Writing Faster Code

Most examples are from Hager/Wellein: Introduction to High Performance Computing for Scientists and Engineers, 2010

- Do less work!
- Avoid expensive operations by precomputation.
- Shrink the work set.
- Avoid branches.
- Unroll loops.
- Use SIMD instructions.
- Inline functions.

Do less Work: Implement a cheaper, mathematically equivalent expression

Linear algebra: Consider $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{n \times o}$, $v \in \mathbb{R}^o$

$$A \cdot B \cdot v = \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} b_{11} & \dots & b_{1o} \\ \vdots & \ddots & \vdots \\ b_{n1} & \dots & b_{no} \end{bmatrix} \begin{bmatrix} v_1 \\ \vdots \\ v_o \end{bmatrix}.$$

a) Computing

$$(A \cdot B)v$$

uses $m \cdot o \cdot n$ flops to compute the $m \times o$-matrix $C = A \cdot B$ and, additionally, $m \cdot o$ flops to compute $C \cdot v \Longrightarrow$ **Total flops** $m \cdot o \cdot (n + 1)$**.**

b) Computing instead

$$A \cdot (Bv)$$

needs $n \cdot o$ flops to compute the vector $w = Bv$ of length n and, additionally, $m \cdot n$ flops to compute the vector $A \cdot w$ of length $m \Longrightarrow$ **Total flops** $n \cdot o + m \cdot n$**.**

$\Longrightarrow$ For $n \approx m \approx o$ we have $O(n^3)$ flops for the slow version vs. $O(n^2)$ flops for the fast version!

Do less Work: Rearrange Code in Loops

Rearrange code to eliminate operations.

```
int i, FLAG = 0;
for (i = 0; i < N; i++) {
  if (complex_func(i) < THRESHOLD) {
    FLAG = 1;
  }
}
```

If, on average, the boolean expression is true for many array elements, exitig the loop early can save many unnecessary instructions.

```
int i, FLAG = 0;
for (i = 0; i < N; i++) {
  if (complex_func(i) < THRESHOLD) {
    FLAG = 1;
    break;
  }
}
```

Do less Work: Eliminate Common Subexpressions

Subexpressions, particularly containing strong operations, are sometimes better precomputed and stored. Example: loop-invariant code motion

```
int i;
double A[N], x, s, r;
for (i = 0; i < N; i++) {
  A[i] = A[i] + s + r*sin(x);
}
```

```
int i;
double A[N], x, s, r, tmp;
tmp = s + r*sin(x);
for (i = 0; i < N; i++) {
  A[i] = A[i] + tmp;
}
```

Note: Compilers often fail to make such optimizations. Caveat: Modified code will in general not yield bitwise identical results.

Sometimes mathematically equivalent rearrangements may affect numerical stability, see, e.g. modified Gram-Schmidt vs. Gram-Schmidt.

Avoiding expensive operations: “Strength Reduction”

Calls like

```
int i=2;
y=pow(x,i);
```

will, most likely, not be optimized by the compiler. If always $i = 2$ then

$$y = x * x;$$

will be much faster!

Avoiding expensive operations: Precompute!

Strong operations: division, transcendental functions, etc.; can take hundreds of clock cycles (remember: sin/cos 50-200 clock cycles).

Weak operations: cheaper alternatives.

```
int iL, iR, iU, iO, iS, iN;
double edelz, tt, BF;
// inside some expensive loop:
edelz = iL+iR+iU+iO+iS+iN;
BF = 0.5*(1.+tanh(edelz/tt)); // expensive operation
```

Precomputed table look-up needs no further computation; data fits in L1 cache.

```
int iL, iR, iU, iO, iS, iN, i;
double tt, BF, tanh_table[13];
// precompute table once and for all
for (i = 0; i <= 13; i++)
  tanh_table[i] = 0.5*(1.+tanh((i-6.)/tt));
// ... //
// usage of table data inside some expensive loop:
BF = tanh_table[iL+iR+iU+iO+iS+iN+6];
```

Shrink the Working Set

Shrink the working set such that it may fit into L1/L2 or L3-cache.

Working Set := Amount of memory accessed during a significant portion (e.g. an iteration) of an algorithm.

Helps to

- increase number of cache hits
- may eliminate swapping to disk (not allowed in supercomputers anyway)

Note that the use of shorter data types (i.e. `short int`, `char`, or bits in an `int` to store several flags instead of using several `bool` may save memory but will/may affect performance. For example alignment (dt. “Speicherausrichtung”) issues can occur: Typically processors are fastest at accessing data that is aligned according to the processor word, i.e. a data type should be stored at an address that is a multiple of the word size.

Compilers can/will take care of this, e.g., by *padding* data structures with zeros. But this may annihilate the attempts to reduce the size of a data structure in memory.

Avoid Branches!

Tight loops: loops containing few operations. Ideal candidates for pipelining or loop unrolling. Compiler cannot do this in presence of branch statements. Branch prediction can guess wrong branch (branch miss), ! pipeline flush.

Instead of a single outer loop with branches inside (left) use several outer loops with no branches (right).

```
int i, j, sign;
double A[N][N], B[N], C[N];
for (j = 0; j < N; j++) {
  for (i = 0; i < N; i++) {
    if (i >= j) {
      sign = 1;
    } else if (i < j) {
      sign = -1;
    } else {
      sign = 0;
    }
  C[j] += sign*A[i][j]*B[i];
  }
}
```

```
int i, j, sign;
double A[N][N], B[N], C[N];
for (j = 0; j < N; j++) {
  for (i = j+1; i < N; i++) {
    C[j] += A[i][j]*B[i];
  }
}

for (j = 0; j < N; j++) {
  for (i = 0; i < j; i++) {
    C[j] -= A[i][j]*B[i];
  }
}
```

Use of SIMD Instructions

Vector instructions/units on recent CPUs (MMX, SSEn, AVX). Smaller data type can double execution speed (float vs. double). No improvement will be obtained for memory-bound code.

Example loop unrolling:

```
float r[N], x[N], y[N];
for (i=0; i<N; i++)
   r[i] = x[i] + y[i];
```

Each loop iteration independent, no branches.

Collect several iterations into group whose data fits in one SIMD register.

Finish with any necessary remainder pass.

Alignment issues exist.

Use SIMD Instructions (2)

```
// Pseudocode of a vectorized loop
rem = N % 4

for (i=0; i<N-rem; i+=4) {
"load R1 = [ x[i] x[i+1] x[i+2] x[i+3] ];"
"load R2 = [ y[i] y[i+1] y[i+2] y[i+3] ];"

// SIMD add of 4 packed SP in 1 cycle
"R3 = add(R1,R2);"

"store [ r[i] r[i+1] r[i+2] r[i+3] ] = R3;"
}

// remainder is done sequentially
for (i=N-rem; i<N; i++) r[i] = x[i] + y[i];
```

GCC will generate SSE instructions from floating point code if `-mfpmath=sse` is enabled. This is the default choice for the x86-64 compiler and the “resulting code should be considerably faster in the majority of cases” (man page).

Use SIMD Instructions (3)

SSE on Intel CPUs: permit independent move of lo/hi 64 bits of SIMD register. Can mix scalar loads/stores with SIMD arithmetic. Example: vector addition loop (pseudo code):

```
rem = N % 2;
for (i=0; i<N-rem; i+=2) {
  // scalar loads
  "load R1.lo = x[i];"
  "load R1.hi = x[i+1];"
  "load R2.lo = y[i];"
  "load R2.hi = y[i+1];"

  // packed addition
  "R3 = add(R1,R2);"

  // scalar stores
  "store r[i] = R3.lo;"
  "store r[i+1] = R3.hi;"

  // remainder loop
  if (rem==1) r[N] = x[N] + y[N];
```

Use SIMD Instructions (4)

In a loop contains a dependency

```
for (i=1; i<N; i++) a[i] = s*a[i-1];
```

no SIMD acceleration will be done, i.e. compiler will reverts to scalar operation.

Use SIMD Instructions (5)

- Use of SIMD/SSE is compiler specific. Can be achieved by compiler directives or inline assembly code (not portable).
- Use of SSE can be checked by inspecting the assembly code.
- Some recent compilers. i.e. by MS) do not support inline assembly code. Here, usage of "compiler intrinsics" is the only possibility. Compiler intrinsics such as `_byteswap_uint64`, declaration

  ```
  unsigned __int64 __cdecl _byteswap_uint64(unsigned __int64);
  ```

  are fast but not portable.
- Usage of optimized linear algebra libraries (e.g. an optimized blas).

Remark: `__cdecl` the standard C and C++ calling convention. Here, the parameters are removed from the stack by the caller. As result functions with a variable number of arguments are allowed, see e.g. `printf`, as opposed to `__stdcall`. `__cdecl` does not affect the C++ name mangling.

Use SIMD Instructions (6)

```
typedef union {
  int val[4];
  __m128i vec;
  } mmevec;
void simd2() {
  int i, r[N], x[N], y[N], rem = N % 4;
  mmevec a1, a2, a3;
  for (i = 0; i < N-rem; i+=4) {
    a1.vec = _mm_set_epi32(x[i], x[i+1], x[i+2], x[i+3]);
    a2.vec = _mm_set_epi32(y[i], y[i+1], y[i+2], y[i+3]);
    a3.vec = _mm_add_epi32(a1.vec, a2.vec);
    r[i] = a3.val[0];
    r[i+1] = a3.val[1];
    r[i+2] = a3.val[2];
    r[i+3] = a3.val[3];
  }
  for (i = N-rem; i < N; i++) {
    r[i] = x[i] + y[i];
  }
}
```

Inlining functions!

Function calls have an overhead (i.e. jump to function address, create local variables on the stack, return from function) that should be avoided for small functions that are called often.

Solution: Copy the code to all locations or use `inline` keyword, i.e. return the dimension of a vector in C++

```
template <typename Number>
inline unsigned int Vector<Number>::size () const
{
  return dim;
}
```

Note that the `inline` keyword is only a suggestion to the compiler.

Inlining may also make debugging more difficult as the function itself may not exist any more “in its own right”. Calling the function from other object files is then also not possible (use `-fkeep-inline-functions`).

Inlining functions: Compiler Options

Related compiled options

```
-finline-small-functions
          Integrate functions into their callers when their body is smaller
          than expected function call code (so overall size of program gets
          smaller).  The compiler heuristically
          decides which functions are simple enough to be worth integrating in this
          Enabled at level -O2.
-finline-functions
          Integrate all simple functions into their callers.  The compiler
          heuristically decides which functions are simple enough to be worth
          integrating in this way.
          If all calls to a given function are integrated, and the function
          is declared "static", then the function is normally not output as
          assembler code in its own right.
          Enabled at level -O3.
```

Inlining functions: More Compiler Options

```
-finline-functions-called-once
          Consider all "static" functions called once for inlining into their
          caller even if they are not marked "inline".  If a call to a given
          function is integrated, then the function
          is not output as assembler code in its own right.
-fkeep-inline-functions
          In C, emit "static" functions that are declared "inline" into the
          object file, even if the function has been inlined into all of its
          callers. This switch does not affect functions using the "extern
          inline" extension in GNU C89.
          In C++, emit any and all inline functions into the object file.
--param max-inline-recursive-depth-auto
              Specifies maximum recursion depth used by the recursive inlining.
              For functions declared inline
              --parammax-inline-recursive-depth
              is taken into account. For function not declared inline,
              recursive inlining happens only when
              -finline-functions (included in -O3) is enabled and
              --param max-inline-recursive-depth-auto is used.  The default value
```

Explicit Loop Unrolling (deal.II code)

Skalar product $\langle v, w \rangle$ implemented in C++ as

```
const_iterator ptr  = begin(),
               vptr = v.begin(),
               eptr = ptr + (dim/4)*4;
while (ptr!=eptr) {
    sum0 += (*ptr++ * *vptr++);
    sum1 += (*ptr++ * *vptr++);
    sum2 += (*ptr++ * *vptr++);
    sum3 += (*ptr++ * *vptr++);
}
                                    // add up remaining elements
while (ptr != end())
  sum0 += *ptr++ * *vptr++;

return sum0+sum1+sum2+sum3;
```

Explicit Loop Unrolling (2)

Loop unrolling may help the compiler to generate efficient code.

Explicit and/or automatic loop unrolling (also using `-funroll-loops` will produce larger code that may in return run slower, e.g. it may not fit into instruction cache.

The GNU compiler documentation warns that

```
-funroll-all-loops
  Unroll all loops, even if their number of iterations is uncertain when the
  loop is entered. This usually makes programs run more slowly.
  -funroll-all-loops implies the same options as -funroll-loops,
```

Whether explicit loop unrolling is beneficial has to be tested for the specific combination algorithm, compiler and processor. Due to more advanced compiler optimization as well as advanced hardware it is less important than it used to be.

Explicit Loop Unrolling and Inlining (deal.II code)

Vector-2-Norm $\sqrt{\langle v, v \rangle}$ implemented in C++ as

```
const_iterator ptr  = begin(),
               eptr = ptr + (dim/4)*4;
while (ptr!=eptr) {
    sum0 += ::sqr(*ptr++);
    sum1 += ::sqr(*ptr++);
    sum2 += ::sqr(*ptr++);
    sum3 += ::sqr(*ptr++);
}

while (ptr != end()) // add up remaining elements
  sum0 += ::sqr(*ptr++);

return sum0+sum1+sum2+sum3;
```

where `sqr()` is an inline function.

```
template <typename Number>
static inline Number sqr (const Number x) {
  return x*x;
};
```

Aliasing in C/C++!

Consider the following function

```
void scale_and_shift(double *a, double *b, double s, int n)
{
  for(int i=1;i<n;i++)
    {
      a[i]=s*b[i-1];
    }
}
```

In the standard case, i.e.,, if the memory of a and b do not overlap this loop can be vectorized. However, if that is not the case, e.g. $a == b$ then the loop exhibits a dependency `a[i]=s*a[i-1]` which does not allow vectorization.

This is called aliasing since the same memory location may be accessed using the two aliases a and b.

The compiler can not know in what way this function will be called and has to be conservative. The loop can be unrolled by hand and a remark should be included in the documentation that "the memory of a and b should not overlap".

Aliasing in C/C++!

Using the options

`-fno-fnalias (Intel)`
`-fargument-noalias (GNU)`

the compiler can be told that aliasing will not occur. If it still does, die code will give incorrect results.

C++-Optimizations: Avoiding Temporary Variables

Programming using a high level of abstraction often comes with a performance penalty because of

- indirect access, e.g., in use of late binding, i.e. virtual function tables
- unnecessary temporary objects
- overloading of `operator[]` that may prevent SIMD vectorization.

Reminder: C++-Templates

```
template<class T> class Vektor {
  T* v;
  int sz;
public:
  Vektor();
  T& getelement(int i) {
    if(i>=0 && i<sz)
       return v[i]
    else
       std::cout<<"Vektor error: Out of bounds:"<<i<<std::endl;
  }
...
};
void main()
{
  Vektor<int> v;
  Vektor<std::string> s;
  Vektor<int *> p;
...
}
```

Reminder: C++ Standard Template Library

```
#include <vector>
using std;

vector<double> x[10];
x[0]=1;
cout << x[1]; // Achtung undefiniert:
              // auch vector<> wird nicht automatisch initialisiert
cout << x.at(1);    // wirft eine Ausnahme

x push_back(3.141); // Ein Element x[10]=3.141 anfuegen
cout << x.size();   // Die Laenge von x ausgeben (11)
x.pop_back();       // Feld wieder um eins verkuerzen
x.clear();          // Vektor leeren; Laenge auf 0

x.resize(20);       // Die Laenge des Vektors auf 20 setzen
                    // Achtung, wie push_back evtl. eine langsame Operation
                    // da intern umkopiert werden muss
x.reserve(20);      // Reserviert internen Platz fuer Elemente
vector<double> y;
y=x;                // kopiert alle Elemente
```

Use Templates instead of Inheritance (if possible)

Classes are runtime polymorphism.

`ParsingAlgorithm` is an abstract base class from which parsers for certain data types are inherited. The ParsingAlgorithm is then used (via a pointer) by a class TextParser to parse a text.

```
class TextParser
{
   //...
   public:
      void Parse(ParsingAlgorithm *p);
   //...
}

void TextParser::Parse(ParsingAlgorithm* p)
{
   value = p->Parse(text);
}
```

Every call `p->Parse(text)` involves dereferencing `p` and then looking up the member function `Parse` in the virtual function table.

Use Templates instead of Inheritance (2)

Use compile time binding instead of runtime binding if possible.

Templates can be seen as compile time polymorphism or as a code generation tool. Here, the Parser is parametrized using the template parameter `Parser`.

```
template <class Parser>
class TextParser
{    //...
   public:
      void Parse()
      {
           value = my_parser.Parse(text);
      }

   private:
      Parser my_parser;
   //...
}
```

Here, no pointer is dereferenced. Templates were invented to make C++ as fast as FORTRAN always has always has been.

Avoiding Temporary Objects

Example: Use `operator +=` instead of `operator +`.

The possibility of operator overloading in C++ may mislead to use mathematical notation when programming numerical algorithms.

```
struct Vec2D
{
    union {
        struct {
            double x;
            double y;
        };
        double val[2];
    };

    Vec2D(){};
    Vec2D(double xx, double yy):x(xx),y(yy){ }
//...
```

Avoiding Temporary Objects (2)

```
//...
    Vec2D operator + (const Vec2D &p) const { return Vec2D(x+p.x,y+p.y); }
     // uses temporary
    Vec2D operator * (const double s) const { return Vec2D(s*x,s*y); } // nur p*2 g
     // uses temporary

    Vec2D operator += (const Vec2D &p) { x+=p.x; y+=p.y; return *this; }
    Vec2D operator *= (const Vec2D &p) { x*=p.x; y*=p.y; return *this; }

    std::ostream & Write(std::ostream &o=std::cout) const
        { return o<<x<<" "<<y<<" "; }
}; // End of class declaration
```

Avoiding Temporary Objects (3)

A reference `&p` has been used for the parameter `p`. This avoids copying the object (of course in printing speed it is not relevant). The reference has been declared `const` since it was not introduced to modify `p`.

Note: For a very simple object like this (containing only two doubles) a copy may be as fast (or faster) than a pointer or reference.

```
std::ostream & operator << (std::ostream & o, const Vec2D &p)
{
    o.precision(7);
    p.Write(o);
    return o;
}

inline bool Vec2D::operator < (const Vec2D &p) const
{
    return std::lexicographical_compare(val,val+2,p.val,p.val+2);
}
```

Avoiding Temporary Objects (2)

```
Vec2D a(0,0),b(1,1),c(2,3);
c=2*a+b
```

Uses temporay objects for $(2 * a)$ and $(2 * a + b)$ whereas

```
Vec2D a(0,0),b(1,1),c(2,3);
c=a; c*=2; c+=b;
```

avoid temporary objects.

Lazy Construction

Allocating and deallocating memory takes time, especially if it is initialized.

```
std::vector<double> v(1000);
switch(i)
{
  case 1: /* */ break;
  case 2: /* */ break;
  /* */
  case 10: std::copy(w.begin(),w.end(),v.begin()); break;
  default: /* */ break;
};
```

Will allocate memory, initialize it using 0 and then values will be copied.

But of course remember: When programming an algorithm correctness is first, optimization is last! Initialize all variables, use `v.at(i)` everywhere instead of `v[i]`, etc.

Lazy Construction (2)

```
switch(i)
{
  case 1: /* */ break;
  case 2: /* */ break;
  /* */
  case 10:
  {
    vector<double> v(w);  // allocates and initializes using copy constructor
  }
  default: /* */ break;
};
```

Early/Static Construction

If the length of a vector is known and if it is known that it will be used often then an early or even static construction may be the fastest option although it may be unelegant.

This is especially the case in `Matlab` which is lousy at dynamically growing vectors or matrices.

```
{
  static vector<double> v(1000);
  std::copy(w.begin(),w.end(),v.begin()); break;
}
```

The Curiously Recurring Template Pattern (CRTP)

The so-called CRTP is a technique to implement compile-time polymorphism

```
template<class Derived>
class Base
{
    // methods within Base can use template to access members of Derived
};
class Derived : public Base<Derived>
{
    // ...
};
```

The code of the base class is instantiated at compile time of the `Derived` class and therefore is able to use/call functions of the `Derived` class without the use of virtual functions!

The function which is called is clearly determined at compile time and not at runtime (as in polymorphism) but in many cases this is enough.

See also `http://en.wikipedia.org/wiki/Curiously_recurring_template_pattern`

Expression Templates

A technique which allows to use the easily readible expression as

$$c = s * a + a * b$$

without performance penalty. Here, $a * b$ means the component-wise multiplication.

The technique will lead to a call like

```
x = Vector< VecAdd<double, vector<double>, vector<double>> >(
    VecAdd<double, vector<double>, vector<double>>(
    Vector< double, VecScale<double, double, vector<double>> >(
    VecScale<double, double, vector<double>>(c, x)),
    Vector< double, VecMul<double, vector<double>, vector<double>> >)(
    VecMul<double, vector<double>, vector<double>>(x, y))(x, y))
)
```

Note that the code will NOT be faster than good C or Fortran code. It rather serves to eliminate performance penalties inherent in the C++ language.

The code will clearly be more difficult to debug for correctness and for performance!

Dynamically Growing Arrays

C++ STL vectors can be grown dynamically efficiently, i.e., in almost linear time, i.e.,

```
vector<double> v;
for(int i=0;i<100000;i++)
   v.push_back(i);
```

will be efficient.

Preallocating or reserving memory (i.e. `v.reserve(100000)`) will usually still make the code faster.

Remark: In Matab

```
for i=1:100000
   v(i)=i;
```

will be slow (even quadratic time in older versions of Matlab) but

```
v=zeros(1,100000);
for i=1:100000
   v(i)=i;
```

will be fast.

Use of Iterators Instead of Overloaded Operator [ ]

Although, in principle, this code could be vectorized by compilers currently may not.

```
template<class T> T sprod(const vector<T> a, const vector<T> b)
 {
   T res=T(0);
   int s=a.size();    // to avoid calls to size() (Compiler may or may not inline)
   for(int i=0;i<s;++i)    // ++i (for objects) in general faster than i++
     {
       res+=a[i]*b[i];
     }
   return res;
 }
```

Calls `const T & vector<T> :: operator []` for `a` and `b`. Although, in principle this could be vectorized after inlining `operator []` it is not.

Why $++i$ is (or can be) faster

hier

Use of Iterators Instead of Overloaded Operator[] (2)

Compilers do not have problems with SIMD vectorization in this version.

```
template<class T> T sprod(const vector<T> a, const vector<T> b)
 {
   typename vector<T>::const iterator a_it=a.begin(), b_it=b.begin();
   typename vector<T>::const iterator a_end=a.end();
   T res=T(0);
   for(;a_it!=a_end;++a_it,++b_it)    // ++i (for objects) in general faster than i
     {
       res+=(*a)*(*b);
     }
   return res;
 }
```

Performance critical loops can be written in C or Fortran and be called from C++.

Row major vs column major order

Fortran stores an two dimensional array column-wise (=column major order) whereas C/C++ use row-wise storage (=row major order).

C/C++ uses row major order, i.e

```
double a[m][n];
for(int i=0;i<m;++i)
   for(int j=0;j<n;++j)
      a[i][j]=i+j;
```

will access the memory using stride 1 which is good for locality.

Since Fortran uses column major order, in

```
do i=1,m
  do j=1,n
     a[i][j]=i*j;
  enddo
enddo
```

will access the memory using stride `n`, i.e. the memory adress is increased by `n*sizeof(double)`. Interchanging `i` and `j` helps.

Example Templates

```
#include <iostream>      /* Point_templates */
using namespace std;

template <class T>
class Point
{
   public:
   T x;
   T y;
   Point(){};
   Point(T a,T b){ x=a; y=b; }
   template <class S>
   friend ostream & operator << (ostream & o, const Point<S> &p);
};

template <class S>
ostream & operator << (ostream & o, const Point<S> &p)
{
    o<<p.x<<" "<<p.y;
}
```

Example Templates

```
int main()
{
   Point<int> p(1,1);
   cout<<p<<"\n";
   Point<double> q(1.0,3.14);
   cout<<q<<"\n";
}

/* Output:
$> ./a.out
1 1
1 3.14
*/
```