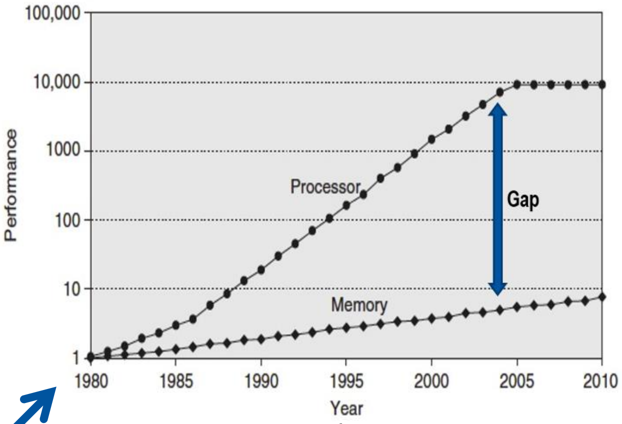
Outline

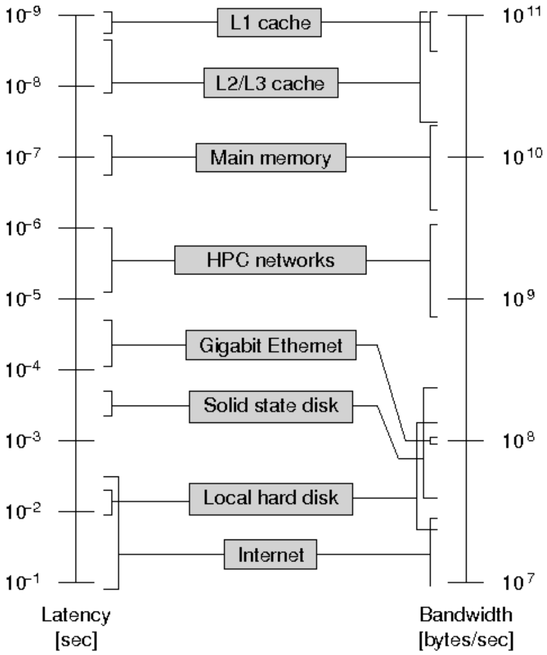
1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. **Data access optimization**
   - → Balance analysis
   - → Algorithm classification and access optimizations
     - → $\mathrm{O}(N)/\mathrm{O}(N)$
     - → $O(N^2)/\mathrm{O}(N^2)$
     - → $O(N^3)/\mathrm{O}(N^2)$
   - → Case study: sparse matrix-vector multiply
5. Parallel computers
6. Parallelization and optimization strategies
7. Parallel algorithms
8. Distributed-memory programming with MPI
9. Shared-memory programming with OpenMP
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

Motivation

- **Data access: most important performance-limiting factor in HPC**
  - → In science & engineering: often loop-based codes that move large amounts of data in/ out the CPU
    - →Often on-chip resources are underutilized
    - →Limited by slowest data path
- **Recap**
  - → Imbalance between theoretical peak performance & memory bandwidth on modern microprocessors
  - → Bandwidth in typical data paths can be ~3 orders of magnitude away from the function units
- **Optimization aim: reduce traffic over slow data paths**



Source: Hennessy and Patterson, Computer Architecture: A quantitative Approach



Source. Prof. Dr. G. Wellein, Dr. G. Hager, M. Kreutzer Uni Erlangen-Nürnberg

Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. **Data access optimization**
    - → **Balance analysis**
    - → Algorithm classification and access optimizations
        - → $\mathrm{O}(N)/\mathrm{O}(N)$
        - → $O(N^2)/\mathrm{O}(N^2)$
        - → $O(N^3)/\mathrm{O}(N^2)$
    - → Case study: sparse matrix-vector multiply
5. Parallel computers
6. Parallelization and optimization strategies
7. Parallel algorithms
8. Distributed-memory programming with MPI
9. Shared-memory programming with OpenMP
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

Balance metric

- **Before trying to improve the code: check whether resources are already used in the best possible way**
  - → Model for theoretical performance of loop-based codes that are bound by bandwidth limitations

→ **Balance metric**

  - → E.g. machine balance of a processor chip

Machine balance

- **Machine balance $\boldsymbol{B_m}$ of a specific computer for data and memory accesses**
  - → Ratio of possible *memory* bandwidth [GWords/sec] to peak performance [GFlop/sec] *or*
  - → How many operands can be delivered for each FP operation?
  - → $$\boldsymbol{B_m = \frac{b_s}{P_{max}}}$$

b_s: achievable bandwidth over slowest data path [words/s]

P_{max}: peak amount of floating-point operations per seconds [Flop/s]

  - → Assumption: latencies are hidden by prefetching, software pipelining,…

- **Typical values for different processors (main memory)**
  - → AMD Interlagos (2.3 GHz): $\boldsymbol{B_m \approx 0.029}$
  - → Intel Sandy Bridge EP (2.7 GHz): $\boldsymbol{B_m \approx 0.025}$
  - → NEC SX9 (vectorcomputer): $\boldsymbol{B_m \approx 0.3}$

Typical machine balance values for different data paths

- ***Memory* bandwidth can also be substituted by other bandwidths**
  - → E.g. to caches or to the network
  - → But, most often used for real memory-bound codes
- **Typical balance values for different data paths**

Data path	B_m [W/F]
Cache	0.5 – 1.0
Main memory	0.01 – 0.5
Interconnect (Infiniband)	0.001 – 0.002
Interconnect (GBit Ethernet)	0.0001 – 0.0007
Disk	0.0001 – 0.001

Basis: double precision (64 bit), dual-socket nodes (for network and disk values)

- **Growing DRAM gap + more cores → machine balance will further decrease in future**

Machine balance

- **Recap: machine balance $B_m = \frac{b_s}{P_{max}}$**
  → How many operands can be delivered for each FP operation?

- **Reciprocal $B_m{}^{-1} = \frac{1}{B_m} = \frac{P_{max}}{b_s}$**
  → How many FP operations can be performed for each operand?

b_s:	achievable bandwidth over slowest data path [words/s]
P_{max}:	peak amount of floating-point operations per seconds [Flop/s]

Code balance

- **Machine balance B_m: what the hardware can deliver at most**

- **Code balance B_c**
  - → Describes requirements of the code
  - → $B_c = \frac{data\ transfers(LOAD, STORE)}{arithmetic\ operations}\ \frac{[Words]}{[Flop]}$
  - → Computational intensity: $\frac{1}{B_c}$

What the machine can achieve (what we get)

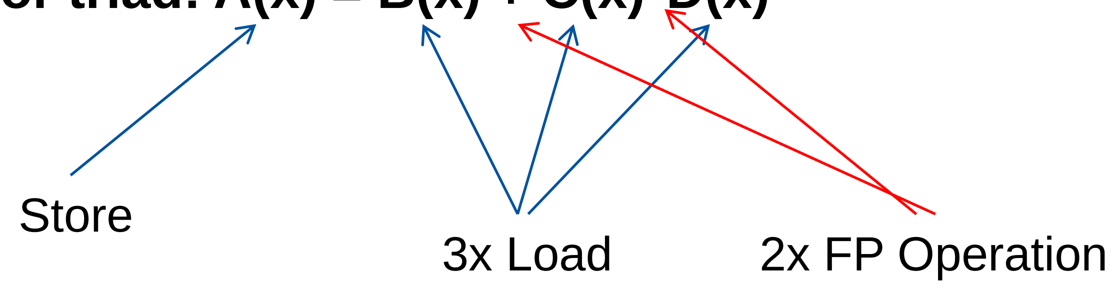
- **Expected max fraction of peak performance: $l = min(1, \frac{B_m}{B_c})$**

  ("lightspeed" of a loop)

What the algorithm requires (what we need)

Code balance – Examples

- **Vector triad: A(x) = B(x) + C(x)*D(x)**



Store

3x Load

2x FP Operation

- **Requires**
  - → 3 Loads (B, C, D)
  - → 1 Store (+1 load to cache)
  - → 2 FP operations

- **Balance metric**
  - → Code balance $\boldsymbol{B_c} = \frac{\mathbf{5}}{\mathbf{2}} = \mathbf{2.5}\,\frac{\boldsymbol{W}}{\boldsymbol{F}}$
  - → $\frac{\boldsymbol{B_m}}{\boldsymbol{B_c}} = \frac{\mathbf{0.029}}{\mathbf{2.5}} = \mathbf{0.012}$
  - → On 2.3 GHz Interlagos. 1.2% of peak performance

Code balance – Examples

type	kernel	DP words	Flops	B_c
COPY	`for(x=0; x < N; ++x)` `a[x] = b[x];`	2 (3)	0	N/A
SCALE	`for(x=0; x < N; ++x)` `a[x] = b[x] * s;`	2 (3)	1	2.0 (3.0)
ADD	`for(x=0; x < N; ++x)` `a[x] = b[x] + c[x];`	3 (4)	1	3.0 (4.0)
TRIAD	`for(x=0; x < N; ++x)` `a[x] = b[x] + c[x]*s;`	3 (4)	2	1.5 (2.0)
Sc. Add	`for(x=0; x < N; ++x)` `s=s+a[x]*a[x];`	1	2	0.5
Sc. Add	`for(x=0; x < N; ++x)` `s=s+a[x]*b[x];`	2	2	1

write allocates

Can be kept in register

Lightspeed for absolute performance

- **Recap: lightspeed of a loop**

  → $\boldsymbol{l = min(1, \frac{B_m}{B_c})}$

  with $\boldsymbol{B_m = \frac{b_s}{P_{max}}}$

P_{max}: CPU's peak performance [Flop/s]
l: lightspeed of loop [-]
b_s: achievable bandwidth over slowest data path [words/s]
B_m: machine balance [words/Flop]
B_c: code balance [words/Flop]

- **Lightspeed for absolute performance [GFlop/s]**

  → $\boldsymbol{P = l \cdot P_{max} = min\left(P_{max}, \frac{b_s}{B_c}\right)}$

Further reading: *Roofline Model*

(Crucial) assumptions for balance metric

- **Loop code makes use of all available arithmetic units (MULT and ADD) in an optimal way**
  - → For other scenarios the input parameter P_{max} has to be adjusted. e.g. $P_{max} \rightarrow cP_{max} | \; c \in (0,1]$
- **Attainable bandwidth of code is known**
  - → It can be determined using simple streaming benchmarks
- **Data transfer and arithmetical operations overlap perfectly**
- **Only the slowest data path is modeled**
  - → Assumption: others are infinitely fast
- **Latency is ignored**

Model is limited

- **Model is often accurate enough to estimate performance of loop codes**

- **In other cases there exist more sophisticated strategies for performance modeling (see Literature)**

- **Recap: STREAM benchmark**
  - → Implements simple synthetic kernel loops
    - →ADD, SCALE, COPY, TRIAD.. (not to be confused with the vector triad)
  - → Measurements reflect the capabilities of machines in terms of memory bandwidth

Stream benchmark for lightspeed measurements

- **Neither STREAM nor the vector triad normally reach theoretical performance levels that are predicted by balance metrics**
  - → Maximum bandwidth is often not available in a bidirectional way (read + write)
    - →A typical proportion from read to write bandwidth is: 2/1
  - → Paths between L1 cache & registers can be unidirectional
  - → Protocol overhead, error correcting chips, large latencies,…
- **But, no real application performs better than the STREAM benchmark**
  - → STREAM measured bandwidth $\boldsymbol{b_s}$ should be used to measure lightspeed $\boldsymbol{l}$ (rather than hardware's theoretical limit)
- **Receiving a significant fraction (e.g. 80%) of the predicted performance based on STREAM results for a code**
  - → Indication for the best utilization of the memory interface

Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. **Data access optimization**
    - → Balance analysis
    - → **Algorithm classification and access optimizations**
        - → $O(N)/O(N)$
        - → $O(N^2)/O(N^2)$
        - → $O(N^3)/O(N^2)$
    - → Case study: sparse matrix-vector multiply
5. Parallel computers
6. Parallelization and optimization strategies
7. Parallel algorithms
8. Distributed-memory programming with MPI
9. Shared-memory programming with OpenMP
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

Algorithm classification

- **Optimization potential can easily be estimated by just investigating basic parameters**
  - → E.g. scaling behavior of data transfers and arithmetic operations

    vs. problem size
  - → Decide whether optimization effort make sense

- **Classifications**
  - → $\mathrm{O}(N)$/ $\mathrm{O}(N)$
  - → $O(N^2)/\mathrm{O}(N^2)$
  - → $O(N^3)/\mathrm{O}(N^2)$

Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. **Data access optimization**
   - → Balance analysis
   - → **Algorithm classification and access optimizations**
     - → $\mathbf{O(N)}$**/** $\mathbf{O(N)}$
     - → $O(N^2)$/$O(N^2)$
     - → $O(N^3)$/$O(N^2)$
   - → Case study: sparse matrix-vector multiply
5. Parallel computers
6. Parallelization and optimization strategies
7. Parallel algorithms
8. Distributed-memory programming with MPI
9. Shared-memory programming with OpenMP
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

O(N)/O(N)

- **Given (N is problem size or loop length)**
  - → O(N) arithmetic operations
  - → O(N) data accesses (loads, stores)

$$\frac{\boldsymbol{arithmetic\ operations}}{\boldsymbol{data\ transfers}} \rightarrow \frac{\boldsymbol{O}(\boldsymbol{N})}{\boldsymbol{O}(\boldsymbol{N})}$$

- **Examples: scalar product, vector addition, sparse matrix-vector multiplication**
- **Performance is limited by memory bandwidth for large N**
  - → "Memory-bound" algorithm
- **Optimization potential usually very limited**
  - → Compiler/ architecture often already produce good code (simple analysis)

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

$$B_c = \frac{6}{2} = 3$$

Loop fusion

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

$$B_c = \frac{5}{2} = 2.5$$

O(N) data reuse!

Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. **Data access optimization**
   - → Balance analysis
   - → **Algorithm classification and access optimizations**
     - → $O(N)$/ $O(N)$
     - → $\boldsymbol{O(N^2)/O(N^2)}$
     - → $O(N^3)/O(N^2)$
   - → Case study: sparse matrix-vector multiply
5. Parallel computers
6. Parallelization and optimization strategies
7. Parallel algorithms
8. Distributed-memory programming with MPI
9. Shared-memory programming with OpenMP
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

O(N^2)/O(N^2)

- **Given (N is problem size or length of outer & inner loop)**
  - → O(N^2) arithmetic operations
  - → O(N^2) data accesses

$$\frac{arithmetic\ operations}{data\ transfers} \rightarrow \frac{O(N^2)}{O(N^2)}$$

- **Examples: dense matrix-vector multiply, matrix transpose, matrix addition**

- **Problems usually memory-bound for large N**
- **Optimization potential: at most a constant factor of improvement**
  - → Code balance can often be enhanced by either outer loop unrolling or spatial blocking

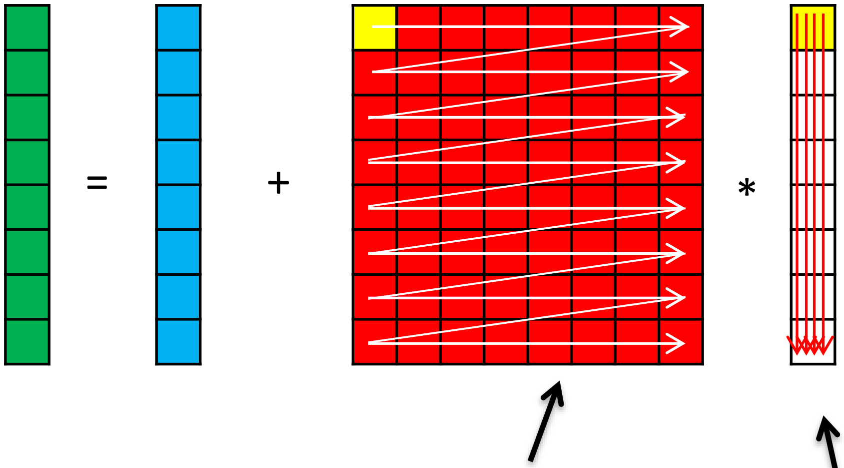
O(N^2)/O(N^2) – Example

- **Dense matrix-vector multiply**

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

2 FP operations 2 loads → $B_c = \frac{2}{2} = 1$

updates of c[i] go to a register (indexed by outer loop)



loaded once

loaded N times!!!

O(N^2)/O(N^2) – Example

Jam ≙ Fuse
In literature, usually "jam" is used.

- **Unroll & Jam:**

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

Unroll

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

e.g., stride 2

replication of inner loop (2 times)

What is missing?

Jam

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

B[j] can be reused once. Saves 1 Load.

O(N^2)/O(N^2) – Example

- **Unroll & Jam:**

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

Unroll

```
for(i=0; i < N; i+=2)
{
   for(j=0; j < N; ++j)
   {
      c[i] = c[i] + a[i][j]*b[j];
   }
   for(j=0; j < N; ++j)
   {
      c[i+1] = c[i+1] + a[i+1][j]*b[j];
   }
}
// remainder loop
for(k=i; k < i + (N % 2); ++k)
{
   for (j=0; j < N; ++j)
   {
      c[k] = c[k] + a[k][j]*b[j];
   }
}
```

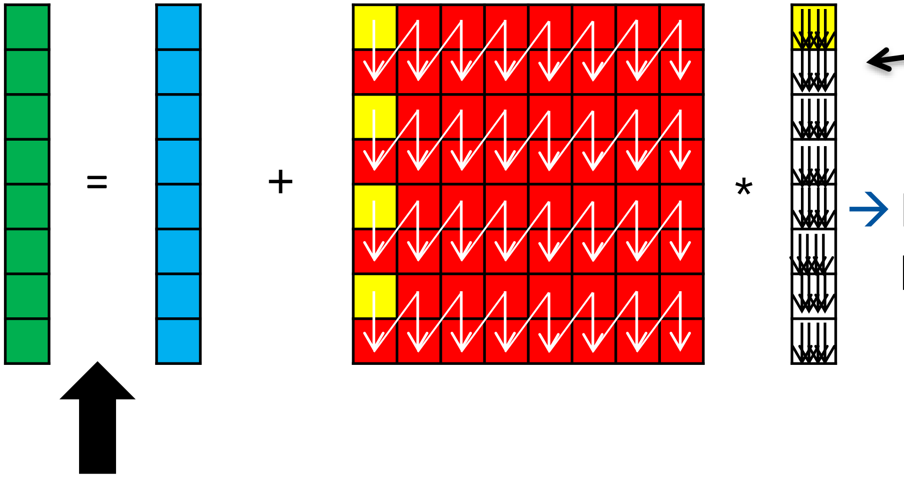
Jam

```
for(i=0; i < N; i+=2)   {
   for(j=0; j < N; ++j)   {
      c[i] =     c[i] +      a[i][j]    *b[j];
      c[i+1] = c[i+1] + a[i+1][j]*b[j];
}  }
// remainder loop
for(k=i; k < i + (N % 2); ++k)   {
   for (j=0; j < N; ++j)   {
      c[k] = c[k] + a[k][j]*b[j];
}  }
```

O(N^2)/O(N^2) – Example

- **Data access pattern of 2-way unroll & jam**

loaded N/2 times



$\rightarrow B_c = \frac{3}{4} < 1$

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

→ Data transfers could be reduced further by unrolling more “aggressively”
  - → m-way instead of 2-way
  - → For m-way unrolling: $B_c = \frac{(m+1)}{(2\mathrm{m})}$ and thus clearly < 1
  - → At max ($n \gg m$, $m \rightarrow \infty$): $B_c = \frac{1}{2}$ can be reached by unrolling
  - → Total memory traffic: $N^2\left(1 + \frac{1}{m}\right) + N$

→ As unrolling implies significant bulks of code, try to use compiler directives if possible

Unroll & jam

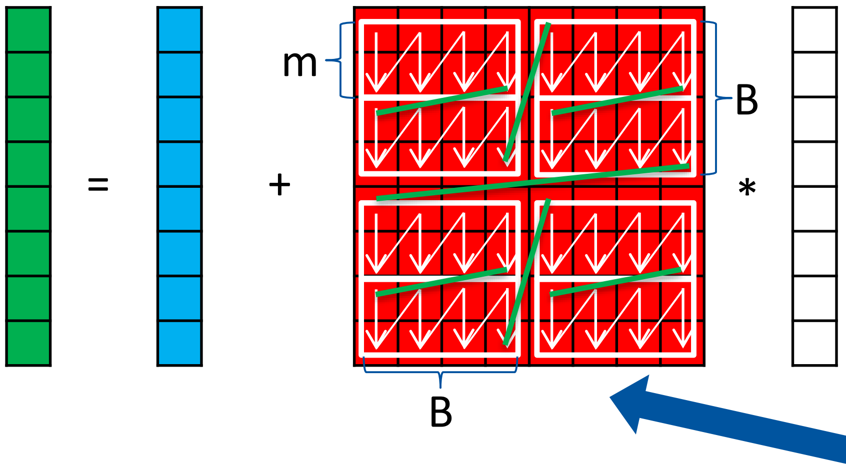
- **Can be done by the compiler at high optimization levels**

- **But: Complex loop bodies may hide important information from the compiler and prevent it from doing so**
  - → Knowledge of the “concept” of unrolling & jamming is important for possibly needed manual optimizations
  - → Directives are the preferred alternative for “manual” unrolling as explicit commands to the compiler to unroll a certain loop

- **Assumption**
  - → CPU has enough registers to avoid register spilling
  - → If m grows to large, it might be required to *spill* register data to cache temporarily, slowing down the computation.

Further optimization: loop blocking

- **Makes better use of caches**
  - → Depending on the blocking factor B
- **Can often not be performed by compilers**
- **Example**
  - → 2D blocking with each blocking factor B
  - → No change in no of needed registers
  - → Better cache line access characteristic



```
for(ii=0; ii < N; ii+=B)
{
  istart = ii; iend = ii+B;
  for(jj=0; jj < N; jj+=B)
  {
    jstart = jj; jend = jj+B;
    for(i=istart; i < iend; i+=m)
    {
      for(j=jstart; j < jend; ++j)
      {
        c[i] =    c[i] +     a[i][j]   *b[j];
        [...]
        c[i+m-1] = c[i+1] + a[i+m-1][j]*b[j];
      }
    }
  }
}
```

blocking

unroll & jam

Further optimization: loop blocking

- **Block factor has to be chosen carefully:**
  - → Mostly experimentally, through benchmarking
  - → Value may depend on actual cache sizes
    - → If blocking for a certain cache, chose factor so that working set size fills not more than half the cache
  - → Can be chosen for different cache levels
    - → No general guideline which cache level the optimization should target

Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. **Data access optimization**
   - → Balance analysis
   - → **Algorithm classification and access optimizations**
     - → $\mathrm{O}(N)$/ $\mathrm{O}(N)$
     - → $O(N^2)$/$\mathrm{O}(N^2)$
     - → $\boldsymbol{O}(\boldsymbol{N}^3)$/$\mathbf{O}(\boldsymbol{N}^2)$
   - → Case study: sparse matrix-vector multiply
5. Parallel computers
6. Parallelization and optimization strategies
7. Parallel algorithms
8. Distributed-memory programming with MPI
9. Shared-memory programming with OpenMP
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

O(N^3)/O(N^2)

- **Given**
  - → O(N^3) arithmetic operations
  - → O(N^2) data accesses

$$\frac{arithmetic\ operations}{data\ transfers} \rightarrow \frac{O(N^3)}{O(N^2)}$$

→ **Most favorable case**
  - → Computation outweighs data traffic by factor of N

- **Examples: dense matrix multiplication, dense matrix diagonalization**

- **Problems have tremendous optimization potential**
  - → Proper optimization can render the problem cache-bound if N is large enough

O(N^3)/O(N^2) – Example

- **Examples for O(N^3)/O(N^2) algorithms are complex**

- **To simplify, we use an O(N^2)/O(N) algorithm to illustrate possible optimization:**

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

b is loaded from memory N times

a is loaded once per inner loop

Thus memory traffic amounts to N(N+1) words.

- **Loop unroll & jam will reduce this to N(N/m + 1)**

  → with m-way unrolling

$O(N^3)$/$O(N^2)$ – Example

- **Blocking the inner loop:** →
  - → With blocking factor B
- **Introduces two effects**
  - → Array b is only loaded once from memory now
    - → as long as factor B is small enough that parts loaded from b fit into the cache and stay there as long as needed
  - → Array a is loaded from memory N/B times instead of once
- **Effective memory traffic: N(N/B+1)**
- **Blocking is the method of choice here, as the blocking factor can be increased to higher values than the unrolling factor**

```
for(jj=0; jj < N; jj+=B)
{
  jstart=jj; jend=jj+B;
  for(i = 0; i < N; ++i )
  {
    for(j=jstart; j < jend; ++j)
    {
      sum += a[i] * b[j];
    }
  }
}
```

Data access patterns – O(N^3)/O(N^2)

- **Algorithms of the O(N^3)/O(N^2) class are generally candidates that can be optimized to performance near the theoretical maximum.**

- **E.g. dense matrix multiplication can reach up to 90% of theoretical peak performance**
  - → with blocking, unrolling and appropriate chosen factors
  - → for N x N matrices if the dimension N is sufficiently high

→ **Highly-optimized libraries for linear algebra are provided, so there is no need to hand code these algorithms**
  - → The BLAS Library (Basic Linear Algebra Subsystem)
  - → But: The principle of blocking and/or unroll + jam can be applied in many real-world codes

Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. **Data access optimization**
   - → Balance analysis
   - → Algorithm classification and access optimizations
     - → $O(N)$/ $O(N)$
     - → $O(N^2)$/$O(N^2)$
     - → $O(N^3)$/$O(N^2)$
   - → **Case study: sparse matrix-vector multiply**
5. Parallel computers
6. Parallelization and optimization strategies
7. Parallel algorithms
8. Distributed-memory programming with MPI
9. Shared-memory programming with OpenMP
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

Sparse matrix-vector multiply

- **Interesting algorithm for the application of the previously discussed strategies**

- **Relevant in many real-world codes**
  - → E.g. matrix diagonalization, iterative solvers for sparse systems

- **"Sparse" matrix**
  - → Number of nonzero entries $\boldsymbol{NNZ}$ grows linearly with number of matrix rows $\boldsymbol{N_r}$
  - → In special storage forms, only the nonzero entries of the matrix are stored in memory (e.g. CRS)

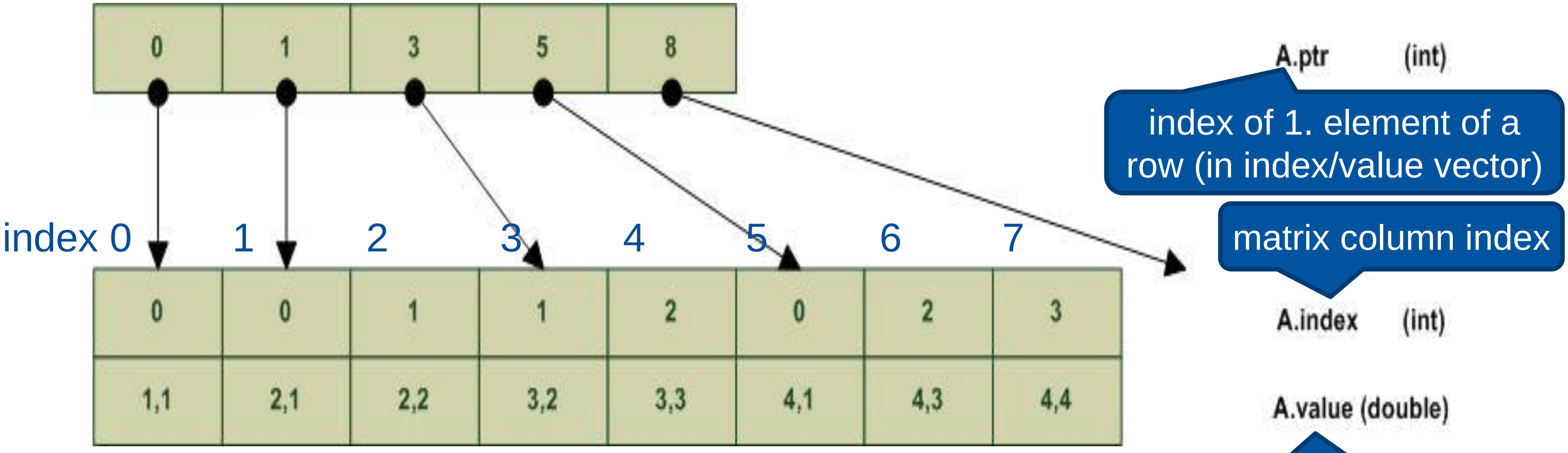
→ **O($\boldsymbol{N_r}$)/O($\boldsymbol{N_r}$) problem**

CRS format

- **CRS = Compress Row Storage**
  - → Just stores nonzero entries
- **Suppose we have a matrix**

$$\begin{array}{c} \text{column} \\ A = \end{array} \begin{array}{cccc} 0 & 1 & 2 & 3 \\ \left(\begin{matrix} 1.1 & 0 & 0 & 0 \\ 2.1 & 2.2 & 0 & 0 \\ 0 & 3.2 & 3.3 & 0 \\ 4.1 & 0 & 4.3 & 4.4 \end{matrix}\right) \end{array}$$

- **The CRS format encodes only the nonzeros in a row based way:**



index of 1. element of a row (in index/value vector)

matrix column index

all nonzero matrix values

Basic implementation

$y = Ax$:

```
for(i=0; i < $N_r$; ++i)
{
  y[i] = 0;
  for(j=A.ptr[i]; j < A.ptr[i+1]; ++j)
  {
     y[i] += A.value[j] * x[A.index[j]];
  }
}
```

- **(Long) outer loop over matrix rows N_r**
- **Access to result vector y[i] is linear**
  - → Only loaded once from memory
- **Nonzeros (A.value[j]) are accesses with stride 1**
- **RHS accessed indirectly (x[A.index[j]])**
  - → Spatial locality cannot be predicted
  - → But generally not a problem if most *NNZ* entries are around diagonal

“Naive” approach: unroll & jam

```
/* Computes: y = A * x;
  * A: sparse matrix, stored in CRS format */
for(i=0; i < N_r; i+=2)  {
  y[i] = 0;
  y[i+1] = 0;
  len1 = A.ptr[i+1] – A.ptr[i];
  len2 = A.ptr[i+2] – A.ptr[i+1];
  minLen = min(len1,len2);
  jstart1 = A.ptr[i];
  jstart2 = A.ptr[i+1];
  for(j=0; j < minLen; ++j)  {
     y[i] += A.value[jstart1+j] * x[A.index[jstart1+j]];
     y[i+1] += A.value[jstart2+j] * x[A.index[jstart2+j]];
  }
  for(; j < len1; ++j)
     y[i] += A.value[jstart1+j] * x[A.index[jstart1+j]];
  for(; j < len2; ++j)
     y[i+1] += A.value[jstart2+j] * x[A.index[jstart2+j]];
}
for(k=i; k < i + (N_r % 2); ++k)  {
  y[k] = 0;
  for (j=A.ptr[k]; j < A.ptr[k+1]; ++j)  {
     y[k] += A.value[j] * x[A.index[j]];
} }
```

unroll factor 2

jammed loop

remainder loops
with respect to
min computation

remainder loop
with respect to
unrolling

What you have learnt

- **What can be modelled with the balance metric?**
  - → How are lightspeed, machine and code balance defined?
  - → How can we get values for lightspeed, machine and code balance?
  - → What are the limitations of this model?
- **How can algorithms be classified depending on the number of arithmetic operations and data transfers?**
  - → Which algorithm types have potential for optimizations?
  - → Which optimizations should be applied?
- **Sparse matrix-vector multiply**
  - → How can sparse matrices be stored and why?
  - → How is the sparse matrix-vector multiply computed and optimized?