To code or not to code?

Programming techniques – week 10

Optimization and numerical libraries

Optimization

- First rule: Do not optimize!
- What if the program is too slow?
  - find optimal algorithm
  - use libraries
- What if the program is still too slow?
  - use profiling to determine which parts are slow
  - investigate slow part and check that data structures are optimal
    - are arrays, lists or trees better?
  - is the algorithm optimal?
    - check literature for better algorithms
    - use libraries
  - only then think about optimizing
- Consider parallelization or vectorization

Profiling

- is used top determine how much time is spent in which program parts
- Three easy steps:
  - compile the program with the `-p` option
  - run the program
  - use `prof` to look at the performance data
- Alternative using gprof:
  - compile with the `-pg` option
  - run the program
  - use `gprof` to look at the performance data
  - includes time spent in called functions
- See the man pages for details about these programs

Choice of data structures

- choose your data structures depending on the use
- was discussed before and in the exercises:
  - if you need random access use an array
  - if you need to insert in the middle use a list
  - if you need both use a tree
- use the standard C++ library containers wherever possible. They are (nearly) optimal.
- if you need a container not included:
  - design your own in the STL style
  - make it available to others

Example: the best data structure for the Penna model

- We picked a linked list because removal from the middle of a vector is slow
- However a vector might be faster:
  - We do not care about the order of the animals
  - We can implement a special remove_if:
    - Replaces the removed animal with the last one
    - This makes removal fast
- We can code a container derived from vector with a special remove_if
  - Will be faster than a std::list
  - Will require only a one-line change in the Penna code
- Look at penna_vector.h

Choice of algorithms

- Look at the scaling of the algorithms with problem size:
- Fourier transform
  - Simple: $O(N^2)$
  - Fast Fourier Transform: $O(N \log N)$
- Matrix-Matrix multiplication
  - Simple: $O(N^3)$
  - Strassen: $O(N^{2.8})$
  - Coppersmith and Winograd: $O(N^{2.376})$
- Eigenvalues:
  - all eigenvalues, dense matrix: $O(N^3)$
  - some eigenvalues, dense matrix: $O(N^2)$
  - some eigenvalues, sparse matrix: $O(N)$

The Strassen algorithm

- is one example why you should use libraries even for trivial-looking operations
- Normal matrix-matrix multiplication is order $O(N^3)$
- Strassen algorithm is $O(N^{\log 7/\log 2})=O(N^{2.8})$
  - write matrix as four submatrices

$$C = AB \qquad \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

  - use a clever scheme

$$Q_1 = (a_{11} + a_{22})(b_{11} + b_{22})$$
$$Q_2 = (a_{21} + a_{22})b_{11}$$
$$Q_3 = a_{11}(b_{12} - b_{22})$$
$$Q_4 = a_{22}(-b_{11} + b_{21})$$
$$Q_5 = (a_{11} + a_{12})b_{22}$$
$$Q_6 = (-a_{11} + a_{12})(b_{11} + b_{12})$$
$$Q_7 = (a_{12} - a_{22})(b_{21} + b_{22})$$

$$c_{11} = Q_1 + Q_4 - Q_5 + Q_7$$
$$c_{21} = Q_2 + Q_4$$
$$c_{12} = Q_3 + Q_5$$
$$c_{22} = Q_1 + Q_3 - Q_2 + Q_6$$

Comparing matrix multiplication algorithms

- Standard algorithm is $O(N^3)$
  - N^3 multiplications
  - $N^2(N-1)$ additions
- Strassen algorithm takes
  - 7 multiplications of matrices of size $N/2$
  - 18 additions of matrices of size $N/2$
- What is the complexity $T_{\text{strassen}}(N)$?
  - $T_{\text{strassen}}(N) = 7\, T_{\text{strassen}}(N/2) + 18/4\, N^2$
- Assuming $T_{\text{strassen}}(N) > O(N^2)$
  - $O(T_{\text{strassen}}(N)) = 7\, O(T_{\text{strassen}}(N/2))$
  - $O(T_{\text{strassen}}(2N)) = 7\, O(T_{\text{strassen}}(N))$
  - $\Rightarrow T_{\text{strassen}}(N) = O(N^{\log 7/\log 2})$

How do we find the best algorithm?

- Look in books of Knuth
- Search the SIAM journals
- Do not trust the Numerical Recipes too much
- But the easiest solution is: use a library
  - bug free (less buggy than your codes)
  - optimized (probably better than you can do)
  - well documented (do you ever document your codes?)
  - supported on most architectures
- A huge collection is available on netlib at http://www.netlib.org/
- In the next weeks we will introduce a variety of useful libraries

How to optimize

- Generally you should use a library instead of optimizing yourself.
- But as computational scientists you will sometimes
  - have to write libraries
  - enter new research fields and algorithms where there is no library
- We will learn how to optimize
  - Optimization using assembly language
  - Classical optimization techniques for any language
  - Optimization in C++
- And look at libraries using these optimization techniques

Optimization in assembly language

- Sometimes the CPU possesses machine language instructions that cannot be used directly from a high level language
  - Bit counts
  - Vector instructions (discussed in “Numerisches Paralleles Rechnen”)
    - MMX and SSE on Pentium
    - Altivec on PowerPC
- Assembly languages instructions can be mixed with C++
  - Advantage: can speed up code
  - Disadvantage: code becomes non-portable
  - useful only in very rare cases, but can potentially make a big difference
- Best approach
  - Encapsulate assembly language call in a library

Example: counting leading zeroes in an integer

- Problem: count the number of leading zeroes in a 32-bit integer
  - Can be used to get the position of the highest bit set
  - Can be used to calculates the logarithm base 2 of an integer
- Solution in C++: requires a loop

```
int count_leading_zeroes(int x) {
  for (int i = 0 ; i<32 ;++i)
    if (x&(1<<(31-i)))
      return i;
  return 32;
}
```

- Solution in PowerPC-assembler: (powerpc_asm.C)

```
inline int count_leading_zeros (int x) {
  int c;
  asm ("cntlzw %1,%0" : "=r" (c) : "r" (x) );
  return c;
}
```

Inline assembly statements

- We used an inline assembly statement, which mixes assembly language with C++:
  - `asm ("cntlzw %1,%0" : "=r" (c) : "r" (x) );`
- Explaining the syntax:
  - `asm(…)`: inserts an inline assembly language statement
  - `cntlzw r9,r15` : puts the number of leading zeroes in register 9 into register 15
  - `cntlzw %1, %0` : we do not know which register the compiler will use and thus use placeholders `%0` and `%1` (use `%2` … if more registers are needed)
  - `: "=r" (c)` : puts the variable `c` into the register marked by `%0` (and after the execution assigns the value of the register `%0` to `c`
  - `: "r" (x):` : The second `:` marks the input variables that will not be modified. This statement tells the compiler to load variable `x` into register `%1`
- To learn more, search the webs to find processor-specific instructions
  - But be warned that it is tricky

Another example: long integers

- How is 64-bit addition implemented on a 32-bit machine?
- Just as you learned adding numbers in primary school:
  - Add the low words and remember the carry
  - Add the high words and the carry
- Example: `add64.C`
  - `g++ -c -save-temps -O add64.C`
  - Look at `add64.s`
- Compare to a 64-bit machine
  - Addition done in one step!

128 bit integers in `int128.C`

- If we need 128 bit integers we need to define a new class:
  - Build a 128 bit integer from two 64 bit ones:

```
struct int128 {
  unsigned long long low;
  long long high;
};
```

- How do we add them?
  - Adding low and high words separately will not be correct since the carry is not used

```
int128 operator+(int128 x, int128 y) {
  int128 result;
  result.low=x.low+y.low;
  result.high=x.high+y.high;
  // wrong result: this does not use carry of previous addition
  return result;
}
```

  - Inline assembly language can be used to change “add without carry” to “add with carry”

Helping the compiler optimize

- Using an optimizing compiler is easier than writing fast code in assembly language
- We will now discuss techniques to optimize code.
- Some can be done by the compiler
  - You need to know about them to realize which optimizations you do not need to perform
  - Not optimizing manually what the compiler can do for you can help keep the code cleaner
- Some have to be done by you
  - But only after you have determined by profiling that which function is the bottleneck

Copy propagation (automatic)

- is usually done by any modern compiler and need not be done by you.
- It changes

```
x = y;
z = 1 + x;
```

- to

```
x = y;
z = 1 + y;
```

- and allows pipelining of the two statements

Constant folding (automatic)

- Is also done by modern compilers and need not be done by you.
- It changes

```
const int x = 100;
int z = 2*x;
```

- to

```
const int x = 100;
int z = 200;
```

- And performs the multiplication at compile-time

Dead code removal (automatic)

- Is most useful in connection with template parameters. The compiler can detect if a statement is never executed
- It changes

```
int n = 100;
if (n<1)
  std::cerr << "n less than one";
…
```

- to

```
int n = 100;
…
```

- thus removing the code that will never be executed

Strength reduction (automatic)

- The compiler often realizes how to simplify expressions, making them faster
- It changes

```
x = 2 * y;
```

- to

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

- or (for integer y)

```
x= ( y << 1 );
```

- And performs the faster operation

Variable renaming (automatic)

- Is also often done by the compiler to expose potentials for pipelining
- It changes

```
int x = y * z;
int q = r + x * x;
    x = a + b;
```

- to

```
int x0 = y * z;
int q = r + x0 * x0;
int x = a + b;
```

- And can now pipeline the last two statements

Common subexpression elimination (automatic)

- Can be done by the compiler in simple cases:
- It changes

```
d = c * (a + b);
e = (a + b) / 2;
```

- to

```
temp = (a + b);
d = c * temp
e = temp / 2;
```

- And saves one addition

Common subexpression elimination (manual)

- If a function call is involved you have to perform common subexpression elimination manually!
- You have to manually change

```
d = c * f(x);
e = f(x) / 2;
```

- to

```
temp = f(x);
d = c * temp
e = temp / 2;
```

- Since the compiler does not know whether `f(x)` is always the same number
  - maybe `f` is your name for a random number generator ….

Loop invariant code motion (automatic)

- Scientific programs spend most of their time in loops. We have to minimize the work done in those loops. A compiler can help in simple loops:
- It changes

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

- to

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

Loop invariant code motion (manual)

- In complex loops or I function calls are involved, we have to manually optimize
- We have to manually change

```
for (int i=0; i<n; ++i) {
  a[i] = b[i] + f(x);
  e = g(y);
}
```

- to

```
temp = f(x);
for (int i=0; i<n; ++i) {
  a[i] = b[i] + temp;
}
e = g(y);
```

Induction Variable Simplification (automatic / manual)

- Induction variable simplification is changing

```
for (int i=0; i<n; ++i) {
  k =  4*i + m;
  …
}
```

- to

```
k = m;
for (int i=0; i<n; ++i) {
  …
  k += 4;
}
```

Importance of Induction Variable Simplification

- Take care of hidden complexities in array subscripts: the code

```
for (int i=0; i<n; ++i) {
  x[4*i] = …
 }
```

- Is actually

```
for (int i=0; i<n; ++i) {
   *(x+4*i) = …
}
```

- And is faster coded as

```
for (T* p=x; p<x+4*n; p+=4) {
   *p = …
}
```

Loop unrolling (automatic / manual)

- The loop for a scalar product

```
double s=0.;
for (int i=0; i<3; ++i)
   s += x[i] * y[i];
```

- Is much faster when unrolled as

```
double s = x[0] * y[0] + x[1] * y[1] + x[2] * y[2];
```

- For two reasons:
  - No loop control statements
  - Easy pipelining
- Simple loops can be unrolled by compilers with high enough optimization settings (-funroll-loops on gcc)

Partial loop unrolling (automatic / manual)

- The loop for an array product

```
for (int i=0; i<N; ++i)
  a[i] = b[i] * c[i];
```

- Is much faster when partially unrolled as (for `N` a multiple of 4)

```
for (int i=0; i<N; i+=4) {
  a[i]   = b[i]   * c[i];
  a[i+1] = b[i+1] * c[i+1];
  a[i+2] = b[i+2] * c[i+2];
  a[i+3] = b[i+3] * c[i+3];
}
```

- Because pipelining can again be used

Aiming for unit stride (manual)

- The loop for a matrix sum

```
for (int i=0; i<N; ++i)
  for (int j=0; j<N; ++j)
    a[i][j] = b[i][j] + c[i][j];
```

- Is much faster than

```
for (int i=0; i<N; ++i)
  for (int j=0; j<N; ++j)
    a[j][i] = b[j][i] + c[j][i];
```

- Because the unit stride (sequential memory access) in the inner loop uses the cache much better

In-cache matrix-matrix multiplications

- The matrix multiplication

```
for (int i=0; i<N; ++i)
  for (int j=0; j<N; ++j)
    for (int k=0; k<N; ++k)
      a[i][j] += b[i][k] * c[k][j];
```

- Is better changed to get unit stride in the inner loop

```
for (int i=0; i<N; ++i)
  for (int k=0; k<N; ++k) {
    temp = b[i][k];
    for (int j=0; j<N; ++j)
      a[i][j] += temp * c[k][j];
  }
```

Out-of-cache matrix multiplications: blocking

- Performance degrades if the matrix does not fit into the cache
- Split the matrix into smaller blocks and perform in-cache multiplications of the blocks:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} \begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{pmatrix}$$

- The size of the blocks a_{ij}, b_{ij} and c_{ij} depends on the types and sizes of the caches.
- This is tricky and we will learn about libraries doing it for you next week

Libraries for linear algebra

- Fortran libraries
  - BLAS
  - LAPACK
- C++ libraries
  - Blitz++
  - uBlas
  - ITL and IETL
  - POOMA
- The Fortran libraries are well optimized but difficult to call
- The C++ libraries are easier to use but not as complete yet
- Fortran can also be called from C++, as we will do in one of the exercises

Calling Fortran from C++

- declare the function `extern "C"`
- pass all parameters by pointers or reference
- The naming depends on the machine
  - Fortran `FUNC` -> C `func_` with GNU or Intel compilers
  - Fortran `FUNC` -> C `func` with IBM or Cray compilers
- Program has to be linked with Fortran runtime libraries
- Take care of:
  - Fortran real is float on most workstations but double on Cray
  - Fortran integer is usually an int
  - Array indices in Fortran usually start from 1
  - Storage order of matrices is reversed
  - Fortran `a(i,j)` is C `a[j-1][i-1]`

A calling example: DDOT

- The DDOT function in the BLAS library calculates the scalar (dot) product of two double precision vectors:
  - ```
    DOUBLE PRECISION FUNCTION DDOT(N,X,INCX,Y,INCY)
    DOUBLE PRECISION X(*),Y(*)
    INTEGER INCX,INCY,N
    ```
- To call DDOT from C++ we need to declare it as:
  - ```
    extern "C" double ddot_(int& n, double *x, int& incx,
                            double *y, int& incy);
    ```
- To link we need to add the following options:
  - On the D-PHYS Linux machines: -lblas -lg2c -lm
  - On MacOS X: -framework vecLib
  - How to find options for other machines will be explained in the exercises

BLAS

- is short for Basic Linear Algebra Subroutines
- is a Fortran library
- BLAS level 1
  - vector operations: addition, dot product, …
- BLAS level 2
  - matrix-vector operations
- BLAS level 3
  - matrix-matrix operations
- use the BLAS wherever possible
  - optimized assembler code versions available on most machines
  - generic Fortran version available on www.netlib.org
- Homework: if you have a Unix or Linux machine at home download and install BLAS and LAPACK

ATLAS

- We learned in the last weeks that optimizing matrix operations can be tricky:
  - For which sizes should we use Strassen's algorithm?
  - How large should we choose sub-blocks to be get optimal cache effects by blocking?
- The Fortran BLAS on netlib works on all machines and thus cannot be optimized to the CPU, cache size and cache type of your machine
- On supercomputers the vendors provide a hand-optimized BLAS
- ATLAS is the solution for the rest of us:
  - A self-tuning library
  - When being installed it benchmarks hundreds of blocking strategies until it finds the optimal one for your machine
  - It then compiles a BLAS with these optimal settings

ATLAS benchmark example 1

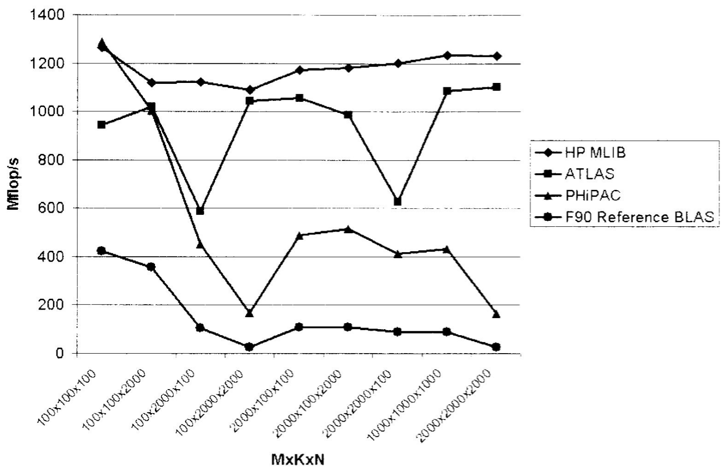

Figure 9-1 Comparison of matrix multiplication performance with various software packages.

ATLAS benchmark example 2

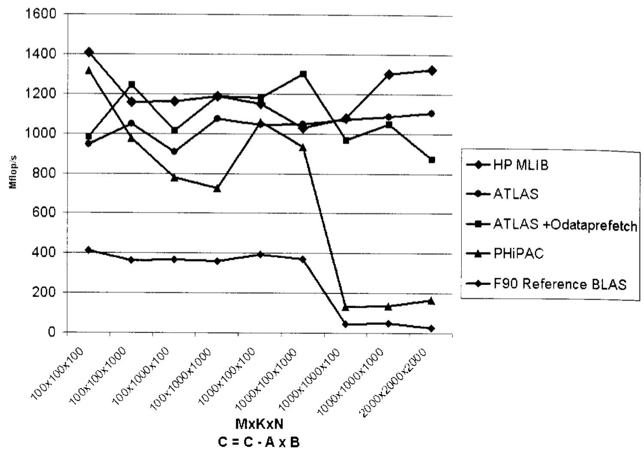

Figure 9-2 Comparison of transposed matrix multiplication performance using various software packages.

LAPACK Overview

- is a Linear Algebra PACKage
- ScaLAPACK is the parallel version
- has functions for
  - eigenvalues and -vectors
  - linear equation solvers
  - matrix inversions
  - determinants
  - ...
- special functions for
  - symmetric or Hermitian matrices
  - tridiagonal matrices
  - banded matrices

LAPACK & BLAS naming conventions

- functions are of the form
  - PTTXXX
- where P is the precision
  - **S** single precision real
  - **D** double precision real
  - **C** single precision complex
  - **Z** double precision complex
- TT is the matrix type:
  - **GE** general,
  - **SY** symmetric
  - **HE** Hermitian
  - **...**
- Example: DGEEV is the double precision general eigensolver

Important LAPACK functions

- Eigensolvers: ***EV for
  - we will use DSYEV or SSYEV for the exercises
- Linear equation solvers: ***SV
- Linear least squares: ***LS
- Factorizations:
  - LQ: ***LQF
  - QL: ***QLF
- Matrix inverse: ***TRI

FFTW

- The fastest open source Fourier transfrom library is the self-tuning FFTW ("Fastest Fourier Transform in the West")
- Available from http://www.fftw.org/

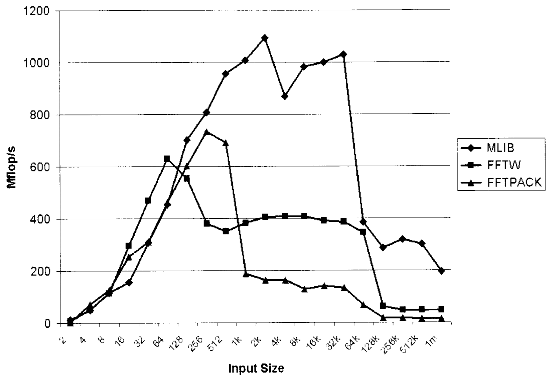

Figure 9-3 Comparison of one-dimensional, complex double precision FFT performance using various software packages.

Commercial libraries: NAG, IMSL, …

- add many more functions, like:
  - optimizations
  - non-linear root solvers
  - interpolation
  - statistical functions
  - …
- They are however not free but commercial libraries
  - cost a lot of money
  - not suitable for private use
  - ETH has a site license: you can us them in your research