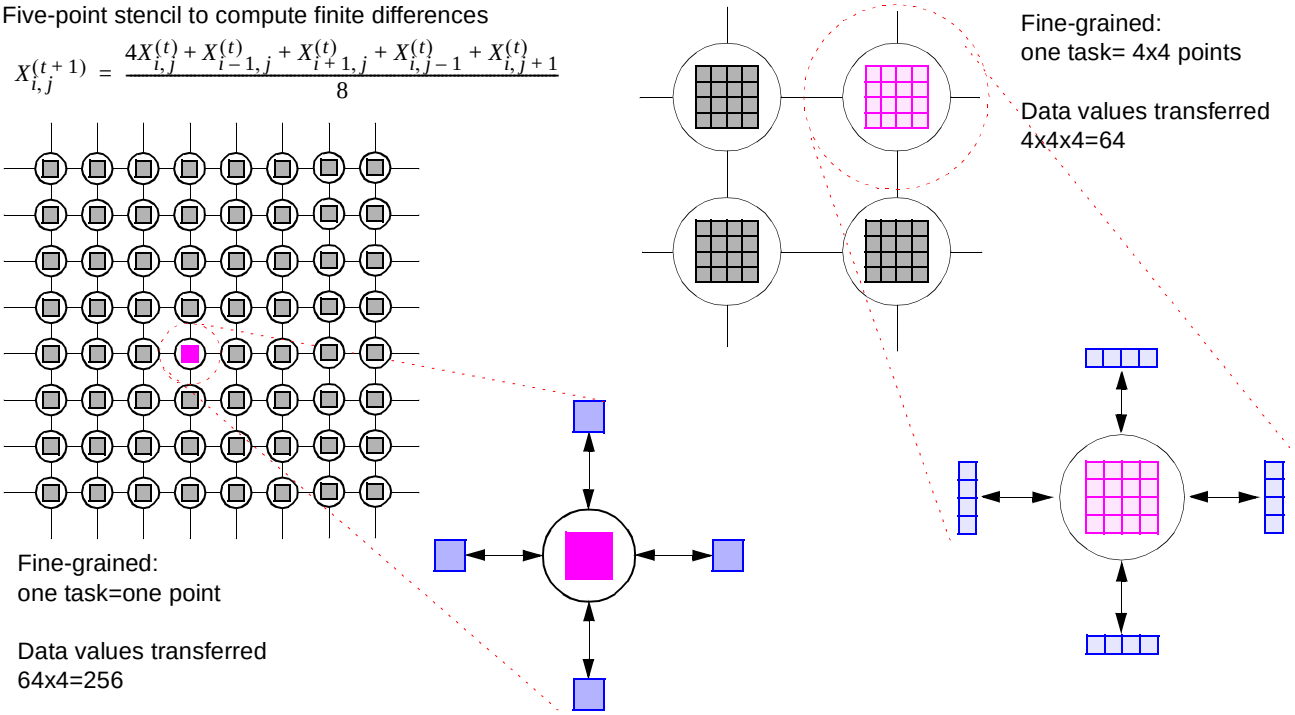
Parallel computations

3) The partition and communication defined in steps 1 and 2 are evaluated with respect to performance requirements and communication costs. If necessary, many partial task are **agglomerated** to form larger ones.

- Increasing granularity: reduced communication (surface-to-volume ratio)

Five-point stencil to compute finite differences

$$X_{i,j}^{(t+1)} = \frac{4X_{i,j}^{(t)} + X_{i-1,j}^{(t)} + X_{i+1,j}^{(t)} + X_{i,j-1}^{(t)} + X_{i,j+1}^{(t)}}{8}$$



Fine-grained:
one task=one point

Data values transferred
64x4=256

Fine-grained:
one task= 4x4 points

Data values transferred
4x4x4=64

Parallel computations

- Replicate data: save in computing and communication.
- Replicate computation: save in commmunication.

- Checklist:
  1) Has agglomeration decreased the amount of communication by increasing locality?
  2) If computations have been replicated do the benefits outweight the cost?
  3) If data has been replicated ensure that it does not prevent the scalability.
  4) Do the task have equal amounts of computation and communication?
  5) Ensure that the agglomeration does not affect scalability and concurrency.

Parallel computations

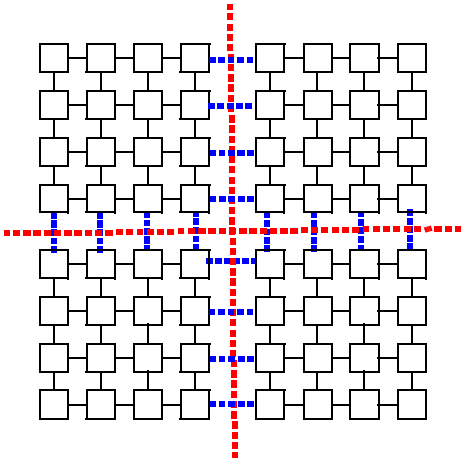
4) Partial tasks are **mapped** to processors while maximizing processor utilization and minimizing communication costs.

- Minimize total executing time:
  - Place tasks that are able to execute concurrently on different processors.
  - Place tasks that communicate frequently on the same processor.

  - Straightforward in many algorithms developed using domain decomposition.
    - Fixed number of equal-sized tasks and structured local and global communication.

  - In more complex algorithms load balancing may be needed.
  - Number of tasks or amount of computation and communication per task changes during execution: dynamic load balancing.

  - Functional decomposition: task scheduling
    - Manager/worker approach: each worker repeatedly requests and executes a problem from the manager.



Parallel computations

• *Performance of a parallel program can be measured in various ways.*
  - Execution time
  - Scalability when the number of processors is increased
  - Scalability when the problem size grows

  - **Amdahl's law** tells how the execution time is scaled when the number of processors is increased (problem size stays constant):

$$S_p = \frac{W_p + W_r}{W_p + W_r / p},$$

where S_p is the ratio of the execution times with one and p processors

W_p is the execution time of the sequential part of the program

W_r is the execution time of the parallel part when using one processor.

  - By normalizing the one-processor execution time to one ($W_p + W_r = 1$) we get

$$S_p = \frac{1}{\alpha + (1-\alpha)/p},$$

where $W_p = \alpha$ and $W_r = 1 - \alpha$.

Parallel computations

- Performance can also be measured by efficiency e which is defined as

$$e = \frac{S_p}{p}.$$

- In the ideal case efficiency is $e = 1$.

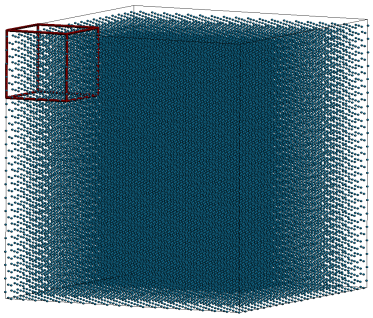
- In **Gustafson's law** we keep the execution time constant and increse the problem size:

$$S'_p = \frac{W_p + pW_r}{W_p + W_r}$$

- By normalizing $W_p + W_r = 1$, $\alpha' = W_p$, we get the law in form

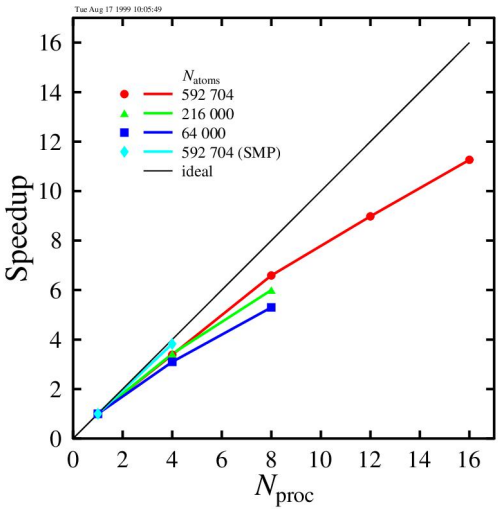
$$S'_p = p - \alpha'(p-1)$$

Parallel computations



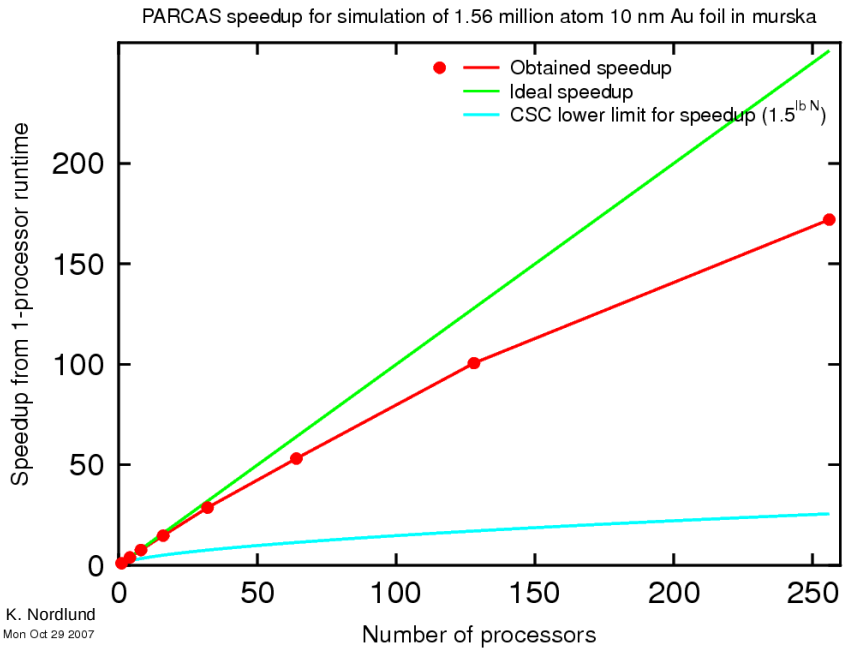
• *Example: parallel molecular dynamics (MD) simulations*
- Newton's equations of motion of atoms integrated numerically.
- Interaction via classical forces (may be many-body forces).
- Practical simulations: $N_{\text{atoms}} = 10^2 \ldots 10^8$
- Parallellization by domain decomposition: each processor takes care of atoms located in a certain part of the system.
- Communication: Each processor has to send receive positions of those atoms in its neighboring processors with which its atoms interact. I.e. thin 'skins' of each processor atom data are exchanged between neighbor processors.
- Atoms may move from one processor to another: at every time step atoms must be reorganized.
- Parallellization by message passing.

- On the right: speedup of MD code PARCAS in a workstation cluster and a SMP machine.
  - Technical details:
    - Classical MD simulation with nearest neighbor potential (Tersoff for Si).
    - Cluster: Linux Alpha processors connected with 100 MB/s Ethernet.
    - SMP: Sun 4-processor machine.



Parallel computations

• *A newer example:* MD code PARCAS on **murska.csc.fi**:



Murska is an HP CP4000 BL ProLiant supercluster. Murska has 2048 compute cores, or 1024 dual-core 2.6 GHz AMD Opteron 64-bit processors. A compute node consists of two processors, 16 nodes make a blade enclosure, and there are 32 blade enclosures in 8 compute cabinets. In addition there are three login nodes and one control node for administrative purposes.

The compute nodes are equipped with three different memory configurations: 4 GB per core (512 cores), 2 GB per core (512 cores) and 1 GB per core (1024 cores).

Processors are connected via InfiniBand (4x DDR) network. The theoretical peak performance of Murska is 10.6 Tflops. Murska has 98 TB of local disk space using the fast SFS/Lustre file system.

The cluster has RHEL 4 Linux based HP XC cluster software.

Source: **http://www.csc.fi/english/pages/murska_guide/introduction/overview**

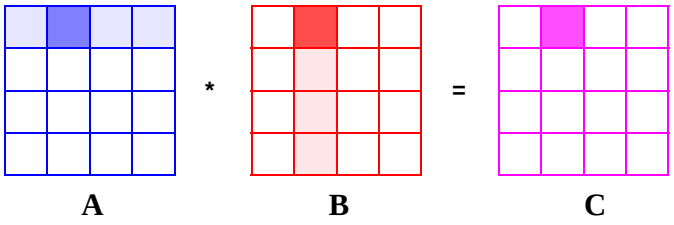
Parallel computations

• *Example: parallel matrix computations*

- Matrix multiplication of two $N \times N$ matrices

$$\mathbf{C} = \mathbf{AB} \quad \text{or} \quad C_{ij} = \sum_{k=1}^{N} A_{ik} B_{kj}$$

- P tasks (processors)
- 2D decomposition: each task takes care of a $N/P \times N/P$ block.

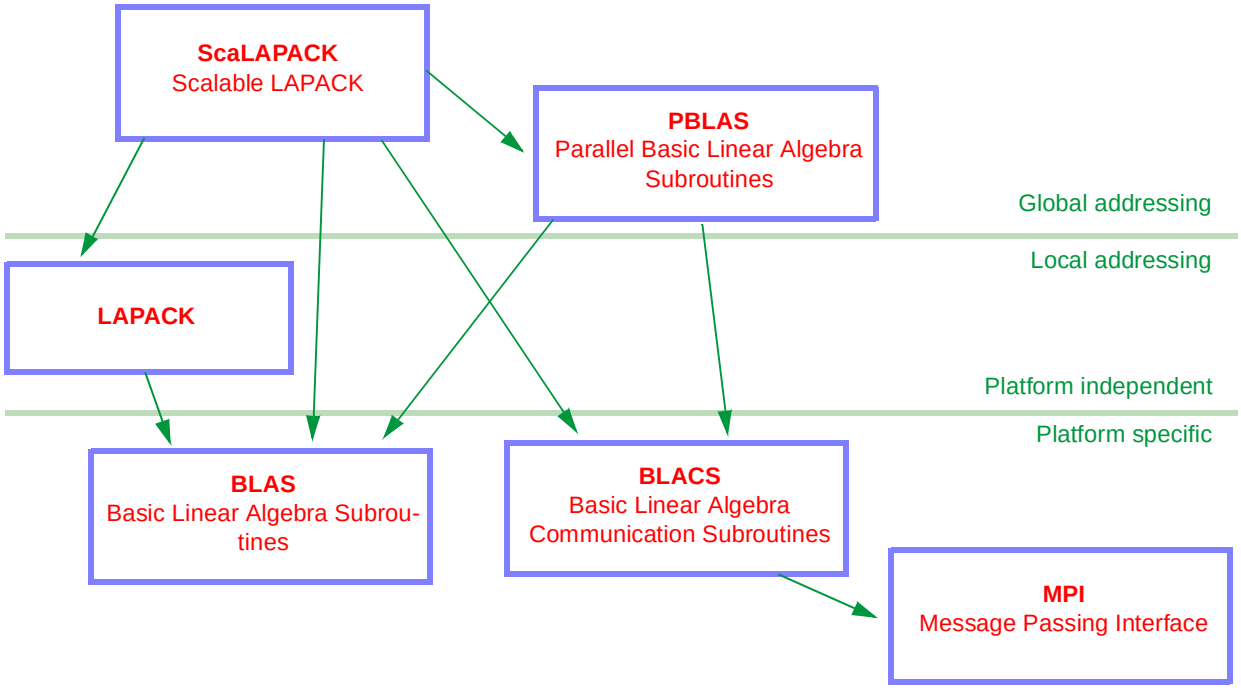


- 16 processors
- each multiplies $N/4 \times N/4$ submatrices

- Note that computation of one block in matrix $\mathbf{C}$ requires the data from all block matrices in the same row in $\mathbf{A}$ and in the same column in $\mathbf{B}$.

Parallel computations

- Parallel version of **LAPACK** package: **ScaLAPACK** (**http://www.netlib.org/scalapack/**)
  - Solves systems of linear equations, least squares and eigenvalue problems



Parallel computations

- Compromise between communication overhead, loadbalancing and ability to use level 3 BLAS routines within a single processor: **two-dimensional block cyclic distribution** of matrix

(a)

	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	1	1	2	2	0	0	1	1	2	2
1	0	0	1	1	2	2	0	0	1	1	2	2
2	3	3	4	4	5	5	3	3	4	4	5	5
3	3	3	4	4	5	5	3	3	4	4	5	5
4	0	0	1	1	2	2	0	0	1	1	2	2
5	0	0	1	1	2	2	0	0	1	1	2	2
6	3	3	4	4	5	5	3	3	4	4	5	5
7	3	3	4	4	5	5	3	3	4	4	5	5
8	0	0	1	1	2	2	0	0	1	1	2	2
9	0	0	1	1	2	2	0	0	1	1	2	2
10	3	3	4	4	5	5	3	3	4	4	5	5
11	3	3	4	4	5	5	3	3	4	4	5	5

(b)

	0	1	6	7	2	3	8	9	4	5	10	11
0	0	0	0	0	1	1	1	1	2	2	2	2
1	0	0	0	0	1	1	1	1	2	2	2	2
4	0	0	0	0	1	1	1	1	2	2	2	2
5	0	0	0	0	1	1	1	1	2	2	2	2
8	0	0	0	0	1	1	1	1	2	2	2	2
9	0	0	0	0	1	1	1	1	2	2	2	2
2	3	3	3	3	4	4	4	4	5	5	5	5
3	3	3	3	3	4	4	4	4	5	5	5	5
6	3	3	3	3	4	4	4	4	5	5	5	5
7	3	3	3	3	4	4	4	4	5	5	5	5
10	3	3	3	3	4	4	4	4	5	5	5	5
11	3	3	3	3	4	4	4	4	5	5	5	5

FIG. 2. Example of a homogeneous block cyclic distribution of a 12×12 matrix over 2×3 process grid with the block size 2×2—(a) matrix distribution over grid and (b) distribution from processor point of view.

A. Kalinov, A. Lastovetsky, *Journal of Parallel and Distributed Computing* **61**, 520-535 (2001).

Parallel computations: MPI

• ***MPI*** *(Message Passing Interface) is a library specification for message-passing, proposed as a standard by a broadly based committee of vendors, implementors, and users.*

- Message passing is a 'low level' parallelization method: programmer has to do everything herself
- On the other hand (at least currently) the only way to write efficient and scalable parallel code for more than ten processors.
- Other message passing tools is PVM (Parallel Virtual Machine). However, its use is not recommended.

- Home page: http://www-unix.mcs.anl.gov/mpi/
- MPI is
  a standard.
  availabe as open source and vendor supplied implementations.
  the most common parallel programming tool in scientific computing.

- Because MPI is a standard a program written with it is easily ported to another environment.

- MPI is intended to be used with Fortran 77 and ANSI C.
  - No standardized support for advanced Fortran features.

- Strong points of MPI are
  1) versatile point-to-point communication
  2) a large number of collective communication routines
  3) group communication is flexible and easy to program
  4) you can define your own data types to be used in communication

Parallel computations: MPI

• *Now back to learning MPI itself.*

- The programming model of MPI is SPMD (Single Program Multiple Data).
- Every task is a separate process. They can execute on the same or different processors.
  - All variables are local.
  - Communication by MPI subroutine calls.
  - Programmer responsible for task synchronization.
- Every task has a unique id number which can be used to control the execution.
  - E.g. process 0 may the master taking care of all data collection and IO.

- MPI has more than 120 routines.
- However, with only 6 routines you can do a lot:

```
mpi_init
mpi_comm_size
mpi_comm_rank
mpi_send
mpi_recv
mpi_finalize
```

- A new version, MPI-2, is already standardized.
  - New features: dynamic process management, one-sided operations, parallel I/O
    - We probably take a look at these new features later.

- We use the **Open MPI** implementation (**http://www.open-mpi.org/**)
  - It is open source: you can compile it from sources or install prebuild binaries for various Linux distributions.
  - Newest versions have support for various batch job systems (including SGE that we will be using)

Parallel computations: MPI

- A simple example[1] (non-trivial; we already saw the 'Hello world' program):

```
program mpiexample1
  use mpi
  implicit none
  integer,parameter :: tag = 50
  integer :: id,ntasks,source_id,dest_id,rc,i
  integer,dimension(mpi_status_size) :: status
  integer,dimension(2) :: msg

  call mpi_init(rc)
  if (rc/=mpi_success) then
     print *,'MPI initialization failed.'
     stop
  end if
  call mpi_comm_size(mpi_comm_world,ntasks,rc)
  call mpi_comm_rank(mpi_comm_world,id,rc)

  if (id/=0) then
     msg(1)=id
     msg(2)=ntasks
     dest_id=0
     call mpi_send(msg,2,mpi_integer,dest_id,tag,mpi_comm_world,rc)
  else
     do i=1,ntasks-1
        call mpi_recv(msg,2,mpi_integer,mpi_any_source,tag,&
             &mpi_comm_world,status,rc)
        source_id=status(mpi_source)
        print *,'message:',msg,'sender:',source_id
     end do
  end if

  call mpi_finalize(rc)
end program mpiexample1
```

```
example> mpif90 mpiexample1.f90
example> mpirun -hostfile host.file -np 4 a.out
 message:           3           4 sender:           3
 message:           1           4 sender:           1
 message:           2           4 sender:           2
```

File `host.file`:

```
mill.physics.helsinki.fi
compute-1-11.local
compute-1-12.local
compute-1-14.local
compute-1-15.local
compute-1-16.local
```

Instead of the use statement, you can use the include statement:

```
program mpiexample1
  implicit none
  include 'mpif.h'
```

1. From J. Haataja, K. Mustikkamäki: Rinnakkaisohjelmointi MPI:llä, CSC, 1997. http://www.csc.fi/oppaat/mpi/

Parallel computations: MPI

- All definitions are included from module **`mpi`** file **`mpif.h`** .

- The routines called here are the following

Initialization: **`mpi_init(rc)`**
Initialization. Returns **`mpi_success`** in **`rc`** if all went ok.
`mpi_comm_size(mpi_comm_world,ntasks,rc)`
Return the number of processors of the communicator (group) **`mpi_comm_world`** in variable **`ntasks`**.
`mpi_comm_rank(mpi_comm_world,id,rc)`
Return my id number in the communicator **`mpi_comm_world`**. in the variable **`id`**.

Send data: **`mpi_send(msg,2,mpi_integer,dest_id,tag,mpi_comm_world,rc)`**
Send two integers in array msg to destination processor dest_id

Receive data: **`mpi_recv(msg,2,mpi_integer,mpi_any_source,tag,mpi_comm_world,status,rc)`**
Receive two integers in arrays **`msg`** from any processor.

Clean up: **`mpi_finalize(rc)`**

Parallel computations: MPI

- The same program in C[1]

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

main(int argc,char *argv[])
{
  const int tag=50;
  int id,ntasks,source_id,dest_id,rc,i;
  MPI_Status status;
  int msg[2];

  rc=MPI_Init(&argc,&argv);
  if (rc != MPI_SUCCESS) {
    printf("MPI initialization failed\n");
    exit(1);
  }
  rc=MPI_Comm_size(MPI_COMM_WORLD,&ntasks);
  rc=MPI_Comm_rank(MPI_COMM_WORLD,&id);

  if (id != 0) {
    msg[0]=id;
    msg[1]=ntasks;
    dest_id=0;
    rc=MPI_Send(msg,2,MPI_INT,dest_id,tag,MPI_COMM_WORLD);
  } else {
    for (i=1; i < ntasks; i++) {
      rc=MPI_Recv(msg,2,MPI_INT,MPI_ANY_SOURCE,tag,MPI_COMM_WORLD,&status);
      source_id=status.MPI_SOURCE;
      printf("message: %d %d sender: %d\n",msg[0],msg[1],source_id);
    }
  }
  rc=MPI_Finalize();
  exit(0);
}
```

```
example> mpicc mpiexample1.c
example> mpirun -hostfile host.file -np 4 a.out
message: 3 4 sender: 3
message: 1 4 sender: 1
message: 2 4 sender: 2
```

1. From J. Haataja, K. Mustikkamäki: Rinnakkaisohjelmointi MPI:llä, CSC, 1997. http://www.csc.fi/oppaat/mpi/

Parallel computations: MPI

- Subroutines passing messages (**mpi_send** and **mpi_recv** in these examples) have as their parameters (in addition to the data itself) the
  1) MPI data type
  2) number of elements
  3) senders id
  4) message tag (can be chosen by the programmer); must be >0
  5) receivers id(s).

- Point-to-point messages sent by the two processors to each other preserve their ordering in time.

- **mpi_send** and **mpi_recv** are *blocking*: program calling **mpi_send** returns from the routine only after the send buffer can be used again. Correspondingly **mpi_recv** returns when the incoming data is copied to the input buffer.

- Sending routine call in detail:

```
integer :: count,datatype,dest,tag,comm,rc
whatever :: buf(*)

call mpi_send(buf,count,datatype,dest,tag,comm,rc)
```

buf	data to be sent
count	number of elements in buf
datatype	type of the elements
dest	id of the receiver
tag	id number of the message
comm	communication group

Parallel computations: MPI

- Receiving routine call in detail:

```
integer :: count,datatype,src,tag,comm,rc
integer :: status(mpi_status_size)
whatever :: buf(*)

call mpi_recv(buf,count,datatype,src,tag,comm,status,rc)
```

`src`	id of the sender
`status`	information related to message passing

- The receiving `tag` must be the same as the sending one, unless `mpi_any_tag` is used.

- In Fortran, `status` is an array of integers of size `mpi_status_size`. The constants `mpi_source`, `mpi_tag` and `mpi_error` are the indices of the entries that store the source, tag and error fields.

- In C, `status` is a structure that contains three fields named `MPI_SOURCE`, `MPI_TAG`, and `MPI_ERROR`; the structure may contain additional fields.

Parallel computations: MPI at mill

• *Now a few words about using MPI in practice.*

• *MPI is installed at mill.physics.helsinki.fi.*
• *Below you can find some instructions in using it (and using MPI in general).*

- Compilation is done by commands
  `mpif90`
  `mpicc`
  - These commands are wrappers that pass their options to the underlying F90 or C compiler (in our case `gfortran` or `gcc`)
  - They also add proper options so that the MPI libraries are linked.
  - Help: **`mpif90 --help`**

- All MPI commands and routines have man pages; e.g. **`man MPI_Send`**

- Running an MPI program is accomplished by the command mpirun:
  `mpirun -hostfile host.file -np 4 a.out`

  - With option **`-np`** you give the number of processors..
  - Note that option **`-hostfile`** is not needed when running MPI programs under batch job system (SGE).

Parallel computations: MPI at mill

- Programs on **mill** are run under a batch job system called SGE (Sun Grid Engine)
- A package containing instructions to run MPI programs under SGE can be found at
  `http://www.physics.helsinki.fi/courses/s/stltk/progs/mpi/openmpi_starter_kit.tgz`
- It is SGE that takes care of finding free processors for your job to run on; no need to specify hostfile in **`mpirun`**.
- Below is the example batch job submit script (file `submit` in the package mentioned above):

```
#!/bin/sh
#--------------------------------------
# Options for SGE
#$ -o mpiexample_$JOB_ID.log
#$ -N mpiexample
#$ -S /bin/bash
#$ -V
#$ -cwd
#$ -j y
#$ -l h_cpu=0:10:00          <- maximum CPU time (10 min)
#$ -notify
#$ -pe openmpi 4             <- parallel environment
#$ -q student.q              <- batch queue
#$ -R y
#--------------------------------------

# Establish the submission work directory.
sdir=`pwd`

# Create the computation work directory.
prefix=/work/$USER
if [ ! -d $prefix ]; then
   mkdir $prefix
fi

dir=${sdir/$HOME/}
dir=${dir//\//__}
dir=$prefix/${dir}_$$
mkdir $dir
cd $dir/
```

Lines beginning with **`#$`** are interpreted by SGE.

Batch jobs use local disks of nodes mounted on **`/work`** .

```
echo "Date:                " `date`
echo "Submission directory: " $sdir
echo "Running directory:    " `pwd`
echo "Host:                 " `hostname`

# Run your executable
mpirun -np 4 $sdir/mpiexample &> example.out

# Copy results back to submission work directory.
cp example.out $sdir/
```


Parallel computations: MPI

• *And now back to learning MPI itself.*

• *In order to communication succeed one has to ensure that*

1) Sender must give a reasonable receiver id.
2) Receiver must give a reasonable sender id or `MPI_ANY_SOURCE`.
3) Sender and receiver communication groups must be the same.
4) Message tags must be the same or receiver must use `MPI_ANY_TAG`.
5) Possible buffers must be large enough.
6) Sender and receiver data types must be the same.

C	Fortran
MPI_CHAR	MPI_CHARACTER
MPI_SHORT	MPI_INTEGER
MPI_INT	MPI_REAL
MPI_LONG	MPI_COMPLEX
MPI_UNSIGNED_CHAR	MPI_DOUBLE_PRECISION
MPI_UNSIGNED_SHORT	MPI_DOUBLE_COMPLEX
MPI_UNSIGNED	MPI_LOGICAL
MPI_UNSIGNED_LONG	MPI_BYTE
MPI_FLOAT	MPI_PACKED
MPI_DOUBLE	
MPI_LONG_DOUBLE	
MPI_BYTE	
MPI_PACKED	

• *MPI uses its own 'data types' in communication.*

- Table on the right lists the MPI data types in C and Fortran.
- MPI quarantees successful communication in a heterogeneous environment.
- Data types are determined in send and receive routines.
- The real data types of the elements must agree with the MPI datatypes.
- MPI data types can only be used in MPI routines, not to define variables in F90 or C.
- However, user can define his own data types. For example an integer vector:

```
integer :: count,newtype,ierr
...
call mpi_type_contiguous(count,MPI_INTEGER,newtype,ierr)
call mpi_type_commit(newtype,ierr)
```

Parallel computations: MPI

- Note that we could always use MPI_BYTE data type. But then you should know the lengths of the types in bytes.
- This would produce code that is tedious to port.
- All versions below work, but only the first one should be used.

```
integer,dimension(2) :: msg
...
call mpi_send(msg,2,mpi_integer,dest_id,tag,mpi_comm_world,rc)

call mpi_recv(msg,2,mpi_integer,mpi_any_source,tag,mpi_comm_world,status,rc)
```

```
integer,dimension(2) :: msg
...
call mpi_send(msg,8,mpi_byte,dest_id,tag,mpi_comm_world,rc)

call mpi_recv(msg,2,mpi_integer,mpi_any_source,tag,mpi_comm_world,status,rc)
```

```
integer,dimension(2) :: msg
...
call mpi_send(msg,8,mpi_byte,dest_id,tag,mpi_comm_world,rc)

call mpi_recv(msg,8,mpi_byte,mpi_any_source,tag,mpi_comm_world,status,rc)
```


Parallel computations: MPI

• *MPI has also a large number of **collective** routines.*
- All the processors (in the communication group) participate in the operation.
- I.e. all the processes must call the corresponding MPI routine.

- Collective routines are used for
  1) Process synchronization (`mpi_barrier`)
  2) Reduction operations (`mpi_reduce`)
  3) Boradcasting data (`mpi_bcast`, `mpi_scatter`)
  4) Collect information (`mpi_gather`)
- The routines do not have a `tag` argument.
- They are all blocking.

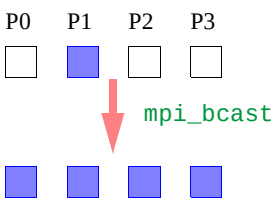
- **Synchronization** of processes is accomplished by

```
call mpi_barrier(mpi_comm_world,ierr)
```

  - Processes can continue execution only after all processes have finished the call.
  - Not needed often. Usually parallel programs are synchronized.

- Data can be **broadcasted** from the `root` process to all others by

```
call mpi_bcast(buffer,count,datatype,root,comm,ierr)
```



Parallel computations: MPI

- A simple example of `mpi_bcast`[1]:

```
program mpibcast
  use mpi
  implicit none
  integer,parameter :: tag = 50
  integer :: id,ntasks,source_id,dest_id,rc,i
  integer,dimension(mpi_status_size) :: status
  integer,dimension(2) :: msg

  call mpi_init(rc)
  call mpi_comm_size(mpi_comm_world,ntasks,rc)
  call mpi_comm_rank(mpi_comm_world,id,rc)

  if (id==0) then
     msg(1)=id
     msg(2)=ntasks
     dest_id=0
  else
     msg=0
  end if
  call mpi_bcast(msg,2,MPI_INTEGER,0,mpi_comm_world,rc)
  print *,'Proc: ',id,' msg: ',msg

  call mpi_finalize(rc)
end program mpibcast
```

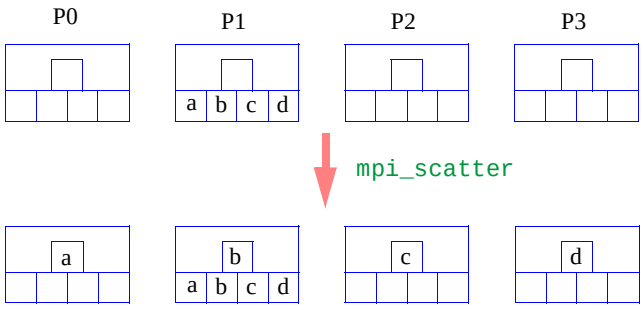
```
mpi> mpif90 mpibcast.f90
mpi> mpirun -np 4 a.out
 Proc:            0  msg:           0           4
 Proc:            1  msg:           0           4
 Proc:            3  msg:           0           4
 Proc:            2  msg:           0           4
```

1. Example programs can be found at `http://www.physics.helsinki.fi/courses/s/stltk/progs/mpi/`

Parallel computations: MPI

- Using `mpi_scatter` one process divides the data and sends a part of it to each process:

```
call mpi_scatter(sendbuf,sendcount,sendtype,recvbuf,recvcount,recvtype,root,comm,ierr)
```



- With routine `mpi_scatterv` different amount of data can be sent to different processors.

```
call mpi_scatterv(sendbuf,sendcounts,displs,snedtype,recvbuf,recvcount,&
                  & recvtype,root,comm,rc)
```

`sendcount` - integer array (of length group size) specifying the number of elements to send to each processor
`displs` - integer array. Entry `i` specifies the displacement (relative to sendbuf) from which to take the outgoing data to process `i`
`sendtype` - data type of send buffer elements
`recvcount` - number of elements in receive buffer
`recvtype` - data type of receive buffer elements
`root` - rank of sending process
`comm` - communicator

Parallel computations: MPI

- Example of `mpi_scatter`:

```
mpi> mpirun -np 4 a.out
sendbuf:  8  8  8  8  7  7  7  7  6  6  6  6  5  5  5  5
 Proc:           0  recv:          8          8          8          8
 Proc:           1  recv:          7          7          7          7
 Proc:           2  recv:          6          6          6          6
 Proc:           3  recv:          5          5          5          5
```

```
program mpiscatter
  use mpi
  implicit none
  integer,parameter :: tag = 50
  integer :: id,ntasks,source_id,dest_id,rc,i
  integer,dimension(mpi_status_size) :: status
  integer,allocatable :: sendbuf(:),recvbuf(:)
  integer :: slocal=4

  call mpi_init(rc)
  call mpi_comm_size(mpi_comm_world,ntasks,rc)
  call mpi_comm_rank(mpi_comm_world,id,rc)
  allocate(sendbuf(ntasks*slocal),recvbuf(slocal))

  if (id==0) then
     do i=1,ntasks*slocal,slocal
        sendbuf(i:i+3)=ntasks-i/4+4
     end do
     print '(a,100i3)','sendbuf:',sendbuf
  end if
  call mpi_scatter(sendbuf,slocal,mpi_integer,recvbuf,slocal,mpi_integer,0,mpi_comm_world,rc)

  print *,'Proc: ',id,' recv:',recvbuf

  call mpi_finalize(rc)
end program mpiscatter
```


Parallel computations: MPI

- Example of `mpi_scatterv`:

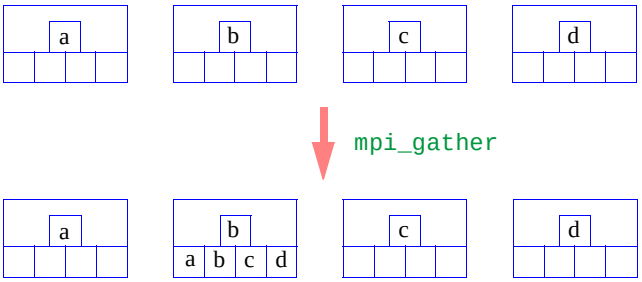
```
program mpiscatterv
  ! From http://www.csc.fi/oppaat/mpi/esimerkit/esim-3.5.f90
  implicit none
  include 'mpif.h'
  integer,parameter :: n = 4,root_id = 0
  real,dimension(:),allocatable :: recvbuf,sendbuf
  integer,dimension(:),allocatable :: sendcounts,displs
  integer :: ntasks,id,rc,recvcount,i

  call mpi_init(rc)
  call mpi_comm_size(MPI_COMM_WORLD,ntasks,rc)
  call mpi_comm_rank(MPI_COMM_WORLD,id,rc)
  if (id == root_id) then
    allocate(sendbuf(n*ntasks),displs(ntasks))
    sendbuf = 10*(/ (i,i = 1,n*ntasks) /)
    displs = n*(/ (i,i = 0,ntasks-1) /)
    allocate(sendcounts(0:ntasks-1))
    sendcounts = min(n,(/ (i,i = 1,ntasks) /))
  end if
  recvcount = min(n,id + 1)
  allocate(recvbuf(recvcount))

  recvbuf = 0.0
  call mpi_scatterv(sendbuf,sendcounts,displs,&
      MPI_REAL,recvbuf,recvcount,MPI_REAL,&
      root_id,MPI_COMM_WORLD,rc)
  write (*,'(A,I3,",",(5F8.2))') 'id,recvbuf: ',id,recvbuf

  call mpi_finalize(rc)
end program mpiscatterv
```

```
mpi> mpif90 mpi_scatterv.f90
mpi> mpirun -np 4 a.out
id,recvbuf:   0,   10.00
id,recvbuf:   3,  130.00  140.00  150.00  160.00
id,recvbuf:   2,   90.00  100.00  110.00
id,recvbuf:   1,   50.00   60.00
```

Parallel computations: MPI

- Conversely, `mpi_gather` collects data from all processes to the root process. Note that this is not a reduction operation.

```
call mpi_gather(sendbuf,sendcount,senbtype,recvbuf,recvcount,recvtype,root,comm,ierr)
```



- Similarly, with routine `mpi_gatherv` different amount of data can be collected from different processors.

```
call mpi_gatherv(sendbuf,sendcount,sendtype,recvbuf,recvcounts,&
                 & displs,recvtype,root,comm,ierror)
```


Parallel computations: MPI

- Example `mpi_gather`:

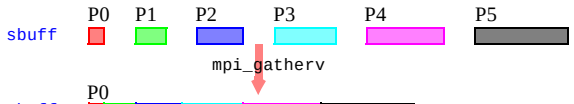
```
mpi> mpif90 mpigather.f90
mpi> mpirun -np 4 a.out
id:  0  sendbuf:  0  0  0  0
id:  1  sendbuf:  1  2  3  4
id:  2  sendbuf:  2  4  6  8
id:  3  sendbuf:  3  6  9 12
recv:  0  0  0  0  1  2  3  4  2  4  6  8  3  6  9 12
```

```
program mpigather
  use mpi
  implicit none
  integer,parameter :: tag = 50
  integer :: id,ntasks,source_id,dest_id,rc,i
  integer,dimension(mpi_status_size) :: status
  integer,allocatable :: sendbuf(:),recvbuf(:)
  integer :: slocal=4

  call mpi_init(rc)
  call mpi_comm_size(mpi_comm_world,ntasks,rc)
  call mpi_comm_rank(mpi_comm_world,id,rc)
  allocate(sendbuf(slocal),recvbuf(ntasks*slocal))

  do i=1,slocal
     sendbuf(i)=i*id
  end do
  print '(a,i3,a,100i3)','id:',id,'  sendbuf:',sendbuf

  call mpi_gather(sendbuf,slocal,mpi_integer,recvbuf,slocal,mpi_integer,0,mpi_comm_world,rc)

  if (id==0) then
     print '(a,100i3)','recv:',recvbuf
  end if

  call mpi_finalize(rc)
end program mpigather
```

Parallel computations: MPI

- Example of `mpi_gatherv`:

```
program gatherv
  use mpi
  implicit none
  integer,allocatable :: sbuff(:),rbuff(:),rcounts(:),displs(:)
  integer :: scount,ierr,i,myid,nprocs,rsize

  call mpi_init(ierr)
  call mpi_comm_size(mpi_comm_world,nprocs,ierr)
  call mpi_comm_rank(mpi_comm_world,myid,ierr)

  scount=myid+1
  rsize=nprocs*(nprocs+1)/2
  allocate(sbuff(scount),rcounts(nprocs),displs(nprocs),rbuff(rsize))
  rbuff=0
  sbuff=myid+1
  displs(1) = 0
  rcounts(1) = 1
  do i=2,nprocs
     displs(i)=displs(i-1)+i-1
     rcounts(i)=i
  enddo
  call mpi_gatherv(sbuff,scount,mpi_integer,rbuff,rcounts,displs,mpi_integer,0,mpi_comm_world,ierr)
  if (myid==0) print *,rbuff

  call mpi_finalize(ierr)
end program gatherv
```




```
 mpi> mpif90 mpi_gatherv.f90
 mpi> mpirun -np 6 a.out
           1           2           2           3           3           3
           4           4           4           4           5           5
           5           5           5           6           6           6
           6           6           6
```


Parallel computations: MPI

- MPI supports many reduction operations.

- The call has the form

```
call mpi_reduce(sendbuf,recvbuf,count,datatype,op,root,comm,ierror)
```

  - Routine does the reduction operation `op` for the elements in `sendbuf` and stores it in `recvbuf` in process `root`.
  - Possible operations are

`MPI_MAX`	`maximum value`
`MPI_MIN`	`minimum value`
`MPI_SUM`	`sum`
`MPI_PROD`	`product`
`MPI_LAND`	`logical and`
`MPI_BAND`	`bitwise and`
`MPI_LOR`	`logical or`
`MPI_BOR`	`bitwise or`
`MPI_LXOR`	`logical xor`
`MPI_BXOR`	`bitwise xor`
`MPI_MAXLOC`	`maximum value and location`
`MPI_MINLOC`	`minimum value and location`

  - Users can also define their own reduction operations with routine `mpi_op_create`:

```
call mpi_op_create(func,commute,operator,ierr)
```

  - Routine `mpi_allreduce` does the same as `mpi_reduce` except the it also broadcasts the result to all processors. Thus, there is no need for the `root` argument:

```
call mpi_allreduce(sendbuf,recvbuf,count,datatype,op,comm,ierror)
```

Parallel computations: MPI

- Example of `mpi_reduce`:

```
program reduce_maxloc
  use sizes
  !use mpi
  implicit none
  include 'mpif.h'
  integer,parameter :: ROOT=0,BLEN=5
  integer :: ierr,procnum,numprocs,nlen
  real(rk) :: x(2,BLEN),maxx(2,BLEN)
  character(len=80) :: host,argu
  integer :: s
  integer,allocatable :: seed(:)
  call mpi_init(ierr)
  call mpi_comm_rank(mpi_comm_world,procnum,ierr)
  call mpi_comm_size(mpi_comm_world,numprocs,ierr)
  call mpi_get_processor_name(host,nlen,ierr)
  call random_seed(size=s)
  allocate(seed(s))
  seed=getseed()+procnum
  call random_seed(put=seed)
  call random_number(x(1,:))
  x(2,:)=procnum
  maxx=0
  print '(a,i2,a,i2,a40,10f6.3)','Proc ',procnum,&
     &' out of ',numprocs,' at '//trim(host)//': x=',x(1,:)
  call mpi_reduce(x,maxx,BLEN,MPI_2DOUBLE_PRECISION,MPI_MAXLOC,&
                &ROOT,MPI_COMM_WORLD,ierr)
  if (procnum==0) then
     print '(a,10f6.3)','Max is ',maxx(1,:)
     print '(a,10i6)',  '       ',int(maxx(2,:))
  end if
  call mpi_finalize(ierr)
  stop
contains
  function getseed()
    implicit none
    integer :: getseed
    integer :: t(8),i
    call date_and_time(values=t)
    getseed=t(7)+60*(t(6)+60*(t(5)+&
     &24*(t(3)-1+31*(t(2)-1+12*t(1)))))+t(8)
  end function getseed
end program reduce_maxloc
```

Note that when using `MPI_MAXLOC` or `MPI_MINLOC` with Fortran you have to use a 2D array with the first column giving the actual value and the second the processor id number. Awkward, but it works.
In C `structs` are used:

```
struct {
     double value;
     int rank;
   } x, maxx;
MPI_Reduce(&x,&maxx,1,MPI_DOUBLE_INT,
                MPI_MAXLOC,root,
                MPI_COMM_WORLD);
```

```
mpi> mpif90 mpi_reduce_maxloc.f90
mpi> mpirun -np 4 a.out
Proc  0 out of  4         at mill.physics.helsinki.fi: x= 0.361 0.648 0.649 0.176 0.380
Max is  0.361 0.648 0.728 0.525 0.380
              1     0     3     1     0
Proc  2 out of  4               at compute-1-1.local: x= 0.361 0.075 0.390 0.524 0.311
Proc  3 out of  4               at compute-1-2.local: x= 0.361 0.425 0.728 0.164 0.154
Proc  1 out of  4               at compute-1-0.local: x= 0.361 0.228 0.505 0.525 0.090
```


Parallel computations: MPI

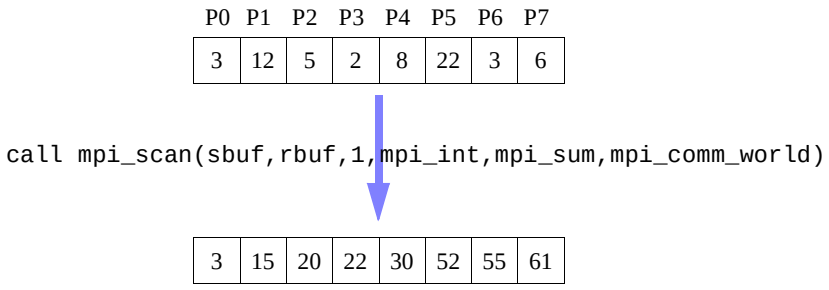
- Reduce and scatterv operations are combined in the routine `mpi_reduce_scatter`:

```
call mpi_reduce_scatter(sendbuf,recvbuf,recvcounts,mpi_datatype,&
                        & mpi_operator,comm,ierr)
```

- Routine `mpi_scan` is used to perform a prefix reduction on data distributed across the group:

```
call mpi_scan(sendbuf,recvbuf,count,datatype,op,comm,ierror)
```

  - The operation returns, in the receive buffer of the process with rank `i`, the reduction of the values in the send buffers of processes with ranks `0,...,i` (inclusive).
  - The type of operations supported, their semantics, and the constraints on send and receive buffers are as for `mpi_reduce`.



Parallel computations: MPI

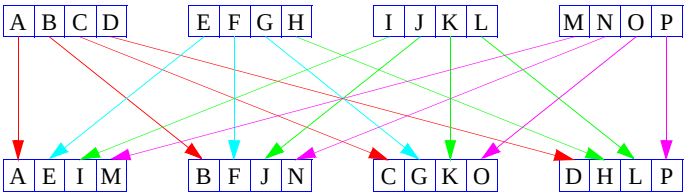
- With routine `mpi_alltoall` every process send a block of data to all other processes:

```
call mpi_alltoall(sendbuf,sendcount,sendtype,recvbuf,recvcnt,recvtype,comm,ierr)
```

- Example:

```
program alltoall
  use mpi
  implicit none
  integer,parameter :: mlen=1
  integer :: i,ierr,myid,nprocs,ac
  character(len=mlen),allocatable :: sb(:),rb(:)
  call mpi_init(ierr)
  call mpi_comm_rank(mpi_comm_world,myid,ierr)
  call mpi_comm_size(mpi_comm_world,nprocs,ierr)
  allocate(sb(0:nprocs-1),rb(0:nprocs-1))
  ac=ichar("A")
  do i=0,nprocs
     sb(i)=char(ac+i+myid*nprocs)
  end do
  print *,'Before: ',myid,sb
  call mpi_alltoall(sb,mlen,mpi_character,rb,mlen,mpi_character,mpi_comm_world,ierr)
  print *,'After:  ',myid,rb
  call mpi_finalize(ierr)
 end program alltoall
```

```
mpi> mpirun -np 4 a.out
 Before:           0 ABCD
 Before:           1 EFGH
 Before:           2 IJKL
 Before:           3 MNOP
 After:            0 AEIM
 After:            1 BFJN
 After:            2 CGKO
 After:            3 DHLP
```

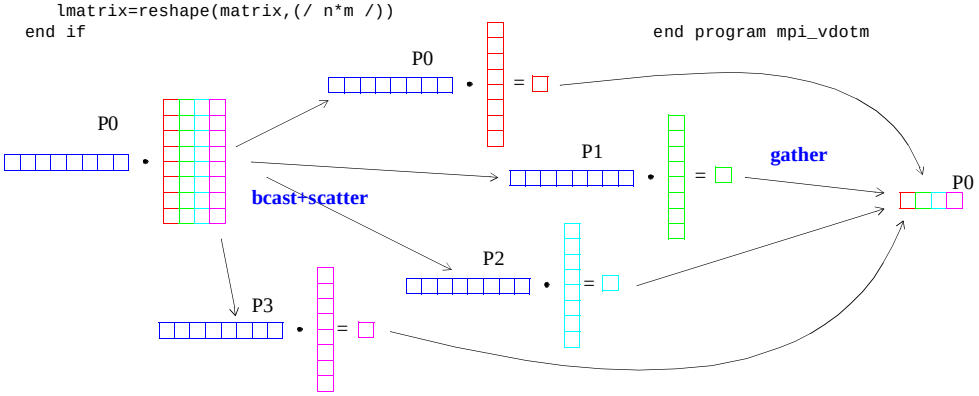


- With routine `mpi_alltoallv` blocks of different size can be sent to all other processes.

Parallel computations: MPI

- A simple example of linear algebra: parallel multiplication of a matrix with a vector:

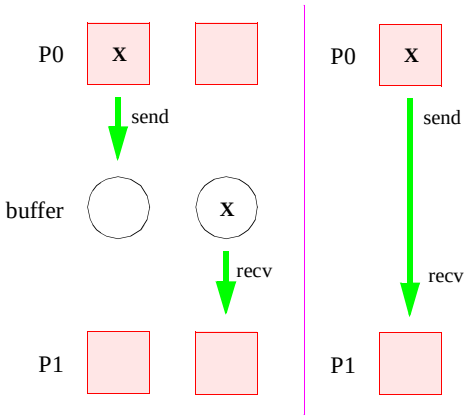
```
program mpi_vdotm
  use mpi
  use sizes
  implicit none
  real(rk),allocatable :: matrix(:,:),vector(:),&
       &prod(:),lmatrix(:),column(:)
  real(rk) :: s
  integer,parameter :: n=4
  integer :: ierr,i,myid,nprocs,m

  call mpi_init(ierr)
  call mpi_comm_size(mpi_comm_world,nprocs,ierr)
  call mpi_comm_rank(mpi_comm_world,myid,ierr)
  m=nprocs
  allocate(matrix(n,m),vector(n),column(n),prod(m),&
       &lmatrix(n*m))
  if (myid==0) then
     call random_number(matrix)
     call random_number(vector)
     lmatrix=reshape(matrix,(/ n*m /))
  end if
  call mpi_bcast(vector,n,mpi_double_precision,0,&
       &mpi_comm_world,ierr)
  call mpi_scatter(lmatrix,n,mpi_double_precision,&
  &column,n,mpi_double_precision,0,mpi_comm_world,ierr)
  s=0.0
  do i=1,n
     s=s+vector(i)*column(i)
  end do

  call mpi_gather(s,1,mpi_double_precision,prod,1,&
       &mpi_double_precision,0,mpi_comm_world,ierr)
  if (myid==0) then
     print '(a,100f10.6)','Parallel: ',prod
     print '(a,100f10.6)','Matmul:   ',&
       &matmul(vector,matrix)
  end if

  call mpi_finalize(ierr)
  stop
end program mpi_vdotm
```



Parallel computations: MPI

• *Different modes can be used in p2p[1] communication depending whether the message is buffered.*
- Buffering decouples the send and receive operations enabling the sending routine to return early.
- There are four modes in p2p communication MPI:
  1. Standard
  - Mode used in mpi_send. MPI decides whether buffering is used. As already said don't trust that the message is buffered.
  2. Buffered
  - User must provide the buffer space. Does not need the matching receive operation to be already started.
  3. Synchronous
  - Waits until the matching receive operation is finished.
  4. Ready
  - Assumes that the matching reveice operation is already started.
- There is a send routine for all four modes (argument list as in `mpi_send`):

standard:	`mpi_send`
buffered:	`mpi_bsend`
synchronous:	`mpi_ssend`
ready:	`mpi_rsend`



- When using `mpi_bsend` buffer space must be reserved by the routine `mpi_buffer_attach`:

```
integer,parameter :: bufsize=10000
integer :: buffer(bufsize)
call mpi_buffer_attach(buffer,bufsize,ierr)
```

- Buffer space is freed by

```
call mpi_buffer_detach(buffer,bufsize,ierr)
```

1. point-to-point

Parallel computations: MPI

- Communication should be designed in such a way that deadlocks never occur (examples for two processors):

Works always

```
if (myid==0) then
   call mpi_send(sbuf,N,mpi_integer,1,tag,mpi_comm_world,ierr)
   call mpi_recv(rbuf,N,mpi_integer,1,mpi_any_tag,mpi_comm_world,status,ierr)
else if (myid==1) then
   call mpi_recv(rbuf,N,mpi_integer,0,mpi_any_tag,mpi_comm_world,status,ierr)
   call mpi_send(sbuf,N,mpi_integer,0,tag,mpi_comm_world,ierr)
end if
```

Deadlocks always

```
if (myid==0) then
   call mpi_recv(rbuf,N,mpi_integer,1,mpi_any_tag,mpi_comm_world,status,ierr)
   call mpi_send(sbuf,N,mpi_integer,1,tag,mpi_comm_world,ierr)
else if (myid==1) then
   call mpi_recv(rbuf,N,mpi_integer,0,mpi_any_tag,mpi_comm_world,status,ierr)
   call mpi_send(sbuf,N,mpi_integer,0,tag,mpi_comm_world,ierr)
end if
```

Probably works

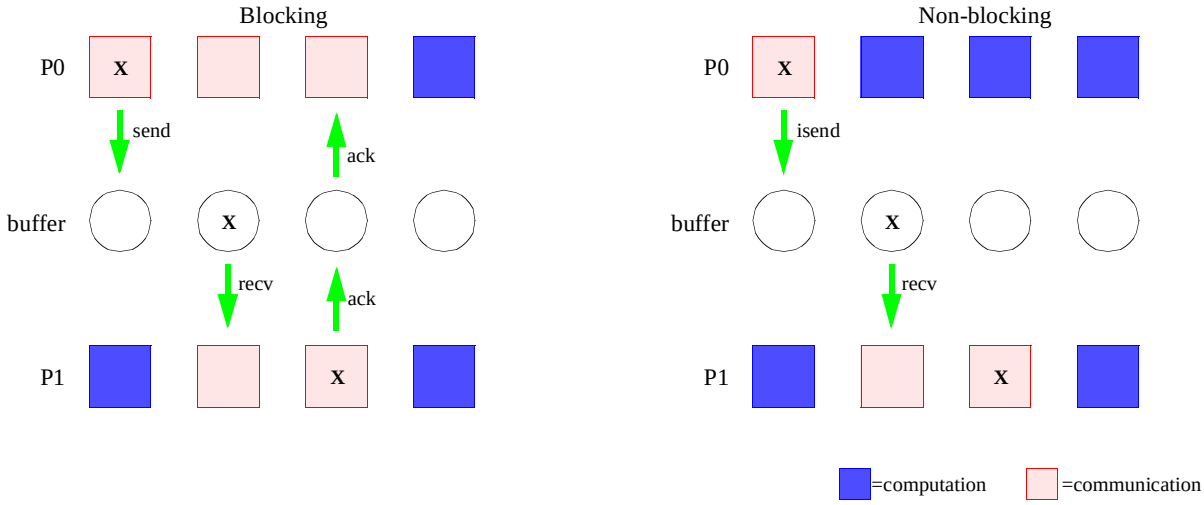
```
if (myid==0) then
   call mpi_send(sbuf,N,mpi_integer,1,tag,mpi_comm_world,ierr)
   call mpi_recv(rbuf,N,mpi_integer,1,mpi_any_tag,mpi_comm_world,status,ierr)
else if (myid==1) then
   call mpi_send(sbuf,N,mpi_integer,0,tag,mpi_comm_world,ierr)
   call mpi_recv(rbuf,N,mpi_integer,0,mpi_any_tag,mpi_comm_world,status,ierr)
end if
```

- The may be differences in the performance of the different modes. Buffering may slow down the communication (message must be copied) and consumes memory.
- The best performace can be achieved by the synchronous mode.
- However, in order to see differences large message sizes may be needed.

Parallel computations: MPI

- In addition to different modes, routines can be divided to the following classes based on *when they return*.
  1. Routines that wait for the operation to be finished. The are called **blocking** routines.
     - Actually the `mpi_send` routine returns when the receiving end has begun its operation or when the outgoing data is copied to a system buffer. So, in order to be on the safe side assume that `mpi_send` returns only after receive has started.
  2. Routines that return the control immediately to the caller program (**non-blocking**).
- When the operation is finished can be checked by using the query subroutines `mpi_wait` and `mpi_test`.

- By using non-blocking routines you can overlap computation and communication:



Parallel computations: MPI

- Non-blocking version of sending is `MPI_Isend`[1]:

```
call mpi_isend(buf,N,mpi_datatype,dest,tag,mpi_comm_world,request,ierr)
```

- Here the additional argument `request` enables queries about the operation. It is of type `integer` in Fortran and `MPI_Request` in C.

- Similarly is defined the receiving routine:

```
call mpi_irecv(buf,N,mpi_datatype,src,tag,mpi_comm_world,request,rc)
```

- Routines `mpi_wait` and `mpi_test` can be used to query the faith of the matching receive operation:

```
call mpi_test(request, done, status, rc)
call mpi_wait(request, status, rc)
```

- `mpi_test` checks the status of the request. Logical variable `done` tells whether the operation has been completed.
- `mpi_wait` waits until the operation is completed.
  - The meaning of 'completed' depends on the communication mode.
  - You should not reuse the send buffer before the send is completed.

- All communication modes have an immediate version for the sending routine:

```
mpi_isend
mpi_ibsend
mpi_issend
mpi_irsend
```

1. Immediately returning send.

Parallel computations: MPI

- An example from the CSC MPI guide:

```
program mpi_nonblocking
  ! From http://www.csc.fi/oppaat/mpi/esimerkit/esim-4.1.f90
  use mpi
  implicit none
  integer,parameter :: n = 100000,tag = 100
  integer,dimension(n) :: a
  integer :: rc,id,comm_request,i
  integer,dimension(MPI_STATUS_SIZE) :: status
  logical :: comm_done

  call mpi_init(rc)
  call mpi_comm_rank(MPI_COMM_WORLD,id,rc)

  a=(/ (i,i=1,size(a)) /)

  if (id==0) then
     call mpi_isend(a,n,MPI_INTEGER,1,tag,&
         &MPI_COMM_WORLD,comm_request,rc)
     print *,'Message sent from ',id,&
         &' Let''s do something else.'
     do
        call mpi_test(comm_request,comm_done,status,rc)
        if (comm_done) exit
        ! Do something else. Don't touch array a.
        print *,'Doing other things in ',id
     end do
  else if (id==1) then
     call mpi_irecv(a,n,MPI_INTEGER,0,tag,&
         &MPI_COMM_WORLD,comm_request,rc)
     ! Do something else. Don't touch array a.
     print *,'Doing other things in ',id
     call mpi_wait(comm_request,status,rc)
     print *,'Message received at ',id
  end if

  print *,'Finishing at ',id
  call mpi_finalize(rc)

end program mpi_nonblocking
```


Parallel computations: MPI

- Another example (cf. exercise 9, problem 1):

```
program mpi_testtest
  use sizes
  use mpi
  implicit none
  integer,parameter :: root=0,tag=1
  integer :: rc,myid,nprocs,req,&
        &status(mpi_status_size)
  integer :: count,winner,i,s
  integer,allocatable :: seed(:)
  real(rk) :: x
  real(rk),parameter :: limit=1e-7
  logical :: done

  call mpi_init(rc)
  call mpi_comm_rank(mpi_comm_world,myid,rc)
  call mpi_comm_size(mpi_comm_world,nprocs,rc)

  count=0
  call mpi_irecv(winner,1,mpi_integer,&
       &mpi_any_source,tag,mpi_comm_world,req,rc)
  call random_seed(size=s)
  allocate(seed(s))
  seed=getseed()+myid
  call random_seed(put=seed)

searchloop: do
   call random_number(x)
   if (x<limit) then
      print *,myid,': found ',x
      do i=0,nprocs-1
         call mpi_send(myid,1,mpi_integer,&
             &i,tag,mpi_comm_world,rc)
      end do
      exit searchloop
   end if
   call mpi_test(req,done,status,rc)
   if (done) then
      print *,myid,': lost to ',winner
      exit searchloop
   end if
end do searchloop

  call mpi_finalize(rc)
contains
  function getseed()
   getseed=... ! seed from time
  end function getseed
end program mpi_testtest
```

```
mpi> mpif90 mpi_testtest.f90
mpi> mpirun -np 4 a.out
           0 : lost to            1
           2 : lost to            1
           1 : found   8.521601895026920E-008
           3 : lost to            1
mpi> mpirun -np 4 a.out
           0 : lost to            1
           1 : found   9.778887420522695E-009
           2 : lost to            1
           3 : lost to            1
```

But what would happen if we would decrease `limit` to, say, 1e-4?

Parallel computations: MPI

• *In MPI there are two concepts that are used to classify processes:* ***communicators*** *and* ***groups***[1]*:*

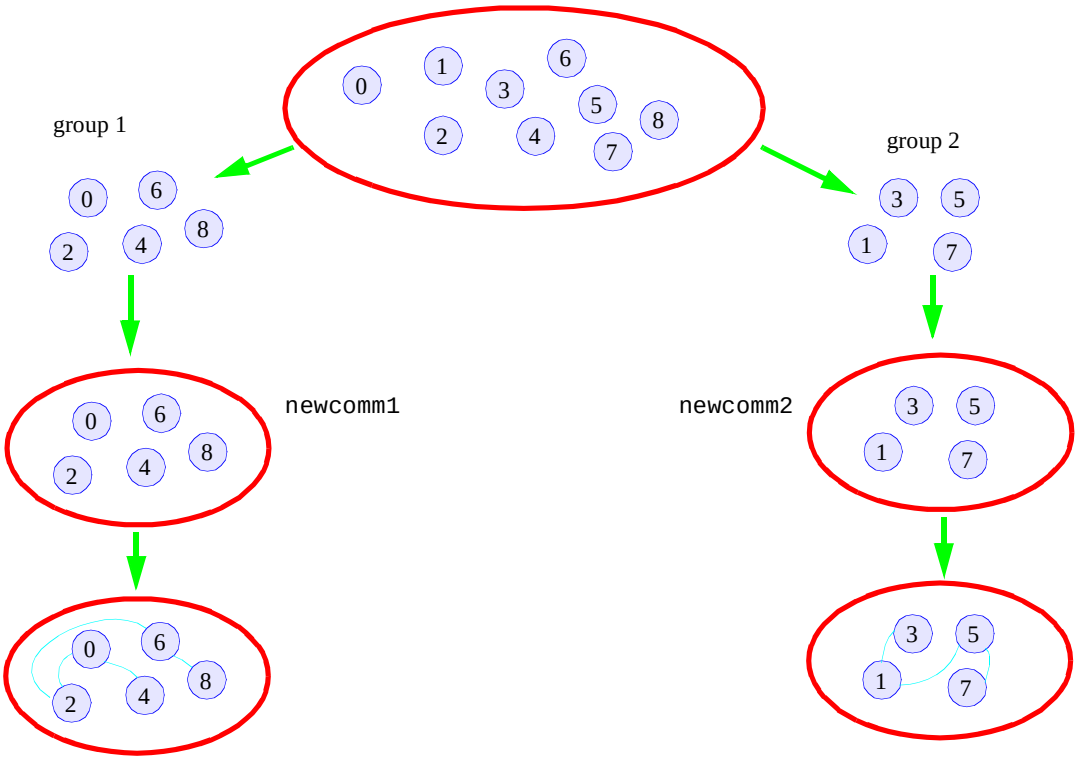
- A communicator is an opaque object with a number of attributes together with simple rules that govern its creation, use, and destruction.
- The communicator determines the scope and the "communication universe" in which a point-to-point or collective operation is to operate.
- Each communicator contains a group of valid participants. The source and destination of a message is identified by process rank within that group.
- Intracommunicator is the communicator used for communicating within a single group of processes.
- Intercommunicator is used for communicating within two or more groups of processes. In MPI-1, an intercommunicator is used for point-to-point communication between two disjoint groups of processes.
- Communicators are dynamic, i.e., they can be created and destroyed during program execution.

- A group is an ordered set of processes. Each process in a group is associated with a unique integer rank. Rank values start at zero and go to `N-1`, where `N` is the number of processes in the group.
- One process can belong to two or more groups.
- Groups are represented by opaque group objects, and hence cannot be directly transferred from one process to another.
- A group is used within a communicator to describe the participants in a communication "universe" and to rank such participants.
- A group always includes the same local process. The source and destination of a message is identified by process rank within that group.
- Group is a dynamic object in MPI and can be created and destroyed during program execution.

1. Quoting http://www.msi.umn.edu/tutorial/MPI/content_communicator.html

Parallel computations: MPI

• *The programmer can create a group and associate a communicator with that group.*



Parallel computations: MPI

- Typical usage of groups and communicators
  - Extract handle of global group from `mpi_comm_world` using `MPI_Comm_group`
  - Form new group as a subset of global group using `MPI_Group_incl`
  - Create new communicator for new group using `MPI_Comm_create`
  - Determine new rank in new communicator using `MPI_Comm_rank`
  - Conduct communications using any MPI message passing routines
  - When finished, free up new communicator and group using `MPI_Comm_free` and `MPI_Group_free`
  - Example:

```
program mpi_groups1
  use mpi
  implicit none
  integer :: ierr,myid,nprocs,rank,new_myid,sendbuf,recvbuf,ranks1(4),ranks2(4)
  integer :: orig_group,new_group,new_comm
  ranks1=(/ 0,1,2,3 /); ranks2=(/ 4,5,6,7 /)
  call mpi_init(ierr)
  call mpi_comm_rank(mpi_comm_world,myid,ierr)
  call mpi_comm_size(mpi_comm_world,nprocs,ierr)
  if (nprocs/=8) then
     if (myid==0) print *,'NP should be 8.'
     call mpi_finalize(ierr)
     stop
  end if
  sendbuf=myid
  call mpi_comm_group(mpi_comm_world,orig_group,ierr)
  if (myid<nprocs/2) then
     call mpi_group_incl(orig_group,nprocs/2,ranks1,new_group,ierr)
  else
     call mpi_group_incl(orig_group,nprocs/2,ranks2,new_group,ierr)
  end if
  call mpi_group_rank(new_group,new_myid,ierr)
  call mpi_comm_create(mpi_comm_world,new_group,new_comm,ierr)
  call mpi_allreduce(sendbuf,recvbuf,1,mpi_integer,mpi_sum,new_comm,ierr)
  print * ,'myid ',myid,' new_myid ',new_myid,' recvbuf ',recvbuf

  call mpi_finalize(ierr)
end program mpi_groups1
```

```
mpi> mpirun -np 8 a.out | sort
 myid          0  new_myid          0  recvbuf          6
 myid          1  new_myid          1  recvbuf          6
 myid          2  new_myid          2  recvbuf          6
 myid          3  new_myid          3  recvbuf          6
 myid          4  new_myid          0  recvbuf         22
 myid          5  new_myid          1  recvbuf         22
 myid          6  new_myid          2  recvbuf         22
 myid          7  new_myid          3  recvbuf         22
```

Example program from http://www.msi.umn.edu/tutorial/MPI/content_communicator.html translated to Fortran.

Parallel computations: MPI

- A new communicator can be created from the old one by routine `mpi_comm_dup`:

```
call mpi_comm_dup(comm,newcomm,ierr)
```

  - Here a new communicator `newcomm` is created as a copy of the old one. Process ranks are inherited from the old.

- Creation of a new communicator can be done using the routines mentioned above. Another way is to use `mpi_comm_split`:

```
call mpi_comm_split(comm,color,key,newcomm,ierr)
```

  - Here `comm` is the communicator from which the group is formed (e.g. `mpi_comm_world`) and `newcomm` is the handle for the new communicators of the groups.
  - Note that if there are more than one value for `color` within the processes calling the routine, more than one commnunicators are created.
  - All processes that have the same `color` belong the the same group in the new communicator.
  - The rank of the process in the new group is based on `key`. If more than one process have the same `key` the order is determined based on the old communicator `comm`.
  - Example:

```
program comm_split
  use mpi
  implicit none
  integer :: myid,myid1,ntasks,ntasks1,color,newcomm,rc

  call mpi_init(rc)
  call mpi_comm_size(mpi_comm_world,ntasks,rc)
  call mpi_comm_rank(mpi_comm_world,myid,rc)
  call mpi_comm_split(mpi_comm_world,mod(myid,4),myid,newcomm,rc)
  call mpi_comm_size(newcomm,ntasks1,rc)
  call mpi_comm_rank(newcomm,myid1,rc)
  print '(4i6)',myid,myid1,ntasks,ntasks1

  call mpi_finalize(rc)
end program comm_split
```

```
mpi> mpirun -np 8 a.out | sort
     0     0     8     2
     1     0     8     2
     2     0     8     2
     3     0     8     2
     4     1     8     2
     5     1     8     2
     6     1     8     2
     7     1     8     2
```

Parallel computations: MPI

- A more elaborate example of `mpi_comm_split`[1]:

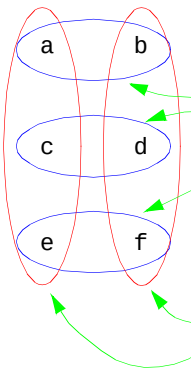
```
program mpi_groups
  ! From http://scv.bu.edu/SCV/Tutorials/MPI/
  use mpi
  implicit none
  integer :: irow,jcol,i,j
  integer,parameter :: nrow=3,mcol=2,ndim=2
  integer :: p,ierr,row_comm,col_comm,comm2D
  integer :: myid,me,row_id,col_id
  integer :: row_group,row_key,map(0:5)=(/2,1,2,1,0,1/)

  call MPI_Init(ierr)
  call MPI_Comm_rank(MPI_COMM_WORLD,myid,ierr)
  call MPI_Comm_size(MPI_COMM_WORLD,p,ierr)
  if(myid .eq. 0) then
     print *
     print *,'Example of MPI_Comm_split Usage'
     print *,'Split 3x2 grid into 2 different communicators'
     print *,'which correspond to 3 rows and 2 columns.'
     print *
     print *,'       myid    irow    jcol  row-id  col-id'
  endif

  irow = myid/mcol
  jcol = mod(myid,mcol)
  comm2D = MPI_COMM_WORLD
  call MPI_Comm_split(comm2D,irow,jcol,row_comm,ierr)
  call MPI_Comm_split(comm2D,jcol,irow,col_comm,ierr)

  call MPI_Comm_rank(row_comm,row_id,ierr)
  call MPI_Comm_rank(col_comm,col_id,ierr)
  call MPI_Barrier(MPI_COMM_WORLD,ierr)

  if(myid .eq. 0) then
     print *
     print *,'Next,create more general communicator'
     print *,'which consists of two groups :'
     print *,'Rows 1 and 2 belongs to group 1 and row 3 is group 2'
     print *
  endif

  row_group = myid/4
  row_key = myid - row_group*4    ! group1:0,1,2,3; group2:0,1
  call MPI_Comm_split(comm2D,row_group,row_key,row_comm,ierr)
  call MPI_Comm_rank(row_comm,row_id,ierr)
  print '(a,2x,9i8)','b',myid,row_id
  call MPI_Barrier(MPI_COMM_WORLD,ierr)

  if(myid .eq. 0) then
     print *
     print *,'If two processes have same key,the ranks'
     print *,'of these two processes in the new'
     print *,'communicator will be ordered according'
     print *,'to their order in the old communicator'
     print *,' key = map(myid); map = (2,1,2,1,0,1)'
     print *
  endif

  row_group = myid/4
  row_key = map(myid)
  call MPI_Comm_split(comm2D,row_group,row_key,row_comm,ierr)
  call MPI_Comm_rank(row_comm,row_id,ierr)
  call MPI_Barrier(MPI_COMM_WORLD,ierr)
```

1. From http://scv.bu.edu/SCV/Tutorials/MPI/alliance/communicators/codes/comm_split_example.html

Parallel computations: MPI

- And running the code:

```
mpi> mpirun -np 6 a.out

 Example of MPI_Comm_split Usage
 Split 3x2 grid into 2 different communicators
 which correspond to 3 rows and 2 columns.

       myid    irow    jcol  row-id  col-id
         0       0       0       0       0
         1       0       1       1       0
         2       1       0       0       1
         3       1       1       1       1
         4       2       0       0       2
         5       2       1       1       2

 Next,create more general communicator
 which consists of two groups :
 Rows 1 and 2 belongs to group 1 and row 3 is group 2

         0       0
         1       1
         2       2
         3       3
         4       0
         5       1

 If two processes have same key,the ranks
 of these two processes in the new
 communicator will be ordered according
 to their order in the old communicator
  key = map(myid); map = (2,1,2,1,0,1)

         0       2
         1       0
         2       3
         3       1
         4       0
         5       1
```




```
        a   b   c   d   e   f
myid    0   1   2   3   4   5
irow    0   0   1   1   2   2
jcol    0   1   0   1   0   1

row_comm:  irow=0   a b
           irow=1   c d
           irow=2   e f

col_comm:  jcol=0   a c e
           jcol=1   b d f
```

Parallel computations: MPI

• *In many parallel problems it is useful to define* **topologies** *for groups of processes.*

- Topologies can be either
  1) Cartesian: processors form e.g. a simple cubic lattice.
  2) Graph topologies: more general.
- Defining a topology means that we map the linear process rank to 2D or 3D virtual rank.
- Topology is associated with a communicator.
- In many parallel problems using domain decomposition a common communication pattern is the exchange of information between neighboring processes.
  - Using cartesian topologies one can easily determine the ranks of the neighbors.

- To create a new communicator with cartesian topology `mpi_cart_create` is called:

```
call mpi_cart_create(oldcomm,ndims,dims,periodic,reorder,newcomm,ierr)
```

`oldcomm` is the communicator from which `newcomm` is created
`ndims` gives the number of dimensions
array `dims` gives the number of processes in each dimension `1..ndims`
array `periodic` tells whether each dimension is periodic or not
`reorder` tells whether the order of processes can be changed in order to make communication more efficient.

- With routine `mpi_cartdim_get` the number of dimensions can be queried:

```
call mpi_cartdim_get(comm,ndims,ierr)
```

- Routine `mpi_cart_get` return information about the current process and a cartesian communicator:

```
call mpi_cart_get(comm,maxdims,dims,periodic,coords,ierr)
```

- Routine `mpi_cart_rank` returns the rank of the process in the communicator:

```
call mpi_cart_rank(comm,coords,rank,ierr)
```

- Routine `mpi_cart_coords` is the inverse of mpi_cart_rank: it returns the coordinates corresponding to a given rank

```
call mpi_cart_coords(comm,rank,maxdims,coords,ierr)
```


Parallel computations: MPI

- With routine `mpi_cart_shift` one can find out the neighboring processes in the defined topology:

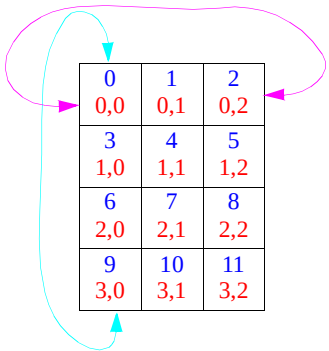
```
call mpi_cart_shift(comm,dir,disp,source,dest,ierr)
```

here `dir` tells in which dimension the system is shifted and `disp` tells how much,
`source` and `dest` return the ranks of the source and destination processes.

- Example:

```
program mpi_topo1
  use mpi
  implicit none
  integer,parameter :: ndims=2
  integer :: rc,myid,nprocs,comm,dims(ndims)
  integer :: coords(ndims),source,dest,dir
  logical :: periods(ndims),reorder=.true.

  call mpi_init(rc)
  call mpi_comm_rank(mpi_comm_world,myid,rc)
  call mpi_comm_size(mpi_comm_world,nprocs,rc)

  periods=.true.
  dims=0          <-- convenience function to determine dimensions
  call mpi_dims_create(nprocs,ndims,dims,rc)
  if (myid==0) print *,dims
  call mpi_cart_create(mpi_comm_world,ndims,&
       &dims,periods,reorder,comm,rc)

  do dir=0,1
     call mpi_cart_coords(comm,myid,ndims,coords,rc)
     call mpi_cart_shift(comm,dir,1,source,dest,rc)
     print '(i2,5x,i3.2,2(3x,2i4))',dir,myid,&
        &coords,source,dest
  end do

  call mpi_finalize(rc)
end program mpi_topo1
```

```
mpi> mpirun -np 12 a.out | sort -n
 *** dims           4           3

dir     id      x   y    src dst
---     --      -   -     --  --
 0      00      0   0      9   3
 0      01      0   1     10   4
 0      02      0   2     11   5
 0      03      1   0      0   6
 0      04      1   1      1   7
 0      05      1   2      2   8
 0      06      2   0      3   9
 0      07      2   1      4  10
 0      08      2   2      5  11
 0      09      3   0      6   0
 0      10      3   1      7   1
 0      11      3   2      8   2

 1      00      0   0      2   1
 1      01      0   1      0   2
 1      02      0   2      1   0
 1      03      1   0      5   4
 1      04      1   1      3   5
 1      05      1   2      4   3
 1      06      2   0      8   7
 1      07      2   1      6   8
 1      08      2   2      7   6
 1      09      3   0     11  10
 1      10      3   1      9  11
 1      11      3   2     10   9
```



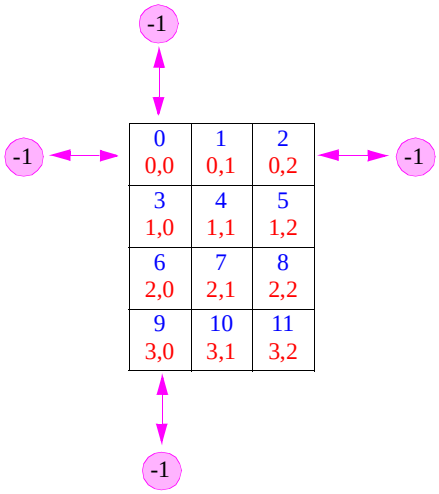
Note the row-major numbering.

Parallel computations: MPI

- Without periodic boundaries we get:

```
...
periods=.false.
...
```

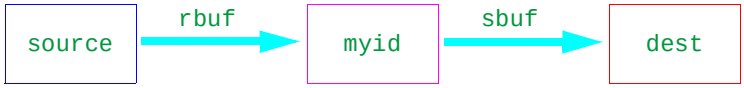
```
mpi> mpirun -np 12 a.out| sort -n

dir     id      x   y    src dst
---     --      -   -     --  --
 0      00      0   0     -1   3
 0      01      0   1     -1   4
 0      02      0   2     -1   5
 0      03      1   0      0   6
 0      04      1   1      1   7
 0      05      1   2      2   8
 0      06      2   0      3   9
 0      07      2   1      4  10
 0      08      2   2      5  11
 0      09      3   0      6  -1
 0      10      3   1      7  -1
 0      11      3   2      8  -1

 1      00      0   0     -1   1
 1      01      0   1      0   2
 1      02      0   2      1  -1
 1      03      1   0     -1   4
 1      04      1   1      3   5
 1      05      1   2      4  -1
 1      06      2   0     -1   7
 1      07      2   1      6   8
 1      08      2   2      7  -1
 1      09      3   0     -1  10
 1      10      3   1      9  11
 1      11      3   2     10  -1
```



Parallel computations: MPI

- `mpi_cart_shift` is often used in combination with `mpi_sendrecv`:

```
call mpi_sendrecv(sbuf,scount,stype,dest,stag,&
                  & rbuf,rcount,rtype,source,rtag,comm,status,ierr)
```



- Example:

```
program mpi_sendrecv1
  use mpi
  implicit none
  integer,parameter :: ndims=2,tag=999
  integer :: rc,myid,nprocs,comm,dims(ndims)
  integer :: coords(ndims),source,dest,dir
  integer :: sbuf,rbuf,status(mpi_status_size)
  logical :: periods(ndims),reorder=.true.

  call mpi_init(rc)
  call mpi_comm_rank(mpi_comm_world,myid,rc)
  call mpi_comm_size(mpi_comm_world,nprocs,rc)
  periods=.true.
  dims=0
  call mpi_dims_create(nprocs,ndims,dims,rc)
  call mpi_cart_create(mpi_comm_world,ndims,dims,periods,reorder,comm,rc)

  sbuf=myid
  do dir=0,1
     call mpi_cart_shift(comm,dir,1,source,dest,rc)
     call mpi_sendrecv(sbuf,1,mpi_integer,dest,tag,rbuf,1,mpi_integer,source,tag,comm,status,rc)
     print '(i2,5x,i4.2,i4)',dir,myid,rbuf
  end do

  call mpi_finalize(rc)
end program mpi_sendrecv1
```

Parallel computations: MPI

- The convenience routine `mpi_dims_create` helps the user to create a balanced distribution of processes per coordinate direction:

```
call mpi_dims_create(nprocs,ndims,dims)
```

`nprocs` is the number of processors
`ndims` is the number of dimensions
`dims` is the integer array with `ndims` elements and after the call contains the size of the grid in each dimension `1..ndims`

- If element `dims(i)>0` in calling the routine then dimension `i` is not modified. In this way certain dimension can be forced to have a predetermined size.

```
...

periods=.true.
dims=(/2,0/)
call mpi_dims_create(nprocs,ndims,dims,rc)
call mpi_cart_create(mpi_comm_world,ndims,dims,&
     &periods,reorder,comm,rc)

sbuf=myid
do dir=0,1
   call mpi_cart_shift(comm,dir,1,source,dest,rc)
   call mpi_sendrecv(sbuf,1,mpi_integer,dest,tag,&
                  & rbuf,1,mpi_integer,source,tag,comm,&
                & status,rc)
   print '(i2,5x,i4.2,i4)',dir,myid,rbuf
end do

...
```

```
mpi> mpirun -np 12 a.out | sort -n
 0       00   6
 0       01   7
 0       02   8
 0       03   9
 0       04  10
 0       05  11
 0       06   0
 0       07   1
 0       08   2
 0       09   3
 0       10   4
 0       11   5
 1       00   5
 1       01   0
 1       02   1
 1       03   2
 1       04   3
 1       05   4
 1       06  11
 1       07   6
 1       08   7
 1       09   8
 1       10   9
 1       11  10
```

0 0,0	1 0,1	2 0,2	3 0,3	4 0,4	5 0,5
6 1,0	7 1,1	8 1,2	9 1,3	10 1,4	11 1,5

Parallel computations: MPI

- More general topologies can be created by the `mpi_graph_create`:

```
call mpi_graph_create(oldcomm,nnodes,index,edges,reorder,newcomm,ierr)
```

`nnodes` is the number of nodes (processes) in the graph
`index(i)` is the total number of neighbors of the first `i` nodes
`edges` holds the neighbors of the each node it is accessed with the help of array index
Other parameters are as in `mpi_cart_create`.

- For example assume we have 4 processes 0,1,2,3 with the following adjacency matrix:

```
process  neighbors
     0   1,3
     1   0
     2   3
     3   0,2
```

- Then the input data for the routine is

```
nnodes = 4
index  = 2,3,4,6
edges  = 1,3,0,3,0,2
```

- We will not go into more details of the graph topologies. The definite source of information on graph topologies (and anything else) in MPI is the MPI standard itself: **http://www.mpi-forum.org/docs/mpi1-report.pdf**

Parallel computations: MPI

• *As mentioned earlier it is possible to define* **user's own data types** *in MPI.*

- These **derived datatypes** can be used all MPI communication routines where a data type is needed.
- Note that in a way these derived types have nothing to do with the derived datatypes in the programming language. They are defined only to make communication easier.

- The simplest datatype constructor is the routine `mpi_type_contiguous`:

```
call mpi_type_contiguous(count,oldtype,newtype,ierr)
```

  - Routine creates a new type called `newtype` by concatenating `count` copies of `oldtype`.

- Routine `mpi_type_vector` is a more general constructor that allows replication of a datatype into locations that consist of equally spaced blocks:

```
call mpi_type_vector(count,blocklength,stride,oldtype,newtype)
```

  - Here `count` is the number of blocks, `blocklength` the number of elements in each block, `stride` the number of elements between start of each block.

- The routine `mpi_type_indexed` allows replication of an old datatype into a sequence of blocks (each block is a concatenation of the old datatype), where each block can contain a different number of copies and have a different displacement. All block displacements are multiples of the old type extent.

```
call mpi_type_indexed(count,array_of_blocklengths,array_of_displacements,&
                      &oldtype,newtype,ierr)
integer count,array_of_blocklengths(:),array_of_displacements(:),&
                      &oldtype,newtype, ierr
```

- The most general type constructor is `mpi_type_struct`:

```
call mpi_type_struct(count,array_of_blocklengths,array_of_displacements,&
     &array_of_types,newtype,ierr)
integer count,array_of_blocklengths(:),array_of_displacements(:),&
     &array_of_types(:),newtype,ierr
```


Parallel computations: MPI

- In order to be able to use the new datatype it must be committed:

```
call mpi_type_commit(datatype,ierr)
```

- When the datatype is no more needed it may be freed:

```
call mpi_type_free(datatype,ierr)
```

- A simple example of a vector:

```
program mpi_vectortype
  use mpi
  implicit none
  integer :: rc,myid,nprocs,vector,status(mpi_status_size)
  integer,parameter :: count=10,blocklength=1,stride=2,n=20
  integer :: v(n),i

  call mpi_init(rc)
  call mpi_comm_rank(mpi_comm_world,myid,rc)
  call mpi_comm_size(mpi_comm_world,nprocs,rc)

  call mpi_type_vector(count,blocklength,stride,&
            &mpi_integer,vector,rc)
  call mpi_type_commit(vector,rc)
  if (myid==0) then
     v=(/ (i,i=1,n) /)
     call mpi_send(v,1,vector,1,1,mpi_comm_world,rc)
  else if (myid==1) then
     v=0
     call mpi_recv(v,1,vector,0,1,mpi_comm_world,status,rc)
     print *,v
  end if

  call mpi_finalize(rc)
end program mpi_vectortype
```

```
mpi> mpif90 mpi_vectortype.f90
mpi> mpirun -np 2 a.out
        1        0        3        0        5        0
        7        0        9        0       11        0
       13        0       15        0       17        0
       19        0
```

Parallel computations: MPI

• *Differences in using MPI in Fortran and C.*
- So far we have only shown the MPI usage in Fortran.
- Transforming the routine calls and argument list to C is straightforward.
- Moreover, all MIPCH man pages describe the C versions.
- One thing to remember is that *Fortran is case-insensitive but C is case-sensitive*.
- In order to find the right name of the routine see the directory **`/usr/local/openmpi/man/man3/`**.

- The most important difference between C and Fortran is that that all routines are functions in C and return their status. In Fortran user must give an extra parameter as the last one in the list where the routine status is assigned.

- In many cases where in Fortran there is an integer array in C the corresponding parameter is a `struct`.

- In Fortran most of the opaque MPI objects are of type `integer` but in C they may have their own type (well, which is in most cases defined as an `int`).

Parallel computations: threads, OpenMP and such

• *A central concept in modern operating systems is the process*[1].
- It can be defined as a single series of machine instructions managed by the operating system kernel.
- It has its own
  1) memory area
  2) process stack
  3) program counter
- New processes in Unix are can be created by e.g. the `fork()` system call:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

main (int argc, char **argv)
{
  pid_t pid;

  pid=fork();
  if (pid==0)
    printf("Hello I”m the child!\n");
  else if (pid>0)
    printf("I”m the parent and I forked process %d.\n",(int)pid);
  else
    printf("fork failed!\n");

}
```

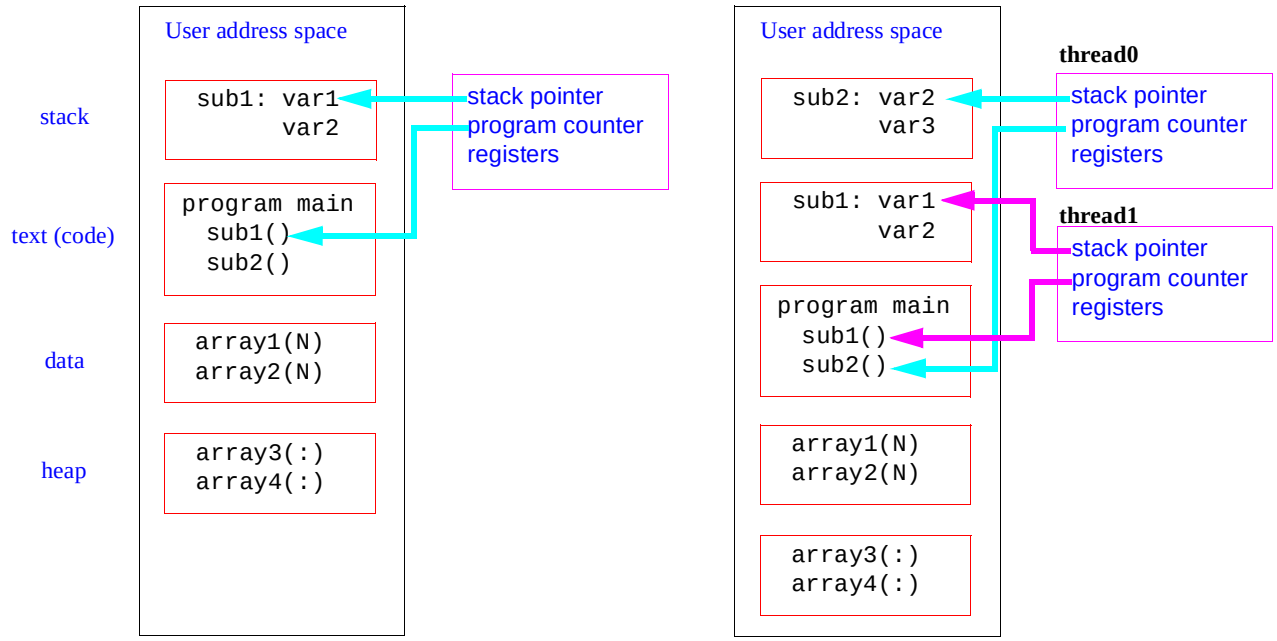
```
threads> gcc fork.c
threads> a.out
Hello I’m the child!
I’m the parent and I forked process 7960.
```

- Different processes can communicate with each other by various means: pipes, sockets, message queues, shared memory, semaphores.
  - However, all communication using these methods must be written explicitely by the programmer.
- As the whole memory space of the parent process must be copied to the child forking new processes is a relatively slow operation.

1. Partly adopted from http://www.llnl.gov/computing/tutorials/pthreads/

Parallel computations: threads, OpenMP and such

• **Threads** *could be thought as lightweight processes working in parallel within one process.*
- Threads duplicate only the essential resources it needs to be independently schedulable.
- Threads of the same process share most things as opposed to distinct processes:



Parallel computations: threads, OpenMP and such

- Different threads of the same process are **scheduled separately**.

- Because threads share the same memory space communication is straigthforward: shared memory.
- However, this also means trouble.
  - Imagine a situation where two threads update the same variable: `i=i+1`
    - The steps of the operation are
      1) Load `i`
      2) Increment `i`
      3) Store `i`
    - If thread `t1` loads `i` between steps 1 and 2 of thread `t0` then the result of thread `t0` is lost.
    - This can be prevented by using so called **mutex**es.
    - This shows that although explicit communication (as in MPI) is not needed in threads it is the responsibility of the programmer to ensure the proper co-operation of the threads.

Parallel computations: threads, OpenMP and such

- Comparing the speeds of process and thread creation can be demonstrated by the following program[1]:

```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <sys/time.h>
#define N 10000

void *thread_function(void *arg) {return NULL;}

int main(void) {
  pthread_t mythread;
  int i,j;
  clock_t t;
  struct timeval tv0,tv1;
  struct timezone tz;
  double sec;
  pid_t pid;

  t=clock(); gettimeofday(&tv0, &tz);
  for (i=1;i<N;i++) {
    j=pthread_create(&mythread,NULL,thread_function,NULL);
    j=pthread_join(mythread,NULL);
  }
  gettimeofday(&tv1, &tz); t=clock()-t;
  sec=(double)(tv1.tv_sec-tv0.tv_sec)+(double)(tv1.tv_usec-tv0.tv_usec)/1e6;
  printf("Thread: %g %g\n",(double)t/(double)CLOCKS_PER_SEC,sec);
  t=clock(); gettimeofday(&tv0, &tz);
  for (i=1;i<N;i++) {
    pid=fork();
    if (pid==0) exit(0);
  }
  gettimeofday(&tv1, &tz); t=clock()-t;
  sec=(double)(tv1.tv_sec-tv0.tv_sec)+(double)(tv1.tv_usec-tv0.tv_usec)/1e6;
  printf("Fork  : %g %g\n",(double)t/(double)CLOCKS_PER_SEC,sec);
}
```

```
threads> gcc fork_vs_thread.c -lpthread
threads> a.out
Thread: 1.08 1.18018
Fork  : 1.21 8.76349
```

1. Note that we are now using C because there are no thread libraries for Fortran.

Parallel computations: threads, OpenMP and such

- **POSIX threads** standard defines a set of subroutines (or an API[1]) to handle threads. These are implemented also in Linux.

- Posix threads API contains
  - **Thread management**: The first class of functions work directly on threads - creating, detaching, joining, etc. They include functions to set/query thread attributes (joinable, scheduling etc.)
  - **Mutexes**: The second class of functions deal with synchronization, called a "mutex", which is an abbreviation for "mutual exclusion". Mutex functions provide for creating, destroying, locking and unlocking mutexes. They are also supplemented by mutex attribute functions that set or modify attributes associated with mutexes.
  - **Condition variables**: The third class of functions address communications between threads that share a mutex. They are based upon programmer specified conditions. This class includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set/query condition variable attributes are also included

- Initially when a program is started it contains one thread.
- A new thread is created by routine `pthread_create`:

```
int pthread_create(pthread_t *thread, pthread_attr_t *attr, void
                   *(*start_routine)(void *), void *arg);
```

- A thread exits by calling `pthread_exit`:

```
void pthread_exit(void *retval);
```

- To wait for another thread to terminate one can use the routine `pthread_join`:

```
int pthread_join(pthread_t th, void **thread_return);
```

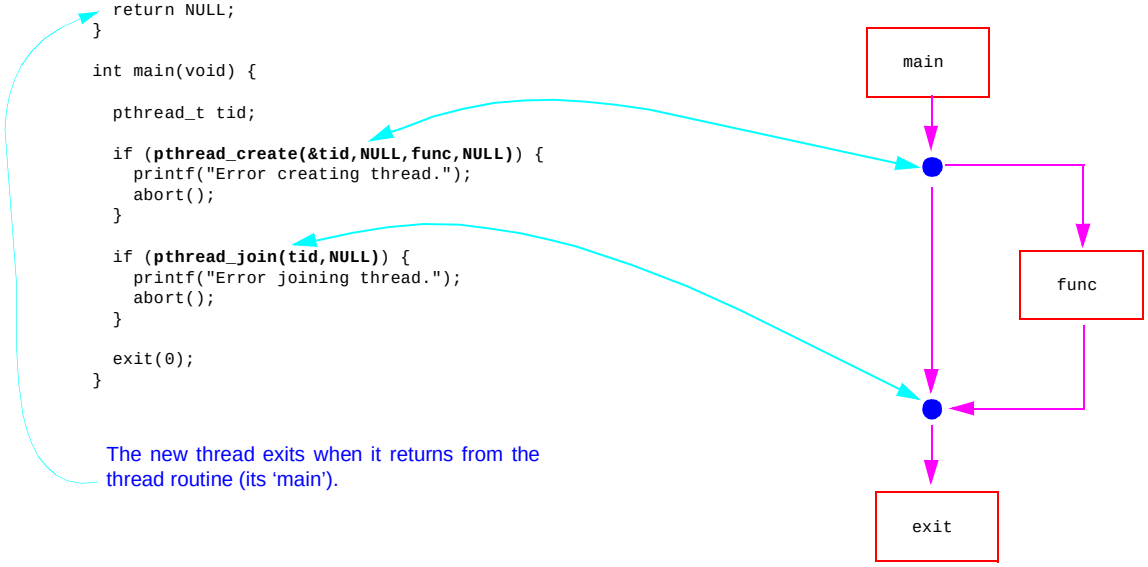
1. Application Program Interface

Parallel computations: threads, OpenMP and such

- A simple example[1]:

```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

void *func(void *arg) {
  sleep(2);
  printf("Hello world!\n");
  sleep(2);
  return NULL;
}

int main(void) {

  pthread_t tid;

  if (pthread_create(&tid,NULL,func,NULL)) {
    printf("Error creating thread.");
    abort();
  }

  if (pthread_join(tid,NULL)) {
    printf("Error joining thread.");
    abort();
  }

  exit(0);
}
```

```
threads> gcc thread1.c -lpthread
threads> a.out
Hello world!
```



1. All programs can be downloaded from http://www.acclab.helsinki.fi/~akuronen/suurteho/progs/threads/

Parallel computations: threads, OpenMP and such

- There are several ways in which a thread may be terminated:
  1. The thread returns from its starting routine.
  2. The thread makes a call to the `pthread_exit` subroutine.
  3. The thread is canceled by another thread via the `pthread_cancel` routine.
  4. The entire process is terminated due to a call to either the `exec` or `exit` subroutines.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#define NUM_THREADS  5

void *PrintHello(void *threadid)
{
  printf("Hello from thread %d (%d).\n", threadid,(int)getpid());
  pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
  pthread_t threads[NUM_THREADS];
  int rc, t;
  for(t=0;t<NUM_THREADS;t++){
    printf("Creating thread %d\n", t);
    if (pthread_create(&threads[t],NULL,PrintHello,(void *)t)) {
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
    }
  }
  pthread_exit(NULL);
}
```

```
llnl> gcc thread_exit.c -lpthread
llnl> a.out
Creating thread 0
Creating thread 1
Creating thread 2
Creating thread 3
Creating thread 4
Hello from thread 0 (18143).
Hello from thread 1 (18143).
Hello from thread 2 (18143).
Hello from thread 3 (18143).
Hello from thread 4 (18143).
```

Parallel computations: threads, OpenMP and such

- A thread can wait for another one to finish by calling the `pthread_join` function:

```
int pthread_join(pthread_t th, void **thread_return)
```

```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

int n=0;

void *thread_function(void *arg) {
  int i;
  n++;
  return NULL;
}

int main(void) {
  pthread_t mythread;
  int i;

  for (i=0;i<10;i++) {
    if (pthread_create( &mythread, NULL, thread_function, NULL)) {
      printf("error creating thread.");
      exit(-1);
    }
    if (pthread_join(mythread,NULL)) {
      printf("error joining thread.");
      exit(-1);
    }
  }
  printf("%d\n",n);
  exit(0);
}
```

```
threads> gcc thread_join.c -lpthread
threads> a.out
10
```


Parallel computations: threads, OpenMP and such

- Thread creation routine allows the programmer to pass one pointer to the new thread.
- The pointer must be cast as `(void * )`.

```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

#define MAXTH 5

struct argu {
  int i;
  char *c;
};

void *thread_function(void *arg) {
  struct argu *thargu;
  thargu=(struct argu *)arg;
    printf("%d: %s (%x)\n",thargu->i,
       thargu->c,(int)pthread_self());
  return NULL;
}

int main(void) {
  pthread_t mythread;
  int i;
  char *msg[MAXTH];
  struct argu thargu[MAXTH];

  msg[0]="Thread 0";
  msg[1]="Säie   1";
  msg[2]="Nauha  2";
  msg[3]="Nuora  3";
  msg[4]="Lanka  4";

  for (i=0;i<MAXTH;i++) {
  thargu[i].i=i;
  thargu[i].c=msg[i];
  if (pthread_create(&mythread,NULL,
         thread_function,(void *)&thargu[i])) {
    printf("error creating thread.");
    exit(-1);
  }
  if (pthread_join(mythread,NULL)) {
    printf("error joining thread.");
    exit(-1);
  }
 }
 exit(0);
}
```

```
threads> gcc thread_arg.c -lpthread
threads> a.out
0: Thread 0 (b7fe0bb0)
1: Säie   1 (b7fe0bb0)
2: Nauha  2 (b7fe0bb0)
3: Nuora  3 (b7fe0bb0)
4: Lanka  4 (b7fe0bb0)
```

Parallel computations: threads, OpenMP and such

- Note that we have an array of thread argument structs. Let's see what happens when we replace the array with one pointer and replace the `pthread_join` by `sleep`:

```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

#define MAXTH 5

struct argu {
  int i;
  char *c;
};

void *thread_function(void *arg) {
  struct argu *thargu;
  thargu=(struct argu *)arg;
  printf("%d: %s (%x)\n",
    thargu->i,thargu->c,(int)pthread_self());
  return NULL;
}

int main(void) {
  pthread_t mythread;
  int i;
  char *msg[MAXTH];
  struct argu thargu;

  msg[0]="Thread 0";
  msg[1]="Säie   1";
  msg[2]="Nauha  2";
  msg[3]="Nuora  3";
  msg[4]="Lanka  4";

  for (i=0;i<MAXTH;i++) {
    thargu.i=i;
    thargu.c=msg[i];
    if (pthread_create(&mythread,NULL,
        thread_function,(void *)&thargu)) {
      printf("error creating thread.");
      exit(-1);
    }
  }
  sleep(5);
  exit(0);
}
```

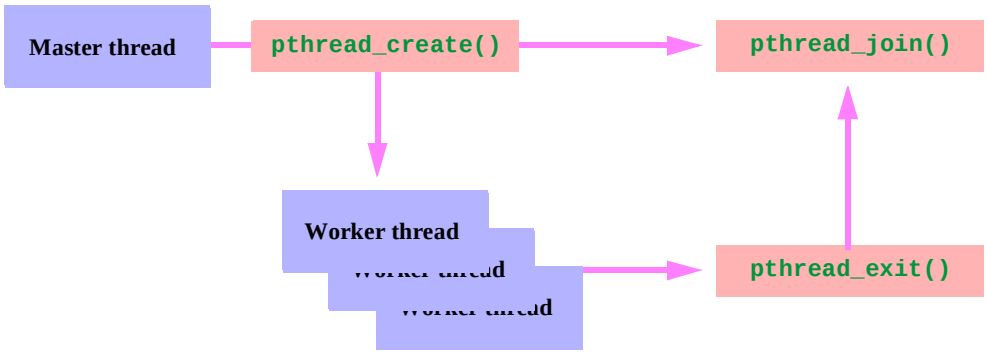
```
threads> gcc thread_arg_wrong.c -lpthread
threads> a.out
4: Lanka  4 (b7fe0bb0)
4: Lanka  4 (b75dfbb0)
4: Lanka  4 (b61ddbb0)
3: Nuora  3 (b6bdebb0)
4: Lanka  4 (b57dcbb0)
```

!

Before a new thread is started the main thread has already changed the fields of `thargu`.

Parallel computations: threads, OpenMP and such

- Joining is a way to synchronize thread execution.



- Function call `pthread_join(threadid)` blocks the calling thread until the thread `threadid` terminates.
  - As mentioned a thread is terminated if it returns from its initial routine or calls `pthread_exit`.

  - Attributes of a thread can be changed so that it is not joinable by sett the appropriate argument in `pthread_create`.
  - A joinable thread can be detached (i.e. made unjoinable) by the routine `pthread_detach`.

Parallel computations: threads, OpenMP and such

- A more elaborate example of joining[1]:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS  3

void *BusyWork(void *null) {
  int i;
  double result=0.0;
  for (i=0; i<1000000; i++) {
    result = result + (double)random();
  }
  printf("Thread result = %e\n",result);
  pthread_exit((void *) 0);
}

int main(int argc, char *argv[]) {
  pthread_t thread[NUM_THREADS];
  pthread_attr_t attr;
  int rc, t, status;

  /* Initialize and set thread detached attribute */
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

  for(t=0;t<NUM_THREADS;t++) {
    printf("Creating thread %d\n", t);
    if (rc=rpthread_create(&thread[t],&attr,BusyWork,NULL)) {
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
    }
  }

  /* Free attribute and wait for the other threads */
  pthread_attr_destroy(&attr);
  for(t=0;t<NUM_THREADS;t++) {
    if (rc=pthread_join(thread[t], (void **)&status)) {
      printf("ERROR return code from pthread_join() is %d\n", rc);
      exit(-1);
    }
    printf("Completed join with thread %d status= %d\n",t, status);
  }

  pthread_exit(NULL);
}
```

```
threads> gcc thread_join1.c -lpthread
threads> a.out
Creating thread 0
Creating thread 1
Creating thread 2
Thread result = 1.072782e+15
Thread result = 1.073960e+15
Completed join with thread 0 status= 0
Thread result = 1.074233e+15
Completed join with thread 1 status= 0
Completed join with thread 2 status= 0
```

1. http://www.llnl.gov/computing/tutorials/pthreads/samples/join1.c

Parallel computations: threads, OpenMP and such

- Stacksizes of threads can be manipulated by the routines

```
pthread_attr_getstacksize(attr,stacksize)
pthread_attr_setstacksize(attr,stacksize)
pthread_attr_getstackaddr(attr,stackaddr)
pthread_attr_setstackaddr(attr,stackaddr)
```

- For example:

```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

pthread_attr_t attr;

void *func(void *arg) {
  size_t stack;
  pthread_attr_getstacksize(&attr,&stack);
  printf("Stacksize in thread: %d.\n",(int)stack);
  return NULL;
}

int main(void) {
  pthread_t tid;
  size_t stack;

  pthread_attr_init(&attr);
  pthread_attr_getstacksize(&attr,&stack);
  printf("Default stacksize: %d.\n",(int)stack);
  stack*=2;
  pthread_attr_setstacksize(&attr,stack);
  pthread_create(&tid,&attr,func,NULL);
  pthread_join(tid,NULL);

  exit(0);
}
```

```
threads> gcc thread_stack.c -lpthread
threads> a.out
Default stacksize: 10485760.
Stacksize in thread: 20971520.
```

Note how the thread attributes are changed:

```
pthread_attr_t attr;
pthread_attr_init(&attr);
```

Then you can set or query[1] thread properties by routines:

```
pthread_attr_setdetachstate()
pthread_attr_setscope()
pthread_attr_setguardsize()
pthread_attr_setstack()
pthread_attr_setinheritsched()
pthread_attr_setstackaddr()
pthread_attr_setschedparam()
pthread_attr_setstacksize()
pthread_attr_setschedpolicy()
```

and give the `attr` as a argument to `pthread_create`.

1. replace `set®get`

Parallel computations: threads, OpenMP and such

- Function `pthread_self` returns the thread identifier of the running thread:

```
pthread_t pthread_self(void);
```

- Function `pthread_equal` returns zero if the two thread identifiers equal:

```
int pthread_equal(pthread_t thread1, pthread_t thread2);
```

  - Note that although thread ids seem to be integers it is not wise to use the C comparison operator `==` to compare threads because the ids are opaque objects and are supposed to be accessed only by the pthread functions. (From `/usr/include/bits/pthreadtypes.h`:

```
/* Thread identifiers */
typedef unsigned long int pthread_t;
```


Parallel computations: threads, OpenMP and such

- Function `pthread_once` executes the `init_routine` exactly once in a process. The first call to this routine by any thread in the process executes the given `init_routine`, without parameters. Any subsequent call will have no effect.

```
pthread_once_t once_control = PTHREAD_ONCE_INIT;
int pthread_once(pthread_once_t *once_control, void (*init_routine) (void));
```

```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

pthread_once_t once_control = PTHREAD_ONCE_INIT;

void *init_routine(void) {
  printf("Initializing %d.\n",(int)once_control);
  return NULL;
}

void *thread_function(void *arg) {
  pthread_once(&once_control,init_routine);
  return NULL;
}
```

```
int main(void) {
  pthread_t mythread;
  int i;

  for (i=0;i<10;i++) {
    if (pthread_create( &mythread, NULL, thread_function, NULL)) {
      printf("error creating thread.");
      exit(-1);
    }
    if (pthread_join(mythread,NULL)) {
      printf("error joining thread.");
      exit(-1);
    }
  }
  exit(0);
}
```

```
threads> gcc thread_once.c -lpthread
thread_once.c: In function ‘thread_function’:
thread_once.c:13: warning: passing arg 2 of ‘pthread_once’ from incompatible pointer type
threads> a.out
Initializing 1.
```

Parallel computations: threads, OpenMP and such

- As we have seen the programmer has to be careful in order not to make a mess of the threads accessing common variables simultaneously.
- Note that all threads of the same process are scheduled separately. This means that they are not by default synchronized in any way. This is left for the programmer.

- One tool for synchronizing and protecting shared data when simultaneous writes occur is the use of **mutex**es (mutex ← mutual exclusion).

- A **mutex variable** acts like a "lock" protecting access to a shared data resource.
- Only one thread can lock (or own) a mutex variable at any given time.
- No other thread can own that mutex until the owning thread unlocks that mutex.
  - Remember the simple example of a variable `i` incremented simultaneously by many threads:
    1) Load `i`
    2) Increment `i`
    3) Store `i`

- A typical sequence in the use of a mutex is as follows:
  1. Create and initialize a mutex variable.
  2. Several threads attempt to lock the mutex.
  3. Only one succeeds and that thread owns the mutex.
  4. The owner thread performs some set of actions.
  5. The owner unlocks the mutex.
  6. Another thread acquires the mutex and repeats the process.
  7. Finally the mutex is destroyed.

  - When several threads compete for a mutex, the losers block at that call.
  - An unblocking call is available with "trylock" instead of the "lock" call.

Parallel computations: threads, OpenMP and such

- A mutex is of type `pthread_mutex_t`:

```
pthread_mutex_t mutex_variable;
```

- It is initialized by the function `pthread_mutex_init`:

```
int pthread_mutex_init(pthread_mutex_t *mutex_variable,
                       const pthread_mutexattr_t *mutexattr);
```

- If you are satisfied with the default attributes the call can be simplified as

```
pthread_mutex_init(&mutex_variable,NULL);
```

- The critical regions that only one thread should be executing at a time must be surrounded by the lock and unlock routine calls:

```
pthread_mutex_lock(&mutex_variable);
/* Only one thread executes this. */
...
pthread_mutex_unlock(&mutex_variable);
```

- When a mutex is no more needed it is destroyed by

```
pthread_mutex_destroy(&mutex_variable);
```

Parallel computations: threads, OpenMP and such

- A real example of a **race condition**:

```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

#define NTHREADS 10
#define JMAX 100
int n=0,nmax=10000;

void *thread_function(void *arg) {
  int i;
  for (i=0;i<nmax;i++) n++;
  return NULL;
}

int main(void) {
  pthread_t mythread[NTHREADS];
  int i,j;

  for (j=0;j<JMAX;j++) {
    n=0;
    for (i=0;i<NTHREADS;i++) {
      if (pthread_create(&mythread[i],NULL,
         thread_function,NULL)) {
  printf("Error creating thread.");
  exit(-1);
      }
    }
    for (i=0;i<NTHREADS;i++) {
      if (pthread_join(mythread[i],NULL)) {
  printf("Error joining thread.");
  exit(-1);
      }
    }
    printf("%d ",n);
  }
  exit(0);
}
```

```
threads> gcc thread_nomutex.c -lpthread
threads> a.out
100000 100000 84326 42374 100000 62151 50738 100000 100000 100000
100000 91099 100000 100000 90267 75141 69221 85619 95413 100000
86413 71398 100000 100000 69553 85787 69494 100000 100000 100000
100000 100000 100000 100000 100000 71932 87371 100000 100000 88332
100000 100000 83665 100000 100000 67910 100000 100000 100000 75437
100000 100000 76236 100000 100000 100000 66852 54856 54190 100000
59509 85602 75169 98049 100000 63484 88433 100000 100000 74665
100000 100000 85254 86860 92991 100000 100000 96001 83891 100000
100000 100000 87201 100000 81207 91372 70503 97963 100000 100000
71715 100000 100000 100000 100000 58840 100000 92083 100000 59735
```

Many threads trying to do this simultaneously.

Parallel computations: threads, OpenMP and such

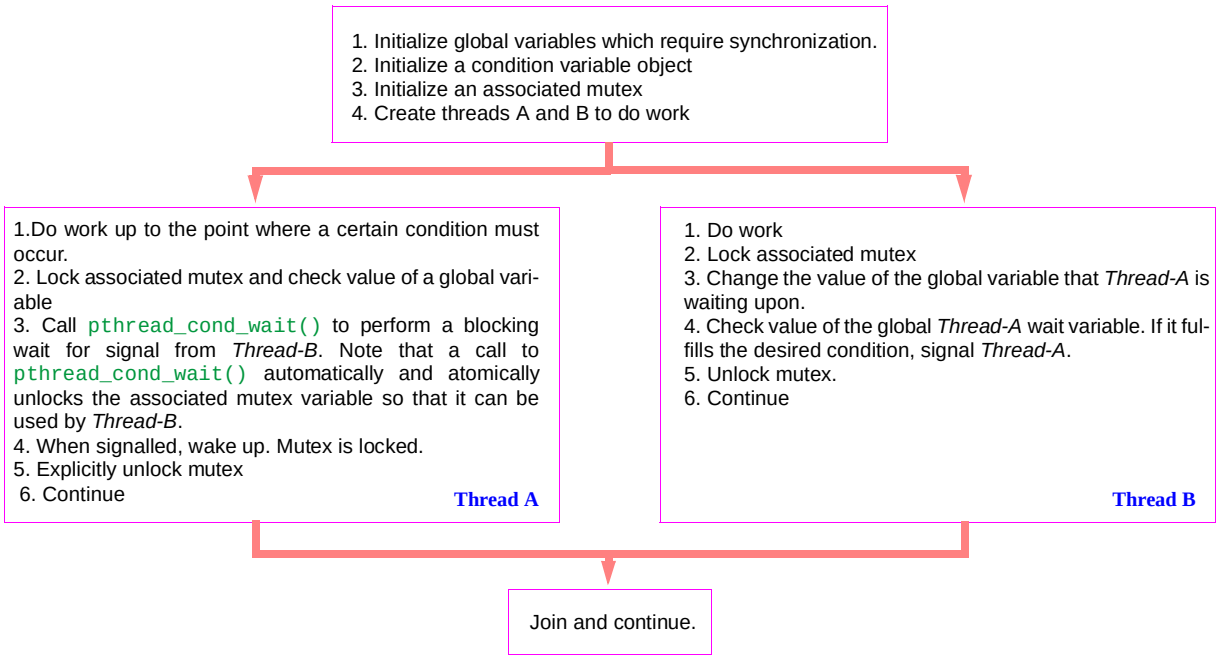
- Use of a mutex variable ensures that only one thread updates the variable at a time.

```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#define NTHREADS 10
#define JMAX 100
int n=0,nmax=10000;
pthread_mutex_t xetum;

void *thread_function(void *arg) {
  int i;
  for (i=0;i<nmax;i++) {
    pthread_mutex_lock(&xetum);
    n++;
    pthread_mutex_unlock(&xetum);
  }
  return NULL;
}

int main(void) {
  pthread_t mythread[NTHREADS];
  int i,j;
  pthread_mutex_init(&xetum,NULL);
  for (j=0;j<JMAX;j++) {
    n=0;
    for (i=0;i<NTHREADS;i++) {
      if (pthread_create(&mythread[i],NULL,thread_function,NULL)) {
        printf("Error creating thread.");;exit(-1);
      }
    }
    for (i=0;i<NTHREADS;i++) {
      if (pthread_join(mythread[i],NULL)) {
        printf("Error joining thread."); exit(-1);
      }
    }
    printf("%d ",n);
  }
  pthread_mutex_destroy(&xetum);
  exit(0);
}
```

```
threads> gcc thread_mutex.c -lpthread
threads> a.out
100000 100000 100000 100000 100000 100000 100000 100000 100000 100000
100000 100000 100000 100000 100000 100000 100000 100000 100000 100000
100000 100000 100000 100000 100000 100000 100000 100000 100000 100000
100000 100000 100000 100000 100000 100000 100000 100000 100000 100000
100000 100000 100000 100000 100000 100000 100000 100000 100000 100000
100000 100000 100000 100000 100000 100000 100000 100000 100000 100000
100000 100000 100000 100000 100000 100000 100000 100000 100000 100000
100000 100000 100000 100000 100000 100000 100000 100000 100000 100000
100000 100000 100000 100000 100000 100000 100000 100000 100000 100000
100000 100000 100000 100000 100000 100000 100000 100000 100000 100000
```

Parallel computations: threads, OpenMP and such

- A more versatile synchronization is achived by using **condition variables**.
- Condition variables allow threads to synchronize based upon the actual value of data.
- Without condition variables, the programmer would need to have threads continually polling to check if the condition is met.
- A condition variable is always used in conjunction with a mutex lock.



Parallel computations: threads, OpenMP and such

- A simple example shows best how condition variables are used.

Main

```
int main(void) {
  pthread_t incid,decid;
  seed=getseed();
  pthread_mutex_init(&limit_mutex,NULL);
  pthread_cond_init(&limit_cond,NULL);
  if (pthread_create(&incid,NULL,inc,NULL)) {printf("Error creating inc.");exit(-1);}
  if (pthread_create(&decid,NULL,dec,NULL)) {printf("Error creating dec.");exit(-1);}
  if (pthread_join(incid,NULL)) {printf("Error joining inc.");exit(-1);}
  if (pthread_join(decid,NULL)) {printf("Error joining dec.");exit(-1);}
  printf("Finished: %g\n",x);
  exit(0);
}
```

Definitions

```
int nmax=100000000;
double x=0.0;
int seed;
double limit=10000000.0;
pthread_mutex_t
limit_mutex;
pthread_cond_t limit_cond;
```

Thread A

```
void *inc(void *arg) {
  int n;
  /* A small delay so that routine dec
     has time to begin waiting. */
  sleep(1);
  printf("Started inc.\n");
  pthread_mutex_lock(&limit_mutex);
  for (n=0;n<nmax;n++) {
    x+=lcg(&seed);
    if (x>=limit) {
      pthread_cond_signal(&limit_cond);
      printf("Inc sent the signal. (%d)\n",n);
      pthread_mutex_unlock(&limit_mutex);
      break;
    }
  }
  printf("Inc finished.\n");
  pthread_exit(NULL);
}
```

Thread B

```
void *dec(void *arg) {
  int n;
  printf("Started dec.\n");
  pthread_mutex_lock(&limit_mutex);
  while (x<=limit) {
    printf("Dec waiting for limit_cond.\n");
    pthread_cond_wait(&limit_cond,&limit_mutex);
    printf("Dec received condition signal.\n");
  }
  pthread_mutex_unlock(&limit_mutex);
  printf("Dec doing the rest.\n");
  for (n=0;n<nmax;n++) {
    x-=lcg(&seed);
  }
  printf("Dec finished.\n");
  pthread_exit(NULL);
}
```

Parallel computations: threads, OpenMP and such

- And the output of the program:

```
threads> gcc thread_condvar.c -lpthread
threads> a.out
Started dec.
Dec waiting for limit_cond.
Started inc.
Inc sent the signal. (19998079)
Dec received condition signal.
Dec doing the rest.
Inc finished.
Dec finished.
Finished: -3.99986e+07
```


Parallel computations: threads, OpenMP and such

• *Doing real parallel computations on SMP machines can be accomplished by using threads.*
• *Posix threads can be found on most Unix and Linux machines.*

• *However, rather low level programming is needed because the synchronization and data sharing must be explicitely done by the programmer.*

• *Also, there is no standard API for Posix threads for Fortran.*
  - Of course, you can write C wrappers for pthread routines.
  - However, it is by no means certain that Fortran libraries are thread-safe.
  - Well, if a Fortran compiler supports OpenMP then you can probably assume that the libraries are thread-safe because OpenMP is implemented using threads.

• *Good sources of information are not hard to find by Google:*
  - http://www.llnl.gov/computing/tutorials/pthreads/ contains a good tutorial to pthreads. There are also examples of combining MPI and threads.
  - A short introduction to pthreads can also be found on IBM Developerworks: http://www-128.ibm.com/developerworks/linux/library/l-posix1.html
  - http://math.arizona.edu/~swig/documentation/pthreads/ contains example programs doing threaded matrix multiplication and LU decomposition.

Parallel computations: threads, OpenMP and such

• *A higher level API (Application Program Interface) to threads is* **OpenMP**[1].
- It is a **standard** describing an API for shared-memory parallel programming in both C/C++ and Fortran.
- Parallelization of program execution is done by compiler directives. Directives look like comments but are interpreted by an OpenMP aware compiler.
- An OpenMP program starts its execution as a single thread until a parallel region is encountered.

C

```
#include <omp.h>

main ()  {
int var1, var2, var3;

/* Serial code */
       . . .

/* Beginning of parallel section. */
#pragma omp parallel private(var1,var2) shared(var3)
  {
    /* Parallel section executed by all threads. */

       . . .

    /* All threads join master thread. */
  }

/* Resume serial code. */

       . . .
}
```

Fortran

```
program hello
  integer :: var1, var2, var3

  ! Serial code
      . . .


  ! Beginning of parallel section.
  !$omp parallel private(var1,var2) shared(var3)

  ! Parallel section executed by all threads.
      . . .
  ! All threads join master thread.

  !$omp end parallel

  ! Resume serial code
      . . .

end program hello
```

- This means that the OpenMP directives added to an existing program don't change its behavior when compiled using a OpenMP-ignorant compiler.

1. http://www.openmp.org/

Parallel computations: threads, OpenMP and such

- In C OpenMP directives are given as a preprocessor command in the form
  `#pragma omp directive-name [clause,...]`
- Each directive applies to at most one succeeding statement, which must be a structured block.
- In Fortran directives have the form
  `!$OMP directive-name [clause,...]`
- Many Fortran directives come in pairs and have the form

```
!$OMP  directive
    [structured block of code]
!$OMP end  directive
```

Continued lines in Fortran:

```
!$OMP parallel &
!$OMP private(x)
```

Or:

```
!$OMP parallel &
!$OMP& private(x)
```

- Note that in C the OpenMP directives are case-sensitive but in Fortran they are case-insensitive.
- OpenMP specification also allows conditional compilation: if the compiler is OpenMP compliant the following Fortran and C constructs are compiled:

```
      . . .
!$ i=omp_get_thread_num()
!$ print *,'I am thread ',i
      . . .
```

```
#ifdef _OPENMP
i=omp_get_thread_num();
printf("I am thread %d\n",i);
#endif
```

Example

```
openmp> ifort -openmp omp_hello.f90
openmp> a.out
 Hello world from thread =            0  out of          4           0
 Hello world from thread =            1  out of          4           1
 Hello world from thread =            2  out of          4           2
 Hello world from thread =            3  out of          4           3
openmp> ifort omp_hello.f90
openmp> a.out
 Hello world from thread =      9999999  out of    8888888     9999999
```

```
program omp_hello
  use omp_lib
  implicit none
  integer :: tid,threads,i

  tid=9999999
  threads=8888888

  !$ call omp_set_num_threads(4)

  !$omp parallel private(tid),shared(i)

  !$ tid = omp_get_thread_num()
  !$ threads=omp_get_num_threads()
  i=tid
  print *, 'Hello world from thread = ',&
   &tid,' out of ',threads,i

  !$omp end parallel

end program omp_hello
```

Parallel computations: threads, OpenMP and such

- In order to enable OpenMP the compiler must be given the proper option:

In Intel Linux (C and Fortran):	option `-openmp`
Pathscale and Portland Group compilers[1]:	option `-mp`
IBMSC at CSC:	`xlf90_r -qsmp=omp omp_hello.f` `xlc_r -qsmp=omp`
Tru64 Unix	option `-omp`

- In addition to compiler directives OpenMP includes some library routines and evnironmental variable to control the program behavior.

- The way to get the OpenMP library routines visible to the program source also varies:

In C an `.h` file needs to be included:	`#include <omp.h>`
Intel Fortran for Linux:	`use omp_lib`
Pathscale[2]:	`use omp_lib`
IBM SC:	`use omp_lib`
Tru64 Unix:	`include 'frompdef.f'`

- The OpenMP specifications for C and Fortran can be downloaded from
  http://www.openmp.org/ ½→ Specifications

- Below we go through the most common OpenMP directives and library routines[3].

1. Installed on ametisti.
2. For the Portland Group compiler I didn't find a proper `.h` file or module. By explicitly defining all used OpenMP functions as external compilation works.
3. http://www.llnl.gov/computing/tutorials/openMP/, http://www.nersc.gov/nusers/help/tutorials/openmp/

Parallel computations: threads, OpenMP and such

- The fundamental parallel construct is the `parallel` directive:

```
program omp_hello
  use omp_lib
  implicit none
  integer :: tid

  call omp_set_num_threads(4)

  !$omp parallel private(tid)
  tid = omp_get_thread_num()
  print *, 'Hello world from thread = ', tid
  !$omp end parallel

end program omp_hello
```

```
#include <stdio.h>
#include <omp.h>
int main ()  {
  int nthreads, tid;

  omp_set_num_threads(4);

#pragma omp parallel private(tid)
  {
    tid = omp_get_thread_num();
    printf("Hello World from thread = %d\n", tid);
  }

  exit(0);
}
```

```
openmp> ifort -openmp omp_hello.f90
ifort: Command line warning: openmp requires C style preprocessing; using fpp to preprocess
omp_hello.f90(6) : (col. 6) remark: OpenMP DEFINED REGION WAS PARALLELIZED.
openmp> a.out
 Hello world from thread =           0
 Hello world from thread =           1
 Hello world from thread =           2
 Hello world from thread =           3
```

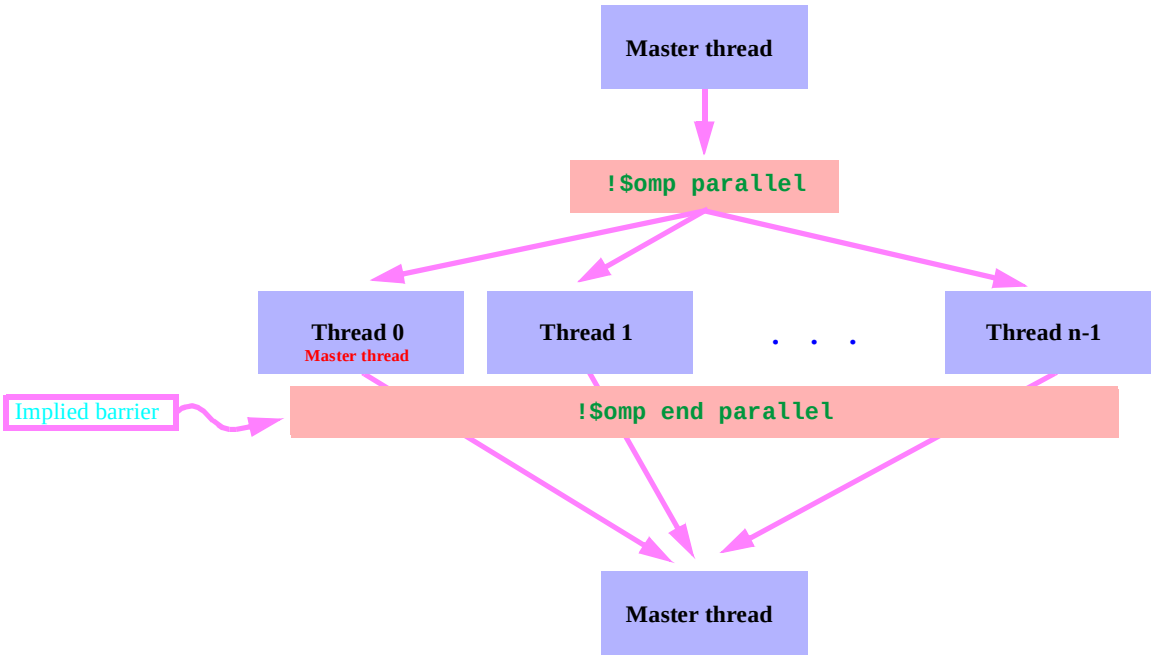
```
~/stltk> pgf90 -mp omp_hello.f90
~/stltk> a.out
Warning: omp_set_num_threads (4) greater than available cpus (2)
 Hello world from thread =           0
 Hello world from thread =           1
 Hello world from thread =           2
 Hello world from thread =           3
```

Also on ametisti.

```
~/stltk> pathf90 -mp omp_hello.f90
~/stltk> a.out
 Hello world from tHello world from thread =   2
 Hello world from thread =  0
 Hello world from tHello world from thread =   2
 Hello world from thread =  3
```

Parallel computations: threads, OpenMP and such

- Statements between the Fortran directives or within the statement block in C are executed in parallel in separate threads.



Parallel computations: threads, OpenMP and such

- A couple of definitions:
  - **Nested**: A parallel region is said to be nested if it appears within the dynamic extent of a `parallel`.
  - **Lexical extend**: Statements lexically contained within a structured block.
  - **Dynamical extend**: In addition to the lexical extend includes the routines called from within the construct.

```
#include <omp.h>

void sub(int tid);

void main ()  {
  int tid;

  omp_set_num_threads(4);

#pragma omp parallel private(tid)
  {
    tid = omp_get_thread_num();
    printf("This is main: %d.\n",tid);
    sub(tid);
  }

  exit(0);
}
```

Lexical extend

Dynamical extend

```
void sub(int tid) {
  printf("This is sub:  %d.\n",tid);
}
```

Parallel computations: threads, OpenMP and such

- The default number of threads created in the parallel region seems to be implementation dependent.
  - In many cases it is the number of CPUs in the machine.

- The block of code between the parallel directives (Fortran) or within the braces (C) should be a structured block:
  1) It has one entry point at the top.
  2) One point of exit at the bottom.

  - Number of threads can be set explicitly by the library routine `omp_set_num_threads`:
    Fortran: `call omp_set_num_threads(4)`
    C: `omp_set_num_threads(4);`
  - Number of threads can also be set by using the environmental variable `OMP_NUM_THREADS`.
  - The precedence of different ways to set the number of threads is
    1) `num_threads` clause (see below)
    2) `omp_set_num_threads` function call
    3) `OMP_NUM_THREADS` environmental variable
  - The total number of threads (in the current position of the code) can be obtained by:
    Fortran: `nthreads=omp_get_num_threads()`
    C: `nthreads=omp_get_num_threads();`
  - The number of the current thread (as rank in MPI) is queried by
    Fortran: `tid=omp_get_thread_num()`
    C: `tid=omp_get_thread_num();`

Parallel computations: threads, OpenMP and such

- The directive has a couple of optional **clauses** that specify details of the parallel section.
  - In C the most common ones are:

```
#pragma omp parallel [clause ...]
clause = if (scalar_expression)
```

If `scalar_expression` evaluates to true only then is the section executed in many threads. Otherwise the main thread executes it serially.

```
         private (list)
```

List of variables that are private to each thread.

```
         shared (list)
```

List of variables that are common to each thread.

```
         default (private | shared | none)
```

Default status of variables in the parallel block.

```
         firstprivate (list)
```

Initialize each private copy to the value of the variable in the main thread.

```
         reduction (operator: list)
```

Reduction operation of variables using `operator`.

```
         num_threads(scalar_integer_expression)
```

Start this many threads.

  - And in Fortran:

```
!$OMP PARALLEL [clause ...]
clause = IF (scalar_logical_expression)
         PRIVATE (list)
         SHARED (list)
         DEFAULT (PRIVATE | SHARED | NONE)
         FIRSTPRIVATE (list)
         REDUCTION (operator: list)
         NUM_THREADS(scalar_integer_expression)
```

Parallel computations: threads, OpenMP and such

- The clause `private(var1,var2,...)` creates a new objects for each thread for the variables in the list.
  - This also applies to the master thread:

```
program omp_private
  use omp_lib
  implicit none
  integer :: tid,threads

  tid=9999999
  threads=8888888
  call omp_set_num_threads(4)

  !$omp parallel private(tid,threads)
  tid = omp_get_thread_num()
  threads=omp_get_num_threads()
  print *, 'Thread ',tid,' out of ',threads
  !$omp end parallel

  print *,'After parallel:',tid,threads
end program omp_private
```

```
openmp> ifort -fpp -openmp omp_private.f90
openmp> a.out
 Thread            0  out of            4
 Thread            1  out of            4
 Thread            2  out of            4
 Thread            3  out of            4
 After parallel:     9999999     8888888
```

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

int main ()  {
  int tid,threads;

  tid=9999999;
  threads=8888888;

  omp_set_num_threads(4);

#pragma omp parallel private(tid,threads)
  {
    tid = omp_get_thread_num();
    threads = omp_get_num_threads();
    printf("Thread %d out of %d\n",tid,threads);
  }

  printf("After parallel: %d %d\n",tid,threads);
  exit(0);
}
```


Parallel computations: threads, OpenMP and such

- The `reduction(operator:var1,var2,...)` clause performs the reduction of variables `var1`,`var2`,... using `operator`. Instead of operator an intrinsic function `max`, `min`, `iand`, `ior`, `ieor` can be used in Fortran.

```
program omp_reduction
  use omp_lib
  implicit none
  integer :: s,p,m,nt

  call omp_set_num_threads(4)
  s=0;p=1;m=-huge(m)

  !$omp parallel private(nt) &
  !$omp reduction(+:s) reduction(*:p) reduction(max:m)
  nt=omp_get_thread_num()
  s=nt+1
  p=nt+1
  m=nt+1
  !$omp end parallel

  print *,s,p,m
end program omp_reduction
```

```
openmp> ifort -fpp -openmp omp_reduction.f90
openmp> a.out
          10          24           4
```

Note the initialization.

The IBM compiler seems to allow only reduction operation inside the parallel region for the reduction variables. So the statements should be changed to:

```
s=s+nt+1
p=p*(nt+1)
m=max(m,nt+1)
```

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

int main ()  {
  int s,p,nt;

  omp_set_num_threads(4);
  s=0;
  p=1;

#pragma omp parallel private(nt) reduction(+:s) reduction(*:p)
  {
    nt=omp_get_thread_num();
    s=p=nt+1;
  }
  printf("%d %d\n",s,p);
  exit(0);
}
```

Parallel computations: threads, OpenMP and such

- So called work-sharing directives divide the execution of the enclosed code region among those threads that encounter it. They do not create threads themselves, so they must be inside a parallel region in order to execute in parallel.
  - OpenMP allows to use shortcuts so that a parallel work-sharing directive can be specified on one line:

```
$!omp parallel
!$omp work-sharing-directive
   . . .
!$omp end work-sharing-directive
!$omp end parallel
```

→

```
!$omp parallel work-sharing-directive
   . . .
!$omp end parallel work-sharing-directive
```

  - Work-sharing directives are

```
do
sections
single
workshare
```

  - Parallel version of these are obtained by prepending the word `parallel` before them:

```
!$omp parallel do
!$omp parallel sections
!$omp parallel single
!$omp parallel workshare
```

  - C doesn't have `do` but `for` loops:

```
#pragma omp parallel for
```

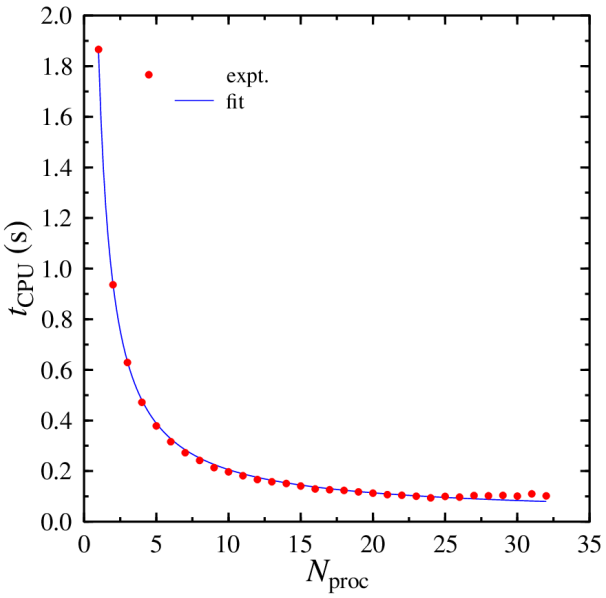

Parallel computations: threads, OpenMP and such

- A trivial example of a parallel `do`:

```
program omp_pardo
  use omp_lib
  use sizes
  implicit none
  integer :: tid,threads,i,n,nt
  real(rk) :: s,t0,t1
  character(len=80) :: argu

  call getarg(1,argu); read(argu,*) nt
  call getarg(2,argu); read(argu,*) n

  call omp_set_num_threads(nt)
  tid = omp_get_thread_num()
  threads=omp_get_num_threads()

  s=0.0
  t0=omp_get_wtime()
  !$omp parallel do reduction(+:s)
  do i=1,n
     s=s+log(real(i,rk))
  end do
  !$omp end parallel do
  t1=omp_get_wtime()
  print *,nt,t1-t0,s

end program omp_pardo
```

Just like `MPI_Wtime()`

Run at ibmsc.csc.fi



Runscript

```
#! /bin/csh
set n=10000000
foreach p (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32)
    ./omp_pardo $p $n
end
```

Parallel computations: threads, OpenMP and such

- What happens if we leave out the `reduction` clause:

tof.acclab.helsinki.fi with `reduction(+:s)`

```
openmp> ifort -fpp -openmp omp_pardo.f90
openmp> a.out 1 1000 ; a.out 2 1000 ; a.out 4 1000 ; a.out 8 1000 ;
           1  3.139972686767578E-004   5912.12817848817
           2  1.631975173950195E-003   5912.12817848817
           4  4.837894439697266E-002   5912.12817848817
           8  0.104144096374512        5912.12817848816
```

tof.acclab.helsinki.fi without `reduction(+:s)`

```
openmp> ifort -fpp -openmp omp_pardo.f90
openmp> a.out 1 1000 ; a.out 2 1000 ; a.out 4 1000 ; a.out 8 1000 ;
           1  2.939701080322266E-004   5912.12817848817
           2  0.101110935211182        5912.12817848817
           4  4.389095306396484E-002   5912.12817848817
           8  9.845399856567383E-002   5912.12817848817
```

ibmsc.csc.fi with `reduction(+:s)`

```
~/openmp> xlf90_r -qsmp=omp omp_pardo.f
~/openmp> a.out 1 1000 ; a.out 2 1000 ; a.out 4 1000 ; a.out 8 1000 ;
 1 0.2498626708984375OE-03 5912.12817848817122
 2 0.4539489746093750OE-03 5912.12817848816667
 4 0.7965564727783203l2E-03 5912.12817848816576
 8 0.1912021636962890l2E-01 5912.12817848816485
```

ibmsc.csc.fi without `reduction(+:s)`

```
~/openmp> xlf90_r -qsmp=omp omp_pardo.f
~/openmp> a.out 1 1000 ; a.out 2 1000 ; a.out 4 1000 ; a.out 8 1000 ;
 1 0.2291202545166015625E-03 5912.12817848817122
 2 0.4298686981201171875E-03 3300.79772002800746
 4 0.6847381591796875000E-03 1607.95353268518625
 8 0.2038240432739257812E-01 4304.17464580297474
```

```
program omp_pardo
  use omp_lib
  use sizes
  implicit none
  integer :: tid,threads,i,n,nt
  real(rk) :: s,t0,t1
  character(len=80) :: argu

  call getarg(1,argu); read(argu,*) nt
  call getarg(2,argu); read(argu,*) n

  call omp_set_num_threads(nt)
  tid = omp_get_thread_num()
  threads=omp_get_num_threads()

  s=0.0
  t0=omp_get_wtime()
  !$omp parallel do reduction(+:s)
  do i=1,n
     s=s+log(real(i,rk))
  end do
  !$omp end parallel do
  t1=omp_get_wtime()
  print *,nt,t1-t0,s

end program omp_pardo
```


Parallel computations: threads, OpenMP and such

- The `do` directive can have the following clauses:

`private(list)`, `firstprivate(list)`, `reduction(operator:list)`
: As in `parallel` directive.

`lastprivate(list)`
: Update the variables in list to the values of the sequentially last thread.

Code from previous page:

```
!$omp parallel do lastprivate(i)
do i=1,n
   s=s+log(real(i,rk))
end do
!$omp end parallel do
t1=omp_get_wtime()
print *,nt,t1-t0,s,i
```

```
openmp> a.out 4 1000
4  1.02210E-02 5912.12   1001
```

`schedule(type,[chunk])`
: Specifies how to divide the iterations among the threads.
`type` can be one of `static`, `dynamic`, `guided`, `runtime`.
For example `schedule(static,100)` divides the loop into pieces of 100 iteration and statically (in a round-robin fashion) allocates threads to these pieces.
Using schedule(static,100) threads are dynamically allocated. When a thread has finished one part of the iteration its starts another one. For more details see the OpenMP specification[1].

`ordered`
: Causes the iteration to be done in the sequential order.

1. http://www.openmp.org/

Parallel computations: threads, OpenMP and such

- The `sections` directive specifies that code in the the enclosed `section` blocks are to be divided among the threads in the team. Each section is executed once.

```
program omp_sections
  use omp_lib
  implicit none
  integer :: tid,nt
  character(len=80) :: argu

  call getarg(1,argu); read(argu,*) nt
  call omp_set_num_threads(nt)

  !$omp parallel private(tid)
  !$omp sections

  !$omp section
  tid=omp_get_thread_num()
  print *,'I am thread    ',tid
  !$omp section
  tid=omp_get_thread_num()
  print *,'Minä olen säie ',tid
  !$omp section
  tid=omp_get_thread_num()
  print *,'Jag är tråd    ',tid

  !$omp end sections
  !$omp end parallel

end program omp_sections
```

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

int main (int argc, char **argv)  {
  int tid,nt;

  nt=atoi(*++argv);
  omp_set_num_threads(nt);

#pragma omp parallel sections private(tid)
  {
    #pragma omp section
    {
      tid=omp_get_thread_num();
      printf("I am thread    %d\n",tid);
    }
    #pragma omp section
    {
      tid=omp_get_thread_num();
      printf("Minä olen säie %d\n",tid);
    }
    #pragma omp section
    {
      tid=omp_get_thread_num();
      printf("Jag är tråd    %d\n",tid);
    }
  }

  exit(0);
}
```

```
~/openmp>     xlc_r     -
qsmp=omp omp_sections.c

~/openmp> a.out 1
I am thread    0
Minä olen säie 0
Jag är tråd    0

~/openmp> a.out 2
I am thread    0
Jag är tråd    0
Minä olen säie 1

~/openmp> a.out 3
I am thread    2
Minä olen säie 0
Jag är tråd    1

~/openmp> a.out 4
I am thread    3
Minä olen säie 1
Jag är tråd    0

~/openmp> a.out 4
I am thread    1
Jag är tråd    2
Minä olen säie 3
```


Parallel computations: threads, OpenMP and such

- Clause `lastprivate` in `sections` directive:

```
program omp_sections_lastprivate
  use omp_lib
  implicit none
  integer :: i

  call omp_set_num_threads(4)

  !$omp parallel
  !$omp sections lastprivate(i)

  !$omp section
  i=omp_get_thread_num()
  !$omp section
  i=omp_get_thread_num()*10
  !$omp section
  i=omp_get_thread_num()*100
  !$omp section
  i=omp_get_thread_num()*1000

  !$omp end sections
  !$omp end parallel

  print *,i
end program omp_sections_lastprivate
```

```
openmp> ifort -fpp -openmp omp_sections_lastprivate.f90
omp_sections_lastprivate.f90(10) : (col. 6) remark: OpenMP DEFINED SECTION WAS PARALLELIZED.
omp_sections_lastprivate.f90(9) : (col. 6) remark: OpenMP DEFINED REGION WAS PARALLELIZED.
openmp> a.out
        3000
```

`lastprivate` takes the *lexically last* value of the variable.

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

int main ()  {
  int i;

  omp_set_num_threads(4);

#pragma omp parallel sections lastprivate(i)
  {
#pragma omp section
    {i=omp_get_thread_num();}
#pragma omp section
    {i=omp_get_thread_num()*10;}
#pragma omp section
    {i=omp_get_thread_num()*100;}
#pragma omp section
    {i=omp_get_thread_num()*1000;}
  }
  printf("%d\n",i);
  exit(0);
}
```

Parallel computations: threads, OpenMP and such

- The `single` directive specifies that the enclosed code is to be executed by only one thread.
  - Other threads wait at the `end single` directive unless `nowait` is specified.
  - It is illegal to branch out of a `single` block.

```
!$omp single [clause ...]

   . . .

!$omp end single [end_single_modifier]


clause =
       private(list)
       firstprivate(list)

end_single_modifier =
       copyprivate(list)
       nowait
```

```
#pragma omp single [clause ...]
{
   . . .
}

clause =
       private(list)
       firstprivate(list)
       copyprivate(list)
       nowait
```

- Clause `copyprivate(list)` copies the values of the variables in `list` to corresponding variables in other threads.

Parallel computations: threads, OpenMP and such

- Example:

```
program omp_single
  use omp_lib
  implicit none
  integer :: n,t

  call omp_set_num_threads(4)
  n=9999

  !$omp parallel private(t) firstprivate(n)

  t=omp_get_thread_num()
  print *,'Before:',t,n

  !$omp barrier
  !$omp single
  write(6,'(a)',advance='no') 'Give n: '
  read(5,*) n
  !$omp end single copyprivate(n)

  print *,'After:',t,n

  !$omp end parallel

end program omp_single
```

```
openmp> ifort -fpp -openmp omp_single.f90
omp_single.f90(9) : (col. 6) remark: OpenMP DEFINED REGION
WAS PARALLELIZED.
openmp> a.out
 Before:           0        9999
 Before:           1        9999
 Before:           2        9999
 Before:           3        9999
Give n: 10
 After:           0          10
 After:           1          10
 After:           2          10
 After:           3          10
```

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

int main ()  {
  int n,t;

  omp_set_num_threads(4);
  n=9999;

#pragma omp parallel private(t) firstprivate(n)
  {
    t=omp_get_thread_num();
    printf("Before: %d %d\n",t,n);
#pragma omp barrier
#pragma omp single copyprivate(n)
    {
      printf("Give n: ");
      scanf("%d",&n);
    }
    printf("After: %d %d\n",t,n);
  }
  exit(0);
}
```

Parallel computations: threads, OpenMP and such

- The `workshare` directive is a tool to divide mainly array related tasks between threads.
  - It is only available in Fortran (Sorry guys!).

```
!$omp workshare
   . . .
!$omp end workshare [nowait]
```

  - The `workshare` directive divides the work of the enclosed code into separate units of work and distributes it to the threads.
  - The units of work may be assigned to threads in any manner as long as each unit is executed exactly once.
  - In the OpenMP specification there are accurate definitions of what constitutes a unit of work.

  - For array expressions within each statement:
    - Evaluation of each element of the array expression is a unit of work.
    - Evaluation of transformational array intrinsic functions may be subdivided into any number of units of work.
  - In array assignments, the assignment of each element is a unit of work.
  - Each scalar assignment operation is a single unit of work.
  - In application of an elemental function to an array argument each array element is treated as a unit of work.
  - `where` and `forall` constructs can be divided into units of work.
  - Each `atomic` directive and `critical` construct is a unit of work.
  - Each `parallel` construct is a single unit of work with respect to the `workshare` construct.
  - If none of the rules above apply to a portion of a statement in block, then that portion is a single unit of work.

  - The transformational array intrinsic functions are `matmul`, `dot_product`, `sum`, `product`, `maxval`, `minval`, `count`, `any`, `all`, `spread`, `pack`, `unpack`, `reshape`, `transpose`, `eoshift`, `cshift`, `minloc`, `maxloc`

Parallel computations: threads, OpenMP and such

- The code block within the `master` directive is executed only by the master thread:

```
!$omp master
   . . .
!$omp end master [nowait]
```

```
#pragma omp master
  {
    . . .
  }
```

- And just as in MPI OpenMP has the `barrier` directive:

```
!$omp barrier
```

```
#pragma omp barrier
```

  - The execution of threads is continued only after all threads have reached the `barrier` directive.

- To prevent race conditions a section of parallel code that should be executed by only one thread at a time can be enclosed by the `critical` directive:

```
!$omp critical [name]
   . . .
!$omp end critical [name]
```

```
#pragma omp critical [nanme]
  {
    . . .
  }
```

  - A critical section can be given a name which is visible to all other threads.
  - A thread waits at the beginning of a critical section until no other thread is executing a critical section with the same name.
  - All unnamed CRITICAL directives map to the same name.

Parallel computations: threads, OpenMP and such

- The `atomic` directive ensures that a specific memory location is updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads,

```
!$omp atomic
expression-statement
```

```
#pragma omp atomic
expression-statement
```

where `expression-statement` is one of

Fortran

```
x = x operator expr
x = expr operator x
x = intrinsic_procedute(x,expr_list)
x = intrinsic_procedute(expr_list,x)

operator = + * - / .and. .or. .eqv. .neqv.
```

C

```
x operator=expr
x++
++x
x--
--x

operator = + * - / & ^ | << >>
```

- The `flush` directive identifies a point at which the implementation is required to ensure that each thread in the team has a consistent view of certain variables in memory. It has the form

```
!$omp flush [list]
```

```
#pragma omp [list]
```

  - The directive should be used so that the threads using it are synchronized.
  - Many directives do an implied `flush`: `barrier`, `critical`, `end critical`, `end do`, `end sections`, `end single`, `end workshare`, `ordered`, `end ordered`, `parallel`, `end parallel`, `parallel do`, `end parallel`, `do parallel sections`, `end parallel sections`, `parallel workshare`, `end parallel workshare`
  - The `flush` directive is not implied if a `nowait` clause is present.

Parallel computations: threads, OpenMP and such

- OpenMP specification defines a couple of run-time library routines.

- Execution environment routines

`omp_set_num_threads(integer)`
  Sets the number of threads to use in the subsequent parallel regions.

`omp_get_num_threads()`
  Returns the number of threads currently executing.

`omp_get_max_threads()`
  Returns the maximum value that can be returned by the function `omp_get_num_threads()`.

`omp_get_thread_num()`
  Returns the number of the current thread. It is in the interval $[0 \ldots (\texttt{omp\_get\_num\_threads()} - 1)]$.

`omp_get_num_procs()`
  Returns the number of processors that are available to the program.

`omp_in_parallel()`
  Returns true if the current point of execution is in the parallel region.

Parallel computations: threads, OpenMP and such

- If **dynamic adjustment of number of threads** is enabled, the number of threads used in parallel regions can be adjusted automatically by the run-time environment.

`omp_set_dynamic(logical)` Enables or disables dynamic adjustment of the number of threads available for execution of parallel regions.
`omp_get_dynamic()` Returns true if the dynamic adjustment is enabled.

  - The default value returned by `omp_get_dynamic()` seems to vary:
    Intel Linux compilers, Pathscale, Tru64 Unix: `.false.`
    IBM SC with `xlf90_r`: `.true.`

- If **nested parallelism** is disabled (default), nested parallel regions are serialized and executed by the current thread. If enabled, nested parallel regions can deploy additional threads to form the team.

`omp_set_nested(logical)` Enables or disables nested parallelism.
`omp_get_nested()` Returns true if nested parallelism is enabled.

Parallel computations: threads, OpenMP and such

- Lock routines take lock variables as arguments. A lock variable must be accessed only through the routines described in this section. For all of these routines, a lock variable should be of type integer and of a large enough to hold an address[1].
  - **Nestable locks** may be locked multiple times by the same thread before being unlocked; **simple locks** may not be locked if they are already in a locked state. Below `svar` is a simple lock variable and `nvar` is a nestable lock variable.

`omp_init_lock(svar)`, `omp_init_nest_lock(nvar)`
Initialize lock variables
`omp_destroy_lock(svar)`, `omp_destroy_nest_lock(nvar)`
Free the lock variables.
`omp_set_lock(svar)`, `omp_set_nest_lock(nvar)`
Force the thread executing the subroutine to wait until the specified lock is available and then set the lock.
A nestable lock is available if it is unlocked or if it is already owned by the thread executing the subroutine.
`omp_unset_lock(svar)`, `omp_unset_nest_lock(nvar)`
Release ownership of the lock.
`omp_test_lock(svar)`, `omp_test_nest_lock(nvar)`
Attempt to set a lock but do not cause the execution of the thread to wait.

  - Timing routines

`omp_get_wtime()`
Returns a double precision value equal to the elapsed wallclock time in seconds since an arbitrary time in the past. Cf. `MPI_Wtime()`.
`omp_get_wtick()`
Returns a double precision value equal to the number of seconds between successive clock ticks.

1. In some impllementation a constant `omp_lock_kind` is defined for Fortran.

Parallel computations: threads, OpenMP and such

- An example of using lock with OpenMP.

```
/* Example from the OpenMP specification document. */
/* www.openmp.org */

#include <stdio.h>
#include <omp.h>

void work(int id);
void skip(int id);

int main()
{
  omp_lock_t lck;
  int id;

  omp_set_num_threads(4);
  omp_init_lock(&lck);

#pragma omp parallel shared(lck) private(id)
  {
    id = omp_get_thread_num();
    omp_set_lock(&lck);
    printf("My thread id is %d.\n", id);
    /* only one thread at a time can execute this printf */
    omp_unset_lock(&lck);
    while (! omp_test_lock(&lck)) {
      skip(id); /* we do not yet have the lock, so we must do something else */
    }
    work(id); /* we now have the lock and can do the work */
    omp_unset_lock(&lck);
  }

  omp_destroy_lock(&lck);
}

void work(int id) {};void skip(int id) {};
```


Parallel computations: threads, OpenMP and such

- Finally certain features of the OpenMP execution environment can be controlled by environmental variables.

`OMP_SCHEDULE`
: Applies to `do` and `do parallel` directives that have the schedule type `runtime`.

`OMP_NUM_THREADS`
: Sets the number of threads to use during execution, unless changed by `omp_set_num_threads` routine call.

`OMP_DYNAMIC`
: Enables or disables dynamic adjustment of the number of threads. Allowed values are `TRUE` and `FALSE`.

`OMP_NESTED`
: Enables or disabels nested parallelism. Allowed values are TRUE and FALSE.

Parallel computations: threads, OpenMP and such

• *This was a short introduction to parallel programming in shared memory machines using OpenMP.*

• *Resources for more information is easy to find:*

- One of the most important one is the OpenMP home page at http://www.openmp.org/
  - The OpenMP specification documents have many example code snippets.
    However, they are in PDF documents; I haven't found a link to download them in plain ASCII.
- NERSC[1] has a nice short Open MP tutorial (only Fortran and a bit IBM specific) at
  http://www.nersc.gov/nusers/help/tutorials/openmp/
- LLNL[2] OpenMP tutorial (both Fortran and C) is also good: http://www.llnl.gov/computing/tutorials/openMP/
  - They have also other tutorials; check out http://www.llnl.gov/computing/hpc/training/

1. National Energy Research Scientific Computing Center
2. Lawrence Livermore National Laboratory