An Overview of Parallel Computing

Flynn's Taxonomy (1966). Classification of computers according to number of instruction and data streams.

- SISD: Single Instruction Single Data, the standard uniprocessor computer (workstation).
- MIMD: Multiple Instruction Multiple Data, collection of autonomous processors working on their own data; the most general case.
- SIMD: Single Instruction Multiple Data; several CPUs performing the same instructions on different data.
  The CPUs are synchronized.
  Massively parallel computers.
  Works well on regular problems. PDE-grids, image processing.
  Often special languages and hardware. Not portable.

  Typical example, the Connection Machines from Thinking Machines (bankruptcy 1994).
  The CM-2 had up to 65536 (simple processors).
  PDC had a 16384 proc. CM200.

  Often called "data parallel".

Two other important terms:

- fine-grain parallelism - small tasks in terms of code size and execution time
- coarse-grain parallelism - the opposite

We talk about granularity.

MIMD Systems

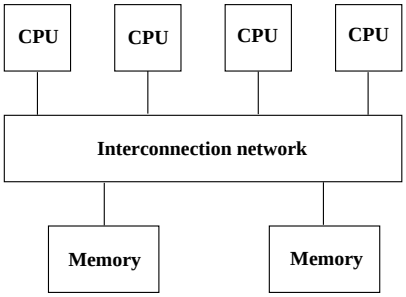
Asynchronous (the processes work independently).

- Shared-memory systems. The programmer sees one big memory. The physical memory can be distributed.
- Distributed-memory systems.
  Each processor has its own memory. The programmer has to partition the data.

The terminology is slightly confusing. A shared memory system usually has distributed memory (distributed shared memory). Hardware & OS handle the administration of memory.

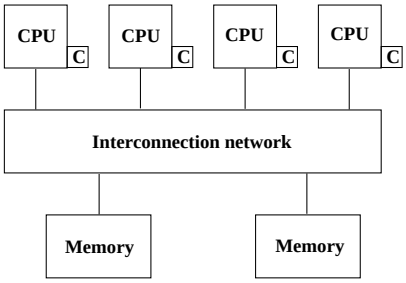
Shared memory

Bus-based architecture



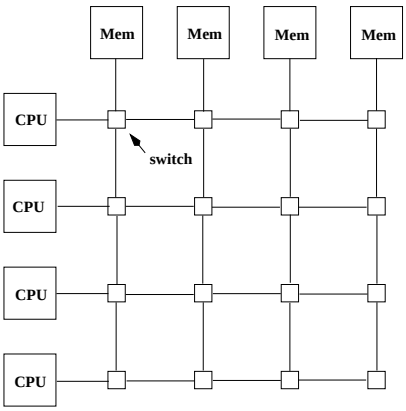
- Limited bandwidth (the amount of data that can be sent through a given communications circuit per second).
- Do not scale to a large number of processors. 30-40 CPUs common.

To work well each CPU has a cache (a local memory) for temporary storage.



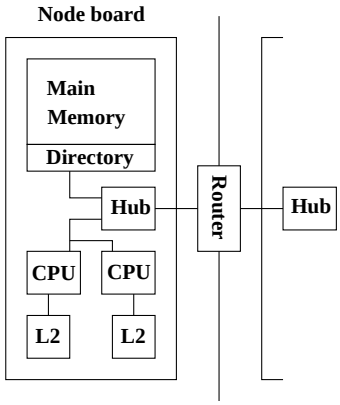
I have denoted the caches by C. Cache coherence.

Common to use a switch to increase the bandwidth. Crossbar:

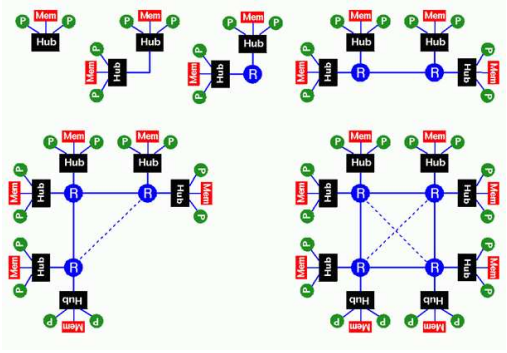


- Any processor can access any memory module. Any other processor can simultaneously access any other memory module.
- Expensive.
- Common with a memory hierarchy. Several crossbars may be connected by a cheaper network. NonUniform Memory Access (NUMA).

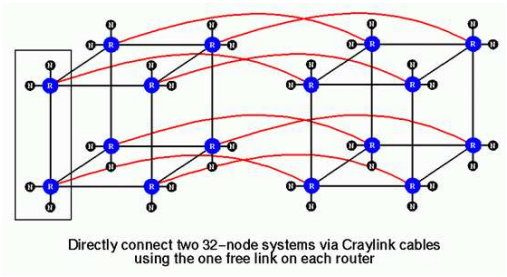
Example of a NUMA architecture: SGI Origin 2000, R10000 CPUS connected by a fast network.



The hub manages each processor's access to memory (both local and remote) and I/O. Local memory accesses can be done independently of each other. Accessing remote memory is more complicated and takes more time.



More than two nodes are connected via a router. A router has six ports. Hypercube configuration. When the system grows, add communication hardware for scalability.



Two important parameters of a network:

Latency is the startup time (the time it takes to send a small amount of data, e.g. one byte).

Bandwidth is the other important parameter.
How many bytes can we transfer per second (once the communication has started)?

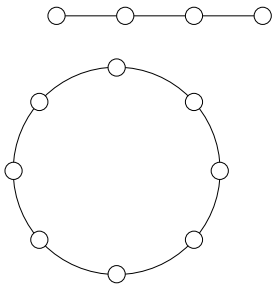
A simple model for communication:

$$time\ to\ transfer\ n\ bytes\ =\ latency + n\ /\ bandwidth$$

Distributed memory

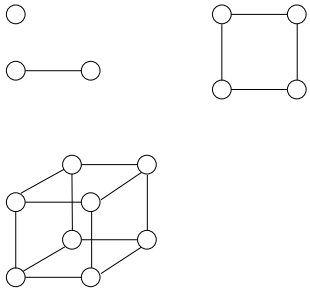
In a distributed memory system, each processor has its own private memory. A simple distributed memory system can be constructed by a number of workstations and a local network.

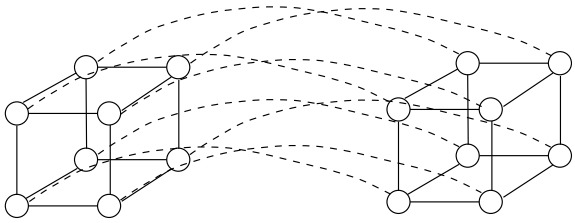
Some examples:



A linear array and a ring (each circle is a CPU with memory).

Hypercubes of dimensions 0, 1, 2 and 3.





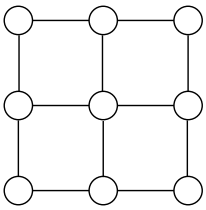
A 4-dimensional hypercube. Generally, a hypercube of dimension d+1 is constructed by connecting corresponding processors in two hypercubes of dimension d.

If d is the dimension we have 2^d CPUs, and the shortest path between any two nodes is at most d steps (passing d wires). This is much better than in a linear array or a ring. We can try to partition data so that the most frequent communication takes place between neighbours.

A high degree of connectivity is good because it makes it possible for several CPUs to communicate simultaneously (less competition for bandwidth). It is more expensive though.

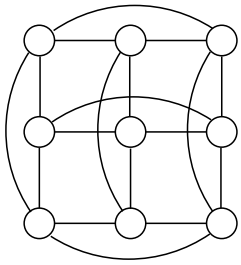
If the available connectivity (for a specific machine) is sufficient depends on the problem and the data layout.

This is a mesh:



We can have meshes of higher dimension.

If we connect the outer nodes in a mesh we get a torus:



A Note on Cluster Computing

Many modern parallel computers are built by off-the-shelf components, using personal computer hardware, Intel CPUs and Linux. Some years ago the computers were connected by an Ethernet network but faster (and more expensive) technologies are available. To run programs in parallel, explicit message passing is used (MPI, PVM).

The first systems were called Beowulf computers named after the hero in an Old English poem from around year 1000. They are also called Linux clusters and one talks about cluster computing.

In the poem, Beowulf, a hero of a tribe, from southern Sweden, called the Geats, travels to Denmark to help defeat Grendel (a monster), Grendel's mother and a dragon.
The first few lines (of about 3000) first in Old English and then in modern English:

■wæs on burgum
Beowulf Scyldinga,
leof leodcyning, longe þrage
folcum gefræge (fæder ellor hwearf,
aldor of earde), oþ■æt him eft onwoc
heah Healfdene; heold þenden lifde,
gamol ond gu■reouw, glæde Scyldingas.

Now Beowulf bode in the burg of the Scyldings,
leader beloved, and long he ruled
in fame with all folk, since his father had gone
away from the world, till awoke an heir,
haughty Healfdene, who held through life,
sage and sturdy, the Scyldings glad.

A look at the Lenngren cluster at PDC

PDC (Parallell-Dator-Centrum) is the Center for Parallel Computers, Royal Institute of Technology in Stockholm.

Lenngren (after the Swedish poet Anna Maria Lenngren, 1754-1817) is a distributed memory computer from Dell consisting of 442 nodes. Each node has two 3.4GHz EMT64-Xeon processors (EM64T stands for Extended Memory x 64-bit Technology) and 8GB of main memory. The peak performance of the system is 6Tflop/s. The nodes are connected with gigabit ethernet for login and filesystem traffic. A high performance Infiniband network from Mellanox is used for the MPI traffic.

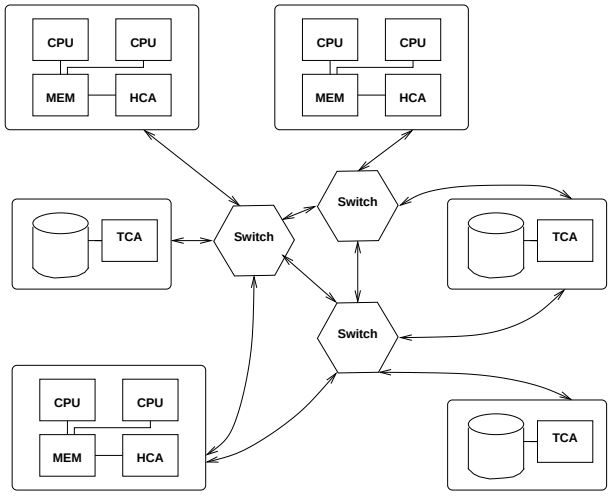
A word on Infiniband. First a quote from
`http://www.infinibandta.org/`:
"InfiniBand is a high performance, switched fabric interconnect standard for servers. ... Founded in 1999, the InfiniBand Trade Association (IBTA) is comprised of leading enterprise IT vendors including Agilent, Dell, Hewlett-Packard, IBM, SilverStorm, Intel, Mellanox, Network Appliance, Oracle, Sun, Topspin and Voltaire. The organization completed its first specification in October 2000."

Another useful reference is `http://en.wikipedia.org`.

InfiniBand uses a bidirectional serial bus, 2.5 Gbit/s in each direction. It also supports double and quad data rates for 5 Gbit/s or 10 Gbit/s respectively. For electrical signal reasons 8-bit
symbols are sent using 10-bits (8B/10B encoding), so the actual data rate is 4/5ths of the raw rate.
Thus the single, double and quad data rates carry 2, 4 or 8 Gbit/s respectively.

Links can be aggregated in units of 4 or 12, called 4X or 12X. A quad-rate 12X link therefore carries 120 Gbit/s raw, or 96 Gbit/s of user data.

InfiniBand uses a switched fabric topology so several devices can share the network at the same time (as opposed to a bus topology). Data is transmitted in packets of up to 4 kB. All transmissions begin or end with a channel adapter. Each processor contains a host channel adapter (HCA) and each peripheral has a target channel adapter (TCA). It may look something like this:



Switches forward packets between two of their ports based on an established routing table and the addressing information stored on the packets. A subnet, like the one above, can be connected to another subnet by a router.
Each channel adapter may have one or more ports. A channel adapter with more than one port, may be connected to multiple switch ports. This allows for multiple paths between a source and a destination, resulting in performance and reliability benefits.

A simple example

Consider the following algorithm (the power method). A is a square matrix of order n (n rows and columns) and $x^{(k)}$, $k = 1, 2, 3, \ldots$ a sequence of column vectors, each with n elements.

$x^{(1)} = random\ vector$
for k = 1, 2, 3, ...
  $x^{(k+1)} = Ax^{(k)}$
end

If A has a dominant eigenvalue λ ($|\lambda|$ is strictly greater than all the other eigenvalues) with eigenvector x, then $x^{(k)}$ will be a good approximation of an eigenvector for sufficiently large k (provided $x^{(1)}$ has a nonzero component of x).

An Example:

```
>> A=[-10 3 6;0 5 2;0 0 1]  % it is not necessary
A =                         % that A is triangular
   -10     3     6
     0     5     2
     0     0     1
>> x = randn(3, 1);
>> for k = 1:8, x(:, k+1) = A * x(:, k); end
>> x(:,1:4)
ans =
  -6.8078e-01   5.0786e+00  -5.0010e+01   5.1340e+02
   4.7055e-01   1.3058e+00   5.4821e+00   2.6364e+01
  -5.2347e-01  -5.2347e-01  -5.2347e-01  -5.2347e-01

>> x(:,5:8)
ans =
  -5.0581e+03   5.0970e+04  -5.0774e+05   5.0872e+06
   1.3077e+02   6.5281e+02   3.2630e+03   1.6314e+04
  -5.2347e-01  -5.2347e-01  -5.2347e-01  -5.2347e-01
```

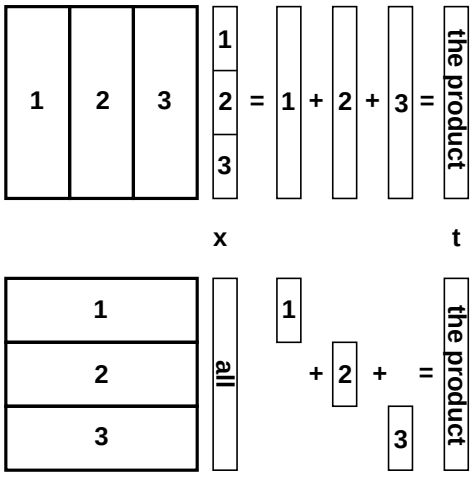
Note that $x^{(k)}$ does not "converge" in the ordinary sense.
We may have problems with over/underflow.

Revised algorithm, where we scale $x^{(k)}$ and keep only one copy.

$x = random\ vector$
$x = x\ (1/\max(|x|))$ Divide by the largest element
for k = 1, 2, 3, ...
  $t = Ax$
  $x = t\ (1/\max(|t|))$
end

λ can be computed in several ways, e.g. x^TAx/x^Tx (and we already have $t = Ax$). In practice we need to terminate the iteration as well. Let us skip those details.

How can we make this algorithm parallel on a distributed memory MIMD-machine (given A)? One obvious way is to compute $t = Ax$ in parallel. In order to do so we must know the topology of the network and how to partition the data.



Suppose that we have a ring with $\#p$ processors and that $\#p$ divides n. We partition A in blocks of $\beta = n/\#p$ (β for block size) rows (or columns) each, so that processor 1 would store rows 1 through β, processor 2 rows $1+\beta$ through 2β etc. Let us denote these blocks of rows by A_1, $A_2, \ldots, A_{\#p}$. If we partition t in the same way t_1 contains the first β elements, t_2 the next β etc, t can be computed as:

$$\begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_{\#p} \end{bmatrix} = Ax = \begin{bmatrix} A_1x \\ A_2x \\ \vdots \\ A_{\#p}x \end{bmatrix} \begin{matrix} \leftarrow \text{ on proc. 1} \\ \leftarrow \text{ on proc. 2} \\ \vdots \\ \leftarrow \text{ on proc. \#p} \end{matrix}$$

In order to perform the next iteration processor one needs $t_2, \ldots, t_{\#p}$, processor two needs $t_1, t_3, \ldots, t_{\#p}$ etc.
The processors must communicate, in other words.

Another problem is how each processor should get its part, A_j, of the matrix A. This could be solved in different ways:

- one CPU gets the task to read A and distributes the parts to the other processors
- perhaps each CPU can construct its A_j by computation
- perhaps each CPU can read its part from a file (or from files)

Let us assume that the A_j have been distributed and look at the matrix-vector multiply.

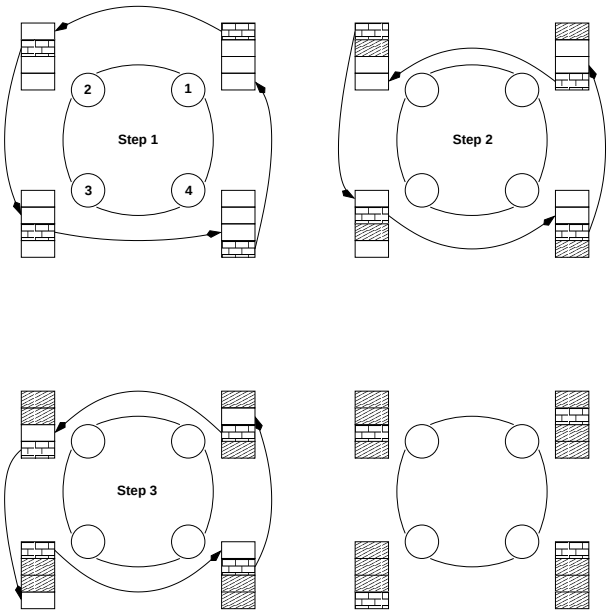
Processor number p would do the following (I have changed the logic slightly):

$p = which\_processor\_am\_i()$ $(1, 2, \ldots, \#p)$
for $k = 0, 1, 2, \ldots$ *do*
  if ($k == 0$) *then* not so nice (but short)
    $x_p = random\ vector$ of length β
  else
    $t_p = A_px$
    $\mu = 1/\max(\mu_1, \mu_2, \ldots, \mu_{\#p})$
    $x_p = \mu\ t_p$
  end if
  $\mu_p = \max(|x_p|)$

  $seg = p$
  for $j = 1$ to $\#p - 1$ *do*
    send x_{seg}, μ_{seg} to the next processor
    compute seg
    receive x_{seg}, μ_{seg} from the previous processor
  end do
end do

An alternative to computing seg is to send a message containing seg; "send seg, x_{seg}, μ_{seg}". The program looks very much like a SIMD-program.

Here is an image showing (part of) the algorithm, when $\#p = 4$. White boxes show not yet received parts of the vector. The brick pattern shows the latest part of the vector and the boxes with diagonal lines show old pieces.



Some important terminology:

Let wct (wallclock time) be the time we have to wait for the run to finish (i.e. not the total cputime). wct is a function of $\#p$, $wct(\#p)$ (although it may not be so realistic to change $\#p$ in a ring.
This is a simple model of this function (for one iteration):

$$wct(\#p) = \frac{2n^2}{\#p} T_{flop} + (\#p - 1)\left[T_{lat} + \frac{n}{\#p} T_{bandw}\right]$$

where T_{flop} is the time for one flop, T_{lat} is the latency for the communication and T_{bandw} is time it takes to transfer one double precision number.

It is often the case that (roughly):

$$wct(\#p) \;=\; seq.\ part\ of\ comp. + \frac{parallel\ part\ of\ comp.}{\#p} + \#p \;\; (communication)$$

wct has a minimum with respect to $\#p$ (it is not optimal with $\#p = \infty$). The computational time decreases with $\#p$ but the communication increases.

The $speedup$ is defined as the ratio:

$$speedup(\#p) = \frac{wct(1)}{wct(\#p)}$$

What we hope for is linear speedup, i.e. $speedup(\#p) = \#p$.

If you have a problem to solve (rather than an algorithm to study) a more interesting definition may be:

$$speedup(\#p) = \frac{time\ for\ best\ implementation\ on\ one\ processor}{wct(\#p)}$$

It is possible to have super linear speedup, $speedup(\#p) > \#p$; this is usually due to better cache locality or decreased paging.

If our algorithm contains a section that is sequential (cannot be parallelized), it will limit the $speedup$. This is known as Amdahl's law. Let us denote the sequential part with with s, $0 \leq s \leq 1$ (part wrt time), so the part that can be parallelized is $1 - s$. Hence,

$$speedup(\#p) = \frac{1}{s + (1-s)/\#p} \leq \frac{1}{s}$$

regardless of the number of processors.

If you have to pay for the computer time (or if you share resources) the efficiency is interesting. The efficiency measures the fraction of time that a typical processor is usefully employed.

$$efficiency(\#p) = \frac{speedup(\#p)}{\#p}$$

We would like to have $efficiency(\#p) = 1$.

The proportion of unused time per processor is:

$$\frac{wct(\#p) - \frac{wct(1)}{\#p}}{wct(\#p)} = 1 - \frac{wct(1)}{wct(\#p)\#p} = 1 - efficiency(\#p)$$

Instead of studying how the $speedup$ depends on $\#p$ we can fix $\#p$ and see what happens when we change the size of the problem n. Does the $speedup$ scale well with n? In our case:

$$\begin{aligned} speedup(n) \;&=\; \frac{2n^2 T_{flop}}{\frac{2n^2 T_{flop}}{\#p} + (\#p - 1)\left[T_{lat} + \frac{nT_{bandw}}{\#p}\right]} \\ &=\; \frac{\#p}{1 + (\#p - 1)\left[\frac{\#p T_{lat}}{2n^2 T_{flop}} + \frac{T_{bandw}}{2nT_{flop}}\right]} \end{aligned}$$

So

$$\lim_{n \to \infty} speedup(n) = \#p$$

This is very nice! The computation is $\mathcal{O}(n^2)$ and the communication is $\mathcal{O}(n)$. This is not always the case.

Exercise: partition A by columns instead.

What happens if the processors differ in speed and amount of memory? We have a load balancing problem.

Static load balancing: find a partitioning $\beta_1, \beta_2, \ldots, \beta_{\#p}$ such that processor p stores β_p rows and so that wct is minimized over this partitioning. We must make sure that a block fits in the available memory on node p. This leads to the optimization problem:

$$\min_{\beta_1, \beta_2, \ldots, \beta_{\#p}} wct(\beta_1, \beta_2, \ldots, \beta_{\#p}),$$

subject to the equality constraint $\sum_{p=1}^{\#p} \beta_p = n$ and the p inequality constraints $8n\beta_p \leq M_p$, if M_p is the amount of memory (bytes) available on node p.

If

- the amount of work varies with time
- we share the processors with other users
- processors crash ($\#p$ changes)

we may have to rebalance; dynamic load balancing.

Even if the processors are identical (and with equal amount of memory) we may have to compute a more complicated partitioning. Suppose that A is upper triangular (zeros below the diagonal). (We would not use an iterative method to compute an eigenvector in this case.) The triangular matrix is easy to partition, it is worse if A is a general sparse matrix (many elements are zero).

Some matrices require a change of algorithm as well. Suppose that A is symmetric, $A = A^T$ and that we store A in a compact way (only one triangle).

Say, $A = U^T + D + U$ (Upper^T+ Diagonal + Upper).

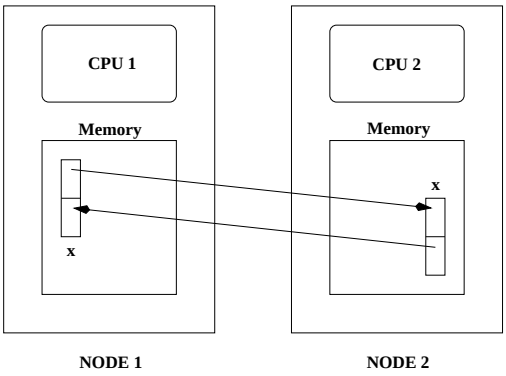
If we store U and D by rows it is easy to compute $Ux + Dx$ using our row-oriented algorithm. To compute $U^T x$ requires a column-oriented approach (if U is partitioned by rows, U^T will be partitioned by columns, and a column-oriented algorithm seems reasonable). So the program is a combination of a row and a column algorithm.

A few words about communication

In our program we had the loop:

for $j = 1$ to $\#p - 1$
  send x_{seg}, μ_{seg} to the next processor
  compute seg
  receive x_{seg}, μ_{seg} from the previous processor
end

Suppose $\#p = 2$ and that we can transfer data from memory (from x_1 on processor one to x_1 on processor two, from x_2 on processor two to x_2 on processor one).



There are several problems with this type of communication, e.g.:

- if CPU 1 has been delayed it may be using x_2 when CPU 2 is writing in it
- several CPUs may try to write to the same memory location (in a more general setting)
- CPU 1 may try to use data before CPU 2 has written it

So, a few things we would like to able to do:

- wait for a message until we are ready to take care of it
- do other work while waiting (to check now and then)
- find out which processor has sent the message
- have identities of messages (one CPU could send several; how do we distinguish between them)
- see how large the message is before unpacking it
- send to a group of CPUs (broadcast)

An obvious way to solve the first problem is to use synchronisation. Suppose CPU 1 is delayed. CPU 2 will send a "ready to send"-message to CPU 1 but it will not start sending data until CPU 1 has sent a "ready to receive"-message.

This can cause problems. Suppose we have a program where both CPUs make a send and then a receive. If the two CPUs make sends to each other the CPUs will "hang". Each CPU is waiting for the other CPU to give a "ready to receive"-message. We have what is known as a deadlock.

One way to avoid this situation is to use a buffer. When CPU 1 calls the send routine the system copies the array to a temporary location, a buffer. CPU 1 can continue executing and CPU 2 can read from the buffer (using the receive call) when it is ready. The drawback is that we need extra memory and an extra copy
operation.

Suppose now that CPU 1 lies ahead and calls receive before CPU 2 has sent. We could then use a blocking receive that waits until the messages is available (this could involve synchronised or buffered communication). An alternative is to use a nonblocking receive. So the receive asks: is there a message? If not, the CPU could continue working and ask again later.

Process control under unix

Processes are created using the **fork**-system call. System call: the mechanism used by an application program to request service from the operating system (from the unix-kernel). **man -s2 intro**, **man -s2 syscalls** . **printf** (for example) is not a system call but a library function. **man -s3 intro** for details.

```
#include <sys/wait.h>   /* for wait             */
#include <sys/types.h>  /* for wait and fork    */
#include <unistd.h>     /* for fork and getppid */
#include <stdio.h>

int main()
{
  int             var, exit_stat;
  pid_t           pid;

  var = 10;
  printf("Before fork\n");

  if ((pid = fork()) < 0) {  /* note ( ) */
    printf("fork error\n");
    return 1;
  } else if (pid == 0) {     /* I am a child */
    var++;
    printf("child\n");
    sleep(60);               /* do some work  */
  } else {                   /* I am a parent */
    printf("parent\n");
    wait(&exit_stat);        /* wait for (one)     */
  }                          /* child to exit; not */
                             /* necessary to wait  */

  printf("ppid = %6ld, pid = %6ld, var  = %d\n",
         getppid(), pid, var); /* get parent proc id */
  return 0;
}
```

```
% a.out
Before fork
child
parent
ppid =   6843, pid =      0, var  = 11   child
ppid =  27174, pid =   6844, var  = 10   parent
```

fork creates the child process (from the parent process) by making a copy of the parent (so the child gets copies of the heap and stack for example). The child and parent continue executing with the instruction that follows the call to **fork**. So **fork** is called from the parent and returns both to the parent and the child.

Every process is identified by a number, the process id. or pid. We can list the pids (and some other properties) of all the processes running in the computer (this list has been shortened). The **ps**-commando takes a huge number of options.

```
% ps -fel | grep thomas
UID        PID    PPID   CMD
thomas    5442   27174   xterm
thomas    5446    5442   -csh

thomas    6843   27174   a.out  <-- parent
thomas    6844    6843   a.out  <-- child

thomas    6851    5446   ps -fel
thomas    6852    5446   grep thomas

thomas   27174   27171   -tcsh
thomas   27171   27152   sshd: thomas@pts/62
root     27152    3203   sshd: thomas [priv]
root      3203       1   /usr/sbin/sshd
root         1       0   init [5]
 ...
```


A process that hangs (not uncommon in parallel programming) can be terminated using the **kill**-command which sends a signal to a process. There are different signals and they can be used for communication between processes. Signal number 9, sigkill, cannot be caught.

```
% kill -l
HUP INT QUIT ILL TRAP ABRT BUS FPE KILL USR1 SEGV USR2
 ...

% ps U thomas
  PID TTY      STAT   TIME COMMAND
 8604 pts/62   S+     0:00 a.out   <-- kill this one
...
% kill -9 8604 (or kill -KILL 8604)
```

A process can choose to catch the signal using a a signal handler routine. It can also ignore (some) signals:

```
#include <signal.h>
#include <stdio.h>
int main()
{
  /* SIGINT is defined /usr/include/bits/signum.h */
  if ( sigignore(SIGINT) == -1 )
    printf("*** Error when calling sigignore.\n");

  while(1)      /* loop forever */
        ;
  return 0;
}

% gcc signal.c
% a.out
^C^C^C^C^C^C^C^C^C^C^\Quit

% /bin/stty -a
intr = ^C; quit = ^\; erase = ^H; etc....
```


To start a child process that differs from the parent we use the **exec** system call (there are several forms). **exec** replaces the child (the process which it is called from) with a new program.

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

int main()
{
  int             exit_stat;
  pid_t           pid;

  if ((pid = fork()) < 0) {
    printf("fork error\n");
    return 1;
  } else if (pid == 0) {  /* I am a child */
    /* replace this process by another   */
    /* execlp( file, name_shown_by_ps,
               arg1, ..., argn, NULL)    */

    /* (char *) 0 is a null pointer. (char *)
       is a type cast. See the C FAQ for details. */

    /* new is a compiled C-program */
    if(execlp("new", "new_name", (char *) 0) < 0) {
           printf("*** execlp error\n");
           return 1;
    }

  } else                  /* I am a parent. Wait  */
    wait(&exit_stat);     /* or do something else */
  return 0;
}
```

Very common usage in **command&** .

Interprocess communication

Most parallel computing tasks require communication between processes. This can be accomplished in several different ways on a unix system. The pipe, **|**, is a standard example:

```
% ps aux | grep a.out
```

The **ps** and **grep** processes are running in parallel and are communicating using a pipe. Data flows in one direction and the processes must have a common ancestor. The pipe handles synchronisation of data (**grep** must wait for data from **ps** and **ps** may not deliver data faster than **grep** can handle, for example).

The communication is usually local to one system, but using **rsh** (remote shell) or **ssh** (secure shell) it may be possible to communicate between different computers:

```
% ps aux | ssh other_computer "grep a.out > /tmp/junk"
```

/tmp/junk is created on **other_computer**. (There are other remote commands such as **rcp/scp**, remote copy).

FIFOs (or named pipes) can be used to communicate between two unrelated processes. A general way to communicate between computers over a network is to use so called sockets.

When a (parallel) computer has shared memory it is possible to communicate via the memory. Two (or more processes) can share a portion of the memory. Here comes a master (parent) program.

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

int main()
{
  int             exit_stat, shmid, info, k;
  pid_t           pid;
  struct shmid_ds buf;
  double         *shmaddr;
  char            s_shmid[10];
  /*
   * Create new shared memory segment and then
   * attach it to the address space of the process.
   */
  shmid=shmget(IPC_PRIVATE, (size_t) 512, SHM_R|SHM_W);
  shmaddr = shmat(shmid, (void *) 0, 0);

  /* Store some values */
  for (k = 0; k < 512 / 8; k++)
    *(shmaddr + k) = k;

  /* Create new proces */
  if ((pid = fork()) < 0) {
    printf("fork error\n");
    return 1;
  } else if (pid == 0) {        /* I am a child */
```



```
    /* convert int to string */
    sprintf(s_shmid, "%d", shmid);

    if (execlp("./child", "child_name", s_shmid,
               (char *) 0) < 0) {
      printf("*** In main: execlp error.\n");
      return 1;
    }
  } else {
    wait(&exit_stat);
    /* Remove the segment. */
    info = shmctl(shmid, IPC_RMID, &buf);
  }
  return 0;
}
```

Here comes a slave (child) program.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main(int argc, char *argv[])
{ int             k, shmid;
  double         *shmaddr;

  printf("In child\n"); printf("argc = %d\n", argc);
  printf("argv[0] = %s\nargv[1] = %s\n",argv[0],argv[1]);

  shmid = atoi(argv[1]);        /* convert to int */
  printf("shmid   = %d\n", shmid);
  shmaddr = shmat(shmid, (void *) 0, SHM_RDONLY);

  for (k = 0; k < 5; k++) /* "Fetch" and print values.*/
    printf("*(shmaddr+%d) = %f\n", k, *(shmaddr + k));
  return 0;
}
```



```
% gcc -o master master.c
% gcc -o child child.c
% master
In child
argc = 2
argv[0] = child_name
argv[1] = 22183946
shmid   = 22183946
*(shmaddr+0) = 0.000000
*(shmaddr+1) = 1.000000
*(shmaddr+2) = 2.000000
*(shmaddr+3) = 3.000000
*(shmaddr+4) = 4.000000
```

In general some kind of synchronisation must be used when accessing the memory. There are such tools (e.g. semaphores) but since we will look at a similar construction in the next section we drop the subject for now.

Using the command **ipcs** we can get a list of segments. It may look like:

```
% ipcs
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      na
status
0x00000000 22249482   thomas    600        512        0

... more stuff
```

In case of problems we can remove segments, e.g.
ipcrm -m 22249482.

POSIX Threads (pthreads)

(POSIX: Portable Operating System Interface, A set of IEEE standards designed to provide application portability between Unix variants. IEEE: Institute of Electrical and Electronics Engineers, Inc. The world's largest technical professional society, based in the USA.)

Unix process creation (and context switching) is rather slow and different processes do not share much (if any) information (i.e. they may take up a lot of space).

A thread is like a "small" process. It originates from a process and is a part of that process. All the threads share global variables, files, code, PID etc. but they have their individual stacks and program counters.

When the process has started, one thread, the master thread, is running. Using routines from the pthreads library we can start more threads.

If we we have a shared memory parallel computer each thread may run on its own processor, but threads are a convenient programming tool on a uniprocessor as well.

In the example below a dot product, $\sum_{i=1}^{n} a_i b_i$, will be computed in parallel. Each thread will compute part of the sum. We could, however, have heterogeneous tasks (the threads do not have do do the same thing).

We compile by:

```
gcc prog.c -lpthread
```

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

/* global shared variables */
#define VEC_LEN   400
#define N_THREADS   4
double            a[VEC_LEN], b[VEC_LEN], sum;
pthread_mutex_t   mutexsum;

void *dotprod(void *arg) /* the slave */
{
  int     i, start, end, i_am, len;
  double  mysum;

  i_am = (int) arg;

  /* assume that N_THREADS divides VEC_LEN */
  len   = VEC_LEN / N_THREADS;
  start = i_am  * len;
  end   = start + len;

  mysum = 0.0; /* local sum */
  for (i = start; i < end; i++)
    mysum += a[i] * b[i];

  /* update global sum with local sum */
  pthread_mutex_lock(&mutexsum);
    sum += mysum;  /* critical section */
  pthread_mutex_unlock(&mutexsum);

  /* terminate the thread, NULL is the null-pointer */
  pthread_exit(NULL); /* not really needed */
  return NULL;        /* to silence lint   */
}
```



```c
int main()
{
  pthread_t    thread_id[N_THREADS];
  int          i, ret;

  for (i = 0; i < VEC_LEN; i++) {
    a[i] =  1.0;  /* initialize */
    b[i] = a[i];
  }
  sum = 0.0;      /* global sum, NOTE declared global */

/* Initialize the mutex (mutual exclusion lock). */
  pthread_mutex_init(&mutexsum, NULL);

/* Create threads to perform the dotproduct
   NUll implies default properties.         */

  for(i = 0; i < N_THREADS; i++)
    if( ret = pthread_create(&thread_id[i], NULL,
                             dotprod, (void *) i)){
      printf ("Error in thread create\n");
      exit(1);
    }

/* Wait for the other threads. If the main thread
   exits all the slave threads will exit as well.  */

  for(i = 0; i < N_THREADS; i++)
     if( ret = pthread_join(thread_id[i], NULL) ) {
       printf ("Error in thread join %d \n", ret);
       exit(1);
     }

  printf ("sum = %f\n", sum);
  pthread_mutex_destroy(&mutexsum);
  return 0;
}
```


This is what the run looks like. Since the threads have the same PID we must give a special option to the ps-command to see them.

```
% a.out
sum = 400.000000
  ...

% ps -felL | grep thomas | grep a.out (edited)
UID        PID  PPID   LWP  NLWP CMD
thomas   15483 27174 15483     5 a.out  <-- master
thomas   15483 27174 15484     5 a.out
thomas   15483 27174 15485     5 a.out
thomas   15483 27174 15486     5 a.out
thomas   15483 27174 15487     5 a.out
```

LWP id. of light weight process (thread).
NLWP number of lwps in the process.

Note that the PID is the same.

Race conditions, deadlock etc.

When writing parallel programs it is important not to make any assumptions about the order of execution of threads or processes (e.g that a certain thread is the first to initialize a global variable). If one makes such assumptions the program may fail occasionally (if another thread would come first). When threads compete for resources (e.g. shared memory) in this way we have a race condition. It could even happen that threads deadlock (deadlock is a situation where two or more processes are unable to proceed because each is waiting for one of the others to do something).

From the web: I've noticed that under LinuxThreads (a kernel-level POSIX threads package for Linux) it's possible for thread B to be starved in a bit of code like the fragment at the end of this message (not included). I interpreted this as a bug in the mutex code, fixed it, and sent a patch to the author. He replied by saying that the behavior I observed was correct, it is perfectly OK for a thread to be starved by another thread of equal priority, and that POSIX makes no guarantees about mutex lock ordering. ... I wonder (1) if the behavior I observed is within the standard and (2) if it is, what the f%^& were the POSIX people thinking? ...
Sorry, I'm just a bit aggravated by this.
Any info appreciated,
Bill Gribble

According to one answer it is within the standard.

When I taught the course 2002, Solaris pthreads behaved this way, but this has changed in Solaris 9. Under Linux (2005) there are no problems, so I will not say more about this subject.

Message Passing Software

Several packages available. The two most common are PVM (Parallel Virtual Machine) and MPI (Message Passing Interface).

The basic idea in these two packages is to start several processes and let these processes communicate through explicit message passing. This is done using a subroutine library (Fortran & C). The subroutine library usually uses unix sockets (on a low level). It is possible to run the packages on a shared memory machine in which case the packages can communicate via the shared memory. This makes it possible to run the code on many different systems.

```
  call pvmfinitsend( PVMDEFAULT, bufid )
  call pvmfpack( INTEGER4, n, 1, 1, info )
  call pvmfpack( REAL8, x, n, 1, info )
  call pvmfsend( tid, msgtag, info )

  bufid = pvm_initsend( PvmDataDefault );
  info  = pvm_pkint( &n, 1, 1 );
  info  = pvm_pkdouble( x, n, 1 );
  info  = pvm_send( tid, msgtag );


  call MPI_Send(x, n, MPI_DOUBLE_PRECISION, dest, &
                tag, MPI_COMM_WORLD, err)

  err = MPI_Send(x, n, MPI_DOUBLE, dest,
                 tag, MPI_COMM_WORLD);
```

In MPI one has to work a bit more to send a message consisting of several variables. In PVM it is possible to start processes dynamically, and to run several different **a.out**-files. In MPI the processes must be started using a special unix-script and only one **a.out** is allowed (at least in MPI version 1).

PVM is available in one distribution, **pvm3.4.4**, (see the home page). (Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, Vaidy Sunderam.) Free book available on the net (PostScript & HTML).

Some of the systems PVM runs on (this is an old list; systems have been added):

AFX8, Alliant FX/8, ALPHA, DEC Alpha/OSF-1, ALPHAMP, DEC Alpha/OSF-1 / using shared memory, APOLLO, HP 300 running Domain/OS, ATT, AT&T/NCR 3600 running SysVR4, BAL, Sequent Balance, BFLY, BBN Butterfly TC2000, BSD386, 80[345]86 running BSDI or BSD386, CM2, Thinking Machines CM-2 Sun front-end, CM5, Thinking Machines CM-5, CNVX, Convex using IEEE floating-point, CNVXN, Convex using native f.p., CRAY, Cray, CRAY2, Cray-2, CRAYSMP, Cray S-MP, CSPP, Convex Exemplar, DGAV, Data General Aviion, E88K, Encore 88000, FREEBSD, 80[345]86 running FreeBSD, HP300, HP 9000 68000 cpu, HPPA, HP 9000 PA-Risc, HPPAMP, HP 9000 PA-Risc / shared memory transport, KSR1, Kendall Square, I860, Intel RX Hypercube, IPSC2, Intel IPSC/2, LINUX, 80[345]86 running Linux, M88K, Motorola M88100 running Real/IX, MASPAR, Maspar, MIPS, Mips, NETBSDAMIGA, Amiga running NetBSD, NETBSDHP300, HP 300 running NetBSD, NETBSDI386, 80[345]86 running NetBSD, NETBSDMAC68K, Macintosh running NetBSD, NETBSDPMAX, DEC Pmax running NetBSD, NETBSDSPARC, Sparc running NetBSD, NETBSDSUN3, SUN 3 running NetBSD, NEXT, NeXT, PGON, Intel Paragon, PMAX, DEC/Mips arch (3100, 5000, etc.), RS6K, IBM/RS6000, RS6KMP, IBM SMP / shared memory transport, RT, IBM/RT, SCO, 80[345]86 running SCO Unix, SGI, Silicon Graphics IRIS, SGI5, Silicon Graphics IRIS running OS $\geq$ 5.0, SGI64, Silicon Graphics IRIS running OS $\geq$ 6.0, SGIMP, Silicon Graphics IRIS / OS 5.x / using shared memory, SGIMP64, Silicon Graphics IRIS / OS 6.x / using shared memory, SP2MPI, IBM SP-2 / using MPI, SUN3, Sun 3, SUN4, Sun 4, 4c, sparc, etc., SUN4SOL2, Sun 4 running Solaris 2.x, SUNMP, Sun 4 / using shared memory / Solaris 2.x, SX3, NEC SX-3, SYMM, Sequent Symmetry, TITN, Stardent Titan, U370, IBM 3090 running AIX, UTS2, Amdahl running UTS, UVAX, DEC/Microvax, UXPM, Fujitsu running UXP/M, VCM2, Thinking Machines CM-2 Vax front-end, X86SOL2, 80[345]86 running Solaris 2.x.

PVM can be run in several different ways. Here we add machines to the virtual machine by using the PVM-console:

```
pvm> conf
1 host, 1 data format
                    HOST     DTID     ARCH   SPEED
   ries.math.chalmers.se    40000 SUN4SOL2    1000
pvm> add fibonacci
1 successful
                    HOST     DTID
                fibonacci    80000
pvm> add fourier
1 successful
                    HOST     DTID
                 fourier    c0000
pvm> add pom.unicc
1 successful
                    HOST     DTID
                pom.unicc   100000
pvm> conf
4 hosts, 1 data format
                    HOST     DTID     ARCH   SPEED
   ries.math.chalmers.se    40000 SUN4SOL2    1000
                fibonacci    80000 SUN4SOL2    1000
                  fourier    c0000 SUN4SOL2    1000
                pom.unicc   100000    SUNMP    1000
pvm> help
help   - Print helpful information about a command
Syntax:  help [ command ]
Commands are:
  add    - Add hosts to virtual machine
  alias  - Define/list command aliases
  conf   - List virtual machine configuration
  delete - Delete hosts from virtual machine
  etc.

pvm> halt
```


It is possible to add machines that are far away and of different architectures. The add command start a **pvmd** on each machine (**pvmd** pvm-daemon). The **pvmd**s relay messages between hosts.

The PVM-versions that are supplied by the vendors are based on the public domain (pd) version.
Common to write master/slave-programs (two separate main-programs). Here is the beginning of a master:

```
  program master
#include "fpvm3.h"
  ...
  call pvmfmytid ( mytid ) ! Enroll program in pvm
  print*, 'How many slaves'
  read*,   nslaves

  name_of_slave  = 'slave' ! pvmd looks in a spec. dir.
  arch           = '*'     ! any will do
  call pvmfspawn ( name_of_slave, PVMDEFAULT, arch,
 +                 nslaves, tids, numt )
```

The beginning of the slave may look like:

```
  program slave
#include "fpvm3.h"
  ...
  call pvmfmytid ( mytid )   ! Enroll program in pvm
  call pvmfparent ( master ) ! Get the master's task id.
*     Receive data from master.
  call pvmfrecv ( master, MATCH_ANYTHING, info )
  call pvmfunpack ( INTEGER4, command, 1, 1, info )
```

There are several pd-versions of MPI. The Sun-implementation is based on mpich (Argonne National Lab.).

Here comes a simple MPI-program.

```
#include <stdio.h>
#include "mpi.h"    /* Important */

int main(int argc, char *argv[])
{
  int             message, length, source, dest, tag;
  int             n_procs; /* number of processes */
  int             my_rank; /* 0, ..., n_procs-1   */
  MPI_Status      status;

  MPI_Init(&argc, &argv);  /* Start up MPI */

  /* Find out the number of processes and my rank */
  MPI_Comm_size(MPI_COMM_WORLD, &n_procs);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

  tag    = 1;
  length = 1;        /* Length of message         */

  if (my_rank == 0) { /* I'm the master process   */
    printf("Number of processes = %d\n", n_procs);
    dest    = 1;     /* Send to the other process */
    message = 1;     /* Just send one int         */

    /* Send message to slave */
    MPI_Send(&message, length, MPI_INT, dest,
             tag, MPI_COMM_WORLD);
    printf("After MPI_Send\n");

    source = 1;
    /* Receive message from slave. length is how much
       room we have and NOT the length of the message */
    MPI_Recv(&message, length, MPI_INT, source, tag,
             MPI_COMM_WORLD, &status);

    printf("After MPI_Recv, message = %d\n", message);
```



```
  } else {    /* I'm the slave process */

    source = 0;
    /* Receive message from master */
    MPI_Recv(&message, length, MPI_INT, source, tag,
             MPI_COMM_WORLD, &status);

    dest = 0;   /* Send to the other process */
    message++;  /* Increase message */

    /* Send message to master */
    MPI_Send(&message, length, MPI_INT, dest,
             tag, MPI_COMM_WORLD);
  }

  MPI_Finalize();    /* Shut down MPI */
  return 0;
}
```

To run: read the MPI-assignment. Something like:

```
% lamboot bhost
  ...
% mpicc simple.c
% mpirun c0-1 a.out
Number of processes = 2
After MPI_Send
After MPI_Recv, message = 2
% lamhalt    when we are finished for the day
```

One can print in the slave as well, but it may not work in all MPI-implementations and the order of the output is not deterministic. It may be interleaved or buffered.

We may not be able to start processes from inside the program (permitted in MPI 2.0 but may not be implemented).

Let us look at each call in some detail: Almost all the MPI-routines in C are integer functions returning a status value. I have ignored these values in the example program. In Fortran there are subroutines instead. The status value is returned as an extra integer parameter (the last one).

Start and stop MPI (it is possible to do non-MPI stuff before Init and after Finalize). These routines must be called:

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

MPI_COMM_WORLD is a communicator, a group of processes. The program can find out the number of processes by calling **MPI_Comm_size** (note that **&** is necessary since we require a return value).

```
MPI_Comm_size(MPI_COMM_WORLD, &n_procs);
```

Each process is numbered from **0** to **n_procs-1**. To find the number (rank) we can use **MPI_Comm_rank**.

```
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

We need the rank when sending messages and to decide how the work should be shared:

```
if ( my_rank == 0 ) {
  I'm the master
} elseif ( my_rank == 1 ) {
 ...
```


The two most basic communication routines (there are many) are:

```
MPI_Send(&message, length, MPI_INT, dest, tag,
         MPI_COMM_WORLD);

MPI_Recv(&message, length, MPI_INT, source, tag,
         MPI_COMM_WORLD, &status);
```

If the message is an array there should be no **&**.

Some other datatypes are **MPI_FLOAT** and **MPI_DOUBLE**.

The Fortran names are **MPI_INTEGER**, **MPI_REAL** and **MPI_DOUBLE_PRECISION**.

Note that **length** is the number of elements of the specific type (not the number of bytes).

length in **MPI_Send** is the number of elements we are sending (the **message**-array may be longer). **length** in **MPI_Recv** is amount of storage available to store the message.

If this value is less than the length of the message we get:

```
After MPI_SendMPI_Recv: message truncated
  (rank 1, MPI_COMM_WORLD)

One of the processes started by mpirun has exited with
a nonzero exit code.  ...
```

dest is the rank of the receiving process. **tag** is a number of the message that the programmer can use to keep track of messages ($0 \leq$ tag $\leq$ at least 32767).

The same holds for **MPI_Recv**, with the difference that **source** is the rank of the sender.

If we will accept a message from any sender we can use the constant (from the header file) **MPI_ANY_SOURCE**.

If we accept any tag we can use **MPI_ANY_TAG**.
So, we can use **tag** and **source** to pick a specific message from a queue of messages.

status is a so called structure (a record) consisting of at least three members (**MPI_SOURCE**, **MPI_TAG** and **MPI_ERROR** (some systems may have additional members).

We can do the following:

```
printf("status.MPI_SOURCE = %d\n", status.MPI_SOURCE);
printf("status.MPI_TAG    = %d\n", status.MPI_TAG);
printf("status.MPI_ERROR  = %d\n", status.MPI_ERROR);
```

To find out the actual length of the message we can do:

```
MPI_Get_count(&status, MPI_INT, &size);
printf("size = %d\n", size);
```

Here comes the simple program in Fortran.


```
program simple
  implicit    none
  include    "mpif.h"
  integer     message, length, source, dest, tag
  integer     my_rank, err
  integer     n_procs    ! number of processes
  integer     status(MPI_STATUS_SIZE)


  call MPI_Init(err)  ! Start up MPI

! Find out the number of n_processes and my rank
  call MPI_Comm_rank(MPI_COMM_WORLD, my_rank, err)
  call MPI_Comm_size(MPI_COMM_WORLD, n_procs, err)

  tag    = 1
  length = 1   ! Length of message

  if ( my_rank == 0 ) then ! I'm the master process
    print*, "Number of processes = ", n_procs
    dest    = 1  ! Send to the other process
    message = 1  ! Just send one integer

!   Send message to slave
    call MPI_Send(message, length, MPI_INTEGER, dest, &
                  tag, MPI_COMM_WORLD, err)
    print*, "After MPI_Send"

    source = 1
!   Receive message from slave
    call MPI_Recv(message, length, MPI_INTEGER, source,&
                  tag, MPI_COMM_WORLD, status, err)

    print*, "After MPI_Recv, message = ", message
```



```
  else  ! I'm the slave process
    source = 0
!   Receive message from master
    call MPI_Recv(message, length, MPI_INTEGER, source,&
                  tag, MPI_COMM_WORLD, status, err)

    dest    = 0            ! Send to the other process
    message = message + 1  ! Increase message
!   Send message to master
    call MPI_Send(message, length, MPI_INTEGER, dest, &
                  tag, MPI_COMM_WORLD, err)
  end if

  call MPI_Finalize(err)  ! Shut down MPI

end program simple
```

Note that the Fortran-routines are subroutines (not functions) and that they have an extra parameter, **err**.

One problem in Fortran77 is that **status**, in **MPI_Recv**, is a structure. The solution is: **status(MPI_SOURCE)**, **status(MPI_TAG)** and **status(MPI_ERROR)** contain, respectively, the source, tag and error code of the received message.

To compile and run (one can add **-O3** etc.):

```
mpif77 simple.f90   I have not made any mpif90
mpirun c0-1 a.out
```

^C usually kills all the processes.

There are blocking and nonblocking point-to-point Send/Receive-routines in MPI. The communication can be done in different modes (buffered, synchronised, and a few more). The Send/Receive we have used are blocking, but we do not really know if they are buffered or not (the standard leaves this open). This is a very important question. Consider the following code:

```
...
  integer, parameter        :: MASTER = 0, SLAVE = 1
  integer, parameter        :: N_MAX = 10000
  integer, dimension(N_MAX) :: vec = 1

  call MPI_Init(err)
  call MPI_Comm_rank(MPI_COMM_WORLD, my_rank, err)
  call MPI_Comm_size(MPI_COMM_WORLD, n_procs, err)

  msg_len = N_MAX;   buf_len = N_MAX

  if ( my_rank == MASTER ) then
    send_to = SLAVE; tag = 1
    call MPI_Send(vec, msg_len, MPI_INTEGER,   &
                  send_to, tag, MPI_COMM_WORLD, err)

    recv_from = SLAVE; tag = 2
    call MPI_Recv(vec, buf_len, MPI_INTEGER, &
                  recv_from,  tag,           &
                  MPI_COMM_WORLD, status, err)
  else
    send_to = MASTER; tag = 2
    call MPI_Send(vec, msg_len, MPI_INTEGER,   &
                  send_to, tag, MPI_COMM_WORLD, err)

    recv_from = MASTER; tag = 1
    call MPI_Recv(vec, buf_len, MPI_INTEGER, &
                  recv_from,  tag,           &
                  MPI_COMM_WORLD, status, err)
  end if
...
```

This code works (under LAM) when **N_MAX = 1000**, but it hangs, it deadlocks, when **N_MAX = 10000**. One can suspect that buffering is used for short messages but not for long ones. This is usually the case in all MPI-implementations. Since the buffer size is not standardized we cannot rely on buffering though.

There are several ways to fix the problem. One is to let the master node do a Send followed by the Receive. The slave does the opposite, a Receive followed by the Send.

```
        master                  slave
    call MPI_Send(...)      call MPI_Recv(...)
    call MPI_Recv(...)      call MPI_Send(...)
```

Another way is to use the deadlock-free **MPI_Sendrecv**-routine. As it says in the LAM man-page: "This function is guaranteed not to deadlock in situations where pairs of blocking sends and receives may deadlock."

The code in the example can then be written:

```
program dead_lock
  include "mpif.h"

  integer :: rec_from, snd_to, snd_tag, rec_tag, &
             my_rank, err, n_procs, snd_len, buf_len
  integer, dimension(MPI_STATUS_SIZE) :: status

  integer, parameter        :: MASTER = 0, SLAVE = 1
  integer, parameter        :: N_MAX = 100
  integer, dimension(N_MAX) :: snd_buf, rec_buf

  call MPI_Init(err)
  call MPI_Comm_rank(MPI_COMM_WORLD, my_rank, err)
  call MPI_Comm_size(MPI_COMM_WORLD, n_procs, err)

  snd_len = N_MAX;    buf_len = N_MAX
```



```
  if ( my_rank == MASTER ) then
    snd_buf  = 10  ! init the array
    snd_to   = SLAVE;  snd_tag = 1
    rec_from = SLAVE;  rec_tag = 2
    call MPI_Sendrecv(snd_buf, snd_len, MPI_INTEGER, &
                 snd_to, snd_tag, rec_buf, buf_len,  &
                 MPI_INTEGER, rec_from, rec_tag,     &
                 MPI_COMM_WORLD, status, err)
    print*, 'master, rec_buf(1:5) = ', rec_buf(1:5)
  else
    snd_buf  = 20  ! init the array
    snd_to   = MASTER;  snd_tag = 2
    rec_from = MASTER;  rec_tag = 1

    call MPI_Sendrecv(snd_buf, snd_len, MPI_INTEGER, &
                 snd_to, snd_tag, rec_buf, buf_len,  &
                 MPI_INTEGER, rec_from, rec_tag,     &
                 MPI_COMM_WORLD, status, err)
    print*, 'slave,  rec_buf(1:5) = ', rec_buf(1:5)
  end if

  call MPI_Finalize(err)

end program dead_lock
% mpirun c0-1 ./a.out
 master, rec_buf(1:5) =  20 20 20 20 20
 slave,  rec_buf(1:5) =  10 10 10 10 10
```

Another situation where we get a deadlock is when a send is missing:

```
        master                  slave
         ...                call MPI_Recv(...)
```

A blocking receive will wait forever (until we kill the processes).

Sending messages to many processes

There are broadcast operations in MPI, where one process can send to all the others.

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{
  int             message[10], length, root, my_rank;
  int             n_procs, j;

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

  length = 10;
  root   = 2;  /* Note: the same for all.   */
               /* Need not be 2, of course. */
  if (my_rank == 2) {
    for (j = 0; j < length; j++)
           message[j] = j;

    /* Here is the broadcast. Note, no tag. */
    MPI_Bcast(message, length, MPI_INT, root,
              MPI_COMM_WORLD);
  } else {
    /* The slaves have exactly the same call */
    MPI_Bcast(message, length, MPI_INT, root,
              MPI_COMM_WORLD);

    printf("%d: message[0..2] = %d %d %d\n",
           my_rank, message[0], message[1],
           message[2]);
  }
  MPI_Finalize();
  return 0;
}
```



```
% mpirun c0-3 a.out
0: message[0..2] = 0 1 2
1: message[0..2] = 0 1 2
3: message[0..2] = 0 1 2
```

Why should we use a broadcast instead of several **MPI_Send**? The answer is that it may be possible to implement the broadcast in a more efficient manner:

```
timestep 0:    0 -> 1   (-> means send to)

timestep 1:    0 -> 2, 1 -> 3

timestep 2:    0 -> 4, 1 -> 5, 2 -> 6, 3 -> 7

etc.
```

So, provided we have a network topology that supports parallel sends we can decrease the number of send-steps significantly. In lam this is used if **n_procs** is greater than four. Otherwise a linear algorithm is used.

There are other global communication routines.

Let us compute an integral by dividing the interval in $\#p$ pieces:

$$\int_a^b f(x)dx = \int_a^{a+h} f(x)dx + \int_{a+h}^{a+2h} f(x)dx + \cdots + \int_{a+(\#p-1)h}^{b} f(x)dx$$

where $h = \frac{b-a}{\#p}$.

Each process computes its own part, and the master has to add all the parts together. Adding parts together this way is called a reduction.

We will use the trapezoidal method (we would not use that in a real application).

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

/* Note */
#define MASTER 0

/* Prototypes */
double trapez(double, double, int);
double f(double);

int main(int argc, char *argv[])
{
  int     n_procs, my_rank, msg_len;
  double  a, b, interval, I, my_int, message[2];

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



```
  if (my_rank == MASTER) {
    a = 0.0; b = 4.0; /* or read some values */

    /* compute the length of the subinterval */
    interval   = (b - a) / n_procs;
    message[0] = a;         /* left endpoint */
    message[1] = interval;

    msg_len = 2;
    MPI_Bcast(message, msg_len, MPI_DOUBLE, MASTER,
              MPI_COMM_WORLD);

    /* compute my part of the integral */
    my_int = trapez(a, a + interval, 100);
    /* my_int is the MASTER's part of the integral.
       All parts are accumulated in I, but only in
       the master process.
    */

    msg_len = 1;
    MPI_Reduce(&my_int, &I, msg_len, MPI_DOUBLE,
               MPI_SUM, MASTER, MPI_COMM_WORLD);

    printf("The integral = %e\n", I);
  } else {  /* I'm a slave */
    msg_len = 2;
    MPI_Bcast(message, msg_len, MPI_DOUBLE, MASTER,
              MPI_COMM_WORLD);

    /* unpack the message   */
    a        = message[0];
    interval = message[1];

    /* compute my endpoints */
    a = a + my_rank * interval;
    b = a + interval;
```



```
    /* approximate the integral */
    my_int = trapez(a, b, 100);

    msg_len = 1;
    MPI_Reduce(&my_int, &I, msg_len, MPI_DOUBLE,
               MPI_SUM, MASTER, MPI_COMM_WORLD);
  }

  MPI_Finalize();
  return 0;
}
double f(double x)  /* The integrand */
{
  return exp(-x * cos(x));
}

/* An extremely primitive quadrature method.
   Approximate integral from a to b of f(x) dx.
   We integrate over [a, b] which is different
   from the [a, b] in the main program.
*/

double trapez(double a, double b, int n)
{
  int       k;
  double I, h;

  h = (b - a) / n;

  I = 0.5 * (f(a) + f(b));
  for(k = 1; k < n; k++) {
    a += h;
    I += f(a);
  }

  return h * I;
}
```


To get good speedup the function should require a huge amount of cputime to evaluate.

There are several operators (not only **MPI_SUM**) that can be used together with **MPI_Reduce**.

MPI_MAX	return the maximum
MPI_MIN	return the minimum
MPI_SUM	return the sum
MPI_PROD	return the product
MPI_LAND	return the logical and
MPI_BAND	return the bitwise and
MPI_LOR	return the logical or
MPI_BOR	return the bitwise of
MPI_LXOR	return the logical exclusive or
MPI_BXOR	return the bitwise exclusive or
MPI_MINLOC	return the minimum and the location (actually, the value of the second element of the structure where the minimum of the first is found)
MPI_MAXLOC	return the maximum and the location

If all the processes need the result (**I**) we could do a broadcast afterwards, but there is a more efficient routine, **MPI_Allreduce**. See the web for details (under **Documentation**, **MPI-routines**).

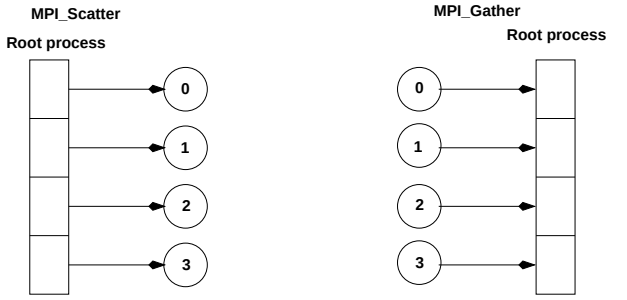
The **MPI_Allreduce** may be performed in an efficient way. Suppose we have eight processes, **0**, ..., **7**. **|** denotes a split.

```
          0 1 2 3 | 4 5 6 7         0<->4, 1<->5 etc
   0 1 | 2 3             4 5 | 6 7        0<->2 etc
0 | 1      2 | 3      4 | 5      6 | 7    0<->1 etc
```

Each process accumulates its own sum (and sends it on):

```
s0 = x[0] + x[4], s2 = x[2] + x[6], ...
s0 = s0 + s2 = (x[0] + x[4] + x[2] + x[6])
s0 = s0 + s1 = x[0] + ... + x[7]
```

A common operation is to gather, **MPI_Gather** (bring to one process) sets of data. **MPI_Scatter** is the reverse of gather, it distributes pieces of a vector. See the manual for both of these.



There is also an **MPI_Allgather** that gathers pieces to a long vector (as gather) but where each process gets a copy of the long vector. Another "All"-routine is **MPI_Allreduce** as we just saw.

Nonblocking communication - a small example

Suppose we have a pool of tasks where the amount of time to complete a task is unpredictable and varies between tasks.
We want to write an MPI-program, where each process will ask the master-process for a task, complete it, and then go back and ask for more work. Let us also assume that the tasks can be finished in any order, and that the task can be defined by a single integer and the result is an integer as well (to simplify the coding).
The master will perform other work, interfacing with the user, doing some computation etc. while waiting for the tasks to be finished.
We could divide all the tasks between the processes at the beginning, but that may lead to load inbalance.
An alternative to the solution, on the next page, is to create two threads in the master process. One thread handles the communication with the slaves and the other thread takes care of the user interface.
One has to very careful when mixing threads and MPI, since the MPI-system may not be thread safe, or not completely thread safe. The MPI-2.0 standard defines the following four levels:

- **MPI_THREAD_SINGLE**: Only one thread will execute.
- **MPI_THREAD_FUNNELED**: The process may be multi-threaded, but only the main thread will make MPI calls (all MPI calls are "funneled" to the main thread).
- **MPI_THREAD_SERIALIZED**: The process may be multi-threaded, and multiple threads may make MPI calls, but only one at a time: MPI calls are not made concurrently from two distinct threads (all MPI calls are "serialized").
- **MPI_THREAD_MULTIPLE**: Multiple threads may call MPI, with no restrictions.

See the standard for more details.

The master does the following:

```
set_of_tasks = { task_id:s }

Send a task_id to each slave and remove
these task_id:s from set_of_tasks

while ( not all results have been received ) {
  while ( no slave has reported a result )  // NB
    do some, but not too much, work

  if ( tasks remaining ) {
    pick a task_id from the set_of_tasks and
    remove it from the set_of_tasks
    send task_id to the slave
    (i.e. to the slave that reported the result)
  } else
    send task_id = QUIT to slave
}
```

Here is the slave code:

```
dont_stop = 1  /* continue is a keyword in C */
while ( dont_stop ) {
  wait for task_id from master
  dont_stop = task_id != QUIT

  if ( dont_stop ) {
    work on the task
    send result to master
  }
}
```

The nonblocking communication is used in the **while**-loop marked **NB**. If the master is doing too much work in the loop, in may delay the slaves.

Details about nonblocking communication

A nonblocking send start call initiates the send operation, but does not complete it. The send start call will return before the message was copied out of the send buffer. A separate send complete call is needed to complete the communication, i.e., to verify that the data has been copied out of the send buffer.

Similarly, a nonblocking receive start call initiates the receive operation, but does not complete it. The call will return before a message is stored into the receive buffer. A separate receive complete call is needed to complete the receive operation and verify that the data has been received into the receive buffer.

This is where the master can do some work in parallel with the wait. Using a blocking receive the master could not work in parallel.

If the send mode is standard then the send-complete call may return before a matching receive occurred, if the message is buffered. On the other hand, the send-complete may not complete until a matching receive occurred, and the message was copied into the receive buffer.

Nonblocking sends can be matched with blocking receives, and vice-versa.

Here is comes a nonblocking send:

```
MPI_Request request;

MPI_Isend(&message, msg_len, MPI_INT, rank, tag,
          MPI_COMM_WORLD, &request);
```

It looks very much like a blocking send, the only differences are the name **MPI_Isend** (**I** stands for an almost immediate return), and the extra parameter, **request**. The variable is a handle to a so-called opaque object.

Think of ths communication object as being a C-structure with variables keeping track of the tag and destination etc. **request** is used to identify communication operations and match the operation that initiates the communication with the operation that terminates it. We are not supposed to access the information in the object, and its contents is not standardised.

A nonblocking receive may look like:

```
MPI_Request request;

MPI_Irecv(&message, msg_len, MPI_INT, rank,
          tag, MPI_COMM_WORLD, &request);
```

Here are some functions for completing a call:

```
MPI_Request request, requests[count];
MPI_Status  status;

MPI_Wait(&request, &status);
MPI_Test(&request, &flag, &status);
MPI_Testany(count, requests, &index, &flag, &status);
```

and here is a simplified description. **request** is a handle to a communication object, referred to as object.

MPI_Wait returns when the operation identified by **request** is complete. So it is like a blocking wait. If the object was created by a nonblocking send or receive call, then the object is deallocated and **request** is set to **MPI_REQUEST_NULL**.

MPI_Test returns **flag = true** if the operation identified by **request** is complete. In such a case, **status** contains information on the completed operation; if the object was created by a nonblocking send or receive, then it is deallocated and **request** is set to **MPI_REQUEST_NULL**.
The call returns **flag = false**, otherwise. In this case, the value of **status** is undefined.

Finally **MPI_Testany**. If the array of requests contains active handles then the execution of **MPI_Testany** has the same effect as the execution of

```
MPI_Test( &requests[i], flag, status),
  for i=0, 1 ,..., count-1,
```

in some arbitrary order, until one call returns **flag = true**, or all fail. In the former case, **index** is set to the last value of **i**, and in the latter case, it is set to **MPI_UNDEFINED**.

If **request** (or **requests**) does not correspond to an ongoing operation, the routines return immediately.

Now it is time for the example. We have **n_slaves** numbered from 0 up to **n_procs - 2**. The master has rank **n_procs - 1**. The number of tasks are **n_tasks** and we assume that the number of slaves is not greater than the number of tasks. **task_ids** is an array containing a non-negative integer identifying the task. A task id of **QUIT = -1** tells the slave to finish.

The computed results (integers) are returned in the array **results**.

next_task points to the next task in **task_ids** and **n_received** keeps track of how many tasks have been finished by the slaves.

Here comes the code. First the master-routine.


```
void master_proc(int n_procs, int n_slaves, int n_tasks,
                 int task_ids[], int results[])
{
  const int max_slaves = 10, tag = 1, msg_len = 1;
  int hit, message, n_received, slave, next_task, flag;
  double d;
  MPI_Request requests[max_slaves];
  MPI_Status status;

  next_task = n_received = 0;

  /* Initial distribution of tasks */
  for (slave = 0; slave < n_slaves; slave++) {
    MPI_Send(&task_ids[next_task], msg_len, MPI_INT,
             slave, tag, MPI_COMM_WORLD);

    /* Start a nonblocking receive */
    MPI_Irecv(&results[next_task], msg_len, MPI_INT,
              MPI_ANY_SOURCE, MPI_ANY_TAG,
              MPI_COMM_WORLD, &requests[slave]);
    next_task++;
  }

  /* Wait for all results to come in ... */
  while (n_received < n_tasks) {
    flag = 0;
    while (!flag) {
      /* Complete the receive */
      MPI_Testany(n_slaves, requests, &hit, &flag,
                  &status);
      d = master_work();  /* Do some work */
    }
```



```
    n_received++;               /* Got one result */
    slave = status.MPI_SOURCE;  /* from where?    */

    /* Hand out a new task to the slave,
       unless we are done
     */
    if (next_task < n_tasks) {
      MPI_Send(&task_ids[next_task], msg_len, MPI_INT,
               slave, tag, MPI_COMM_WORLD);

      MPI_Irecv(&results[next_task], msg_len, MPI_INT,
                MPI_ANY_SOURCE, MPI_ANY_TAG,
                MPI_COMM_WORLD, &requests[hit]);
      next_task++;
    } else {  /* No more tasks */
      message = QUIT;
      MPI_Send(&message, msg_len, MPI_INT, slave, tag,
               MPI_COMM_WORLD);
    }
  }
}
```

and then the code for the slaves

```
void slave_proc(int my_rank, int master)
{
  const int msg_len = 1, tag = 1;
  int message, result, dont_stop;
  MPI_Status status;

  dont_stop = 1;
  while (dont_stop) {
    MPI_Recv(&message, msg_len, MPI_INT, master,
             MPI_ANY_TAG, MPI_COMM_WORLD, &status);

    dont_stop = message != QUIT;
    if (dont_stop) {
      /* Simulate work */
      result = 100 * message + my_rank;
      sleep(message);

      MPI_Send(&result, msg_len, MPI_INT, master,
               tag, MPI_COMM_WORLD);
    }
  }
}
```


Suppose we are using three slaves and have ten tasks, the **task_ids**-array takes indices from zero to nine.

The work is simulated by using the **sleep**-function and the ten tasks correspond to sleeping 1, 2, 3, 1, 2, 3, 1, 2, 3, 1 seconds. The work done by the master, in **master_work**, takes 0.12 s per call.

The table below shows the results from one run.
When a number is repeated two times the slave worked with this task for two seconds (similarly for a repetition of three).

	slaves			task number	sleep time
	0	1	2	0	1
time				1	2
1	0	1	2	2	3
2	3	1	2	3	1
4	5	4	2	4	2
4	5	4	6	5	3
5	5	7	8	6	1
6	9	7	8	7	2
7			8	8	3
				9	1

So had it been optimal, the run should have taken 7 s wallclock time (the sum of the times is 19, so it must take more than 6 s wallclock time, as $3 \cdot 6 < 19$. The optimal time must be an integer, and the next is 7). The time needed was 7.5 s and the master was essentially working all this time as well.

Using two slaves the optimal time is 10 s, and the run took 10.8 s.

A page about distributed Gaussian elimination

In standard GE we take linear combinations of rows to zero elements in the pivot columns. We end up with a triangular matrix.

How should we distribute the matrix if we are using MPI?

The obvious way is to partition the rows exactly as in our power method (a row distribution). This leads to poor load balancing, since as soon as the first block has been triangularized processor 0 will be idle.

After two elimination steps we have the picture (**x** is nonzero and the block size is 2):

```
x x x x x x x x   proc 0
0 x x x x x x x   proc 0
0 0 x x x x x x   proc 1
0 0 x x x x x x   proc 1
0 0 x x x x x x   proc 2
0 0 x x x x x x   proc 2
0 0 x x x x x x   proc 3
0 0 x x x x x x   proc 3
```

Another alternative is to use a cyclic row distribution. Suppose we have four processors, then processor 0 stores rows 1, 5, 9, 13, ... Processor 2 stores rows 2, 6, 10 etc. This leads to a good balance, but makes it impossible to use BLAS2 and 3 routines (since it is vector oriented).

There are a few other distributions to consider, but we skip the details since they require a more thorough knowledge about algorithms for GE.

One word about Scalapack

ScaLAPACK (Scalable Linear Algebra PACKage) is a distributed and parallel version of Lapack. ScaLAPACK uses BLAS on one processor and distributed-memory forms of BLAS on several (PBLAS, Parallel BLAS and BLACS, C for Communication). BLACS uses PVM or MPI.

Scalapack uses a block cyclic distribution of (dense) matrices. Suppose we have processors numbered **0**, **1**, **2** and **3** and a block size of **32**. This figure shows a matrix of order **8 · 32**.

```
0 1 0 1 0 1 0 1
2 3 2 3 2 3 2 3
0 1 0 1 0 1 0 1
2 3 2 3 2 3 2 3
0 1 0 1 0 1 0 1
2 3 2 3 2 3 2 3
0 1 0 1 0 1 0 1
2 3 2 3 2 3 2 3
```

It turns out that this layout gives a good opportunity for parallelism, good load balancing and the possibility to use BLAS2 and BLAS3.

Doing a Cholesky factorization on the Sun using MPI:

```
n          = 4000
block size = 32

#CPUs      = 4
time       = 27.5
rate       = 765 Mflops
```

The uniprocessor Lapack routine takes **145**s.

Some other things MPI can do

- Suppose you would like to send an int, a double array, and int array etc. in the same message. One way is to pack things into the message yourself. Another way is to use **MPI_Pack**/**MPI_Unpack** or (more complicated) to create a new MPI datatype (almost like a C-structure).
- It is possible to divide the processes into subgroups and make a broadcast (for example) in this group.
- You can create virtual topologies in MPI, e.g. you can map the processors to a rectangular grid, and then address the processors with row- and column-indices.
- There is some support for measuring performance.
- It is possible to control how a message is passed from one process to another. Do the processes synchronise or is a buffer used, for example.
- There are more routines for collective communication.

In MPI-2.0 there are several new features, some of these are:

- Dynamic process creation.
- One-sided communication, a process can directly access memory of another process (similar to shared memory model).
- Parallel I/O, allows several processes to access a file in a co-ordinated way.

Matlab and parallel computing

Two major options.

1. Threads & shared memory by using the parallel capabilities of the underlying numerical libraries.
2. Message passing by using the "Distributed Computing Toolbox" (a large toolbox, the User's Guide is 529 pages).

Threads can be switched on in two ways. From the GUI: Preferences/General/Multithreading or by using **maxNumCompThreads**. Here is a small example:

```
T = [];
for thr = 1:4
  maxNumCompThreads(thr);  % set #threads
  j = 1;
  for n = [100 200 400 800 1600 3200]
    A = randn(n);
    B = randn(n);
    t = clock;
      C = A * B;
    T(thr, j) = etime(clock, t);
    j = j + 1;
  end
end
```

The speedup depends on the library. This is how yo can find out:

```
% setenv LAPACK_VERBOSITY 1
cpu_id: x86 Family 15 Model 1 Stepping 0, AuthenticAMD
  etc.
libmwblas: loading acml.so
libmwblas: resolved caxpy_ in 0x524c10
libmwblas: resolved ccopy_ in 0x524c10
etc.
```

So ACML is used. Another, slower alternative is using MKL (see the gui-help for **BLAS_VERSION**).

We tested solving linear systems and computing eigenvalues as well. Here are the times using one to four threads:

```
C = A * B
  100      200      400      800     1600     3200
1.2e-02  4.9e-03  3.7e-02  2.8e-01  2.1e+00  1.7e+01
2.4e-02  2.8e-03  2.1e-02  1.5e-01  1.1e+00  8.5e+00
1.2e-02  2.0e-03  1.6e-02  1.1e-01  8.2e-01  6.0e+00
1.3e-02  2.8e-03  2.1e-02  8.7e-02  6.1e-01  4.6e+00

x = A \ b, b a vector
  100      200      400      800     1600     3200
2.7e-03  4.1e-03  3.5e-02  1.9e-01  1.1e+00  7.9e+00
1.4e-03  4.1e-03  2.8e-02  1.4e-01  7.4e-01  4.8e+00
2.9e-03  9.8e-03  2.4e-02  1.2e-01  6.0e-01  4.0e+00
1.8e-03  5.6e-03  2.6e-02  1.1e-01  5.4e-01  3.5e+00

l = eig(A)
  100      200      400      800     1600     3200
1.5e-02  8.6e-02  4.7e-01  3.3e+00  2.0e+01  1.2e+02
1.5e-02  9.3e-02  4.2e-01  2.5e+00  1.3e+01  8.7e+01
1.5e-02  9.4e-02  4.1e-01  2.4e+00  1.2e+01  8.1e+01
2.0e-02  9.4e-02  3.9e-01  2.3e+00  1.2e+01  8.0e+01
```

So, using several threads can be an option if we have a large problem. We get a better speedup for the multiplication, than for **eig**, which seems reasonable.

This method can be used to speed up the computation of elementary functions as well.

Now to a simple example using the Toolbox.
The programs computes the eigenvalues of $T + \rho E$ where T is a tridiagonal matrix, $E = e_n e_n^T$ and ρ is a real parameter.


```
j = 1;
T = [];
m = 100;
params = linspace(0, 1);
for n = [500 1000 2000 4000]
  T = spdiags(ones(n, 3), -1:1, n, n);  % create data
  E = sparse(n, n, 1);
  eigs_p = zeros(n, m);                 % preallocate

  t1 = clock;
    matlabpool open 4                   % 4 new Matlabs
    t2 = clock;

      parfor(k = 1:m)                   % parallel loop
        eigs_p(:, k) = eig(T + params(k) * E);
      end

    t2 = etime(clock, t2);
    matlabpool close                    % close
  t1 = etime(clock, t1);

% The same computation one one CPU
  eigs_s = zeros(n, m);
  t3 = clock;
    for k = 1:m
      eigs_s(:, k) = eig(T + params(k) * E);
    end
  t3 = etime(clock, t3);

  times(j, :) = [t1, t2, t3]            % save times
  j = j + 1;
end
```

Here are the times:

n	t1	t2	t3
500	10.58	1.63	1.00
1000	18.84	2.40	3.89
2000	16.77	5.38	15.07
4000	39.18	16.49	58.11

`t1 - t2` gives the overhead for starting the processes.

For large problems this can be useful. The toolbox can handle more complex problems, see the User's Guide for details.

OpenMP - shared memory parallelism

OpenMP is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify shared memory parallelism in Fortran and C/C++ programs.

Fortran version 1.0, Oct 1997, ver. 2.0 Nov. 2000.
C/C++ ver. 1.0 Oct. 1998, ver. 2.0 Mar. 2002.

Version 2.5 May 2005, combines the Fortran and C/C++ specifications into a single one and fixes inconsistencies.

Specifications (in PDF): `www.openmp.org`
Good readability to be standards.

From www: The public discussion period for the draft OpenMP 3.0 specifications closed in January. The draft is now under final review by the Architecture Review Board (ARB), with a final vote due in a few weeks. Stay tuned. Posted on April 29, 2008

Books:

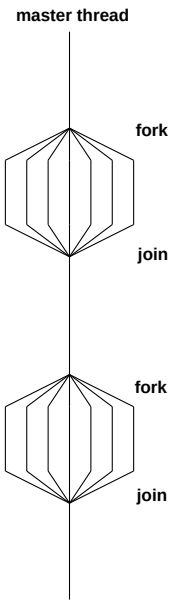
Parallel Programming in OpenMP,
R Chandra, D Kohr, R Menon, L Dagum, D Maydan,
J McDonald.
Morgan Kaufmann, 2000. 231 pages.

Parallel Programming in C with MPI and OpenMP,
M J Quinn.
McGraw-Hill Education, 2003. 544 pages.

Patterns for Parallel Programming,
T Mattson, B Sanders, B Massingill.
Addison Wesley Professional, 2004, 384 pages.

Basic idea - fork-join programming model

```
program test

... serial code ...

!$OMP parallel shared(A, n)

... code run i parallel ...

!$OMP end parallel


... serial code ...

!$OMP parallel do shared(b) private(x)

... code run i parallel ...

!$OMP end parallel do


... serial code ...
```



- when reaching a parallel part the master thread (original process) creates a team of threads and it becomes the master of the team
- the team execute concurrently on different parts of the loop (parallel construct)
- upon completion of the parallel construct, the threads in the team synchronise at an implicit barrier, and only the master thread continues execution
- the number of threads in the team is controlled by environment variables and/or library calls, e.g.
  `setenv OMP_NUM_THREADS 7`
  `call omp_set_num_threads(5)`
- the code executed by a thread must not depend on the result produced by a different thread

So what is a thread?

A thread originates from a process and is a part of that process. The threads (belonging to the particular process) share global variables, files, code, PID etc. but they have their individual stacks and program counters.

Note that we have several processes in MPI.

Since all the threads can access the shared data (a matrix say) it is easy to write code so that threads can work on different parts of the matrix in parallel.

It is possible to use threads directly but we will use the OpenMP-directives. The directives are analysed by a compiler or preprocessor which produces the threaded code.

MPI versus OpenMP

Parallelising using distributed memory (MPI):

- Requires large grain parallelism to be efficient (process based).
- Large rewrites of the code often necessary
  difficult with "dusty decks".
  May end up with parallel and non-parallel versions.
- Domain decomposition; indexing relative to the blocks.
- Requires global understanding of the code.
- Hard to debug.
- Runs on most types of computers.

Using shared memory (OpenMP)

- Can utilise parallelism on loop level (thread based).
  Harder on subroutine level, resembles MPI-programming.
- Minor changes to the code necessary. A detailed knowledge of the code not necessary. Only one version.
  Can parallelise using simple directives in the code.
- No partitioning of the data.
- Less hard to debug.
- Not so portable; requires a shared memory computer.
- Less control over the "hidden" message passing and memory allocation.

A simple example

Not all compilers support the full 2.5-standard and compiler flags may differ. In order for of program of this type to be efficient n must be fairly large.

```
 1 program example
 2   use omp_lib  ! or include "omp_lib.h"
 3                ! or something non-standard
 4   implicit none
 5   integer                          :: i, i_am
 6   integer, parameter               :: n = 10000
 7   double precision, dimension(n) :: a, b, c
 8
 9   c = 1.242d0 ! can be used inside the loop
10 !$omp parallel do private(i), shared(a, b, c)
11   do i = 1, n
12     b(i) = 0.5d0 * i
13     a(i) = 1.23d0 * b(i) + 3.45d0 * c(i)
14   end do
15 !$omp end parallel do   ! not necessary
16
17   print*, a(1), a(n)    ! only the master
18
19 !$omp parallel private(i_am)  ! a parallel region
20   i_am = omp_get_thread_num() ! 0, ..., #threads - 1
21   print*, 'i_am = ', i_am
22
23   !$omp master
24     print*, 'num threads = ', omp_get_num_threads()
25     print*, 'max threads = ', omp_get_max_threads()
26     print*, 'max cpus    = ', omp_get_num_procs()
27   !$omp end master
28
29 !$omp end parallel
30
31 end program example
```


10: A parallel do-loop. **!$omp** or **!$OMP**. See the standard for Fortran77.

Use **shared** when:

- a variable is not modified in the loop or
- when it is an array in which each iteration of the loop accesses a different element

All variables except the loop-iteration variable are **shared** by default. To turn off the default, use **default(none)**.

Suppose we are using four threads. The first thread may work on the first 2500 iterations (**n = 10000**), the next thread on the next group of 2500 iterations etc.

15: Not necessary, the **end do** on line 14 is sufficient. When the threads join at the **end do** they synchronise, there is an implicit barrier.

19-29: A parallel region. The code in the region is run in parallel.

20: **i_am** will be the number of the current thread. Threads are numbered from zero to the number of threads minus one.

21: All threads will print. Output from several threads may be interleaved (you may need a special compiler).

23-27: To avoid multiple prints we ask the master thread (thread zero) to print. Number of executing threads, maximum number of threads that can be created (can be changed by setting **OMP_NUM_THREADS** or by calling **omp_set_num_threads**) and available number of processors (cpus).


```
% f90 -mp omp1.f90 -lmp     May need special library
% setenv OMP_NUM_THREADS 1
% a.out
 3.8801893470010604,  6153.2651893470011
 i_am =  0
 num threads =  1
 max threads =  1
 max cpus    =  8

% setenv OMP_NUM_THREADS 4
% a.out
 3.8801893470010604,  6153.2651893470011
 i_am =  0
 i_am =  2
 num threads =  4
 i_am =  3
 max threads =  4
 i_am =  1
 max cpus    =  8

% setenv OMP_NUM_THREADS 9
% a.out
Warning: MP_SET_NUMTHREADS greater than available cpus
(set to 9; cpus = 8)
 3.8801893470010604,  6153.2651893470011
 i_am =  2
 num threads =  9
 i_am =  8
 max threads =  9
  ...
```

Make no assumptions about the order of execution between threads. Output from several threads may be interleaved (you may need a special compiler).
On Itanium: **ifort -openmp ....**
New this year: **gcc -fopenmp ...**, **gfortran -fopenmp ...**

The same program in C

```
#include <stdio.h>
#include <omp.h>
#define _N 10000

int main()
{
  int             i, i_am;
  const int       n = _N;
  double          a[_N], b[_N], c[_N];

  for (i = 0; i < n; i++)
    c[i] = 1.242;

/* pragma omp instead of !$omp */
#pragma omp parallel for private(i) shared(a, b, c)
  for (i = 0; i < n; i++) {
    b[i] = 0.5 * (i + 1);
    a[i] = 1.23 * b[i] + 3.45 * c[i];
  }
  printf("%f, %f\n", a[0], a[n - 1]);

#pragma omp parallel private(i_am)
  {
    i_am = omp_get_thread_num();
    printf("i_am = %d\n", i_am);

#pragma omp master
    {
     printf("num threads = %d\n",omp_get_num_threads());
     printf("max threads = %d\n",omp_get_max_threads());
     printf("max cpus    = %d\n",omp_get_num_procs());
    }  /* Pairs of { } instead of end */
  }
  return 0;
}
```


Problems

Do not do like this:

```
program ex2
!$omp parallel do private(i), shared(a)
  do i = 1, 1000
    a = i
  end do

  print*, a

end program ex2
```

Will give you different values **1000.**, **875.** etc.

```
program ex3
  integer                 :: i
  integer, dimension(12) :: a, b

  a = 1  ! a vector of ones
  b = 2
!$omp parallel do private(i) shared(a, b)
  do i = 1, 11
    a(i + 1) = a(i) + b(i)
  end do

  print*, a

end program ex3

A few runs:
 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23  one thread
 1, 3, 5, 7, 9, 11, 13,  3,  5,  7,  9, 11  four
 1, 3, 5, 7, 9, 11, 13, 15, 17, 19,  3,  5  four
 1, 3, 5, 7, 9, 11, 13,  3,  5,  7,  3,  5  four
```


Why?

```
    thread          computation
      0          a(2)  = a(1) + b(1)
      0          a(3)  = a(2) + b(2)
      0          a(4)  = a(3) + b(3)    <--|
                                           |  Problem
      1          a(5)  = a(4) + b(4)    <--|
      1          a(6)  = a(5) + b(5)
      1          a(7)  = a(6) + b(6)    <--|
                                           |  Problem
      2          a(8)  = a(7) + b(7)    <--|
      2          a(9)  = a(8) + b(8)
      2          a(10) = a(9) + b(9)    <--|
                                           |  Problem
      3          a(11) = a(10) + b(10)  <--|
      3          a(12) = a(11) + b(11)
```

We have a data dependency between iterations, causing a so-called race condition.

Can "fix" the problem:

```
! You need ordered in both places

!$omp parallel do private(i) shared(a, b) ordered
  do i = 1, 11
    !$omp ordered
      a(i + 1) = a(i) + b(i)
    !$omp end ordered
  end do
```

but in this case the threads do not run in parallel.

Load balancing

```
!$omp parallel do private(k) shared(x, n) &
!$omp          schedule(static, 4)  ! 4 = chunk
  do k = 1, n
    ...
  end do
```

```
                            1                   2
k       : 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
thread 0: x x x x                 x x x x
thread 1:         x x x x                 x x x x
thread 2:                 x x x x
```

Default chunk, roughly = n / number_of_threads

Low overhead, good if the same amount of work in each iteration. **chunk** can be used to access array elements in groups (may be me more efficient, e.g. using cache memories in better way).

```
!$omp parallel do private(k) shared(x, n) &
!$omp          schedule(dynamic, 8)
 ...
```

Threads compete for **chunk**-sized assignments. Useful if the amount of work varies between iterations.

There is also **schedule(guided, chunk)** assigning pieces of work ($\geq$ **chunk**) proportional to the number of remaining iterations divided by the number of threads. It requires fewer synchronisations than **dynamic**.

Suppose we parallelize m iterations over P processors.
No default scheduling is defined in the OpenMP-standard,
but **`schedule(static, m / P)`** is a common choice
(assuming that P divides m).

Here comes an example where this strategy works badly.
So do not always use the standard choice.

We have nested loops, where the number of iterations in the inner loop depends on the loop index in the outer loop.

```
!$omp ...
do j = 1, m           ! parallelize this loop
  do k = j + 1, m     ! NOTE: k = j + 1
     call work(...)   ! each call takes the same time
  end do
end do
```

Suppose m is large and let T_{ser} be the total run time on one thread. If there is no overhead, the time, T_t, for thread number t is approximately:

$$T_t \approx \frac{2T_{ser}}{P}\ \left(1 - \frac{t+1/2}{P}\right), \quad t = 0, \ldots, P-1$$

So thread zero has much more work to do compared to the last thread:

$$\frac{T_0}{T_{P-1}} \approx 2P - 1$$

a very poor balance. The speedup is bounded by T_0:

$$\text{speedup} = \frac{T_{ser}}{T_0} \approx \frac{P}{2}$$

and not the optimal P.

We will come back to this example when we look at some case studies.

The reduction clause

Reducing a vector expression to a scalar is called a reduction.

```
program ex8
  integer                          :: i
  integer, parameter               :: n = 10000
  double precision, dimension(n) :: x = 1, y = 2
  double precision                 :: s

  s = 0.0d0
!$omp parallel do private(i) shared(x, y) &
!$omp reduction(+: s)
  do i = 1, n
    s = s + x(i) * y(i)
  end do

  print*, s

end program ex8
```

In general:
`reduction(operator or intrinsic:  variable list)`

Valid operators are: **`+, -, *, .and., .or., .eqv., .neqv.`** and intrinsics: **`max, min, iand, ior, ieor`** (the **`iand`** is bitwise and, etc.)

The operator/intrinsic must be used in one of the following ways:

- **`x = x operator expression`**
- **`x = expression operator x`** (except for subtraction)
- **`x = intrinsic(x, expression)`**
- **`x = intrinsic(expression, x)`**

where **`expression`** does not involve **`x`**.

This is what happens in our example above:

- each thread gets its local sum-variable, $s_{\#thread}$ say
- $s_{\#thread} = 0$ before the loop (the thread private variables are initialised in different ways depending on the operation, zero for **`+`** and **`-`**, one for **`*`**). See the standard for the other cases.
- each thread computes its sum in $s_{\#thread}$
- after the loop all the $s_{\#thread}$ are added to **`s`** in a safe way

We can implement our summation example without using **`reduction`**-variables. The problem is to update the shared sum in a safe way. This can be done using critical sections.


```
  ...

  double precision :: private_s, shared_s

  shared_s  = 0.0d0
!
! This is a more general parallel construct.
! Not only the do-loop is done in parallel.
!
!$omp parallel private(private_s) &
!$omp          shared(x, y, shared_s, n)
  private_s = 0.0d0

!$omp do private(i)      ! Here comes the loop:
  do i = 1, n
    private_s = private_s + x(i) * y(i)
  end do
!$omp end do
!
! Here we specify a critical section.
! Only one thread at a time may pass through.
!
!$omp critical
  shared_s = shared_s + private_s
!$omp end critical

!$omp end parallel

  print*, shared_s

  ...
```

Nested loops, matrix-vector multiply

```
a = 0.0
do j = 1, n
  do i = 1, m
    a(i) = a(i) + C(i, j) * b(j)
  end  do
end do
```

Can be parallelised with respect to **i** but not with respect to **j** (since different threads will write to the same **a(i)**).

May be inefficient since parallel execution is initiated **n** times (procedure calls). OK if **n** small and **m** large.

Switch loops.

```
a = 0.0
do i = 1, m
  do j = 1, n
    a(i) = a(i) + C(i, j) * b(j)
  end  do
end do
```

The **do i** can be parallelised. Bad cache locality for **C**.

Test on KALLSUP2 (Power3), using **-O3** (implies blocking). Times in seconds for one to four threads. **dgemv** takes 0.18s.

m	n	first loop				second loop			
		1	2	3	4	1	2	3	4
4000	4000	0.3	0.2	0.2	0.2	5.9	3.1	2.1	1.6
40000	400	0.3	0.2	0.1	0.1	3.5	1.8	1.2	0.9
400	40000	0.3	1.0	1.0	1.2	11.4	6.7	4.9	4.1

- Cache locality is important.
- If second loop necessary, OpenMP gives speedup.
- Large n gives slowdown in first loop.

Some other OpenMP directives

```
...
!$omp parallel shared(a, n)    ! a parallel region

        ... code run in parallel

!$omp single   ! only ONE thread will execute the code
        ... code
!$omp end single

!$omp barrier  ! wait for all the other threads
      ... code

!$omp do private(k)
        do ...
        end do
!$omp end do nowait  ! don't wait (to wait is default)

      do ... ! NOTE: all iterations run by all threads
      end do

!$omp sections
          ... code executed by one thread
!$omp   section
          ... code executed by another thread
!$omp   section
          ... code executed by yet another thread
!$omp end sections  ! implicit barrier

!$ Fortran statements ...  Included if we use OpenMP,
!$ but not otherwise (conditional compilation)

!$omp end parallel  ! end of the parallel section
...
```


Some, but not all, compilers support parallelization of Fortran90 array operations, e.g.

```
  ... code
! a, b and c are arrays

!$omp parallel shared(a, b, c)
!$omp   workshare
          a = 2.0 * cos(a) + 3.0 * sin(c)
!$omp   end workshare
!$omp end parallel
  ... code
```

or shorter

```
  ... code
!$omp parallel workshare
        a = 2.0 * cos(a) + 3.0 * sin(c)
!$omp end parallel workshare
  ... code
```

Here comes a first example of where we call a subroutine from a parallel region.


```
program example
  use omp_lib
  implicit none
  integer, dimension(0:3) :: a = 99
  integer                 :: i_am

!$omp parallel private(i_am) shared(a)
        i_am = omp_get_thread_num()
        call work(a, i_am)

!$omp   single
          print*, 'a = ', a
!$omp   end single

!$omp end parallel

end program example

subroutine work(a, i_am)
! Dummy arguments inherit the data-sharing
! attributes of the associated actual arguments.
!
  integer, dimension(0:3) :: a    ! becomes shared
  integer                 :: i_am ! becomes private

  print*, 'work', i_am
  a(i_am) = i_am

end subroutine work
```

```
% a.out
 work 1
 work 3
 a =  99,  1,  99,  3
 work 2
 work 0
```

Print after **!$omp end parallel** (or add a barrier):

```
!$omp barrier
!$omp single
      print*, 'a = ', a
!$omp end single

% a.out
 work 0
 work 1
 work 3
 work 2
 a =  0,  1,  2,  3
```

There are more things in the standard (directives, locking routines).

OpenMP makes no guarantee that input or output to the same file is synchronous when executed in parallel. You may need to link with a special thread safe I/O-library.

More on OpenMP and subprograms

A few examples:

• Calling a subroutine, containing OpenMP-directives, from a serial part of the program. Essentially what we have done so far.

• Suppose now that we have the following situation:

```
!$omp parallel ...
  ... code
        call a parallel subroutine
  ... code
!$omp end parallel ...
```

i.e. we are calling a subroutine, containing OpenMP-directives, from a parallel part of the program.

To understand what happens we have to read (part of) the following sections (ver. 2.0, integrated in the new version) in the OpenMP standard:

- "Data Environment Rules", details about data scope, what becomes private, shared.
- "Directive Binding": Rules with respect to the dynamic binding of directives. What happens if we put a loop in a subroutine called from a parallel region?
- "Directive Nesting". What happens if we put a parallel region inside a parallel region, for example?

We need to have heard the term "Orphaned Directives" as well.


```
  ...
!$omp parallel shared(s) private(p)  ---
!$omp do                               |
        do j = 1, m                    | lexical extent
          ...                          | of the
        end do                         | parallel region
                                       | (dynamic as
      call sub(s, p)                   |  well)
!$omp end parallel                   ---
  ...

end
! ---------------------------
subroutine sub(s, p)
  integer  :: s          ! shared
  integer  :: p          ! private
  integer  :: local_var  ! private
  ...

!$omp do ...                 ---
        do k = 1, n            | dynamic extent of the
          ...                  |
        end do                 | parallel region
!$omp end do                 ---
  ...
end subroutine sub
```

The **!$omp do** in **sub** is an orphaned directive (it appears in the dynamic extent of the parallel region but not in the lexical extent). This **do** binds to the dynamically enclosing parallel directive and so the iterations in the **do** will be done in parallel (they will be divided between threads). Lexical/dynamic terminology from ver. 2.0 but easier to understand.

Suppose now that **sub** contains the following three loops and that we have three threads:

```
  character (len = *), parameter :: f = '(a, 3i5)'
  ...
  i_am = omp_get_thread_num()

!$omp do private(k)
        do k = 1, 6       ! LOOP 1
          print f, '1:', i_am, omp_get_thread_num(), k
        end do
!$omp end do

        do k = 1, 6       ! LOOP 2
          print f, '2:', i_am, omp_get_thread_num(), k
        end do

!$omp parallel do private(k)
        do k = 1, 6       ! LOOP 3
          print f, '3:', i_am, omp_get_thread_num(), k
        end do
!$omp end parallel do
```

In **LOOP 1** thread 0 will do the first two iterations, thread 1 performs the following two and thread 2 takes the last two.
In **LOOP 2** all threads will do the full six iterations.
In the third case we have:

> A **PARALLEL** directive dynamically inside another **PARALLEL** directive logically establishes a new team, which is composed of only the current thread, unless nested parallelism is established.

We say that the loops is serialized. All threads perform six iterations each.

If we want the iterations to be shared between new threads we can set an environment variable, **`setenv OMP_NESTED TRUE`**, or **`call omp_set_nested(.true.)`** .
If we enable nested parallelism we get three teams consisting of three threads each, in this example.

This is what the (edited) printout from the different loops may look like. **`omp()`** is the value reurned by **`omp_get_thread_num()`**. The output from the loops may be interlaced though.

```
      i_am  omp() k          i_am  omp() k
1:      1    1    3    3:     1    0    1
1:      1    1    4    3:     1    0    2
1:      2    2    5    3:     1    2    5
1:      2    2    6    3:     1    2    6
1:      0    0    1    3:     1    1    3
1:      0    0    2    3:     1    1    4
                       3:     2    0    1
2:      0    0    1    3:     2    0    2
2:      1    1    1    3:     2    1    3
2:      1    1    2    3:     2    1    4
2:      2    2    1    3:     2    2    5
2:      0    0    2    3:     2    2    6
2:      0    0    3    3:     0    0    1
2:      1    1    3    3:     0    0    2
2:      1    1    4    3:     0    1    3
2:      1    1    5    3:     0    1    4
2:      1    1    6    3:     0    2    5
2:      2    2    2    3:     0    2    6
2:      2    2    3
2:      2    2    4
2:      2    2    5
2:      2    2    6
2:      0    0    4
2:      0    0    5
2:      0    0    6
```


Case study I: solving a large and stiff IVP

$$y'(t) = f(t, y(t)),\ y(0) = y_0,\ y,\ y_0 \in \Re^n,\quad f : \Re \times \Re^n \to \Re^n$$

where $f(t, y)$ is expensive to evaluate.

LSODE (Livermore Solver for ODE, Alan Hindmarsh) from netlib. BDF routines; Backward Differentiation Formulas.

Implicit method: t_k present time, $y^{(k)}$ approximation of $y(t_k)$.

Backward Euler (simplest BDF-method). Find $y^{(k+1)}$ such that:

$$y^{(k+1)} = y^{(k)} + hf(t_{k+1}, y^{(k+1)})$$

LSODE is adaptive (can change both h and the order).

Use Newton's method to solve for $z \equiv y^{(k+1)}$:

$$z - y^{(k)} - hf(t_{k+1}, z) = 0$$

One step of Newton's method reads:

$$z^{(i+1)} = z^{(i)} - \left[I - h\frac{\partial f}{\partial y}(t_{k+1}, z^{(i)})\right]^{-1} (z^{(i)} - y^{(k)} - hf(t_{k+1}, z^{(i)}))$$

The Jacobian $\frac{\partial f}{\partial y}$ is approximated by finite differences one column at a time. Each Jacobian requires n evaluations of f.

$$\frac{\partial f}{\partial y}\, e_j \approx \left[f(t_{k+1}, z^{(i)} + e_j\delta_j) - f(t_{k+1}, z^{(i)})\right]/\delta_j$$

e_j is column j in the identity matrix I.

Parallelise the computation of the Jacobian, by computing columns in parallel. Embarrassingly parallel.

Major costs in LSODE:

1. Computing the Jacobian, J, (provided f takes time).
2. LU-factorization of the Jacobian (once for each time step).
3. Solving the linear systems, given L and U.

What speedup can we expect?

Disregarding communication, the wall clock time for p threads, looks something like (if we compute J in parallel):

$$wct(p) = time(\text{LU}) + time(\text{solve}) + \frac{time(\text{computing J})}{p}$$

If the parallel part, "computing J", dominates we expect good speedup at least for small p. Speedup may be close to linear, $wct(p) = wct(1)/p$.

For large p the serial (non-parallel) part will start to dominate.

How should we speed up the serial part?

1. Switch from Linpack, used in LSODE, to Lapack.
2. Try to use a parallel library like **`complib.sgimath_mp`** (SGI).

This is an old test of solving using **`dposv`** to solve a full, positive definite, and symmetric $Ax = b$-problem, $n = 2500$:

```
> f90 -O3 main.f -lcomplib.sgimath_mp -lmp
```

```
p       time command          p       time command
1       22.58                 5        8.61
2       13.75                 6        8.06
3       10.80                 7        7.58
4        9.42                 8        7.35
```


After having searched LSODE (Fortran 66):

```
c if miter = 2, make n calls to f to approximate j.
 ...
      j1 = 2
      do 230 j = 1,n
        yj = y(j)
        r = dmax1(srur*dabs(yj),r0/ewt(j))
        y(j) = y(j) + r
        fac = -hl0/r
        call f (neq, tn, y, ftem)
        do 220 i = 1,n
220       wm(i+j1) = (ftem(i) - savf(i))*fac
        y(j) = yj
        j1 = j1 + n
230     continue
 ...
c add identity matrix.
 ...
c do lu decomposition on p.
       call dgefa (wm(3), n, n, iwm(21), ier)
...
 100  call dgesl (wm(3), n, n, iwm(21), x, 0)
```

We see that
`r` $= \delta_j$
`fac` $= -h/\delta_j$
`tn` $= t_{k+1}$
`ftem` $= f(t_{k+1}, z^{(i)} + e_j\delta_j)$
`wm(2...)` is the approximation to the Jacobian.

From reading the code: **`neq`** is an array but **`neq(1) = n`**.

The parallel version

- `j, i, yj, r, fac, ftem` are private
  `ftem` is the output (y') from the subroutine
- `j1 = 2` offset in the Jacobian; use `wm(i+2+(j-1)*n)`
  no index conflicts
- `srur, r0, ewt, hl0, wm, savf, n, tn` are shared
- `y` is a problem since it is modified. `shared` does not work. `private(y)` will not work either; we get an uninitialised copy.
  In the revision of the OpenMP-standard there is `firstprivate` which makes a private and initialised copy.

```
c$omp  parallel do private(j, yj, r, fac, ftem, first)
c$omp+ shared(f, srur, r0, ewt, hl0, wm, savf,n,neq,tn)
c$omp+ firstprivate(y)
      do j = 1,n
        yj = y(j)
        r = dmax1(srur*dabs(yj),r0/ewt(j))
        y(j) = y(j) + r
        fac = -hl0/r
        call f (neq, tn, y, ftem)
        do i = 1,n
          wm(i+2+(j-1)*n) = (ftem(i) - savf(i))*fac
        end do
        y(j) = yj
      end do
```

Did not converge! After reading of the code:

```
 dimension neq(1), y(1), yh(nyh,1), ewt(1), ftem(1)
change to
 dimension neq(1), y(n), yh(nyh,1), ewt(1), ftem(n)
```


Case study II: sparse matrix multiplication

Task: given a matrix A which is large, sparse and symmetric we want to:

- compute a few of its smallest eigenvalues OR
- solve the linear system $Ax = b$

n is the dimension of A and nz is the number of nonzeros.

Some background, which you may read after the lecture:

We will study iterative algorithms based on forming the Krylov subspace: $\{v, Av, A^2v, \ldots, A^{j-1}v\}$. v is a random-vector. So, Paige-style Lanczos for the eigenvalue problem and the conjugate-gradient method for the linear system, for example. When solving $Ax = b$ we probably have a preconditioner as well, but let us skip that part.

The vectors in the Krylov subspace tend to become almost linearly dependent so we compute an orthonormal basis of the subspace using Gram-Schmidt. Store the basis-vectors as columns in the $n \times j$-matrix V_j.

Project the problem onto the subspace, forming $T_j = V_j^T A V_j$ (tridiagonal) and solve the appropriate smaller problem, then transform back.

T_j and the basis-vectors can be formed as we iterate on j. In exact arithmetic it is sufficient to store the three latest v-vectors in each iteration.

p is the maximum number of iterations.

A Lanczos-algorithm may look something like:

	# operations
$v =$ `randn(n, 1)`	$\mathcal{O}(n)$
$v = v/\|\|v\|\|_2$	$\mathcal{O}(n)$
for $j = 1$ to p do	
$t = Av$	$\mathcal{O}(nz)$
if $j > 1$ then $t = t - \beta_{j-1}w$ endif	$\mathcal{O}(n)$
$\alpha_j = t^T v$	$\mathcal{O}(n)$
$t = t - \alpha_j v$	$\mathcal{O}(n)$
$\beta_j = \|\|t\|\|_2$	$\mathcal{O}(n)$
$w = v$	$\mathcal{O}(n)$
$v = t/\beta_j$	$\mathcal{O}(n)$
Solve the projected problem and and check for convergence	$\mathcal{O}(j)$
end for	

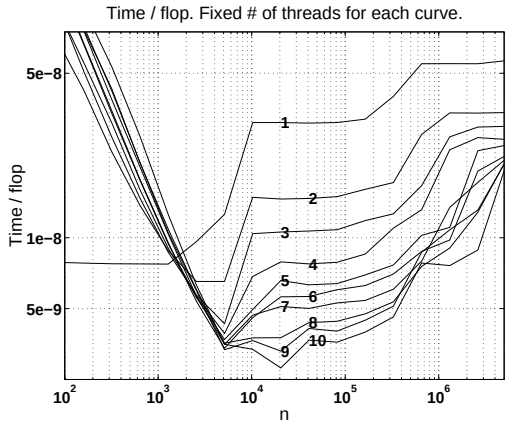
The diagonal of T_j is $\alpha_1, \ldots, \alpha_j$ and the sub- and super-diagonals contain $\beta_1, \ldots, \beta_{j-1}$.

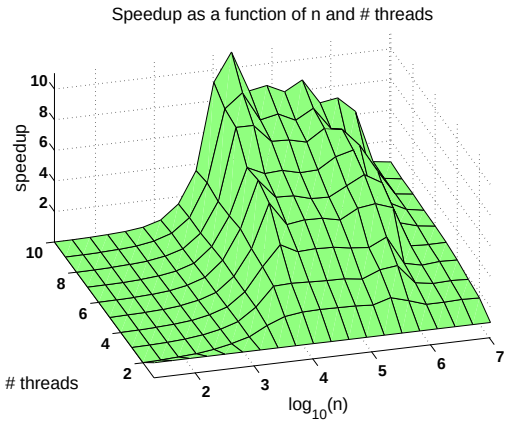
How can we parallelise this algorithm?

- The j-iterations and the statements in each iteration must be done in order. Not possible to parallelise.
- It is easy to parallelise each of the simple vector operations (the ones that cost $\mathcal{O}(n)$). May not give any speedup though.
- The expensive operation in an iteration is usually Av.
- Solving the projected problem is rather fast and not so easy to parallelise (let us forget it).

We will not look at graph-based pre-ordering algorithms.
A block diagonal matrix would be convenient, for example.

Vectors must not be too short if we are going to succeed.
The figures show how boye (SGI) computes `daxpy` for different n and number of threads.





The tricky part, parallelising $t = Av$

A is large, sparse and symmetric so we need a special data structure which takes the sparsity and the symmetry into account.

First try: store all triples $(r, c, a_{r,c})$ where $a_{r,c} \neq 0$ and $r \leq c$. I.e. we are storing the nonzeros in the upper triangle of the matrix.

The triples can be stored in three arrays, **rows**, **cols** and **A** or as an array of triples. Let us use the three arrays and let us change the meaning of **nz** to mean the number of stored nonzeros. The first coding attempt may look like:

```
do k = 1, nz
  if ( rows(k) == cols(k) ) then
    ...                 ! diagonal element
  else
    ...                 ! off-diagonal element
  end if
end do
```

If-statements in loops mat degrade performance, so we must think some more.

If A has a dense diagonal we can store it in a separate array, **diag_A** say. We use the triples for all $a_{r,c} \neq 0$ and $r < c$ (i.e. elements in the strictly upper triangle).

If the diagonal is sparse we can use pairs $(r, a_{r,r})$ where $a_{r,r} \neq 0$. Another way is to use the triples format but store the diagonal first, or to store $a_{k,k}/2$ instead of $a_{k,k}$.

Our second try may look like this, where now **nz** is the number stored nonzeros in the strictly upper triangle of A.

```
! compute t = diag(A) * t
 ...

  do k = 1, nz  ! take care of the off-diagonals
    r    = rows(k)
    c    = cols(k)
    t(r) = t(r) + A(k) * v(c)  ! upper triangle
    t(c) = t(c) + A(k) * v(r)  ! lower triangle
  end do
```

$$\begin{bmatrix} \vdots \\ t_r \\ \vdots \\ t_c \\ \vdots \end{bmatrix} = \begin{bmatrix} \ddots & \vdots & & \vdots & \\ \cdots & a_{r,r} & \cdots & a_{r,c} & \cdots \\ & \vdots & \ddots & \vdots & \\ \cdots & a_{c,r} & \cdots & a_{c,c} & \cdots \\ & \vdots & & \vdots & \ddots \end{bmatrix} \begin{bmatrix} \vdots \\ v_r \\ \vdots \\ v_c \\ \vdots \end{bmatrix}$$

Let us now concentrate on the loops for the off-diagonals and make it parallel using OpenMP.

Note that we access the elements in A once.


```
! Take care of diag(A)
 ...
!$omp do default(none), private(k, r, c), &
!$omp    shared(rows, cols, A, nz, v, t)
  do k = 1, nz  ! take care of the off-diagonals
    r    = rows(k)
    c    = cols(k)
    t(r) = t(r) + A(k) * v(c)  ! upper triangle
    t(c) = t(c) + A(k) * v(r)  ! lower triangle
  end do
```

This will probably give us the wrong answer (if we use more than one thread) since two threads can try to update the same **t**-element.

Example: The first row in A it will affect t_1, t_3 and t_5, and the second row in A will affect t_2, t_4 and t_5. So there is a potential conflict when updating t_5 if the two rows are handled by different threads.

$$\begin{bmatrix} t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \end{bmatrix} = \begin{bmatrix} 0 & 0 & a_{1,3} & 0 & a_{1,5} \\ 0 & 0 & 0 & a_{2,4} & a_{2,5} \\ a_{1,3} & 0 & 0 & 0 & 0 \\ 0 & a_{2,4} & 0 & 0 & 0 \\ a_{1,5} & a_{2,5} & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{bmatrix}$$

If the first row is full it will affect all the other rows. A block diagonal matrix would be nice.

As in the previous example it is not possible to use critical sections. Vector reduction is an option and we can do our own exactly as in case study II. Here is a slightly different version using a public matrix, instead.

X has n rows and as many columns as there are threads, **num_thr** below. Each thread stores its sum in **X(:, thr)**, where **thr** is the index of a particular thread.

Here is the code:

```
!$omp parallel shared(X, ...)
      ...
      i_am = omp_get_thread_num() + 1
      ...
      do i = 1, n         ! done by all threads
        X(i, i_am) = 0.0 ! one column each
      end do

!$omp do
        do i = n + 1, nz
          r = rows(i)
          c = cols(i)
          X(r, i_am) = X(r, i_am) + A(i) * v(c)
          X(c, i_am) = X(c, i_am) + A(i) * v(r)
        end do
!$omp end do

!$omp do
       do i = 1, n
          do thr = 1, num_thr
            t(i) = t(i) + X(i, thr)
          end do
        end do
      ...
!$omp end parallel
```

The addition loop is now parallel, but we have bad cache locality when accessing **X** (this can be fixed). None of the parallel loops should end with **nowait**.
One can get a reasonable speedup (depends on problem and system).

Compressed storage

The triples-format is not the most compact possible. A common format is the following compressed form. We store the diagonal separately as before and the off-diagonals are stored in order, one row after the other. We store **cols** as before, but **rows** now points into **cols** and **A** where each new row begins. Here is an example (only the strictly upper triangle is shown):

$$\begin{bmatrix} 0 & a_{1,2} & a_{1,3} & 0 & a_{1,5} \\ 0 & 0 & a_{2,3} & a_{2,4} & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & a_{4,5} \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

is stored as **A** = [$a_{1,2}$ $a_{1,3}$ $a_{1,5}$ | $a_{2,3}$ $a_{2,4}$ | 0 | $a_{4,5}$],
cols = [2 3 5 | 3 4 | • | 5], (• fairly arbitrary, n say)
rows = [1 4 6 7 8]. (8 is one step after the last)

Note that **rows** now only contains n elements.
The multiplication can be coded like this (no OpenMP yet):

```
  ... take care of diagonal, t = diag(A) * v

do r = 1, n - 1  ! take care of the off-diagonals
  do k = rows(r), rows(r + 1) - 1
    c    = cols(k)
    t(r) = t(r) + A(k) * v(c)  ! upper triangle
    t(c) = t(c) + A(k) * v(r)  ! lower triangle
  end do
end do
```


We can parallelise this loop (with respect to **do r**) in the same way as we handled the previous one (using the extra array **X**).

There is one additional problem though.
Suppose that the number of nonzeros per row is fairly constant and that the nonzeros in a row is evenly distributed over the columns.

If we use default static scheduling the iterations are divided among the threads in contiguous pieces, and one piece is assigned to each thread. This will lead to a load imbalance, since the upper triangle becomes narrower for increasing **r**.

To make this effect very clear I am using a full matrix (stored using a sparse format).

A hundred matrix-vector multiplies with a full matrix of order 2000 takes (wall-clock-times):

#threads →	1	2	3	4
triple storage	19.7	10.1	7.1	6.9
compressed, static	20.1	16.6	12.6	10.1
compressed, static, 10	20.1	11.2	8.8	7.5

The time when using no OpenMP is essentially equal to the time for one thread.