High Performance and Parallel Computing (MA 6241)

Lecture 5

Advanced Performance Considerations

Ulrich Rüde
Lehrstuhl für Simulation
Universität Erlangen-Nürnberg

visiting Professor at
Department of Mathematics
National University of Singapore

Contents of High Performance and Parallel Computing

- Introduction
  - Computational Science and Engineering
  - High Performance Computing
- Computer Architecture
  - memory hierarchy
  - pipeline
  - instruction level parallelism
  - multi-core systems
  - parallel clusters
- Efficient Programming
  - computational complexity
  - efficient coding
  - architecture aware programming
  - shared memory parallel programming (OpenMP)
  - distributed memory parallel programming (MPI)
  - parallel programming with Graphics Cards

Homework 2 (20%)

1. Your task is to implement a matrix-matrix multiplication
   $C = A \cdot B$, where A, B, and C are square matrices.
2. Feel free to use every known algorithm and programming technique to decrease the single-core runtime of your program as long as you adhere to the following guidelines. All three matrices are
   a) represented as a linearized one-dimensional array with adjacent elements.
   b) passed to your multiplication routine in a row-major format. Meaning: it is not allowed to store one of the input matrices in a transposed layout. However, the computation of the transpose is allowed to be a part of the multiplication routine itself such that the matrix is transposed after the start of the time measurement.
   c) Make sure you use double precision floating-point operations for your multiplication. The use of threads is prohibited.
3. Measure run times for matrix sizes of 1000×1000, 1500x1500, 2000×2000, 3000×3000, 4000×4000, 5000×5000,
4. Compare your run times with that of a highly optimized professional routine (eg. dgemm from BLAS level 3)
5. Summarize your findings on p pages with $p \leqq 2$
   1. due date Feb 16, 2015.
   2. submit as pdf file by e-mail to ulrich.ruede@fau.de

Review and Summary

Memory Aliasing

Subroutine calling overhead

- Subroutines (functions) are very important for structured, modular programming.
- Subroutine calls are not cheap (on the order of up to 100 cycles).
- Passing value arguments (copying data) can be extremely expensive, when used inappropriately.
- Passing reference arguments (as in FORTRAN) may be dangerous (from a point of view of correct software).
- Reference arguments (as in C++) with const declaration.
- Generally, in *tight loops*, no subroutine calls should be used

Inlining (by using macros)

- inline declaration in C++, or done automatically by compiler.
- Macros in C (or any other language)

```
#define sqre(a)    (a)*(a)
```

- What can go wrong:

```
sqre(x+y)     ---->     x+y*x+y

sqre(f(x))    ---->     f(x) * f(x)
```

- What if `f` has side effects?

- What if `f` has no side effects, but the compiler cannot deduce that?

Aliasing

Arrays (or other data) that refer to same memory.

FORTRAN forbids aliasing, C permits aliasing $\Rightarrow$

one important reason why FORTRAN compilers may produce faster code than C compilers.

Aliasing Example

```
subroutine sub(n, a,b,c, sum)
double precision a(n), b(n), c(n)
sum= 0d0
do i=1,n
   a(i)= b(i)+ 2.0d0*c(i)
   sum= sum+b(i)
enddo
```

FORTRAN rule: Two variables cannot be aliased, when one or both of them are modified in the subroutine.

Correct call:

```
double precision a(n), b(n), c(n), sum
call sub(n,a,b,c,sum)
```

Incorrect call (forbidden). The result is undefined, i.e. compiler may produce any result:

```
double precision a(n), b(n), c(n)
call sub(n,a,a,c,sum)
```

Why should aliasing be forbidden?

Consider a compiler attempting to software-pipeline the subroutine:

```
tb= b(1)
tc= c(1)
do i=1,n-1
   ta= tb+2.d0*tc
   tc= c(i+1)

   sum= sum+tb
   tb= b(i+1)

   a(i)= ta
enddo
i=    n
ta=   tb+2.0*tc
sum=  sum+tb
a(i)= ta
```

- Assume that each block of ops can be executed in 1 cycle
- latency 1 cycle for load
- latency 2 cycles for a flop
- Program produces „wrong“ results when used with second call, since $a \equiv b$.

Original code

```
subroutine sub(n, a,b,c, sum)
double precision a(n), b(n), c(n)
sum= 0d0
do i=1,n
   a(i)= b(i)+ 2.0d0*c(i)
   sum= sum+b(i)
enddo
```

Why should aliasing be forbidden?

Consider version that is correct with aliasing (even for „sum“):

```
do i=1,n
   LOAD tc= c(i)
   LOAD tb= b(i)
   ta= tb*2d0*tc
   NOOP
   STORE a(i)= ta
   LOAD sum
   LOAD tb= b(i)
   sum= sum+ tb
   STORE sum
enddo
```

- `NOOP` indicates CPU waiting because of dependencies
- explicit `LOAD/STORE`
- couldn't we save `LOAD/STORE` of sum?

C/C++ optimizers are much more difficult to write, sometimes they cannot produce as fast code as FORTRAN

Aliasing

- Aliasing forbidden in FORTRAN (result undefined when used)
- Aliasing in C/C++ legal: Compiler must produce conservative code
- More complicated aliasing could occur, e.g. `a(i)` with `a(i+2)`.
- C/C++ compilers have a key word *restrict* or a compiler option *-noalias*.
- Programmer help for C-compilers:

```
double precision ah, bh, ch
sum= 0d0
do i=1,n
   ah= a(i), bh= b(i), ch= c(i)
   ah= bh+ 2.0d0*ch
   sum= sum+bh
   a(i)= ah
enddo
```

Another aliasing example

```
subroutine sub(n,a,b,ind)
   double precision a(n), b(n)
   integer ind(n)
   do i= 1,n
            b(ind(i))= 1.0d0 + 2.0*b(ind(i)) + 3.0*a(i)
   enddo
   return
end subroutine
```

Loop iterations are not independent (and cannot be easily pipelined) when `ind(i) = ind(i+1)`.

Some compilers have options/directives to permit aggressive optimization when the programmer guarantees that `ind(i)` is an injective mapping.

Pointers and indirect addressing are dangerous wrt. aliasing.

Optimization of Memory Access

- On machines with memory hierarchy (caches), memory access is the single most important factor for performance.
- Memory access optimization must go hand-in-hand with other optimizations (form previous chapters) either by compiler or by hand.
- IBM 590: Measure time for repeatedly copying.
  - Varying strides
  - Size of data set

  - Performance first increases then deteriorates with data set size.
  - Increase: diminishing loop overhead.
  - Decrease: L1 cache, TLB.
  - Performance deteriorates with increasing stride.
  - Less data used from each cache line.

4	1.822								
5	1.338	5.4							
6	1.021	3.3	5.3						
7	.850	1.7	3.2	5.3					
8	.772	1.6	2.1	3.3	5.3				
9	.729	1.3	1.6	2.1	3.3	5.2			
10	.713	1.1	1.3	1.6	2.2	3.3	5.1		
11	.696	1.1	1.2	1.3	1.6	2.1	3.3	5.3	
12	.691	1.0	1.1	1.1	1.3	1.6	2.1	3.3	5.2
13	.688	1.0	1.0	1.1	1.2	1.3	1.6	2.2	3.3
14	.688	1.0	1.1	1.1	1.2	1.4	1.3	1.6	2.1
15	1.829	3.7	7.7	15.2	29.3	52.5	52.6	52.9	53.3
16	1.836	3.7	7.8	15.1	29.3	52.3	52.3	52.6	52.6
17	1.837	3.7	7.8	15.2	29.2	52.6	52.4	52.3	52.7
18	1.845	3.7	8.0	15.3	31.0	53.6	53.1	52.9	52.7
19	1.855	3.7	8.0	15.4	29.6	53.1	53.1	53.0	53.3
20	1.856	3.7	8.0	15.4	29.8	53.1	53.3	53.4	53.4
	0	1	2	3	4	5	6	7	8

← data set size

stride →

Memory access cost

- sudden increase in time when vectors are longer than 2^{14} words
- small increase in time when vectors are longer than 2^{17} words
- L1 cache that can hold $2^{15}=32768$ words
- TLB that has a capacity of 2^{18}
- stride: performance decreases continously
- saturates for strides larger than or equal to 32
- 27 to 28 cycles for load or store
- cache line size: 32 words

Data locality

- Best: temporal locality
  - data fits in cache
  - reused many times in the repeated copy operations
  - load/store in this case is only .3 cycles
- Second best: spatial locality
  - stride one case
  - all the array elements that are loaded into cache are used at least once
  - cost for a cache miss is distributed over all the entries of one cache line
  - cost for a load/store is close to one cycle
- Temoral locality is difficult to achieve, often impossible due to structure of the algorithms
- Spatial locality can often be achieved by clever data structures.

Loop fusion to reduce memory references

```
do i=1,n
   do j=1,3
      at = fxyz(j,i)
      vxyz(j,i)= vxyz(j,i) + (.5d0*dt/rmass) * (at + gxyz(j,i))
      gxyz(j,i)= at
enddo enddo
do i=1,n
   do j=1,3
      rxyz(j,i)= rxyz(j,i) + dt*vxyz(j,i)+(.5d0*dt*dt/rmass) * fxyz(j,i)
enddo enddo
```

versus

```
do i=1,n
   do j=1,3
      at = fxyz(j,i)
      vxyz(j,i)= vxyz(j,i) + (.5d0*dt/rmass) * (at + gxyz(j,i))
      gxyz(j,i)= at
      rxyz(j,i)= rxyz(j,i) + dt*vxyz(j,i) + (.5d0*dt*dt/rmass) * fxyz(j,i)
enddo enddo
```

Data locality and the conceptual layout of a program

```
dimension rx(n),ry(n),rz(n),fx(n),fy(n),fz(n)
do i=1,n
do j=1,i-1
   dist2= (rx(i)-rx(j))**2 +
          (ry(i)-ry(j))**2 + (rz(i)-rz(j))**2
            if (dist2.le.cutoff2) then
               dfx= ......
               dfy= ......
               dfz= ......

               fx(j)= fx(j)+dfx
               fy(j)= fy(j)+dfy
               fz(j)= fz(j)+dfz
               fx(i)= fx(i)-dfx
               fy(i)= fy(i)-dfy
               fz(i)= fz(i)-dfz
            endif
enddo
enddo
```

Data locality and the conceptual layout of a program

```
dimension r(3,n), f(3,n)
do i=1,n
do j=1,i-1
   dist2= (r(1,i)-r(1,j))**2 +
         (r(2,i)-r(2,j))**2 + (r(3,i)-r(3,j))**2
   if (dist2.le.cutoff2) then
       dfx= ......
       dfy= ......
       dfz= ......

       f(1,j)= f(1,j)+dfx
       f(2,j)= f(2,j)+dfy
       f(3,j)= f(3,j)+dfz
       f(1,i)= f(1,i)-dfx
       f(2,i)= f(2,i)-dfy
       f(3,i)= f(3,i)-dfz
    endif
enddo
enddo
```

Cache thrashing

```
program cache_thrash
   implicit real*8 (a-h,o-z)
   parameter(nx=2**13,nbuf=0)
   parameter(nx=2**13,nbuf=81)
   dimension x(nx+nbuf,6)
   nl=2**10
   call sub(nl, x(1,1), x(1,2), x(1,3), x(1,4), x(1,5), x(1,6))
end
subroutine sub(n, x1, x2, x3, x4, y1, y2)
   implicit real*8 (a-h,o-z)
   dimension x1(n), x2(n), x3(n), x4(n), y1(n), y2(n)
   do i=1,n-1,2
       y1(i+0)= x1(i+0) + x2(i+0)
       y1(i+1)= x1(i+1) + x2(i+1)
       y2(i+0)= x3(i+0) + x4(i+0)
       y2(i+1)= x3(i+1) + x4(i+1)
   enddo
   return
end
```

Constants alteratively defined like this or that

Cache thrashing

- Data for IBM 590 (a now obsolete machine, just for demo purposes)
- nbuf=0: 3.5 Mflops
- nbuf=81: 67.0 Mflops
- Note: subroutine remains unchanged!
- Reason: For nbuf=0, all six memory references are mapped to the same cache slots.
- In case of the IBM there are just 4 of them (4-way set-associative cache)
- Note that working set size is $6 \times 2^{10} = 6144$ words is smaller than the cache size, so they should all fit in cache.
- Conflict misses vs. capacity misses.
- Array padding (using dummy variables or extra array elements) can help with avoiding conflict misses.
- Especially dangereous: array dimensions which are powers of 2.

Avoiding Conflict Misses (example)

```
implicit real*8 (a-h,o-z)
parameter(nn= 1024, nx= 2**15)
dimension w(nn*6),x(nx,6),ist(6)
nl=....  x=.....
iist=1
do j=1,6
   ist(j)= iist
   do i=1,nl
              w(iist)= x(i,j)
              iist=    iist+1
   enddo
enddo
call sub(nl, w(ist(1)), w(ist(2)), w(ist(3),
                  w(ist(4)), w(ist(5)),w(ist(6)))
end
```

Other considerations

Languages with array syntax FORTRAN 90 or operator overloading in C++ permit expressions like

```
D= A*B*C
```

(meaning matrix products). Here the extra creation of temporaries may hamper performance.

Type conversions:

```
i= i+1 versus    i= i+1.0
```

Sign conversions:

```
a(i)= -(1.d0 - .5d0*b(i)) versus
a(i)= -1.d0+.5d0*b(i))
```

Complex arithmetic is ``good", because it has a good ratio of memory operations to floating point operations, but unfortunately not built into the laguage for C++/C

Assume that `a(i)` changes only little from `i` to `i+1`

original version

```
do i=1,n
   b(i)= a(i)**(1d0/3d0)
enddo
```

modified version (1)

```
x= a(1)**(1d0/3d0)
b(1)= x
do i=2,n
   r= a(i)
   do
      d= x-r/x**2
      x= x-(1d0/3d0)*d
   until d<TOL
   b(i)= x
enddo
```

The modified version applies a Newton iteration to find $\rho^{1/3}$ using

$$x \leftarrow x - 1/3\ (x - \rho/x^2)$$

When the initial guess (using the previous x) is good enough, and, if additionally, the `i` loop is unrolled and independent variables are introduced, the code may be significantly faster than the original one.

modified version (2) - after loop unrolling

```
"initialize x0,x1,x2,x3“
do i= 1,n,4
ro0=a(i+0)
ro1=a(i+1)
ro2=a(i+2)
ro3=a(i+3)
do
        d0=x0-ro0/x0**2
        d1=x1-ro1/x1**2
        d2=x2-ro2/x2**2
        d3=x3-ro3/x3**2
        x0=x0-.33333333333333d0*d0
        x1=x1-.33333333333333d0*d1
        x2=x2-.33333333333333d0*d2
        x3=x3-.33333333333333d0*d3
until d0**2+d1**2+d2**2+d3**2 < 1.d-20
"store results"
```

Eliminating Overheads: IF statements

IF statements

- prohibit some optimizations (e.g. loop unrolling in some cases)
- evaluating the condition takes time
- CPU pipeline is interrupted
- (jump prediction)
- Goal: Avoid IF-statements in the innermost loops by restructuring the program.
- No generally applicable techniques exist, each case must be studied individually.

Eliminating IF-statements

```
do j=-n,n
do i=-n,n
   if(i**2 + j**2) .ne. 0) then
       u(i,j) = v(i,j)/(i**2 + j**2)
   else
       u(i,j) = 0d0
   endif
enddo
enddo
```

Either: Partition loop into blocks

`j=-n,-1; j=0; j=1,n`

and `i` accordingly, so that no IF statements are necessary

Or: tolerate DIV by 0 exception and fix wrong value afterwards.
(special Flags for Compiler may be necessary, to exploit special features of IEEE floating point standard)

Another example with IF

```
subroutine thresh(n,a,th,ic)
   double precision a(n)
   ic= 0
   tt= 0d0
   do j=1,n
      tt= tt+a(j)*a(j)
      if (sqrt(tt) .gt. th) then
         ic= j
         return
      endif
   enddo
   return
end
```

avoid sqrt in condition

add `tt` in blocks of e.g. 100 (without condition)
and repeat last block, when condition is violated

```
ic=0
t1=0.d0;  t2=0.d0;  t3=0.d0; t4=0.d0
do j=1, n-lot+1, lot
   do i=0, lot-1-3, 4
      t1=t1+a(j+i+0)**2;
      t2=t2+a(j+i+1)**2
      t3=t3+a(j+i+2)**2;
      t4=t4+a(j+i+3)**2
   enddo
   tt=t1+t2+t3+t4
   if (tt.ge.threshsq) then
      do i=lot-1, 0, -1
         tt=tt-a(j+i)**2
         if (tt.lt.threshsq) then
            ic=j+i
            return
         endif
      enddo
   endif
enddo
```

Another example with IF - continued

```
sqrt(tt) .gt. th
```

replaced by

```
tt .gt. th**2
```

add `tt` in blocks of `lot` (e.g. =100), without condition

repeat last block, when condition is violated

Prefetching

- memory access limiting performance bottleneck
- two possible reasons: bandwith or latency
  - latency: If load instructions are scheduled briefly before the data item is needed, it takes several cycles for the cache miss to be serviced. The CPU will idle until the data arrives in registers.
- Prefetching: request data long before needed.
  - bridge time between fetch and usage by other computation
- compiler tries to schedule loads ahead
  - limits due to limited number of registers
- solution: instruction that transfers one or several cache lines from the main memory into cache
  - no longer necessary to schedule the load long ahead, but only slightly earlier
  - Prefetch instruction
  - not part of programming languages, but available to the programmer on some machines

Misalignment of Data

- Double precision floating point numbers, 8 bytes long, are usually aligned in such a way that they start at a memory location that is a multiple of 8 bytes.
  - misalignment especially problematic when using SSE/ AVX instructions
  - Some language instructions, specifying exact memory locations for data, can prevent the compiler from doing the best data alignment
  - common blocks in Fortran
  - structs in C, class variables in C++?
- compiler warnings?

Meta-Programming in C++

- Contant evaluation works for

```
x= 3*4.0+y
```

  - Compiler generates

```
x= 12.0+y
```

- But what happens for

```
x= factorial(4)+y
```

(where `factorial(n) = n! = n*factorial(n-1)`)

Evaluation of Factorial at Compile-Time

- Consider the C++ construction

```
template<int N> class Factorial {
public:
    enum { value = N * Factorial<N-1>::value };
};
class Factorial<1> {
public:
    enum { value = 1 };
};
```

- Then, the compiler replaces

```
x = Factorial<4>::value + y;
```

- by

```
x = 24 + y;
```