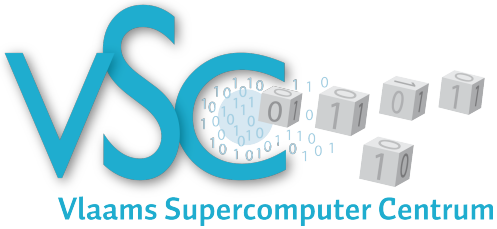
HPC TNT - 1

Tips and tricks – how to efficiently parallelize your code

CalcUA Core Facility



ANNIE CUYT ☐ STEFAN BECUWE ☐ FRANKY BACKELJAUW ☐ GEERT BORSTLAP ☐ ENGELBERT TIJSKENS

Overview

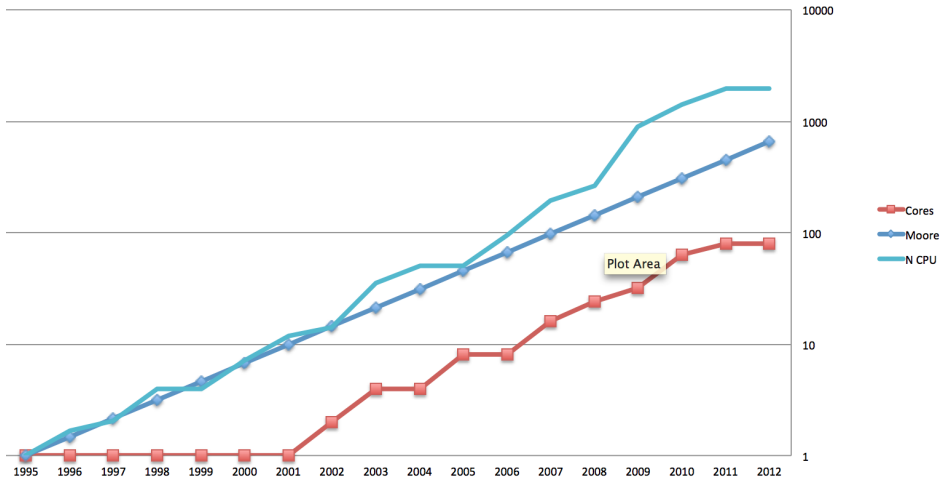
Why parallelize?

1. Reduce time to solution
   - Machines have limited peak performance
2. Solve bigger problems in the same time
   - Machines have limited amount of memory
3. Produce more accurate solutions
   - typically implies
     - More data, e.g. more elements, basis functions, atoms, …
     - More computations, e.g. more detailed physics, …
     - Both
4. …

Why parallelize?

- Computers get faster anyway, no?
- Moore's law still holds

  # of transistors on a chip doubles every 18 months

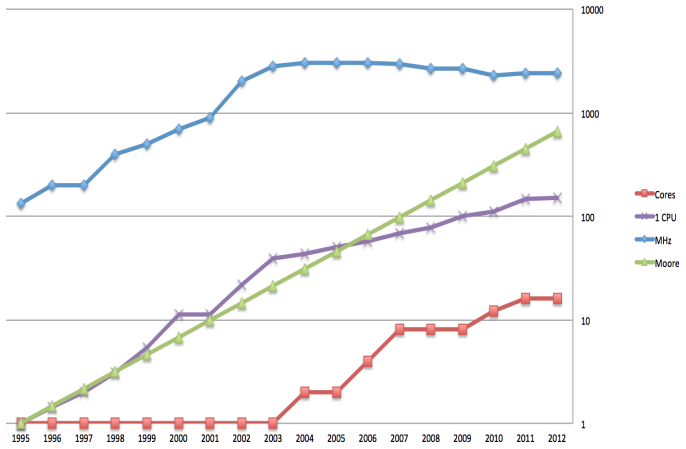


- Used to be equivalent to

  program execution speed doubles every 18 months
- Not any more …

Why parallelize?

- Clock frequency not going up anymore



- Peak performance still increases because it is multiplied by

  number_of_cores
  x threads_per_core
  x vector_width

CPU	GHz ×	sockets x cores ×	threads per core ×	vector width =	GHz x f (Gflops)	GB/s	Flops/GB SP
Harpertown	2.5	4	1	4/2	40/20	4	10
Westmere	2.26	2x6	2	4/2	217/109	8	27.8
Xeon Phi	1.24	61	4	16/8	4841/2420	160	30
Ivy Bridge	2.8	2x10	2	8/4	896/484	80	11.2

Universiteit Antwerpen

VSC Vlaams Supercomputer Centrum

Why parallelize?

- Speed of memory increases much slower than peak performance
- 5 years ago most applications were compute bound
  - Speed ~ peak performance
- Today most applications are memory bound
  - Speed ~ peak bandwidth

Why parallelize?

- Conclusion:

  On future CPUs the speed of a serial code will certainly not increase in pace with Moore's law
- Unless you parallelize

4. Keep your code's performance in pace with Moore's law and stay competetive

Overview

- Why parallelize?

- **When parallelize?**

- Know your goal and minimize coding effort
- Common approaches towards parallelization
- What to parallelize?
- Case study

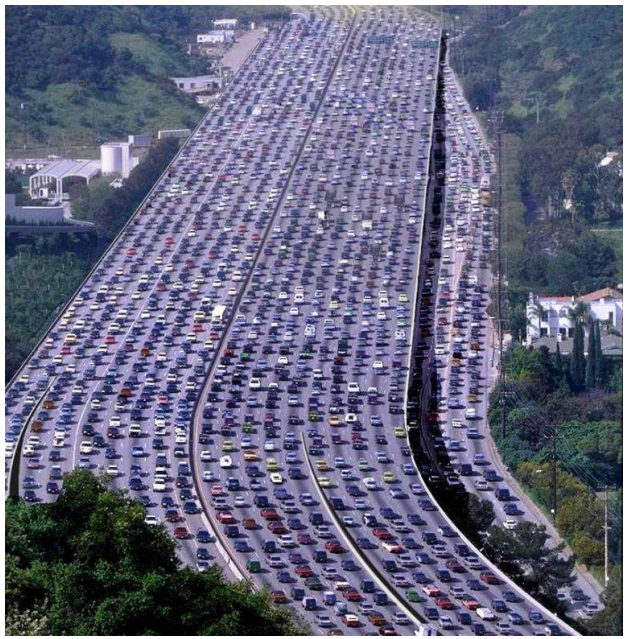
When parallelize?

- When you hit the wall(s) …
  - Program takes too long — Serial optimization
  - Single node memory too small — Shared memory machine
  - Outperformed by competitor
- You need to parallelize … — Are you sure? Consider

When parallelize?
what to do first …

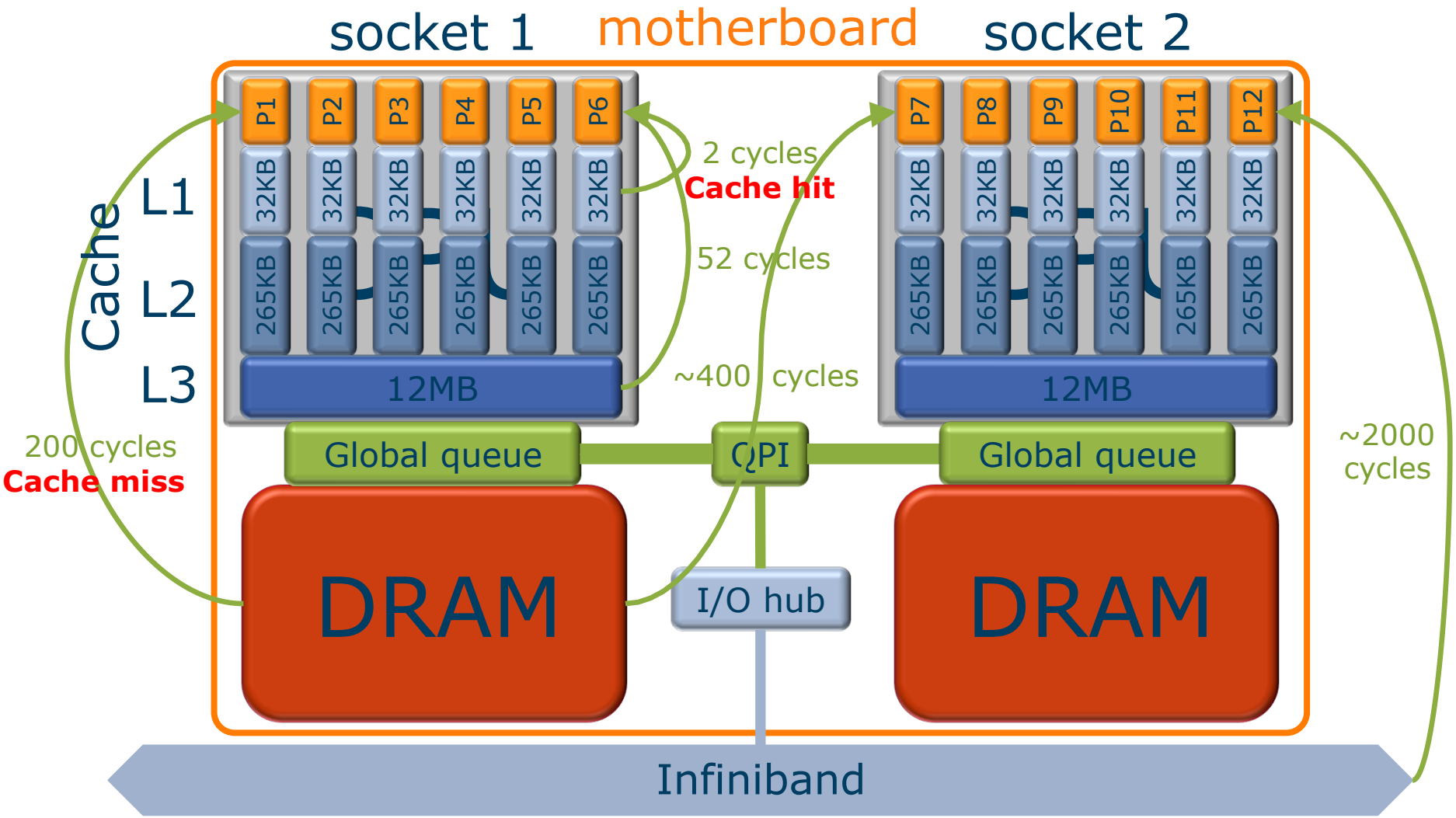
- Optimize serial code – often significant speedup possible
  - Use appropriate data layouts
  - Use appropriate algorithms
  - Use good tools
    - Intel compiler suite generally better than gcc
    - Use HPC libraries
    - Prefer to extend existing codes rather reinventing the wheel
      - Look for well documented code
      - User forum traffic
      - Download counts
      - Citations
  - Compile for CPU you want to run on: -xhost -O3 [-fast]
  - Understand what influences the performance of your code

Serial optimization
what influences serial performance?



- Data traffic
  - Need ~200 cycles to move data from main memory to CPU
  - Core can do 4x200 single precision operations in that time

The cost of memory
Westmere node layout



The cost of memory

- UMA
  - Uniform memory access
  - Is gone
- ccNUMA
  - Cache coherent non-uniform memory access
  - Multiple copies of data
  - Mechanisms to maintain coherency
  - The farther away data is from processor, the longer it takes to fetch it
  - Complicating factor
  - It is here to stay …
  - Overlap computation and communication
    = form of parallelization

Serial optimization
what influences serial performance?

- Vectorization = parallelization in a single thread
  - SIMD, FMA
    - Apply instruction to vector register instead of scalar register
    - Register width
      - 128 bit = 4 SP = 2 DP Harpertown, Westmere (Turing)
      - 256 bit = 8 SP = 4 DP Ivy Bridge (Hopper)
      - 512 bit = 16 SP = 8 DP Xeon Phi (Vic3)
  - Pipelines
    - LI0 DI0 XI0 LI1 DI1 XI1 LI2 DI2 XI2 …
    - 5 stage pipeline

```
LI0 DI0 XI0 LI5 DI5 XI5 LI10 DI10 XI10 …
    LI1 DI1 XI1 LI6 DI6 XI6  LI10 DI10 …
        LI2 DI2 XI2 LI7 DI7  XI7  LI10 …
            LI3 DI3 XI3 LI8  DI8  XI8  …
                LI4 DI4 XI4  LI9  DI9  …
```

    - Pipelines can be broken by
      - load/store from/to main memory
      - conditional branches

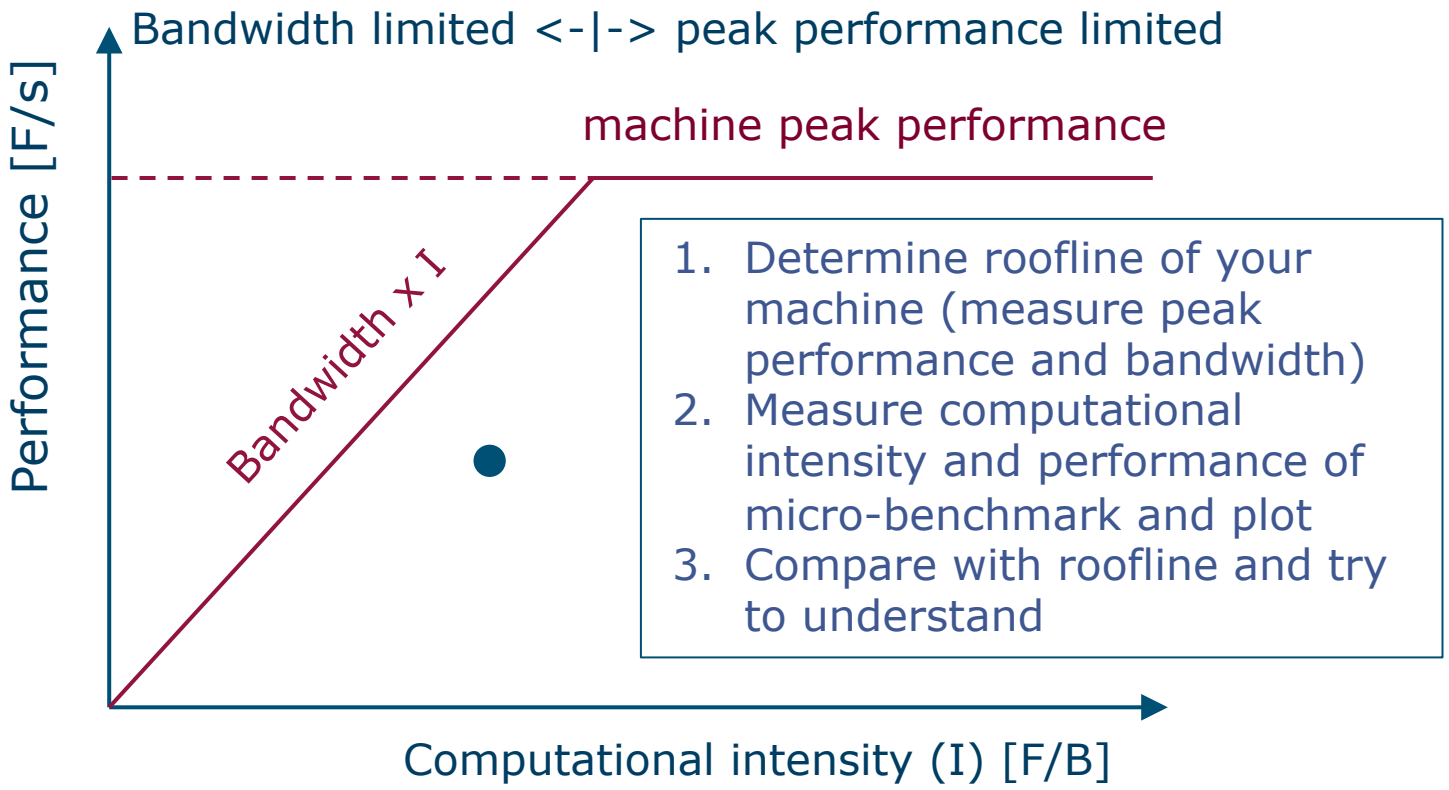
When to parallelize?
and what to do first …

- Criteria for efficient loops – 1

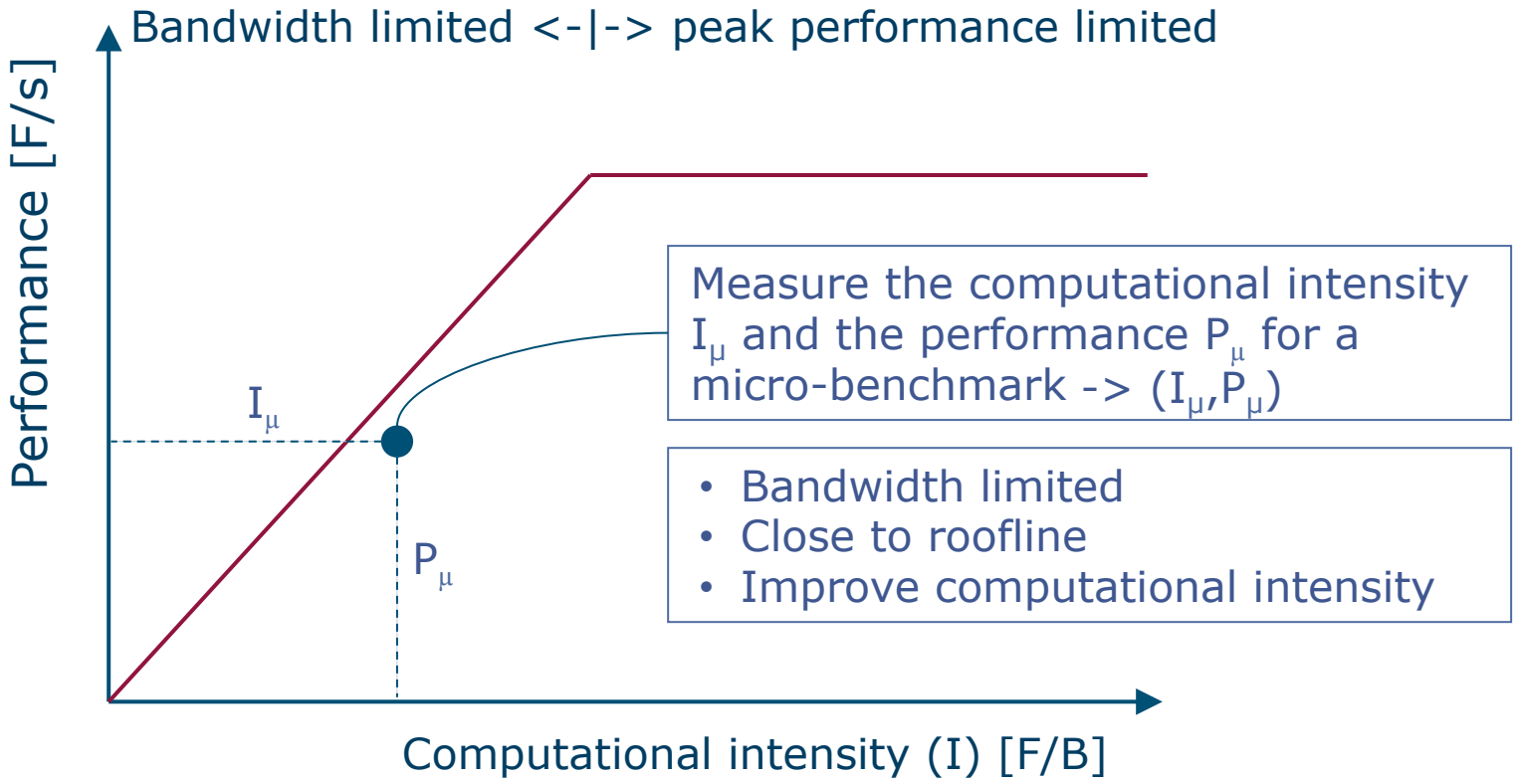
Prefer loops with high computational intensity

- Number of Flops per memory access
- The higher the better
- SIMD very powerful, won't help if memory bound
- Code balance = (Computational intensity)$^{-1}$
- Don't cheat, measure useful work

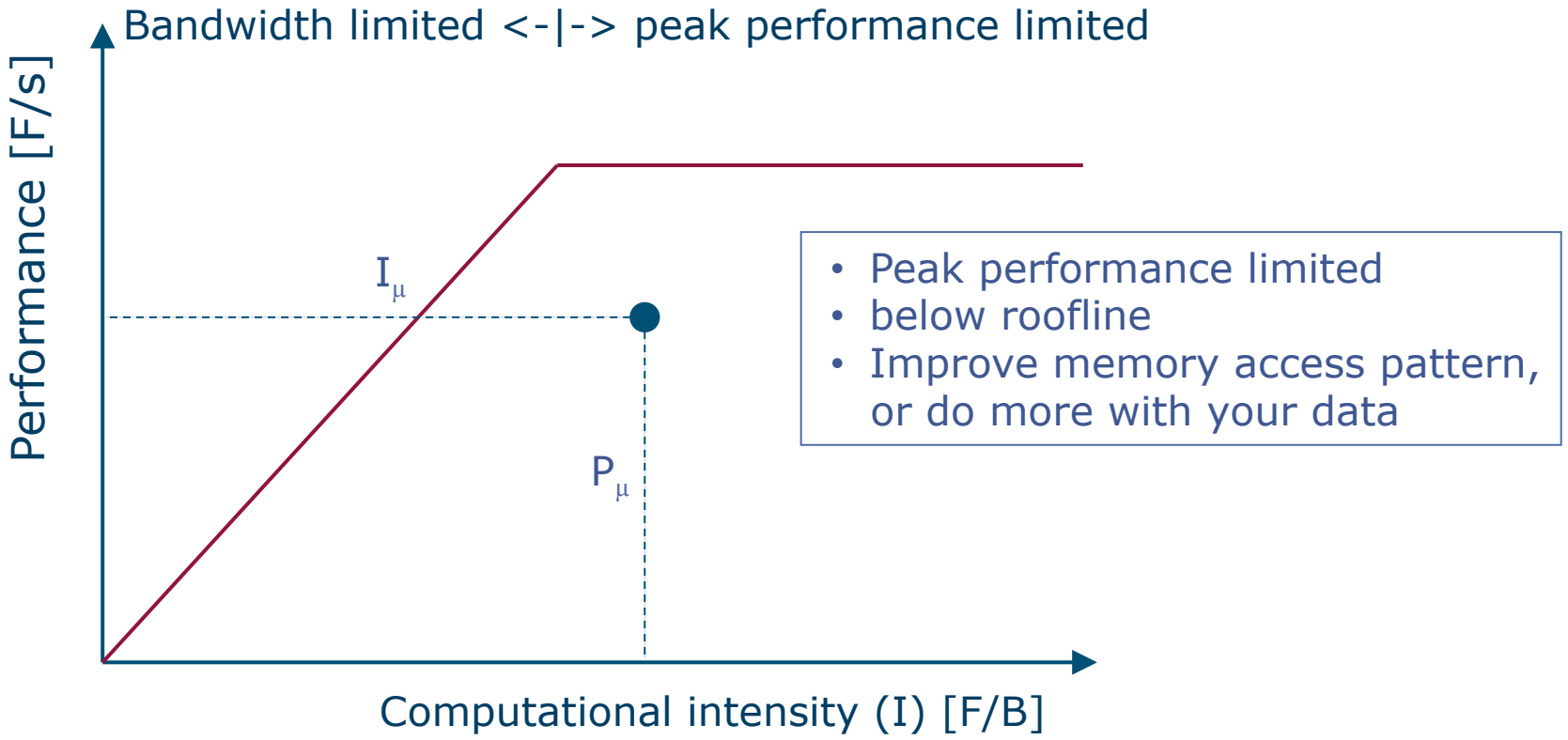
Roofline model



Roofline model



Roofline model



- peak performance/bandwidth
  = FLOPs per byte you need to keep CPU busy
- (for loops that read from main memory)

CPU	GHz	sockets x cores	threads per core	vector width	GHz x f (Gflops)	GB/s	FLOPs/B
Harpertown	2.5	4	1	4/2	40	4	10
Westmere	2.26	2x6	2	4/2	217	8	27.8
Xeon Phi	1.24	61	4	16/8	4841	160	30
Ivy Bridge	2.8	2x10	2	8/4	896	80	11.2

When to parallelize?
and what to do first …

- Criteria for efficient loops – 2

Prefer loops with unit stride

```
float a[N];
for(int i=0; i<N; i+=stride) { a[i] = … }
```

- every load/store loads/stores an entire cache line
  - typically 64 Bytes = 16 SP = 8 DP
- Avoid filling the cache with data you do not need
- Make sure you use the data that are loaded in the cache

Prefer data structures enabling unit stride

- Array of structure (AoS) vs structure of arrays (SoA)

```
struct Particle // AoS
{ double x,y,z,vx,vz,vy,m; };
Particle particles[N];
// not efficient since
// non-unit stride for loops
// which do not use all
// properties in the struct.
// Also bad for SIMD vectorization
```

```
struct Particles // SoA
{ double x[N],y[N],…; };
Particles particles;
// efficient, also for SIMD
// but mind indirect data access!
```

The cost of memory

- Criteria for efficient loops – 3

Prefer predictable loops

- Latency can be hidden by prefetching (compiler does this)

Avoid unpredictable loops

- Conditional branches
  - Break pipelines
- Indirect addressing/Pointer chasing
  - Bad memory access patterns, many cache misses
- Compiler has no clue …

When to parallelize? Or what to do first …

- Cache size = 32 Kb = 4K doubles
- Cache line size is 64 bytes or 4 DP or 8 SP
  - Every read will transfer 64 bytes to L1
  - Main memory->L1 200 cycles
  - L3->L1 52 cycles

- **Cache misses are expensive:**
  you easily miss hundreds of compute cycles
- Exploit spatial and temporal data locality

When to parallelize? Or what to do first …

- Spatial data locality =

  make sure you use all data in a cache line

  don’t jump around in main memory

  > Sort data approximately in the order that you need them in loops

- Temporal data locality =

  once your data is in L1 cache, use it as much as possible

  > Apply tiling

Tiling

```
// inefficient
// except small problems

For all items
   Do this
For all items
   Do that
For all items
   Do something else


// data transferred 3x
```

```
// efficient
// chunk = collection of items
//         that fits in L1 cache
For all chunks
   For all items in chunk
      Do this
   For all items in chunk
      Do that
   For all items in chunk
      Do something else


// data transferred once
```

When to parallelize?
and what to do first …

- Criteria for efficient loops – 4

Prefer long loops

- Amortize startup and cleanup cost of pipelines and loop overhead

Criteria for efficient loops – summary

Prefer loops which

- **have high computational intensity**
- **have unit stride**
- **are predictable**
- **are long**

The numbers tell the tale

- Know where to optimize
  - Think! – may be sufficient for small program
  - Otherwise, measure performance using profiling tools
    - gprof
    - Llkwid-perfctr
    - PerfExpert
    - Intel Vtune
    - Allinea Map
  - Hardware counters
    - # instructions
    - # cache misses
    - # memory read/writes
    - # TLB misses

Overview

- Why parallelize?
- When to parallelize?

- **Know your goal and minimize coding effort**

- Common approaches towards parallelization
- What to parallelize?
- Case study

Know your goal
Minimize coding effort

- How much memory do you need?
- Required time to solution?
  - May include development time
- How many cpu years will your code run?
- How much time can you afford to spend on coding?
- Is anything available in open source community?
  - A few days of *googling* around may save you months of development
  - Your programming skills will improve **more** by using someone else's good code than by trying to reinvent the wheel

Overview

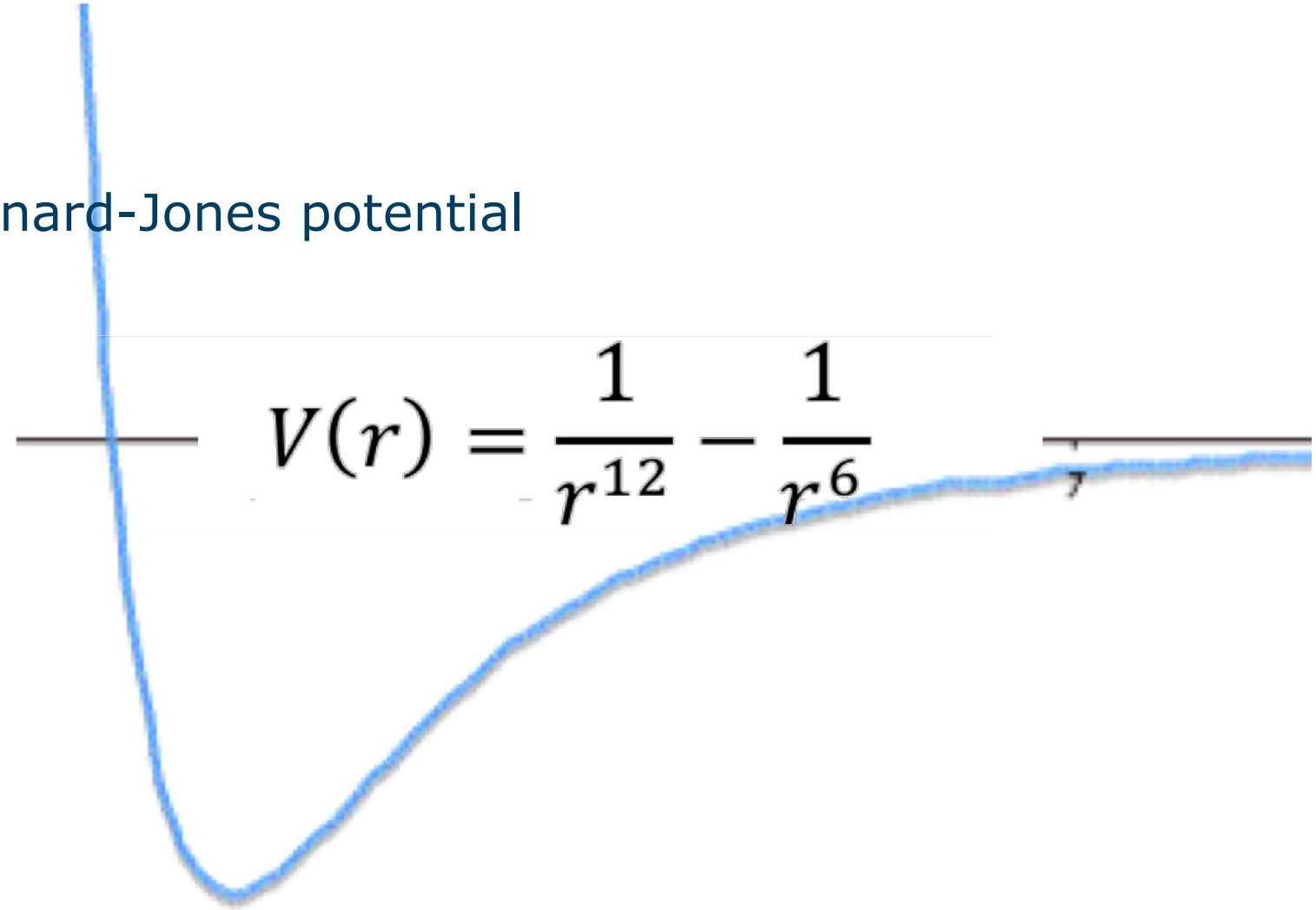
- Why parallelize?
- When to parallelize?
- Know your goal and minimize coding effort

- **Interludium**

- Common approaches towards parallelization
- What to parallelize?
- Case study

How (not) to program …

- Lennard-Jones potential



How (not) to program ...

```
double VLJ0( double r ) {
    return 1./pow(r,12) - 1./pow(r,6);
}                                                    // 18.0 x slower
double VLJ1( double r )_{
    return std::pow(r,-12) - std::pow(r,-6);
}                                                    // 14.9 x slower
double VLJ2( double r ) {
    double tmp = std::pow(r,-6);
    return tmp*(tmp-1.0);
}                                                     // 7.8 x slower
double VLJ3( double r ) {
    double tmp = 1.0/(r*r*r*r*r*r);
    return tmp*(tmp-1.0);
}                                                    // 1.01 x slower
double VLJ( Real_t r ) {
    double rr = 1./r;
    rr *= rr;
    double rr6 = rr*rr*rr;
    return rr6*(rr6-1);
}                                                      // 1 x slower
```

Overview

- Why parallelize?
- When to parallelize?
- Know your goal and minimize coding effort

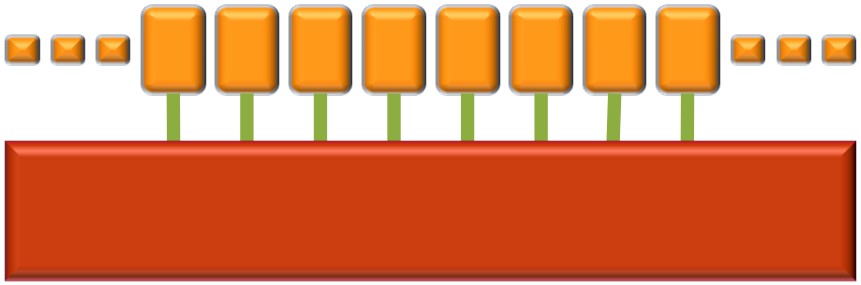
- **Common approaches towards parallelization**

- What to parallelize?
- Case study

Common parallelization approaches

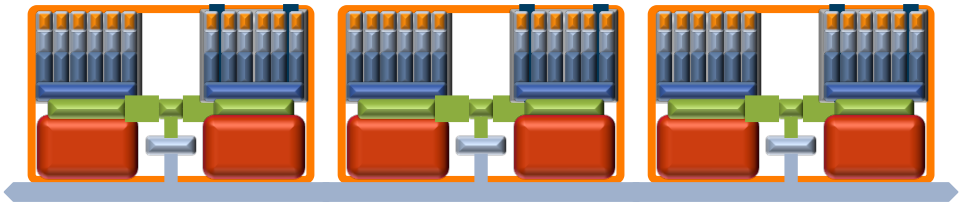
Shared memory machine

- One global address space (not necessarily uniform)
- No (explicit) communication



Distributed memory machine

- No global address space
- One process per thread
- Each process has its own address space
- Communication between processes to share data



Hybrid machine

- Each process manages several threads
- One global address space per process
- One process per socket, or per pair of SMT threads, …

Common parallelization approaches

- **Shared memory**
  - OpenMP (C/C++/Fortran)
  - Intel TBB (C++)
  - Intel cilk++ (C++)
  - (Raw threads)
  - (MPI) (C/C++/Fortran)
  - Charm++ parallel objects (C++)
  - Global Array toolkit (C/C++)
- **Distributed memory**
  - MPI
  - Charm++ parallel objects
  - Global array toolkit
- **Hybrid**
  - MPI between nodes, shared memory approach in each process

Overview

- Why parallelize? Computers get faster anyway, No?
- When to parallelize? Or what to do first
- Know your goal and minimize coding effort
- Common approaches towards parallelization

- **What to parallelize?**

- Case study

Universiteit Antwerpen

Vlaams Supercomputer Centrum

What to parallelize?

- Communication is overhead and slow
  - Bandwidth and latency order of magnitude worse than main memory access
  - Try overlapping communication and computation
- Generally, prefer parallelizing
  - Large loops with high computational intensity
  - Tasks with little communication
  - Big chunks of code over small ones (coarse grained)
  - Chunks of a fixed load over variable loads
  - Otherwise, schedule largest tasks first

overview

- Why parallelize?
- When to parallelize?
- Know your goal and minimize coding effort
- Common approaches towards parallelization
- What to parallelize?

- **Case study – Molecular Dynamics**

Test case

- Small molecular dynamics code
- Kindly provided by Jesus Eduardo Galvan Moya from Physics Department – Condensed Matter theory
- Serves many didactical (HPC) purposes
  - Simple code, not to big, easy to understand, …
  - Full of issues you should learn to pay attention to

Universiteit Antwerpen

VSC Vlaams Supercomputer Centrum

Case study

- Ground state energy calculation of atomistic system
- 0K, no velocities
- 10-150 atoms
- Pairwise interaction potential, brute force (no cut off)
  - 50*49/2 = 1125 pair potential evaluations
  - 150*149/2 = 11175 pair potential evaluations
- 1000 runs of 200000 atom moves (Monte Carlo samples)
- Followed by quasi-Newton method to improve the MC minimum
- Fortran90

Case study

```
do i_run=1,1000                                  local minimum loop   }
    do i_mcs=1,200000                            Monte Carlo loop }   }
        Etot=0                                                   }   }
        do i=1,n                                 energy loop  }   }   }
            do j=i+1,n                                           }   }   }
               Etot = Etot + f(r_ij)                             }   }   }
            end do                                               }   }   }
        end do                                                   }   }   }
        keep lowest energy configuration                         }   }
    end do                                                       }   }
    improve local minimum with quasi-newton                              }
end do                                                                   }
keep lowest energy configuration = global minimum (hopefully)
```

Case study
memory footprint

- Problem size
  - 3 position coordinates * nAtoms * 8 bytes/coordinate
  - 50 atoms -> 1200 bytes ~ 1.2 Kb
  - 150 atoms -> 3600 bytes ~ 3.6 Kb
- Turing harpertown nodes – 2 quad cores
  - 2x4=8 threads
  - L1 Cache size = 32 Kb per core
- Turing Westmere nodes – 2 six cores with SMT
  - 2x6x2=24 threads
  - L1 Cache size = 32 Kb per core
- Fits in L1 cache easily, no need for tiling

Case study - optimization

- Which parts in the code need optimization?
  - Energy loop (pairs of atoms)
    - Small part of code
    - Executed most often 1125x200000x1000
    - Definitely needs optimization ("*What to do first*")
  - Monte Carlo loop
    - NR part represents small fraction of loop, little to be gained
  - Global minimum loop
    - Mainly just a loop

Case study – optimization
Energy loop

```
do i=1,N
   ri(1) = x(i)
   ri(2) = y(i)
   ri(3) = z(i)
   do j=i+1,N
      rj(1) = x(j)
      rj(2) = y(j)
      rj(3) = z(j)
      Etotal = Etotal + Energy(ri,rj)
    end do
end do
```

Case study – optimization
Energy loop

```
function Energy(ra,rb)
!...
  r=sqrt((rb(1)-ra(1))**2+(rb(2)-ra(2))**2+(rb(3)-ra(3))**2)
  Energy = intpot_fitting(r)
return
end

function intpot_fitting(r)
!...
  intpot_fitting = Acoeff*exp(-alpha*r)/r**npow
    - Bcoeff*exp(-beta*(r-catt))/((r-catt)**nattractive+datt)
    - 0*Ccoeff/r
return
end
```

Case study – optimization
Energy loop

```
function Energy(ra,rb)
!...
  r=sqrt((rb(1)-ra(1))**2+(rb(2)-ra(2))**2+(rb(3)-ra(3))**2)
  Energy = intpot_fitting(r)
return
end

function intpot_fitting(r)
!...
  intpot_fitting = Acoeff*exp(-alpha*r)/r**npow
    - Bcoeff*exp(-beta*(r-catt))/((r-catt)**nattractive+datt)
    - 0*Ccoeff/r
Return
end
```

- Both functions have high computational intensity
- Loops will be compute bound

+,-,* are cheap	1 cycle
/, exp(.), sqrt(.) are expensive	~20 cycles
pow(.,.) is very expensive	~100 cycles

Case study – optimization
Energy loop

- 50 atoms
  - Fortran version 150 µs (auto-vectorization turned off)
  - C/C++ version 144 µs
- I am not a Fortran specialist
- Tried to optimize a C/C++ version first

Case study – optimization
Energy loop

- Energy loop contains two nested function calls
- Prevents C++ from vectorizing the inner loop
- Remove call to Energy, compute interatomic distance rij in loop body and call intpot_fitting
- 144->96 µs (now vectorized)

Case study – optimization
Energy loop

- Intpot_fitting contains division by power:

  ```
  Acoeff*exp(-alpha*r)/r**npow
  ```

- Equivalent to

  ```
  Acoeff*exp(-alpha*r)*r**(-npow)
  ```

- But saves a division ~20 cycles
- 144->96->93 µs
- Only small gain because Intpot_fitting is expensive anyway (5 intrinsic function calls)

Case study – optimization
Energy loop

- Remove the call to intpot_fitting and compute in the body of the loop
- 144->96->93->93 µs
- Compiler good at inlining 1 function call, not 2.

Case study Energy loop

- Inner loop runs over [i+1,N[
- So its data is not always well aligned
- Can slow down vectorization
- Let inner loop run over [0,i[
- 144->96->93->93->93 µs

Case study – optimization
Energy loop

- N-1 inner loops, not very long
- Split loop
  - Compute and store interatomic distance in array of length N*(N-1)/2 = 1225
  - loop over atomic distance array and compute intpot_fitting in one long array instead of many short ones
- 144->96->93->93->93->86µs

- Criteria for efficient loops – 4 – Prefer long loops

Case study – optimization
Energy loop

```
for( int i=1, ij=0; i<N; ++i ) {// may safely skip i=1
   Real ri[3] = { x[i],y[i],z[i] };
   for( int j=0; j<i; ++j, ++ij ) {
      r[ij] = sqrt( sq(x[j]-ri[0])
                  + sq(y[j]-ri[1])
                  + sq(z[j]-ri[2]) );
   }
}
Etotal = 0;
for( int ij=0; ij<N*(N-1)/2; ++ij ) {
   Etotal += Acoeff*exp(-alpha*r[ij])/pow(r[ij],npow)
           - Bcoeff*exp(-beta*(r[ij]-catt))/
                (pow(r[ij]-catt,nattractive)+datt)
           - 0*Ccoeff/r[ij];
}
```

Keep sqrt for computational intensity

Case study Energy loop

- Applying the same techniques in fortran
- Auto-vectorization
- 150->95 µs
- Lower triangle and remove a division
- 150->95->88 µs
- Loop splitting
- 150->95->88->85 µs

Case study – optimization
Energy loop

- Same result as C++ -> confidence
  - Analysis
  - Programming language used
- Common misconception
  - Fortran is efficient
  - C++ is not efficient
- Rather :
  - Computational efficiency necessitates a particular programming style
    (stay away from high level C++ features)

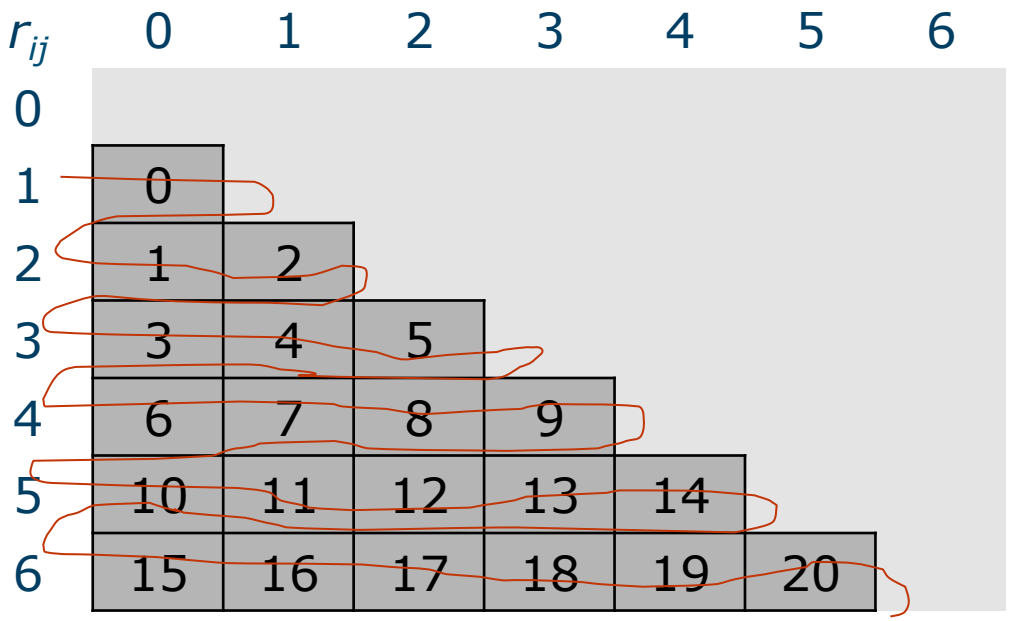
Are we satisfied …

- Speedup 95->85 µs = 1.12 ☹
- 150 atoms (x3) -> 747µs (x8.7 ~ $x3^2$)
- Energy loop is $O(N^2)$
- Atoms x10 -> cputime x100 ☹
- Alternatives?

Case study – optimization
Energy loop

- **Understand your code!**
- This Monte Carlo sampling moves only **one** atom at a time
- For N atoms only N-1 interatomic distances and interaction energies change instead of all N(N-1)/2
  - $N(N-1)/2 \sim N^2/2$
- Exploit this!

Store r_{ij}

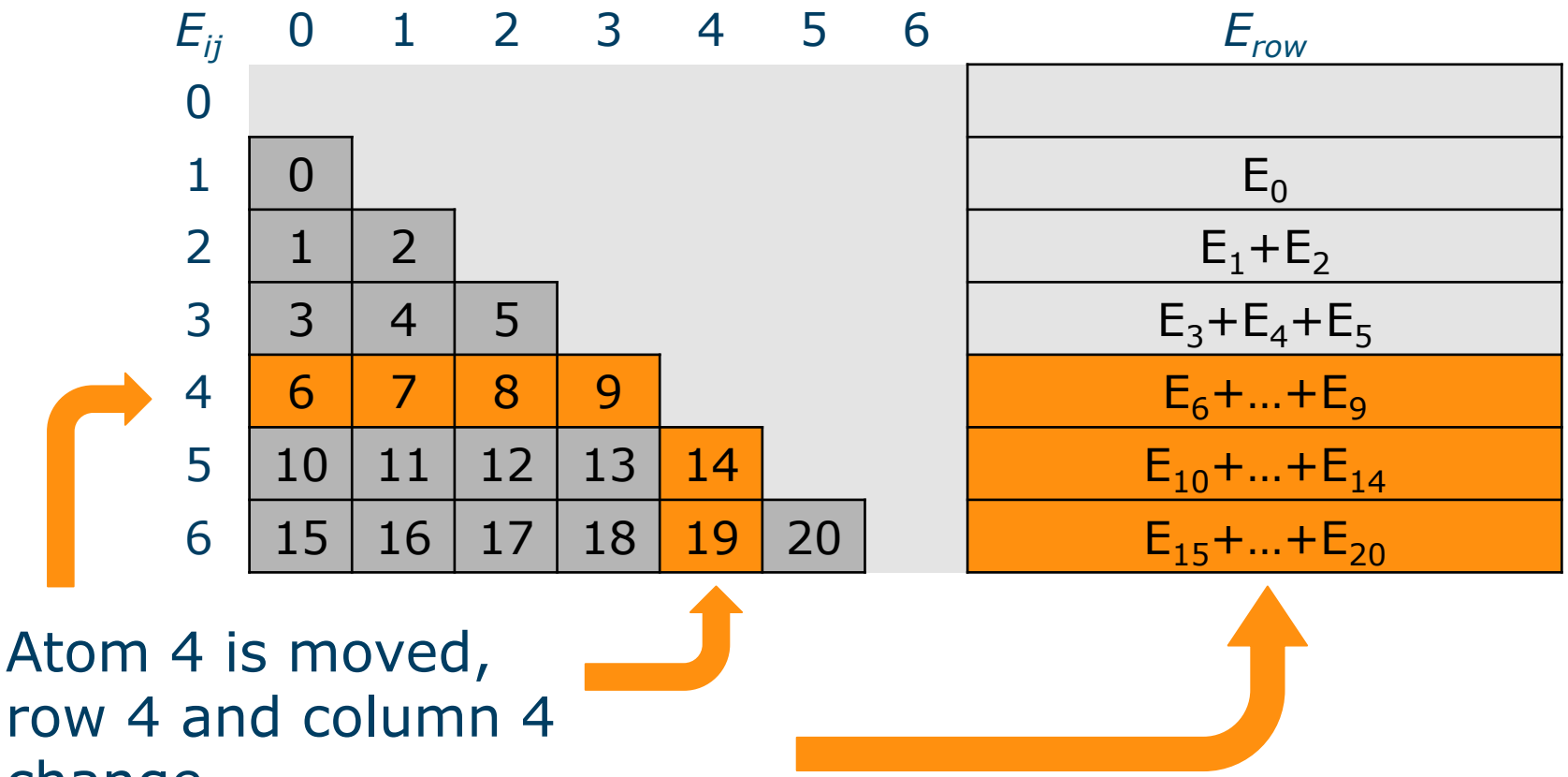


- Linear array containing
  r_{10}, r_{20}, r_{21}, r_{30}, r_{31}, r_{32}, r_{40}, r_{41}, r_{42}, r_{43}, ...

Store E_{ij} and row sums

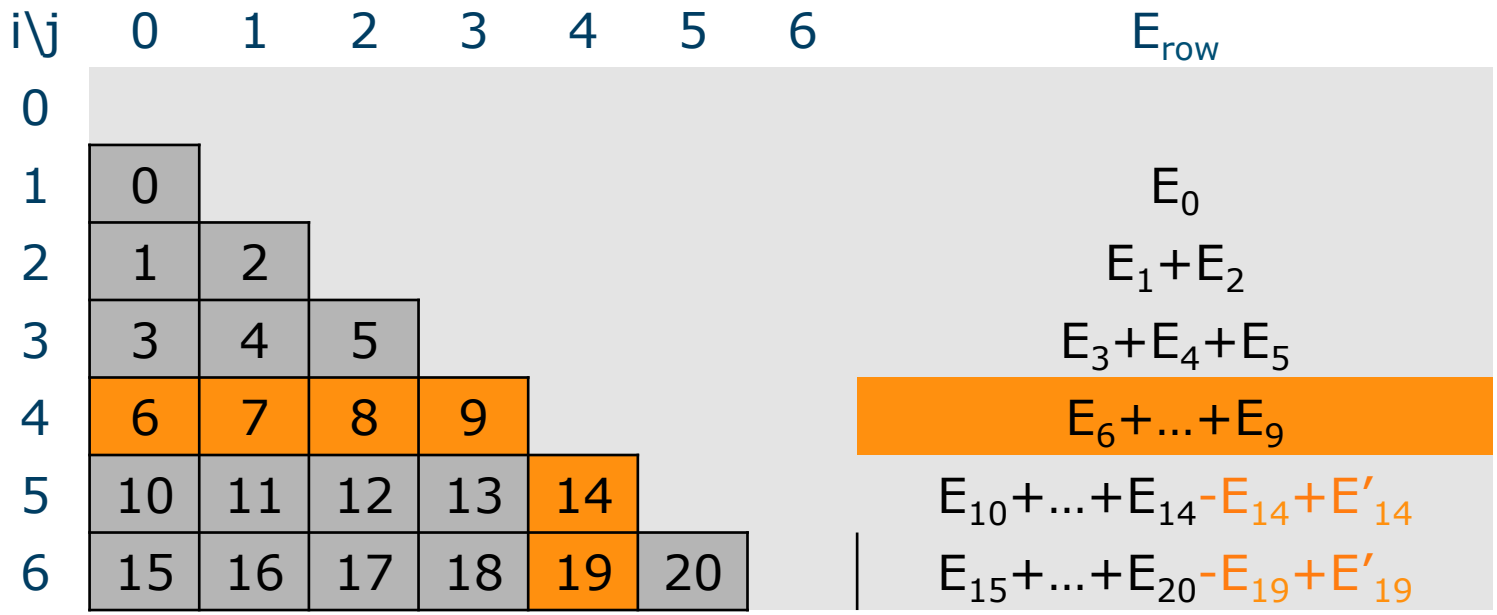
E_{ij}	0	1	2	3	4	5	6	E_{row}
0								
1	0							E_0
2	1	2						E_1+E_2
3	3	4	5					$E_3+E_4+E_5$
4	6	7	8	9				$E_6+\ldots+E_9$
5	10	11	12	13	14			$E_{10}+\ldots+E_{14}$
6	15	16	17	18	19	20		$E_{15}+\ldots+E_{20}$

Move 1 atom



Atom 4 is moved, row 4 and column 4 change

New algorithm



- More code, loops harder to optimize for compiler (shorter loops, varying stride, …)
  - E.g. recompute loops not vectorized yet …
  - Little hope for auto-vectorization
  - SIMD Vectorization certainly possible using Vectorization library (e.g. Vc or Boost.simd in NT^2). Expect speedup x2 on Turing, x4 on Hopper
- Nontrivial code requires documentation

What did we gain?

nAtoms		$O(N^2)$		O(N)		Speedup
50		86		5.7		15.1
150	(x3)	747	(x9)	17.3	(x3)	43.2
500	(x10)	8616	(x100)	57	(x10)	151.2

- Always look for O(N) algorithms if problem size is likely to grow in the future
- Always start with the simplest algorithm so you have a reference for correctness testing

Profiling with PerfExpert

- PerfExpert is a tool that combines a simple user interface with a sophisticated analysis engine to:
  - Detect and diagnose the causes for core, socket, and node-level performance issues.
  - Provide a performance analysis report and suggestions for remediation.
  - Apply pattern-based software transformations to enhance performance.

Profiling with PerfExpert

PerfExpert output of inner loop above:

```
Loop in function __svml_pow2_h9 in ~unknown-file~:0 (68.58% of the total runtime)
================================================================================
ratio to total instrns    %  0.........25.........50.........75.........100
 - floating point       100.0 ****************************************
 - data accesses         25.4 **********
* GFLOPS (% max)         27.3 **********
 - packed                15.2 ********
 - scalar                12.1 ******
--------------------------------------------------------------------------------
performance assessment  LCPI good.....okay.....fair.....poor.....bad
* overall               0.58 >>>>>>>>>>>
* data accesses         0.91 >>>>>>>>>>>>>>>>>
 - L1d hits             0.89 >>>>>>>>>>>>>>>>>
 - L2d hits             0.02
 - L3d hits             0.00
 - LLC misses           0.00
...
```

The program spends 69% evaluating std::pow(.,.) (Also 18% on std::exp(.)) These are called by $f(r_{ij})$.

The program is compute bound (as expected).

The overall performance is considered okay

Roughly half of the instructions is still scalar, although pow and exp are vectorized (svml prefix).

Data fits in L1d cache (as expected).

Still room for improvement?

- Interaction potential very expensive
  - 2 pow, 2 exp, 1 sqrt, 3 div
- Still high fraction of scalar code

Case study - Parallelization

- Which part in the code can be parallelized?
  - Energy loop (pairs of atoms)
    - Small part of code
    - Executed most often 1125x1000x200000
  - MC sampling loop
  - Local minimum loop
- Each can be parallelized
- Which is best?

Case study

- Approach -> OpenMP
  - Simple from programmer's point of view
- When using OpenMP always go for intel compiler suite overhead start/stop/synchronization is much larger for gcc OpenMP than for intel OpenMP
  - Intel ~ 100s of cycles
  - Gcc ~ 1000s of cycles
- There are very good alternatives to OpenMP
  - Intel Threading Building Blocks (TBB)
  - Intel Cilk++
  - (but not for Fortran)
- Disadvantage = shared memory

Case study
// energy loop

- Energy loop relatively short (trip count 1225) and fine grained
  - Overhead of starting and stopping OpenMP threads will kill us
  - O(N) algorithm even worse

Case study
// MC loop

- MC sampling loop much longer (trip count 200.000) and coarse-grained
  - Using OpenMP we should be able to run 24 threads
    - good code balance -> threads can be kept busy
    - small memory footprint -> everything can stay in cache
    - Ideal situation
  - Each thread would do 200.000/24 atom moves (and Etot computations)
  - Sufficient to amortize overhead of starting and stopping OpenMP threads
  - This leaves quasi-Newton minimization serial

Case study
// global minimum loop

- Global minimum loop small trip count (1000) but very coarse grained
- Could be parallelized with OpenMP
  - Limited to single node = 24 OpenMP threads
  - time to solution is 1000/24=42 x serial execution time

Case study // global minimum loop

- **Think!**
- Global minimum loop iterations are completely independent – no communication between iterations
- Apply process parallelism

Case study
process parallelism

- Strip outer loop from program
  - new program computes single local minimum
- Run this program 1000 times
  - Worker framework – can be done with one job
  - Westmere node can run 24 instances of program simultaneously
- Use a script (python, bash) to process the output of the jobs to pick the global minimum

Case study
SMT

- Westmere nodes can run 2 threads per core (SMT)
- SMT = simultaneous multithreading
  - While thread 1 is not using some functional unit, that functional unit can be used by thread 2
  - Thread 2 “feeds on leftovers of thread 1”
  - Overlap memory traffic and instruction execution
  - Thread scheduling done by hardware
  - Speedup at most 2, usually less

Case study
SMT

- Westmere nodes can run 2 threads per core (SMT)
- Use worker framework
  - 12 threads (no SMT) 68.5 cpu seconds/thread
  - 24 threads (SMT) 96.8 cpu seconds/thread
  - Slowdown of 1.4 for 2x instances
  - Throughput x 1.42
- Time to solution for 1000 runs
  - Using 1000/24 = 42 nodes ~ 1.42 x t_serial_execution
  - Using 1000/12 = 84 nodes ~ 1.00 x t_serial_execution

Case study
process parallelism

- Program parallelization only necessary if the time to solution must be less than t_serial_execution
- This comes at a cost
  - More cputime because of communication
  - Development time

Overview

- Why parallelize?
- When parallelize?
- Know your goal and minimize coding effort
- Common approaches towards parallelization
- What to parallelize?
- Case study – Molecular Dynamics

- **Final remarks**

Final remarks

- Reasons to parallelize
- But, before you parallelize,
- But, before you optimize,
- Before you write code,
- **In any case,**
- Optimize!
- Profile your code
- Consider reusing OS code
- **Talk to us**

engelbert.tijskens@uantwerpen.be

Test

- If you would have to reduce time to solution, which loop would you parallelize?
  - Energy loop
  - MC sampling loop
  - Local minimum loop Loop
- Which approach would you use, ant why
  - OpenMP
  - MPI