Performance Analysis and Optimization

Parallel Programming

2025-10-23

Dr. Georgiana Mania, Dr. Jannek Squar, Prof. Dr. Michael Kuhn

mania@dkrz.de, jannek.squar@uni-hamburg.de, michael.kuhn@ovgu.de

Parallel Computing and I/O
Institute for Intelligent Cooperating Systems
Faculty of Computer Science
Otto von Guericke University Magdeburg
https://parcio.ovgu.de

Outline

Motivation

- Parallel programming is used to increase application performance
  - Parallel applications use multiple cores or even machines
  - Using more resources also increases runtime costs
  - Make sure that resources are used as efficiently as possible
- Parallel computers are complex
  - Measuring performance is not always straightforward
  - Estimating potential performance is even harder

Motivation...

- There are several goals for performance optimization
  1. Minimizing runtime
     - Allows getting the results as fast as possible
     - Typically the most important factor for users
  2. Maximizing throughput
     - Executes as many jobs as possible within a given time
     - Does not necessarily say anything about performance
  3. Maximizing utilization
     - Makes the best use of investment for resources
     - Does not necessarily match the above goals
- Performance measurements are necessary to check goals
  - Measure, assess and optimize

Approaches

- When doing performance optimization, there is a loop:
  1. Conduct performance measurements
     - Running the application, measuring time etc.
  2. Check if performance is satisfactory
     - Might not have anything to do with actual utilization
     - Should also check whether performance is already optimal
  3. Speculate about the reason for the performance problems
     - Measurements can point you in the right direction
  4. Fix performance problems
     - You might actually fix something else (or nothing at all)
- This is more or less "debugging for performance"

Approaches…

- There are two major approaches for performance measurements
  1. Offline approaches
     - Record metrics at runtime, write them to storage
     - Analyze performance afterwards
  2. Online approaches
     - Record metrics at runtime, forward them to a tool
     - Analyze performance at runtime
- In practice, the approaches we use are a mix of both

Offline Approaches

- Benefits
  - Metrics are available for multiple analyses
    - You might want to look at different metrics etc.
  - Allows easily comparing multiple runs
- Drawbacks
  - Typically constant overhead for collecting metrics
  - There is often not an easy way to refine collection
    - If you notice a performance hotspot, you have to rerun the application
  - Metrics can get quite large
    - Up to gigabytes or even terabytes for large applications

Online Approaches

- Benefits
  - Allows adapting collected metrics and thus overhead
  - Easy to switch collection on and off
    - Possible to collect performance metrics in production runs
- Drawbacks
  - Typically not possible to analyze performance afterwards
    - Collected metrics are transient and lost after the application finishes
  - Requires a separate tool that can process online metrics
    - This also makes the whole approach more complex

Outline

Measurements

- It is difficult to measure performance correctly
  - There are many factors and components to consider
  - Random errors can influence results significantly
  - Systematic errors can invalidate all results
- Measuring performance is a complex process
  - Performance is influenced by caching, network, I/O etc.
  - Which components are involved and have to be measured?
  - Which performance can we expect on a given system?

Measurements. . .

- Optimization requires deep knowledge of the hardware
  - How do the different levels of caches interact?
  - Can we reach the main memory from all cores with the same speed?
  - How does our application behave with more cores?
- There are also technical issues to take into account
  - HPC applications are typically run via a batch scheduler
  - Operating system services can influence performance

Measurements…

- Our goal is to collect metrics quantitatively
    - Metrics include runtime, throughput, latency and more
    - The metrics to collect depend on the software and hardware
- Published measurements should be scientifically sound
    - Other scientists should be able to reproduce your findings
    - Measurements of metrics have errors that have to be accounted for
- Results always vary slightly even for the same configuration

Quiz

- Application A runs for 4.274 s, application B for 4.176 s. Which one is faster?
  1. Application A
  2. Application B
  3. Difference is negligible, performance is the same
  4. Not enough information

- Application A runs for 4.274 s, application B for 4.176 s. Which one is faster?
    1. Application A
    2. Application B
    3. Difference is negligible, performance is the same
    4. Not enough information ✔

- Single measurements are more or less random
    - Processor might be busy with something else
    - Some other application is currently occupying the network
    - There is a certain variability for each component
- It is never enough to do a single measurement
    - Always repeat measurements at least three times
    - If you talk to physicists, they will probably say 30 times
- Averaging the metrics is also not enough
    - There are important derived metrics, such as standard deviation etc.

Measurements…

```
Benchmark #1: ./sincos-O2
Time (mean +- sig):    4.192 s +-   0.033 s    [User: 4.181 s, System: 0.001 s]
Range (min .. max):    4.160 s ..   4.274 s    10 runs

Benchmark #2: ./sincos-O3
Time (mean +- sig):    4.191 s +-   0.016 s    [User: 4.179 s, System: 0.001 s]
Range (min .. max):    4.176 s ..   4.221 s    10 runs

Summary
'./sincos-O3' ran
    1.00 +- 0.01 times faster than './sincos-O2'
```

- Application A and B have the same performance
  - Both previous results were extreme values (minimum and maximum)

- There are two kinds of errors
    1. Random errors
        - Cancel out after infinite measurements
        - Might be caused by operating system activity in the background
        - Performance of most hardware varies a bit
        - Larger variations are also possible due to hardware defects, load balancing etc.
    2. Systematic errors
        - These errors do not cancel out with more measurements
        - They are caused by wrong methodology/implementation
        - For instance, you want to measure disk speed but measure the cache

Measurements. . .

- Always use a well-defined hardware/software environment
  - Document the setup, including version numbers etc.
- Minimize external influence to keep random errors low
  - Use resources exclusively if possible
  - For example, do not run anything in the background
- Increase measurement time and repeat measurements
  - This helps canceling out random errors
- Compare results with expected performance
  - "My application finishes in two hours. Could it finish in one?"
  - This typically involves some kind of performance modeling

Measurements…

- Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers by David Bailey [Bailey, 1991]

- Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers by David Bailey [Bailey, 1991]

1. "Quote only 32-bit performance results, not 64-bit results."

- Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers by David Bailey [Bailey, 1991]

1. "Quote only 32-bit performance results, not 64-bit results."
7. "When direct run time comparisons are required, compare with an old code on an obsolete system."

- Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers by David Bailey [Bailey, 1991]

1. "Quote only 32-bit performance results, not 64-bit results."
7. "When direct run time comparisons are required, compare with an old code on an obsolete system."
9. "Quote performance in terms of processor utilization, parallel speedups or MFLOPS per dollar."

- Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers by David Bailey [Bailey, 1991]

1. "Quote only 32-bit performance results, not 64-bit results."
7. "When direct run time comparisons are required, compare with an old code on an obsolete system."
9. "Quote performance in terms of processor utilization, parallel speedups or MFLOPS per dollar."
11. "Measure parallel run times on a dedicated system, but measure conventional run times in a busy environment."

- Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers by David Bailey [Bailey, 1991]

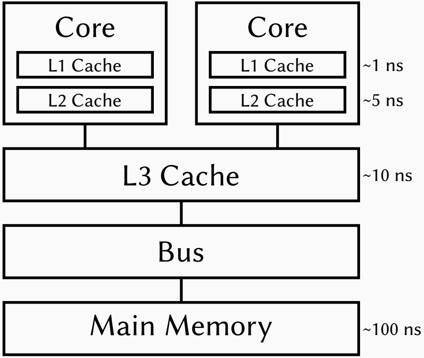
1. "Quote only 32-bit performance results, not 64-bit results."
7. "When direct run time comparisons are required, compare with an old code on an obsolete system."
9. "Quote performance in terms of processor utilization, parallel speedups or MFLOPS per dollar."
11. "Measure parallel run times on a dedicated system, but measure conventional run times in a busy environment."
12. "If all else fails, show pretty pictures and animated videos, and don't talk about performance."

Metrics

- The simplest performance metric: Wall-clock time (or real time)
  - Measure how long the application runs
- There are different kinds of times
  - CPU time denotes the time the processor spent running the application
    - Can be lower or higher than wall-clock time
    - Lower: Two applications share a core, that is, each gets 50 % of CPU time
    - Higher: An application runs on ten cores for one hour, that is, for ten CPU hours
  - User time denotes the time spent in user mode
    - This counts normal calculations etc.
  - System time denotes the time spent in kernel mode
    - This counts system calls, such as I/O

Problems

- Numerous reasons for performance problems
- Inefficient access to resources
  - These are often caused by latencies
  - Data not available in fastest cache
  - Main memory is relatively slow
  - Indirect memory access
- Access conflicts on shared resources
  - Multiple applications want to access the bus
  - File systems are typically shared



Problems…

- Processor utilization is often not optimal
  - Sometimes only 1–10 % are used, especially for parallel applications
  - Parallel applications have communication and synchronization overhead
- Scientific software is often not well-optimized
  - Domain scientists are interested in scientific results, not optimizing software
  - Domain scientists often do not have a computer science background
  - Best case: Domain scientist + mathematician/physicist + computer scientist

Problems...

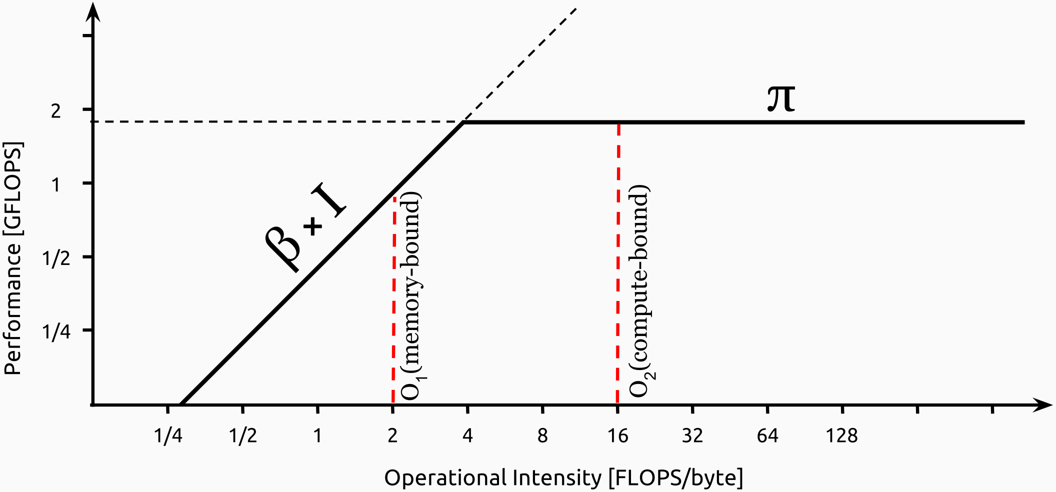
- Application-specific limitations
  - CPU-bound: Limited by processor
    - For instance, processor cannot do more floating point operations
    - Could be solved by increasing the clock rate or adding more floating point units
  - Memory-bound: Limited by memory
    - Data cannot be transferred from the main memory to the processor fast enough
    - Typically caused by not doing enough operations per transferred byte
  - I/O-bound: Limited by storage and/or network
    - Data cannot be transferred to/from storage fast enough
- Unrealistic performance gains, such as superlinear speedup
  - For instance, making the problem smaller allows it to fit into the cache

Approaches

- Theoretical
  - Determine time and memory complexity
  - Can be impractical for general applications
  - Helps to have at least a rough understanding of complexity
    - Get a feeling for potential runtime/memory consumption
- Practical
  - Measure time and memory consumption
  - Relatively easy to do with the right tools
- A combination of both approaches makes most sense

Roofline Model

- One way to assess performance is the so-called roofline model
    - Visual representation of performance limits in current architectures
    - Requires finding out peak memory throughput and computational performance
    - Application's operational intensity has to be determined
    - Can be extended using other factors important for performance
- The performance metric given most attention in HPC is FLOPS
    - FLOPS = Floating point operations per second
    - Different metrics are discussed since FLOPS are only one aspect



[Giu.natale, 2016]

Outline

Goals

- The overall goal is to optimize resource usage
  - This applies to all involved components
  - Processor, storage, network etc. require different approaches
- Resources are typically used exclusively in HPC
  - There are exceptions; for example, the file system is shared
  - Problems cannot be compensated by running additional applications
  - Users should make sure that they do not underutilize resources
- Also important for shared resources
  - Worst case: A single application can bring down performance for everyone
  - Applications should not overload the file system

Approaches

- We will focus on the computational performance for now
    - Moreover, we will mainly look at numerical applications

1. Optimize the mathematics and algorithms
    - Requires the most knowledge about the problem
    - Should rather be done by a domain scientist and/or mathematician
2. Optimize the code manually
    - Determine which data structures and algorithms are best suited
        - Vectorization can be a huge performance benefit
    - Take software and hardware characteristics into account
        - How much main memory is available? How does the compiler align/order data?
3. Optimize the code automatically
    - The compiler can take care of a lot of optimizations for us

Programming Languages

- The programming language can also have a huge influence on performance
  - In the end, use the language you are most comfortable with
  - Using a new language will not automatically make your application faster
- There is a wide range of programming languages to choose from
  - C, C++, Fortran, Python, Java, MATLAB etc.
- Some languages are better suited for specific problems
  - For example, good data science and machine learning support for Python

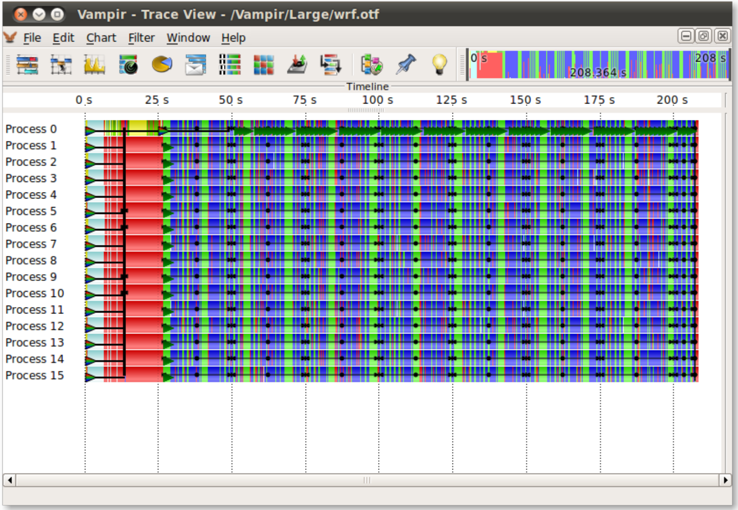
- C (which we will use in the lecture and exercises)
  - Allows low-level programming and direct access to the hardware
  - Requires you to take care of memory management yourself
  - Compilers are mature and produce efficient code
  - Most functionality like threading is supported
  - A lot of performance-critical libraries and framework are written in C
- C++
  - More or less the same benefits and drawbacks as C with a nicer syntax
  - More convenient memory management than C
- Fortran (from Formula Translation)
  - Easier to handle for non-computer scientists
  - Has a long history and is still updated frequently

Programming Languages...

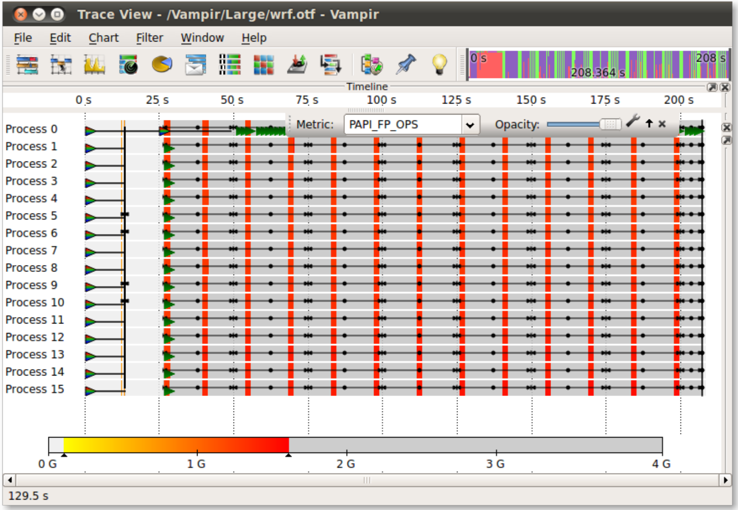
- Python
  - Very popular right now and has a huge community
  - Many modules are available, providing a lot of features
  - Standard version is interpreted and thus slow
    - There are a number of modules written in C for high performance
  - There is no easily usable threading support
- Java
  - Popular in industry, large community and many features
  - Byte code can be optimized at runtime

Tools

- Time measurement
  - `time` and `/usr/bin/time` are available everywhere
  - Can also be done manually using, for example, `clock_gettime`
- Profiling
  - `gprof` can be used to display application profiles
- Dedicated performance analysis
  - `perf` is part of the Linux kernel and features many dedicated metrics
- Graphical applications
  - Vampir is a commercial tool to display traces and profiles



[GWT-TUD GmbH, 2020]



[GWT-TUD GmbH, 2020]

- Simple numerical application
  - Nested loop with calculations
- Two complex operations
  - Plus two simple operations
- Performance expectations
  - sin and cos are expensive
  - Maximum is hard to judge

```c
int main (void) {
    double result = 0.0;
    for (int i = 0; i < 20000; i++) {
        for (int j = 0; j < 20000; j++) {
            result += sin(i) + cos(j);
        }
    }
    printf("result=%f\n", result);
    return 0;
}
```

```
$ time ./sincos
result=10120.671812
./sincos  8.88s user 0.00s system 99% cpu 8.896 total

$ /usr/bin/time ./sincos
result=10120.671812
8.88user 0.00system 0:08.89elapsed 99%CPU (... 2132maxresident)k
0inputs+0outputs (0major+78minor)pagefaults 0swaps
```

- `time` is a shell built-in
  - `/usr/bin/time` is a regular system tool
- Both show user, system and total time as well as processor utilization
  - `/usr/bin/time` also provides memory consumption etc.

- Profiling using gprof does not help in this case
  - Everything is contained in the `main` function
- Compile the application with the -pg flag
  - Running it will automatically produce a profile called `gmon.out`
- Most of the time is probably spent in `sin` and `cos`

```
$ gprof ./sincos
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
101.86      0.81     0.81                             main
```

```
$ perf stat ./sincos
result=10120.671812
 Performance counter stats for './sincos':
          9,016.15 msec task-clock:u              #    0.998 CPUs utilized
                 0      context-switches:u        #    0.000 K/sec
                 0      cpu-migrations:u          #    0.000 K/sec
                68      page-faults:u             #    0.008 K/sec
    37,667,245,120      cycles:u                  #    4.178 GHz
        46,473,927      stalled-cycles-frontend:u #    0.12% frontend cycles idle
    23,374,754,930      stalled-cycles-backend:u  #   62.06% backend cycles idle
    89,573,942,974      instructions:u            #    2.38  insn per cycle
                                                  #    0.26  stalled cycles per insn
    11,597,942,217      branches:u                # 1286.352 M/sec
        45,071,449      branch-misses:u           #    0.39% of all branches
       9.035267264 seconds time elapsed
       9.013823000 seconds user
       0.000000000 seconds sys
```

- `perf` shows a number of different performance metrics
  - Runtime is just one of them
- Context switches occur when talking to the kernel
  - They are relatively fast but should be taken into account
- CPU migrations can have negative influence on caching
  - Moving the application to another core or processor will invalidate caches
- Cycles and instructions show how much the processor had to do
  - Modern processors can do multiple instructions per cycle
- Branches can be bad for performance if there are many misses

Compilers

- Compilers can do a lot of optimizations for us
  - Can also be tuned for specific architectures
    - Takes instruction sets, number of registers etc. into account
- `-O0`
  - Default, no optimizations are performed
- `-O1`
  - Basic optimizations, compilation requires more time and memory
- `-O2`
  - More optimizations, often used as the “default” optimization
- `-O3`
  - Even more optimizations, including vectorization

- `-Og`
  - Optimize for debugging, some important passes are disabled at `-O0`
- `-Os`
  - Optimize for size, good for embedded systems with little storage
- `-Ofast`
  - Optimize by disregarding standards compliance, might influence results

- Inlining allows avoiding function calls (starting from -O1)
  - Function calls require putting arguments onto the stack
  - Afterwards, there are jumps into the function and back to the original location
- Loop unrolling (-O3)
  - Loops also require jumps, which can be negative for performance

```c
for (int i = 0; i < 3; i++) {
    a[i] += b[i];
}
```

→

```c
a[0] += b[0];
a[1] += b[1];
a[2] += b[2];
```

- Vectorization can perform multiple operations at once (-O3)
  - Especially useful in combination with loop unrolling

- Which speedup can we get for our application with compiler optimizations alone?
    1. None
    2. Factor 10
    3. Factor 100
    4. Factor 1,000

```
$ perf stat ./sincos
result=10120.671812
 Performance counter stats for './sincos':
          9,016.15 msec task-clock:u
                 0      context-switches:u
                 0      cpu-migrations:u
                68      page-faults:u
    37,667,245,120      cycles:u
        46,473,927      stalled-frontend:u
    23,374,754,930      stalled-backend:u
    89,573,942,974      instructions:u

    11,597,942,217      branches:u
        45,071,449      branch-misses:u
       9.035267264 seconds time elapsed
       9.013823000 seconds user
       0.000000000 seconds sys
```

```
$ perf stat ./sincos-O3
result=10120.671812
 Performance counter stats for './sincos':
         4,278.80 msec task-clock:u
                0      context-switches:u
                0      cpu-migrations:u
               67      page-faults:u
   17,886,687,516      cycles:u
       19,370,964      stalled-frontend:u
   11,376,027,366      stalled-backend:u
   45,200,173,879      instructions:u

    6,000,368,555      branches:u
       19,211,736      branch-misses:u
      4.288728446 seconds time elapsed
      4.278149000 seconds user
      0.000000000 seconds sys
```

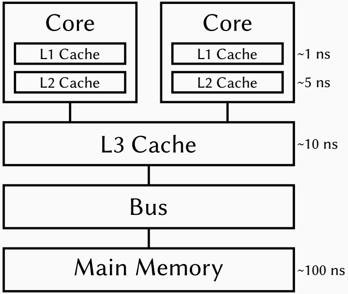
```
$ perf stat ./sincos
result=10120.671812
 Performance counter stats for './sincos':
         9,016.15 msec task-clock:u
                0      context-switches:u
                0      cpu-migrations:u
               68      page-faults:u
   37,667,245,120      cycles:u
       46,473,927      stalled-frontend:u
   23,374,754,930      stalled-backend:u
   89,573,942,974      instructions:u

   11,597,942,217      branches:u
       45,071,449      branch-misses:u
      9.035267264 seconds time elapsed
      9.013823000 seconds user
      0.000000000 seconds sys
```

Compilers...

- This time, `sincos` was compiled with `-O3`
  - Runtime was more than halved from 9 s to 4.3 s
  - Cycles, instructions and branches were roughly halved
  - Instructions per cycle went up slightly
- Teaser: `-Ofast` achieves a runtime of only 1.5 s
  - `-Ofast` also requires linking with `libmvec`, that is, uses vectorization
  - Optimizing for the architecture with `-march=native` gets it down to 0.5 s

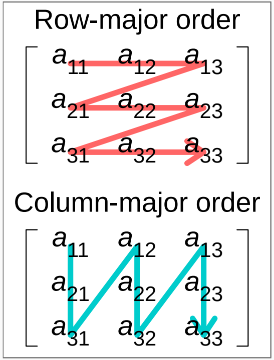
Memory Access

- Memory access and caches important for performance
  - Access to main memory takes approximately 100 ns
  - At 3 GHz (at least) 300 instructions in 100 ns
- Caches can help get data to the processor fast enough
  - Processors will speculatively load data into the cache
  - Typically assume spatial locality, that is, nearby memory will be accessed in the future
- Caches work well if you access data the right way
  - Jumping around randomly will destroy locality



Memory Access...

- Memory access depends on the programming language
  - C stores memory in row-major order
  - Fortran stores memory in column-major order
- Access in the wrong order will reduce performance
  - Has to be considered when porting code
- Combining programming languages can be problematic
  - For instance, using a C library from Fortran



[Cmglee, 2017]

Memory Access…

- C application with row-major matrix
  - Still potential performance problems
- Gray cells contain calculation values
  - Blue cells are loaded into cache
  - CPU-bound given enough math

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

- C application with row-major matrix
  - Still potential performance problems
- Gray cells contain calculation values
  - Blue cells are loaded into cache
  - CPU-bound given enough math
- White cells are empty
  - Values are still loaded into cache
  - Memory-bound due to unused values
- Special data structures for efficient access to sparse matrices

1	2								
11	12	13							
	22	23	24						
		33	34	35					
			44	45	46				
				55	56	57			
					66	67	68		
						77	78	79	
							88	89	90
							99	100	

- Memory interleaving
  - Important for performance
- Array of structures
  - Intuitive representation
  - Potentially bad cache utilization

```c
struct coordinate
{
    double x;
    double y;
    double z;
};

int main (void) {
    struct coordinate e[N] = { 0 };
    double result = 0.0;
    for (int i = 0; i < N; i++)
    {
        result += e[i].x * e[i].y;
    }
    return 0;
}
```

Memory Access…

- Memory interleaving
  - Important for performance
- Array of structures
  - Intuitive representation
  - Potentially bad cache utilization
- Structure of arrays
  - Potentially better for vectorization

```
struct coordinates
{
    double x[N];
    double y[N];
    double z[N];
};

int main (void) {
    struct coordinates e = { 0 };
    double result = 0.0;
    for (int i = 0; i < N; i++)
    {
        result += e.x[i] * e.y[i];
    }
    return 0;
}
```

Outline

Summary

- There is a range of approaches and tools to find performance problems
  - Parallel computers and applications are complex
- Performance measurements require a thought-out approach
  - Single measurements can be more or less random
- Performance optimizations can be done on several levels
  - Code optimizations can be done manually or automatically
- Compilers often can take care of sophisticated optimizations
  - It is important to understand the compiler's capabilities

References

[Bailey, 1991] Bailey, D. (1991). **Twelve ways to fool the masses when giving performance results on parallel computers.** *Supercomputing Review*, pages 54–55.

[Cmglee, 2017] Cmglee (2017). **Illustration of row- and column-major order.** `https://en.wikipedia.org/wiki/File:Row_and_column_major_order.svg`.

[Giu.natale, 2016] Giu.natale (2016). **Example of a naive Roofline model.** `https://en.wikipedia.org/wiki/File:Example_of_a_naive_Roofline_model.svg`.

[GWT-TUD GmbH, 2020] GWT-TUD GmbH (2020). **Vampir.** `https://vampir.eu/`.

Approaches

- O(1)
  - Constant runtime/memory consumption
  - Example: Array access, hash tables
- O(n)
  - Linear runtime/memory consumption
  - Touch every data point once (or a few times)
  - Example: Calculating the sum of a list
- O(n^2)
  - Quadratic runtime/memory consumption
  - Example: (Bad) sorting algorithms

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        result += sin(i) + cos(j);
    }
}
```