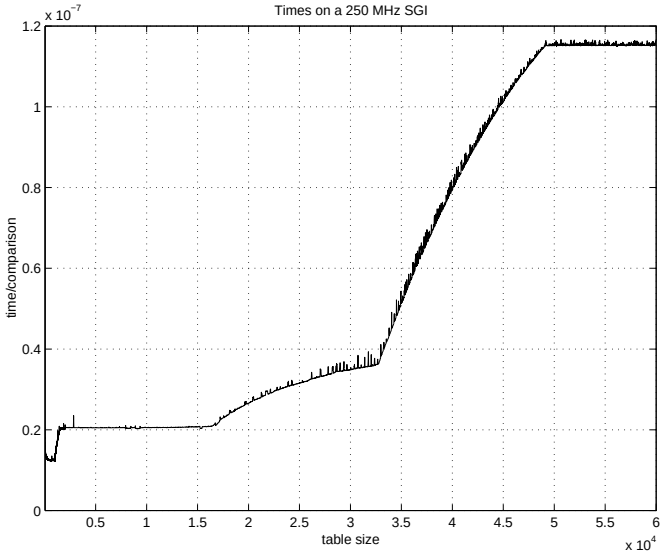
Lecture Notes
on
High Performance Computing



Thomas Ericsson
Mathematics
Chalmers/GU
2012

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/tma881/1112

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 $\geq$ 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.

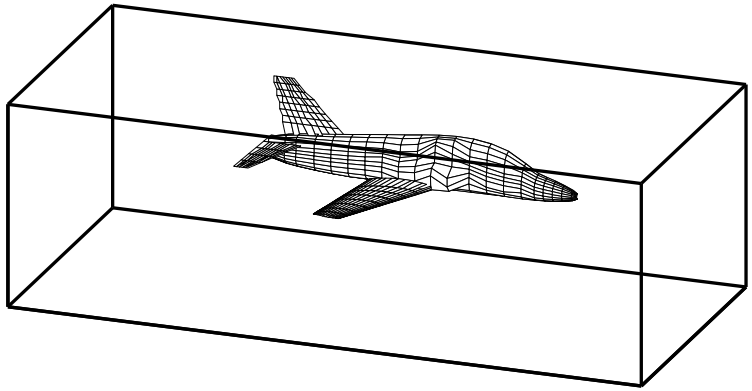
- Is this reasonable, is the problem really this hard?
- Are Matlab and **quad** good tools for such problems?
- How can one handle singularities and infinite intervals?

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

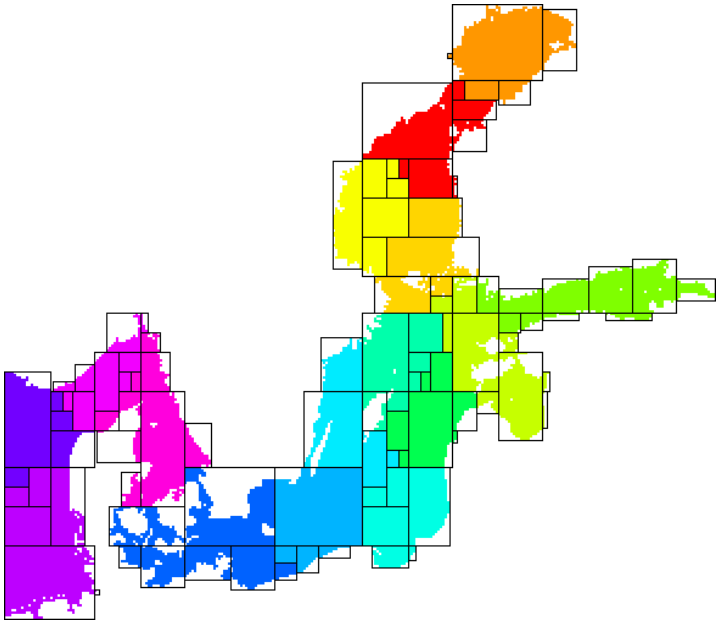
SMHI, Swedish Meteorological and Hydrological Institute.
HIRLAM (HIgh Resolution Limited Area Model).
HIROMB (HIgh Resolution Operational Model of the Baltic).

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.

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 C and Fortran90 (77) for the labs, tutorial.
- 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.
- A few words about GPU-programming, CUDA.

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. There are a fair number of HPC-books 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 books, have a look at the homepage under Documentation.

Read: "Introduction to C, Fortran 90, FORTRAN 77, tcsh and bash" which you can find on the homepage under Diary. I no longer lecture C and Fortran

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.

Operating Systems

Most HPC-systems are Linux/Unix-based. Here is a list from `www.top500.org` listing the operating systems used on the 500 fastest systems in the world (Nov 2011).

OS family	Count	Share %
Linux	457	91.4 %
Unix	30	6.0 %
Mixed	11	2.2 %
Windows	1	0.2 %
BSD Based	1	0.2 %

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).

  You can switch off/on JIT by `feature accel off`, `feature accel on` Try it!
- 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**.

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
     gcc -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.

A few words about header files. Suppose `main.c` depends on `defs.h` and `params.h`. `main.c` calls `sub1.c` and `sub2.c`, where `sub2.c` depends on `defs.h` and `constants.h`, which in turn includes `const.h`. A suitable makefile might be:

```
OBJS   = main.o sub1.o sub2.o
CC     = gcc
CFLAGS = -O3
LIBS   = -lm

a.out: $(OBJS)
	$(CC) -o $@  $(CFLAGS) $(OBJS) $(LIBS)

main.o: defs.h params.h
sub2.o: defs.h constants.h const.h

# Remove objects and executable
clean:
	rm -f $(OBJS) a.out
```

In can be complicated to create the header file dependencies, the `makedepend`-program may help. Say we have this makefile (named `makefile` or `Makefile`, with no header file dependencies):

```
OBJS   = main.o sub1.o sub2.o
CC     = gcc
CFLAGS = -O3
LIBS   = -lm

a.out: $(OBJS)
	$(CC) -o $@  $(CFLAGS) $(OBJS) $(LIBS)

# Remove objects and executable
clean:
	rm -f $(OBJS) a.out
```


`makedepend` on the source files will append the dependencies on the existing makefile:

```
% makedepend *.c
% cat Makefile
OBJS   = main.o sub1.o sub2.o
CC     = gcc
CFLAGS = -O3
LIBS   = -lm

a.out: $(OBJS)
	$(CC) -o $@  $(CFLAGS) $(OBJS) $(LIBS)

# Remove objects and executable
clean:
	rm -f $(OBJS) a.out
# DO NOT DELETE

main.o: defs.h params.h
sub2.o: defs.h constants.h const.h
```

This is how the new makefile works:

```
% make   let us assume a.out is up to date
'a.out' is up to date.

% touch defs.h     changing defs.h
% make
gcc -O3   -c -o main.o main.c
gcc -O3   -c -o sub2.o sub2.c
gcc -o a.out  -O3 main.o sub1.o sub2.o -lm
```

`makedepend` works for Fortran as well provided you use `#include`.

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

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 32-bit student system, 2010
% a.out
sizeof(short int) = 4
sizeof(int)       = 4
sizeof(long int)  = 4

% a.out            On the 64-bit student system
sizeof(short int) = 4
sizeof(int)       = 4
sizeof(long int)  = 8   4 if gcc -m32
```

Running the 32-bit binary on the 64-bit system behaves like the 32-bit system. One cannot run the 64-bit binary om the 32-bit system.

CISC (Complex Instruction Set Computers) before $\approx$ 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 $\Rightarrow$ 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
- $\approx$ 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)

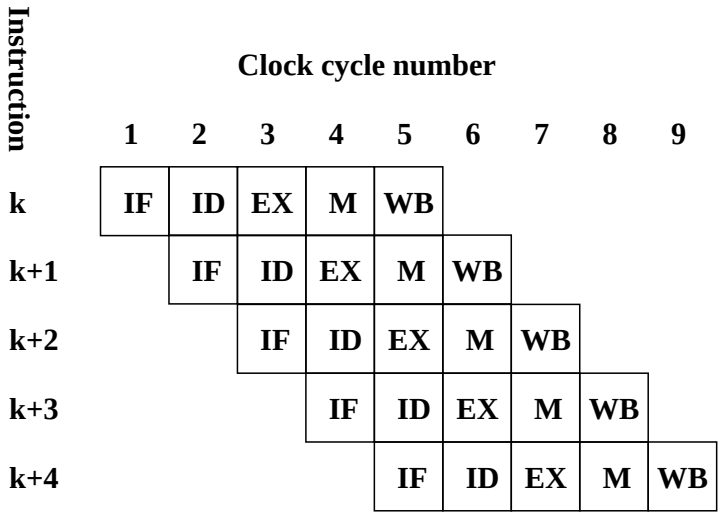
The most widely-used type of microprocessor, the x86 (Intel), is CISC rather than RISC, although the internal design of newer x86 family members is said to be RISC-like. All modern CPUs share some RISC characteristics, although the details may differ substantially.

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.



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.
- 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 10-20 cycles:

```
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 cores × flop / s =
# of cores × # flop / clock cycle × clock frequency

Looking in `instruction_tables.pdf` at `www.agner.org` one can find out the performance of several CPUs. One core in the student machines (Intel Pentium Dual-core, E6300, Wolfdale-3M) can, in the best of cases, finish 1 addition and 0.5 multiplication per clock cycle using x87-instructions. Divisions, which are not pipelined, may take up to 20 cycles.

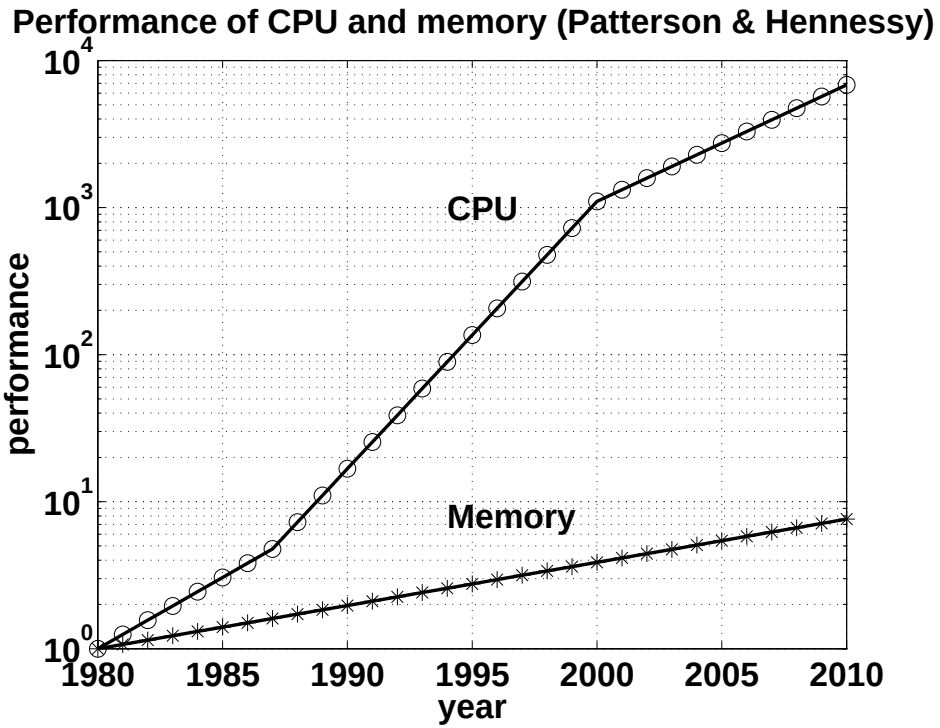
It is, however, common with vector units, like Intel's SSE, in modern CPUs. These units can work in parallel on short vectors of numbers.

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.

Each core in the lab-computers can execute a vectorized add and a vectorized multiply operation per cycle. Each operation can work on two double (or four single) precision numbers in parallel. Division is still slow.

See `www.spec.org` for benchmarks with real applications.

Memory is the problem - caches

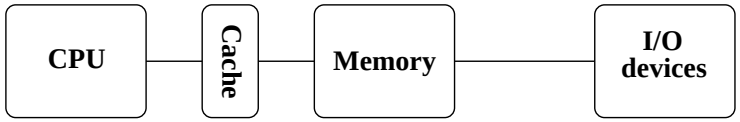


CPU: increase 1.35 improvement/year until 1986,
and a 1.55 improvement/year thereafter.

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

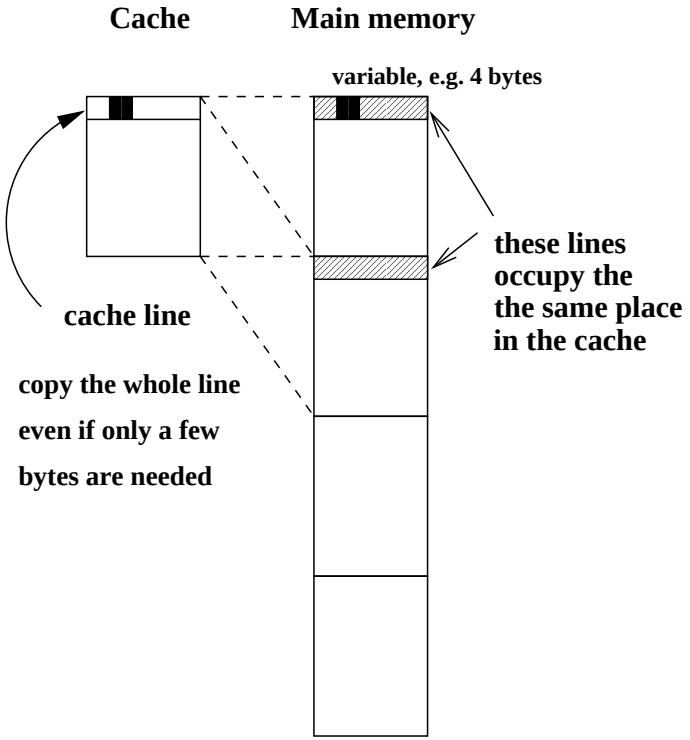
Use SRAM (static random access memory) fast & expensive for cache.

Direct mapped cache

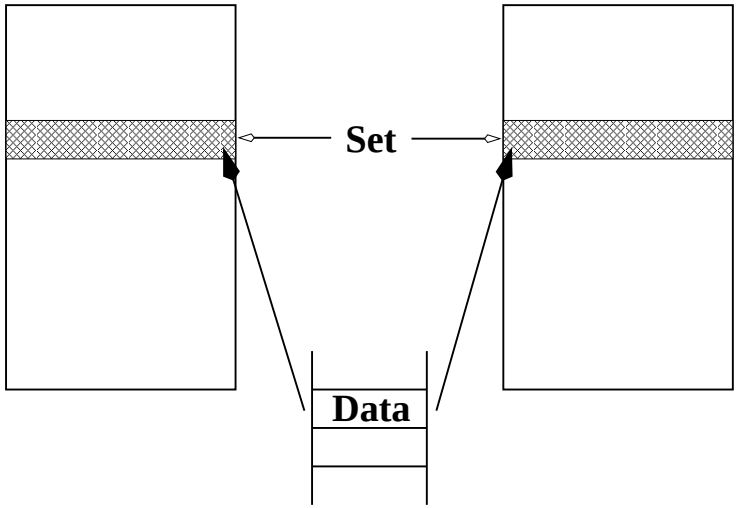


The cache is a small and fast memory used for storing both instructions and data.

This is the simplest form of cache-construction.

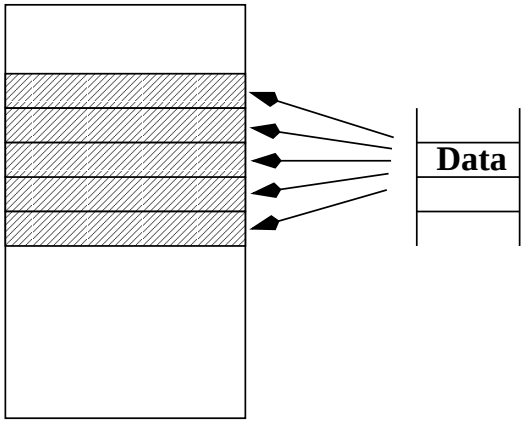


There are more general cache constructions.
This is a two-way set associative cache:



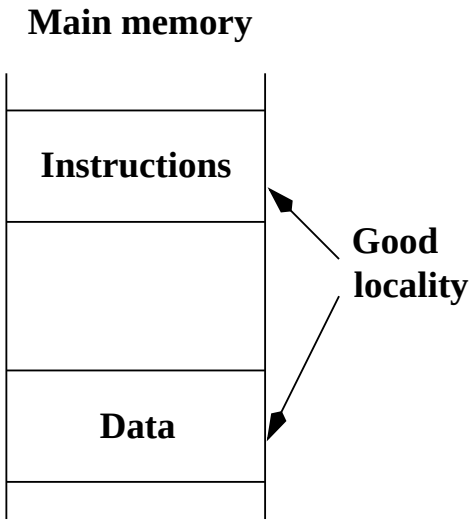
A direct mapped cache is one-way set associative.

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)

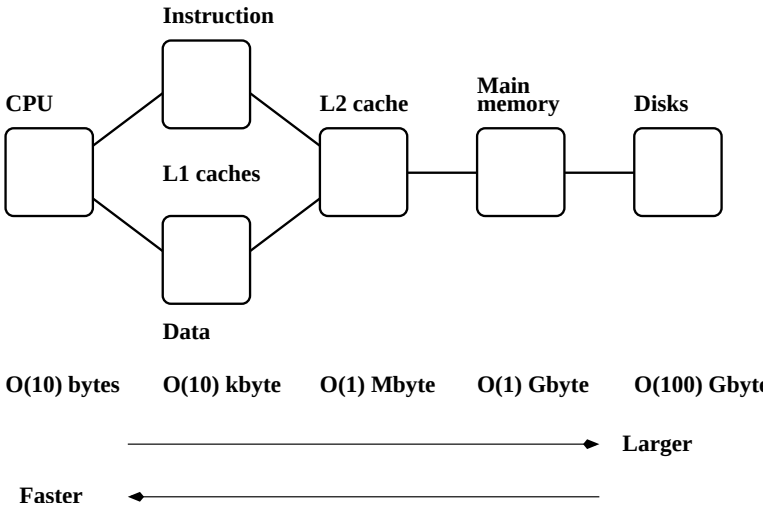


Not necessarily good locality together.

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 student machines

(Some) Intel and AMD cpu:s 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.
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. There is a cpu_id-code available from **`http://linux.softpedia.com/`**(search for cpuid) .
You find info. in **`/proc/cpuinfo`**and **`/proc/meminfo`**as well.
These files and the above program provide the following information (and more) about the student machines:

```
Model: Intel Core i5-650, 3.20 GHz.

L1 Data Cache:   32 k,  8-way
L2 Cache:       256 k,  8-way
L3 Cache:         4 M, 16-way

All the cashes have a 64-byte line size.

The TLB has several levels (like the ordinary caches).
data TLB:        4-way,  64 entries
instruction TLB: 4-way,  64 entries
L2 TLB:          4-way, 512 entries

The system has a pagesize of 4 kbyte.
% getconf PAGESIZE
4096

4 Gbyte main memory

Two cores and hyper-threading, so /proc/cpuinfo and
top report four cores.
```

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**.

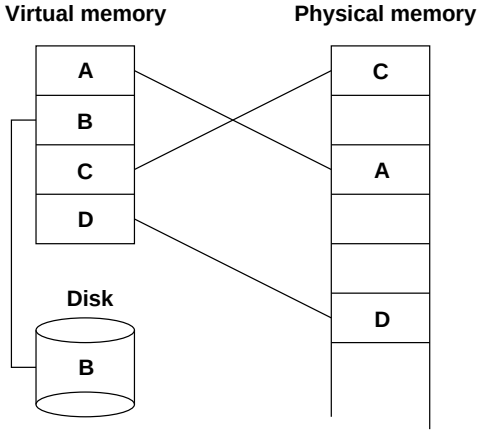
Unlike the Sparc, the AMD can multiply with an operand in memory (the number does not have to be fetched first). So the **fmull** multiplies the top-element on the stack with the number at address **%ecx + %eax*8** and replaces the top-element with the product.

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).

Please see the separate Beamer-file, Code Optimization.

Low level profiling

valgrind and PAPI are two tools for counting cache misses.

http://valgrind.org/.

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
```

I have installed the latest version under:
/chalmers/sw/unsup64/valgrind-3.7.0 (binaries in **bin**).

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];  // Bad locality
}
```

Compile with **-g -O3** (try without **-O3** and see the difference):

```
% gcc -g -O3 main.c sub.c
```

I have edited the following printout:

```
% valgrind --tool=cachegrind a.out
Cachegrind, a cache and branch-prediction profiler
Copyright (C) 2002-2011, and GNU GPL'd, by
Nicholas Nethercote et al.
Using Valgrind-3.7.0 and LibVEX;
rerun with -h for copyright info
Command: a.out

9.990000e+08 8.464193e-02
```

```
I   refs:      13,117,835
I1  misses:           855
LLi misses:           852
I1  miss rate:       0.00%
LLi miss rate:       0.00%

D   refs:      3,039,758  (1,028,285 rd + 2,011,473 wr)
D1  misses:    1,126,162  (1,000,876 rd +   125,286 wr)
LLd misses:      246,498  (  121,231 rd +   125,267 wr)
D1  miss rate:      37.0% (     97.3%   +       6.2%  )
LLd miss rate:       8.1% (     11.7%   +       6.2%  )

LL refs:       1,127,017  (1,001,731 rd +   125,286 wr)
LL misses:       247,350  (  122,083 rd +   125,267 wr)
LL miss rate:        1.5% (      0.8%   +       6.2%  )
```

LL last level (three on our machines).
valgrind produced the file, **cachegrind.out.21981**, in this run, (**21981** is a pid).
To see what source lines are responsible for the cache misses we use **cg_annotate cachegrind.out.21981 -auto=yes** I have edited the listing and removed the columns dealing with the instruction caches (the lines are too long otherwise).

```
      Dr    D1mr    DLmr         Dw D1mw DLmw
       0       0       0          1    0    0  *s = 0;
       1       1       1          0    0    0  for (j = 0
       0       0       0          0    0    0  for (k = 0
1,000,000 999,995 120,408 1,000,000    0    0 *s += A[k]
```

Dr: data cache reads (ie. memory reads).
D1mr: L1 data cache read misses.
DLmr: L3 cache data read misses.
Dw: D cache writes (ie. memory writes).
D1mw: L1 data cache write misses.
DLmw: L3 cache data write misses.

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

```
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 are the counts:

```
   Dr     D1mr    D3mr      Dw
1000000  999995  120408  1000000  original
1000000  124999  124999  1000000  interchange
1000000  999999  125407        1  local_s
1000000  125000  125000        1  local_s, interchange
```

valgrind cannot count TLB-misses, so switch to PAPI, which can. PAPI = Performance Application Programming Interface
http://icl.cs.utk.edu/papi
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).

Update 2012: **papiex** is not supported any longer (but can be fetched from **icl.cs.utk.edu/~mucci/papiex/**
An alternative is to make calls from withing the source code.

```
% 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 local	icc -O3 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** 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
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 **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:

```
gfortran -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: Solve $Ax = b$ where A is a 5000×5000 non-singular matrix. Test Lapack's **dgesv** and Linpack's **dgefa** and **dgesl**. First **gprof**:

```
% gprof | less
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.

First the Lapack-run. The following has been compressed sideways.

```
Each sample counts as 0.01 seconds.
 %   cumulative  self           self    total
time   seconds  seconds  calls  s/call  s/call  name
97.54    42.80   42.80     78    0.55    0.55  dgemm_
 1.37    43.40    0.60     80    0.01    0.01  dtrsm_
 0.93    43.81    0.41   4921    0.00    0.00  dger_
 0.16    43.88    0.07      1    0.07   43.90  MAIN__
 0.05    43.90    0.02   5000    0.00    0.00  idamax_
 0.00    43.90    0.00   4999    0.00    0.00  dscal_
 0.00    43.90    0.00    875    0.00    0.00  lsame_
 0.00    43.90    0.00    158    0.00    0.00  dlaswp_
 0.00    43.90    0.00     79    0.00    0.01  dgetf2_
 0.00    43.90    0.00     79    0.00    0.00  dlamch_
 0.00    43.90    0.00      1    0.00   43.83  dgesv_
 0.00    43.90    0.00      1    0.00   43.82  dgetrf_
 0.00    43.90    0.00      1    0.00    0.02  dgetrs_
 0.00    43.90    0.00      1    0.00    0.00  ilaenv_
```


% time: the percentage of the total running time of the program used by this function.

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.

name: the name of the function. This is the minor sort for this listing. The index shows the location of the function in the gprof listing. If the index is in parenthesis it shows where it would appear in the gprof listing if it were to be printed.

The top three routines are BLAS-routines:
dgemm, matrix multiplication
dtrsm, solves a triangular system
dger, rank-one update.

It is *very important with fast BLAS*. using Goto-BLAS, the execution times goes down from 43 s to 7.5 s.

Using Linpack (BLAS1-routines) takes 51.6 s.
Goto-BLAS cannot help much, 45.9 s.

```
 %   cumulative   self              self    total
time   seconds   seconds  calls   s/call  s/call  name
98.06    50.83   50.83 12507499   0.00    0.00  daxpy_
 1.78    51.75    0.92        1   0.92   51.75  dgefa_
 0.14    51.82    0.07        1   0.07   51.86  MAIN__
 0.06    51.85    0.03     4999   0.00    0.00  dscal_
 0.02    51.86    0.01     4999   0.00    0.00  idamax_
 0.00    51.86    0.00        1   0.00    0.04  dgesl_
```

Let us run **gcov** on **daxpy**.

```
Part of the output (compressed):
% gcov daxpy.o
File 'daxpy.f'
Lines executed:66.67% of 24
daxpy.f:creating 'daxpy.f.gcov'
 ...
10427606236:   40:    do 50 i = mp1,n,4
10415098750:   41:      dy(i)   = dy(i)   + da*dx(i)
10415098750:   42:      dy(i+1) = dy(i+1) + da*dx(i+1)
10415098750:   43:      dy(i+2) = dy(i+2) + da*dx(i+2)
10415098750:   44:      dy(i+3) = dy(i+3) + da*dx(i+3)
        -:   45:   50 continue
 ...
```


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 is the Lapack-run.

```
index %time self  children  called   name
                                ...  deleted lines
-----------------------------------------------
             0.00   43.82     1/1         dgesv_ [3]
[4]  99.8    0.00   43.82     1       dgetrf_ [4]
            42.80    0.00    78/78        dgemm_ [5]
             0.59    0.00    78/80        dtrsm_ [6]
             0.00    0.43    79/79        dgetf2_ [7]
             0.00    0.00   157/158       dlaswp_ [13]
             0.00    0.00     1/1         ilaenv_ [15]
-----------------------------------------------
            42.80    0.00    78/78        dgetrf_ [4]
[5]  97.5   42.80    0.00    78       dgemm_ [5]
             0.00    0.00   156/875       lsame_ [12]
-----------------------------------------------
                                ...  deleted lines
```

Each routine has an index (see table at the end) and is presented between **---**lines. Let us look at **dgemm**.

42.8s was spent in **dgemm** itself, 97.5% of total (including calls from **dgemm**). **dgetrf** (parent) called **dgemm** which in turn called **lsame**, a child.

dgetrf made 78 out of 78 calls of **dgemm**. **dgemm** called **lsame** 156 out of 875 times.

See the documentation for an explanation of all the times (**self** and **children**).

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 number
        of columns of the matrix B.  NRHS >= 0.

A       (input/output) DOUBLE PRECISION array, dimension
        (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, dimension
        (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>

#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 = 10, lda = n, ldb = n, nrhs = 1,
           info, row, col, ipiv[n];
  double   A[lda][n], b[n], s, max_err;

  // Make sure you have the correct mix of types.
  printf("sizeof(int) = %ld\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;
  }
}
```


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 to 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 C99-style comments).
I will assume we use a 64-bit system.

```
#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[])

// See the C-tutorial for a discussion of const.

{
  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);
  if ( nlhs == 2 )  // if two output arguments
    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);

  if (info)
    mexWarnMsgTxt("Non zero info from dpbsv.");
  if ( nlhs == 2 )
    *mxGetPr(plhs[1]) = info; // () higher prec.
                              // than  *

  // 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**. Since we would like to change some parameters when compiling, we will copy and edit an options file, **mexopts.sh**

```
% which matlab
/chalmers/sw/sup64/matlab-2011b/bin/matlab
(ls -ld /chalmers/sw/sup64/matlab to see the versions)

% cp /chalmers/sw/sup64/matlab-2011b/bin/mexopts.sh .
% chmod u+w mexopts.sh     add write permissons for you
```

Edit **mexopts.sh** and search for **glnxa64**, change

```
  CFLAGS='-ansi -D_GNU_SOURCE'
to
  CFLAGS='-Wall -std=c99 -D_GNU_SOURCE'
```

to get more warnings and to use C99-style comments.

Use default **gfortran** or you may have link-problems.

Now it is time to compile, I assume we have the Fortran-files available:

```
% mex -f ./mexopts.sh bandsolve.c *.f
```

which creates **bandsolve.mexa64**

You will get an error message:
Warning: You are using gcc version "4.1.2".etc. but it seems to work OK anyhow.

We can now 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'
ans = 1.0000e+00 1.0000e+00 1.0000e+00 1.0000e+00 1.0000
>> info
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 equals b, since b is copied to x
>> info
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=50:

```
>> tic; [x, info] = bandsolve(A, b); toc
Elapsed time is 0.099192 seconds.

>> tic; [x, info] = bandsolve(A, b); toc
Elapsed time is 0.055137 seconds.

>> tic; [x, info] = bandsolve(A, b); toc
Elapsed time is 0.055036 seconds.
```

Now to some larger problems:
With n=1000000 and kd=10, **dpbsv** takes 0.9 s and sparse backslash 1.3 s on the 64-bit math compute server.
kd=20 gives the times 1.3 s and 2.5 s respectively.

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 required 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 not 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-2.5.so

% ls -l /lib/libm-2.5.so
-rwxr-xr-x 1 root root 208352  6 jan  2009
    /lib/libm-2.5.so*
```

What is this last file?

```
% ar -t /lib/libm-2.5.so
ar: /lib/libm-2.5.so: File format not recognized

Look for symbols (names of functions etc.):
% objdump -t /lib/libm-2.5.so | grep expm1
 ...
009fa690  w    F .text  0000005b   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 sub.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 must
  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** (Bourne shell) you may do something like:

```
$ LD_LIBRARY_PATH=$LD_LIBRARY_PATH:Examples
$ 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 sub.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/Makefile
```

One would usually compress it:

```
% gzip My_package.tar  (or using bzip2)   or
% tar zcvf My_package.tz My_package  or  tar jcvf ...
```

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

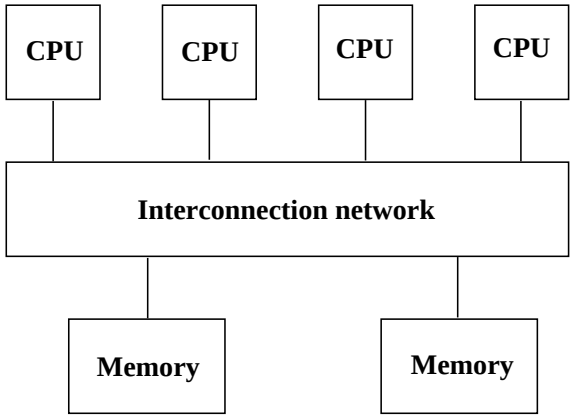
Asynchronous (the processes work independently).

- 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.

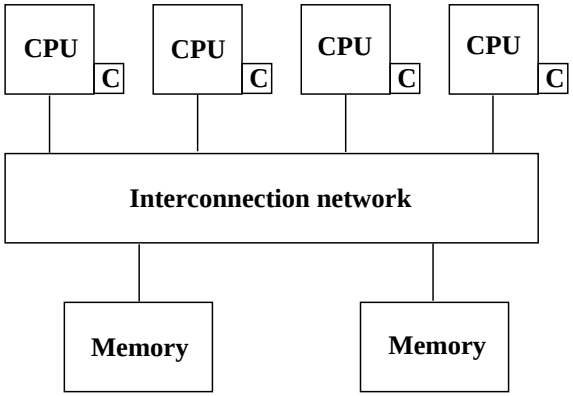
Shared memory

Bus-based architecture



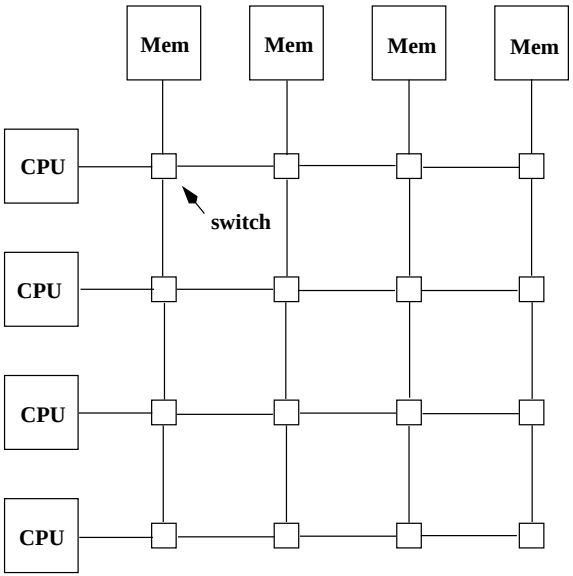
- Limited bandwidth (the amount of data that can be sent through a given communications circuit per second).
- Do not scale to a large number of processors. 30-40 CPUs common.

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



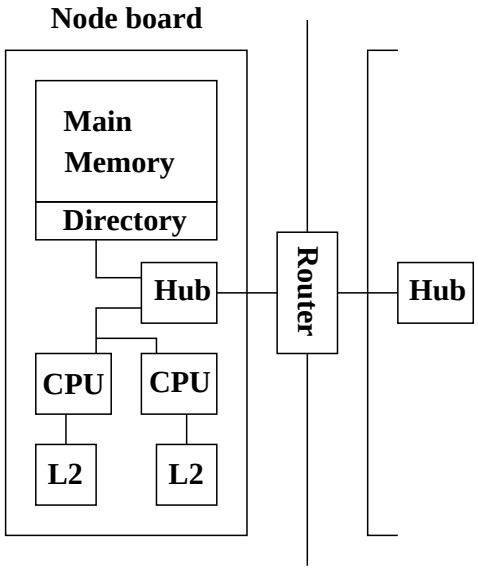
I have denoted the caches by C. Cache coherence.

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

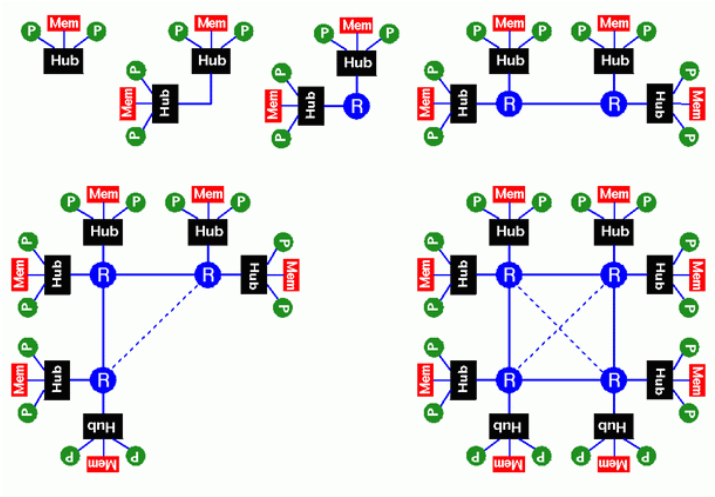


- Any processor can access any memory module. Any other processor can simultaneously access any other memory module.
- Expensive.
- Common with a memory hierarchy. Several crossbars may be connected by a cheaper network. NonUniform Memory Access (NUMA).

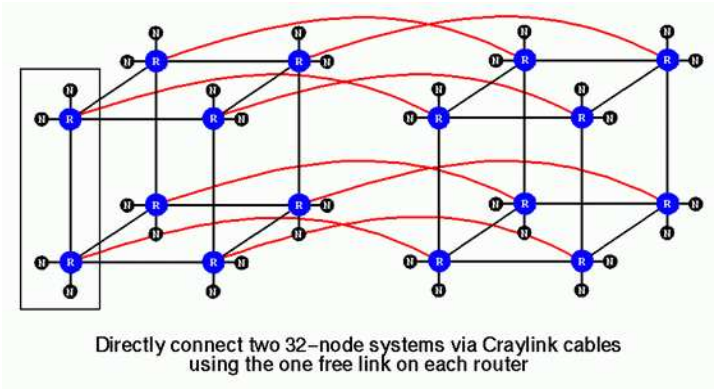
Example of a NUMA architecture: SGI Origin 2000, R10000 CPUS connected by a fast network.



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)?

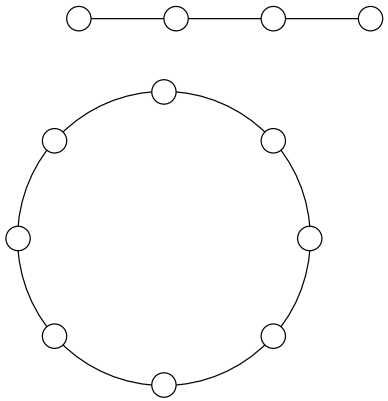
A simple model for communication:

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

Distributed memory

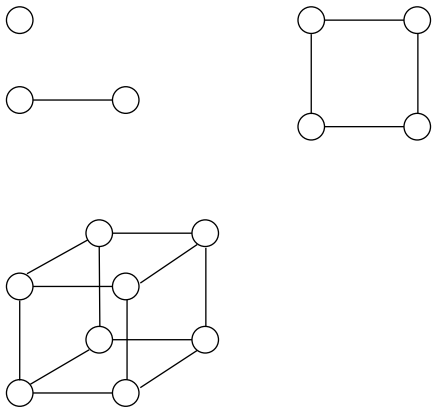
In a distributed memory system, each processor has its own private memory. A simple distributed memory system can be constructed by a number of workstations and a local network.

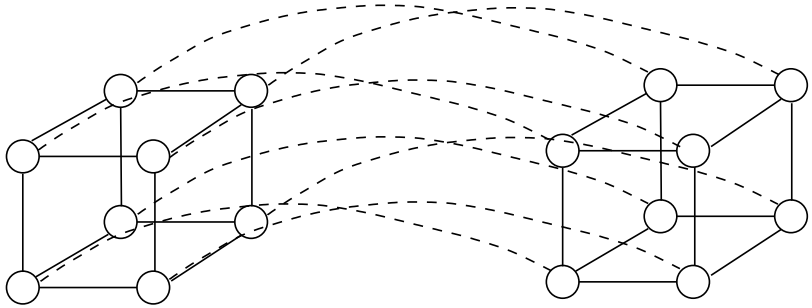
Some examples:



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

Hypercubes of dimensions 0, 1, 2 and 3.





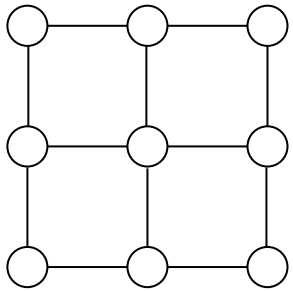
A 4-dimensional hypercube. Generally, a hypercube of dimension d+1 is constructed by connecting corresponding processors in two hypercubes of dimension d.

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

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

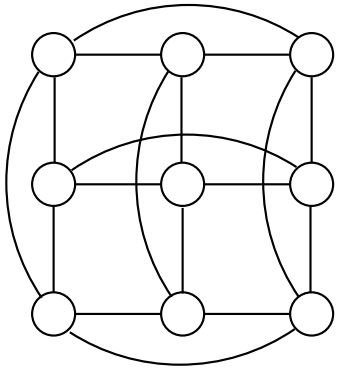
If the available connectivity (for a specific machine) is sufficient depends on the problem and the data layout.

This is a mesh:



We can have meshes of higher dimension.

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



A Note on Cluster Computing

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

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

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

■wæs on burgum
Beowulf Scyldinga,
leof leodcyning, longe þrage
folcum gefræge (fæder ellor hwearf,
aldor of earde), 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.

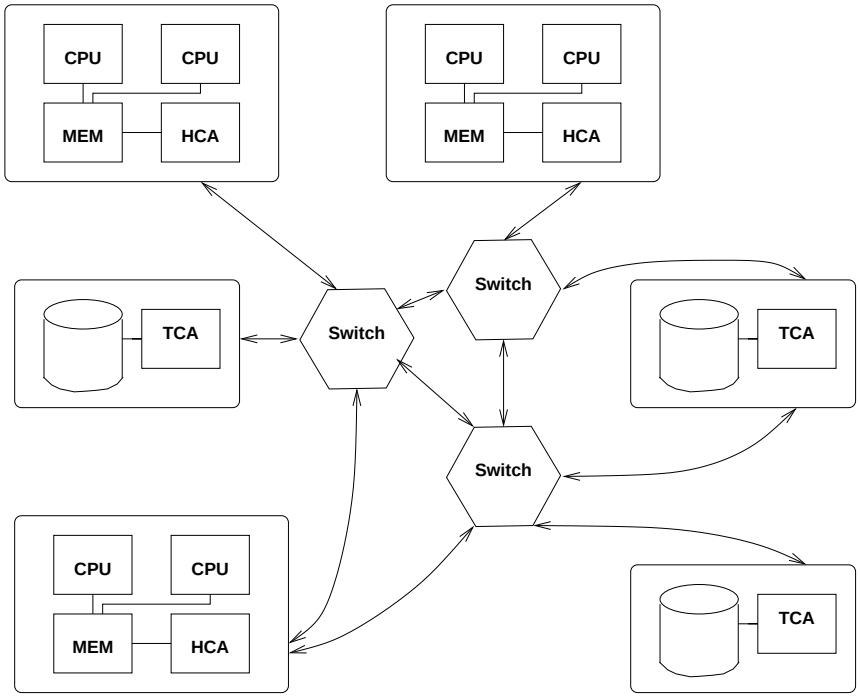
A word on Infiniband. First a quote from
`http://www.infinibandta.org/`
"InfiniBand is a high performance, switched fabric interconnect standard for servers. ... Founded in 1999, the InfiniBand Trade Association (IBTA) is comprised of leading enterprise IT vendors including Agilent, Dell, Hewlett-Packard, 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
```

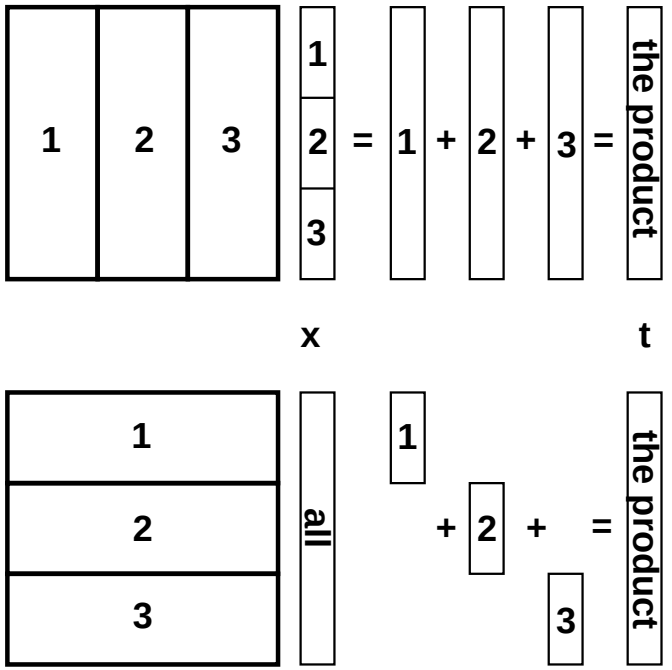
Note that $x^{(k)}$ does not "converge" in the ordinary sense.
We may have problems with over/underflow.

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:

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

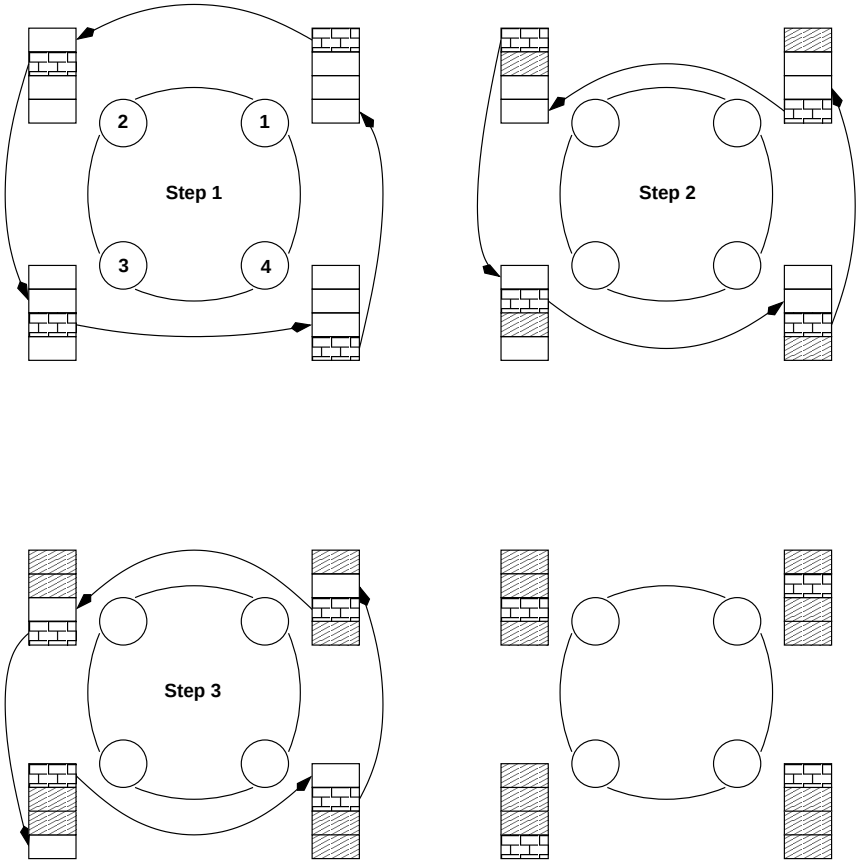
In order to perform the next iteration processor one needs $t_2, \ldots, t_{\#p}$, processor two needs $t_1, t_3, \ldots, t_{\#p}$ etc.
The processors must communicate, in other words.

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.

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



Some important terminology:

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

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

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

It is often the case that (roughly):

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

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

The *speedup* is defined as the ratio:

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

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

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

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

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

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

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

regardless of the number of processors.

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

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

So

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

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

Exercise: partition A by columns instead.

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

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

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

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

If

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

we may have to rebalance; dynamic load balancing.

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

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

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

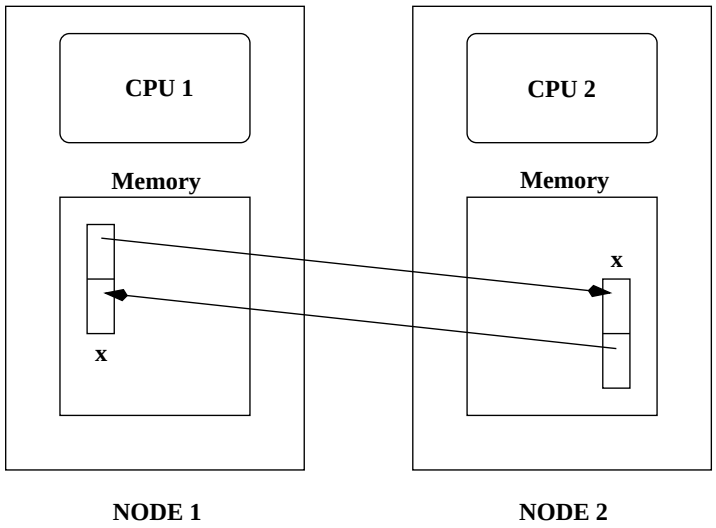
If we store U and D by rows it is easy to compute $Ux + Dx$ using our row-oriented algorithm. To compute $U^T x$ requires a column-oriented approach (if U is partitioned by rows, U^T will be partitioned by columns, and a column-oriented algorithm seems reasonable). So the program is a combination of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

A few words about communication

In our program we had the loop:

for $j = 1$ to $\#p - 1$
  send $x_{segment}$ to the next processor
  compute *segment*
  receive $x_{segment}$ 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.

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 -std=c99 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 *restrict arg) // the slave
{
  int      i, start, end, i_am, len;
  double   mysum;

  // typecasts, need both, see www for details
  i_am  = (int) (long) arg;
  len   = VEC_LEN / N_THREADS; // assume N_THREADS
  start = i_am  * len;         // divides VEC_LEN
  end   = start + len;

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

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

  // terminate the thread, NULL is the null-pointer
  pthread_exit(NULL); // not really needed
  return NULL;        // to silence splint
}
int main()
{
  pthread_t    thread_id[N_THREADS];
  int          i, ret;
```

```
  printf("sizeof(void *restrict) = %d\n",
         sizeof(void *restrict));  // to be sure
  printf("sizeof(long)           = %d\n", sizeof(long));

  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 *) (long) 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
sizeof(void *restrict) = 8
sizeof(long)           = 8
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.

If you use **top** and press **H** you well see the individual threads as well.

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, we are using MPICH2 from 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
% mpiexec -n 2 ./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_WORLD` is a communicator, a group of processes. The program can find out the number of processes by calling **`MPI_Comm_size`** (note that **`&`** is necessary since we require a return value).

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

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

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

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

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

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

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

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

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

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

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

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

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

If this value is less than the length of the message, the MPI-system prints an error message telling us that the message has been truncated.

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

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

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

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

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

We can do the following:

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

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

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

Here comes the simple program in Fortran.

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


  call MPI_Init(err)  ! Start up MPI

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

  tag    = 1
  length = 1   ! Length of message

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

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

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

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



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

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

  call MPI_Finalize(err)  ! Shut down MPI

end program simple
```

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

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

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

```
mpif90 simple.f90
mpiexec -n 2 ./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 MPICH2) 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
% mpiexec -n 2 ./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;
}
```

```
% mpiexec -n 4 ./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**).

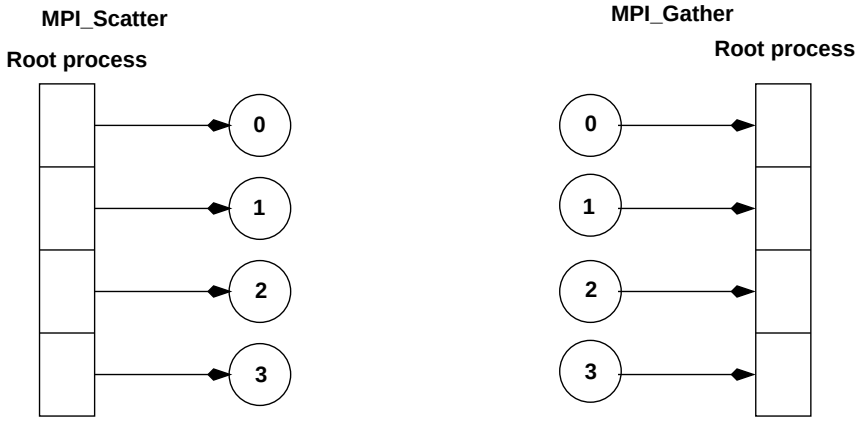
The **MPI_Allreduce** may be performed in an efficient way. Suppose we have eight processes, **0**, ..., **7**. **|** denotes a split.

```
           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.

Now for some routines which are practical when we would like to divide the processes into subsets.

So far we have only used one communicator containing all the processes. Sometimes it is useful to create subsets of processes, groups, e.g. for performing a brodacast within this subset. A *group* is a ordered collection of processes with ranks from zero to the number of processes minus one.

Assume we have started nine processes and we want to create a communicator, **My_Comm**, corresponding to ranks 0, 4 and 8. Then *all* processes (and not just the ones forming the new communicator) should do something like this (once): In Fortran90:

```
  integer :: new_group, world_group, err, My_Comm
  integer, dimension(3) :: ranks

  call MPI_Init(err)
! more code ...

  ranks = (/ 0, 4, 8 /)  ! the subset
! fetch the group, world_group, corresponding
! to MPI_COMM_WORLD
  call MPI_Comm_group(MPI_COMM_WORLD, world_group, err)

! create the new group from the (old) ranks in
! the old group, world_group
! size(ranks) = 3, is the number of ranks in subset
  call MPI_Group_incl(world_group, size(ranks), &
                      ranks, new_group, err)

! create the new communicator, My_Comm, from new_group
  call MPI_Comm_create(MPI_COMM_WORLD, new_group, &
                       My_Comm, err)
```

Groups and communicators are *opaque objects*, we do not know the internal representation in the MPI-system. Think of **world_group** (an integer) as a handle to the information.
One can create several communicators storing them as elements in an integer array, if convenient.


```
  MPI_Group new_group, world_group;
  MPI_Comm My_Comm;
  int ranks[] = {0, 4, 8};

  MPI_Init(&argc, &argv);
// more code ...

  MPI_Comm_group(MPI_COMM_WORLD, &world_group);
  MPI_Group_incl(world_group, 3, ranks, &new_group);
  MPI_Comm_create(MPI_COMM_WORLD, new_group, &My_Comm);
```

Important: note that the old ranks 0, 4, 8 in **MPI_COMM_WORLD** correspond to the new ranks 0, 1, 2 in **My_Comm**. Note also that only processes belonging to **My_Comm** may take part in the communication if the communicator is **My_Comm**. If, for example, the process with rank 7 in **MPI_COMM_WORLD** participates, the MPI-system will give an error message and halt the program.

Suppose we would like to create several groups (row-groups for a rectangular mesh, for example), there is a more convenient routine, **MPI_Comm_split** Here are a few examples, first a simple one, where we create three new communicators (with the same name), based on three subsets of ranks.

```
program Comm_split_1
  implicit none
  include    "mpif.h"
  integer :: err, My_comm, rank, my_new_rank, &
             group, gr_size

  call MPI_Init(err)
  call MPI_Comm_rank(MPI_COMM_WORLD, rank, err)

  if ( any(rank == (/ 0, 5 /)) ) &
    call MPI_Comm_split(MPI_COMM_WORLD, 1, 0, &
                        My_comm, err)
```

Ranks with same color (number **1** in this call) are grouped together. **0** is the key, and it is used to order new ranks in case we have several calls with the same colour.

```
  if ( any(rank == (/ 1, 3, 7 /)) ) &
    call MPI_Comm_split(MPI_COMM_WORLD, 2, 0, &
                        My_comm, err)

  if ( any(rank == (/ 2, 4, 6, 8 /)) ) &
    call MPI_Comm_split(MPI_COMM_WORLD, 3, 0, &
                        My_comm, err)

! Not necessary, just for pedagogical reasons
  call MPI_Comm_group ( My_comm, group, err ) ! group?
  call MPI_Group_size ( group, gr_size, err ) ! size?
  call MPI_Comm_rank  ( My_comm, my_new_rank, err )
  print*, 'rank, new_rank, size ', &
           rank, my_new_rank, gr_size

  call MPI_Finalize(err)
end
```

```
% mpif90 Comm_split1.f90
% mpiexec -n 9 ./a.out | sort -n     NOTE: sort
 rank, new_rank, size       0      0      2
 rank, new_rank, size       1      0      3
 rank, new_rank, size       2      0      4
 rank, new_rank, size       3      1      3
 rank, new_rank, size       4      1      4
 rank, new_rank, size       5      1      2
 rank, new_rank, size       6      2      4
 rank, new_rank, size       7      2      3
 rank, new_rank, size       8      3      4
```

The third argument (the key) to **MPI_Comm_split** is unused in the above example but can be used to order the ranks in case we have several calls with the same color.


```
  if ( any(rank == (/ 2, 4 /)) ) &
    call MPI_Comm_split(MPI_COMM_WORLD, 3, 1, &
                        My_comm, err) !    ^ NOTE

  if ( any(rank == (/ 6, 8 /)) ) &
    call MPI_Comm_split(MPI_COMM_WORLD, 3, 0, &
                        My_comm, err) !    ^ NOTE
```

gives the order 6, 8, 2, 4 (the new ranks are 0, 1, 2, 3).

Let us use **MPI_Comm_split** to create communicators suitable for a rectangular grid with four rows and five columns. We need a communicator for each row. Ranks range from 0 to $4 \cdot 5 - 1 = 19$. **floor(rank / 5)** gives the values

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

producing four subsets (the rows). Here is the C-code:

```
...
  MPI_Comm My_comm;
  int my_rank, my_row, gr_size, my_new_rank;
  MPI_Group group;

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

  my_row = my_rank / 5;         // integer division
  MPI_Comm_split(MPI_COMM_WORLD, my_row, 0, &My_comm);

  // For pedagogigal reasons
  MPI_Comm_group ( My_comm, &group );
  MPI_Group_size ( group, &gr_size );
  MPI_Comm_rank  ( My_comm, &my_new_rank );
  printf("rank, new_rank, size  %02d, %3d, %3d\n",
          my_rank, my_new_rank, gr_size);
...
```

and here is the run

```
% mpicc Comm_split3.c
% mpiexec -n 20 ./a.out | sort -n
rank, new_rank, size  00,   0,   5
rank, new_rank, size  01,   1,   5
rank, new_rank, size  02,   2,   5
rank, new_rank, size  03,   3,   5
rank, new_rank, size  04,   4,   5
rank, new_rank, size  05,   0,   5
rank, new_rank, size  06,   1,   5
  etc.
rank, new_rank, size  18,   3,   5
rank, new_rank, size  19,   4,   5
```

Now for an example with two rows and columns.

```
...
  MPI_Comm My_comm;
  int my_rank, my_row, my_new_rank, data[5];

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

  for(int k = 0; k < 5; k++)
    data[k] = 10 * my_rank + k;

  my_row = my_rank / 2;         // integer division
  MPI_Comm_split(MPI_COMM_WORLD, my_row, 0, &My_comm);

  // 0 is the root in My_comm
  MPI_Bcast(data, 5, MPI_INT, 0, My_comm);

  // Not necessary
  MPI_Comm_rank(My_comm, &my_new_rank);
  printf("%d, %d: %3d %3d %3d %3d %3d\n", my_rank,
         my_new_rank, data[0], data[1], data[2],
         data[3], data[4]);
...
```



```
% mpicc -std=c99 Comm_split4.c
% mpiexec -n 4 ./a.out
0, 0:   0   1   2   3   4   my_rank = 0 bcasting
2, 0:  20  21  22  23  24   my_rank = 2 bcasting
1, 1:   0   1   2   3   4   my_rank = 0 bcasting
3, 1:  20  21  22  23  24   my_rank = 2 bcasting
```

Let us continue with something related, topologies, being able to index a process using row- and column indices.

```
...
  MPI_Comm Grid_comm;
  int my_rank, my_new_rank, coords[2];
  int grid_sz[] = {3, 3};  // 3 x 3-grid
  int wrap[]    = {0, 1};  // wrap around?
  int reorder   = 0;       // reorder for efficiency?

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  // Create Grid_comm
  MPI_Cart_create(MPI_COMM_WORLD, 2, grid_sz,
                  wrap, reorder, &Grid_comm);

  MPI_Comm_rank(Grid_comm, &my_new_rank);
  // my_new_rank -> coords
  MPI_Cart_coords(Grid_comm, my_new_rank, 2, coords);
  printf(
    "rank, new_rank, coords:  %3d, %3d,  (%1d, %1d)\n",
     my_rank, my_new_rank, coords[0], coords[1]);

  MPI_Barrier(Grid_comm);  // wait for printf
  if ( my_rank == 0 ) {
    coords[0] = 1; coords[1] = 2;
    // coords -> my_new_rank
    MPI_Cart_rank(Grid_comm, coords, &my_new_rank);
    printf("new_rank, coords:  %3d, (%2d, %2d)\n",
           my_new_rank, coords[0], coords[1]);
```

```
    // coords[1] = 3; is OK since wrap[1] = 1;
    coords[1] = 3;
    MPI_Cart_rank(Grid_comm, coords, &my_new_rank);
    printf("new_rank, coords:  %3d, (%2d, %2d)\n",
           my_new_rank, coords[0], coords[1]);

    coords[1] = -2;
    MPI_Cart_rank(Grid_comm, coords, &my_new_rank);
    printf("new_rank, coords:  %3d, (%2d, %2d)\n",
           my_new_rank, coords[0], coords[1]);
  }
...
```

```
% mpicc -std=c99 Comm_grid.c
% mpiexec -n 9 ./a.out
rank, new_rank, coords:    0,   0,  (0, 0)
rank, new_rank, coords:    1,   1,  (0, 1)
rank, new_rank, coords:    2,   2,  (0, 2)
rank, new_rank, coords:    4,   4,  (1, 1)
rank, new_rank, coords:    8,   8,  (2, 2)
rank, new_rank, coords:    6,   6,  (2, 0)
rank, new_rank, coords:    7,   7,  (2, 1)
rank, new_rank, coords:    5,   5,  (1, 2)
rank, new_rank, coords:    3,   3,  (1, 0)
new_rank, coords:    5, ( 1,  2)
new_rank, coords:    3, ( 1,  3)
new_rank, coords:    4, ( 1, -2)
```

The **reorder**-parameter is ignored in MPICH2.

We would now like to partition this grid into subgrids corresponding to rows (and columns).
The function **MPI_Cart_sub** does that for us.


```
...
  MPI_Comm Grid_comm, Row_comm;
  int my_rank, grid_rank, row_rank, coords[2], coordsr;
  int grid_sz[]    = {3, 3};   // 3 x 3-grid
  int wrap[]       = {0, 1};   // wrap around?
  int reorder      = 0;        // OK to reorder
  int free_coord[] = {0, 1};   // lock rows

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Cart_create(MPI_COMM_WORLD, 2, grid_sz,
                  wrap, reorder, &Grid_comm);

// Create row communicator
  MPI_Cart_sub(Grid_comm, free_coord, &Row_comm);
// Not necessary
  MPI_Comm_rank(Grid_comm, &grid_rank);
  MPI_Comm_rank(Row_comm,  &row_rank);
  MPI_Cart_coords(Grid_comm, grid_rank, 2, coords);
  MPI_Cart_coords(Row_comm, row_rank, 1, &coordsr);
  printf(
    "rank, grid, row, coords:  %3d, %3d, %d,  (%1d, %1d)
     my_rank, grid_rank, row_rank, coords[0], coords[1],
          coordsr);
...
```

```
% mpicc Comm_grid_sub.c
% mpiexec -n 9 ./a.out | sort -n
rank, grid, row, coords:    0,   0, 0,  (0, 0), 0
rank, grid, row, coords:    1,   1, 1,  (0, 1), 1
rank, grid, row, coords:    2,   2, 2,  (0, 2), 2
rank, grid, row, coords:    3,   3, 0,  (1, 0), 0
rank, grid, row, coords:    4,   4, 1,  (1, 1), 1
rank, grid, row, coords:    5,   5, 2,  (1, 2), 2
rank, grid, row, coords:    6,   6, 0,  (2, 0), 0
rank, grid, row, coords:    7,   7, 1,  (2, 1), 1
rank, grid, row, coords:    8,   8, 2,  (2, 2), 2
```

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 $\mathbf{8 \cdot 32}$.

```
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).
- 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 (usually ACML or MKL).
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
```

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.

According to MathWorks:
`maxNumCompThreads` will be removed in a future version. You can set the `-singleCompThread` option when starting MATLAB to limit MATLAB to a single computational thread. By default, MATLAB makes use of the multithreading capabilities of the computer on which it is running.

Matlab R2011b (and later) has support for Nvidia's CUDA (Compute Unified Device Architecture), executing code on the GPU. We cannot test this on the math-machines:

```
>> g = gpuArray(rand(10, 1, 'single'));
Error using gpuArray (line 28)
No device supporting CUDA was found.
```

In the Matlab-help it is stated:
The following are required for using the GPU with MATLAB:
NVIDIA CUDA-enabled device with compute capability of 1.3 or greater. The latest NVIDIA CUDA device driver.

The GeForce 9500 GT (in the math-machines) has compute capability 1.1 (use the Matlab-command `gpuDevice` to find out, for example), and the latest driver is not installed.
A short section on using CUDA from C will come at the end of the course.

Please see the separate Beamer-file, OpenMP.