NHR FAU

lrz

(Some) performance pitfalls – and remedies

HPC High Performance Computing

General optimization: Outline

- The limits of scalability
- “Common sense” optimizations
- Characterization of memory hierarchies
- Loop optimizations and code balance
- ccNUMA and first-touch initialization

NHR FAU

Irz

Limits of scalability

HPC High Performance Computing

Metrics to quantify the efficiency of parallel computing

- $T(N)$: execution time of some fixed workload with N workers
- How much faster than with a single worker?

  → parallel speedup: $S(N) = \frac{T(1)}{T(N)}$

- How efficiently do those N workers do their work?

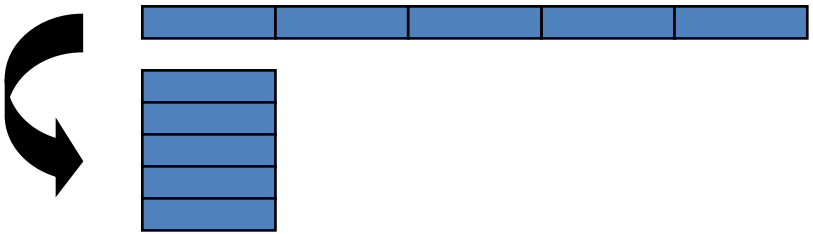
  → parallel efficiency: $\varepsilon(N) = \frac{S(N)}{N}$

- Warning: These metrics are not performance metrics!

Can we predict $S(N)$? Are there limits to it?

Assumptions for basic scalability models

- Scalable hardware: N times the iron can work N times faster
- Work is either fully parallelizable or not at all
- For the time being, assume a constant workload



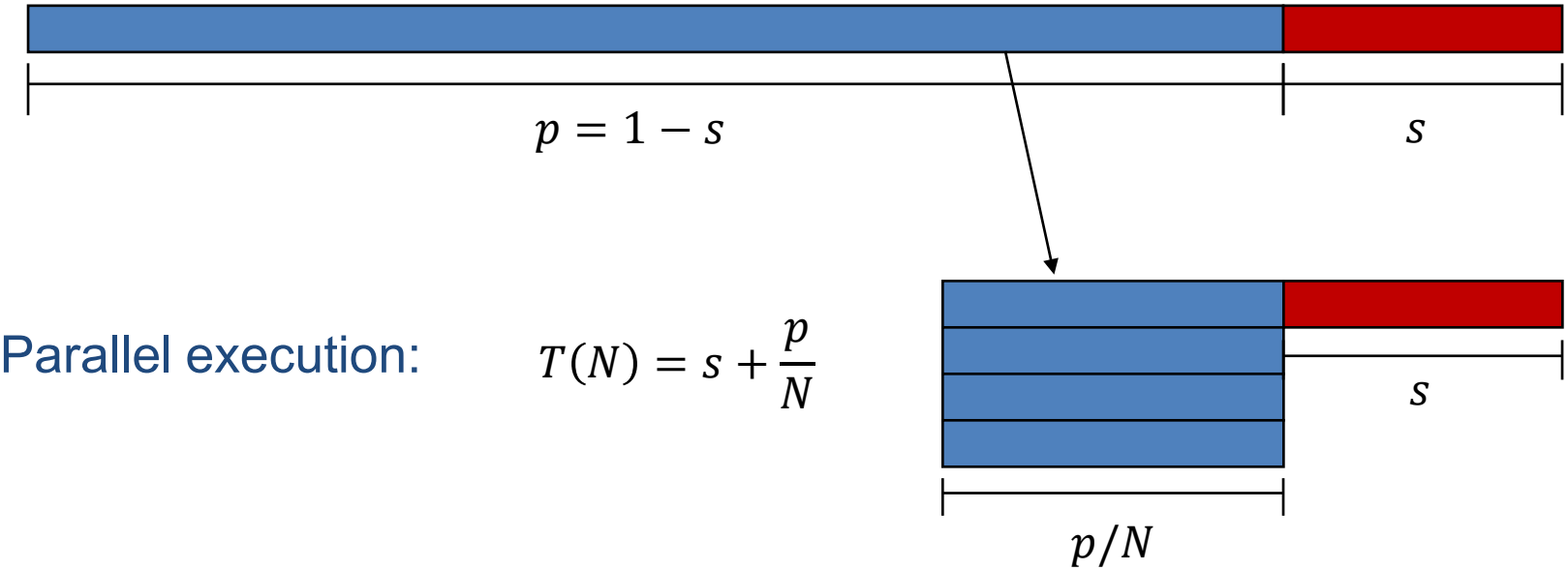
Ideal world:
All work is perfectly parallelizable
$S(N) = N, \qquad \varepsilon = 1$

A simple speedup model for fixed workload

One worker normalized execution time: $T(1) = s + p = 1$

s: runtime of purely serial part

p: runtime of perfectly parallelizable part



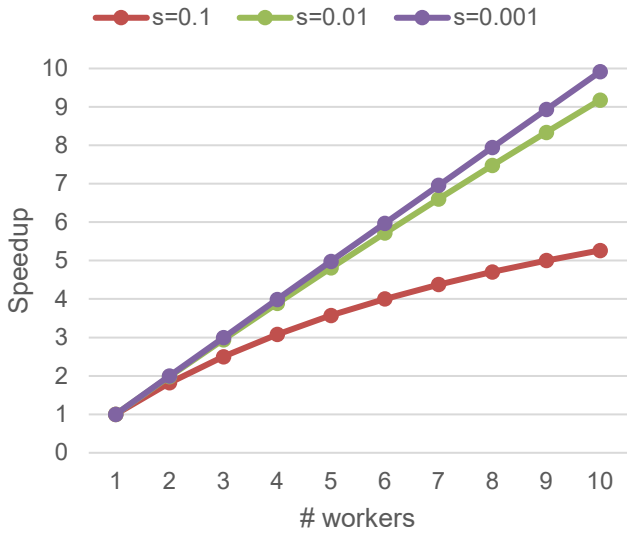
Parallel execution: $T(N) = s + \frac{p}{N}$

Amdahl's Law (1967) – "Strong Scaling"

- Fixed workload speedup with s being the fraction of nonparallelizable work

$$S(N) = \frac{T(1)}{T(N)} = \frac{1}{s + \frac{1-s}{N}}$$

- Parallel efficiency: $\varepsilon(N) = \frac{1}{s(N-1)+1}$



Gene M. Amdahl: *Validity of the single processor approach to achieving large scale computing capabilities*. In Proceedings of the April 18-20, 1967, spring joint computer conference (AFIPS '67 (Spring)). Association for Computing Machinery, New York, NY, USA, 483–485. DOI:10.1145/1465482.1465560

Fundamental limits in Amdahl's Law

- Asymptotic speedup

$$\lim_{N\to\infty} S(N) = \frac{1}{s}$$

- Asymptotic parallel efficiency

$$\lim_{N\to\infty} \varepsilon(N) = 0$$

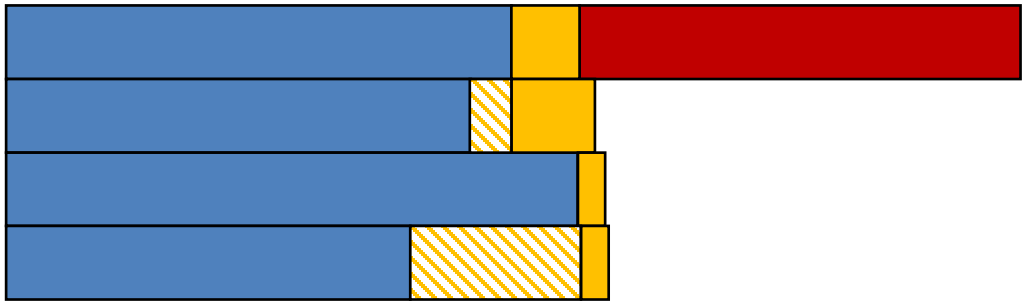
→ Asymptotically, nobody is doing anything except the worker that gets the serial work!

- In reality, it's even worse…

Reality is even worse…

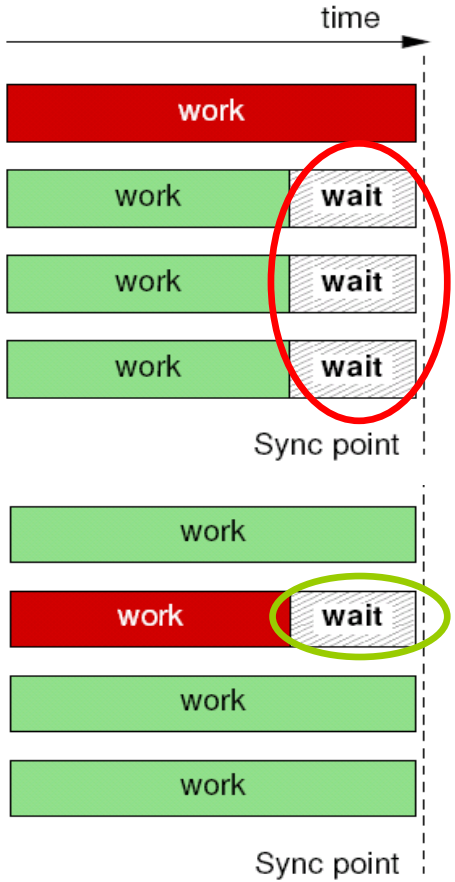
- Load (actually, time) imbalance across workers
  - Serial fraction is a “special case” of this
- Synchronization time
- Communication overhead
- Waiting time due to dependencies
- Resource bottlenecks (e.g., memory or network bandwidth)

$$S(N) = \frac{T(1)}{T(N)} = \frac{1}{s + \frac{1-s}{N} + c(N)}$$



Amdahl generalized: load imbalance

- Load imbalance at sync points
  - More specifically, execution time imbalance
  - p/N assumption no longer valid in general
- Hard to model in general, but two corner cases:
  - A few “laggers” waste lots of resources
    - Single lagger → Amdahl’s Law
  - A few “speeders” might be harmless
- Tuning advice
  - Avoid sync points
  - Turn laggers into speeders

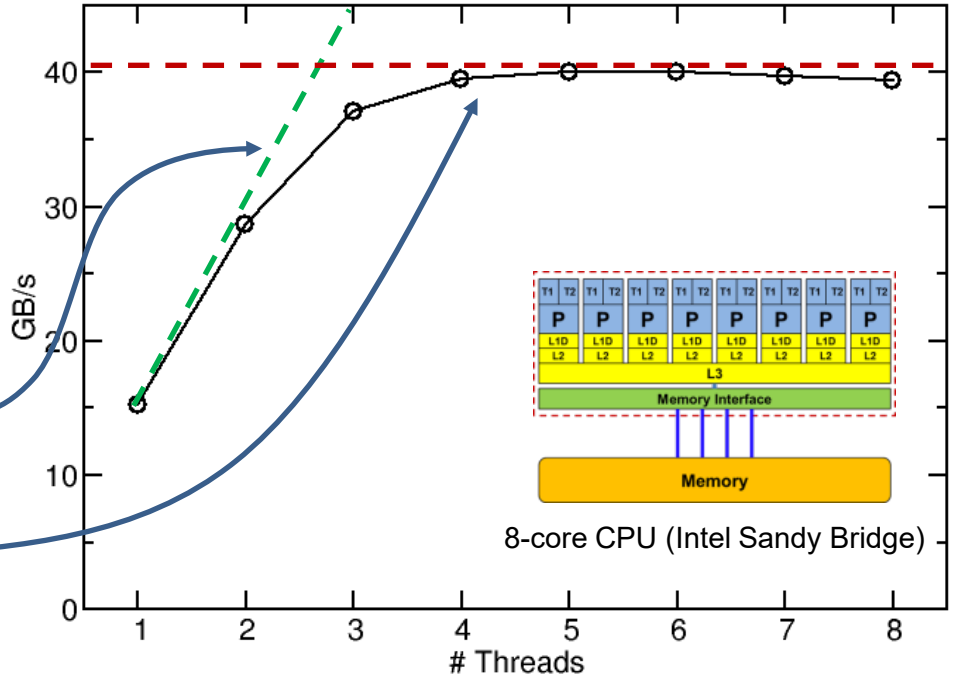


Resource bottlenecks

- Amdahl's Law assumes perfect scalability of resources
- Reality: Computer architecture is plagued by bottlenecks!
- Example: array update loop

```
#pragma omp parallel for
for(i=0; i<10000000; ++i)
  a[i] = a[i] + s * c[i];
```

- Amdahl's: $s = 0$, $c(N) = 0$
  - Perfect speedup? No!
  - Saturation because of memory bandwidth exhaustion



NHR FAU

lrz

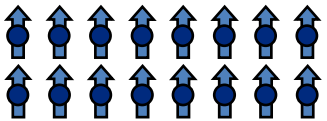
“Common sense” optimizations: A Monte Carlo spin code

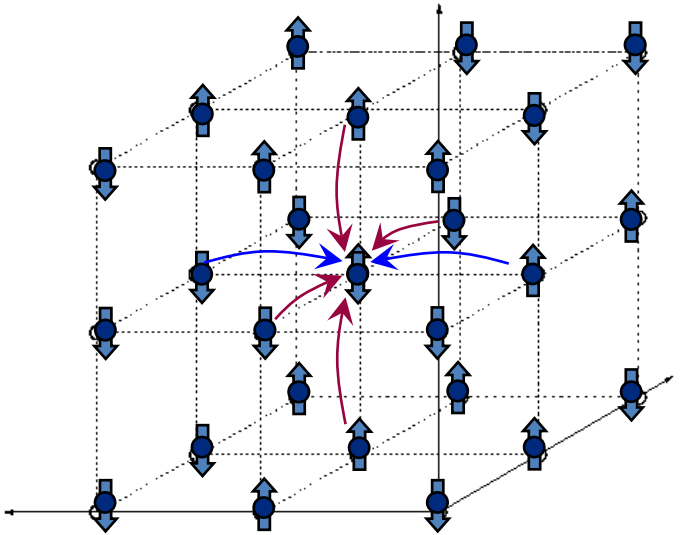
HPC High Performance Computing

Optimization of a Spin System Simulation

- 3-D cubic lattice
- One variable (“spin”) per grid point with values
- +1 or -1

- Next-neighbor interaction terms
- Code chooses spins randomly and flips them as required by MC algorithm





Optimization of a Spin System Simulation

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

- Memory requirement of original program ≈ 1 MByte

Optimization of a Spin System Simulation: *Original Code*

- Program Kernel:

```
      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)

      edelz=iL+iR+iU+iO+iS+iN

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
```

Load neighbors of a random spin

calculate magnetic field

decide about spin orientation

Optimization of a Spin System Simulation: *Code Analysis*

- Profiling shows that
  - 30% of computing time is spent in the **`tanh`** function
  - Rest is spent in the line calculating **`edelz`**
- Why?
  - **`tanh`** is expensive by itself
  - Compiler fuses spin loads and calculation of **`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 (**`tt`** is always 1)
  - Observation: Spin variables only hold values +1 or -1

Optimization of a Spin System Simulation: *Making it Faster*

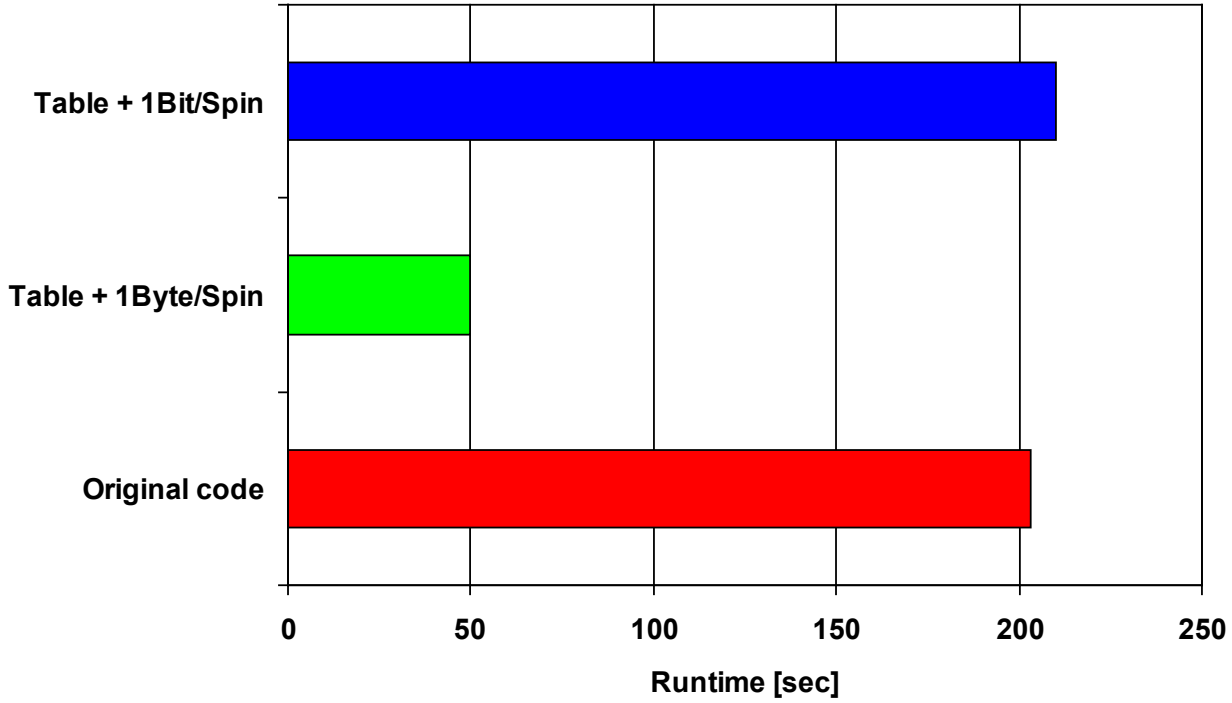
- Strength reduction by tabulation of **`tanh`** function

```
BF = 0.5d0*(1.d0+tanh_table(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 ¼
  - 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 → no benefit
- Potential for a variety of data access optimizations
  - But: choice of spin must be absolutely random!

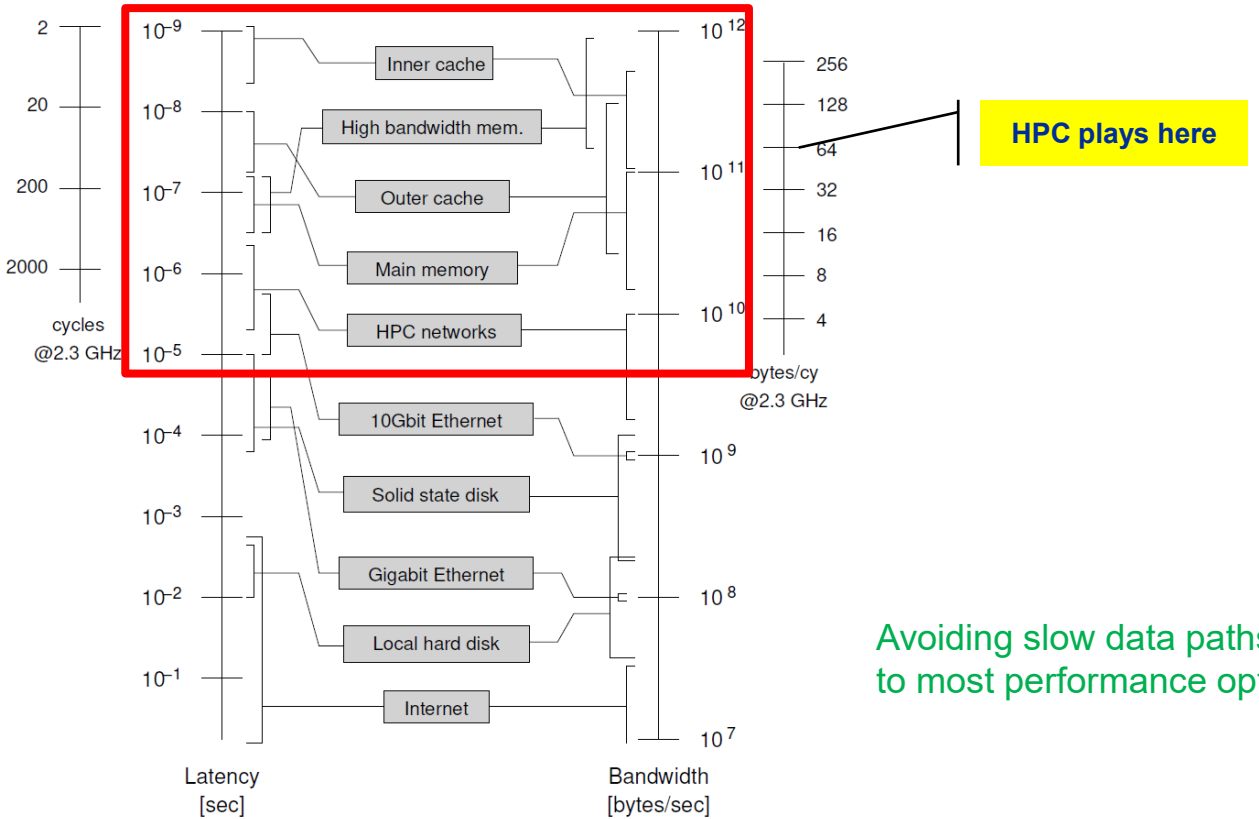
Pentium 4 (2.4 GHz)



Code balance and machine balance

High Performance Computing

Latency and bandwidth in modern computer environments



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

Code balance

- Code balance (B_C) quantifies the requirements of a loop code:

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

- Example: Vector triad `A(:)=B(:)+C(:)*D(:)`
  - B_c = (4+1) Words / 2 Flops = 20 bytes/flop (including write allocate)
  - Often used: "Computational Intensity" $I = 1/B_c$

- General rule: Reducing the code balance of a loop by optimizations will do something good for the performance!

- For refined analysis, code balance can also be defined for all memory hierarchy levels: $B_c^{mem}, B_c^{L3}, B_c^{L2}$
  - Memory transfers are not always the data bottleneck!

Machine Balance

- For quick comparisons the concept of machine balance is useful

$$B_m = \frac{b_S}{P_{\text{peak}}}$$

b_S: Maximum memory bandwidth (meas.)

P_{peak}: Theoretical peak performance

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

Intel Sandy Bridge 8-core 2.7 GHz (2011)	≈ 0.23 B/F
Intel Haswell 14-core 2.3 GHz (2014)	
B_m = 60 GB/s / (14 x 2.3 x 16) GF/s	≈ 0.12 B/F
Nvidia Ampere A100 (2021)	≈ 0.13 B/F (0.65 B/F)
Intel Xeon Ice Lake Platinum (2021)	≈ 0.07 B/F

NHR FAU

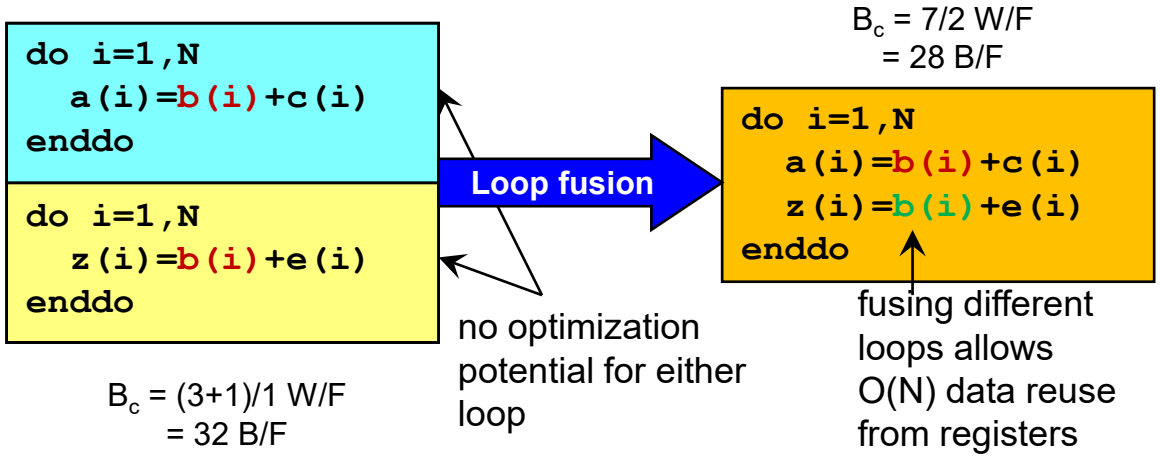
lrz

Code optimization by data access optimization

HPC High Performance Computing

Data access – general considerations

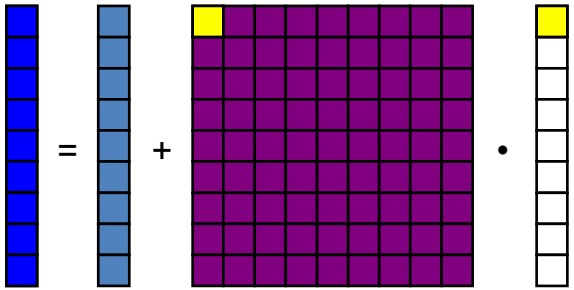
- Case 1: O(N)/O(N) Algorithms
  - O(N) arithmetic operations vs. O(N) data access operations
  - Examples: Scalar product, vector addition, sparse MVM etc.
  - Performance limited by memory BW for large N (“memory bound”)
  - Limited optimization potential for single loops
    - …at most a constant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



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

```
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

- $O(N^2)/O(N^2)$ algorithms cont'd
  - "Unroll & jam" optimization (or "outer loop unrolling")

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

unroll

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

jam

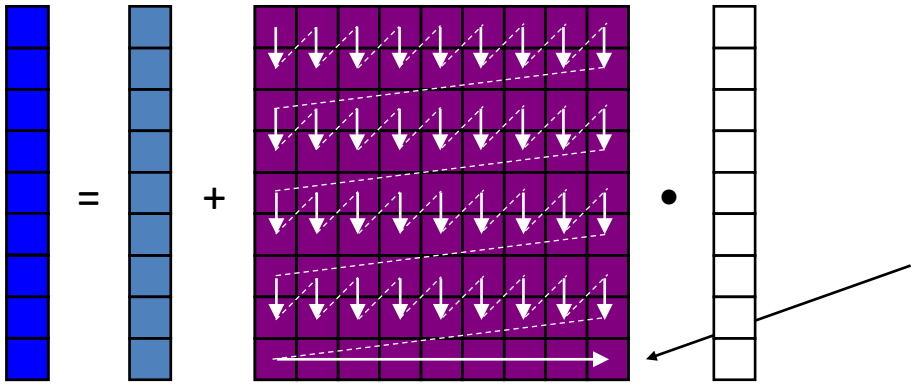
```
do i=1,N,2
 do j=1,N
  c(i)   =c(i)   +a(j,i)   * b(j)
  c(i+1)=c(i+1)+a(j,i+1)* b(j)
 enddo
enddo
```

`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 `b[]` now only loaded N/2 times!

Remainder loop handled separately

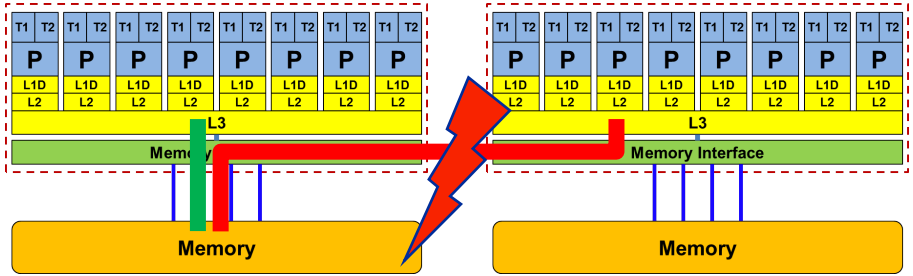
  - Still lower code balance by more aggressive unrolling (i.e., m-way instead of 2-way)
  - Significant code bloat (try to use compiler directives if possible)
  - Large cache → `b[]` may be in cache even without unrolling!

ccNUMA and its implications for performance

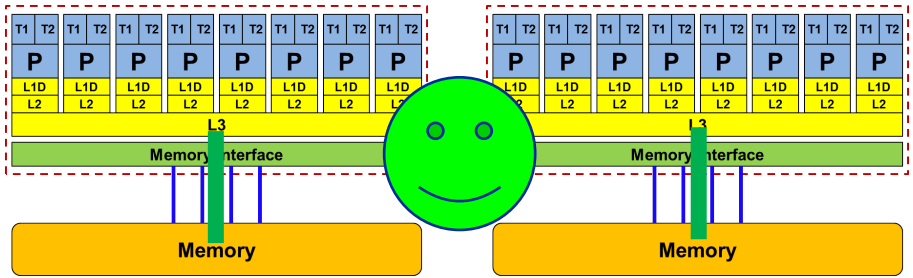
High Performance Computing

ccNUMA and node topology

- ccNUMA:
  - Whole memory is transparently accessible by all cores
  - but physically distributed across multiple locality domains (LDs)
  - with varying bandwidth and latency
  - and potential contention (shared memory paths)
- How do we make sure that memory access is always as “local” and “distributed” as possible?



Note: Page placement is implemented in units of OS pages (often 4kB, possibly more)



ccNUMA default placement policy

"Golden Rule" of ccNUMA:

A memory page gets mapped into the local memory of the processor that touches it first!

(Except if there is not enough local memory available)

- Caveat: "to touch" means "to write," not "to allocate"
- Example:

```
double *huge = (double*)malloc(N*sizeof(double));

for(i=0; i<N; i++)
  huge[i] = 0.0;
```

Memory not mapped here yet

Mapping takes place here

- It is sufficient to touch a single item to map the entire page

Coding for ccNUMA data locality

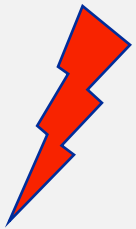
Most simple case: explicit initialization

```
const int n=10000000;
a=(double*)malloc(n*sizeof(double));
b=(double*)malloc(n*sizeof(double));

...


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

#pragma omp parallel for
for(int i=0; i<n; ++i)
    b[i] = function(a[i]);
```



```
const int n=10000000;
a=(double*)malloc(n*sizeof(double));
b=(double*)malloc(n*sizeof(double));
...

#pragma omp parallel
{
#pragma omp for schedule(static)
for(int i=0; i<n; ++i)
    a[i] = 0.;
...
#pragma omp for schedule(static)
for(int i=0; i<n; ++i)
    b[i] = function(a[i]);
}
```

Summary on ccNUMA issues

- If the code is core bound, ccNUMA is not an issue
  - However, most codes have at least some memory boundedness
- Apply first-touch placement
  - Look at initialization loops
  - Consider loop lengths and static scheduling
  - C++ and global/static objects may require special care
- NUMA balancing is active on many Linux systems today
  - Slow process, may take many seconds (configurable), not a silver bullet
  - Still a good idea to do parallel first touch
- If dynamic scheduling cannot be avoided
  - Still a good idea to do parallel first touch