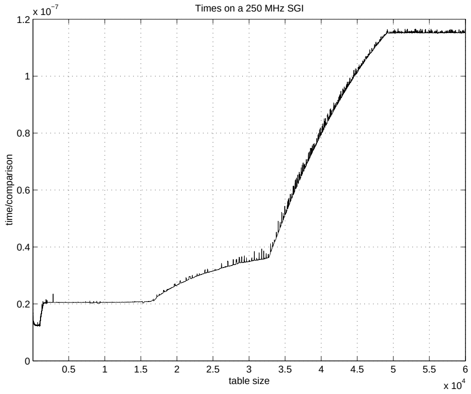
Lecture Notes
on
High Performance Computing



Thomas Ericsson
Mathematics
Chalmers
2010

High Performance Computing

Thomas Ericsson, computational mathematics, Chalmers

office: L2075
phone: `772 10 91`
e-mail: `thomas@chalmers.se`

My homepage:
`http://www.math.chalmers.se/~thomas`

The course komepage:
`http://www.math.chalmers.se/Math/Grundutb/CTH/tma881/0910`

Assignments, copies of handouts (lecture notes etc.), schedule can be found on the www-address.

We have two lectures and two labs per week. No lab today.

Prerequisites:

- a programming course (any language will do)
- the basic numerical analysis course
- an interest in computers and computing; experience of computers and programming

If you do not have a Chalmers/GU-account, you need to contact the helpdesk to get a personal account.

Why do we need HPC?

- Larger and more complex mathematical models require greater computer performance. Scientific computing is replacing physical models, for example.
- When computer performance increases new problems can be attacked.

To solve problems in a reasonable time using available resources in a good way we need competence in the following areas:

1. algorithms
2. hardware; limitations and possibilities
3. code optimization
4. parallelization and decomposition of problems and algorithms
5. software tools

This course deals primarily with points 2-5. When it comes to algorithms we will mostly study the basic building blocks.

The next pages give some, old and new, examples of demanding problems.

For each problem there is a list of typical HPC-questions, some of which you should be able to answer after having passed this course.

A real-time application

Simulation of surgery; deformation of organs; thesis work (examensarbete) Prosolvia. Instead of using a standard deformation more realistic deformations were required. Organs have different structure, lung, liver etc.

Requires convincing computer graphics in real time; no flicker of the screen; refresh rate ≥ 72 Hz.

Using a shell model implies solving a sparse linear system, $Ax = b$, in 0.01 s (on a 180 MHz SGI O2 workstation, the typical customer platform).

- What do we expect? Is there any chance at all?
- Is there a better formulation of the problem?
- What linear solver is fastest? Sparse, band, iterative, ...?
  - Datastructures, e.g.
  - General sparse: $\{j, k, a_{j,k}\}, \ a_{j,k} \neq 0, \ j \geq k$.
  - Dense banded.
  - Memory accesses versus computation.
- Should we write the program or are there existing ones? Can we us the high performance library made by SGI?
- If we write the code, what language should we use, and how should we code in it for maximum performance?
- Can we find out if we get the most out of the CPU? How? Are we using 20% of top speed or 90%?
- What limits the performance, the CPU (the floating point unit) or the memory?

Integrals and probability

Graduate student in mathematical statistics. What remained of the thesis work was to make large tables of integrals:

$$\int_0^\infty f(x)\,dx, \quad \text{where} \quad f(x) \to \infty \quad \text{when} \quad x \to 0+$$

The integrand was complicated and was defined by a Matlab-program. No chance of finding a primitive function. Using the Matlab routine **quad** (plus a substitution), to approximate the integral numerically, the computer time was estimated to several CPU-years. Each integral took more than an hour to compute.

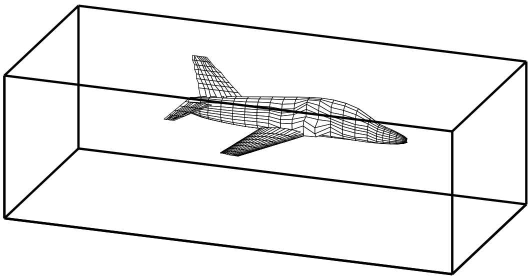
- Is this reasonable, is the problem really this hard?
- Are Matlab and **quad** good tools for such problems?
- How can one handle singularities and infinite intervals?

Solution: Switching to Fortran and Quadpack (a professional package) the time for one integral came down to 0.02 s (with the same accuracy in the result).

- Matlab may be quite slow.
- **quad** is a simple algorithm; there are better methods available now, e.g. **quadl**.

Mesh generation for PDEs in 3D

require huge amounts of storage and computer time.
Airflow around an aircraft; 3 space dimensions and time.
CFD (Computational Fluid Dynamics).



Discretize (divide into small volume elements) the air in the box and outside the aircraft. Mesh generation (using **m3d**, an ITM-project, Swedish Institute of Applied Mathematics) on one RS6000 processor:

```
wed may 29 12:54:44 metdst 1996  So this is old stuff
thu may 30 07:05:53 metdst 1996

183463307 may  2 13:46 s2000r.mu

tetrahedrons in structured mesh:   4 520 413
tetrahedrons in unstructured mesh: 4 811 373
```

- Choice of programming language and data structures.
- Handling files and disk.

Now we must solve the PDE given the mesh...

More PDEs: Weather forecasts

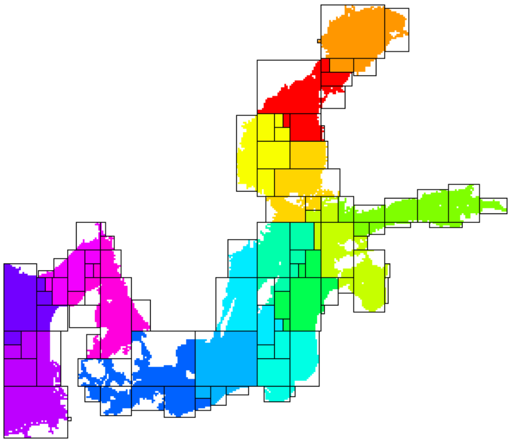
SMHI, Swedish Meteorological and Hydrological Institute.
HIRLAM (HIgh Resolution Limited Area Model).
HIROMB (HIgh Resolution Operational Model of the Baltic).

Must be fast. “Here is the forecast for yesterday.”

Parallel implementation of HIROMB, lic-thesis KTH.

Divide the water volume into pieces and distribute the pieces onto the CPUs.

- Domain decomposition (MPI, Message Passing).
- Load balancing. Communication versus computation.
- Difficult sub-problems. Implicit solver for the ice equations. Large (10^5 equations) sparse Jacobians in Newton’s method.



Three weeks runtime

Consultant work, for Ericsson, by a colleague of mine.

Find the shape of a TV-satellite antenna such that the “image” on the earth has a given form (some TV-programs must no be shown in certain countries).

Algorithm: shape optimization + ray tracing.

Three weeks runtime (for one antenna) on a fast single-CPU PC. One of the weeks for evaluating trigonometric functions.

You do not know much about the code (a common situation). Can you make it faster? How? Where should you optimize?

- Choice of algorithm.
- Profiling.
- Faster trigonometric functions. Vector forms?
- Parallel version? How? Speedup? (OpenMP)

A Problem from Medicine

Inject a radionuclide that attacks a tumour.
How does the radioactivity affect the surrounding tissue?

To be realistic the simulation should contain some $7 \cdot 10^7$ cells.
Matlab-program computing a huge number of integrals (absorbed dose).
The original program would take some 26 000 years runtime.

After switching to Fortran90, changing the algorithm, by precomputing many quantities (the most important part) and cleaning up the code, the code solved the problem in 9 hours on a fast PC (a speedup by a factor of $2.6 \cdot 10^7$).

Contents of the Course

There are several short assignments which illustrate typical problems and possibilities.

- Matlab (get started exercises, not so technical).
- Uniprocessor optimization.
- Low level parallel threads programming.
- MPI, parallel programming using Message Passing.
- OpenMP, more automatic threads programming.

Many small and short programs, matrix- and vector computations (often the basic operations in applications).
Simple algorithms.

E.g. test how indirect addressing (using pointers) affects performance:

```
do k = 1, n
  j    = p(k)   ! p is a pointer array
  y(j) = y(j) + a * x(j)
end do
```

- You will work with C, Fortran and some Matlab and several software tools and packages.
- Java is not so interesting for HPC (so only in the lectures).

At most two students per group. Each group should hand in written reports on paper (Swedish is OK). Not e-mail.

There are deadlines, see www.

In more detail...

- A sufficient amount of Fortran90 (77), C for the labs.
- Computer architecture, RISC/CISC, pipelining, caches...
- Writing efficient programs for uniprocessors, libraries, Lapack, ...
- Necessary tools: **`make`**, **`ld`**, **`prof`**, ...
- Introduction to parallel systems, SIMD, MIMD, shared memory, distributed memory, network topologies, ...
- POSIX threads, pthreads.
- MPI, the Message Passing Interface.
- Shared memory parallelism, OpenMP.
- Parallel numerical analysis, packages.

Note: this is not a numerical analysis course. We will study simple algorithms (mainly from linear algebra).

You will not become an expert in any of the above topics (smörgåsbord), but you will have a good practical understanding of high performance computing.

The course will give you a good basis for future work.

Literature

You can survive on the lecture notes, web-pages and man-pages.

I used to recommend:

K. Dowd & C. Severance, High Performance Computing, O'Reilly, 2nd Edition July 1998, 466 pages.
`http://www.ora.com` The book is a bit dated and can be hard to find (out of print).

Here follow some reference books I have (or will have) in my bookshelf (nice to have, but not necessary for the course). There are many special books in HPC and a huge number of books dealing with parallel computers. Some books are available as E-books (Books24x7) through the Chalmers library homepage.

A web-version of Designing and Building Parallel Programs, by Ian Foster, 1995, can be found at:
`http://www-unix.mcs.anl.gov/dbpp`

For more free manuals for code optimization and system reference see the course homepage.

A few C- and Fortran books (there are many)

- B. W. Kernighan, D. M. Ritchie, The C Programming Language (2nd ed.), Prentice Hall, 1988. Get the ANSI C version.
- P. van der Linden, Expert C Programming : Deep C Secrets, Prentice Hall, 1994.
- M. Metcalf, J. Reid, M. Cohen, Fortran 95/2003 Explained, 3rd ed. (2nd ed., 1999, is OK as well, E-book).
- Suely Oliveira, David E. Stewart, Writing Scientific Software: A Guide for Good Style, Cambridge UP, 2006. E-book.

Code Optimization

- Randy Allen, Ken Kennedy, Optimizing Compilers for Modern Architectures: A Dependence-based Approach, Morgan Kaufmann, 2001.
- Stefan Andersson, Ron Bell, John Hague, Holger Holthoff, Peter Mayes, Jun Nakano, Danny Shieh, Jim Tuccillo, RS/6000 Scientific and Technical Computing: POWER3 Introduction and Tuning Guide, 1998,
IBM Redbook (`www.redbooks.ibm.com`, free).
- Steve Behling, Ron Bell, Peter Farrell, Holger Holthoff, Frank O'Connell, Will Weir, The POWER4 Processor Introduction and Tuning Guide, 2001,
IBM Redbook (`www.redbooks.ibm.com`, free).
- Originally written by David Cortesi, based on the first edition by Jeff Fier; updated by Jean Wilson and Julie Boney, Origin 2000 and Onyx2 Performance Tuning and Optimization Guide, Document Number: 007-3430-003, 2001, (free `techpubs.sgi.com`).
- I. Crawford, K. Wadleigh, Software Optimization for High Performance Computing: Creating Faster Applications, Prentice Hall, 2000 (Hewlett Packard).
- Rajat P. Garg, Ilya Sharapov, Illya Sharapov, Techniques for Optimizing Applications: High Performance Computing, Prentice Hall, 2001 (Sun UltraSPARC platforms).
- Kevin Smith, Richard Gerber, Aart J. C. Bik, The Software Optimization Cookbook, High Performance Recipes for IA 32 Platforms, Intel Press, 2005. E-book.

Computers

- John L. Hennessy, David A. Patterson, Andrea C. Arpaci-Dusseau, Computer Architecture: A Quantitative Approach (with CDROM), Morgan Kaufmann, 2006. E-book.
- W. R. Stevens, Advanced Programming in the UNIX Environment, Addison Wesley, 1992.

- A. Oram, S. Talbott, Managing Projects with make, 2nd ed., O'Reilly, 1991.
- R. Mecklenburg, Managing Projects with GNU Make, 3rd ed, O'Reilly, 2004.
- G. Anderson, P. Anderson, Unix C Shell Field Guide, Prentice Hall, 1986.

Parallel Programming

- Yukiya Aoyama, Jun Nakano, RS/6000 SP: Practical MPI Programming, IBM Redbook (`www.redbooks.ibm.com`, free).
- D. R. Butenhof, Programming With Posix Threads, Addison Wesley, 1997.
- Rohit Chandra, Dave Kohr, Ramesh Menon, Leo Dagum, Dror Maydan, Jeff McDonald, Parallel Programming in OpenMP, Morgan Kaufmann, 2001. E-book.
- Barbara Chapman, Gabriele Jost, Ruud van der Pas, David J. Kuck, Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation), MIT Press, 2007. E-book.
- William Gropp, Ewing Lusk, and Anthony Skjellum, Using MPI, 2nd Edition, MIT Press.
- Lucio Grandinetti (ed), Grid Computing: The New Frontier of High Performance Computing, Elsevier, 2005. E-book.
- Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill, Patterns for Parallel Programming, Addison Wesley Professional, 2004.
- Peter Pacheco, Parallel Programming with Mpi, Morgan Kaufmann, 1997. (Not MPI-2)
- Wesley Petersen, Peter Arbenz, Introduction to Parallel Computing A practical guide with examples in C, 2004, Oxford UP (E-book)
- Michael J. Quinn, Parallel Programming in C with MPI and OpenMP, McGraw-Hill Education, 2003.

- Laurence T. Yang and Minyi Guo (eds), High Performance Computing: Paradigm and Infrastructure, John Wiley, 2006. E-book.

Beowulf computers

- Robert Lucke, Robert W. Lucke, Building Clustered Linux Systems, Prentice Hall, 2004
- Karl Kopper, The Linux Enterprise Cluster, No Starch Press, 2005
- William Gropp, Thomas Sterling, Ewing Lusk, Beowulf Cluster Computing with Linux, MIT press, 2003

Numerical methods

- D. P. Bertsekas, J. N. Tsitsiklis, Parallel and distributed computation, numerical methods, Prentice-Hall, 1989
- P. Bjorstad, Domain Decomposition Methods in Sciences and Engineering, John Wiley & Sons, 1997.
- J. J. Dongarra, I. S. Duff, D. C. Sorensen, H. A. van der Vorst, Numerical Linear Algebra on High-performance Computers, SIAM, 1998.
- Robert A. van de Geijn and Enrique S. Quintana-Ortí, The Science of Programming Matrix Computations, Free download `http://www.lulu.com/content/1911788`. About FLAME-project (Formal Linear Algebra Methods Environment).

Programming languages for HPC

The two dominating language groups are Fortran and C/C++.

Fortran90/95/2003 is more adapted to numerical computations.
It has support for complex numbers, array operations, handling of arithmetic etc. New code is written in Fortran90/95/2003 (Fortran66/77 is used for existing code only).

Fortran90/95 has simple OO (Object Oriented) capabilities (modules, overloading, but no inheritance).

C++ is OO and has support for some numerics (standard library) or can be adapted using classes, overloading etc.

C is less suited for numerical computing (my opinion).
Too few built-in tools and it cannot be modified.

C/C++ is almost the only choice when it comes to low level unix programming. It is not uncommon to code the computational part in Fortran and a computer graphics (or unix) part in C/C++.

Commercial Fortran compilers generate fast code: competition (benchmarks), simple language, no aliasing. One of Fortran's most important advantages. Speed is (almost) everything in many applications.

It is harder to generate fast code from C/C++.
It is very easy to write inefficient programs in C++.

More about performance later on.

Matlab

Matlab is too slow for demanding applications:

- Statements may be interpreted (not compiled, although there is a Matlab compiler). In Matlab 6.5 (and later) there is a JIT-accelerator (JIT = Just In Time).
- The programmer has poor control over memory.
- It is easy to misuse some language constructs, e.g. dynamic memory allocation.
- Matlab is written in C, Java and Fortran.
- Matlab is not always predictable when it comes to performance.
- The first assignment contains more examples and a case study. You can start working with the Matlab assignment **now**.

Fortran90 and C

The next few pages contain the rudiments of Fortran90 and C and a glance at Fortran77. It is sufficient for the assignments, but you need more for real programming.

I have not tried to show all the different ways a program can be written. Both C and Fortran have several forms of some constructs. Professional code may have many extra details as well.

I have not shown any C++ code (but my example is available in C++-form on the web). C++ is too large and complicated and my labs are not OO. But since C++ means C = C + 1, my C-program is also a C++-program.

Some people use the C-part of C++ together with some convenient C++-constructs (e.g. //-comments, reference variables, simplified I/O).

Fortran90 is <u>much</u> nicer than Fortran77, almost a new language. Fortran77 is quite primitive. Fortran77 is still used for HPC.

Millions of lines are available in Fortran77 (some of them will be used in one lab) so it is necessary to understand the basics.

The example code contains one main-program one function and a procedure (void function). The function computes the inner product of two vectors and the procedure sums the elements in an array and returns the sum in a parameter.

```
program main
!
! Comments: everything after !
! Case or blanks (between keywords) are not significant
! (unless they are in strings).
!
! Fortran has an implicit type rule but
! implicit none forces you to declare everything.
!
  implicit none              ! Highly recommended!
  integer          :: k, n, in
  double precision  :: s
  double precision  :: ddot   ! a function

! Arrays start at one by default.
  double precision, dimension(100) :: a, b

  n = 100
  print*, "Type a value for in:"
  read*,  in
  print*, "This is how you write: in = ", in

  do k = 1, n             ! do when k = 1, 2, ..., n
    a(k) = k
    b(k) = -sin(dble(k))  ! using sin
  end do
!
! Call by reference for all variables.
!
  print*, "The inner product is ", ddot(a, b, n)

  call sum_array(a, s, n)    ! NOTE, call
  print*, "The sum of the array is ", s

end program main
```

```
function ddot(x, y, n) result(s)
!
! You give the function a value by assigning
! something to the result variable, s (in this case).
!
  implicit none
  integer :: n
  double precision, dimension(n) :: x, y
  double precision :: s   ! The type of the function

  integer          :: k

  s = 0.0
  do k = 1, n
    s = s + x(k) * y(k)
  end do

end function ddot



subroutine sum_array(a, s, n)
  implicit none
  integer           :: n
  double precision  :: s
  double precision, dimension(n) :: a

  integer :: k

  s = 0.0
  do k = 1, n
    s = s + a(k)
  end do

end subroutine sum_array
```

Some comments. Since Fortran90 has support for array operations the main program could have been shortened:

```
  print*, "The inner product is ", dot_product(a, b)
  print*, "The sum of the array is ", sum(a)
```

`dot_product` and `sum` are built-in.

A long statement can be broken up into several lines. The continued line should end with a **&** .

1 is an integer constant. **1.0** is a real constant (single precision) and **1.0d0** is a double precision constant in Fortran77.

In Fortran90 it is possible to parameterize the real- and integer types and create more portable code using a module (similar to a simple class) e.g.:

```
module floating_point
! sp = at least 5 significant decimals and
! |exponent range| <= 30 which implies
! IEEE single precision.

 integer, parameter :: sp = selected_real_kind(5, 30)
 integer, parameter :: dp = selected_real_kind(10, 300)
 integer, parameter :: prec = dp ! pick one
end module floating_point

program main
  use floating_point  ! gives access to the module
  real (kind = prec)                 :: x, y
  real (kind = prec), dimension(100) :: a, b

  x = 1.24_prec    ! constant
  y = 1.24e-4_prec ! constant
...
```

Here comes the Fortran77-version, but first some comments.

Fortran90 is almost a new language, but in my simple example the differences are not that striking:

- F77 has a column oriented layout dating back to the 80 column punched card.
- No result-statement in functions.
- Different type declarations:

  ```
  double precision  a(n)
  ```

  instead of

  ```
  double precision, dimension(n) :: a
  ```

  although F77-declarations are allowed in F90 as well. A Fortran77-program is (essentially) also a Fortran90-program, so it is possible to mix the two styles.

Fortran90 has array operations, pointers, recursion, prototypes, modules, overloading of operators (and more). Fortran77 has none of these.

The example program, coded in F77, is listed on the following two pages. It violates the ANSI-standard in several ways, the most important being the use of do/enddo. Here is the proper way of writing a F77-loop using labels (you will see it in a lab):

```
       do 10 k = 1, n
         s = s + x(k) * y(k)
10     continue
```

```
      program main
*
*     Comments:           c, C or * in column one
*                        text in columns > 72
*     !                  F90-comment
*     First five columns: labels
*     Continuation line:  non-blank in column 6
*     Statements:         columns 7 through 72
*     Case or blanks are not significant
*     (unless they are in strings).
*
*     Arrays start at one by default.
*234567890
      integer           k, n, in
      double precision  a(100), b(100), sum
      double precision  ddot   ! a function

      n = 100
      print*, "Type a value for in:"
      read*,  in
      print*, "This is how you write: in = ", in

      do k = 1, n    ! do when k = 1, 2, ..., n
        a(k) = k
        b(k) = -sin(dble(k))  ! using sin
      end do
*
*     Call by reference for all variables.
*
      print*, "The inner product is ", ddot(a, b, n)

      call sum_array(a, sum, n) ! NOTE, call
      print*, "The sum of the array is ", sum

      end
```

```
      double precision function ddot(x, y, n)
*
*     Fortran has an implicit type rule but
*     implicit none forces you to declare everything.
*     Highly recommended!
*
      implicit none
      integer           n
      double precision  x(n), y(n)

      integer           k
      double precision  sum

      sum = 0.0
      do k = 1, n
        sum = sum + x(k) * y(k)
      end do

      ddot = sum ! give the function its value

      end


      subroutine sum_array(a, sum, n)
      implicit none
      integer           n
      double precision  a(n), sum

      integer           k

      sum = 0.0   ! 0.0 is single and 0.0d0 double
      do k = 1, n
        sum = sum + a(k)
      end do

      end
```

How to compile

The Fortran compilers available on the student system are: `g77` (Fortran77), `gfortran` and `g95` (both Fortran90 and 77).
It would be interesting to use the Intel `ifort`-compiler, but we do not have a license. You can fetch a free copy for Linux (provided you have the disk space, a few hundred Mbyte). See www.

In these handouts I will use `g95` and I will assume that a Fortran90-program has the suffix `.f90`. Some examples:

```
%                 my prompt
% g95 prog.f90    if everything in one prog.f90
                  prog.f would be Fortran77

Produces the executable file a.out
% a.out           run (or ./a.out if no . in the path)

Suppose we have three files main.f90, dot.f90, sum.f90
% g95 main.f90 dot.f90 sum.f90

Can compile the files one by one.
-c means "compile only", do not link.

% g95 -c main.f90      -> object file main.o
% g95 -c dot.f90       -> object file dot.o
% g95 -c sum.f90       -> object file sum.o
% g95 main.o dot.o sum.o  link the object files

% g95 main.o dot.f90 sum.o  works as well, note .f90

Can give many options (or flags) to the compiler, e.g.

% g95 -O3 prog.f90  optimize the code
                    not standard names
```

Next comes the example program in C. See the course page for the same program in C++.

```
/* /usr/include/stdio.h, definitions for IO.*/
#include <stdio.h>
#include <math.h>

/* Here are the prototypes for our procedures*/
void    sum_array(double[], double*, int);
double  ddot(double[], double[], int);

int main()
{
  int      k, n, in;
  double   a[100], b[100], sum;
  /* Arrays are indexed from 0,  a[0], ..., a[99]*/

  n = 100;
  printf("Type a value for in: ");
  scanf("%d", &in);  /* & = the address, the pointer*/

  /* %d for int, %f or %e for float. \n is newline*/
  printf("This is how you write: in =  %d\n", in);

  for (k = 0; k < n; k++) { /* for k = 0, ..., n-1 */
    a[k] = k + 1;           /* k++ means k = k + 1 */
    b[k] = -sin(k + 1);     /* requires -lm       */
  }

  /* Compute the inner product between a and b.*/
  printf("The inner product is %f\n", ddot(a, b, n));

  /* Send the address of sum.
     Arrays are passed by reference automatically.*/
  sum_array(a, &sum, n);

  printf("The sum of the array is %f\n", sum);

  return 0; /* return status to unix */
}
```

```
double ddot(double x[], double y[], int n)
{
  int               k;
  double            sum;

  sum = 0.0;
  for (k = 0; k < n; k++)
    sum += x[k] * y[k];  /* short for sum = sum + ... */

  return sum;    /* return the result */
}

void sum_array(double a[], double*sum, int n)
/*
 * void, i.e. not a "mathematical" function.
 * double *sum means that *sum is a double. sum
 * itself is the address, the pointer to sum.
 */
{
  int               k;  /* k is local to sum_array */

  *sum = 0;    /* sum = 0 is WRONG; it will give a
                  Segmentation Fault */

  for (k = 0; k < n; k++)
    *sum += a[k];  /* i.e. *sum = *sum + a[k] */
}
```

Compile by: `cc prog.c -lm` or `cc -O3 prog.c -lm` .
`cc` is a link to `gcc`.
Separate files are OK (as for Fortran).

The C++-compiler is called `g++`.
One can fetch the Intel C/C++-compiler `icc` as well.

A few words on the unix path

The location of a file or a directory is given by its path.
An absolute path starts at the root in the file tree.
The root is denoted `/` (slash). The path to the `HPC`-directory is `/chalmers/users/thomas/HPC`. The file `ex.f90`, in this directory, has the path `/chalmers/users/thomas/HPC/ex.f90`.
There are also relative paths.

Suppose the current directory is `/chalmers/users/thomas`. A path to the `ex.f90` is then `HPC/ex.f90`. Suppose your current directory is something else, then `~thomas/HPC/ex.f90` is a path to the file. `~`, by itself, denotes your home directory, `~user`, is the path to the home directory of `user`.
So I could have written, `~/HPC/ex.f90`.
`..` is the level above, and `.` is the current directory. That is why we sometimes write `./a.out`, se below.

The shell (`csh`, `tcsh`, `sh`, `ksh`, `bash`, ...) keeps several variables. One important such variable is the `path`. I will concentrate on `[t]csh`. The path contains a blank-separated list of directories. When you type a command (which is not built-in, such as `cd`) the shell will search for a directory containing the command (an executable file with the given name). If the shell finds the command it will execute it, if not, it will complain:

```
% set path = ( )          no directories
% cd HPC                  cd is built-in
% ls
ls: Command not found.
% /bin/ls                 works
A.mat    ... etc

% set path = ( /bin )
% ls                      now tcsh finds ls
A.mat    ... etc
```

The **set** is local to the particular shell and lasts only the present login session.

Sometimes there are several different versions of a command. The shell will execute the command it finds first (from left to right).

```
% which ls
/bin/ls

% which gfortran
/usr/bin/gfortran        comes with the system

% which gfortran         used in the course 2006
/chalmers/users/thomas/HPC/gfortran/bin/gfortran
```

In the first **which**, **/usr/bin** comes before the HPC-directory, and in the second **/usr/bin** comes after.
If you do not have **.** in your **path**, the shell will not look for executables in the current directory.

```
% pwd                              print current directory
/chalmers/users/thomas/HPC/Test

% a.out
a.out: Command not found.      no . in the path
% ./a.out                      works
% set path = ( $path . )       add . to the path
% a.out                        works
```

$path is the value of **path**. Suppose the **path** contains **~** .

```
% cp a.out ~/a.out1
% which a.out1
a.out1: Command not found.

% rehash            rebuild the internal hash table
% which a.out1
/chalmers/users/thomas/a.out1
```

A command does not have to be a compiled program.

```
% ls -l /bin/ls
-rwxr-xr-x  1 root root 82796 Jun 20 13:52 /bin/ls

% file /bin/ls
/bin/ls: ELF 32-bit LSB executable, Intel 80386,
version 1 (SYSV), for GNU/Linux 2.2.5,
dynamically linked (uses shared libs), stripped

% which cd
cd: shell built-in command.

% which apropos
/usr/bin/apropos
% file /usr/bin/apropos
/usr/bin/apropos: Bourne shell script text executable
% head -3 /usr/bin/apropos
#!/bin/sh
#
# apropos -- search the whatis database for keywords.
```

A user would usually (perhaps not if one is a student; see below for more details) set the **path**-variable in the startup file **.tcshrc** which usually resides in the login directory. The period in the name makes the file invisible. Type **ls -a** to see the names of all the dot-files.

To see your path, type **echo $path**, or give the command **set**, which prints all the shell variables.

Shell-variables are not exported to sub-processes so the shell creates an environment variable, **PATH**, as well. **PATH** is exported to sub-processes and it contains a **:**-separated list of directories).

```
% set var = hello
% echo $var        like print
hello
% tcsh             start a sub-shell
% echo $var
var: Undefined variable.
% exit

% setenv var hello    an environment variable, no =
% tcsh                sub-shell
% echo $var
hello
```

To see all your environment variables, type **setenv**. Another useful environment variable is the manual search path, **MANPATH** and the **LD_LIBRARY_PATH**(much more details later on).

A note on the student environment. To make it easier for beginners (both teachers and students) Medic has set up a standard environment where you do not have to create your own startup files. One does not have to use it (I don't), but if you do this is how you can modify your environment. Edit (create, the first time) the file **.vcs4/tcshrc** on your login-level. Here is sample file:

```
setenv LD_LIBRARY_PATH .
alias m less
set prompt = '> '
set path = ( $path . )
```

It sets an environment variable. An alias is created so that one type **m** instead of **less** (**less** is a so called pager, very useful). I do not like standard prompt so I have set it to **>** . The last row appends **.** to the path. Be careful with the last one!

In order to test that the changes work one can logout and then login. This is a bit slow, and if one has made a mistake in tcshrc it may be impossible to login. One should then be able to do a failsafe login and correct the mistake.

Less time consuming, and safer, is to open a new terminal window (xterm or gnome). Each time a new terminal is opened a shell is created as a subprocess under the terminal and the shell reads the startup file. One can see if the changes work in the new window.

Note that changes in tcshrc do not affect the already existing shells (windows). If one does not use the standard environment it is easy to update the existing shells as well.

More on * and & in C

In main we have reserved storage for the variable sum. The variable has an address in memory. Let us add the following two statements after the declaration of sum:

```
 double *p;         /* p is a pointer to double */
 p = &sum;          /* p = address of sum       */
```

& gives the address of a variable, so the second line assigns the address of sum to the pointer variable p, p points to sum.

Given p we can get the value of sum by using the indirection (or dereferencing) operator, *. Thus *p is the value of sum.

This explains the first line, the declaration of p. It says that *p is a double, so that p must be a pointer to double.

We can now understand how the parameters are passed in the call, sum_array(a, &sum, n); The address of sum, &sum, is passed as a parameter to the function.
Inside the function it is possible to access (read and write) the memory where sum is residing, since the function has been given the address.

Inside the function sum equals the address (it is equivalent to the pointer p above). Since we have passed the address (and not the value) we must use * to access the value itself (and not the address).

The variable n is copied to the function. If we change n inside the function the original n (in main) will not be changed.

The compiler will pass the address of the first element, &a[0], when we write the name of the array. So the function can access all the elements in the array.

If-statements and logical expressions

```
double precision :: a, b, c, d
logical          :: q

if( a < b .and. c == d .or. .not. q ) then
 ...  zero or more statements
else
 ...  zero or more statements
end if

double a, b, c, d;
int q;

if( a < b && c == d || !q ) {
 ...  zero or more statements
} else {
 ...  zero or more statements
}
```

Operation	Fortran77	C	Fortran90
$<$	.lt.	<	<
$\leq$	.le.	<=	<=
$=$	.eq.	==	==
$\neq$	.ne.	!=	/=
$\geq$	.ge.	>=	>=
$>$	.gt.	>	>
and	.and.	&&	.and.
or	.or.	||	.or.
not	.not.	!	.not.
true	.true.	$\neq$ 0	.true.
false	.false.	0	.false.

Note: if (! q == 2) ⇔ if ((!q) == 2),
not if(! (q == 2)).
Look at the predence table at the end of this handout.

Include (header) files

The cc-command first runs the C preprocessor, cpp. cpp looks for lines starting with # followed by a directive (there are several). From the man-page for cpp:

```
#include "filename"
#include <filename>
```

Read in the contents of filename at this location. This data is processed by cpp as if it were part of the current file. When the <filename> notation is used, filename is only searched for in the standard “include” directories. Can tell cpp where to look for files by using the -I-option.

A typical header file contains named constants, macros (somewhat like functions) and function protypes, e.g:

```
#define M_PI   3.14159265358979323846  /* pi */
#define __ARGS(a)       a
extern int MPI_Send __ARGS((void*, int, MPI_Datatype, i
```

It is common to store several versions of a program in one file and to use cpp to extract a special version for one system.

In _ompc_init from Omni, a Japanese implementation of OpenMP:

```
...
#ifdef OMNI_OS_SOLARIS
    lnp = sysconf(_SC_NPROCESSORS_ONLN);
#else
#ifdef OMNI_OS_IRIX
    lnp = sysconf(_SC_NPROC_ONLN);
#else
#ifdef OMNI_OS_LINUX
       ...  deleted code
```

Under Linux we would compile by:

```
  cc -DOMNI_OS_LINUX ...
```

A few words about the C99 standard

Note that it is not supported by all compilers.
C99 extends the previous C-version, C89, and adds support for (among other things):

- a boolean data type, complex numbers
- intermingled declarations and code
- //-comments
- inline functions
- variable-length arrays
- restrict qualifier to allow more aggressive code optimization (more later on)

Here a few lines showing how to use the boolean data type:

```
#include <stdbool.h>
 ...
  bool b;

  b = a > b;
  b = true;
  b = false;
 ...
```

Here complex numbers:

```
#include <complex.h>
 ...
  double complex z, w, wz;

  z = 1 + 2 * I;
  w = 3 + 4 * I;
  wz = 3 * w * z;

  printf("%e %e\n", creal(wz), cimag(wz));
 ...
```

Intermingled declarations and code:

```
#include <stdio.h>

int main()
{
  int k = 22;

  for(int k = 0; k <= 2; k++)  // C++ declaration style
    printf("%d\n", k);

  printf("%d\n", k);

  return 0;
}

% gcc -std=c99 c99_mixed.c   NOTE
% a.out
0
1
2
22
```

Inline functions. From the C-standard:

```
  inline double func(double x)
    ...
```

> Making a function an inline function suggests that calls to the function be as fast as possible. The extent to which such suggestions are effective is implementation-defined.

Variable-length arrays:

```
...
double f(int, double []);
double g(double [], int);

int main()
{
  int  n = 100;
  double vec[n];
...
}

double f(int n, double vec[n])
{
  double tmp[n];
 ...
}

double g(double vec[n], int n) // does not work
{                              // n is undefined
 ...
}
```

Using the man-command

```
% man sin
SIN(3) Linux Programmer's Manual SIN(3)


NAME
   sin, sinf, sinl - sine function

SYNOPSIS
   #include <math.h>

   double sin(double x);

   float sinf(float x);

   long double sinl(long double x);

DESCRIPTION
   The sin() function returns the sine of x,
   where x is given in radians.

RETURN VALUE
   The sin() function returns a value between -1 and 1.

CONFORMING TO
   SVID 3, POSIX, BSD 4.3, ISO 9899. The float and
   the long double variants are C99 requirements.

SEE ALSO
   acos(3), asin(3), atan(3), atan2(3), cos(3), tan(3)
```

You will not find man-pages for everything.
Can try to make a keyword search: `man -k keyword`

Some useful C-tools

`indent` and `cb` (Sun) are pretty printers for C.
`indent file.c`saves the old file in `file.c~`. If the C-program contains syntax errors `indent` can "destroy" it.
I use `indent -i2 -kr -nut file.c`

`lint` is a C program checker (not Linux). From the manual:

lint detects features of C program files which are likely to be bugs, non-portable, or wasteful.

lint also checks type usage more strictly than the compiler. lint issues error and warning messages. Among the things it detects are:

- Unreachable statements
- Loops not entered at the top
- Automatic variables declared and not used
- Logical expressions whose value is constant.

lint checks for functions that return values in some places and not in others, functions called with varying numbers or types of arguments, and functions whose values are not used or whose values are used but none returned.

Using `lint` on our program gives:

```
% lint prog.c -lm

function returns value which is always ignored
    printf              scanf

declared global, could be static
    sum_array           lint.c(37)
    ddot                lint.c(58)
```

Add `static`, `static void sum_array`makes `sum_array` local to the file (`prog.c`) it is defined in.

lint is not available on Linux systems. Here are two alternatives. Ask the compiler to be more careful:

```
 gcc -Wall  programs ...
```

(there are several other **W**-options). No complaints on my example. Much more details can be obtained from **splint** (**www.splint.org**).

You may want to start with **splint -weak**

```
% splint -weak ex.c
Splint 3.1.1 --- 15 Jun 2004

ex.c: (in function main)
ex.c:23:5: Assignment of int to double: a[k] = k + 1
  To allow all numeric types to match, use +relaxtypes.
ex.c:24:17: Function sin expects arg 1 to be double
           gets int: k + 1
Finished checking --- 2 code warnings
```

These are warnings and not errors, but they are still worth to check. By using a type cast we can make **splint** quiet.

```
a[k] = (double) k + 1;  // or (double) (k + 1)
b[k] = -sin((double) k + 1);
```

Since we have a prototype for **sin** (from **math.h**) there was an automatic cast to double in the first version of the code. Just typing **splint ex.c** will lead to more complaints, e.g. that the return status from **scanf** is ignored. They can be fixed by taking care of the status, or explicitly ignoring it:

```
  (void) scanf("%d", &in);
```

Having made the functions local to the file (**static**) there remains a few comments, but they cannot be fixed unless one adds so-called annotations (extra information in the form of special comments) to the code. Another alternative is to switch of a groups of warnings. **splint -strict ex.c** gives even more warnings.

A common Fortran construction

Fortran77 does not have dynamic memory allocation (like Fortran90 and C). If you need an **m** by **n** matrix **A** you would usually reserve space for the largest matrix you may need (for a particular application). If you pass the matrix as an argument to a procedure the procedure must be told about the extent of the first dimension (the number of rows) in order to be able to compute the address of an element. If the maximum number of rows is **max_m** the address, **adr( )**, of **A(j, k)** is given by

```
 adr(A(j, k)) = adr(A(1, 1)) + max_m*(k - 1) + j - 1
```

So, a matrix is stored by columns in Fortran. In C it is stored by rows (so the compiler must know the number of columns in the matrix). Since you can allocate the precise number of elements in C this is less of an issue.

A program may look like this:

```
  integer            :: m, n
  integer, parameter :: max_m = 1000, max_n = 50
  double precision, dimension(max_m, max_n) :: A

  call sub ( A, max_m, m, n )  ! m, n actual sizes
end
subroutine sub ( A, max_m, m, n )
  integer :: max_m, m, n
  double precision, dimension(max_m,*) ::  A
  ... ! can have 1 instead of *
```

Better (but not necessary) is:

```
  call sub ( A, max_m, max_n, m, n )
  ...
  subroutine sub ( A, max_m, max_n, m, n )
  integer :: max_m, m, n
  double precision, dimension(max_m, max_n) :: A
```

since index checks can be made by the compiler.

Part of the manual page for the Lapack routine **dgesv**:

```
NAME
 dgesv - compute the solution to a real system
         of linear equations A * X = B,

SYNOPSIS
 SUBROUTINE DGESV(N, NRHS, A, LDA, IPIVOT, B, LDB, INFO

 INTEGER N, NRHS, LDA, LDB, INFO
 INTEGER IPIVOT(*)
 DOUBLE PRECISION A(LDA,*), B(LDB,*)
 ...
ARGUMENTS
 N (input) The number of linear equations, i.e., the
           order of the matrix A. N >= 0.

 NRHS (input)
           The number of right hand sides, i.e.,  the
           number of columns of the matrix B. NRHS >= 0

 A (input/output)
           On entry, the N-by-N coefficient matrix A.
           On exit, the factors L and U from the
           factorization A = P*L*U; the unit diagonal
           elements of L are not stored.

 LDA (input)
           The leading dimension of the array A.
           LDA >= max(1,N).
...
```

It is possible to construct a nicer interface in Fortran90 (C++). Essentially **subroutine gesv(A, B, ipiv, info)** where **gesv** is polymorphic, (for the four types **S**, **D C**, **Z**) and where the size information is included in the matrices.

Array operations for Fortran90

```
program array_example
  implicit none

  ! works for other types as well
  integer                       :: k
  integer, dimension(-4:3)      :: a     ! Note -4
  integer, dimension(8)         :: b, c  ! Default 1:8
  integer, dimension(-2:3, 3) :: M


  a = 1 ! set all elements to 1
  b = (/ 1, 2, 3, 4, 5, 6, 7, 8 /) ! constant array
  b = 10 * b

  c(1:3) = b(6:8)
  print*, 'size(a), size(c)     = ', size(a),   size(c)
  print*, 'lbound(a), ubound(c) = ',lbound(a),ubound(a)
  print*, 'lbound(c), ubound(c) = ',lbound(c),ubound(c)

  c(4:8) = b(8:4:-1) ! almost like Matlab
  print*, 'c = ', c  ! can print a whole array

  print*, 'minval(c) = ', minval(c) ! a builtin func.
  a = a + b * c                     ! elementwise *
  print*, 'a = ', a
  print*, 'sum(a) = ', sum(a)       ! another builtin
  M = 0
  M(1, :) = b(1:3) ! Row with index one
  print*, ’M(1, :) = ’, M(1, :)

  M(:, 1) = 20     ! The first column
  where ( M == 0 ) ! instead of two loops
    M = -1
  end where

  print*, ’lbound(M) = ’, lbound(M) ! an array

  do k = lbound(M, 1), ubound(M, 1) ! print M
    print ’(a, i2, a, i2, 2i5)’, ’ M(’, k, ’, :) = ’, &
           M(k, :)
  end do
end

% ./a.out
 size(a), size(c)     =  8 8
 lbound(a), ubound(c) =  -4 3
 lbound(c), ubound(c) =  1 8
 c =  60 70 80 80 70 60 50 40
 minval(c) =  40
 a =  601 1401 2401 3201 3501 3601 3501 3201
 sum(a) =  21408
 M(1, :) =  10 20 30
 lbound(M) =  -2 1
 M(-2, :) = 20   -1   -1
 M(-1, :) = 20   -1   -1
 M( 0, :) = 20   -1   -1
 M( 1, :) = 20   20   30
 M( 2, :) = 20   -1   -1
 M( 3, :) = 20   -1   -1
```

Some dangerous things

```
  for(k = 0; k < n; k++)
    a[k] = b[k] + c[k];
    e[k] = f[k] * g[k];
```

is not the same as

```
  for(k = 0; k < n; k++) {
    a[k] = b[k] + c[k];
    e[k] = f[k] * g[k];
  }
```

Similarly for if-statements.

```
  if ( j = 0 ) printf("j is equal to zero\n");

  /* while a valid char... */
  while ((c = getchar()) != EOF )  { ...

  k == 3;
```

Actual and formal parameters lists: check position, number and type. Can use interface blocks (“prototypes”).

```
program main
  double precision :: a, b

  a = 0.0
  call sub(a, 1.0, b)
  print*, a, b
end
subroutine sub(i, j)
  integer :: i, j

  i = i + 1
  j = 10.0
end
```

```
% a.out
Segmentation fault  % result depends on the compiler
```

Remove the line `j = 10.0` and run again:

```
% a.out
     2.1219957909653-314  0.  % depends on the compiler
```

C- and Fortran compilers do not usually check array bounds.

```
void sub(double a[]);

#include <stdio.h>
main()
{
        double          b[1], a[10];

        b[0] = 1;
        sub(a);
        printf("%f\n", b[0]);
}

void
sub(double a[])
{
        a[11] = 12345.0;
}
```

Running this program we get:

```
% a.out
12345.000000
```

Changing `a[10]` to `a[1000000]` gives Segmentation fault.

Some Fortran-compilers can check subscripts (provided you do not lie):

```
program main
  double precision, dimension(10) :: a

  call lie(a)
  print*, ’a(1) = ’, a(1)

end program main
subroutine lie(a)
  double precision, dimension(10) :: a

  do j = 1, 100  !!! NOTE
    a(j) = j
  end do

end subroutine lie

% ifort -CB lie.f90
% ./a.out
forrtl: severe (408): fort: (2): Subscript #1 of the
array A has value 11 which is greater than the upper
 bound of 10


Change dimension(10)   to
       dimension(100)  in lie

% ifort -CB lie.f90
% a.out
 a(1) =     1.0000000000000
```

Precedence and associativity of C-operators

Operators have been grouped in order of decreasing precedence, where operators between horizontal lines have the same precedence.

Operator	Meaning	Associativity
`( )`	function call	→
`[ ]`	vector index	
`->`	structure pointer	
`.`	structure member	
`++`	postfix increment	
`-`	postfix decrement	
`!`	logical negation	←
`~`	bitwise negation	
`++`	prefix increment	
`--`	prefix decrement	
`+`	unary addition	
`-`	unary subtraction	
`*`	indirection	
`&`	address	
`(type)`	type cast	
`sizeof`	number of bytes	
`*`	multiplication	→
`/`	division	
`%`	modulus	
`+`	binary addition	→
`-`	binary subtraction	
«	left shift	→
»	right shift	
`<`	less than	→
`<=`	less or equal	
`>`	greater than	
`>=`	greater or equal	
`==`	equality	→
`!=`	inequality	

Operator	Meaning	Associativity
`&`	bitwise and	→
`^`	bitwise xor	→
`|`	bitwise or	→
`&&`	logical and	→
`||`	logical or	→
`?:`	conditional expression	←
`=`	assignment	←
`+=`	combined assignment and addition	
`-=`	combined assignment and subtraction	
`*=`	combined assignment and multiplication	
`/=`	combined assignment and division	
`%=`	combined assignment and modulus	
`&=`	combined assignment and bitwise and	
`^=`	combined assignment and bitwise xor	
`|=`	combined assignment and bitwise or	
«=	combined assignment and left shift	
»=	combined assignment and right shift	
`,`	comma	→

Here are a few comments, see a textbook or my links for a complete description.

- Left to right associativity (→) means that `a-b-c` is evaluated as `(a-b)-c` and not `a-(b-c)`. `a = b = c`, on the other hand, is evaluated as `a = (b = c)`. Note that the assignment `b = c` returns the value of `c`.

  `if ( a < b < c ) ...;` means `if ( (a < b) < c ) ...;` where `a < b` is 1 (true) if `a < b` and 0 (false) otherwise. This number is then compared to c. The statement does not determine "if b is between a and c".

- `a++;` is short for `a = a + 1;`, so is `++a;`. Both `a++` and `++a` can be used in expressions, e.g. `b = a++;`, `c = ++a`;. The value of `a++;` is `a`'s value before it has been incremented and the value of `++a;` is the new value.
- `a += 3;` is short for `a = a + 3;`.
- As in many languages, integer division is exact (through truncation), so `4 / 3` becomes 1. Similarly, `i = 1.25;`, will drop the decimals if `i` is an integer variable.
- `expr1 ? expr2 : expr3`equals `expr2` if `expr1` is true, and equals `expr3`, otherwise.
- `(type)` is used for type conversions, e.g. `(double) 3` becomes `3.0` and `(int) 3.25` is truncated to `3`.
- `sizeof(type_name)` or `sizeof expression` gives the size in bytes necessary to store the quantity. So, `sizeof(double)` is 8 on the Sun system and `sizeof (1 + 2)` is 4 (four bytes for an integer).
- When two or more expressions are separated by the comma operator, they evaluate from left to right. The result has the type and value of the rightmost expression. In the following example, the value 1 is assigned to `a`, and the value 2 is assigned to `b`. `a = b = 1, b += 2, b -= 1;`
- Do not write too tricky expressions. It is easy to make mistakes or to end up with undefined statements.
  `a[i++] = i;` and `i = ++i + 1;` are both undefined.
  See the standard, section 6.5, if you are interested in why.

Precedence of Fortran 90-operators

Operators between horizontal lines have the same precedence.

Operator	Meaning
unary user-defined operator	
`**`	power
`*`	multiplication
`/`	division
`+`	unary addition
`-`	unary subtraction
`+`	binary addition
`-`	binary subtraction
`//`	string concatenation
`==   .EQ.`	equality
`/=   .NE.`	inequality
`<    .LT.`	less than
`<=   .LE.`	less or equal
`>    .GT.`	greater than
`>=   .GE.`	greater or equal
`.NOT.`	logical negation
`.AND.`	logical and
`.OR.`	logical or
`.EQV.`	logical equivalence
`.NEQV.`	logical non-equivalence
binary user-defined operator	

Comments:
`==` is the Fortran90 form and `.EQ.` is the Fortran77 form, etc.
In Fortran90 lower case is permitted, .e.g `.not.` .

About the user defined operators. In Fortran90 it is possible to define ones own operators by overloading existing operators or by creating one with the name `.name.` where `name` consists of at most 31 letters.

C can be very hard to read. An example from the 3rd International Obfuscated C Code Contest (this program is by one of the winners, Lennart Augustsson).
See **`http://www.ioccc.org/`**for more examples.

```
typedef struct n{int a:3,
b:29;struct n*c;}t;t*
f();r(){}m(u)t*u;{t*w,*z;
z=u->c,q(z),u->b=z->b*10,
w=u->c=f(),w->a=1,w->c=z->
c;}t*k;g(u)t*u;{t*z,*v,*p,
*x;z=u->c,q(z),u->b=z->b,v
=z->c,z->a=2,x=z->c=f(),x
->a=3,x->b=2,p=x->c=f(),p
->c=f(),p->c->a=1,p->c->c=
v;}int i;h(u)t*u;{t*z,*v,*
w;int c,e;z=u->c,v=z->c,q(
v),c=u->b,e=v->b,u->b=z->b
,z->a=3,z->b=c+1,e+9>=c&&(
q(z),e=z->b,u->b+=e/c,w=f(
),w->b=e%c,w->c=z->c,u->c=
w);}int(*y[4])()={r,m,g,h};
char *sbrk();main(){t*e,*p,*o;
o=f(),o->c=o,o->b=1,e=f(),
e->a=2,p=e->c=f(),p->b=2,
p->c=o,q(e),e=e->c,(void)write
(1,"2.",2);for(;;e=e->c){q(e),
e->b=write(1,&e->b["0123456789"],
1);}}t*f(){return i||(i=1000,
k=(t*)sbrk(i*sizeof(t))),k+--i;
}q(p)t*p;{(*y[p->a])(p);}

% a.out
2.718281828459045235360287471352662497757247093699
959574966967627724076630353547594571382178525166427
427466391932003059921817413596629043572900334295260
595630738132328627943490763233829880753195251019011
```

Using make

Make keeps track of modification dates and recompiles the routines that have changed.

Suppose we have the programs **`main.f90`** and **`sub.f90`** and that the executable should be called **`run`**. Here is a simple makefile (it should be called **`Makefile`** or **`makefile`**):

```
run: main.o sub.o
     g95 -o run main.o sub.o

main.o: main.f90
     g95 -c main.f90

sub.o: sub.f90
     g95 -c sub.f90
```

A typical line looks like:

```
target: files that the target depends on
^Ia rule telling make how to produce the target
```

Note the tab character. Make makes the first target in the makefile. **`-c`** means compile only (do not link) and **`-o`** gives the name of the executable.

To use the makefile just give the command **`make`**.

```
% make
g95 -c main.f90
g95 -c sub.f90
g95 -o run main.o sub.o
```

To run the program we would type **`run`** .

If we type **`make`** again nothing happens (no file has changed):

```
% make
'run' is up to date.
```

Now we edit **`sub.f90`** and type **`make`** again:

```
% make
g95 -c sub.f90
g95 -o run main.o sub.o
```

Note that only **`sub.f90`** is compiled. The last step is to link **`main.o`** and **`sub.o`** together (**`g95`** calls the linker, **`ld`**).

Writing makefiles this way is somewhat inconvenient if we have many files. **`make`** may have some builtin rules, specifying how to get from source files to object files, at least for C. The following makefile would then be sufficient:

```
run: main.o sub.o
     g95 -o run main.o sub.o
```

Fortran90 is unknown to some make-implementations and on the student system one gets:

```
% make
make: *** No rule to make target 'main.o',
          needed by 'run'.  Stop.
```

We can fix that by adding a special rule for how to produce an object file from a Fortran90 source file.

```
run: main.o sub.o
     g95 -o run main.o sub.o

.SUFFIXES: .f90
.f90.o:
        g95 -c $<
```

`$<`, a so called macro, is short for the Fortran file.
One can use variables in make, here **`OBJS`** and **`FFLAGS`**.

```
OBJS   = main.o sub.o
FFLAGS = -O3

run: $(OBJS)
        g95 -o run $(FFLAGS) $(OBJS)

.SUFFIXES: .f90
.f90.o:
        g95 -c $(FFLAGS) $<
```

`OBJS` (for objects) is a variable and the first line is an assignment to it. **`$(OBJS)`** is the value (i.e. **`main.o sub.o`**) of the variable **`OBJS`**. **`FFLAGS`** is a standard name for flags to the Fortran compiler. I have switched on optimization in this case. Note that we have changed the suffix rule as well.

Make knows about certain variables, like **`FFLAGS`**. Suppose we would like to use the **`ifort`**-compiler instead. When compiling the source files, make is using the compiler whose name is stored in the variable **`FC`** (or possible **`F90`** or **`F90C`**). We write:

```
OBJS   = main.o sub.o
FC     = ifort
FFLAGS = -O3

run: $(OBJS)
        $(FC) -o run $(FFLAGS) $(OBJS)

.SUFFIXES: .f90
.f90.o:
        $(FC) -c $(FFLAGS) $<
```

It is usually important to use the same compiler for compiling and linking (or we may get the wrong libraries). It may also be important to use the same Fortran flags.

Sometimes we wish the recompile all files (we may have changed `$(FFLAGS)` for example). It is common to have the target `clean`. When having several targets we can specify the one that should be made:

```
OBJS   = main.o sub.o
FC     = g95
FFLAGS = -O3

run: $(OBJS)
     $(FC) -o run  $(FFLAGS) $(OBJS)

# Remove objects and executable
clean:
     rm -f $(OBJS) run

.SUFFIXES: .f90
.f90.o:
        $(FC) -c $(FFLAGS) $<
```

Without `-f`, `rm` will complain if some files are missing.

We type:

```
% make clean
rm -f main.o sub.o run
```

Suppose we like to use a library containing compiled routines. The new makefile may look like:

```
OBJS   = main.o sub.o
FC     = g95
FFLAGS = -O3
LIBS   = -lmy_library

run: $(OBJS)
     $(FC) -o run  $(FFLAGS) $(OBJS) $(LIBS)

.SUFFIXES: .f90
.f90.o:
        $(FC) -c $(FFLAGS) $<
```

If you are using standard functions in C sin, exp etc. you must use the math-library:

```
  cc ... -lm
```

The equivalent makefile for C-programs looks like:

```
OBJS   = main.o sub.o
CC     = cc
CFLAGS = -O3
LIBS   = -lmy_library -lm

run: $(OBJS)
     $(CC) -o run  $(CFLAGS) $(OBJS) $(LIBS)

clean:
     rm -f $(OBJS) run
```

For the assignments it is easiest to have one directory and one makefile for each. It is also possible to have all files in one directory and make one big makefile.

```
OBJS1  = main1.o sub1.o
OBJS2  = main2.o sub2.o
CC     = cc
CFLAGS = -O3
LIBS1  = -lm
LIBS2  = -lmy_library

all: prog1 prog2

prog1: $(OBJS1)
     $(CC) -o $@  $(CFLAGS) $(OBJS1) $(LIBS1)

prog2: $(OBJS2)
     $(CC) -o $@  $(CFLAGS) $(OBJS2) $(LIBS2)

clean:
     rm -f $(OBJS1) $(OBJS2) prog1 prog2
```

When one is working with (and distributing) large projects it is common to use make in a recursive fashion. The source code is distributed in several directories. A makefile on the top-level takes care of descending into each sub-directory and invoking make on a local makefile in each directory.

There is much more to say about make. See e.g. the O'Reilly-book, Robert Mecklenburg, Managing Projects with GNU Make, 3rd ed, 2004.

Computer Architecture

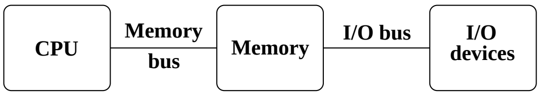
Why this lecture?

Some knowledge about computer architecture is necessary:

- to understand the behaviour of programs
- in order to pick the most efficient algorithm
- to be able to write efficient programs
- to know what computer to run on
  (what type of architecture is your code best suited for)
- to read (some) articles in numerical analysis
- when looking for the next computer to buy
  (and to understand those PC-ads., caches, GHz, RISC...)

The change of computer architecture has made it necessary to re-design software, e.g Linpack $\Rightarrow$ Lapack.

A very simple (and traditional) model of a computer:



The CPU contains the ALU, arithmetic and logic unit and the control unit. The ALU performs operations such as +, -, *, / of integers and Boolean operations.
The control unit is responsible for fetching, decoding and executing instructions.
The memory stores instructions and data. Instructions are fetched to the CPU and data is moved between memory and CPU using buses.
I/O-devices are disks, keyboards etc.

The CPU contains several registers, such as:

- PC, program counter, contains the address of the next instruction to be executed
- IR, instruction register, the executing instruction
- address registers
- data registers

The memory bus usually consist of one address bus and one data bus. The data bus may be 64 bits wide and the address bus may be $\geq$ 32 bits wide. With the introduction of 64-bit computers, buses tend to become increasingly wider. The Itanium 2 uses 128 bits for data and 44 bits for addresses.

Operations in the computer are synchronized by a clock.
A modern CPU may run at a few GHz (clock frequency). The buses are usually a few (4-5) times slower.

A few words on 64-bit systems

Why 64 bit?

- A larger address range, can address more memory.
  With 32 bits we can (directly) address 4 Gbyte, which is rather limited for some applications.
- Wider busses, increased memory bandwidth.
- 64-bit integers.

Be careful when mixing binaries (object libraries) with your own code. Are the integers 4 or 8 bytes?

```
% cat kind.c
#include <stdio.h>

int main()
{
  printf("sizeof(short int) = %d\n", sizeof(short int))
  printf("sizeof(int)       = %d\n", sizeof(int));
  printf("sizeof(long int)  = %d\n", sizeof(long int));

  return 0;
}

% gcc kind.c              On the student system
% a.out
sizeof(short int) = 2
sizeof(int)       = 4
sizeof(long int)  = 4

% a.out                  On another Opteron-system
sizeof(short int) = 2
sizeof(int)       = 4
sizeof(long int)  = 8    4 if gcc -m32
```

CISC (Complex Instruction Set Computers) before ≈ 1985.
Each instruction can perform several low-level operations, such as a load from memory, an arithmetic operation, and a memory store, all in a single instruction.

Why CISC?
For a more detailed history, see the literature.

- Advanced instructions simplified programming
  (writing compilers, assembly language programming).
  Software was expensive.
- Memory was limited and slow so short programs were good.
  (Complex instructions ⇒ compact program.)

Some drawbacks:

- complicated construction could imply a lower clock frequency
- instruction pipelines hard to implement
- long design cycles
- many design errors
- only a small part of the instructions was used
  According to Sun: Sun's C-compiler uses about 30% of the available 68020-instructions (Sun3 architecture). Studies show that approximately 80% of the computations for a typical program requires only 20% of a processor's instruction set.

When memory became cheaper and faster, the decode and execution on the instructions became limiting.

Studies showed that it was possible to improve performance with a simple instruction set and where instructions would execute in one cycle.

RISC - Reduced Instruction Set Computer

- IBM 801, 1979 (publ. 1982)
- 1980, David Patterson, Berkeley, RISC-I, RISC-II
- 1981, John Hennessy, Stanford, MIPS
- ≈ 1986, commercial processors

A processor whose design is based on the rapid execution of a sequence of simple instructions rather than on the provision of a large variety of complex instructions.

Some RISC-characteristics:

- load/store architecture; `C = A + B`

```
    LOAD  R1,A
    LOAD  R2,B
    ADD   R1,R2,R3
    STORE C,R3
```

- fixed-format instructions (the op-code is always in the same bit positions in each instruction which is always one word long)
- a (large) homogeneous register set, allowing any register to be used in any context and simplifying compiler design
- simple addressing modes with more complex modes replaced by sequences of simple arithmetic instructions
- one instruction/cycle
- hardwired instructions and not microcode
- efficient pipelining
- simple FPUs; only +, -, *, / and $\sqrt{}$.
  sin, exp etc. are done in software.

Advantages: Simple design, easier to debug, cheaper to produce, shorter design cycles, faster execution, easier to write optimizing compilers (easier to optimize many simple instructions than a few complicated with dependencies between each other).

CISC - short programs using complex instructions.
RISC - longer programs using simple instructions.

So why is RISC faster?

The simplicity and uniformity of the instructions make it possible to use pipelining, a higher clock frequency and to write optimizing compilers.

Will now look at some techniques used in all RISC-computers:

- instruction pipelining
  work on the fetching, execution etc. of instructions in parallel
- cache memories
  small and fast memories between the main memory and the CPU registers
- superscalar execution
  parallel execution of instructions (e.g. two integer operations, *, + floating point)

The most widely-used type of microprocessor, the x86 (Intel), is CISC rather than RISC, although the internal design of newer x86 family members is said to be RISC-like. All modern CPUs share some RISC characteristics, although the details may differ substantially.

Pipelining - performing a task in several steps, stages

Analogy: building cars using an assembly line in a factory.

Suppose there are five stages (can be more), .e.g

IF	Fetch the next instruction from memory.
ID	Instruction decode.
EX	Execute.
M, WM	Memory access, write to registers.

Clock cycle number

Instruction	1	2	3	4	5	6	7	8	9
k	IF	ID	EX	M	WB				
k+1		IF	ID	EX	M	WB			
k+2			IF	ID	EX	M	WB		
k+3				IF	ID	EX	M	WB	
k+4					IF	ID	EX	M	WB

So one instruction completed per cycle once the pipeline is filled.

Not so simple in real life: different kind of hazards, that prevent the next instruction from executing during its designated clock cycle. Can make it necessary to stall the pipeline (wait cycles).

- Structural hazards arise from resource conflicts, e.g.
  • two instructions need to access the system bus (fetch data, fetch instruction),
  • not fully pipelined functional units (division usually takes 10-20 cycles, for example).
- Data hazards arise when an instruction depends on the results of a previous instruction (will look at some cases in later lectures) e.g.

  ```
  a = b + c
  d = a + e   d depends on a
  ```

  The second addition must not start until `a` is available.
- Control hazards arise from the pipelining of branches (if-statements).

An example of a control hazard:

```
if ( a > b - c * d ) then
  do something
else
  do something else
end if
```

Must wait for the evaluation of the logical expression.

If-statements in loops may cause poor performance.

Several techniques to minimize hazards (look in the literature for details) instead of just stalling. Some examples:

Structural hazard:

Add hardware. If the memory has only one port `LOAD adr,R1` will stall the pipeline (the fetch of data will conflict with a later instruction fetch). Add a memory port (separate data and instruction caches).

Data hazards:

- Forwarding: `b + c` available after EX, special hardware "forwards" the result to the `a + e` computation (without involving the CPU-registers).
- Instruction scheduling. The compiler can try and rearrange the order of instruction to minimize stalls.
  Try to change the order between instructions using the wait-time to do something useful.

```
a = b + c
d = a + e

load b
load c
add  b + c  has to wait for load c to complete

load b
load c
load e      give the load c time to complete
add  b + c  in parallel with load e
```

Control hazards: (many tricks)

- Add hardware; can compute the address of the branch target earlier and can decide whether the branch should be taken or not.
- Branch prediction; try to predict, using "statistics", the way a branch will go. Compile-time/run-time. Can work very well. The branch att the end of a **for**-loops is taken all the times but the last.
- Branch delay slot: let the compiler rearrange instructions so that something useful can be done while it is decided whether we should branch or not.

```
loop: instr                    loop: instr
      instr                          instr
       ...                -->         ...
      FADD R1, R2, R3                if j < n goto loop
      if j < n goto loop             FADD R1, R2, R3
```

Perform the FADD-instruction while waiting for the if to complete.

A more general construction, speculative execution: Assume the branch not taken and continue executing (no stall). If the branch is taken, must be able do undo.

Superscalar CPUs

Fetch, decode and execute more than one instruction in parallel. More than one finished instruction per clock cycle. There may, e.g. be two integer ALUs, one unit for floating point addition and subtraction one for floating point multiplication. The units for +, - and * are usually piplined (they need several clock cycles to execute).
There are also units for floating point division and square root; these units are not (usually) pipelined.

```
MULT  xxxxxxxx
MULT   xxxxxxxx
MULT    xxxxxxxx
```

Compare division; each `xxxxxxxxxx` is 22 cycles (on Sun):

```
DIV  xxxxxxxxxx
DIV            xxxxxxxxxx
DIV                      xxxxxxxxxx
```

How can the CPU keep the different units busy?
The CPU can have circuits for arranging the instructions in suitable order, dynamic scheduling (out-of-order-execution).

To reduce the amount of hardware in the CPU we can let the compiler decide a suitable order. Groups of instructions (that can be executed in parallel) are put together in packages. The CPU fetches a whole package and not individual instructions. VLIW-architecture, Very Long Instruction Word.

The Intel & HP Itanium CPU uses VLIW (plus RISC ideas). Read the free chapter from: W. Triebel, Itanium Architecture for Software Developers. See the first chapter in: IA-32 Intel Architecture Optimization Reference Manual for details aboute the Pentium 4. Read Appendix A in the Software Optimization Guide for AMD64 Processors. See the web-Diary for links.

More on parallel on floating point operations.

flop = floating point operation.
flops = plural of flop or flop / second.
In numerical analysis a flop may be an addition-multiplication pair. Not unreasonable since (+, *) often come in pairs, e.g. in an inner product.

Top floating point speed =
# of processors × flop / s =
# of processors × # flop / clock cycle × clock frequency

On the student machines (AMD64) the theoretical top performance is $2 \cdot 10^9$ additions and multiplications per second. Note that the student machines are running PowerNow!, AMD's technology for dynamically changing the clock frequency of the CPU (to save energy, to reduce the rpm of the cooling fan etc). There are three possible clock frequencies: 2GHz, 1.8GHz and 1GHz. More details are available from the homepage.

An Intel Pentium4 can finish an addition every clock cycle and a multiplication every other.

Some Intel, AMD and Motorola CPUs have another unit (a kind of vector unit) that can work on short vectors of numbers. These technologies have names like: SSE3, 3DNow! and AltiVec. More details later in the course.

To use the the vector unit you need a compiler that can vectorize. The vector unit may not be IEEE 754 compliant (not correctly rounded). So results may differ between the vectorized and unvectorized versions of the same code.
See `www.spec.org` for benchmarks with real applications.

Why do we often only get a small percentage of these speeds? Is it possible to reach the top speed (and how)?

Example on a 167 MHz Sun; top speed 334 Mflops:

```
Instr.  fl. p. registers

fmuld   %f4,%f2,%f6                faddd   %f4,%f2,%f6
faddd   %f8,%f10,%f12              faddd   %f8,%f10,%f12
fmuld   %f26,%f28,%f30             faddd   %f4,%f2,%f6
faddd   %f14,%f16,%f18             faddd   %f8,%f10,%f12
fmuld   %f4,%f2,%f6                faddd   %f4,%f2,%f6
faddd   %f8,%f10,%f12              faddd   %f8,%f10,%f12
                          ...

331.6 Mflops                       166.1 Mflops
```

```
fdivd   %f4,%f2,%f6                fdivd   %f4,%f2,%f6
faddd   %f8,%f10,%f12              fdivd   %f8,%f10,%f12
fdivd   %f4,%f2,%f6                fdivd   %f4,%f2,%f6
faddd   %f8,%f10,%f12              fdivd   %f8,%f10,%f12
                          ...

15.1 Mflops                        7.5 Mflops
```

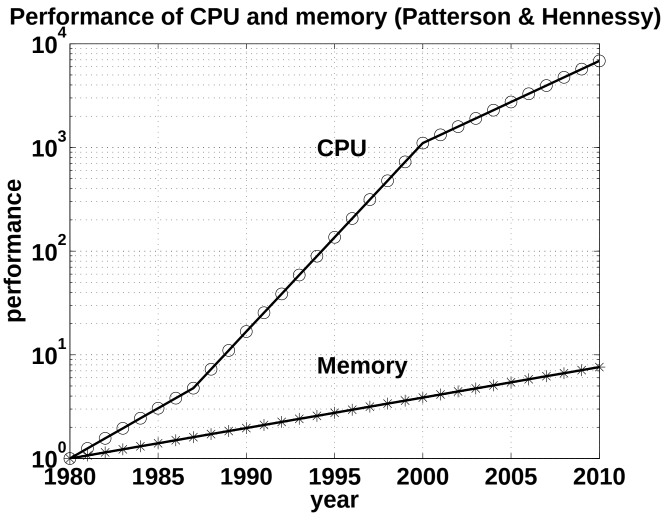
Addition and multiplication are pipelined. Division is not pipelined (so divides do not overlap) and takes 22 cycles for double precision.

$$\frac{167 \cdot 10^6 s^{-1}}{22} \approx 7.6 \cdot 10^6/s$$

So, the answer is sometimes

- provided we have a suitable instruction mix and that
- we do not access memory too often

Memory is the problem - caches

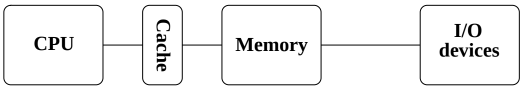


CPU: increase 1.35 improvement/year until 1986,
and a 1.55 improvement/year thereafter.

DRAM (dynamic random access memory), slow and cheap,
1.07 improvement/year.

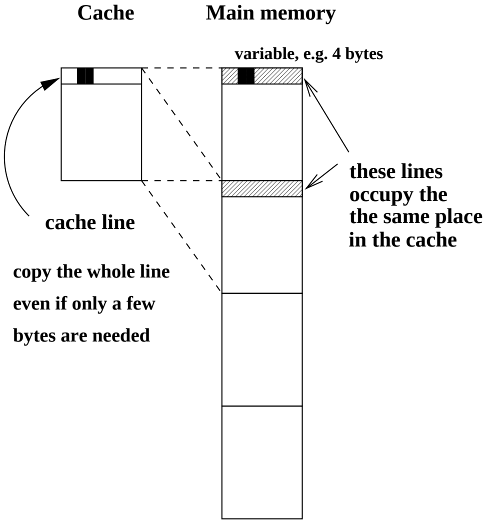
Use SRAM (static random access memory) fast & expensive for cache.

Direct mapped cache



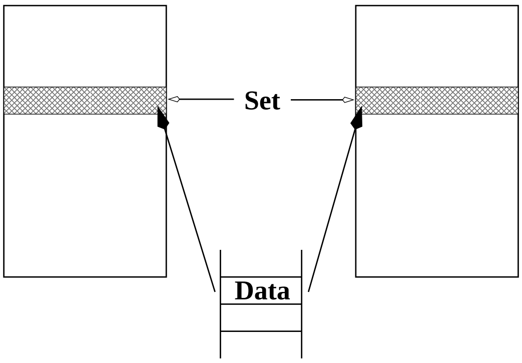
The cache is a small and fast memory used for storing both instructions and data.

This is the simplest form of cache-construction.



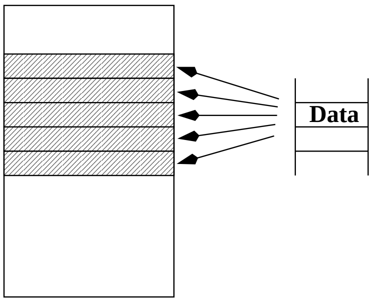
Cache
Main memory
variable, e.g. 4 bytes
these lines occupy the the same place in the cache
cache line
copy the whole line even if only a few bytes are needed

There are more general cache constructions.
This is a two-way set associative cache:



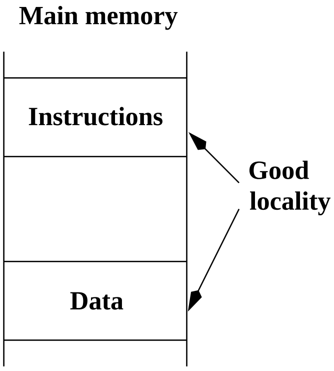
A direct mapped cache is one-way set associative.

In a fully associative cache data can be placed anywhere.



To use a cache efficiently locality is important.

- instructions: small loops, for example
- data: use part of a matrix (blocking)

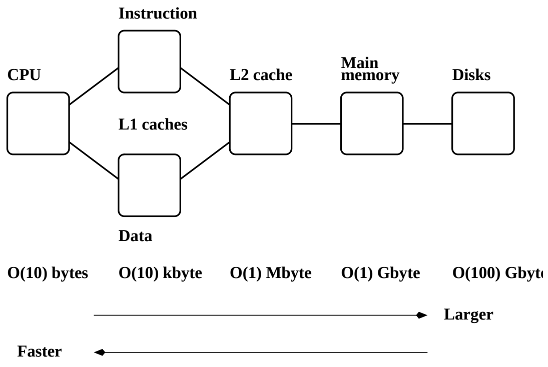


Not necessarily good locality together.

Make separate caches for data and instructions.

Can read instructions and data in parallel.

L1 and L2 caches



Memory hierarchy.

Newer machines even have an L3 cache.

The AMD64 (student machines)

(Some) Intel and AMD-CPUs have an instruction, **cpuid**, that gives details about the CPU, such as model, SSE-features, L1- and L2-cache properties. These values can be hard to find just reading manuals. Some parameters are available in **/proc/cpuinfo** .

Unfortunately one has to code in assembler to access this information. **gcc** supports inlining of assembly code using the **asm**-function. **asm** makes it possible to "connect" registers with C-variables. It may look like this (note that I have broken the string):

```
 ...
  unsigned long before, after;

  /* Does the CPU support the cpuid-instruction?*/
  asm("pushfl; popl %%eax; movl %%eax, %0;
       xorl $0x40000, %%eax; pushl %%eax; popfl; pushfl
       popl %%eax; movl %%eax, %1; pushl %0; popfl "
       : "=r" (before), "=r" (after)  /* output */
       :                              /* input  */
       : "eax"             /* changed registers */
     );
  if ( before != after ) {
    ...                   /* Support. Test more */
  }
 ...
```

One can call **cpuid** with a set of different "arguments", and **cpuid** then returns bit patterns in four registers. Reading the AMD-manual "CPUID Specification" one can interpret the bits. Bits 31-24 in the ECX-register contain the size of L1-data cache in kbyte, for example.

This is some of the facts I found out about the caches (I read some manuals as well):

The L1 data cache is 64 kbyte, 2-way associative and has a cache line size (length) of 64 bytes. Cache-line replacement is based on a least-recently-used (LRU) replacement algorithm.

The L2 cache is "on-die" (on-chip), 512 kbyte and 16-way associative. The cache line size is 64 bytes.

A note on reading assembly output

In the lecture and during the labs I said it was sometimes useful to look at the assembler code produced by the compiler.
Here comes a simple example. Let us look at the the following function.

```
double dot(double x[], double y[], int n)
{
  double s;
  int k;

  s = 0.0;
  for (k = 0; k < n; k++)
    s += x[k] * y[k];

  return s;
}
```

First some typical RISC-code from a Sun ULTRA-Sparc CPU.
I used **gcc** and compiled the code by:

```
    gcc -S -O3 dot.c
```

-S produces the assembler output on **dot.s**.
Here is the loop (code for passing parameters, setting up for the loop, and returning the result is not included).

```
.LL5:                           My translation
  ldd     [%o0+%g1], %f8     %f8  = x[k]
  ldd     [%o1+%g1], %f10    %f10 = y[k]
  add     %g2, 1, %g2        k    = k + 1
  fmuld   %f8, %f10, %f8     %f8  = %f8* %f10
  cmp     %o2, %g2           k == n? Set status reg.
  faddd   %f0, %f8, %f0      %f0  = %f0 + %f8
  bne     .LL5               if not equal, go to .LL5
  add     %g1, 8, %g1        increase offset
```

Some comments.

%f8 and **%f10** are registers in the FPU. When entering the function, the addresses of the first elements in the arrays are stored in registers **%o0** and **%o1**. The addresses of **x[k]** and **y[k]** are given by **%o0 + 8k** and **%o1 + 8k**. The reason for the factor eight is that the memory is byte addressable (each byte has an address). The offset, **8k**, is stored in register **%g1**.

The offset, **8k**, is updated in the last **add**. It looks a bit strange that the **add** comes after the branch, **bne**. The add-instruction is, however, placed in the branch delay slot of the branch-instruction, so it is executed in parallel with the branch.

add is an integer add. **faddd** is a "floating point add double". It updates **%f0**, which stores the sum. **%f0** is set to zero before the loop. **cmp** compares **k** with **n** (the last index) by subtracting the numbers. The result of the compare updates the Z-bit (Z for zero) in the integer condition code register. The branch instruction looks at the Z-bit to see if the branch should be taken or not.

We can make an interesting comparison with code produced on the AMD64. The AMD (Intel-like) has both CISC- and RISC-characteristics. It has fewer registers than the Sparc and it does not use load/store in the same way. The x87 (the FPU) uses a stack with eight registers. In the code below, **eax** etc. are names of 32-bit CPU-registers. (in the assembly language a **%** is added).

```
.L5:
  fldl    (%ebx,%eax,8)
  fmull   (%ecx,%eax,8)
  faddp   %st, %st(1)
  incl    %eax
  cmpl    %eax, %edx
  jne     .L5
```

When the loop is entered **%ebx** and **%ecx** contain the addresses of the first elements of the arrays. Zero has been pushed on the stack as well (corresponds to **s = 0.0**).

fldl (%ebx,%eax,8)loads a 64 bit floating point number. The address is given by **%ebx + %eax*8**. The number is pushed on the top of the stack, given by the stackpointer **%st**.

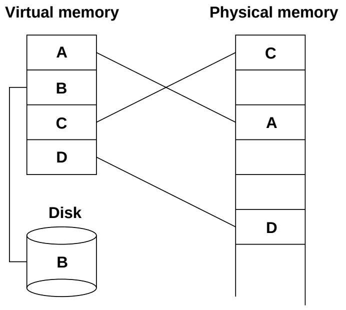
Unlike the Sparc, the AMD can multiply with an operand in memory (the number does not have to be fetched first). So the **fmull** multiplies the top-element on the stack with the number at address **%ecx + %eax*8** and replaces the top-element with the product.

faddp %st, %st(1)adds the top-elements on the stack (the product and the sum, **s**), pops the stack, the **p** in **faddp**, and replaces the top with the new value of **s**.

incl increases **k** (stored in **%eax**) and **cmpl** compares it to **n**. **jne** stands for jump if not equal.

Virtual memory

Use disk to "simulate" a larger memory. The virtual address space is divided into pages e.g. 4 kbytes. A virtual address is translated to the corresponding physical address by hardware and software; address translation.



A page is copied from disk to memory when an attempt is made to access it and it is not already present (page fault). When the main memory is full, pages must be stored on disk (e.g. the least recently used page since the previous page fault). Paging. (Swapping; moving entire processes between disk and memory.)

Some advantages of virtual memory:

- simplifies relocation (loading programs to memory), independece of physical addresses;
  several programs may reside in memory
- security, can check access to protected pages, e.g. read-only data; can protect data belonging to other processes
- allows large programs to run on little memory; only used sections of programs need be present in memory; simplifies programming (e.g. large data structures where only a part is used)

Virtual memory requires locality (re-use of pages) to work well, or thrashing may occur.

A few words on address translation

The following lines sketch one common address translating technique.

A virtual address is made up by two parts, the virtual page number and the page offset (the address from the top of the page).
The page number is an index into a page table:

```
physical page address =
    page_table(virtual page number)
```

The page table is stored in main memory (and is sometimes paged). To speed up the translation (accessing main memory takes time) we store part of the table in a cache, a translation lookaside buffer, TLB which resides in the CPU ($\mathcal{O}(10)$ – $\mathcal{O}(1000)$ entries).

Once again we see that locality is important. If we can keep the references to a few pages, the physical addresses can found in the TLB and we avoid a reference to main memory. If the address is not available in the TLB we get a TLB miss (which is fairly costly, taking tens of clock cycles).

Reading the actual data may require a reference to main memory, but we hope the data resides in the L1 cache.

Second best is the L2 cache, but we may have to make an access to main memory, or worse, we get a page fault and have to make a disk access (taking millions of clock cycles).

Contents

- How does one get good performance from a computer system?
- Focus on systems with one CPU and floating point performance.
- To get maximum performance from a parallel code it is important to tune the code running on each CPU.
- Not much about applications from graphics, audio or video. One example of SSE2 (Streaming SIMD Extensions 2).
- General advice and not specific systems.
- Fortran, some C (hardly any C++) and some Matlab. Some Java in the compendium.

Your situation

- A large and old code which has to be optimized. Even a slight speedup would be of use, since the code may be run on a daily basis.
- A new project, where language and data structures have to be chosen.

  C/C++ usually slower than Fortran for floating point.
  Java? Can be slow and use large amounts of memory.
  See the article (compendium) for an example.

  Should it be parallel?

  Test a simplified version of the computational kernel.
  Fortran for floating point, C/C++ for the rest.
- Things that are done once. Let the computer work.
  Unix-tools, Matlab, Maple, Mathematica ...

More about unix-tools:

- shell scripts (`sh`, `csh`, `tcsh`, `ksh`, `bash`) (`for`, `if`, `|` pipes and lots more)
- `awk` (developed by Alfred Aho, Peter Weinberger, and Brian Kernighan in 1978)
- `sed` (stream editor)
- `grep` (after the qed/ed editor subcommand "g/re/p", where re stands for a regular expression, to Globally search for the Regular Expression and Print)
- `tr` (translate characters)
- `perl` (Larry Wall in 1987; contains the above)
- etc.

Some very simple examples:

Counting the number of lines in a file(s):

```
% wc file     or    wc -l file
% wc files    or    wc -l files
```

Finding a file containing a certain string

```
% grep string files                e.g.
% grep 'program matrix' *.f90      or
% grep -i 'program matrix' *.f90   etc.
```

The `grep`-command takes many flags.

Example: interchange the two blank-separated columns of numbers in a file:

```
% awk '{print $2, $1}' file
```

Example: sum the second columns of a set of datatfiles. Each row contains: `number number text text`
The files are named `data.01`, `data.02`, ...

`foreach?` is the prompt.

```
% foreach f ( data.[0-9][0-9] )
foreach? echo -n $f': '
foreach? awk '{s += $2} END {print s}' $f
foreach? end
data.01: 30
data.02: 60
data.03: 77
data.20: 84
```

Another possibility is:

```
awk '{s += $2} END {print FILENAME ": " s}' $f
```

Just the other day (two years ago) ...

You have ≈ 600 files each consisting of ≈ 24000 lines (a total of ≈ $14 \cdot 10^6$ lines) essentially built up by:

```
<DOC>
<TEXT>
Many lines of text (containing no DOC or TEXT)
</TEXT>
</DOC>
<DOC>
<TEXT>
Many lines of text
</TEXT>
</DOC>
etc.
```

There is a mismatch between the number of `DOC` and `TEXT`.
Find it!

We can localize the file this way:

```
% foreach f ( * )
foreach? if ( ‘grep -c "<DOC>" $f‘ !=   \
              ‘grep -c "<TEXT>" $f‘ ) echo $f
foreach? end
```

Not so efficient; we are reading each file twice.
Takes ≈ 3.5 minutes.
We used binary search to find the place in the file.

The optimization process

Basic: Use an efficient algorithm.

Simple things:

- Use (some of) the optimization options of the compiler.
  Optimization can give large speedups
  (and new bugs, or reveal bugs).
  - Save a copy of the original code.
  - Compare the computational results before and after optimization.
    Results may differ in the last bits and still be OK.
- Read the manual page for your compiler.
  Even better, read the tuning manual for the system.
- Switch compiler and/or system.

The next page lists the compiler options of the Sun Fortran90/95-compiler. The names are not standardized, but it is common that `-c` means “compile only, do not link”.
To produce debug information `-g` is used.

`-O[n]` usually denotes optimization on level `n`. There may be an option, like `-fast`, that gives a combination of suitable optimization options. In Sun’s case `-fast` is equivalent to:

- `-xtarget=native`optimize for host system (sets cache sizes for example).
- `-O5` highest optimization level.
- `-libmil` inline certain math library routines.
- `-fsimple=2` aggressive floating-point optimizations. May cause many programs to produce different numeric results due to changes in rounding.
- `-dalign` align data to allow generation of faster double word load/store instructions.
- `-xlibmopt` link the optimized math library. May produce slightly different results; if so, they usually differ in the last bit.
- `-depend` optimize DO loops better.
- `-fns` for possibly faster (but nonstandard) handling of floating-point arithmetic exceptions and gradual underflow.
- `-ftrap=common`option to set trapping on common floating-point exceptions (this is the default for f95).
- `-pad=local`option to improve use of cache (alignment).
- `-xvector=yes`enable use of the vectorized math library.
- `-xprefetch=yes`enable generation of prefetch instructions on platforms that support it.

```
f95 | f90 [ -a ] [ -aligncommon[=a] ] [ -ansi ]
 [ -autopar ] [ -Bx ] [ -C ] [ -c ] [ -cg89 ] [ -cg92 ]
 [ -copyargs ] [ -Dnm[=def] ] [ -dalign ] [ -db ]
 [ -dbl_align_all[=yes|no] ] [ -depend ] [ -dryrun ]
 [ -d[y|n] ] [ -e ] [ -erroff=taglist ] [ -errtags[=yes
 [ -explicitpar ] [ -ext_names=e ] [ -F ] [ -f ]
 [ -fast ] [ -fixed ] [ -flags ] [ -fnonstd ]
 [ -fns=yes|no ] [ -fpover=yes|no ] [ -fpp ] [ -free ]
 [ -fround=r ] [ -fsimple[=n] ] [ -ftrap=t ] [ -G ]
 [ -g ] [ -hnm ] [ -help ] [ -Idir ] [ -inline=rl ]
 [ -Kpic ] [ -KPIC ] [ -Ldir ] [ -libmil ] [ -loopinfo
 [ -M dir ] [ -mp=x ] [ -mt ] [ -native ] [ -noautopar
 [ -nodepend ] [ -noexplicitpar ] [ -nolib ] [ -nolibmi
 [ -noqueue ] [ -noreduction ] [ -norunpath ] [ -O[n] ]
 [ -o nm ] [ -onetrip ] [ -openmp ] [ -p ] [ -pad[=p] ]
 [ -parallel] [ -pg ] [ -pic ] [ -PIC ] [ -Qoption pr l
 [ -qp ] [ -R list ] [ -r8const ] [ -reduction ] [ -S ]
 [ -s ] [ -sb ] [ -sbfast ] [ -silent ] [ -stackvar ]
 [ -stop_status=yes|no ] [ -temp=dir ] [ -time ] [ -U ]
 [ -Uname ] [ -u ] [ -unroll=n ] [ -V ] [ -v ] [ -vpara
 [ -w ] [ -xa ] [ -xarch=a ] [ -xautopar ] [ -xcache=c
 [ -xcg89 ] [ -xcg92 ] [ -xchip=c ] [ -xcode=v ]
 [ -xcommonchk[=no|yes] ] [ -xcrossfile=n ] [ -xdepend
 [ -xexplicitpar ] [ -xF ] [ -xhasc[=yes|no] ] [ -xhelp
 [ -xia[=i] ] [ -xildoff ] [ -xildon ] [ -xinline=rl ]
 [ -xinterval=i ] [ -xipo[=0|1] ] [ -xlang=language[,la
 [ -xlibmil ] [ -xlibmopt ] [ -xlicinfo ]
 [ -xlic_lib=sunperf ] [ -Xlist ] [ -xloopinfo ] [ -xma
 [ -xmemalign[=ab] ] [ -xnolib ] [ -xnolibmil ] [ -xnol
 [ -xO[n] ] [ -xopenmp ] [ -xpad ] [ -xparallel ] [ -xp
 [ -xpp=p ] [ -xprefetch=a[,a]] [ -xprofile=p ] [ -xrec
 [ -xreduction ] [ -xregs=r ] [ -xs ] [ -xsafe=mem ]
 [ -xsb ] [ -xsbfast ] [ -xspace ] [ -xtarget=t ] [ -xt
 [ -xtypemap=spec ] [ -xunroll=n ] [ -xvector=yes|no ]
   source file(s) ...  [ -lx ]
```

If you are willing to work more...

- Decrease number of disk accesses (I/O, virtual memory)
- (LINPACK, EISPACK) → LAPACK
- Use numerical libraries tuned for the specific system, BLAS

Find bottlenecks in the code (profilers).
Attack the subprograms taking most of the time.
Find and tune the important loops.

Tuning loops has several disadvantages:

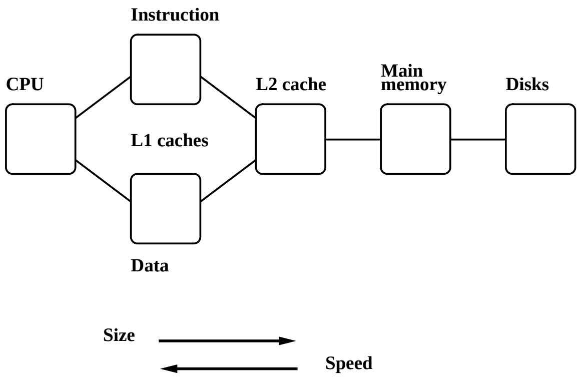
- The code becomes less readable and it is easy to introduce bugs. Check computational results before and after tuning.
- Detailed knowledge about the system, such as cache configuration, may be necessary.
- What is optimal for one system need not be optimal for another; faster on one machine may actually be slower on another.
  This leads to problems with portability.
- Code tuning is not a very deterministic business.
  The combination of tuning and the optimization done by the compiler may give an unexpected result.
- The computing environment is not static; compilers become better and there will be faster hardware of a different construction.
  The new system may require different (or no) tuning.

What should one do with the critical loops?

The goal of the tuning effort is to keep the FPU(s) busy.

Accomplished by efficient use of

- memory hierarchy
- parallel capabilities



Superscalar: start several instructions per cycle.
Pipelining: work on an instruction in parallel.

Locality of reference, data reuse

Avoid data dependencies and other constructions that give pipeline stalls

What can you hope for?

- Many compilers are good.
  May be hard to improve on their job.
  We may even slow the code down.
- Depends on code, language, compiler and hardware.
- Could introduce errors.

But: can give significant speedups.

Not very deterministic, in other words.

Do not rewrite all the loops in your code.

Save a copy of the original code.

Choice of language

Fortran, C/C++ dominating languages for high performance numerical computation.
There are excellent Fortran compilers due to the competition between manufacturers and the design of the language.
It may be harder to generate fast code from C/C++ and it is easy to write inefficient programs in C++

```
void add(const double a[], const double b[],
         double c[], double f, int n)
{
  int k;

  for(k = 0; k < n; k++)
    c[k] = a[k] + f * b[k];
}
```

`n`, was chosen such that the three vectors would fit in the L1-cache, all at the same time.
On the two systems tested (in 2005) the Fortran routine was twice as fast.

From the Fortran 90 standard (section 12.5.2.9):

> “Note that if there is a partial or complete overlap between the actual arguments associated with two different dummy arguments of the same procedure, the overlapped portions must not be defined, redefined, or become undefined during the execution of the procedure.”

Not so in C. Two pointer-variables with different names may refer to the same array.

A Fortran compiler may produce code that works on several iterations in parallel.

```
c(1) = a(1) + f * b(1)
c(2) = a(2) + f * b(2)  ! independent
```

Can use the pipelining in functional units for addition and multiplication.

The assembly code is often unrolled this way as well.
The corresponding C-code may look like:

```
/* This code assumes that n is a multiple of four*/
  for(k = 0; k < n; k += 4) {
    c[k]   = a[k]   + f * b[k];
    c[k+1] = a[k+1] + f * b[k+1];
    c[k+2] = a[k+2] + f * b[k+2];
    c[k+3] = a[k+3] + f * b[k+3];
  }
```

A programmer may write code this way, as well. Unrolling gives:

- fewer branches (tests at the end of the loop)
- more instructions in the loop; a compiler can change the order of instructions and can use prefetching

If we make the following call in Fortran, (illegal in Fortran, legal in C), we have introduced a data dependency.

```
call add(a, c, c(2), f, n-1)
         |  |   |
         a  b   c

c(2) = a(1) + f * c(1)  ! b and c overlap
c(3) = a(2) + f * c(2)  ! c(3) depends on c(2)
c(4) = a(3) + f * c(3)  ! c(4) depends on c(3)
```

If that is the loop you need (in Fortran) write:

```
do k = 1, n - 1
  c(k + 1) = a(k) + f * c(k)
end do
```

This loop is slower than the first one (slower in C as well).

In C, aliased pointers and arrays are allowed which means that it may be harder for a C-compiler to produce efficient code. The C99 `restrict` type qualifier can be used to inform the compiler that aliasing does not occur.

```
void add(double * restrict a, double * restrict b,
         double * restrict c, int n)
```

It is not supported by all compilers and even if it is supported it may not have any effect (you may need a special compiler flag, e.g. `-std=c99`).

An alternative is to use compiler flags, `-fno-alias`, `-xrestrict` etc. supported by some compilers. If you “lie” (or use a Fortran routine with aliasing) you may get the wrong answer!

The compilers on Lucidor (Itanium 2) have improved since 2005, so `restrict` or `-fno-alias` are not needed (for `add`). Restricted pointers give a slight improvement on Lenngren (Intel Xeon) from 2s to 1.3s (10^5 calls of `add` with `n` = 10000).

So this is not a static situation. Compilers and hardware improve every year, so to see the effects of aliasing one may need more complicated examples than `add`. I have kept it because it is easy to understand. On the next page is a slightly more complicated example, but still only a few lines of code, i.e. far from a real code.

Here is a polynomial evaluation using Horner’s method:

```
subroutine horner(px, x, coeff, n)
  integer           j, n
  double precision  px(n), x(n), coeff(0:4), xj

  do j = 1, n
    xj    = x(j)
    px(j) = coeff(0) + xj*(coeff(1) + xj*(coeff(2) &
            + xj*(coeff(3) + xj*coeff(4))))
  end do
end
```

On Lucidor the Fortran code takes 1.6s and the C-code 6.4s. (`n = 1000` and calling the routine 10^6 times).
I compiled using `icc -O3 ...`. Lowering the optimization can help (it did in 2007), but not this year.

On Lenngren the times were 4.9s (Fortran, 2.9s in 2008) 4.8s (C) and on Ferlin it took 1.6s (Fortran) and and 3s (C).

If `-fno-alias` is used, C ≈ Fortran.

It is easy to fix the C-code without using `-fno-alias`

```
...
  double          xj, c0, c1, c2, c3, c4;

  /* no aliasing with local variables*/
  c0 = coeff[0]; c1 = coeff[1]; c2 = coeff[2];
  c3 = coeff[3]; c4 = coeff[4];

  for (j = 0; j < n; j++) {
    xj    = x[j];
    px[j] = c0 + xj*(c1 + xj*(c2 + xj*(c3 + xj*c4)));
  }
...
```

Here comes the assembly output with and without `-fno-alias`.

```
% icc -S -O3 horner.c  % only a part of horner.s
.L2:    fstpl     -8(%ecx,%eax,8)
.L1:
        fldl      (%edi,%eax,8)
        fldl      32(%esi)
        fmul      %st(1), %st
        faddl     24(%esi)
        fmul      %st(1), %st
        faddl     16(%esi)
        fmul      %st(1), %st
        faddl     8(%esi)
        fmulp     %st, %st(1)
        faddl     (%esi)
        addl      $1, %eax
        cmpl      %edx, %eax
        jl        .L2           # Prob 97%
        fstpl     -8(%ecx,%eax,8)

% icc -S -O3 -fno-alias horner.c
.L2:    fstpl     -8(%esi,%eax,8)
.L1:
        fld       %st(0)
        fldl      (%ecx,%eax,8)
        fmul      %st, %st(1)
        fxch      %st(1)
        fadd      %st(3), %st
        fmul      %st(1), %st
        fadd      %st(4), %st
        fmul      %st(1), %st
        fadd      %st(5), %st
        fmulp     %st, %st(1)
        fadd      %st(5), %st
        addl      $1, %eax
        cmpl      %edx, %eax
        jl        .L2           # Prob 97%
        fstpl     -8(%esi,%eax,8)
```

Now to Horner with complex numbers using Fortran (complex is built-in) and C++ (using "C-arrays" of `complex<double>`).

I got the following times, `n = 1000` and calling the routine 10^5 times. Times for default compiler, times in ( ) for the latest.

Compiling using `-O2` or `-O3`, whatever is best.
`i` = Intel, `pg` = Portland Group (latest), `g` = GNU.

System	`ifort`	`icpc`	`pgf90`	`pgCC`	`g95`	`g++`
Lucidor	0.6 (0.6)	1.5 (7.4)	na	na	5.0 (†)	5.4
Lenngren	1.9 (1.9)	3.8 (4.5)	2.3	14.1	3.1	7.0
Ferlin	1.0 (1.0)	2.6 (1.9)	1.2	3.9	1.2 (‡)	2.6

(†) `g77` instead of `g95`. (‡) `gfortran` instead of `g95`.

The tables do show that is important to test different systems, compilers and compile-options.

The behaviour in the above codes changes when `n` becomes very large. *CPU-bound* (the CPU limits the performance) versus *Memory bound* (the memory system limits the performance).

Basic arithmetic and elementary functions

- Common that the FPU can perform `+` and `*` in parallel.
- `a+b*c` can often be performed with one round-off, multiply-add MADD or FMA.
- Several FMAs in parallel on some machines.
- `+` and `*` usually pipelined, so one sum and a product per clock cycle in the best of cases
  (not two sums or two products).
- `/` not usually pipelined and may require around twenty clock cycles.
- Many modern CPUs have several computational cores.

Floating point formats

Type	min denormalized	min normalized	max	bits in mantissa
IEEE 32 bit	$1.4 \cdot 10^{-45}$	$1.2 \cdot 10^{-38}$	$3.4 \cdot 10^{38}$	24
IEEE 64 bit	$4.9 \cdot 10^{-324}$	$2.2 \cdot 10^{-308}$	$1.8 \cdot 10^{308}$	53

- Using single- instead of double precision can give better performance. Fewer bytes must pass through the memory system.
- The arithmetic may not be done more quickly since several systems will use double precision for the computation regardless.

The efficiency of FPUs differ (this on a 2 GHz Opteron).

```
>> A = rand(1000); B = A;
>> tic; C = A * B; toc
Elapsed time is 0.780702 seconds.

>> A = 1e-320 * A;
>> tic; C = A * B; toc
Elapsed time is 43.227665 seconds.
```

For better performance it is sometimes possible to replace a division by a multiplication.

```
vector / scalar          vector * (1.0 / scalar)
```

Integer multiplication and multiply-add are often slower than their floating point equivalents.

```
...
  integer, dimension(10000) :: arr = 1
  integer                   :: s   = 0

  do k = 1, 100000
    s = s + dot_product(arr, arr)
  end do
...
```

Change types to `real` and then to `double precision`

System	integer	single	double
Lucidor	1.7	0.58	0.39
Lenngren	1.0	1.6	1.6
Ferlin	0.92	0.22	0.44

Elementary functions

Often coded in C, may reside in the **`libm`**-library.

- argument reduction
- approximation
- back transformation

Can take a lot of time.

```
>> v = 0.1 * ones(1000, 1);
>> tic; for k = 1:1000, s = sin(v); end; toc
elapsed_time =
    0.039218

>> v = 1e10 * ones(1000, 1);
>> tic; for k = 1:1000, s = sin(v); end; toc
elapsed_time =
    0.717893
```

```
program ugly
  double precision :: x = 2.5d1
  integer          :: k

  do k = 1, 17, 2
    print'(1p2e10.2)', x, sin(x)
    x = x * 1.0d2
  end do

end program ugly

% a.out
  2.50E+01 -1.32E-01
  2.50E+03 -6.50E-01
  2.50E+05 -9.96E-01
  2.50E+07 -4.67E-01
  2.50E+09 -9.92E-01
  2.50E+11 -1.64E-01
  2.50E+13  6.70E-01
  2.50E+15  7.45E-01
  2.50E+17  4.14E+07   <---
```

Some compilers are more clever than others, which is shown on the next page.

You should know that, unless x is an integer, v^x is computed using something like:

$$v^x = e^{\log(v^x)} = e^{x \log v}, \quad 0 < v, x$$

```
subroutine power(vec, n)
  integer                        :: k, n
  double precision, dimension(n) :: vec

  do k = 1, n
    vec(k) = vec(k)**1.5d0  ! so vec(k)^1.5
  end do

end
```

Times with **`n = 10000`** and called **`10000`** on a 2 GHz AMD64.

Compiler **`-O3`**	power	opt. power
Intel	1.2	1.2
`g95`	8.2	1.6
`gfortran`	8.1	1.6

Looking at the assembly output from Intel's compiler:

```
 ...
       fsqrt                     <---- NOTE
       fmulp     %st, %st(1)    <---- NOTE
 ...
```

`g95` and **`gfortran`** call **`pow`** (uses exp and log).

In "opt. power" I have written the loop this way:

```
 ...
  do k = 1, n
    vec(k) = sqrt(vec(k)) * vec(k)
  end do
```

There may be vector versions of elementary functions as well as slightly less accurate versions. AMD's ACML and Intel's MKL both have vector-versions.
Here an example using MKL's VML (Vector Mathematics Library). Read the manual for details (how to use **`vmlSetMode`** to set the accuracy mode, for example).

```
...
  include mkl_vml.fi

  integer, parameter             :: n = 100000
  double precision, dimension(n) :: v, sinv

   v = ...
   call vdsin(n, v, sinv) ! vector-double-sin
...
```

Performance depends on the type of function, range of arguments and vector length. Here are a few examples on Lenngren (1000 repetitions with **`n`** as above). The routines are threaded but seemed to perform best on one thread.

Function	loop	vec	less acc. vec	prec
sin	2.3	0.49	0.40	single
exp	1.6	0.36	0.33	
atan	2.1	0.83	0.51	
sin	3.0	1.3	1.3	double
exp	2.1	0.8	0.8	
atan	7.2	2.2	2.0	

loop means using the standard routine and a loop
(or equivalently **`sinv = sin(v)`**). vec uses the vector routine from VML and less acc. uses the less accurate version.

Switching to a newer compiler changes these numbers dramatically.

```
lenngren > cat sin_ex.f90
subroutine sin_ex(n, v, sinv)
  integer :: n
  double precision, dimension(n) ::  v, sinv

  sinv = sin(v)
end

lenngren > module add i-compilers   <-- default
lenngren > ifort -v
Version 8.1

lenngren > ifort -S sin_ex.f90
lenngren > grep sin sin_ex.s | grep call
        call      sin

lenngren > module dele i-compilers

lenngren > module add i-compilers/latest
lenngren > ifort -v
Version 10.1

lenngren > ifort -S sin_ex.f90
sin_ex.f90(5): (col. 3) remark: LOOP WAS VECTORIZED.

lenngren > grep sin sin_ex.s | grep call | head -1
        call      __svml_sin2
```

On Lenngren versions 8.1, 9.0, 9.1 all use **sin** while versions 10.0, 10.1 call **__svml_sin2** which is just slightly slower than the VML routine. I do not know if this is true for all elementary functions.

Modern CPUs have built-in “vector computers”. (e.g. Intel’s SSE, Streaming SIMD Extensions). Can perform an elementwise vector multiply in parallel: **a = a .* b** (using Matlab notation) where **a** and **b** contain 4 single precision or 2 double precision numbers.

We need an optimizing compiler that produces code using the special vector instructions. For example (using the default compiler):

```
% ifort -O3 -xW -vec_report3    files...
dot_ex.f90(34) : (col. 3) remark: LOOP WAS VECTORIZED.

! A simple benchmark
  s = 0.0
  do k = 1, 10000
    s = s + x(k) * y(k)
  end do
```

Called 100000 times. Times (in s) on Lenngren:

single		double	
no vec	vec	no vec	vec
1.60	0.38	1.80	0.92

On Ferlin the compilers are newer and vectorize automatically. Disadvantage: the x87-FPU uses double extended precision, 64 bit mantissa. SSE2 uses 24 bits (single precision) or 53 bits (double precision). You may get different results. On the next page comes a short example.

```
ferlin > cat sse.f90
program sse
  double precision, parameter     :: ONE = 1.0d0
  double precision, dimension(15) :: s

! ** = power,  (/ ... /) constant vector
  s = 0.1d0**(/ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, &
               11, 12 ,13, 14, 15 /)
  print '(5(1pe9.1))', (ONE + s - ONE) / s - ONE

end

ferlin > ifort -vec_report3 sse.f90
sse.f90(5): (col. 12) remark: loop was not vectorized:
   statement cannot be vectorized.
sse.f90(7): (col. 3) remark: LOOP WAS VECTORIZED.

ferlin > a.out
  8.9E-16  6.7E-16 -1.1E-13 -1.1E-13  6.6E-12
 -8.2E-11  5.8E-10 -6.1E-09  8.3E-08  8.3E-08
  8.3E-08  8.9E-05 -8.0E-04 -8.0E-04  0.0E+00

ferlin > ifort -O0 sse.f90    No optimization
ferlin > a.out
  0.0E+00  0.0E+00  0.0E+00  2.7E-16  2.4E-15
  2.4E-14 -5.2E-13  4.9E-12  1.6E-11  3.4E-10
 -4.0E-09 -4.0E-09 -2.2E-07  3.0E-06 -4.0E-05
```

The second computation is more accurate (the exact answer is a vector of zeros).

Read about **gcc** and vectorization on:
http://gcc.gnu.org/projects/tree-ssa/vectorizationhtml

Eliminating constant expressions from loops

```
  pi = 3.14159265358979d0
  do k = 1, 1000000
    x(k) = (2.0 * pi + 3.0) * y(k)  ! eliminated
  end do

  do k = 1, 1000000
    x(k) = exp(2.0) * y(k)      ! probably eliminated
  end do


  do k = 1, 1000000
    x(k) = my_func(2.0) * y(k)  ! cannot be eliminated
  end do
```

Should use **PURE** functions, **my_func** may have side-effects.

Virtual memory and paging

- Simulate larger memory using disk.
- Virtual memory is divided into pages, perhaps 4 or 8 kbyte.
- Moving pages between disk and physical memory is known as paging.
- Avoid excessive use. Disks are slow.

This test-program was run on a machine with only 64 Mbyte memory, $m * n^2$ is constant, so same number of additions

```
>> type test          % list the program
clear A B C           % remove the matrices
tic                   % start timer
for k = 1:m           % repeat m times
  A = ones(n);        % n x n-matrix of ones
  B = ones(n);        % all are 64-bit numbers
  C = A + B;
end
toc                   % stop timer

% Run three test cases
>> n = 500; m = 16; test  % 5.7 Mbyte for A, B and C
elapsed_time = 1.1287     % roughly ONE SECOND

>> n = 1000; m = 4; test  % 22.9 Mbyte
elapsed_time = 1.1234     % roughly the same as above

>> n = 2000; m = 1; test  % 91.6 Mbyte
elapsed_time = 187.9      % more than THREE MINUTES
```

```
% vmstat 1  (edited)
      page          cpu
  pi     po     us  sy  id
   0      0      0  0  100
   0      0      0  0  100
 352    128      0  0  100  <-- third test is run
 616    304      0  6   94
 608    384      0  2   98
 712    256      0  2   98   etc. for over 3 minutes

 pi = kilobytes paged in  / second
 po = kilobytes paged out / second
```

Input-output

Need to store $5 \cdot 10^6$ double precision numbers in a file.
A local disk was used for the tests. Intel's Fortran compiler on AMD64. Roughly the same times in C.

Test	Statement	time (s)	size (Mbyte)
1	`write(10, '(1pe23.16)') x(k)`	29.4	114.4
2	`write(10) x(k)`	19.5	76.3
3	`write(10) (vec(j), j = 1, 10000)`	0.1	38.2
4	`write(10) vec(1:10000)`	0.1	38.2
5	`write(10) vec`	0.1	38.2

File sizes:

$$1: \quad \underbrace{5 \cdot 10^6}_{\#\text{ of numbers}} \cdot \underbrace{(23+1)}_{\text{characters + newline}} / \underbrace{2^{20}}_{\text{Mbyte}} \approx 114.4$$

$$2: \quad \underbrace{5 \cdot 10^6}_{\#\text{ of numbers}} \cdot \underbrace{(8+4+4)}_{\text{number + delims}} / \underbrace{2^{20}}_{\text{Mbyte}} \approx 76.3$$

$$3-5: \quad \left[\underbrace{5 \cdot 10^6}_{\#\text{ of numbers}} \cdot \underbrace{8}_{\text{number}} + 500 \cdot \underbrace{(4+4)}_{\text{delims}} \right] / \underbrace{2^{20}}_{\text{Mbyte}} \approx 38.2$$

In 5 `vec` has 10000 elements and we write the array 500 times.

`g95` and `gfortran` were slower in all cases but case 2, where `g95` took 6s.

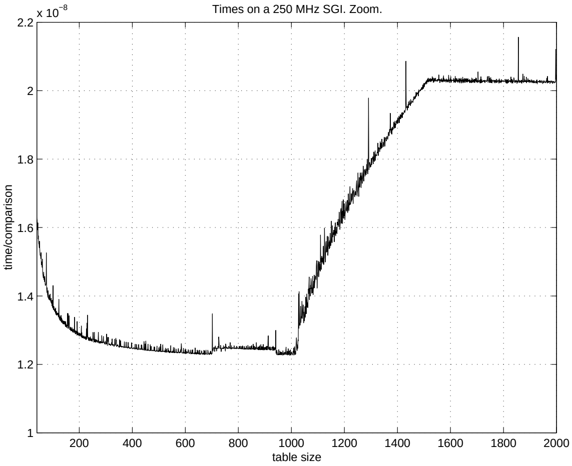
Portability of binary files?

- Perhaps
- File structure may differ
- Byte order may differ
- Big-endian, most significant byte has the lowest address ("big-end-first").
- The Intel processors are little-endian ("little-end-first").

```
On a big-endian machine
write(10) -1.0d-300, -1.0d0, 0.0d0, 1.0d0, 1.0d300


Read on a little-endian
2.11238712E+125  3.04497598E-319  0.
3.03865194E-319 -1.35864115E-171
```

Memory locality and caches

```
  1  2                         # of columns
----------------  ...  -----------------
|  |  |  |  |  |       |  |  |  |  |  |   <-- keys
----------------  ...  -----------------
|  |  |  |  |  |       |  |  |  |  |  |
----------------  ...  -----------------
|  |  |  |  |  |       |  |  |  |  |  |
----------------  ...  -----------------
|  |  |  |  |  |       |  |  |  |  |  |
----------------  ...  -----------------

Search -->
```



Analysis

- Fortran stores matrices by columns
- (C stores matrices by rows)
- L1 data cache is two-way set-associative, two sets with 512 lines each (MIPS R10000, SGI)
- Replacement policy is LRU (Least Recently Used)
- One column per L1 cache line
- When $\leq$ 1024 columns only cache misses in the first search

Suppose we have two sets of four cache lines, instead.

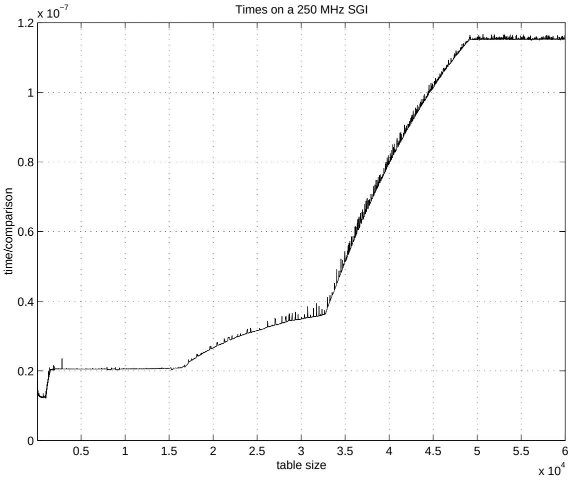
Assume we have nine columns.

```
  sets
|1|  |5|     |9|  |5|     |5|  |1|     |5|  |9|
|2|  |6|     |2|  |6|     |2|  |6|     |2|  |6|
|3|  |7|     |3|  |7|     |3|  |7|     |3|  |7|
|4|  |8|     |4|  |8|     |4|  |8|     |4|  |8|

 after 8     after 9     after 8     after 9
 cols        cols        next search
```

Assume we have twelve columns.

```
   first search                next search
|1| |5|   | 9| |5|    | 9| |1|   |5|  |1|   |5|  | 9|
|2| |6|   |10| |6|    |10| |2|   |6|  |2|   |6|  |10|
|3| |7|   |11| |7|    |11| |3|   |7|  |3|   |7|  |11|
|4| |8|   |12| |8|    |12| |4|   |8|  |4|   |8|  |12|

after 8   after 12    after 4    after 8    after 12
```

L2-cache, two sets of 4096 lines, each with a length of 132 bytes.



Notice that the fastest and slowest case differ by a factor of 9.5.

Change algorithm and data structure. If not: blocking.

Blocking is an efficient technique for data re-use and is used in many matrix algorithms.

More data re-use; loop fusion

Blocking is one method for data re-use.

```
v_min = v(1)
do k = 2, n
  if ( v(k) < v_min ) v_min = v(k) ! fetch v(k)
end do

v_max = v(1)
do k = 2, n
  if ( v(k) > v_max ) v_max = v(k) ! fetch v(k) again
end do
```

Merge loops data re-use, less loop overhead.

```
v_min = v(1)
v_max = v(1)
do k = 2, n
  if ( v(k) < v_min ) then     ! v(k) is fetched here
    v_min = v(k)
  elseif ( v(k) > v_max ) then ! and re-used here
    v_max = v(k)
  end if
end do
```

On some systems the following loop body is faster

```
  vk = v(k)                ! optional
  if(v_min < vk) v_min = vk ! can use v(k) instead
  if(v_max > vk) v_max = vk
```

or

```
  vk    = v(k)
  v_min = min(v_min, vk)
  v_max = max(v_max, vk)
```

When dealing with large, but unrelated, data sets it may be faster to split the loop in order to use the caches better. Here is a contrived example:

```
  integer, parameter                :: n = 5000
  double precision, dimension(n, n) :: A, B, C, D
  ...
  sum_ab = 0.0
  sum_cd = 0.0
  do col = 1, n
    do row = 1, n  ! the two sums are independent
      sum_ab = sum_ab + A(row, col)* B(col, row)
      sum_cd = sum_cd + C(row, col)* D(col, row)
    end do
  end do
!
! Split the computation
!
  sum_ab = 0.0
  do col = 1, n
    do row = 1, n
      sum_ab = sum_ab + A(row, col)* B(col, row)
    end do
  end do

  sum_cd = 0.0
  do col = 1, n
    do row = 1, n
      sum_cd = sum_cd + C(row, col)* D(col, row)
    end do
  end do
```

When `n = 5000` the first loop requires 4.9 s and the second two 0.84 s (together) on a 2.4 GHz, 4 Gbyte, Opteron.

The importance of small strides

If no data re-use, try to have locality of reference.

Small strides.
`v(1), v(2), v(3),..`, stride one
`v(1), v(3), v(5),..`, stride two

```
  slower                        faster
  s = 0.0                       s = 0.0
  do row = 1, n                 do col = 1, n
    do col = 1, n                 do row = 1, n
      s = s + A(row, col)           s = s + A(row, col)
    end do                        end do
  end do                        end do

  A(1, 1)
  A(2, 1)
    ...    first column
  A(n, 1)
----------
  A(1, 2)
  A(2, 2)
    ...    second column
  A(n, 2)
----------
    ....
----------
  A(1, n)
  A(2, n)
    ...    n:th column
  A(n, n)
```

Some compilers can switch loop order (loop interchange).
In C the leftmost alternative will be the faster.

Performance on Lucidor, Lenngren and Ferlin. Compiling using `-O3` in the first test and using `-O3 -ipo` in the second.

	Lucidor		Lenngren		Ferlin	
	C	Fortran	C	Fortran	C	Fortran
By row	0.7 s	2.9 s	0.6 s	2.4 s	0.5 s	1.5 s
By column	4.6 s	0.3 s	2.4 s	0.6 s	1.6 s	0.5 s
By row `-ipo`	0.3 s	0.3 s	0.6 s	0.6 s	0.5 s	0.5 s
By column `-ipo`	2.9 s	0.3 s	0.6 s	0.6 s	1.5 s	0.5 s

`-ipo`, interprocedural optimization i.e. optimization between routines (even in different files) gives a change of loop order, at least for Fortran, in this case. Some Fortran compilers can do this just specifying `-O3`, and this happens on Lucidor and Ferlin if we put the main-program and the subroutines in the same file.

```
ferlin > ifort -O3 main.f90 sub.f90    Separate files
sub.f90(27): remark: LOOP WAS VECTORIZED.

ferlin > ifort -O3 -ipo main.f90 sub.f90
ipo: remark #11000: performing multi-file optimizations
ipo: remark #11005: generating object file /tmp/ipo_ifc
main.f90(13): remark: PERMUTED LOOP WAS VECTORIZED.
main.f90(19): remark: LOOP WAS VECTORIZED.

ferlin > ifort -O3 all.f90           One file
all.f90(13): remark: PERMUTED LOOP WAS VECTORIZED.
all.f90(20): remark: LOOP WAS VECTORIZED.
all.f90(52): remark: LOOP WAS VECTORIZED.
```

Blocking and large strides

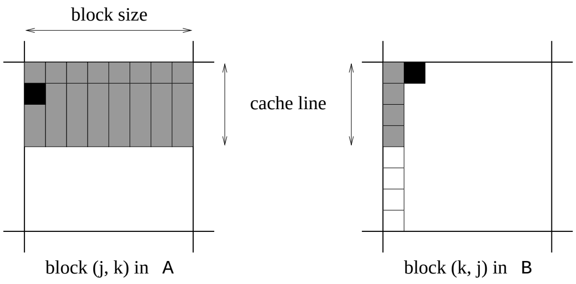
Sometimes loop interchange is of no use.

```
  s = 0.0
  do row = 1, n
    do col = 1, n
      s = s + A(row, col) * B(col, row)
    end do
  end do
```

Blocking is good for data re-use, and when we have large strides.

Partition **A** and **B** in square sub-matrices each having the same order, the block size.

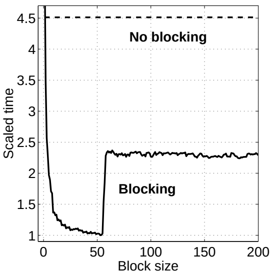
Treat pairs of blocks, one in **A** and one in **B** such that we can use the data which has been fetched to the L1 data cache.
Looking at two blocks:

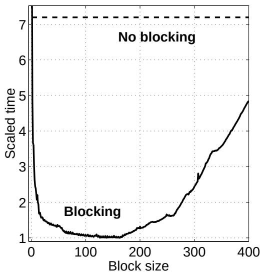


The block size must not be too large. Must be able to hold all the grey elements in **A** in cache (until they have been used).

This code works even if **n** is not divisible by the block size).

```
! first_row = the first row in a block etc.

do first_row = 1, n, block_size
  last_row = min(first_row + block_size - 1, n)
  do first_col = 1, n, block_size
    last_col = min(first_col + block_size - 1, n)
    do row = first_row, last_row      ! sum one block
      do col = first_col, last_col
        s = s + A(row, col) * B(col, row)
      end do
    end do
  end do
end do
```





$n = 2000$.

Note the speedups (4.5 and 7.2).

One can study the behaviour in more detail.

PAPI = Performance Application Programming Interface
http://icl.cs.utk.edu/papi/index.html
PAPI uses hardware performance registers, in the CPU, to count different kinds of events, such as L1 data cache misses and TLB-misses.

TLB = Translation Lookaside Buffer, a cache in the CPU that is used to improve the speed of translating virtual addresses into physical addresses.

See the Springer article for an example.

Two important libraries

BLAS (the Basic Linear Algebra Subprograms) are the standard routines for simple matrix computations, e.g.

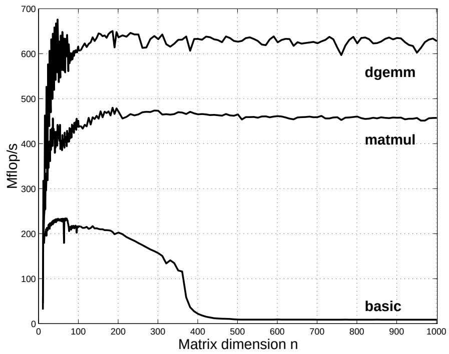
BLAS1: **y := a*x + y** one would use **daxpy**

BLAS2: **dgemv** can compute **y := a*A*x + b*y**

BLAS3: **dgemm** forms **C := a*A*B + b*C**

daxpy: $\mathcal{O}(n)$ data, $\mathcal{O}(n)$ operations
dgemv: $\mathcal{O}(n^2)$ data, $\mathcal{O}(n^2)$ operations
dgemm: $\mathcal{O}(n^2)$ data, $\mathcal{O}(n^3)$ operations, data re-use

Matrix multiplication: "row times column", slow.
Blocking is necessary.



375 MHz machine, start two FMAs per clock cycle, top speed is 750 million FMAs per second.

LAPACK is the standard library for (dense):

- linear systems
- eigenvalue problems
- linear least squares problems

There is no support for large sparse problems, although there are routines for banded matrices of different kinds.

LAPACK is built on top of BLAS (BLAS3 where possible). When using LAPACK, it is important to have optimised BLAS.

Inlining

moving the body of a short procedure to the calling routine.

Calling a procedure or a function takes time and may break the pipelining. So the compiler (or the programmer) can move the body of a *short* subprogram to where it is called. Some compilers do this automatically when the short routine resides in the same file as the calling routine. A compiler may have a flag telling the compiler to look at several files. Using some compilers you can specify which routines are to be inlined.

Indirect addressing, pointers

Sparse matrices, PDE-meshes...
Bad memory locality, poor cache performance.

```
do k = 1, n
  j    = ix(k)
  y(j) = y(j) + a * x(j)
end do
```

system	random **ix**	ordered **ix**	no **ix**
1	39	16	9
2	56	2.7	2.4
3	83	14	10

If-statements

If-statements in a loop may stall the pipeline. Modern CPUs and compilers are rather good at handling branches, so there may not be a large delay.

```
Original version          Optimized version

do k = 1, n               take care of k = 1
  if ( k == 1 ) then      do k = 2, n
    statements              statements for k = 2 to n
  else                    end do
    statements
  end if
end do
```

```
if ( most probable ) then
  ...
else if ( second most probable ) then
  ...
else if ( third most probable ) then
  ...
```

`if (a(k) .and.  b(k)) then`, least likely first

`if (a(k) .or.  b(k)) then`, most likely first

Alignment

```
integer*1  work(100001)
...
! work(some_index) in a more general setting
call do_work(work(2), 12500) ! pass address of work(2
...
end

subroutine do_work(work, n)
integer           n
double precision  work(n)

work(1) = 123
...
```

May produce “Bus error”.
Alignment problems.

It is usually required that double precision variables are stored at an address which is a multiple of eight bytes (multiple of four bytes for a single precision variable).

The slowdown caused by misalignment may easily be a factor of 10 or 100.

Closing notes

Two basic tuning principles:

- Improve the memory access pattern
  - Locality of reference
  - Data re-use

  Stride minimization, blocking, proper alignment and the avoidance of indirect addressing.
- Use parallel capabilities of the CPU
  - Avoid data dependencies
  - Loop unrolling
  - Inlining
  - Elimination of if-statements

Choosing a good algorithm and a fast language, handling files in an efficient manner, getting to know ones compiler and using tuned libraries are other very important points.

Low level profiling

`valgrind` and PAPI are two tools for counting cache misses.

`http://valgrind.org/`, **`man valgrind`** and
`/usr/share/doc/valgrind-3.1.1/html/index.html`

From 22nd stanza in “Grímnismál” (poetic Edda). In old Icelandic and Swedish:

Valgrind heitir,	Valgrind den heter,
er stendr velli á	som varsnas på slätten,
heilög fyr helgum dyrum;	helig framför helig dörrgång;
forn er sú grind,	fornåldrig är grinden,
en ■at fáir vitu,	och få veta,
hve hon er í lás lokin.	hur hon i lås är lyckt.

and a reasonable (I believe) English translation:

Valgrind is the lattice called,
in the plain that stands,
holy before the holy gates:
ancient is that lattice,
but few only know
how it is closed with lock.

The main gate of Valhall (Eng. Valhalla), hall of the heroes slain in battle.

From the manual:
“valgrind is a flexible program for debugging and profiling Linux executables. It consists of a core, which provides a synthetic CPU in software, and a series of ”tools”, each of which is a debugging or profiling tool.”

The memcheck tool performs a range of memory-checking functions, including detecting accesses to uninitialized memory, misuse of allocated memory (double frees, access after free, etc.) and
detecting memory leaks.

We will use the cachegrind tool:
cachegrind is a cache simulator. It can be used to annotate every line of your program with the number of instructions executed and cache misses incurred.

```
valgrind --tool=toolname program args
```

Call the following routine

```
void sub0(double A[1000][1000], double*s)
{
  int j, k, n = 1000;

  *s = 0;

  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
       *s += A[k][j];
}
```

Compile with **-g**:

```
% gcc -g main.c sub.c
```

I have edited the following printout:

```
% valgrind --tool=cachegrind a.out

==5796== Cachegrind, an I1/D1/L2 cache profiler.
==5796== Copyright (C) 2002-2005, and GNU GPL'd,
         by Nicholas Nethercote et al.
==5796== For more details, rerun with: -v
9.990000e+08 6.938910e-01
```

```
I   refs:     46,146,658
I1  misses:          756
L2i misses:          748
I1  miss rate:      0.00%
L2i miss rate:      0.00%

D   refs:     21,073,437  (18,053,809 rd+3,019,628 wr)
D1  misses:      255,683  (   130,426 rd+  125,257 wr)
L2d misses:      251,778  (   126,525 rd+  125,253 wr)
D1  miss rate:       1.2% (       0.7%  +      4.1%  )
L2d miss rate:       1.1% (       0.7%  +      4.1%  )

L2 refs:         256,439  (   131,182 rd+  125,257 wr)
L2 misses:       252,526  (   127,273 rd+  125,253 wr)
L2 miss rate:        0.3% (       0.1%  +      4.1%  )
```

valgrind produced the file, **cachegrind.out.5796** (**5796** is a pid). To see what source lines are responsible for the cache misses we use **cg_annotate -pid source-file** I have edited the listing and removed the columns dealing with the instruction caches (the lines are too long otherwise).

```
% cg_annotate --5796 sub.c
       Dr    D1mr    D2mr        Dw D1mw D2mw
        .       .       .         .    .    .  void sub0
        0       0       0         2    0    0  {
        0       0       0         1    0    0  int j, k,
        1       0       0         2    0    0  *s = 0;
    3,002       0       0         1    0    0  for (j =
3,002,000       0       0     1,000    0    0  for (k =
7,000,000 129,326 125,698 1,000,000    0    0 *s += A[k
        3       0       0         0    0    0  }
```

Dr: data cache reads (ie. memory reads), **D1mr**: L1 data cache read misses **D2mr**: L2 cache data read misses **Dw**: D cache writes (ie. memory writes) **D1mw**: L1 data cache write misses **D2mw**: L2 cache data write misses

To decrease the number of **Dw**:s we use a local summation variable (no aliasing) and optimze, **-O3**.

```
  double local_s = 0;
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      local_s += A[k][j];

  *s = local_s;
```

We can also interchange the loops. Here is the counts for the summation line:

```
       Dr    D1mr    D2mr
7,000,000 129,326 125,698 *s += A[k][j]; previous
1,000,000 125,995 125,696  local_s, -O3
1,000,000 125,000 125,000  above + loop interchange
```

Dw = D1mw = D2mw = 0

valgrind cannot count TLB-misses, so switch to PAPI, which can.
PAPI = Performance Application Programming Interface
http://icl.cs.utk.edu/papi/index.html
PAPI requires root privileges to install, so I have tested the code at PDC.

PAPI uses hardware performance registers, in the CPU, to count different kinds of events, such as L1 data cache misses and TLB-misses. Here is (a shortened example):

```
% icc main.c sub.c
% papiex -m -e PAPI_L1_DCM -e PAPI_L2_DCM \
        -e PAPI_L3_DCM -e PAPI_TLB_DM -- ./a.out

Processor:                Itanium 2
Clockrate:                1299.000732
Real usecs:               880267
Real cycles:              1143457807
Proc usecs:               880000
Proc cycles:              1143120000

PAPI_L1_DCM:              2331
PAPI_L2_DCM:              3837287
PAPI_L3_DCM:              3118846
PAPI_TLB_DM:              24086796

Event descriptions:
Event: PAPI_L1_DCM: Level 1 data cache misses
Event: PAPI_L2_DCM: Level 2 data cache misses
Event: PAPI_L3_DCM: Level 3 data cache misses
Event: PAPI_TLB_DM: Data TLB misses
```

The values change a bit between runs, but the order of magnitude stays the same. Here are a few tests. I call the function 50 times in a row. **time** in seconds. **cycl** = 10^9 process cycles. **L1**, **L2**, **L3** and **TLB** in kilo-misses. **local** using a local summation variable.

	icc -O0	icc -O3	icc -O3	icc -O3	
			local	loop interc	
time:	3.5	0.6	0.07	0.3	
cycl:	4.6	0.8	0.09	0.4	Giga
L1:	13	4	3	4	kilo
L2:	3924	3496	1923	2853	kilo
L3:	3169	3018	1389	2721	kilo
TLB:	24373	24200	24	24	kilo

`time` and `cycl` are roughly the same, since the clockrate is 1.3 GHz. Note that the local summation variable, in column three, makes a dramatic difference. This is the case for loop interchange as well (column four) where we do not have a local summation variable (adding one gives essentially column three).

Note the drastic reduction of TLB-misses in the fast runs.

Here comes PAPI on the blocking example,
`s = s + A(i, k) * B(k, j)`, with `ifort -O3`.
`n = 5000` and ten calls.

On the Itanium:

bs:	NO BL	16	32	40	64	128	
time:	5.6	2.0	1.6	1.5	1.6	5.1	
L1:	69	46	41	43	44	52	kilo
L2:	306	51	48	52	54	59	Mega
L3:	31	33	38	38	36	35	Mega
TLB:	257	19	12	10	15	267	Mega

Note again the drastic reduction of TLB-misses.

Profiling on a higher level

Most unix systems have `prof` and `gprof` which can be used to find the most time consuming routines. `gcov` (Linux) (`tcov` Sun) can find the loops (statements), in a routine, that are executed most frequently.
`man prof`, `man gprof`, `man gcov` for details.

This is how you use `gprof` on the student system.
The flags are not standardised, so you have to read the documentation, as usual.

```
ifort -O3 -qp prog.f90 sub.f90
icc   -O3 -qp prog.c   sub.f90

gfortran -O3 -pg prog.f90  sub.f90
g95      -O3 -pg prog.f90  sub.f90
gcc      -O3 -pg prog.c    sub.c
g++      -O3 -pg prog.cc   sub.c

./a.out  produces gmon.out
gprof
```

One can use other options, of course, and have more than two files. One should link with the profiling options as well since it may
include profiled libraries.
Profiling disturbs the run; it takes more time.

The Intel compilers have support for "Profile-guided Optimization", i.e. the information from the profiled run can be used by the
compiler (the second time you compile) to generate more efficient code.

A few words about `gcov`. This command tells us:

- how often each line of code executes
- what lines of code are actually executed

Compile without optimization. It works only with `gcc`. So it should work with `g95` and `gfortran` as well. There may, however, be problems with different versions of `gcc` and the `gcc`-libraries. See the web-page for the assignment for the latest details.

To use `gcov` on the student system (not Intel in this case) one should be able to type:

```
g95  -fprofile-arcs -ftest-coverage prog.f90 sub.f90
./a.out

gcov prog.f90  creates prog.f90.gcov
gcov sub.f90   creates sub.f90.gcov

less prog.f90.gcov   etc.
```

and for C

```
gcc  -fprofile-arcs -ftest-coverage prog.c sub.c
```

similarly for `gfortran` and `g++`.

Example: Arpack, a package for solving large and sparse eigenvalue problems, $Ax = \lambda x$ and $Ax = \lambda Bx$. I fetched a compressed tar-file, unpacked, read the README-file, edited the configuration file, and compiled using make. After having corrected a few Makefiles everything worked. I then recompiled using the compiler options for `gprof` and `tcov` (on a Sun; I have not run this one the AMD-system).

I used the `f90`-compiler even though Arpack is written in Fortran77. (There is also Arpack++, a collection of classes that offers C++ programmers an interface to Arpack.)

First `gprof`:

```
% gprof | less   (1662 lines, less is a pager)
or
% gprof | more   (or m with    alias m more)
                 (I have       alias m less)
or
% gprof > file_name  (emacs file_name, for example)
etc.
```

The first part of the output is the flat profile, such a profile can be produced by `prof` as well. Part of it, in compressed form, comes on the next page. The flat profile may give a sufficient amount of information.

```
Each sample counts as 0.01 seconds.
 %   cumulative   self           self    total
time   seconds   seconds  calls  s/call  s/call  name
79.10    8.10     8.10    322    0.03     0.03  dgemv_
 8.50    8.97     0.87     60    0.01     0.01  dger_
 4.10    9.39     0.42     58    0.01     0.01  dgttrs_
 3.22    9.72     0.33    519    0.00     0.00  dcopy_
 2.25    9.95     0.23    215    0.00     0.00  dnrm2_
 0.49   10.00     0.05    562    0.00     0.00  __open
        ...  lots of lines deleted  ...
 0.00   10.24     0.00      1    0.00    10.14  main
        ...  lots of lines deleted  ...
 0.00   10.24     0.00      1    0.00     0.00  strchr
```

`name` is the name of the routine (not the source file). The Sun-compiler converts the routine name to lower case and adds _ . `__open` is a system (compiler?) routine.

The columns are:
`% time` the percentage of the total running time of the program used by this function. Not the one it calls, look at `main`.

`cumulative seconds` a running sum of the number of seconds accounted for by this function and those listed above it.

`self seconds` the number of seconds accounted for by this function alone. This is the major sort for this listing.

`calls` the number of times this function was invoked, if this function is profiled, else blank.

`self ms/call` the average number of milliseconds spent in this function per call, if this function is profiled, else blank.

`total ms/call` the average number of milliseconds spent in this function and its descendents per call, if this function is profiled, else blank. Note `main`.

`dgemv` is a BLAS routine, double general matrix vector multiply:

```
 dgemv - perform one of the matrix-vector operations
 y := alpha*A*x + beta*y or y := alpha*A'*x + beta*y
```

I have compiled the Fortran code instead of using a faster performance library so we can look at the source code.

Let us run `tcov` on `dgemv`.

```
Part of the output (compressed):
                    ...
    168 ->       DO 60, J = 1, N
   4782 ->          IF( X(JX).NE.ZERO )THEN
   4740 ->             TEMP = ALPHA*X(JX)
                       DO 50, I = 1, M
77660160 ->                Y(I) = Y(I) + TEMP*A(I, J)
           50          CONTINUE
                    END IF
   4782 ->          JX = JX + INCX
           60    CONTINUE

                Top 10 Blocks

                Line      Count

                 211   77660160
                 238   50519992
                 177     871645
                   ...
```

Note that this code is very poor. Never use the simple Fortran BLAS- or Lapack routines supplied with some packages. One lab deals with this issue.

More about `gprof`

`gprof` produces a call graph as well. It shows, for each function, which functions called it, which other functions it called, and how many times. There is also an estimate of how much time was spent in the subroutines called by each function. This list is edited.

```
index %time self  children  called     name
                                     ...  rm lines
-----------------------------------------------
            0.01   10.13     1/1      main [1]
[3]   99.0  0.01   10.13      1     MAIN_ [3]
            0.00    7.19    59/59     dsaupd_ [5]
            0.00    2.45     1/1      dseupd_ [8]
            0.42    0.00    58/58     dgttrs_ [14]
 ... lines deleted
-----------------------------------------------
            0.83    0.00    33/322    dsapps_ [11]
            1.48    0.00    59/322    dlarf_ [9]
            5.79    0.00   230/322    dsaitr_ [7]
[4]   79.1  8.10    0.00   322      dgemv_ [4]
            0.00    0.00  1120/3179   lsame_ [50]
-----------------------------------------------
```

Each routine has an index (see table at the end) and is presented between `---`-lines. 8.10s was spent in `dgemv` itself, 79.1% of total (including calls from `dgemv`). `dsapps`, `dlarf`, `dsaitr` (parents) called `dgemv` which in turn called `lsame`, a child. `dsapps` made 33 out of 322 calls and `dgemv` took 0.83s for the calls. `dgemv` called `lsame` 1120 of 3179 times, which took no measurable time (`self`).

`children`: For `dgemv` it is the total amount of time spent in all its children (`lsame`). For a parent it is the amount of that time that was propagated, from the function's children (`lsame`), into this parent. For a child it is the amount of time that was propagated from the child's children to `dgemv`.

Profiling in Matlab

Matlab has a built-in profiling tool. `help profile` for more details. Start Matlab (must use the GUI).

```
>> profile on
>> run        % The assignment
Elapsed time is 1.337707 seconds.
Elapsed time is 13.534952 seconds.
>> profile report          % in mozilla or netscape
>> profile off
```

You can start the profiler using the GUI as well
(click in "Profiler" using "Desktop" under the main meny). The output comes in a new window and contains what looks like the flat profile from `gprof`.

One can see the details in individual routines by clicking on the routine under `Function Name` This produces a `gcov`-type of listing. It contains the number of times a line was executed and the time it took.

Using Lapack from Fortran and C

Use Lapack to solve a problem like:

$$\begin{bmatrix} 1 & -1 & -2 & -3 & -4 \\ 1 & 1 & -1 & -2 & -3 \\ 2 & 1 & 1 & -1 & -2 \\ 3 & 2 & 1 & 1 & -1 \\ 4 & 3 & 2 & 1 & 1 \end{bmatrix} x = \begin{bmatrix} -9 \\ -4 \\ 1 \\ 6 \\ 11 \end{bmatrix}$$

The solution is the vector of ones. We use the Lapack-routine `dgesv` from Lapack. Here is a man-page:

```
NAME
DGESV  - compute the solution to a real system of
         linear equations A * X = B,

SYNOPSIS
SUBROUTINE DGESV( N, NRHS, A, LDA, IPIV, B, LDB, INFO )
INTEGER       INFO, LDA, LDB, N, NRHS
INTEGER       IPIV( * )
DOUBLE        PRECISION A( LDA,* ), B( LDB, * )

PURPOSE
DGESV computes the solution to a real system of linear
equations A * X = B, where A is an N-by-N matrix and X
and B are N-by-NRHS matrices.
The LU decomposition with partial pivoting and row
interchanges is used to factor A as  A = P* L * U,
where P is a permutation matrix, L is unit lower
triangular, and U is upper triangular.  The factored
form of A is then used to solve the system of equations
A * X = B.

ARGUMENTS
N       (input) INTEGER
        The number of linear equations, i.e., the order
        of the matrix A.  N >= 0.
```

```
NRHS    (input) INTEGER
        The number of right hand sides, i.e., the numbe
        of columns of the matrix B.  NRHS >= 0.

A       (input/output) DOUBLE PRECISION array, dimensio
        (LDA,N) On entry, the N-by-N coefficient matrix
        A.  On exit, the factors L and U from the
        factorization A = P*L*U; the unit diagonal
        elements of L are not stored.

LDA     (input) INTEGER
        The leading dimension of the array A.
        LDA >= max(1,N).

IPIV    (output) INTEGER array, dimension (N)
        The pivot indices that define the permutation
        matrix P; row i of the matrix was interchanged
        with row IPIV(i).

B       (input/output) DOUBLE PRECISION array, dimensio
        (LDB,NRHS) On  entry, the N-by-NRHS matrix of
        right hand side matrix B.  On exit, if INFO = 0
        the N-by-NRHS solution matrix X.

LDB     (input) INTEGER
        The leading dimension of the array B.
        LDB >= max(1,N).

INFO    (output) INTEGER
        = 0:  successful exit
        < 0:  if INFO = -i, the i-th argument had an
              illegal value
        > 0:  if INFO = i, U(i,i) is exactly zero.  The
              factorization has been completed, but the
              factor U is exactly singular, so the
              solution could not be computed.
```

In Fortran90, but using the F77 interface, and F77-type declarations (to get shorter lines) this may look like:

```
program main
  integer, parameter ::  n   = 10, lda = n, &
                        ldb = n, nrhs = 1
  integer            ::  info, row, col, ipiv(n)
  double precision   ::  A(lda, n), b(ldb)

  do col = 1, n
    do row = 1, n
      A(row, col) = row - col
    end do
    A(col, col) = 1.0d0
    b(col)      = 1 + (n * (2 * col - n - 1)) / 2
  end do

  call dgesv ( n, nrhs, A, lda, ipiv, b, ldb, info )

  if ( info == 0 ) then
    print*, "Maximum error = ", maxval(abs(b - 1.0d0))
  else
    print*, "Error in dgesv: info = ", info
  end if

end program main

% Compile and link, somehow, to Lapack
% a.out
 Maximum error =  4.218847493575595E-15
```

Where can we find `dgesv`? There are several options. Fetching the Fortran-code from Netlib, using a compiled (optimized) library etc. One of the assignments, Lapack (Uniprocessor optimization), deals with these questions.

The following optimized libraries contain Lapack and BLAS (and perhaps routines for fft, sparse linear algebra, etc. as well).

- AMD: ACML (AMD Core Math Library).
- Intel: MKL (Intel Math Kernel library).
- SGI: complib.sgimath (Scientific and Mathematical Library).
- IBM: ESSL (Engineering and Scientific Subroutine Library).
- Sun: Sunperf (Sun Performance Library).

There may be parallel versions.

Now for C and C++

Fairly portable (do not use local extensions of the compiler). Think about: In C/C++

- matrices are stored by row (not by column as in Fortran)
- matrices are indexed from zero
- call by reference for arrays, call by value for scalars
- the Fortran compiler MAY add an underline to the name
- you may have to link with Fortran libraries
  (mixing C and Fortran I/O may cause problems, for example)
- C++ requires an `extern`-declaration, in C you do not have to supply it (but do)
- make sure that C and Fortran types are compatible (number of bytes)
- some systems have C-versions of Lapack

In the example below I have linked with the Fortran-version since not all systems have C-interfaces. Make sure not to call `dgesv` from C on the Sun, if you want the Fortran-version (`dgesv` gives you the C-version).

```
#include <math.h>
#include <stdio.h>
#define  _N   10

#ifdef __cplusplus
extern "C" void                 /* For C++ */
#else
extern void                     /* For C   */
#endif
   dgesv_(int *, int *, double *, int *, int[],
          double[], int *, int *);
/*
 * int [] or int *. double [][] is NOT OK but
 * double [][10] is, provided we
 * call dgesv_ with A and not &A[0][0].
 */

int main()
{
  int      n = _N, lda = _N, ldb = _N, nrhs = 1,
           info, row, col, ipiv[_N];
  double   A[_N][_N], b[_N], s, max_err;

  /* Make sure you have the correct mix of types.*/
  printf("sizeof(int) = %d\n", sizeof(int));

  /* Indexing from zero. */
  for (col = 0; col < n; col++) {
    for (row = 0; row < n; row++)
      A[col][row] = row - col;  /* Note TRANSPOSE */
    b[col] = 1 + (n * (1 + 2 * col - n)) / 2;
    A[col][col] = 1;
  }
```

```
  /* Note underline and & for the scalar types.
   * &A[0][0] not to get a
   * conflict with the prototype.
   */
  dgesv_(&n, &nrhs, &A[0][0], &lda, ipiv, b,
         &ldb, &info);

  if (info) {
    printf("Error in dgesv: info = %d\n", info);
    return 1;
  } else {
    max_err = 0.0;
    for (row = 0; row < n; row++) {
      s = fabs(b[row] - 1.0);
      if (s > max_err)
        max_err = s;
    }
    printf("Maximum error = %e\n", max_err);
    return 0;
  }
}

On a Sun. See the lab for AMD.

% cc -fast extern.c -xlic_lib=sunperf
% a.out
sizeof(int) = 4
Maximum error = 4.218847e-15

% CC -fast extern.c -xlic_lib=sunperf
% a.out
sizeof(int) = 4
Maximum error = 4.218847e-15

% a.out    If you call dgesv and not dgesv_
sizeof(int) = 4
** On entry to DGESV , parameter number 1 has an
   illegal value.  Error in dgesv: info = -1
```

Java

It is possible to mix Java with other languages using JNI, the Java Native Interface. Wikipedia is a good starting point (look for jni).

Here are a few words on Java.

```
% cat test.java
public class test {
  public static void main (String[] args) {
    int n = 10;
    double[] a = new double[n];

    for(int j = 0; j < n; j++)
      a[j] = j;

    System.out.println("a[n-1] = " + a[n-1]);
  }
}

% javac test.java       Produces test.class
% java test             Execute (can optimize, later...
a[n-1] = 9.0
```

javac produces the file **test.class** containing the bytecode.

java is the Java interpreter that reads and executes **test.class**. We can study the bytecode (instructions) using **javap**, the Java class file disassembler. The interpreter uses a stack and has local variables; I have called them **var_1** etc. To make the bytecode easier to read I have used our variable names. Half of the listing (mostly dealing with the print, has been deleted). I have not printed all the pops from the stack.

See Wikipedia (java bytecode) for more details.

```
% javap -verbose test

  ...
public static void main(java.lang.String[]);
Code:
  0: bipush  10        10    -> stack
  2: istore_1          stack -> var_1
  3: iload_1           var_1 -> stack
  4: newarray double   create double a[10], &a[0]->stack
  6: astore_2          &a[0] -> var_2
  7: iconst_0          0 -> stack
  8: istore_3          0 -> var_3 (corresponds to j)

  9: iload_3           j -> stack
 10: iload_1           n -> stack
 11: if_icmpge  25     if ( j >= n ) goto line 11+25
 14: aload_2           &a[0] -> stack
 15: iload_3           j -> stack
 16: iload_3           j -> stack  (used as index)
 17: i2d               double(j) -> stack
 18: dastore           a[j] = double(j), "index reg"
 19: iinc3, 1          j++
 22: goto9             goto line 9:
...
 54:return
}
```

To speed things up the bytecode interpreter (**java**) often uses a JIT (Just In Time) technique. A JIT compiler converts all of the bytecode into native machine code just as a Java program is run. This results in run-time speed improvements over code that is interpreted by a Java virtual machine.

java -client testor
java -server test(usually much faster; default).

One can profile Java programs. from `man java`:

```
-Xprof
  Profiles the running program, and sends profiling
  data to standard output. This option is provided
  as a utility that is useful in program development
  and is not intended to be be used in production
  systems.

-Xrunhprof[:help][:suboption=value,...]
  Enables cpu, heap, or monitor profiling. This
  option is typically followed by a list of
  comma-separated suboption=value pairs. Run the
  command   java -Xrunhprof:help   to obtain a
  list of suboptions and their default values.
```

Interfacing Matlab with C

It is not uncommon that we have a program written in C (or Fortran) and need to communicate between the program and Matlab.

The simplest (but not the most efficient) way the fix the communication is to use ordinary text files. This is portable and cannot go wrong (in any major way). The drawback is that it may be a bit slow and that we have to convert between the internal binary format and text format. We can execute programs by using the `unix`-command (or `!` or `system`).

One can do more, however:

- Reading and writing binary MAT-files from C
- Calling Matlab as a function (Matlab engine)
- Calling a C- or Fortran-function from Matlab (using MEX-files, compiled and dynamically linked C- or Fortran-routines)

In the next few pages comes a short example on how to use MEX-files.

MEX-files

Let us write a C-program that can be called as a Matlab-function. The MEX-routine will call a band solver, written in Fortran, from Lapack for solving an Ax=b-problem. The routine uses a Cholesky decomposition, where A is a banded, symmetric and positive definite matrix.

b contains the right hand side(s) and x the solution(s).
I fetched the routines from `www.netlib.org`

Matlab has support for solving unsymmetric banded systems, but has no special routines for the positive definite case.

We would call the function by typing:

```
>> [x, info] = bandsolve(A, b);
```

where `A` stores the matrix in compact form. `info` returns some status information (`A` not positive definite, for example).

`bandsolve` can be an m-file, calling a MEX-file. Another alternative is to let `bandsolve` be the MEX-file. The first alternative is suitable when we need to prepare the call to the MEX-file or clean up after the call.

The first alternative may look like this:

```
function [x, info] = bandsolve(A, b)
A_tmp = A;  % copy A
b_tmp = b;  % copy b
% Call the MEX-routine
[x, info] = bandsolve_mex(A_tmp, b_tmp);
```

I have chosen to make copies of A and b. The reason is that the Lapack-routine replaces A with the Cholesky factorization and b by the solution. This is not what we expect when we program in Matlab. If we have really big matrices, and if we do not need A and b afterwards we can skip the copy (although the Matlab-documentation says that it “may produce undesired side effects”).

I will show the code for the second case where we call the MEX-file directly. Note that we use the file name, `bandsolve`, when invoking the function. There should always be a `mexFunction` in the file, which is the entry point. This is similar to a C-program, there is always a `main`-routine.

It is possible to write MEX-files in Fortran, but is more natural to use C.

First some details about how to store the matrix (for the band solver). Here an example where we store the lower triangle. The dimension is six and the number of sub- (and super-) diagonals is two.

```
   a11  a22  a33  a44  a55  a66
   a21  a32  a43  a54  a65   *
   a31  a42  a53  a64   *    *

 Array elements marked * are not used by the routine.
```

The Fortran-routine, `dpbsv`, is called the following way:

```
  call dpbsv( uplo, n, kd, nB, A, lda, B, ldb, info )
```

where

```
  uplo = 'U':  Upper triangle of A is stored
         'L':  Lower triangle of A is stored
```

We will assume that `uplo = 'L'` from now on

```
  n    = the dimension of A
  kd   = number of sub-diagonals
  nB   = number of right hand sides (in B)
  A    = packed form of A
  lda  = leading dimension of A
  B    = contains the right hand side(s)
  ldb  = leading dimension of B
  info = 0, successful exit
       < 0, if info = -i, the i-th argument had
            an illegal value
       > 0, if info = i, the leading minor of order i
            of A is not positive definite, so the
            factorization could not be completed,
            and the solution has not been computed.
```

Here comes `bandsolve.c` (I am using C++-style comments):

```
#include <math.h>
// For Matlab
#include "mex.h"

void dpbsv_(char *, int *, int *, int *, double *,
            int *, double *, int *, int *);

void mexFunction(int nlhs,       mxArray*plhs[],
                 int nrhs, const mxArray *prhs[])
{
  double    *px, *pA, *pb, *pA_tmp;
  mxArray   *A_tmp;
  char uplo = 'L';
  int k, A_rows, A_cols, b_rows, b_cols, kd, info;

  // Check for proper number of arguments
  if (nrhs != 2) {
    mexErrMsgTxt("Two input arguments required.");
  } else if (nlhs > 2) {
    mexErrMsgTxt("Too many output arguments.");
  }

  A_rows = mxGetM(prhs[0]);
  kd     = A_rows - 1;       // # of subdiags
  A_cols = mxGetN(prhs[0]);  // = n

  b_rows = mxGetM(prhs[1]);
  b_cols = mxGetN(prhs[1]);

  if (b_rows != A_cols || b_cols <= 0)
    mexErrMsgTxt("Illegal dimension of b.");
```

```
  // Create a matrix for the return argument
  // and for A. dpbsv destroys A and b).
  // Should check the return status.
  plhs[0]=mxCreateDoubleMatrix(b_rows, b_cols, mxREAL);
  plhs[1]=mxCreateDoubleMatrix(1, 1, mxREAL);
  A_tmp  =mxCreateDoubleMatrix(A_rows, A_cols, mxREAL);

  px = mxGetPr(plhs[0]);     // Solution x
  pA = mxGetPr(prhs[0]);     // A
  pA_tmp = mxGetPr(A_tmp);   // temp for A
  pb = mxGetPr(prhs[1]);     // b

  for (k = 0; k < b_rows * b_cols; k++) // b -> x
    *(px + k) = *(pb + k);

  for (k = 0; k < A_rows * A_cols; k++) // A -> A_tmp
    *(pA_tmp + k) = *(pA + k);

  dpbsv_(&uplo, &A_cols, &kd, &b_cols, pA_tmp,
         &A_rows, px, &b_rows, &info);

  *mxGetPr(plhs[1]) = info; // () higher prec. than*
  if (info)
    mexWarnMsgTxt("Non zero info from dpbsv.");

  // Should NOT destroy plhs[0] or plhs[1]
  mxDestroyArray(A_tmp);
}
```

Some comments:
nrhs is the number of input arguments to the MEX-routine. **prhs** is an array of pointers to input arguments. **prhs[0]** points to a so-called, **mxArray**, a C-struct containing size-information and pointers to the matrix-elements.
prhs[0] corresponds to the first input variable, **A** etc.

Since one should not access the member-variables in the struct directly, there are routines to extract size and elements.
A_rows = mxGetM(prhs[0]); extracts the number of rows and
A_cols = mxGetN(prhs[0]); extracts the number of columns.

The lines

```
  plhs[0]=mxCreateDoubleMatrix(b_rows, b_cols, mxREAL);
  plhs[1]=mxCreateDoubleMatrix(1, 1, mxREAL);
```

allocate storage for the results (of type **mxREAL**, i.e. ordinary **double**).

A_tmp = mxCreateDoubleMatrix(A_rows, A_cols, mxREAL);
allocates storage for a copy of **A**, since the Lapack-routine destroys the matrix.
px = mxGetPr(plhs[0]); extracts a pointer to the (real-part) of the matrix elements and stores it in the pointer variable, **px**.

The first for-loop copies **b** to **x** (which will be overwritten by the solution). The second loop copies the matrix to the temporary storage, pointed to by **A_tmp**. This storage is later deallocated using **mxDestroyArray**.

Note that neither the input- nor the output-arguments should be deallocated.

It is now time to compile and link. This is done using the Bourne-shell script **mex**. We must also make a symbolic link. Since we would like to change some parameters when compiling, we will copy and edit an options file, **mexopts.sh**.

```
% which matlab
/chalmers/sw/sup/matlab-2008b/bin/matlab
(ls -ld /chalmers/sw/sup/matlab* to see the versions)
```

Make the link:

```
% ln -s /usr/lib/libstdc++.so.6.0.3 libstdc++.so
```

Copy **mexopts.sh**:

```
% cp /chalmers/sw/sup/matlab-2008b/bin/mexopts.sh .
```

and edit the file (after **glnx86**):

```
  change CC='gcc' to CC='gcc4'
```

if you are using the latest Matlab-version. In the **CFLAGS**-line, change **-ansi** to **-Wall**, to use C++-style comments and to get more warnings.

Add **-L.** to **CLIBS**, and add linker-info. to get Goto-blas:

```
   CLIBS="$RPATH $MLIBS -lm  -L. -lstdc++
          -L/chalmers/sw/unsup/libgoto/lib
          -lgoto_opt32-r0.96"    NOTE: in one long line
  change -O to -O3 in FOPTIMFLAGS
```

Make sure your **LD_LIBRARY_PATH** contains the name of the directory where Goto-blas resides.

I have fetched the lapack-routines from Netlib:

```
% ls lapack
dpbsv.f   dpbtf2.f  dpbtrf.f  dpbtrs.f  dpotf2.f
ieeeck.f  ilaenv.f  lsame.f   xerbla.f
```

Now it is time to compile:

```
% mex -f ./mexopts.sh bandsolve.c lapack/*.f
```

which creates **bandsolve.mexglx**

Now we can test a simple example in Matlab:

```
>> A = [2 * ones(1, 5); ones(1, 5)]
A =
     2     2     2     2     2
     1     1     1     1     1

>> [x, info] = bandsolve(A, [3 4 4 4 3]')
x =
    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
info =
     0
```

Here a case when A is not positive definite:

```
>> A(1, 1) = -2;  % Not positive definite
>> [x, info] = bandsolve(A, [3 4 4 4 3]')
Warning: Non zero info from dpbsv.
x =              % Since b is copied to x
     3
     4
     4
     4
     3
info =
     1
```

Note that the first call of **bandsolve** may take much more time, since the mex-file has to be loaded. Here a small test when n=10000, kd=10:

```
>> tic; [x, info] = bandsolve(A, b); toc
Elapsed time is 0.147128 seconds.

>> tic; [x, info] = bandsolve(A, b); toc
Elapsed time is 0.034625 seconds.

>> tic; [x, info] = bandsolve(A, b); toc
Elapsed time is 0.034950 seconds.
```

Now to some larger problems:
With n=100000 and kd=10, **dpbsv** takes 0.25 s and sparse backslash 0.41 s on a student AMD-computer.
kd=20 gives the times 0.48 s and 0.77 s respectively.

On an Opteron with more memory:
with n=1000000, kd=10 the times are 2.9 s, 4.7 s.
Increasing kd to 50, the times are 15.4 s and 27.6 s.

Libraries, ar, ld

Numerical (and other software) is often available in libraries. To use a subroutine from a library one has to use the linker to include the routine. Advantages:

- Fewer routines to keep track of.
- There is no need to have source code for the library routines that a program calls.
- Only the needed modules are loaded.

These pages deal with how one can make libraries and use the linker, link-editor, **ld**.

```
% cat sub1.f90
subroutine sub1
  print*, 'in sub1'
end

% cat sub2.f90
subroutine sub2
  print*, 'in sub2'
end

% cat sub3.f90
subroutine sub3
  print*, 'in sub3'
  call sub2
end

% cat main.f90
program main
  call sub3
end
```

```
% ls sub*.f90
sub1.f90  sub2.f90  sub3.f90

% g95 -c sub*.f90
sub1.f90:
sub2.f90:
sub3.f90:

% ls sub*
sub1.f90  sub1.o  sub2.f90  sub2.o  sub3.f90  sub3.o

% ar -r libsubs.a sub*.o

% ar -t libsubs.a
sub1.o
sub2.o
sub3.o

% g95 main.f90 -L. -lsubs
% a.out
 in sub3
 in sub2
```

g95 calls the link-editor, **ld**, to combine **main.o** and the object files in the library to produce the executable **a.out**-file. Note that the library routines become part of the executable.

If you write **-lname** the link-editor looks for a library file with name **libname.a** (or **libname.so**).

On some systems you may have to give the location of the library using the flag **-L** (**ld** does not look everywhere). **.** means current working directory, but you could have a longer path, of course. You can have several **-L** flags.

From **man ar**:

ar creates an index to the symbols defined in relocatable object modules in the archive when you specify the modifier s.
...
An archive with such an index speeds up linking to the library, and allows routines in the library to call each other without regard to their placement in the archive.

ar seems to do this even with **ar -r ...** as well.
If your library does not have this index:

```
% g95 main.f90 -L. -lsubs
./libsubs.a: could not read symbols:
 Archive has no index; run ranlib to add one
% ranlib libsubs.a
% g95 main.f90 -L. -lsubs
```

The order of libraries is important:

```
% g95 -c sub4.f90 sub5.f90
sub4.f90:
sub5.f90:

% ar -r libsub45.a sub[45].o

% ar -t libsub45.a
sub4.o
sub5.o
```

```
% cat sub4.f90
subroutine sub4
  print*, 'in sub4'
  call sub2
end

% cat main.f90
program main     ! A NEW main
  call sub4
end

% g95 main.f90 -L. -lsubs -lsub45
./libsub45.a(sub4.o)(.text+0x6f): In function ‘sub4_’:
: undefined reference to ‘sub2_’
```

ld does <u>not</u> go back in the list of libraries.

```
% g95 main.f90 -L. -lsub45 -lsubs
% a.out
 in sub4
 in sub2
```

The compiler uses several system libraries, try **g95 -v ...**.
One such library is the C math-library, **/usr/lib/libm.a**

```
% ar -t /usr/lib/libm.a | grep expm1 | head -1
s_expm1.o

% man expm1
NAME   expm1, expm1f, expm1l  - exponential minus 1

       #include <math.h>
       double expm1(double x);
...
```

```
% cat main.c
#include <math.h>
#include <stdio.h>

int main()
{
  double x = 1.0e-15;

  printf("expm1(x)   = %e\n", expm1(x));
  printf("exp(x) - 1 = %e\n", exp(x) - 1.0);

  return 0;
}

% gcc main.c
/tmp/cc40PH1o.o(.text+0x2b): In function ‘main’:
: undefined reference to ‘expm1’
/tmp/cc40PH1o.o(.text+0x53): In function ‘main’:
: undefined reference to ‘exp’

% gcc main.c -lm
% a.out
expm1(x)   = 1.000000e-15
exp(x) - 1 = 1.110223e-15
```

Shared libraries

More about **libm**. The following output has been shortened.

```
% ls -l /usr/lib/libm.*
  /usr/lib/libm.a
  /usr/lib/libm.so -> ../../lib/libm.so.6

% ls -l /lib/libm.*
  /lib/libm.so.6 -> libm-2.3.4.so

% ls -l /lib/libm-2.3.4.so
-rwxr-xr-x  1 root root 176195 Aug 20 03:21
    /lib/libm-2.3.4.so
```

What is this last file?

```
% ar -t /lib/libm-2.3.4.so
ar: /lib/libm-2.3.4.so: File format not recognized

Look for symbols (names of functions etc.):
% objdump -t /lib/libm-2.3.4.so | grep expm1
 ...
00009420  w    F .text  0000005c   expm1
 ...
```

so means shared object. It is a library where routines are loaded to memory during runtime. This is done by the dynamic linker/loader **ld.so**. The **a.out**-file is not complete in this case, so it will be smaller.

One problem with these libraries is that they are needed at runtime which may be years after the executable was created. Libraries may be deleted, moved, renamed etc.

One advantage is shared libraries can be shared by every process that uses the library (provided the library is constructed in that way).

It is easier to handle new versions, applications do not have to be relinked.

If you link with **-lname**, the first choice is **libname.so** and the second **libname.a**.

/usr/lib/libm.so -> ../../lib/libm.so.6 is a soft link (an "alias").

```
% ln -s full_path alias
```

The order is not important when using shared libraries (the linker has access to all the symbols at the same time).

A shared library is created using **ld** (not **ar**) or the compiler, the **ld**-flags are passed on to the linker.

```
% g95 -o libsubs.so -shared -fpic subs.f90
% g95  main.f90 -L. -lsubs
% ./a.out
 in sub4
 in sub2

From man gcc (edited):

-shared
  Produce a shared object which can then be linked with
  other objects to form an executable. Not all systems
  support this option. For predictable results, you mus
  also specify the same set of options that were used
  to generate code (-fpic, -fPIC, or model suboptions)
  when you specify this option.[1]

-fpic
  Generate position-independent code (PIC) suitable for
  use in a shared library, if supported for the target
  machine. Such code accesses all constant addresses
  through a global offset table (GOT). The dynamic
  loader resolves the GOT entries when the program
```

```
  starts (the dynamic loader is not part of GCC; it is
  part of the operating system). ...
```

Since the subroutines in the library are loaded when we run the program (they are not available in **a.out**) the dynamic linker must know where it can find the library.

```
% cd ..
% Examples/a.out
Examples/a.out: error while loading shared libraries:
 libsubs.so: cannot open shared object file: No such
 file or directory

% setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH\:Examples
% Examples/a.out
 in sub4
 in sub2
```

LD_LIBRARY_PATH contains a colon separated list of paths where **ld.so** will look for libraries. You would probably use a full path and not **Examples**.

$LD_LIBRARY_PATH is the old value (you do not want to do **setenv LD_LIBRARY_PATH Examples** unless **LD_LIBRARY_PATH** is empty to begin with.

The backslash is needed in **[t]csh** (since colon has a special meaning in the shell). In **sh** (Bourbe shell) you may do something like:

```
$ LD_LIBRARY_PATH=$LD_LIBRARY_PATH:Example
$ export LD_LIBRARY_PATH      (or on one line)
```

Some form of **LD_LIBRARY_PATH** is usually available (but the name may be different). The SGI uses the same name for the path but the linker is called **rld**. Under HPUX 10.20, for example, the dynamic loader is called **dld.sl** and the path **SHLIB_PATH**.

It is possible to store the location of the library when creating **a.out**.

```
% unsetenv LD_LIBRARY_PATH
% g95 -o libsubs.so -shared -fpic subs.f90
% g95 main.f90 -L. -lsubs
% a.out
a.out: error while loading shared libraries:
 libsubs.so: cannot open shared object file:
 No such file or directory
```

Add the directory in to the runtime library search path (stored in **a.out**):

```
 -Wl, means pass -rpath 'pwd' to ld

% g95 -Wl,-rpath 'pwd' main.f90 -L. -lsubs

% cd ..   or cd to any directory
% Examples/a.out
 in sub4
 in sub2
```

A useful command is **ldd** (print shared library dependencies):

```
% ldd a.out
  libsubs.so => ./libsubs.so (0x00800000)
  libm.so.6 => /lib/tls/libm.so.6 (0x009e2000)
  libc.so.6 => /lib/tls/libc.so.6 (0x008b6000)
  /lib/ld-linux.so.2 (0x00899000)
```

Used on our **a.out**-file it will, in the first case, give:

```
% ldd Examples/a.out
    libsubs.so => not found
```

In the second case, using **rpath**, **ldd** will print the full path.

And now to something related:

Large software packages are often spread over many directories. When distributing software it is customary to pack all the directories into one file. This can be done with the **tar**-command (tape archive). Some examples:

```
% ls -FR My_package
bin/        doc/      install*  lib/      README
configure*  include/  INSTALL   Makefile  src/

My_package/bin:      binaries

My_package/doc:      documentation
userguide.ps   or in pdf, html etc.

My_package/include:  header files
params.h  sparse.h

My_package/lib:      libraries

My_package/src:      source
main.f  sub.f
```

Other common directories are **man** (for manual pages), **examples**, **util** (for utilities).

README usually contains general information, **INSTALL** contains details about compiling, installation etc. There may be an **install**-script and there is usually a **Makefile** (probably several).

If the package is using X11 graphics there may be an **Imakefile**. The tool **xmkmf** (using **imake**) can generate a Makefile using local definitions and the **Imakefile**.

In a Linux environment binary packages (such as the Intel compilers) may come in RPM-format. See **http://www.rpm.org/** or type **man rpm**, for details.

Let us now create a tar-file for our package.

```
% tar cvf My_package.tar My_package
My_package/
My_package/src/
My_package/src/main.f
My_package/src/sub.f
My_package/doc/
 ...
My_package/Makefile
```

One would usually compress it:

```
% gzip My_package.tar  (or using bzip2)
```

This command produces the file `My_package.tar.gz`.
`.tgz` is a common suffix as well (`tar.bz2` or `.tbz2` for `bzip2`).

To unpack such a file we can do (using gnu `tar`) (`z` for `gunzip`, or `zcat`, `x` for extract, `v` for verbose and `f` for file):

```
% tar zxvf My_package.tar.gz
My_package
My_package/src/
 ...
```

Using `tar`-commands that do not understand `z`:

```
% zcat      My_package.tar.gz | tar vxf -    or
% gunzip -c My_package.tar.gz | tar vxf -    or
% gunzip <  My_package.tar.gz | tar vxf -    or
% gunzip    My_package.tar.gz            followed by
% tar xvf   My_package.tar
```

I recommend that you first try:

```
% tar ztf My_package.tar.gz
My_package/  ...
```

To see that files are placed in a new directory (and that are no name conflicts).
Under GNOME there is an Archive Manager (File Roller) with a GUI. Look under `Applications/System Tools`

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

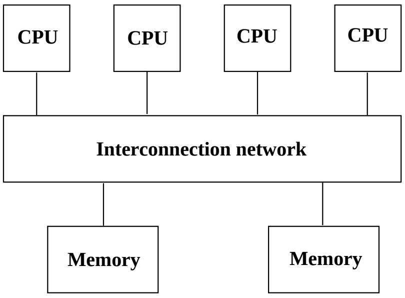
Asynchronous (the processes work independently).

- 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.

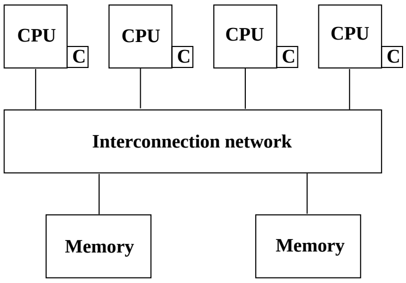
Shared memory

Bus-based architecture



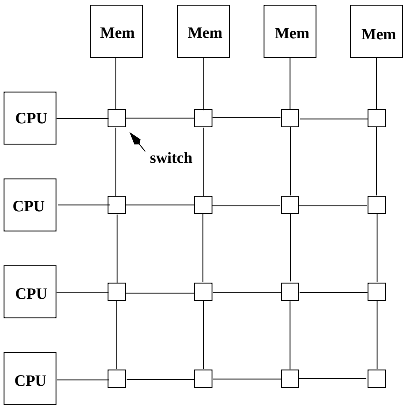
- Limited bandwidth (the amount of data that can be sent through a given communications circuit per second).
- Do not scale to a large number of processors. 30-40 CPUs common.

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



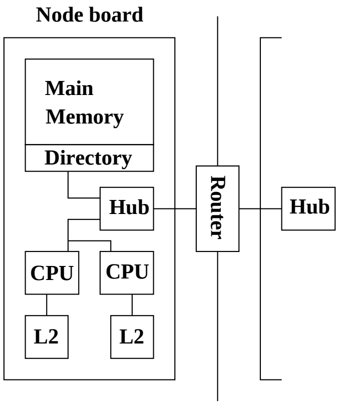
I have denoted the caches by C. Cache coherence.

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

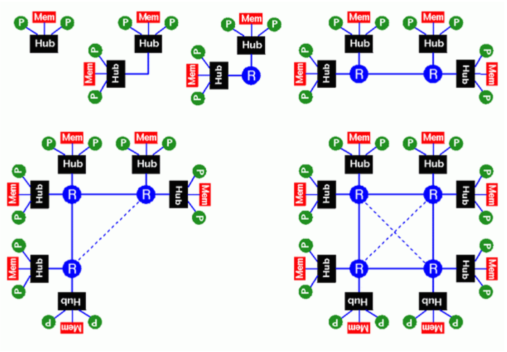


- Any processor can access any memory module. Any other processor can simultaneously access any other memory module.
- Expensive.
- Common with a memory hierarchy. Several crossbars may be connected by a cheaper network. NonUniform Memory Access (NUMA).

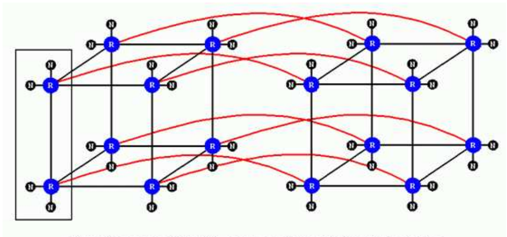
Example of a NUMA architecture: SGI Origin 2000, R10000 CPUS connected by a fast network.



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.



Directly connect two 32–node systems via Craylink cables using the one free link on each router

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)?

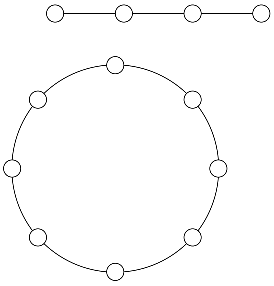
A simple model for communication:

$$\textit{time to transfer n bytes} = \textit{latency} + n \,/\, \textit{bandwidth}$$

Distributed memory

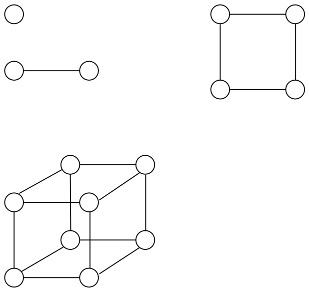
In a distributed memory system, each processor has its own private memory. A simple distributed memory system can be constructed by a number of workstations and a local network.

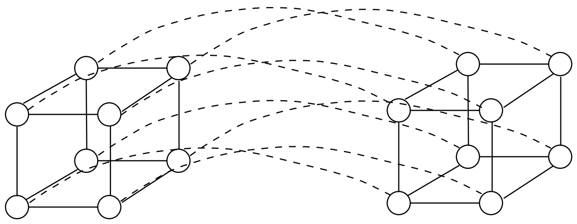
Some examples:



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

Hypercubes of dimensions 0, 1, 2 and 3.





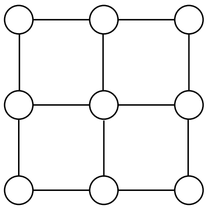
A 4-dimensional hypercube. Generally, a hypercube of dimension d+1 is constructed by connecting corresponding processors in two hypercubes of dimension d.

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.

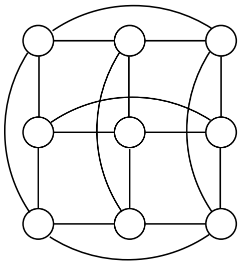
If the available connectivity (for a specific machine) is sufficient depends on the problem and the data layout.

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), o■■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.

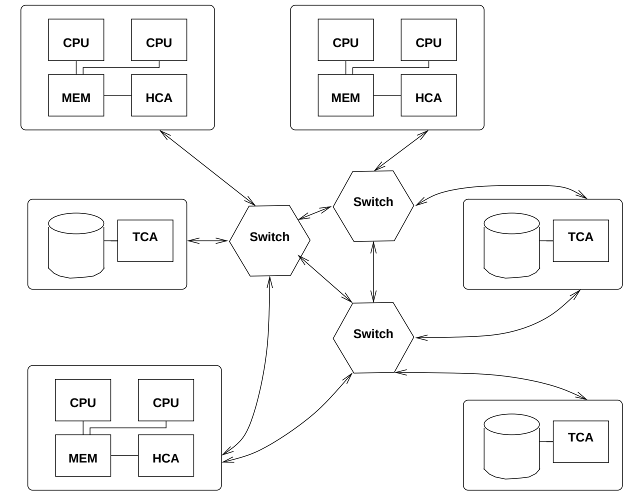
A word on Infiniband. First a quote from
`http://www.infinibandta.org/`
"InfiniBand is a high performance, switched fabric interconnect standard for servers. ... Founded in 1999, the InfiniBand Trade Association (IBTA) is comprised of leading enterprise IT vendors including Agilent, Dell, Hewlett-Packard, 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
```

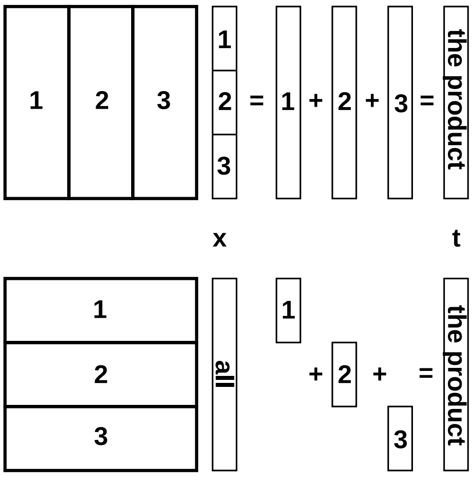
Note that $x^{(k)}$ does not "converge" in the ordinary sense.
We may have problems with over/underflow.

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:

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

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)

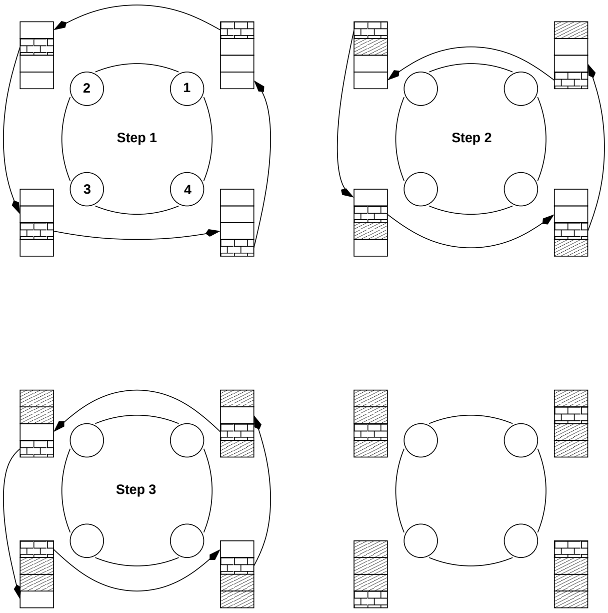
Let us assume that the A_j have been distributed and look at the matrix-vector multiply.

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 $\beta$
  $else$
    $t_p = A_p x$
    $\mu = 1/\max(\mu_1, \mu_2, \ldots, \mu_{\#p})$
    $x_p = \mu\ t_p$
  $end\ if$
  $\mu_p = \max(|x_p|)$

  $seg = p$
  $for\ j = 1$ to $\#p - 1\ do$
    send $x_{seg}, \mu_{seg}$ to the next processor
    compute $seg$
    receive $x_{seg}, \mu_{seg}$ 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):

$$\begin{aligned} wct(\#p) \ &= \ seq.\ part\ of\ comp. + \frac{parallel\ part\ of\ comp.}{\#p} + \\ &\#p \ \ (communication) \end{aligned}$$

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, $0 \le s \le 1$ (part wrt time), so the part that can be parallelized is $1 - s$. Hence,

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

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:

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

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 $8n\beta_p \le M_p$, 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).

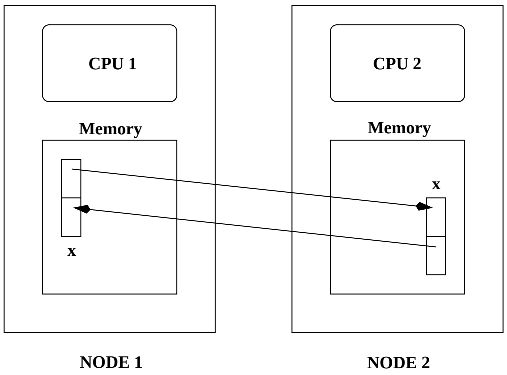
If we store U and D by rows it is easy to compute $Ux + Dx$ using our row-oriented algorithm. To compute U^Tx requires a column-oriented approach (if U is partitioned by rows, U^T will be partitioned by columns, and a column-oriented algorithm seems reasonable). So the program is a combination of a row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

A few words about communication

In our program we had the loop:

```
for $j = 1$ to $\#p - 1$
  send $x_{seg}, \mu_{seg}$ to the next processor
  compute $seg$
  receive $x_{seg}, \mu_{seg}$ 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^\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      r
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
```

```
#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   */
}
```

```
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:

```
% mpicc simple.c
% mpirun -n 2 -mca btl tcp,self ./a.out
Number of processes = 2
After MPI_Send
After MPI_Recv, message = 2
```

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_WORLDis 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 ($0 \leq \text{tag} \leq$ 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 OpenMPI) when `N_MAX = 1000`, but it hangs, it deadlocks, when `N_MAX = 20000`. 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.

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 which may cause a deadlock is having to sends in a row. A silly example 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.

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;
  }

  /* This code is written in SIMD-form*/
  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;

  /* compute my part of the integral*/
  my_int = trapez(a, a + interval, 100);
  /* my_int is my 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);

  if (my_rank == MASTER)
    printf("The integral = %e\n", I);

  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**).

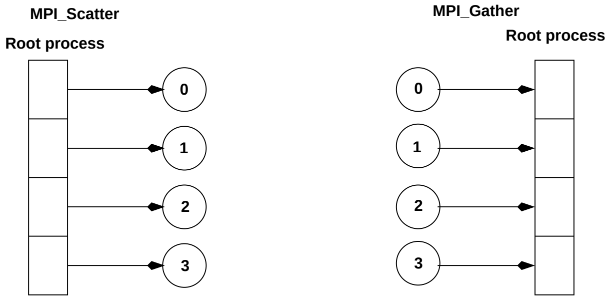
The **MPI_Allreduce** may be performed in an efficient way. Suppose we have eight processes, **0**, ..., **7**. **|** denotes a split.

```
           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 = [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 you 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:

n	C = A * B				x = A \ b				l = eig(A)			
	1	2	3	4	1	2	3	4	1	2	3	4
800	0.3	0.2	0.1	0.1	0.2	0.1	0.1	0.1	3.3	2.5	2.4	2.3
1600	2.1	1.1	0.8	0.6	1.1	0.7	0.6	0.5	20	12	12	12
3200	17.0	8.5	6.0	4.6	7.9	4.8	4.0	3.5	120	87	81	80

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.

Here are the times (to save space):

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. Here comes the program:

```
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
```

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.

Version 3.0, May 2008, not supported by all compilers (supported by `ifort/icc` ver 11.0, for example).

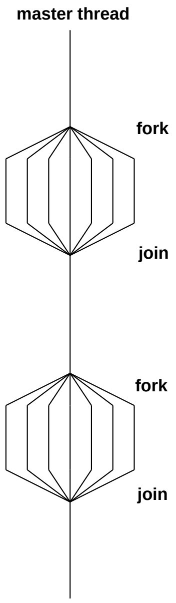
v2.5 mainly supports data parallelism (SIMD), all threads perform the same operations but on different data. In v3.0 there is better support for "tasks", different threads perform different operations (so-called function parallelism, or task parallelism).

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

For books about OpenMP see the introduction.

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 executes 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
  (but common with multi-core computers).
- Less control over the “hidden” message passing and memory allocation.

A simple example

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

int main()
{
  int            i, i_am, n = 10000;
  double         a[n], b[n], c[n];

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

// a parallel for loop
#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]); // the master

// a parallel region
#pragma omp parallel private(i_am)
  {
    i_am = omp_get_thread_num();  // 0 to #threads - 1
    printf("i_am = %d\n", i_am);  // all threads print

    #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());
    }  // use { } for begin/end
  }
  return 0;
}
```

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.

At the end of the parallel for, the threads join and they synchronise at an implicit barrier.

Output from several threads may be interleaved.

To avoid multiple prints we ask the master thread (thread zero) to print. The following numbers are printed:
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).

```
ferlin > icc -openmp omp1.c
ferlin > setenv OMP_NUM_THREADS 1
ferlin > a.out
4.899900, 6154.284900
i_am = 0
num threads = 1
max threads = 1
max cpus    = 8

ferlin > setenv OMP_NUM_THREADS 4
ferlin > a.out
4.899900, 6154.284900
i_am = 3
i_am = 0
num threads = 4
max threads = 4
max cpus    = 8
i_am = 2
i_am = 1

ferlin > setenv OMP_NUM_THREADS 9
ferlin > a.out
4.899900, 6154.284900

etc.
```

On some some systems (# of threads > # of cpus = 8):

```
Warning: MP_SET_NUMTHREADS greater than available cpus
```

Make no assumptions about the order of execution between threads. Output from several threads may be interleaved.

Intel compilers: **`ifort -openmp ...`**, **`icc -openmp ...`**
GNU: **`gfortran -fopenmp ...`**, **`gcc -fopenmp ...`**
Portland group: **`pgf90 -mp...`**, **`pgcc -mp ...`**

The same program in Fortran

```
program example
  use omp_lib  ! or include "omp_lib.h"
               ! or something non-standard
  implicit none
  integer                         :: i, i_am
  integer, parameter              :: n = 10000
  double precision, dimension(n) :: a, b, c

  c = 1.242d0
!$omp parallel do private(i), shared(a, b, c)
  do i = 1, n
    b(i) = 0.5d0 * i
    a(i) = 1.23d0 * b(i) + 3.45d0 * c(i)
  end do
!$omp end parallel do   ! not necessary

  print*, a(1), a(n)    ! only the master

!$omp parallel private(i_am)  ! a parallel region
  i_am = omp_get_thread_num() ! 0, ..., #threads - 1
  print*, 'i_am = ', i_am

  !$omp master
    print*, 'num threads = ', omp_get_num_threads()
    print*, 'max threads = ', omp_get_max_threads()
    print*, 'max cpus    = ', omp_get_num_procs()
  !$omp end master

!$omp end parallel

end program example
```

`!$omp` or `!$OMP`. See the standard for Fortran77.
`!$omp end ...` instead of `}`.

Things one should not do

First a silly example:

```
...
  int a, i;

#pragma omp parallel for private(i) shared(a)
  for (i = 0; i < 1000; i++) {
    a = i;
  }
  printf("%d\n", a);
...
```

Will give you different values **999**, **874** etc.

Now for a less silly one:

```
  int i, n = 12, a[n], b[n];

  for (i = 0; i < n; i++) {
    a[i] = 1; b[i] = 2;         // Init.
  }

#pragma omp parallel for private(i) shared(a, b)
  for (i = 0; i < n - 1; i++) {
    a[i + 1] = a[i] + b[i];
  }

  for (i = 0; i < n; i++)
    printf("%d ", a[i]);        // Print results.
  printf("\n");

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[1]  = a[0] + b[0]
       0         a[2]  = a[1] + b[1]
       0         a[3]  = a[2] + b[2]    <--|
                                           |  Problem
       1         a[4]  = a[3] + b[3]    <--|
       1         a[5]  = a[4] + b[4]
       1         a[6]  = a[5] + b[5]    <--|
                                           |  Problem
       2         a[7]  = a[6] + b[6]    <--|
       2         a[8]  = a[7] + b[7]
       2         a[9]  = a[8] + b[8]    <--|
                                           |  Problem
       3         a[10] = a[9]  + b[9]   <--|
       3         a[11] = a[10] + b[10]
```

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

Can "fix" the problem:

```
// Yes, you need ordered in both places

#pragma omp parallel for private(i) shared(a,b) orderec
  for (i = 0; i < n - 1; i++) {
  #pragma omp ordered
    a[i + 1] = a[i] + b[i];
  }
```

but in this case the threads do not run in parallel. Adding
`printf("%3d %3d\n", i, omp_get_thread_num());`
in the loop produces the printout:

```
  0   0
  1   0
  2   0
  3   1
  4   1
  5   1
  6   2
  7   2
  8   2
  9   3
 10   3
1 3 5 7 9 11 13 15 17 19 21 23
```

It is illegal to jump out from a parallel loop.
The following for-loop in C is illegal:

```
#pragma omp parallel for private(k, s)
  for(k = 0; s <= 10; k++) {
    ...
  }
```

It must be the same variable occurring in all three parts of the loop. More general types of loops are illegal as well, such as

```
  for(;;) {
  }
```

which has no loop variable. In Fortran, `do-while` loops are not allowed. See the standard for details.

Not all compilers provide warnings. Here a Fortran-loop with a jump.

```
program jump
  implicit none
  integer               :: k, b
  integer, parameter    :: n = 6
  integer, dimension(n) :: a

  a = (/ 1, 2, 3, 4, 5, 6 /)
  b = 1

 !$omp parallel do private(k) shared(a)
   do k = 1, n
     a(k) = a(k) + 1
     if ( a(k) > 3 ) exit  ! illegal
   end do

  print*, a
end program jump

% ifort -openmp jump.f90
fortcom: Error: jump.f90, line 13: A RETURN, EXIT or
         CYCLE statement is not legal in a DO loop
         associated with a parallel directive.
     if ( a(k) > 3 ) exit  ! illegal
---------------------^
compilation aborted for jump.f90 (code 1)

% pgf90 -mp jump.f90    the Portland group compiler
% setenv OMP_NUM_THREADS 1
% a.out
    2    3    4    4    5    6
% setenv OMP_NUM_THREADS 2
% a.out
    2    3    4    5    5    6
```

firstprivate variables

When a thread gets a private variable it is not initialised. Using `firstprivate`each thread gets an initialised copy.

In this example we use two threads:

```
...
  int i, v[] = {1, 2, 3, 4, 5};

#pragma omp parallel for private(i) private(v)
  for (i = 0; i < 5; i++)
    printf("%d ", v[i]);
  printf("\n");

#pragma omp parallel for private(i) firstprivate(v)
  for (i = 0; i < 5; i++)
    printf("%d ", v[i]);
  printf("\n");
...

% a.out
40928 10950 151804059 0 0
1 2 4 5 3   (using several threads)
```

Load balancing

We should balance the load (execution time) so that threads finish their job at roughly the same time.
There are three different ways to divide the iterations between threads, `static`, `dynamic` and `guided`. The general format is `schedule(kind of schedule, chunk size`.)

- `static`

Chunks of iterations are assigned to the threads in cyclic order. Size of default chunk, roughly = n / number of threads.

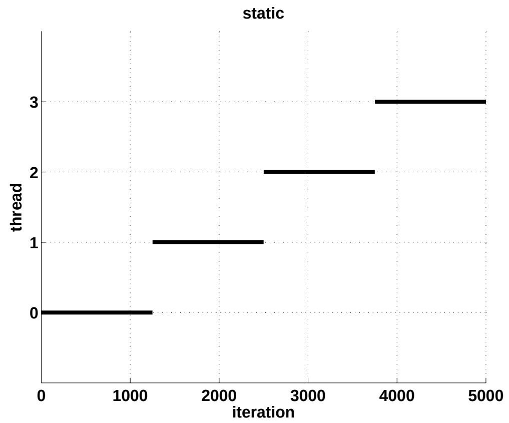
Low overhead, good if the same amount of work in each iteration. `chunk` can be used to access array elements in groups (may be me more efficient, e.g. using cache memories in better way).

Here is a small example:

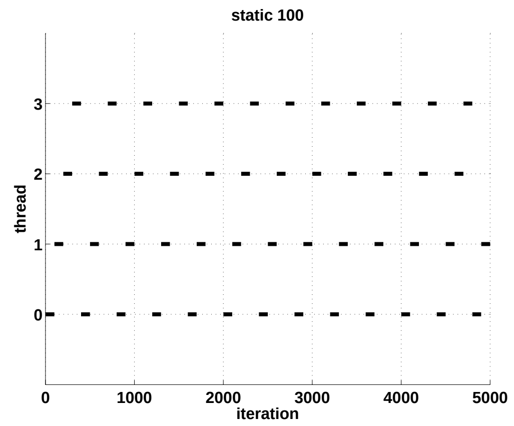
```
!$omp parallel do private(k) shared(x, n) &
!$omp          schedule(static, 4)  ! 4 = chunk
  do k = 1, n
    ...
  end do
```

```
                            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
```

Here is a larger problem, where $n = 5000$, `schedule(static)`, and using four threads.



`schedule(static, 100)`



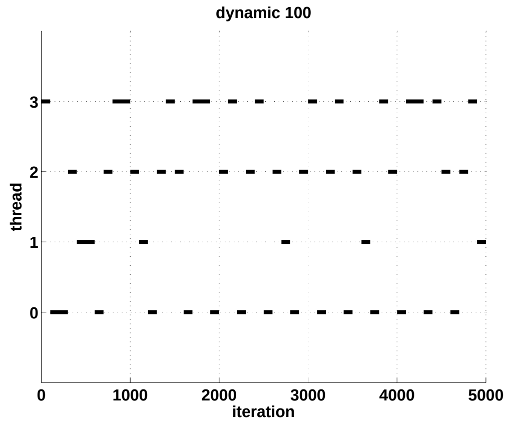
Note that if the chunk size is 5000 (in this example) only the first thread would work, so the chunk size should be chosen relative to the number of iterations.

- `dynamic`

If the amount of work varies between iterations we should use `dynamic` or `guided`. With `dynamic`, threads compete for `chunk`-sized assignments. Note that there is a synchronization overhead for `dynamic` and `guided`.

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

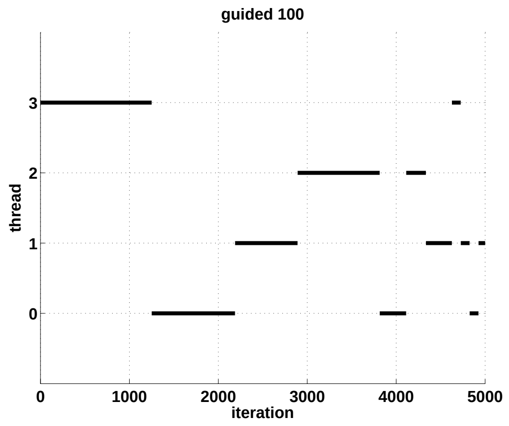
Here a run with `schedule(dynamic,100)`(`schedule(dynamic)` gives a chunks size of one). The amount of works differs between iterations in the following examples.



- `guided`

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.
Large chunks in the beginning smaller at the end. It requires fewer synchronisations than `dynamic`.



- `runtime`

It is also possible to decide the scheduling at runtime, using an environment variable, `OMP_SCHEDULE`, e.g.

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

...

  setenv OMP_SCHEDULE "guided,100"    tcsh
  export OMP_SCHEDULE=dynamic         bash
```

Suppose we parallelise 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            ! parallelise 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:

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

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:

$$\text{speedup} = \frac{T_{ser}}{T_0} \approx \frac{P}{2 - 1/P} \approx \frac{P}{2}$$

and not the optimal P.

Here is a test:

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

void work(double *);

int main()
{
  const int M = 1000, MAX_THREADS = 4;
  double s[MAX_THREADS - 1], time;
  int j, k, i_am, thr;

  for (thr = 1; thr <= MAX_THREADS; thr++) {
    omp_set_num_threads(thr);

    time = omp_get_wtime();  // a builtin function
    #pragma omp parallel private(j, k, i_am) shared(s)
    {
      i_am = omp_get_thread_num();

      #pragma omp for schedule(runtime)
      for (j = 1; j <= M; j++)
        for (k = j + 1; k <= M; k++)
          work(&s[i_am]);
    }

    printf("time = %4.2f\n", omp_get_wtime() - time);
  }

  for (j = 0; j < MAX_THREADS; j++)
    printf("%e ", s[j]);
  printf("\n");

  return 0;
}

void work(double *s)
```

```
{
  int k;

  *s = 0.0;
  for (k = 1; k <= 1000; k++)
    *s += 1.0 / k;
}
```

```
% icc -O3 -openmp load_bal.c
% setenv OMP_SCHEDULE static
% a.out      run on Ferlin (edited)
time = 3.83
time = 2.88
time = 2.13
time = 1.68
time = 1.38
time = 1.17
time = 1.02
time = 0.90

% setenv OMP_SCHEDULE "static,10"
time = 3.83
time = 1.94
time = 1.30
time = 0.99
time = 0.80
time = 0.67
time = 0.58
time = 0.51
```

`dynamic` and `guided` give the same times as `static,10`, in this case. A chunk size of 1-20 works well, but more than 50 gives longer execution times.

Note that $P/(2 - 1/P) \approx 4.3$ and $3.83/0.9 \approx 4.26$ and $3.83/0.51 \approx 7.5$. So the analysis is quite accurate in this simple case.

Do not misuse `dynamic`. Here is a contrived example:

```
...

  int k, i_am, cnt[] = { 0, 0, 0, 0 };
  double time;

  omp_set_num_threads(4);
  time = omp_get_wtime();

  #pragma omp parallel private(k, i_am) shared(cnt)
  {
    i_am = omp_get_thread_num();

    #pragma omp for schedule(runtime)
    for (k = 1; k <= 100000000; k++)
      cnt[i_am]++;
  }
  printf("time = %4.2f, cnt = %d %d %d %d\n",
          omp_get_wtime() - time,
          cnt[0], cnt[1], cnt[2], cnt[3]);
...
ferlin > setenv OMP_SCHEDULE static
time = 0.01, cnt = 25000000 25000000 25000000 25000000

ferlin > setenv OMP_SCHEDULE dynamic
time = 15.53, cnt = 25611510 25229796 25207715 23950979

ferlin > setenv OMP_SCHEDULE "dynamic,10"
time = 1.32, cnt = 25509310 24892310 25799640 23798740

ferlin > setenv OMP_SCHEDULE "dynamic,100"
time = 0.13, cnt = 29569500 24044300 23285700 23100500

ferlin > setenv OMP_SCHEDULE guided
time = 0.00, cnt = 39831740 5928451 19761833 34477976
```

The reduction clause

```
...
  int i, n = 10000;
  double x[n], y[n], s;

  for (i = 0; i < n; i++) {
    x[i] = 1.0; y[i] = 2.0;   // Init.
  }
   s = 0.0;
  #pragma omp parallel for private(i) shared(n, x, y)
  reduction(+: s)   // all on the same line
  for (i = 0; i < n; i++)
    s += x[i] * y[i];
...
```

In general: `reduction(operator: variable list.)`
Valid operators are: `+, *, -, &, |, ^, &&, ||`.
A reduction is typically specified for statements of the form:

```
  x = x op expr
  x = expr op x  (except for subtraction)
  x binop= expr
  x++
  ++x
  x--
  --x
```

where `expr` is of scalar type and does not reference `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

In Fortran:
`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 is 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`**.

Note that **`x`** may be an array in Fortran (vector reduction) but not so in C. Here is a contrived example which computes a matrix-vector product:

```
...
  double precision, dimension(n, n) :: A
  double precision, dimension(n)    :: s, x

  A = ...  ! initialize A and b
  b = ...
  s = 0.0d0

  !$omp parallel do shared(A, x) reduction(+: s) &
  !$omp  private(i) default(none)
  do i = 1, n
    s = s + A(:, i) * x(i)  ! s = A * x
  end do
...
```

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 private_s, shared_s;
...

  shared_s = 0.0;

// a parallel region
#pragma omp parallel private(private_s)
            shared(x, y, shared_s, n)
  {
    private_s = 0.0;  // Done by each thread

    #pragma omp for private(i)  // A parallel loop
    for (i = 0; i < n; i++)
      private_s += x[i] * y[i];

    // Here we specify a critical section.
    // Only one thread at a time may pass through.

    #pragma omp critical
    shared_s += private_s;
  }
...
```

Vector reduction in C

Here are two alternatives in C (and Fortran if the compiler does not support vector reduction).

We introduce a private summation vector, **`partial_sum`**, one for each thread.

```
...
  for(k = 0; k < n; k++)  // done by the master
    shared_sum[k] = 0.0;

  #pragma omp parallel private(partial_sum, k) \
                       shared(shared_sum) ...
  {
  // Each thread updates its own partial_sum.
  // We assume that this is the time consuming part.
    for( ...
      partial_sum[k] = ..
  ...

  // Update the shared sum in a safe way.
  // Not too bad with a critical section here.
    #pragma omp critical
    {
      for(k = 0; k < n; k++)
        shared_sum[k] += partial_sum[k];
    }
  } // end parallel
...
```

We can avoid the critical section if we introduce a shared matrix where each row (or column) corresponds to the **`partial_sum`** from the previous example.

```
...
  for(k = 0; k < n; k++)  // done by the master
    shared_sum[k] = 0.0;

  #pragma omp parallel private(i_am) \
              shared(S, shared_sum, n_threads)
  {
    i_am = omp_get_thread_num();

    for(k = 0; k < n; k++)  // done by all
      S[i_am][k] = 0.0;

   // Each thread updates its own partial_sum.
   // We assume that this is the time consuming part.
    for( ...
      S[i_am][k] = ..

   // Add the partial sums together.
   // The final sum could be stored in S of course.
    #pragma omp for
    for(k = 0; k < n; k++)
      for(j = 0; j < n_threads; j++)
        shared_sum[k] += S[j][k];

  } // end parallel
...
```

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 Ferlin using `ifort -O3 ...`. The loops were run ten times. Times in seconds for one to four threads. `dgemv` from MKL takes 0.23s, 0.21s, 0.34s for the three cases and the builtin `matmul` takes 1.0s, 0.74s, 1.1s. *Use BLAS!*

m	n	first loop				second loop			
		1	2	3	4	1	2	3	4
4000	4000	0.41	0.37	0.36	0.36	2.1	1.2	0.98	0.83
40000	400	0.39	0.32	0.27	0.23	1.5	0.86	0.72	0.58
400	40000	0.49	1.2	1.5	1.7	1.9	2.0	2.3	2.3

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

A few other OpenMP directives, C

```
#pragma omp parallel shared(a, n)

        ... code run in parallel

#pragma omp single   // only ONE thread will
{                    // execute the code
        ... code
}

#pragma omp barrier  // wait for all the other threads
      ... code

// don't wait (to wait is default)
#pragma omp for nowait
    for ( ...

    for ( ... // all iterations run by all threads

#pragma omp sections
{
#pragma omp section
          ... code executed by one thread
#pragma omp section
          ... code executed by another thread
} // end sections, implicit barrier

# ifdef _OPENMP
  C statements ...  Included if we use OpenMP,
  but not otherwise (conditional compilation)
# endif

} // end of the parallel section
```

A few other OpenMP directives, Fortran

```
!$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 ... ! all iterations run by all threads
      end do

!$omp sections
!$omp   section
          ... code executed by one thread
!$omp   section
          ... code executed by another thread
!$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
```

Misuse of critical, atomic

Do not use critical sections and similar constructions too much.
This test compares three ways to compute a sum.
We try reduction, critical and atomic. $n = 10^7$.

```
...
  printf("n_thr    time, reduction\n");
  for(n_thr = 1; n_thr <= 4; n_thr++) {
    omp_set_num_threads(n_thr);
    s = 0.0;
    t = omp_get_wtime();

    #pragma omp parallel for reduction(+: s) private(i)
    for (i = 1; i <= n; i++)
      s += sqrt(i);

    printf("%3d %10.3f\n", n_thr, omp_get_wtime() - t);
  }
  printf("s = %e\n", s);
```

Change the inner loop to

```
    #pragma omp parallel for shared(s) private(i)
    for (i = 1; i <= n; i++) {
      #pragma omp critical
      s += sqrt(i);
    }
```

and then to

```
    #pragma omp parallel for shared(s) private(i)
    for (i = 1; i <= n; i++) {
      #pragma omp atomic
      s += sqrt(i);
    }
```

`atomic` updates a single variable atomically.

Here are the times (on Ferlin):

```
n_thr    time, reduction
  1      0.036
  2      0.020
  3      0.014
  4      0.010

n_thr    time, critical
  1      0.666
  2      5.565
  3      5.558
  4      5.296

n_thr    time, atomic
  1      0.188
  2      0.537
  3      0.842
  4      1.141
```

We get a slowdown instead of a speedup, when using **`critical`** or **`atomic`**.

workshare

Some, but not all, compilers support parallelisation 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(b) + 3.0 * sin(c)
!$omp   end workshare
!$omp end parallel
  ... code
```

or shorter

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

Subroutines and OpenMP

Here comes a first example of where we call a subroutine from a parallel region. If we have time leftover there will be more at the end of the lecture. Formal arguments of called routines, that are passed by reference, inherit the data-sharing attributes of the associated actual parameters. Those that are passed by value become private. So:

```
  void work( double [], double [], double, double,
             double *,  double *);
  ...
  double  pr_vec[10], sh_vec[10], pr_val, sh_val,
          pr_ref,     sh_ref;
  ...

#pragma omp parallel private(pr_vec, pr_val, pr_ref)
                     shared( sh_vec, sh_val, sh_ref)
{
  work(pr_vec, sh_vec, pr_val, sh_val, &pr_ref, &sh_ref
}
...

void work(double pr_vec[], double sh_vec[],
          double pr_val,   double sh_val,
          double *pr_ref,  double *sh_ref)
{
  // pr_vec becomes private
  // sh_vec becomes shared
  // pr_val becomes private
  // sh_val becomes PRIVATE, each thread has its own
  // pr_ref becomes private
  // sh_ref becomes shared

  int k;  // becomes private
 ...
}
```

In Fortran alla variables are passed by reference, so they inherit the data-sharing attributes of the associated actual parameters.

Here comes a simple example in C:

```
#include <stdio.h>
#include <omp.h>
void work(int[], int);

int main()
{
  int a[] = { 99, 99, 99, 99 }, i_am;

  omp_set_num_threads(4);

  #pragma omp parallel private(i_am) shared(a)
  {
    i_am = omp_get_thread_num();
    work(a, i_am);

    #pragma omp single
     printf("a = %d, %d, %d, %d\n",
             a[0], a[1], a[2], a[3]);
  }
  return 0;
}

// a[] becomes shared, i_am becomes private

void work(int a[], int i_am)
{
  int k; // becomes private (not used in this example)

  printf("work %d\n", i_am);
  a[i_am] = i_am;
}
```

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

Print after the parallel region or add a barrier:

```
#pragma omp barrier
#pragma omp single
    printf("a = %d, %d, %d, %d\n",
            a[0], a[1], a[2], a[3]);
% a.out
 work 0
 work 1
 work 3
 work 2
 a =  0,  1,  2,  3
```

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.

Case study: 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 ACML.

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)`
  <u>no</u> 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 <u>not</u> work either; we get an <u>uninitialised</u> copy. `firstprivate` is the proper choice, it makes a private and initialised copy.

```
c$omp  parallel do private(j, yj, r, fac, ftem)
c$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)
```

More on OpenMP and subprograms

So far we have essentially executed a main program containing OpenMP-directives. Suppose now that we call a function, containing OpenMP-directives, from a parallel part of the program, so something like:

```
int main()
{
  ...
  #pragma omp parallel ...  ---
  {                            |
    #pragma omp for ...        | lexical extent of
    ...                        |
    work(...);                 | the parallel region
    ...                        |
  }                         ---
  ...
}

void work(...)
{
  ...
  #pragma omp for            ---
  for (...) {                  | dynamic extent of the
     ...                       | parallel region
    }                        ---
  ...
}
```

The `omp for` in `work` is an orphaned directive (it appears in the dynamic extent of the parallel region but not in the lexical extent). This `for` binds to the dynamically enclosing parallel directive and so the iterations in the `for` will be done in parallel (they will be divided between threads).

Suppose now that `work` contains the following three loops and that we have three threads:

```
...
  int k, i_am;
  char f[] = "%1d:%5d %5d %5d\n";  // a format

  #pragma omp master
  printf("     i_am   omp()  k\n");

  i_am = omp_get_thread_num();

  #pragma omp for private(k)
  for (k = 1; k <= 6; k++)      // LOOP 1
    printf(f, 1, i_am, omp_get_thread_num(), k);

  for (k = 1; k <= 6; k++)      // LOOP 2
    printf(f, 2, i_am, omp_get_thread_num(), k);

  #pragma omp parallel for private(k)
  for (k = 1; k <= 6; k++)      // LOOP 3
    printf(f, 3, i_am, omp_get_thread_num(), k);
...
```

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 serialised. 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 `omp_set_nested(1)`.
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 <u>may</u> look like. `omp()` is the value returned 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 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              $\mathcal{O}(j)$
  and check for convergence
end for
```

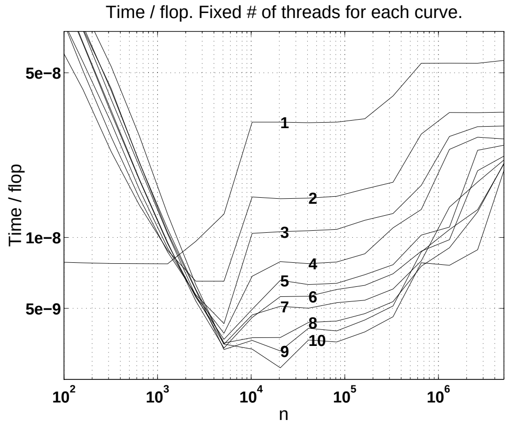
The diagonal of T_j is $\alpha_1, \ldots, \alpha_j$ and the sub- and super-diagonals contain $\beta_1, \ldots, \beta_{j-1}$.

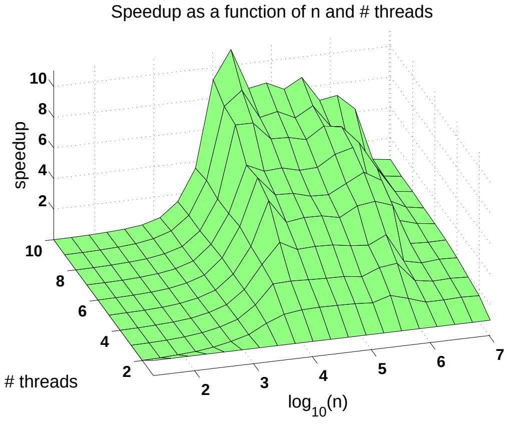
How can we parallelise this algorithm?

- 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 may 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
```

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

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

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

New in OpenMP v3.0

What we have seen so far is primarily data parallelism (SIMD), all threads perform the same operations but on different data. If different threads should perform different operations, so-called function parallelism, or task parallelism (MIMD), we can use the `sections`-construct, or do like this:

```
!$omp parallel private(i_am ...
  ...
  i_am = omp_get_thread_num()
  if ( i_am == 0 ) then
    ! do work 0, e.g   call sub0(...)
      ...
  else if ( i_am == 1 ) then
    ! do work 1
      ...
  else if ( i_am == 2 ) then
      ...
  end if
!$omp end parallel
```

In v3.0 there is better support for "tasks", and one could write things like:

```
!$omp parallel private(...
  !$omp task
    call sub0(...)
  !$omp end task

  do k = 1, n
    !$omp task
      call work(k)
    !$omp end task
  end do

!$omp end parallel
```