General “sequential” code optimization

Georg Hager (RRZE)
Reinhold Bader (LRZ)
General optimization:
Outline

- “Common sense” optimizations
- Characterization of memory hierarchies
  - Bandwidth-based performance modeling
- Loop optimizations to reduce code balance
  - Fusion
  - Unrolling
  - blocking
- Profiling
- Some words about C++
“Common sense” optimizations:
A Monte Carlo spin code
- 3-D cubic lattice
- One variable (“spin”) per grid point with values $+1$ or $-1$

- Next-neighbour interaction terms
- Code chooses spins randomly and flips them as required by MC algorithm
Optimization of a Spin System Simulation

- **Systems under consideration**
  - $50 \cdot 50 \cdot 50 = 125000$ lattice sites
  - $2^{125000}$ different configurations
  - Computer time: $2^{125000} \cdot 1$ ns $\approx 10^{37000}$ years – without MC 😊

- **Memory requirement of original program** $\approx 1$ MByte
Optimization of a Spin System Simulation: 
*Original Code*

- **Program Kernel:**

\[
\begin{align*}
IA &= IZ(KL, KM, KN) \\
IL &= IZ(KLL, KM, KN) \\
IR &= IZ(KLR, KM, KN) \\
IO &= IZ(KL, KMO, KN) \\
IU &= IZ(KL, KMU, KN) \\
IS &= IZ(KL, KM, KNS) \\
IN &= IZ(KL, KM, KNN)
\end{align*}
\]

Load neighbors of a random spin

\[
edelz = iL + iR + iU + iO + iS + iN
\]

calculate magnetic field

```c
C CRITERION FOR FLIPPING THE SPIN

BF = 0.5d0*(1.d0 + tanh(edelz/tt))
if(YHE.LE.BF) then
  iz(kl,km,kn)=1
else
  iz(kl,km,kn)=-1
endif
```

decide about spin orientation
Profiling shows that
- 30% of computing time is spent in the \( \tanh \) function
- Rest is spent in the line calculating \( \text{edelz} \)

Why?
- \( \tanh \) is expensive by itself
- Compiler fuses spin loads and calculation of \( \text{edelz} \) into a single line

What can we do?
- Try to reduce the "strength" of calculations (here \( \tanh \))
- Try to make the CPU move less data

How do we do it?
- Observation: argument of \( \tanh \) is always integer in the range \(-6..6\) (\( \text{tt} \) is always 1)
- Observation: Spin variables only hold values +1 or -1
Optimization of a Spin System Simulation:
Making it Faster

- Strength reduction by **tabulation** of \( \tanh \) function

\[
BF = 0.5d0 \times (1.d0 + \text{tanh	able(\text{edelz})})
\]

- Performance increases by 30% as table lookup is done with “lightspeed” compared to \( \tanh \) calculation

- By declaring spin variables with **INTEGER*1** instead of **INTEGER*4** the memory requirement is reduced to about \( \frac{1}{4} \)
  
  - Better cache reuse
  - Factor 2–4 in performance depending on platform
  - Why don’t we use just one bit per spin?
    - Bit operations (mask, shift, add) too expensive \( \rightarrow \) no benefit

- Potential for a variety of data access optimizations
  
  - But: choice of spin must be absolutely random!
Optimization of a Spin System Simulation: 
Performance Results

- **Pentium 4 (2.4 GHz)**

![Runtime Graph](image)

- Original code
- Table + 1Byte/Spin
- Table + 1Bit/Spin

(C) 2010-2020 RRZE,LRZ

Code optimization
Optimization of data access
Data accesss: Intel processors and the DRAM gap

- Deep pipeline → Fast clock
- SSE2
- Octo-core AVX
- Dual Core
- Quad Core
- 6C
- 12C
- 24C
- 18C, FMA
- 28C
- 3-channel DDR3 on-chip, ccNUMA

(C) 2010-2020
RRZE,LRZ
What conditions must an application code fulfill in order to make good use of the cache?

- **Cache as a high-speed data store:**
  Code must show good **temporal locality**, i.e. data items should be used multiple times before they get evicted from cache.

- **Cache line structure:**
  Code must show good **spatial locality**, i.e. data items just loaded must be near items to be loaded “soon”.

Implementing temporal and spatial locality by code transformations is the most powerful optimization strategy!

**But:** What is “good enough?”
General remarks on algorithms and data access
Latency and bandwidth in modern computer environments

Avoiding slow data paths is the key to most performance optimizations!
Data access

- Data access is the most frequent performance-limiting factor in HPC

- Cache-based microprocessors feature small, fast caches and large, slow memory
  - “Memory Wall”, “DRAM Gap”
  - Latency can be hidden under certain conditions (prefetch, software pipelining)
  - Bandwidth limit cannot be circumvented
    - Instead, modify the code to avoid the slow data paths

- General guideline: examine “traffic-to-work” ratio (balance) of algorithm to get a hint at possible limitations
  - Examination of performance-critical loops is vital
  - Important metric: (“LOADs/STOREs to FLOPs”)
  - Optimization: lower LDST/FLOP ratio

- … and always remember that stride 1 access is best!
Characterization of Memory Hierarchies

Kernel benchmark: Vector Triad

- Kernel benchmarks:
  - Characterize computer architecture (effective performance of processor & memory hierarchies)
  - Results are easy to interpret and compare
  - Provide upper performance bounds for “real” applications
  - E.g.: STREAM, cachebench

- Popular kernel benchmark: Vector Triad

```
REAL*8 (SIZE): A,B,C,D
DO ITER=1,NITER
  DO i = 1,N
    A(i) = B(i) + C(i) * D(i)
  ENDDO
<OBSCURE - use A,B,C,D>
ENDDO
```

Balance calculation
- Computation:
  2 Flops / i-Iteration
- Data transfer:
  (3+1 LD & 1 ST) / i-Iteration
- Code balance = 2.5 W/F
  (20 bytes/flop)
Code balance \((B_c)\) quantifies the requirements of a loop code:

\[
B_c = \frac{\text{data transfer [bytes]}}{\text{arithmetic ops [flops]}}
\]

Example: Vector triad \(A(:) = B(:) + C(:) \times D(:)\)
- \(B_c = (4+1) \text{ Words} / 2 \text{ Flops} = 20 \text{ bytes/flop (including write allocate)}\)

General rule: Reducing the code balance of a loop by optimizations will do something good for the performance!
- Often used: “Computational Intensity” \(I = 1/B_c\)

For refined analysis, code balance can also be defined for all memory hierarchy levels: \(B^{\text{mem}}_c, B^{L3}_c, B^{L2}_c\)
- Memory transfers are not always the data bottleneck!
For quick comparisons the concept of **machine balance** is useful

\[
B_m = \frac{b_s}{P_{\text{peak}}}
\]

- **Machine Balance** = How much input data can be delivered for each FP operation? (“Memory Gap characterization”)
- Assuming balanced MULT/ADD
- Rough estimate: \(B_m \ll B_c \rightarrow\) strongly memory-bound code
- **Typical values** (main memory):

Intel Haswell 14-core 2.3 GHz
\[B_m = \frac{60 \text{ GB/s}}{(14 \times 2.3 \times 16) \text{ GF/s}} \approx 0.12 \text{ B/F}\]

Intel Sandy Bridge 8-core 2.7 GHz \(\approx 0.23 \text{ B/F}\)

Nvidia Volta V100 \(\approx 0.10 \text{ B/F}\)

Intel Xeon Skylake Platinum \(\approx 0.06 \text{ B/F}\)
Evolution of machine balance

Machine balance $B_{m}^{\text{Mem}}$ [byte/flop] vs. Year

- 1994 to 2018 timeline
- Single core to multicore migration
- Key processors:
  - NHL: Nehalem EP
  - SNB: Sandy Bridge EP
  - HSW: Haswell EP
  - BDW: Broadwell EP
  - SKL: Skylake SP
  - KNC: Knights Corner
  - KNL: Knights Landing
  - NEC SX-8 (4 B/F)
  - NEC SX-9 (2.56 B/F)
  - NEC SX-ACE
  - NEC Tsubasa
  - KNC (MCDRAM)
  - KNL (main mem.)
“Roofline model”

Calculate

\[ l = \min \left( 1, \frac{B_m}{B_c} \right) \]

This is the fraction of peak performance that the loop can achieve

Multiply by \( P_{\text{peak}} \):

\[ P = \min \left( P_{\text{peak}}, \frac{b_s}{B_c} \right) \]

This is a simple model to get an upper limit for loop performance

- \( I > B_m^{-1} \) \( \rightarrow \) core-bound code
- \( I < B_m^{-1} \) \( \rightarrow \) memory-bound code
**Code balance: Examples**

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

\(B_C = 3 \text{ WORDS} / 1 \text{ FLOP} = 24 \text{ B/F}\)

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

\(B_C = 3 \text{ WORDS} / 2 \text{ FLOP} = 12 \text{ B/F}\)

Scalar – can be kept in register

```c
s=0.0;
for(i=0; i<N; ++i) {
    s = s + a[i] * a[i];
}
```

\(B_C = 1 \text{ WORDS} / 2 \text{ FLOP} = 4 \text{ B/F}\)

Scalar – can be kept in register

```c
s=0.0;
for(i=0; i<N; ++i) {
    s = s + a[i] * b[i];
}
```

\(B_C = 2 \text{ WORDS} / 2 \text{ FLOP} = 8 \text{ B/F}\)

Scalar – can be kept in register
Case study: OpenMP-parallel sparse matrix-vector multiplication

A simple (but sometimes not-so-simple) example for bandwidth-bound code and saturation effects in memory
Sparse matrix-vector multiply (spMVM)

- Key ingredient in some matrix diagonalization algorithms
  - Lanczos, Davidson, Jacobi-Davidson

- Store only $N_{nz}$ nonzero elements of matrix and RHS, LHS vectors with $N_r$ (number of matrix rows) entries

- “Sparse”: $N_{nz} \sim N_r$

\[
\begin{array}{ccc}
\text{=} & + & \bullet \\
\end{array}
\]

General case: some indirect addressing required!
**CRS matrix storage scheme**

- **val[]** stores all the nonzeros (length $N_{nz}$)
- **col_idx[]** stores the column index of each nonzero (length $N_{nz}$)
- **row_ptr[]** stores the starting index of each new row in **val[]** (length: $N_r$)

---

### CRS Matrix Storage Scheme

<table>
<thead>
<tr>
<th>row index</th>
<th>column index</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 ...</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1 2 3 4 5 ...</td>
</tr>
<tr>
<td>2</td>
<td>1 2 5</td>
</tr>
<tr>
<td>3</td>
<td>1 5 8</td>
</tr>
<tr>
<td>4</td>
<td>...</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>col_idx[]</th>
<th>val[]</th>
<th>row_ptr[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 5 1 2 5 1 3 4 6 3 4 7 1 2 5 8 ...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 5 8 12 15 19 ...</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Case study: Sparse matrix-vector multiply

- **Strongly memory-bound for large data sets**
  - Streaming, with partially indirect access:

```c
!$OMP parallel do
do i = 1,N_r
   do j = row_ptr(i), row_ptr(i+1) - 1
      c(i) = c(i) + val(j) * b(col_idx(j))
   enddo
endo
dono
$OMP end parallel do
```

- Usually many spMVMs required to solve a problem

- Following slides: Performance data on one 24-core AMD Magny Cours node
Application: Sparse matrix-vector multiply

Strong scaling on one XE6 Magny-Cours node

Case 1: Large matrix

Intrasocket bandwidth bottleneck

Good scaling across NUMA domains
Application: Sparse matrix-vector multiply

Strong scaling on one XE6 Magny-Cours node

Case 1: Large matrix

6 B/F model @ 12 Gbyte/s
Case 2: Medium size

Intrasocket bandwidth bottleneck

Working set fits in aggregate cache

Application: Sparse matrix-vector multiply
Strong scaling on one XE6 Magny-Cours node

Intrasocket bandwidth bottleneck

Working set fits in aggregate cache

mc2depi, 525825x525825, non-zero: 2100225
Application: Sparse matrix-vector multiply

Strong scaling on one Magny-Cours node

Case 3: Small size

No bandwidth bottleneck

Parallelization overhead dominates

CrS-magnycours

rbs480a, 480x480, non-zero: 17088

threads

MFLOPS/s

0 5 10 15 20 25

500 1000 1500 2000

2500 3000 3500 4000 4500

(C) 2010-2020 RRZE, LRZ

Code optimization
Conclusions from the spMVM benchmarks

- If the problem is “large”, bandwidth saturation on the socket is a reality
  - There are “spare cores”
  - Very common performance pattern

- **What to do with spare cores?**
  - Let them idle (saves energy) or use them for other tasks, such as MPI communication

- **Can we predict the saturated performance?**
  - Balance-based performance modeling!

- **Can we predict the saturation point?**
  - … and why is this important? → advanced! (ECM Model)
Code optimization by data access optimization
Data access – general considerations

- **Case 1: O(N)/O(N) Algorithms**
  - O(N) arithmetic operations vs. O(N) data access operations
  - Examples: Scalar product, vector addition, sparse MVM etc.
  - Performance **limited by memory BW** for large N (“memory bound”)
  - Limited optimization potential for single loops
    - …at most a constant factor for multi-loop operations
  - Example: successive vector additions

```plaintext
do i=1,N
  a(i)=b(i)+c(i)
endo

do i=1,N
  z(i)=b(i)+e(i)
endo
```

no optimization potential for either loop

- Loop fusion

```plaintext
B_c = (3+1)/1 W/F
     = 32 B/F
```

```plaintext
do i=1,N
  a(i)=b(i)+c(i)
  z(i)=b(i)+e(i)
endo
```

fusing different loops allows O(N) data reuse from registers

```plaintext
B_c = 7/2 W/F
     = 28 B/F
```
Data access – general guidelines

Case 2: \( O(N^2)/O(N^2) \) algorithms

- Examples: dense matrix-vector multiply, matrix addition, dense matrix transposition etc.
  - Nested loops
- Memory bound for large \( N \)
- Some optimization potential (at most constant factor)
  - Can often enhance code balance by outer loop unrolling
- Example: dense matrix-vector multiplication

```plaintext
do i=1,N
  do j=1,N
    c(i)=c(i)+a(j,i)*b(j)
  enddo
enddo
```

Naïve version loads \( b[] \) \( N \) times!
Data access – general guidelines

- **O(N²)/O(N²) algorithms** cont’d
  - “Unroll & jam” optimization (or “outer loop unrolling”)

```plaintext
do i=1,N
  do j=1,N
    c(i)=c(i)+a(j,i)*b(j)
  enddo
enddo
```

**Unroll**

```plaintext
do i=1,N,2
  do j=1,N
    c(i) =c(i) +a(j,i) *b(j)
  enddo
enddo
```

**Jam**

```plaintext
b(j) can be re-used once from register → save 1 data transfer
```

Lowers $B_c$ from 8 to 6 B/F
Data access – general guidelines

- **O(N^2)/O(N^2) algorithms cont’d**
  - Data access pattern for 2-way unrolled dense MVM:
    - Vector \( \mathbf{b}[\cdot] \) now only loaded \( \frac{N}{2} \) times!
    - Remainder loop handled separately

- Code balance can still be enhanced by more aggressive unrolling (i.e., \( m \)-way instead of 2-way)
- Significant code bloat (try to use compiler directives if possible)
  - Ultimate limit: \( \mathbf{b}[\cdot] \) only loaded once from memory (\( B_c \approx \frac{1}{2} W/F \))
  - Beware: CPU registers are a limited resource
  - Excessive unrolling can cause register spills
- Large cache \( \rightarrow \mathbf{b}[\cdot] \) may be in cache even without unrolling!
**Data access – general guidelines**

- **Case 3: \(O(N^3)/O(N^2)\) algorithms**
  - Most favorable case – computation outweighs data traffic by factor of \(N\)
  - Examples: Dense matrix diagonalization, dense matrix-matrix multiplication
  - Huge optimization potential: proper optimization can render the problem cache-bound if \(N\) is large enough
  - Example: dense matrix-matrix multiplication

```plaintext
do i=1,N
  do j=1,N
    do k=1,N
      c(j,i) = c(j,i) + a(k,i) * b(k,j)
    enddo
  enddo
enddo
```

Core task: dense MVM (\(O(N^2)/O(N^2)\)) → memory bound, but there is another loop level!
Case Study: The Jacobi Smoother

The basics in two dimensions
Roofline performance analysis and modeling
Jacobi Iteration in 2D for the heat equation

**Basics**

- Discretization of partial differential equation \( \Delta \Phi = 0 \)
- Discretize equation on a finite set of equidistant mesh points
- Solve with Dirichlet boundary conditions using Jacobi iteration scheme

\[
B_{i,k} = \frac{1}{4} \times (A_{i-1,k} + A_{i+1,k} + A_{i,k+1} + A_{i,k-1})
\]

Naive analysis: 4 FLOPs, 4 LD, 1 ST
A Jacobi smoother

- Laplace equation in 2D: \( \Delta \Phi = 0 \)

- **Solve** with Dirichlet boundary conditions using Jacobi iteration scheme:

```plaintext
double precision, dimension(0:imax+1,0:kmax+1,0:1) :: phi
integer :: t0,t1
t0 = 0 ; t1 = 1
do it = 1,imax    ! choose suitable number of sweeps
  do k = 1,kmax
    do i = 1,imax
      ! four flops, one store, four loads
      phi(i,k,t1) = ( phi(i+1,k,t0) + phi(i-1,k,t0)
                      + phi(i,k+1,t0) + phi(i,k-1,t0) ) * 0.25
    enddo
  enddo
  ! swap arrays
  i = t0 ; t0=t1 ; t1=i
enddo
```

Reuse when computing:

\[ \phi(i+2,k,t1) \]

Naive balance (incl. write allocate):

\[ \phi(:, :, t0) : 3 \text{ reads} + \]
\[ \phi(:, :, t1) : 1 \text{ write} + 1 \text{ write-allocate} \]

\[ B_C = 5 \text{ W} / 4 \text{ FLOPs} = 10 \text{ B/F} \]
Performance metrics: 2D stencil

- **Alternative implementation** ("Macho FLOP version")

  
  ```fortran
  do k = 1,kmax
    do i = 1,imax
      phi(i,k,t1) = 0.25 * phi(i+1,k,t0) + 0.25 * phi(i-1,k,t0)
      + 0.25 * phi(i,k+1,t0) + 0.25 * phi(i,k-1,t0)
    enddo
  enddo
  
  MFlops/sec increases by 7/4 but time to solution remains the same

- Better metric (for many iterative stencil schemes): 
  Lattice Site Updates per Second (LUPs/sec)

  2D Jacobi example: Compute LUPs/sec metric via

  \[
  P[LUPs / s] = \frac{it_{\max} \cdot i_{\max} \cdot k_{\max}}{T_{\text{wall}}}
  \]
If cache is too small to hold three rows:

\[
\phi(:,:,t_0) : 2 \text{ reads } + \phi(:,:,t_1) : 1 \text{ write } + 1 \text{ write-allocate}
\]

\[\Rightarrow B_C = \frac{5 \text{ W}}{4 \text{ F}} = 40 \text{ B/LUP}\]

If cache is large enough to hold at least 3 rows (shaded region): Each \(\phi(:,:,t_0)\) is loaded once from main memory and reused 3 times from cache:

\[
\phi(:,:,t_0) : 1 \text{ read} + \phi(:,:,t_1) : 1 \text{ write } + 1 \text{ write-allocate}
\]

\[\Rightarrow B_C = \frac{3 \text{ W}}{4 \text{ F}} = 24 \text{ B/LUP}\]
3D sweep:

\[
\begin{align*}
\text{do } & k=1,k_{\text{max}} \\
& \text{do } j=1,j_{\text{max}} \\
& \text{do } i=1,i_{\text{max}} \\
\phi(i,j,k,t_1) &= \frac{1}{6} \cdot (\phi(i-1,j,k,t_0)+\phi(i+1,j,k,t_0) &
+ \phi(i,j-1,k,t_0)+\phi(i,j+1,k,t_0) &
+ \phi(i,j,k-1,t_0)+\phi(i,j,k+1,t_0))
\end{align*}
\]

Best case balance: 1 read \(\phi(i,j,k+1,t_0)\)
1 write + 1 write alloc. \(\phi(i,j,k,t_1)\)
6 flops = 1 LUP

\(\rightarrow B_C = 0.5 \text{ W/F (24 bytes/LUP)}\)

No 3-layer condition but 3 rows fit: \(B_C = \frac{5}{6} \text{ W/F (40 bytes/LUP)}\)
Worst case (3 rows do not fit): \(B_C = \frac{7}{6} \text{ W/F (56 bytes/LUP)}\)
3D Jacobi solver

Performance of vanilla code on one Interlagos chip (8 cores)

Problem size: $N^3$

- Roofline inappropriate for unsaturated case
- 3 layers of source array drop out of L2 cache
Conclusions from the Jacobi example

- **We have made sense of the memory-bound performance vs. problem size**
  - “Layer conditions” lead to predictions of code balance
  - Achievable memory bandwidth is input parameter

- **The model works only if the bandwidth is “saturated”**
  - In-cache modeling is more involved

- **Optimization == reducing the code balance by code transformations**
  - See below
Data access optimizations for the Jacobi smoother
Remember the 3D Jacobi solver on Interlagos?

3 layers of source array drop out of L2 cache

→ avoid through spatial blocking!
Layer condition broken because \( imax \) too large

Idea: Rearrange updates so inner dimension becomes smaller; update blocks in turn

Boundary overhead: Often insignificant, but keep an eye on it!
Jacobi iteration (3D): Spatial blocking

**Implementation:**

```fortran
do ioffset=1,imax,iblock
  do joffset=1,jmax,jblock
    do k=1,kmax
      do j=joffset, min(jmax,joffset+jblock-1)
        do i=ioffset, min(imax,ioffset+iblock-1)
          phi(i,j,k,t1) = ( phi(i-1,j,k,t0)+phi(i+1,j,k,t0) + ... + phi(i,j,k-1,t0)+phi(i,j,k+1,t0) )/6.d0
        enddo
      enddo
    enddo
  enddo
enddo
```

**Guidelines:**

- If possible, keep inner block size (loop length) long
- Block sizes \( (iblock, jblock) \) must be small enough to fulfill “layer condition”
- Cache size is a hard limit!
- Blocking loops may have some impact on memory page placement

**Layer condition:**

\[
3 \cdot iblock \cdot jblock \cdot 8 \text{ bytes} \leq \text{CACHE\_SIZE\_PER\_THREAD}/2
\]
Profiling
Key question

- How do I know where my code spends most of its time?
  - This is called “profiling”
  - Many (free and commercial) tools exist
  - C++ code is notoriously hard to profile
    - Overloaded operators, tiny methods
Profiling with gprof

- Basic profiling tool under Linux: \texttt{gprof}

- Compiling for a profiling run

  \texttt{icpc -pg .....}

- After running the binary, a file \texttt{gmon.out} is written to the current directory

- Human readable output:

  \texttt{gprof a.out}

- Inlining should be disabled for profiling
  - But then the executed code isn’t what it should be...
Profiling with gprof
Example

Example with wrapped `double` class:

class D {
    double d;
public:
    D(double _d=0) : d(_d) {}
    D operator+(const D& o) {
        D r;
        r.d = d+o.d;
        return r;
    }
    operator double() {
        return d;
    }
};

const int n=10000000;
D a[n], b[n];
D sum;

for(int i=0; i<n; ++i)
    a[i] = b[i] = 1.5;

double s = timestamp();
for(int k=0; k<10; ++k) {
    for(int i=0; i<n; ++i)
        sum = sum + a[i] + b[i];
}

Main program:
Profiling with gprof
Example profiler output

- icpc -O3 -pg perf.cc

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self</th>
<th>self</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>seconds</td>
<td>seconds</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101.01</td>
<td>0.41</td>
<td>0.41</td>
<td></td>
</tr>
</tbody>
</table>

- icpc -O3 -fno-inline -pg perf.cc

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self</th>
<th>self</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>seconds</td>
<td>seconds</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>46.44</td>
<td>0.59</td>
<td>0.59</td>
<td>200000000</td>
</tr>
<tr>
<td>29.63</td>
<td>0.96</td>
<td>0.37</td>
<td>240000001</td>
</tr>
<tr>
<td>24.82</td>
<td>1.27</td>
<td>0.31</td>
<td></td>
</tr>
</tbody>
</table>

- But where did the time actually go?
  - Butterfly (callgraph) profile also available
  - Real problem also with use of libraries (STL!)
  - Sometimes you have to roll your own little profiler (good idea!)
Manual profiling with a timer class

- Measuring walltime on UNIX (-like) systems
  - Stay away from CPU time – it’s evil!
  - Use `gettimeofday()` to measure walltime:

```c
#include <sys/time.h>

double timestamp(void){
    struct timeval tp;
    gettimeofday(&tp, NULL);
    return((double)(tp.tv_sec + tp.tv_usec/1000000.0));
}
```

- Code available in the exercise templates
Consequences from the saturation pattern for profiling

- Clearly distinguish between “saturating” and “scalable” performance on the chip level.

![Graphs showing performance vs. cores for saturating and scalable types.](image-url)
Consequences from the saturation pattern for profiling

- There is **no clear bottleneck for single-core** execution
- Code profile for single thread ≠ code profile for multiple threads
  - → Single-threaded profiling may be misleading

![Diagram showing saturation and scalable parts](image)
Some words about efficient C++
Writing efficient C++ code: Key questions

- **C++ code is notoriously hard to profile**
  - Overloaded operators, tiny methods

- **What are the most common performance problems with C++?**
  - Too much trust in the compiler’s ability to optimize code
  - Too many useless constructions and destructions
  - Too many temporary objects
  - Overloaded operator overuse
C++ performance pitfalls in a nutshell

- Constructing and destroying objects
  - Be aware that derived class ctors/dtors call their base class variants “recursively”
  - Try to initialize member objects right in the initialization list rather than the body:

```
X::X(point _p) : p(_p) {}
```

is better than

```
X::X(point _p) { p = _p; }
```

- Do not instantiate objects you don’t need:

```
if(a<b) {
    X x(a); // better here than outside
    x.someComplexStuff();
}
```
C++ performance pitfalls in a nutshell

- **Virtual functions**
  - Virtual functions cannot be inlined
  - If there are many small, often-called virtual functions this may have a performance impact

- **Temporary objects**
  - Operator overloading tends to generate many temporaries

```c++
a = b + c + d; // 2 temp objects
```

```c++
a = b; // no temp objects
a += c;
a += d;
```
C++ performance pitfalls in a nutshell

- **Unnecessary copies**
  - Returning an object and passing an object to a function require copies
    - Copying standard data types is most often harmless
  - Try to work with references/pointers where feasible
  - Use *reference counting* if copies cannot be avoided

- **Do not trust the compiler to do “the right thing”**
  - `vector<>::operator[]` is usually slower than plain array access
    - SIMD vectorization is main issue
  - Use *iterators* where possible
  - Tell the compiler that arrays do not overlap: `-fno-alias` and friends
Further thoughts…

“But Expression Templates will do the trick, right?”

No, they will not, at least not in the traditional sense:
DOI: 10.1137/110830125

“But if I write inline in front of all methods/functions, the compiler will optimize perfectly, right?”

No. Inlining will only reduce the most severe overheads. In fact, putting tight, performance-critical loops into separate C compilation units will often lead to better performance. (“Line Update Kernel” principle)

Bottom line: If the code behaves in accordance with your performance model, you’re done.
The Golden Rule of Performance

Performance \cdot Flexibility = \text{const.}
Summary of scalar optimizations

- **#1 optimization in HPC:** Avoid slow data paths!
  - #0 optimization: reduce the amount of work…
  - #2 optimization: speed up in-core code (pipelining, SIMD,…)

- **Machine/code balance performance model** gives estimate for performance of “data streaming” loops

- **Typical balance-reducing optimizations** (avoiding slow data paths)
  - Loop fusion
  - Outer loop unrolling (unroll-and-jam)
  - Loop blocking

- Be aware of **saturation patterns** on the chip level!
- **Estimate potential gain** before embarking on complex code transformations!
- **The compiler will NOT fix it** for you (mostly)!
This course covers performance engineering approaches on the compute node level. Even application developers who are fluent in OpenMP and MPI often lack a good grasp of how much performance could at best be achieved by their code. This is because parallelism takes us only half the way to good performance. Even worse, slow serial code tends to scale very well, hiding the fact that resources are wasted. This course conveys the required knowledge to develop a thorough understanding of the interactions between software and hardware. This process must start at the core, socket, and node level, where the code gets executed that does the actual computational work. We introduce the basic architectural features and bottlenecks of modern processors and compute nodes. Pipelining, SIMD, superscalarity, caches, memory interfaces, ccNUMA, etc., are covered. A cornerstone of node-level performance analysis is the Roofline model, which is introduced in due detail and applied to various examples from computational science. We also show how simple software tools can be used to acquire knowledge about the system, run code in a reproducible way, and validate hypotheses about resource consumption. Finally, once the architectural requirements of a code are understood and correlated with performance measurements, the potential benefit of code changes can often be predicted, replacing hope-for-the-best optimizations by a scientific process.