Writing efficient programs

Thomas Ericsson

Computational Mathematics
Chalmers University of Technology
and the University of Gothenburg

PDC Summer School 2011

2: Contents

- How does one get good performance from a computer system?
- Focus on systems with one CPU (with one core) and floating point performance.
- To get maximum performance from a parallel code it is important to tune the code running on each CPU.
- General advice and not specific systems.
- Fortran, some C (hardly any C++) and some Matlab. Some Java in the compendium.

3: Your situation

- A large and old code which has to be optimized. Even a slight speedup would be of use, since the code may be run on a daily basis.
- A new project, where language and data structures have to be chosen.
  C/C++ usually slower than Fortran for floating point.
  Java? Can be slow and use large amounts of memory.
  See the article (compendium) for an example.
  Should it be parallel?
  Test a simplified version of the computational kernel.
  Fortran for floating point, C/C++ for the rest.
- Things that are done once. Let the computer work.
  Unix-tools, Matlab, Maple, Mathematica ...

4: The optimization process

Basic: Use an efficient algorithm.
Simple things:

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

5: The Intel compiler

- Compiler options, flags, of the Intel Fortran90-compiler, more than 300.
- Names not standardized.
- Some of the flags are passed on to the preprocessor (locations and names of header files) and to the linker (locations and names of libraries).
- There is a user and reference guide, PDF ($>$ 3800 pages, for Fortran, C++-manual 1894 pages).
- Here a few av the more than 1000 lines produced by **`icc -help`** and **`ifort -help`**
- The Cray compiler has similar options (often with the same names). See **`http://docs.cray.com/`**.

6: The Intel compiler cont.

• Optimization
`...`
`-O2` optimize for maximum speed (DEFAULT)
`-O3` optimize for maximum speed and enable more aggressive optimizations that may not improve performance on some programs may be slower, TE's comment
`-O` same as **`-O2`**
`...`
`-O0` disable optimizations
`-fast` enable **`-xHOST -O3 -ipo -no-prec-div -static`**
`-fno-alias` assume no aliasing in program
`...`
• Code Generation
`-x<code1>` generate specialized code to run exclusively on processors indicated by **`<code>`** as described below

7: The Intel compiler cont.

• Interprocedural Optimization (IPO)
`-[no-]ip` enable(DEFAULT)/disable single-file IP optimization within files
`-ipo[n]` enable multi-file IP optimization between files
...
• Advanced Optimizations
...
`-[no-]vec` enables(DEFAULT)/disables vectorization
...
Here is an incomplete list of the remaining categories:
• Profile Guided Optimization (PGO)
• Optimization Reports
• OpenMP* and Parallel Processing
• Floating Point
• Inlining

8: The Intel compiler cont.

• Output, Debug, PCH (pre compiled header files)
`-c` compile to object (.o) only, do not link
`-S` compile to assembly (.s) only, do not link
`-o <file>` name output file
`-g` produce symbolic debug information in object file (implies **`-O0`** when another optimization option is not explicitly set)
• Preprocessor
• Compiler Diagnostics
• Linking/Linker

9: If you are willing to work more...

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

Find bottlenecks in the code (profilers). Attack the subprograms taking most of the time. Find and tune the important loops.
Tuning loops has several disadvantages:

- The code becomes less readable and you may introduce bugs.
- Detailed knowledge about the system, such as cache configuration, may be necessary.
- What is optimal for one system need not be optimal for another; faster on one machine may actually be slower on another. This leads to problems with portability.

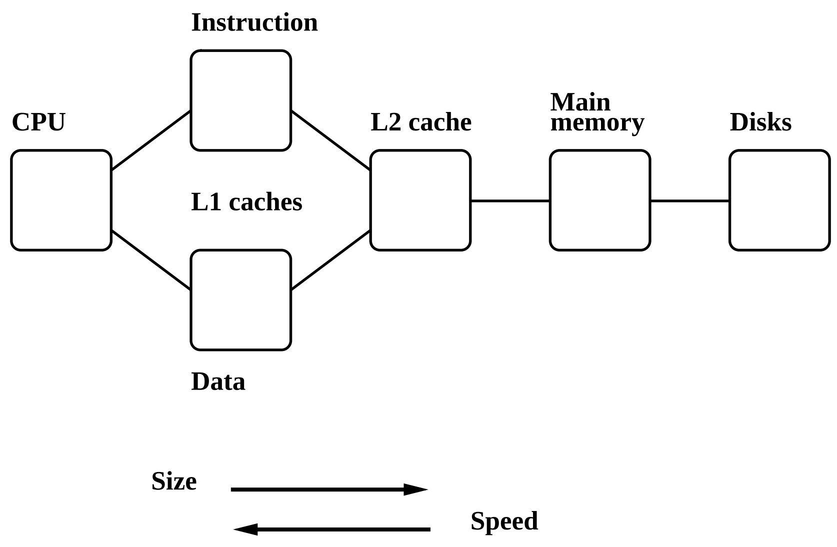
10: If you are willing to work more..., cont.

- Code tuning is not a very deterministic business.
  The combination of tuning and the optimization done by the compiler may give an unexpected result.
- The computing environment is not static; compilers become better and there will be faster hardware of a different construction.
  The new system may require different (or no) tuning.

The goal of the tuning effort is to keep the FPU(s) busy.
Accomplished by efficient use of the

- memory hierarchy
- parallel capabilities

11: If you are willing to work more..., cont.



- Superscalar: start several instructions per cycle.
- Pipelining: work on an instruction in parallel.
- Vectorization: parallel computation on short arrays.

12: If you are willing to work more..., cont.

- Locality of reference, data reuse
- Avoid data dependencies and other constructions that give pipeline stalls

13: What can you hope for?

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

Not very deterministic, in other words.

- Do not rewrite all the loops in your code.
- Save a copy of the original code. If you make large changes to the code, use som kind of version control system.
- Compare computational results before and after tuning.

14: Choice of language

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

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

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

15: Choice of language, contd.

n, was chosen such that the three vectors would fit in the L1-cache, all at the same time.
On some platforms the Fortran routine can be twice as fast.

From the Fortran 90 standard (section 12.5.2.9):

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

Not so in C. Two pointer-variables with different names may refer to the same array, this is called aliasing.

16: Choice of language, cont.

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

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

Can use the pipelining in functional units for addition and multiplication.
The assembly code is often unrolled this way as well. The corresponding C-code may look like:

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

17: Choice of language, cont.

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

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

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

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

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

18: Choice of language, cont.

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

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

This loop is slower than the first one (slower in C as well).
In C, aliased pointers and arrays are allowed which means that it may be harder for a C-compiler to produce efficient code.
The C99 **`restrict`** type qualifier can be used to inform the compiler that aliasing does not occur.

```
void add(double * restrict a,  etc.)
```

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

19: Choice of language, cont.

An alternative is to use compiler flags, **`-fno-alias`**, **`-xrestrict`** etc. supported by <u>some</u> compilers.
If you "lie" (or use a Fortran routine with aliasing) you probably get the wrong answer!

According to an Intel article, their C/C++-compiler can generate dynamic data dependence testing (checking addresses using if-statements) to decrease the problem with aliasing.

To see the effects of aliasing one may need more complicated examples than **`add`**. I have kept it because it is easy to understand. On the next page is a slightly more complicated example (Horner's method for polynomials), but still only a few lines of code, i.e. far from a real code.

20: Choice of language, cont.

Compute $\mathbf{p_j = c_0 + c_1\,x_j + c_2\,x_j^2 + c_3\,x_j^3 + c_4\,x_j^4}$ using Horner's method:

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

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

Using **`n = 1000`** and calling the routine $\mathbf{10^6}$ times the speed advantage of Fortran was a factor of 1 up to 4.
If **`-fno-alias`** is used, C ≈ Fortran.

21: Choice of language, cont.

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

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

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

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

It is instructive to compare the assembly ouput of the two Horner routines. **gcc -O3 -S horner.c** gives assembly on **horner.s**.

22: Choice of language, cont.

```
Original routine (just the loop)      Modified code, st = stack
.L4:                                .L11:
  movl    12(%ebp), %edx                fldl    (%ecx,%eax,8)
  fldl    (%edx,%eax,8)                 fld     %st(0)
  movl    -16(%ebp), %edx               fmul    %st(2), %st
  fld     %st(0)                        fadd    %st(3), %st
  fmull   (%edx)                        fmul    %st(1), %st
  movl    8(%ebp), %edx                 fadd    %st(4), %st
  faddl   (%ecx)                        fmul    %st(1), %st
  fmul    %st(1), %st                   fadd    %st(5), %st
  faddl   (%esi)                        fmulp   %st, %st(1)
  fmul    %st(1), %st                   fadd    %st(5), %st
  faddl   (%edi)                        fstpl   (%ebx,%eax,8)
  fmulp   %st, %st(1)                   addl    $1, %eax
  faddl   (%ebx)                        cmpl    %edx, %eax
  fstpl   (%edx,%eax,8)                 jne     .L11
  addl    $1, %eax
  cmpl    20(%ebp), %eax
  jne     .L4
```

I used **gcc** instead of **icc** which vectorizes the code and makes it very hard to read.

23: Choice of language, cont.

Now to Horner with complex numbers using Fortran (complex is built-in) and C++ (using “C-arrays” of **`complex<double>`**). Times, on three systems using **`n = 1000`** and calling the routine 10^5 times. Compiling using **`-O2`** or **`-O3`**, whatever is best.
`i` = Intel, **`pg`** = Portland Group, **`g`** = GNU.

ifort	**icpc**	**pgf90**	**pgCC**	**g95**	**g++**
0.6	7.4	na	na	5.0 (†)	5.4
1.9	4.5	2.3	14.1	3.1	7.0
1.0	1.9	1.2	3.9	1.2 (‡)	2.6

(†) **`g77`** instead of **`g95`**. (‡) **`gfortran`** instead of **`g95`**.
Important to test different systems, compilers and compile-options.
The behaviour in the above codes changes when **`n`** becomes very large. CPU-bound (the CPU limits the performance) versus memory bound (the memory system limits the performance).

24: Tuning Matlab programs

The timings below are for Matlab version R2010b on a 2.66GHz Intel Xeon. Matlab 6.5 (and newer) has a JIT-accelerator (Just In Time). Older versions can be much slower.

- Use the built-in compiled routines. The Matlab-language is interpreted (unless JIT can be applied).
- Work on the matrix/vector-level, not on element-level. Different programming style.
- Take care when using the dynamic memory allocation. Preallocate.

Some examples, **`n = 2500`**.

```
for i = 1:n
  for j = 1:n
    for k = 1:n
    end
  end
end    Takes 36 s
```

25: Tuning Matlab programs, cont.

Say you want to save a large number of vectors for later analysis.

```
clear A
x = rand(n, 1);
for k = 1:n
  A(:, k) = x;  % could have different arrays
end
Takes 36 s

A = zeros(n);   % preallocate
for k = 1:n
  A(:, k) = x;
end
Takes 0.01 s
```

26: Tuning Matlab programs, cont.

W is a 8000×15-matrix and **x** is a column vector having 8000 elements.

```
y = W * W' * x;              y = W * (W' * x);

Takes 0.9 s                    0.0002 s
```

Note that it may be impossible just to form `W * W'` even though `y = W * (W' * x);` gives no problem.

27: Tuning Matlab programs, cont.

Do not use more general functions than necessary (inline):

```
v = rand(3, 1);  w = rand(3, 1);

for k = 1:100000
  d = dot(v, w);          % inner product
  v(1) = v(1) + 1e-50;    % added to the loops
end                       % below as well
Takes 2.5 s

for k = 1:100000
  c = v' * w;
end
Takes 0.06 s
```

28: Tuning Matlab programs, cont.

```
for k = 1:100000
  c = cross(v, w);
end
Takes 6.7 s

for k = 1:100000
  c = [v(2)*w(3)-v(3)*w(2); v(3)*w(1)-v(1)*w(3); ...
       v(1)*w(2)-v(2)*w(1)];
end
Takes 0.03 s
```

29: Basic arithmetic and elementary functions

Many modern CPUs have vector units which can work in parallel on the elements of short arrays. Intel has the SSE (Streaming SIMD Extensions).
Arrays consist of two double precision numbers or four single precision numbers.
The arithmetic may have different roundoff properties compared to the usual FPU (x87 in an Intel CPU).

- Common that the (x87) FPU can perform `+` and `*` in parallel.
- `a+b*c` can often be performed with one round-off, multiply-add MADD or FMA.
- `+` and `*` usually pipelined, so one sum and a product per clock cycle in the best of cases (not two sums or two products). Often one sum every clock cycle and one product every other.
- `/` not usually pipelined and may require 15-40 clock cycles.
- May have several computational cores as well as vector units.

30: Floating point formats

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

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

The efficiency of FPUs differ (this on a 2.66 GHz Intel Xeon).

```
>> A = rand(1000); B = A;
>> tic; C = A * B; toc
Elapsed time is 0.185060 seconds.
>> A = 1e-320 * A;
>> tic; C = A * B; toc
Elapsed time is 63.908510 seconds.
```

31: Floating point formats, contd.

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

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

Integer multiplication and multiply-add can be slower than their floating point equivalents.

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

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

Change types to **`real`** and then to **`double precision`** A few tests:

integer	single	double
0.5	0.18	0.36
0.7	1.0	1.0
1.0	1.6	1.6
0.92	0.22	0.44

32: Elementary functions

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

- argument reduction
- approximation
- back transformation

Can take a lot of time (much more than `+, *`).

```
>> v = 0.1 * ones(10000, 1);
>> tic; for k = 1:1000, s = sin(v); end; toc
Elapsed time is 0.089619
% time increases after pi/4
>> v = 1e5 * ones(10000, 1); tic etc.
Elapsed time is 0.352703 seconds.

>> v = 1e10 * ones(10000, 1); tic etc.
Elapsed time is 1.711913
```

33: Elementary functions, contd.

```
  double precision :: x = 2.5d1

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

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

34: Elementary functions, contd.

Some compilers are more clever than others, which is shown on the next page. Unless **x** is an integer, $\mathbf{v}^{\mathbf{x}}$ is computed like this:

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

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

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

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

Compiler **-O3**	code above	my opt. code
Intel	1.2	1.2
gfortran	8.1	1.6

35: Elementary functions, contd.

Looking at the assembly output from Intel's compiler:

```
        fsqrt
        fmulp     %st, %st(1)
```

`gfortran` calls `pow` (uses exp and log).
In my optimized routine I have written the loop this way:

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

Interesting when dealing with $1/r^2$-forces.

$$F = c\frac{r/|r|}{|r|^2} = \frac{c\,r}{|r|^3} = \frac{c\,r}{\left(\sqrt{r_1^2 + r_2^2 + r_3^2}\right)^3} = \frac{c\,r}{(r_1^2 + r_2^2 + r_3^2)^{1.5}}$$

36: Elementary functions, contd.

Vector versions of elementary functions as well as slightly less accurate versions area available in AMD's ACML and Intel's MKL. Here an example using MKL's VML (Vector Mathematics Library).

```
  include mkl_vml.fi

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

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

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

37: Elementary functions, contd.

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

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

38: An SSE-example

We need an optimizing compiler that produces code using the special vector instructions (or we can program in assembly).

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

Called 100000 times. Here are some typical times on two systems:

single		double	
no vec	vec	no vec	vec
1.60	0.38	1.80	0.92
0.83	0.41	0.99	0.80

Some compilers vectorize automatically. Speedup may differ, also not all codes can be vectorized. You may get different results using vectorization (due to different round-off properties).

39: Eliminating constant expressions from loops

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

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

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

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

40: Virtual memory and paging

- Simulate larger memory using disk.
- Virtual memory is divided into pages, perhaps 4 or 8 kbyte.
- Moving pages between disk and physical memory is known as paging.
- Avoid excessive use. Disks are slow.
- Paging can be diagnosed by using your ear (if you have a local swap disk), or using the **`sar`**-command,
  `sar -B interval count`, so e.g. **`sar -B 1 3600`** .
  `vmstat` works on some unix-systems as well and the **`time`**-command built into **`tcsh`** reports a short summary.

41: Input-output

We need to store 10^8 double precision numbers in a file.
A local disk was used for the tests. Intel's Fortran compiler on an Intel Core Duo. Roughly the same times in C.

Test	Statement	time (s)	Gbyte
1	`write(10, '(1pe23.16)') x(k)`	415.1	2.24
2	`write(10) x(k)`	274.4	1.49
3	`write(10) (vec(j), j = 1, 10000)`	1.1	0.74

In the third case we write $10^8/10^4$ records of 10^4 numbers each.

42: Input-output, contd.

File sizes:

$$1: \quad \underbrace{10^8}_{\text{\# of numbers}} \cdot \underbrace{(23+1)}_{\text{characters + newline}} \; / \; \underbrace{2^{30}}_{\text{Gbyte}} \approx 2.24$$

$$2: \quad \underbrace{10^8}_{\text{\# of numbers}} \cdot \underbrace{(8+4+4)}_{\text{number + delims}} \; / \; \underbrace{2^{30}}_{\text{Gbyte}} \approx 1.49$$

$$3: \quad \left[\underbrace{10^8}_{\text{\# of numbers}} \cdot \underbrace{8}_{\text{number}} + (10^8/10^4) \cdot \underbrace{(4+4)}_{\text{delims}} \right] \; / \; \underbrace{2^{30}}_{\text{Gbyte}} \approx 0.74$$

43: Portability of binary files?

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

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

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

44: Optimizing for locality, data re-use, loop fusion

Compute **min(v)** and **max(v)**, where **v** is a vector.

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

In **v_min = v(k)**, **v(k)** is stored in a register and not fetched again.

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

45: Optimizing for locality, data re-use, loop fusion, contd.

Merge loops data re-use, less loop overhead.

```
v_min = v(1)
v_max = v(1)
do k = 2, n
  if ( v(k) < v_min ) then     ! fetch
    v_min = v(k)
  elseif ( v(k) > v_max ) then ! re-use
    v_max = v(k)
  end if
end do

  if(v_min < vk) v_min = v(k)! may be faster
  if(v_max > vk) v_max = v(k)! on some systems

  v_min = min(v_min, v(k))! or like this
  v_max = max(v_max, v(k))
```

46: Optimizing for locality, loop-splitting

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

```
integer, parameter                :: n = 7000
double precision, dimension(n, n) :: A,B,C,D
...
sum_ab = 0.0
sum_cd = 0.0
do col = 1, n
  do row = 1, n  ! independent sums
    sum_ab = sum_ab + A(row, col)* B(col, row)
    sum_cd = sum_cd + C(row, col)* D(col, row)
  end do
end do
```

47: Optimizing for locality, loop-splitting, contd.

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

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

On an Intel Core Duo the first loop took 7.1 s and the second two 1.1 s (together). Speedup depends on **n**, hardware and compiler.

48: The importance of small strides

If no data re-use, try to have locality of reference.
Use small strides.
v(1), v(2), v(3),..., stride one
v(1), v(3), v(5),..., stride two

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

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

Some compilers can switch loop order (loop interchange).

49: The importance of small strides, contd.

```
First loop     Second loop
A(1, 1)         A(1, 1)      First column
A(2, 1)         A(2, 1)
A(3, 1)         A(3, 1)
  ...             ...
A(n, 1)         A(n, 1)
A(1, 2)         A(1, 2)     Second column
A(2, 2)         A(2, 2)
  ...             ...
A(n, 2)         A(n, 2)
         ...
A(1, n)         A(1, n)     n:th column
A(2, n)         A(2, n)
  ...             ...
A(n, n)         A(n, n)
```

In C the leftmost alternative will be the faster.

50: The importance of small strides, contd.

Performance on three systems. Compiling using **-O3** in the first test and using **-O3 -ipo** in the second.

	C	Fortran	C	Fortran	C	Fortran
By row	0.7 s	2.9 s	0.6 s	2.4 s	0.5 s	1.5 s
By column	4.6 s	0.3 s	2.4 s	0.6 s	1.6 s	0.5 s
By row **-ipo**	0.3 s	0.3 s	0.6 s	0.6 s	0.5 s	0.5 s
By column **-ipo**	2.9 s	0.3 s	0.6 s	0.6 s	1.5 s	0.5 s

-ipo, interprocedural optimization i.e. optimization between routines (even in different files) gives a change of loop order, at least for Fortran, in this case. Some Fortran compilers can do this just specifying **-O3** (if **s** is local or the return value of a function).

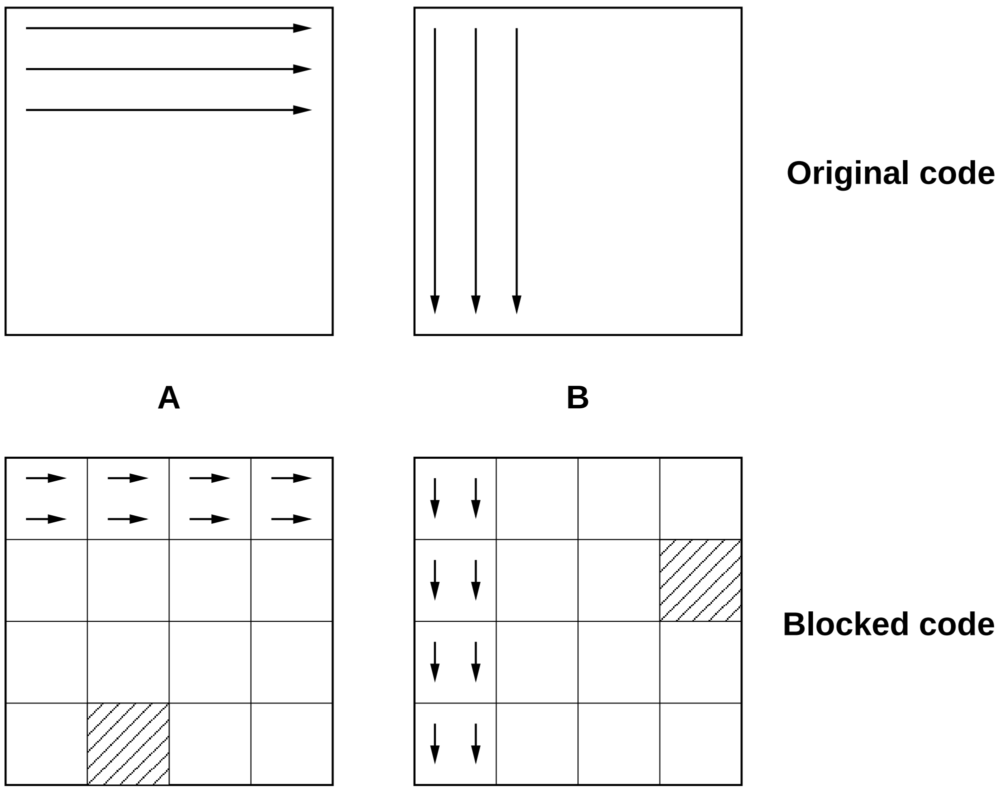
51: Blocking and large strides

Sometimes loop interchange is of no use.

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

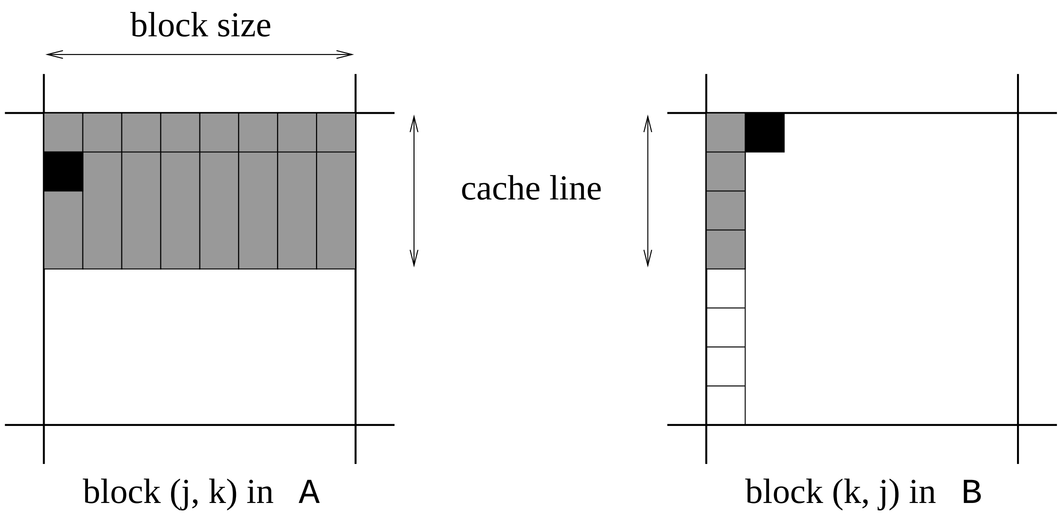
Bad locality for **A** good for **B**.
Blocking is good for data re-use, and when we have large strides.
Partition **A** and **B** in square sub-matrices each having the same order, the block size.
Treat pairs of blocks, one in **A** and one in **B** such that we can use the data which has been fetched to the L1 data cache.

52: Blocking and large strides, contd.



53: Blocking and large strides, contd.

Looking at two (shaded) blocks:



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

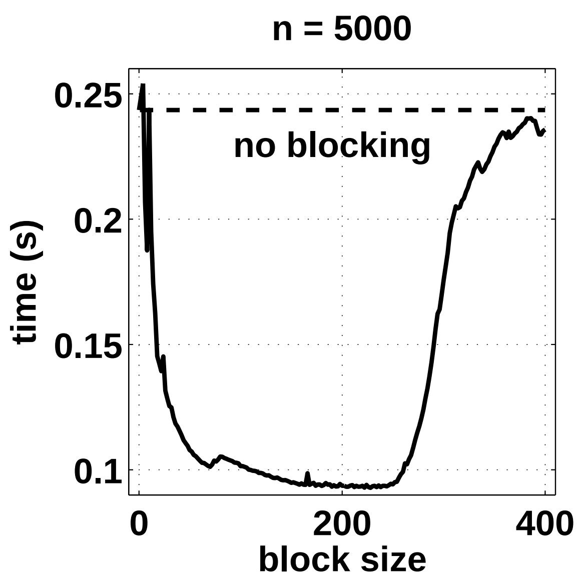
54: Blocking and large strides, contd.

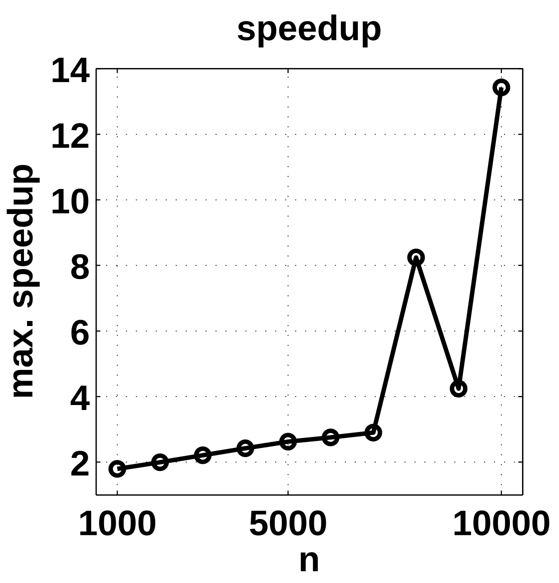
This code works even if **n** is not divisible by the block size).

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

do first_row = 1, n, block_size
  last_row = min(first_row + block_size - 1, n)
  do first_col = 1, n, block_size
    last_col = min(first_col + block_size - 1, n)

    ! sum one block
    do row = first_row, last_row
      do col = first_col, last_col
        s = s + A(row, col) * B(col, row)
      end do
    end do

  end do
end do
```

55: Blocking and large strides, contd.

Left plot, $n = 5000$, different block sizes using **`ifort -O3`** on an Intel Core Duo. Right plot, speedup for $n = 10^3, 2 \cdot 10^3, \ldots, 10^4$ with optimal block size.





56: Blocking and large strides, contd.

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

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

57: Two important libraries

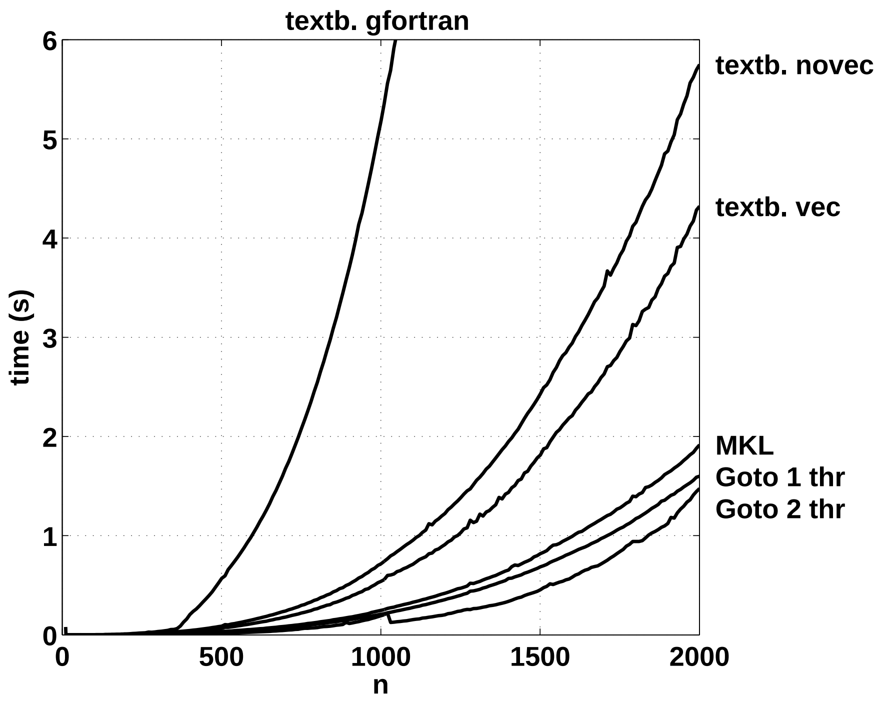
BLAS (the Basic Linear Algebra Subprograms) are the standard routines for simple matrix computations.
(**s** single, **d** double, **c** complex, **z** double complex).

Examples:
BLAS1: **`y := a*x + y`** one would use **`daxpy`**
BLAS2: **`dgemv`** can compute **`y := a*A*x + b*y`**
BLAS3: **`dgemm`** forms **`C := a*A*B + b*C`**

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

58: Two important libraries, contd.

Multiplication of $\mathbf{n} \times \mathbf{n}$-matrices, Intel Core Duo.

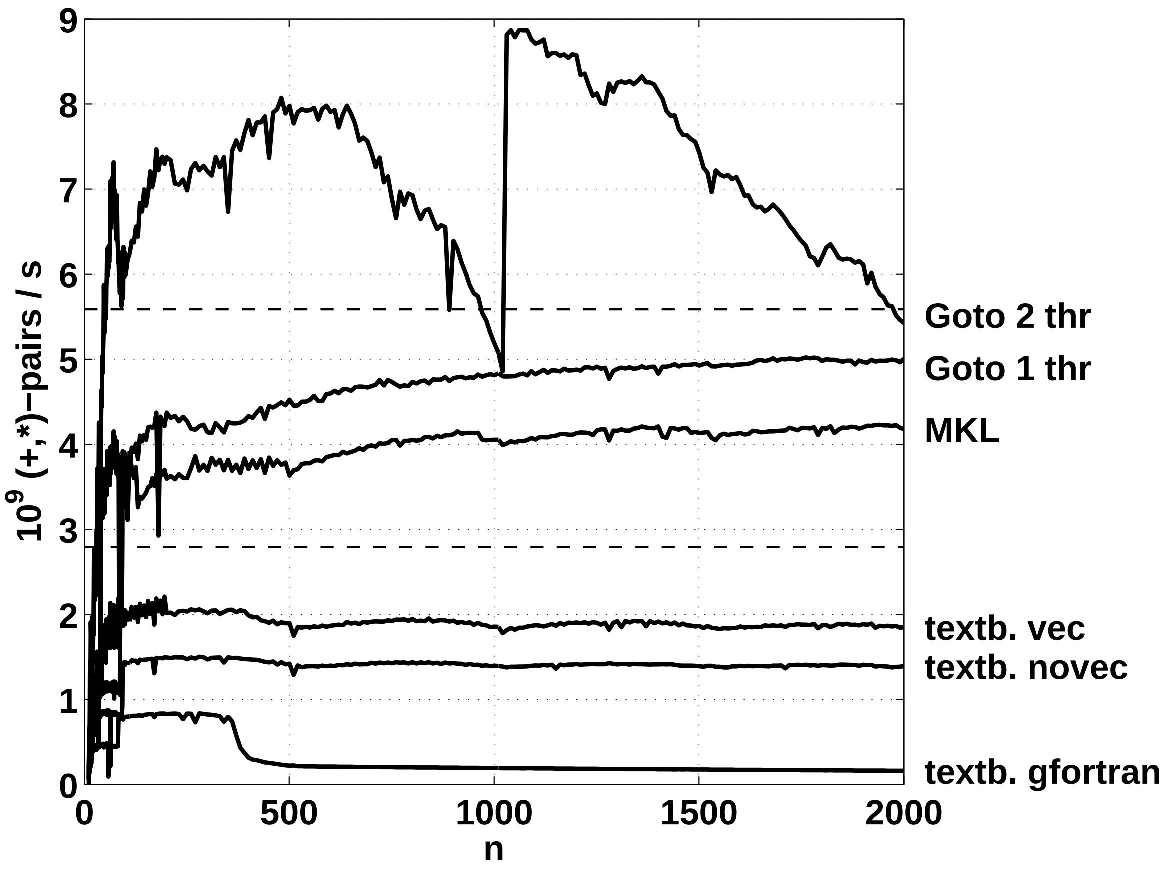


59: Two important libraries, contd.

Tested textbook “row times column” using **`gfortran`** and **`ifort`** with and without vectorization. MKL is Intel’s MKL-library. Goto is Goto-BLAS by Kazushige Goto.
The fast codes use blocking and other tricks. A goal of Goto-BLAS is to minimize the number of TLB-misses.
Goto-BLAS on two threads is roughly equal to MKL on two threads.

The next figure shows the number of (+, *)-pairs executed per second. The dashed lines show the clock frequency and twice the frequency.

60: Two important libraries, contd.



61: Two important libraries, contd.

LAPACK is the standard library for (dense):

- linear systems
- eigenvalue problems
- linear least squares problems

No support for large sparse problems, but there are routines for banded matrices of different kinds.
LAPACK is built on top of BLAS (BLAS3 where possible).
When using LAPACK, it is important to have optimized BLAS.

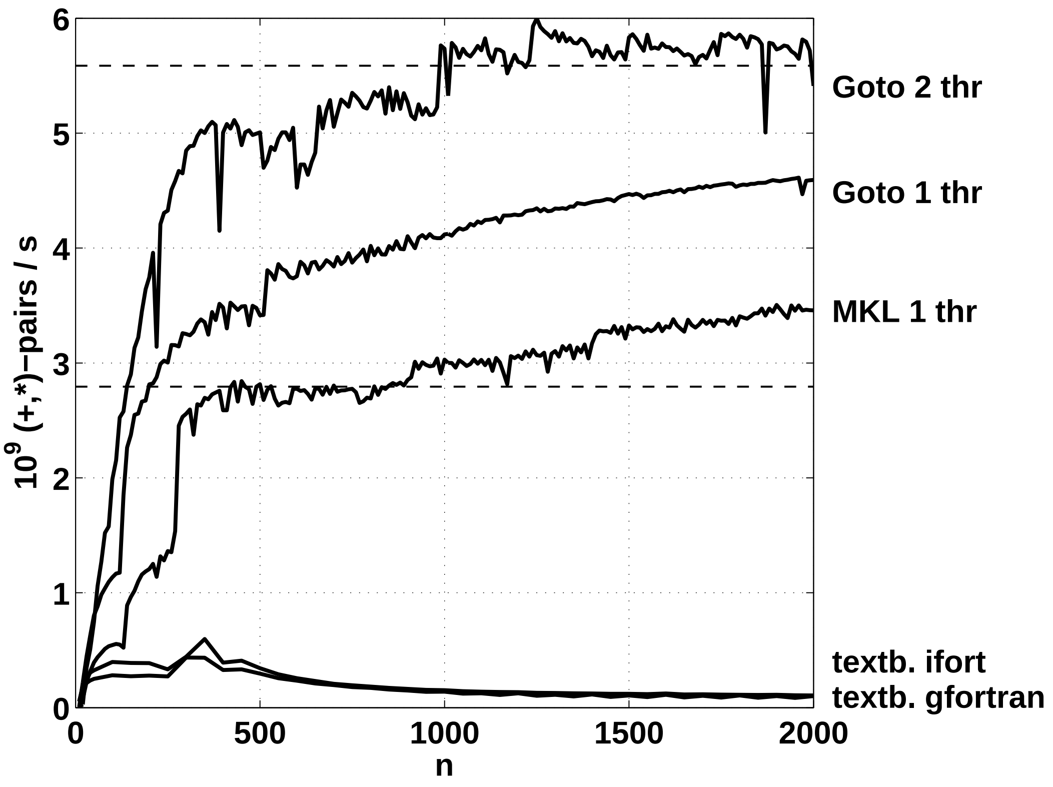
In this example we compute the Cholesky decomposition of a symmetric and positive definite matrix $\mathbf{A}$, so $\mathbf{A} = \mathbf{C}\mathbf{C}^T$, where $\mathbf{C}$ is undertriangular.
"textbook", in the figure on the next page, is one common way, often presented in textbooks, for computing $\mathbf{C}$.
Here is a Matlab-code:

62: Two important libraries, contd.

```
n = length(A);
for k = 1:n
  A(k, k) = sqrt(A(k, k) - sum(A(k, 1:k-1).^2));
  for i = k+1:n
    A(i, k) = (A(i, k) - ...
           sum(A(i, 1:k-1) .* A(k, 1:k-1))) / A(k, k);
  end
end
```

The number of $+$ and $*$ is roughly $\mathbf{n}^3/\mathbf{6}$.
The following figure shows the results of five runs.
The textbook algorithm compiled with **`gfortran`** and **`ifort`**.
Using LAPACK's **`dpotrf`** with Goto-BLAS on one and two threads.
Using MKL's **`dpotrf`** on one thread.

63: Two important libraries, contd.



Do not use simplistic algorithms from textbooks!

64: Inlining

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

65: Indirect addressing, pointers

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

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

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

66: If-statements

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

```
Original version            Optimized version

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

67: If-statements, contd.

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

Suppose **`f`** and **`g`** are (time consuming) logical functions.
`if (f(k) .and.  g(k)) then`, least likely first
`if (f(k) .or.  g(k)) then`, most likely first

Make sure that **`g`** does not have side-effects.

68: Alignment

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

work(1) = 123
```

May produce "Bus error" due to alignment problems. It is usually required that double precision variables are stored at an address which is a multiple of eight bytes (multiple of four bytes for a single precision variable). The slowdown caused by misalignment may easily be a factor of 10 or 100.

69: Closing notes

Two basic tuning principles:

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

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

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