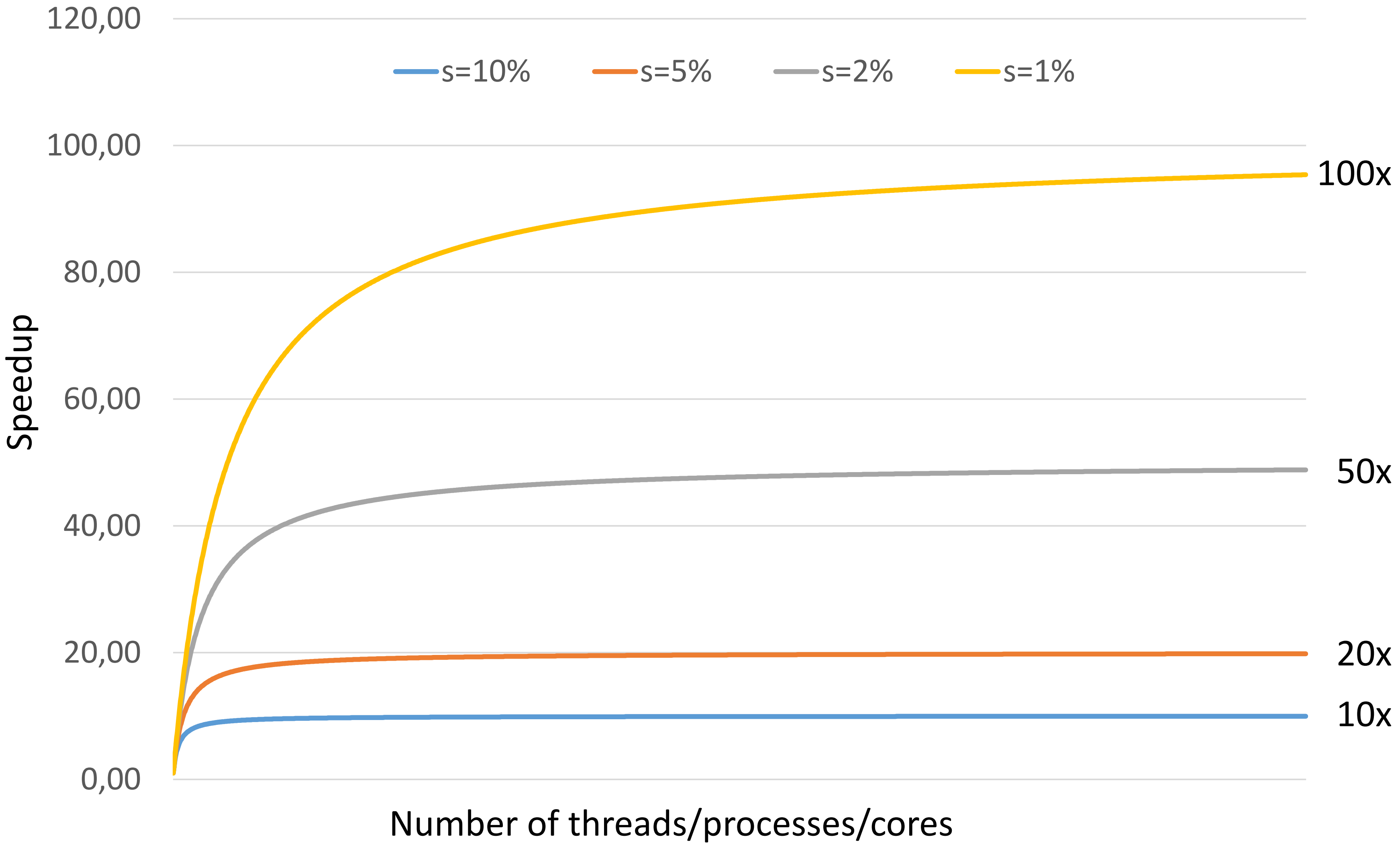
Parallel Algorithms

Review: Amdahl‘s Law



Myth: Parallel Programming is too hard

- **Myth: Parallel programming is too hard for me. There are deadlocks, race conditions, non-deterministic behavior. That’s too much for me!**

- **Well, you are right. Partially.**

- **Structured programming and patterns help tame the problem.**
  → It does not magically go away, but is much less problematic

- **Apply the well-known Software Engineering practices:**
  → Identify a (parallel) pattern
  → Use the pattern library and apply a matching pattern from it

Structured Programming w/ Patterns

- **Patterns are "best practices" for solving specific problems**

  → Known from Software Engineering

- **Patterns can be used to organize your code, leading to algorithms that are more scalable and maintainable.**

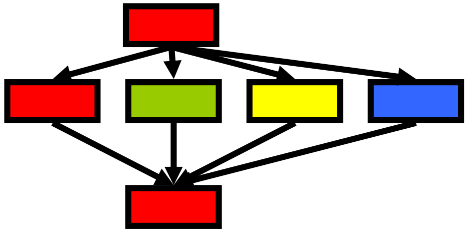
- **A pattern (often) supports a particular "algorithmic structure" with an efficient implementation.**

- **Good parallel programming models support a set of useful parallel patterns with low-overhead implementations.**

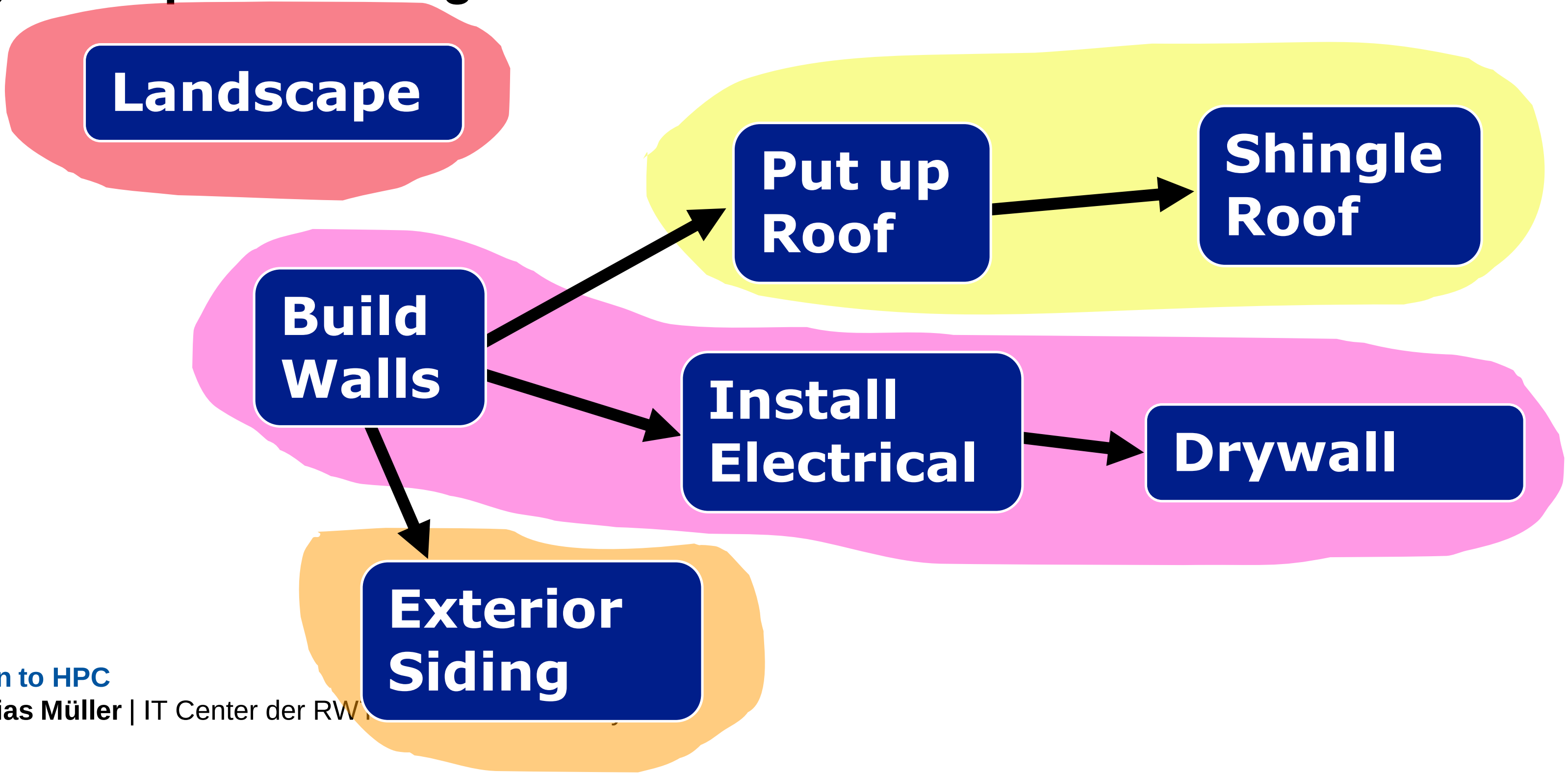
Finding Concurrency

Simultaneous Task Execution

- **Look for tasks that can be executed simultaneously (task parallelism)**

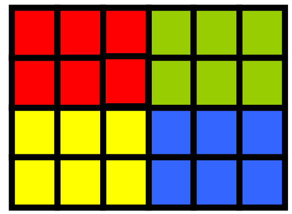


- **Catchy example: building a house**

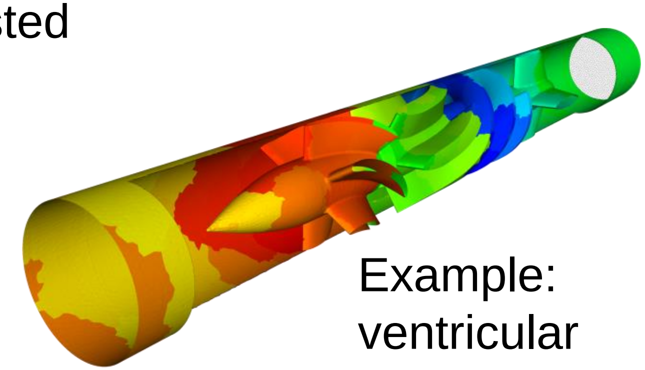


Data Decomposition

- **Decompose data into distinct chunks to be processed independently (data parallelism)**



- **Catchy example: work on a farm**
  - → each worker takes an (unpicked) row and picks the crop
  - → possibly more rows than worker
  - → process continues until all rows have been harvested

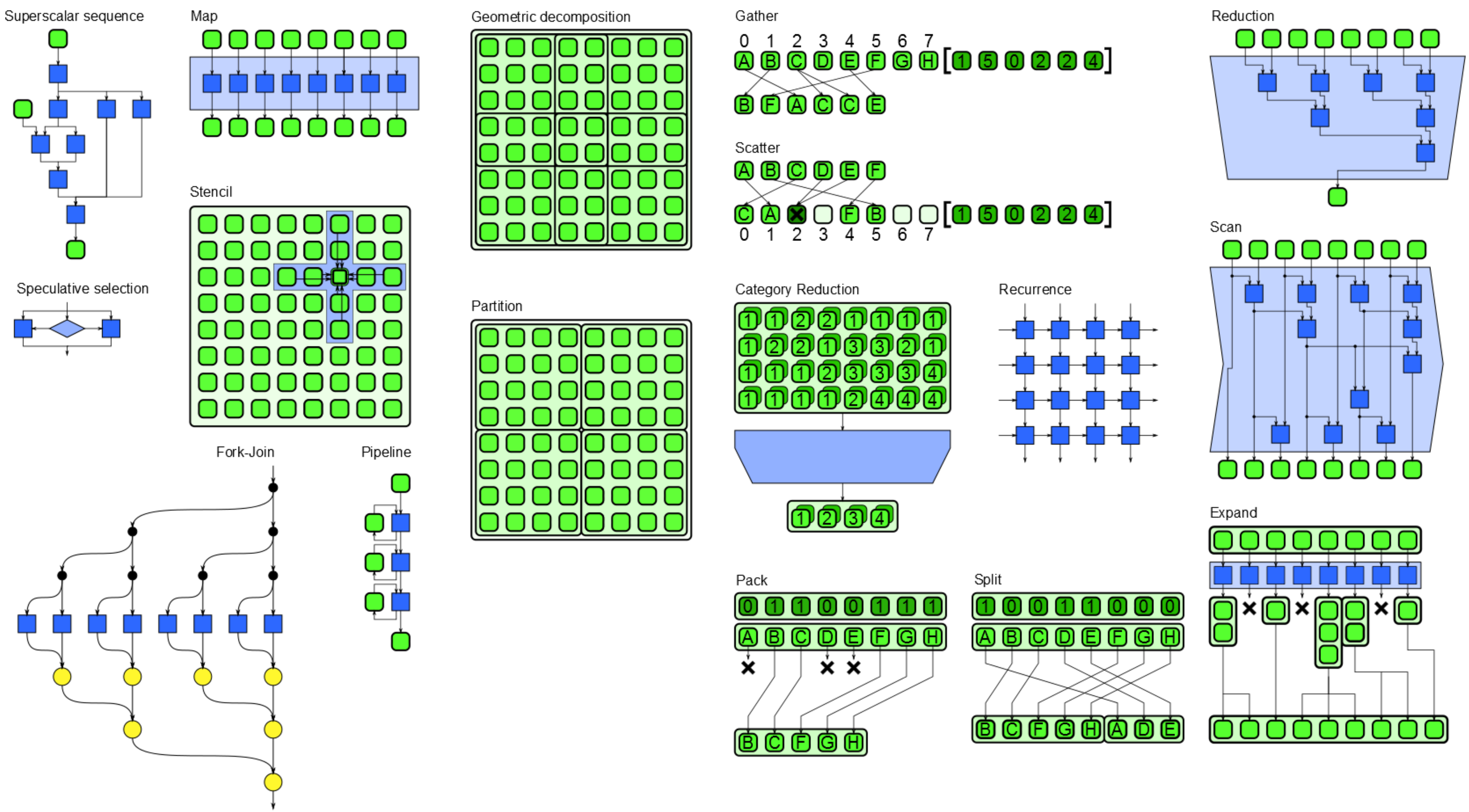


Example: ventricular assist device (VAD)

Structured Parallel Patterns

- **The following additional parallel patterns can be used for "structured parallel programming":**
  - Superscalar sequence
  - Speculative selection
  - Map
  - Recurrence
  - Scan
  - Reduce
  - Pack/expand
  - Fork/join
  - Pipeline
  - Partition
  - Segmentation
  - Stencil
  - Search/match
  - Gather
  - Merge scatter
  - Priority scatter
  - Permutation scatter
  - Atomic scatter

Parallel Patterns: Overview

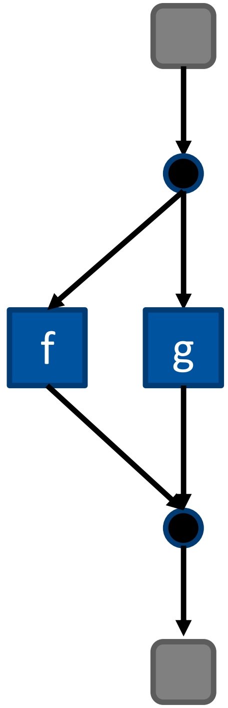


Patterns:

Parallel Control

The Fork-Join Pattern

- **Fundamental pattern to express and exploit parallelism**
  - → control flow *forks* into multiple flows that *join* later
  - → well-versed to go from serial to parallel
  - → can be nested

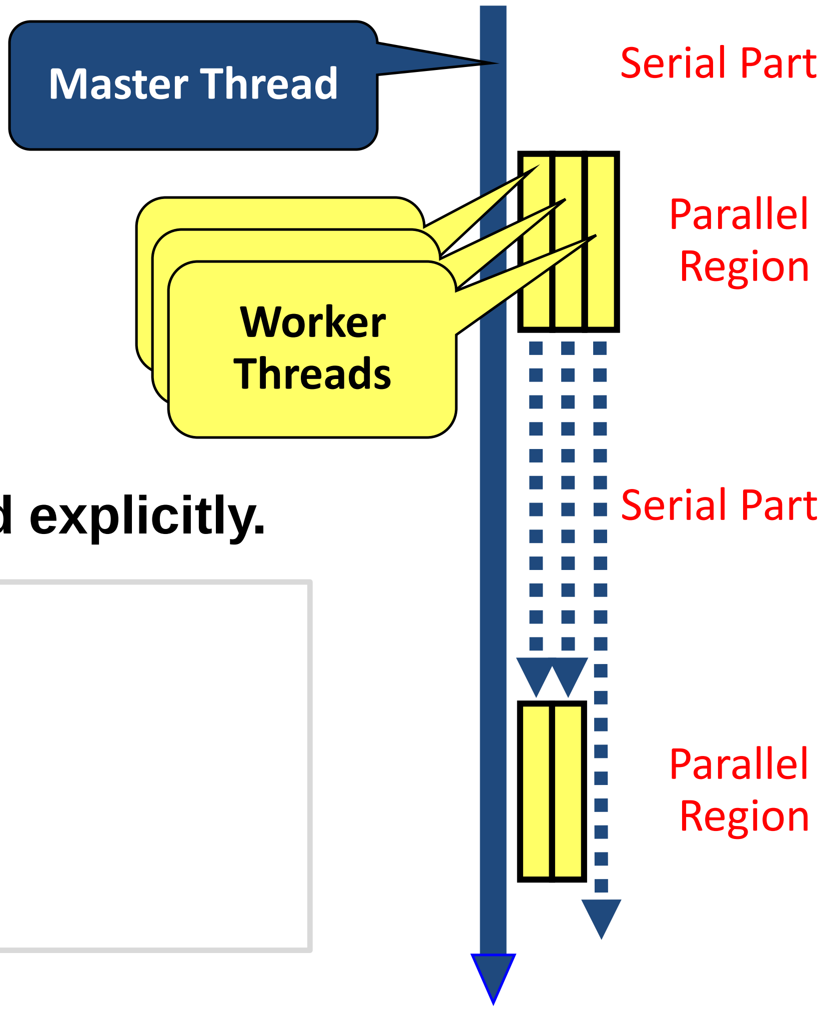


- **As an algorithmic pattern it is a natural fit for parallel divide-and-conquer approaches**

- **(Subtle) Differences among Intel® Cilk Plus and OpenMP*:**
  - → Intel Cilk Plus only uses tasks and an implicit threadpool
  - → OpenMP allows for tasks and threads
  - → OpenMP requires a parallel region to apply any other parallel construct

Fork-Join in OpenMP*

- **OpenMP programs start with just one thread: the *master*.**
- ***Worker* threads are spawned at *parallel regions*, together with the master they form the *team* of threads.**



- **The parallelism has to be expressed explicitly.**

```
#pragma omp parallel
{
   ...
   structured block
   ...
}
```

Fork-Join in Intel® Cilk™ Plus

- **Spawn = asynchronous function call**
  → Arguments to spawned functions are evaluated before fork
- **Example (with optional assignment):**

```
x = cilk_spawn f(*p++);
y = g(*p--);
cilk_sync;
z = x+y;
```

- **Expression spawning possible via lambda expression:**

```
cilk_spawn [&]{
    for(int i=0; i<n; ++i )
        a[i] = 0;
} ();
...
cilk_sync;
```

Fork-Join: Summary

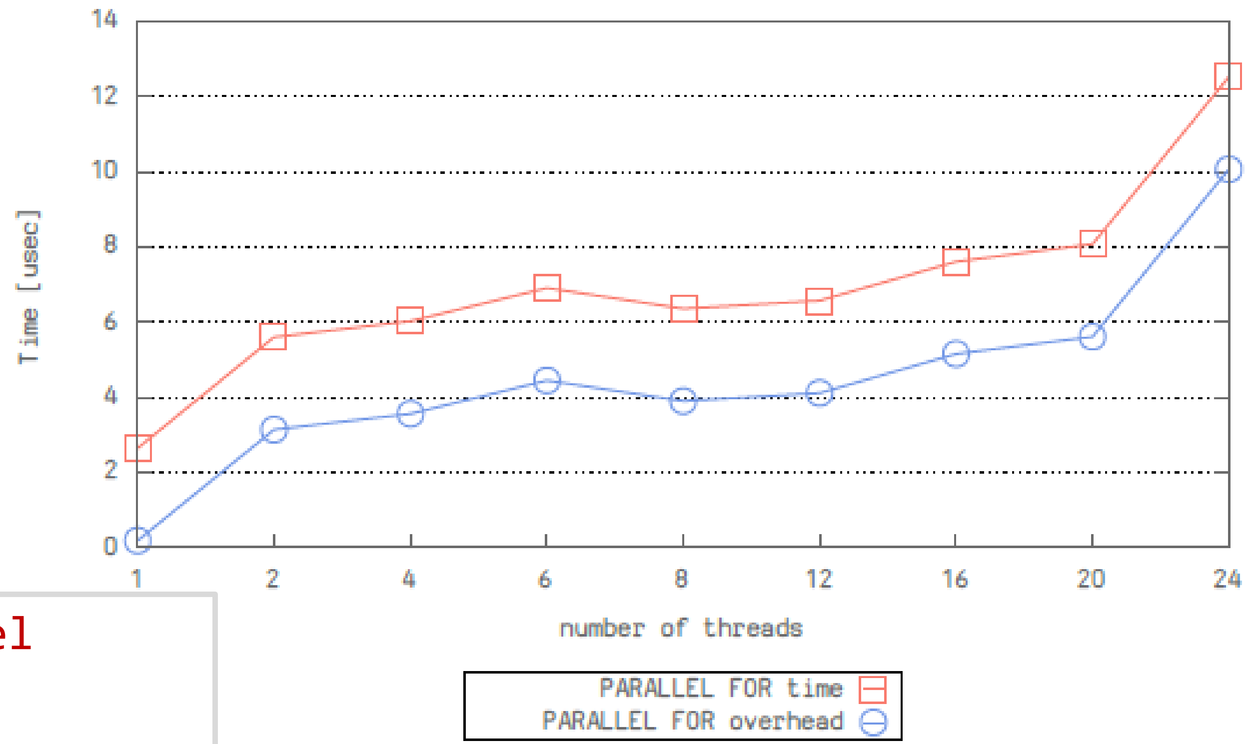
- **How expensive is Fork-Join? OpenMP Overhead Analysis via EPCC micro-benchmarks for a `parallel for` construct:**

Intel® Cilk™ Plus

```
cilk_spawn a();
cilk_spawn b();
c();
cilk_sync();
```

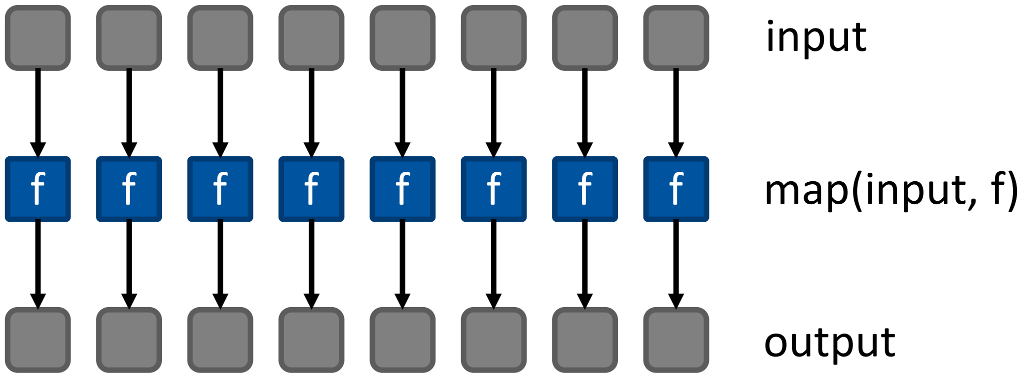
OpenMP*

```
#pragma omp parallel
{
   ...
}
```



The Map Pattern

- Replicate a function over every element of an “index set”
- The index set typically corresponds to the elements of an array



- Some programming languages have support for the map pattern
  → e.g., Fortran, Intel® Cilk™ Plus

Map Pattern: Examples

```
module example
  implicit none
contains
  subroutine example
    integer,parameter :: len=100
    real,dimension(len) :: in=(/(i,i=1,len)/)
    real,dimension(len) :: out
    integer :: i

    ! explicit do loop
    do i=1,size(in)
      output(i)=sqrt(in(i))
    end do

    ! explicit array subsets
    output(:)=sqrt(in(:))

    ! whole array operation
    output=sqrt(in)
    write (*,*), output
  end subroutine
end module
```

Map Pattern: Examples

```fortran
module example
  implicit none
contains
  subroutine example
    integer,parameter :: len=100
    real,dimension(len) :: in=(/(i,i
    real,dimension(len) :: out
    integer :: i

    ! explicit do loop
    do i=1,size(in)
      output(i)=sqrt(in(i))
    end do

    ! explicit array subsets
    output(:)=sqrt(in(:))

    ! whole array operation
    output=sqrt(in)
    write (*,*), output
  end subroutine
end module
```

```cpp
void ex_vector() {
  const size_t len = 100;
  size_t i = 0;
  std::vector<float> in(len);
  std::vector<float> out(len);

  std::generate_n(in.begin(), len, gen);

  std::transform(in.begin(), in.end(),
                 out.begin(), sqrt);

  for (auto it = out.begin();
            it != out.end(); ++it, ++i)
    printf("%f%s",
           *it,
           (i%4==3) ? "\n" : " ");
}
```

Map Pattern: Examples

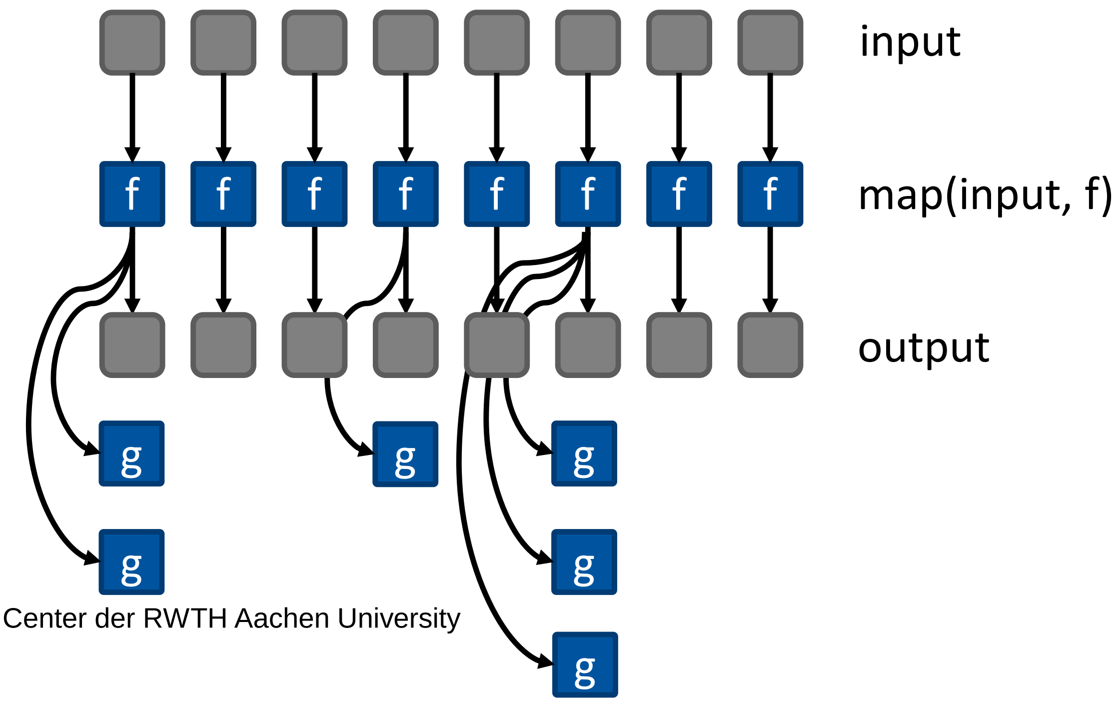
```
module example
  implicit none
contains
  subroutine example
    integer,parameter :: len=100
    real,dimension(le
    real,dimension(le
    integer :: i

    ! explicit do loo
    do i=1,size(in)
      output(i)=sqrt(
    end do

    ! explicit array
    output(:)=sqrt(in

    ! whole array operation
    output=sqrt(in)
    write (*,*), output
  end subroutine
end module
```

```
void ex_vector() {
  const size_t len = 100;
  size_t i = 0;
  std::vector<float> in(len);
  std::vector<float> out(len);

  n.begin(), len, gen);

  .begin(), in.end(),
  t.begin(), sqrt);

  t.begin();
  ut.end(); ++it, ++i)

  ? "\n" : " ");
}
```

Did you see this pattern in any other programming language you have used?

Workpile

- **The Workpile pattern generalizes the Map pattern**
  - → The mapped function can create new parallel work while it is applied
  - → Number of items to work on is not known in advance
- **Example: tree traversal**
  - → Apply Map pattern to root node; create new tasks for a node’s children



Sudoko for Lazy Computer Scientists

- **Lets solve Sudoku puzzles with brute multi-core force**

	6						8	11			15	14			16
15	11				16	14				12			6		
13		9	12					3	16	14		15	11	10	
2		16		11		15	10	1							
	15	11	10			16	2	13	8	9	12				
12	13			4	1	5	6	2	3					11	10
5		6	1	12		9		15	11	10	7	16			3
	2				10		11	6		5			13		9
10	7	15	11	16				12	13						6
9						1			2		16	10			11
1		4	6	9	13			7		11		3	16		
16	14			7		10	15	4	6	1				13	8
11	10		15				16	9	12	13			1	5	4
		12		1	4	6		16				11	10		
		5		8	12	13		10			11	2			14
3	16			10			7			6				12	

(1) Find an empty field

(2) Insert a number

(3) Check Sudoku

(4 a) If invalid:

Delete number,

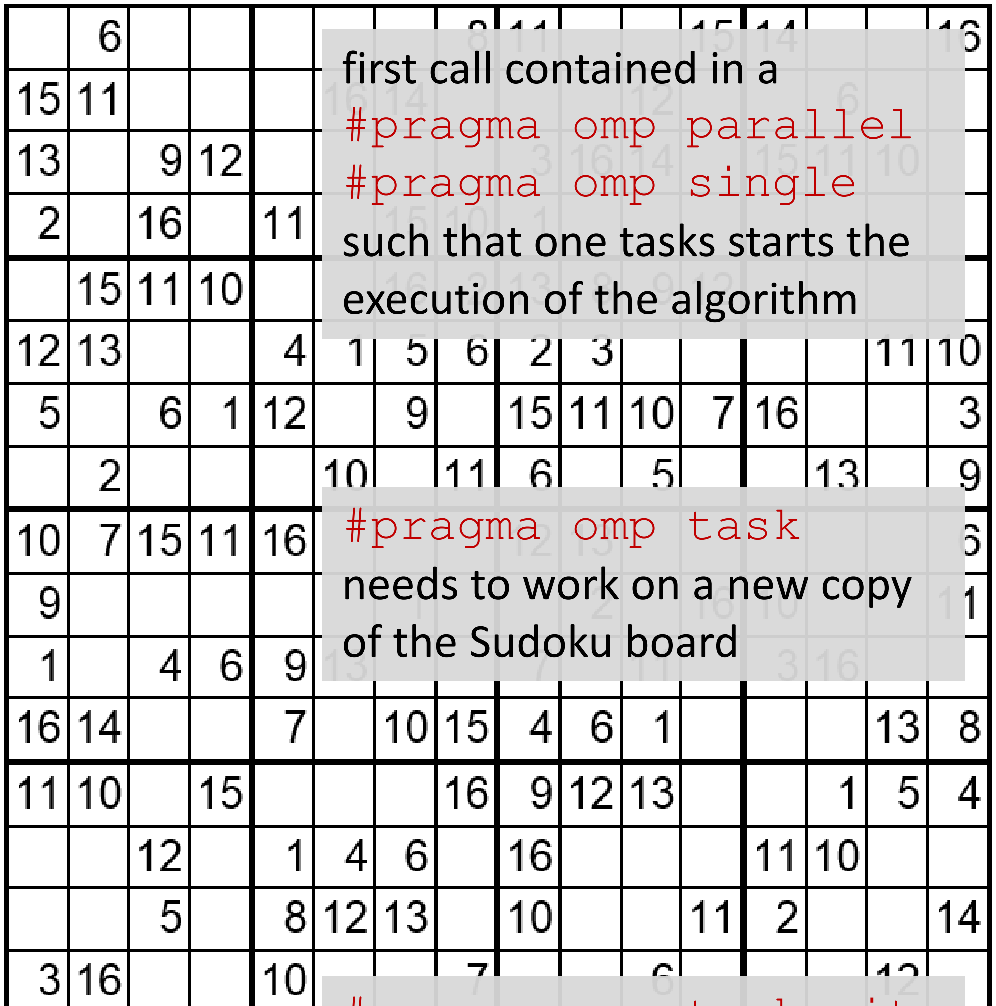
Insert next number

(4 b) If valid:

Go to next field

Parallel Brute-force Sudoku (1/3)

- **This parallel algorithm finds all valid solutions**



first call contained in a

```
#pragma omp parallel
#pragma omp single
```

such that one tasks starts the execution of the algorithm

```
#pragma omp task
```

needs to work on a new copy of the Sudoku board

```
#pragma omp taskwait
```

wait for all child tasks

(1) Search an empty field

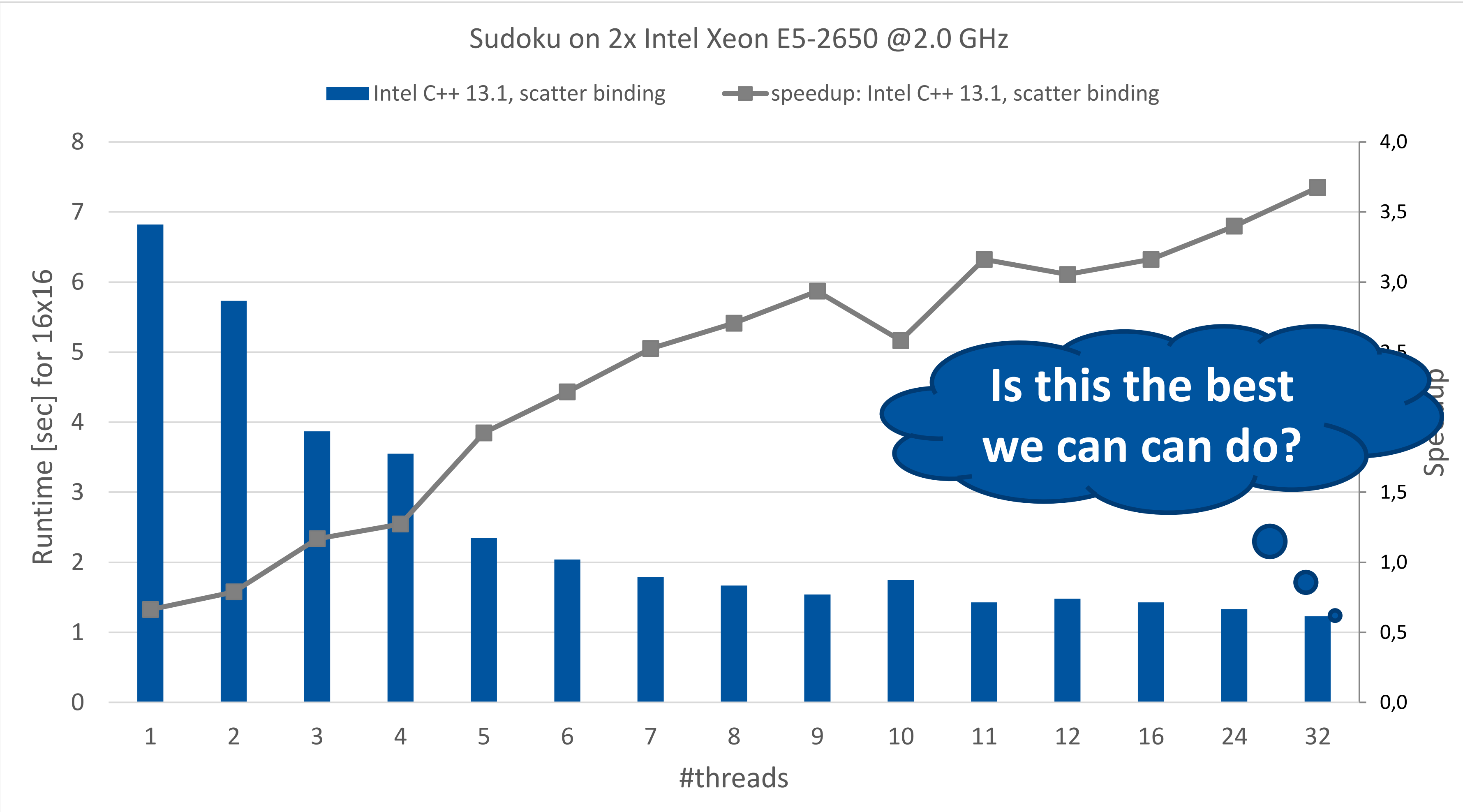
(2) Insert a number

(3) Check Sudoku

(4 a) If invalid:
Delete number,
Insert next number

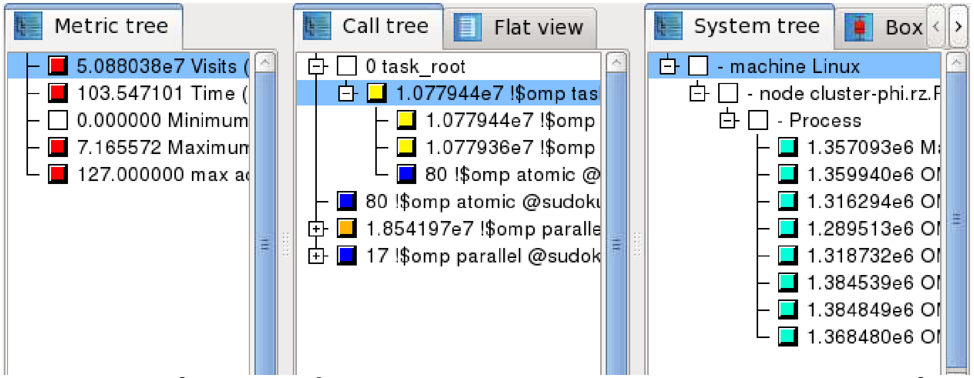
(4 b) If valid:
Go to next field

Performance Evaluation

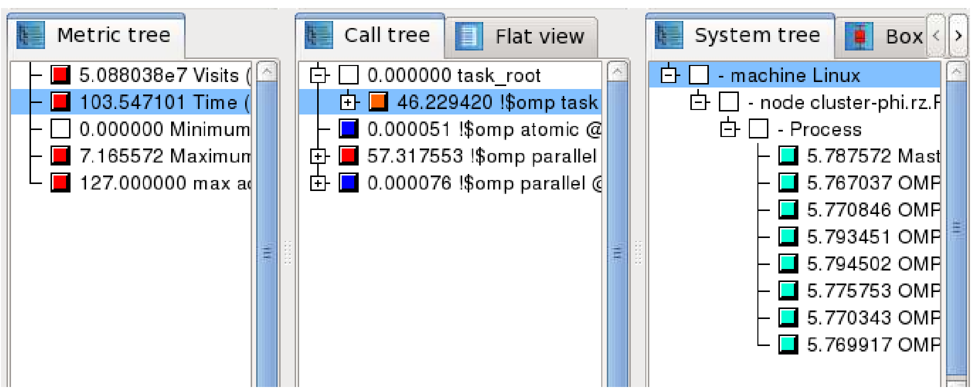


Performance Analysis

Event-based profiling gives a good overview :



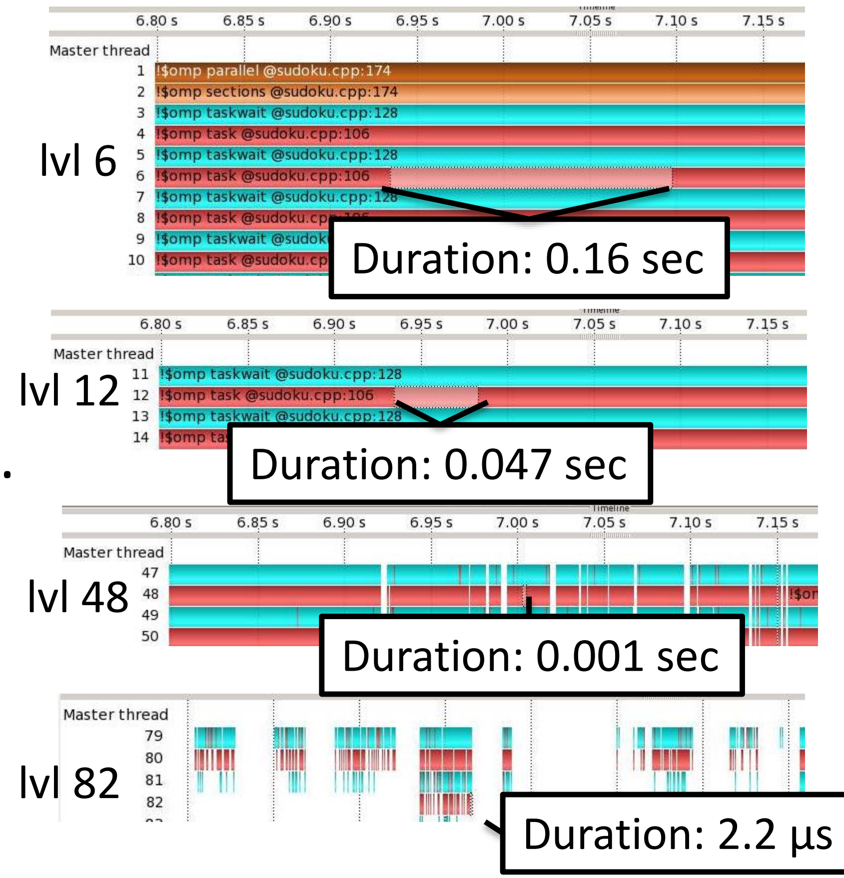
Every thread is executing ~1.3m tasks…



… in ~5.7 seconds.

=> average duration of a task is ~4.4 µs

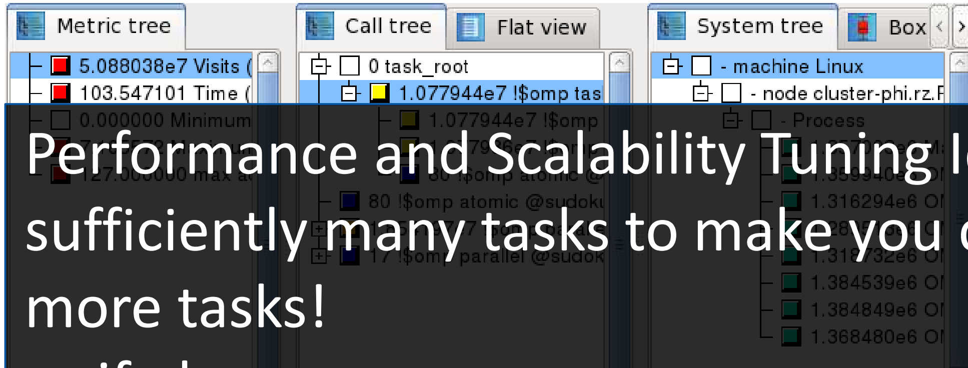
Tracing gives more details:



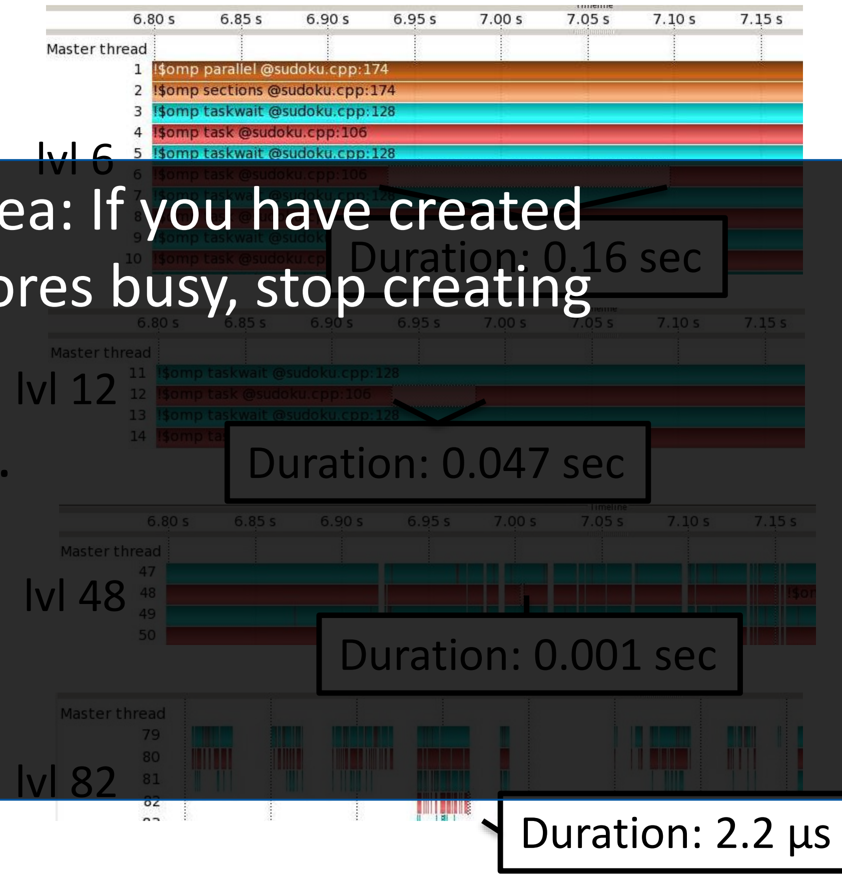
Tasks get much smaller down the call-stack.

Performance Analysis

Event-based profiling gives a good overview :



Tracing gives more details:

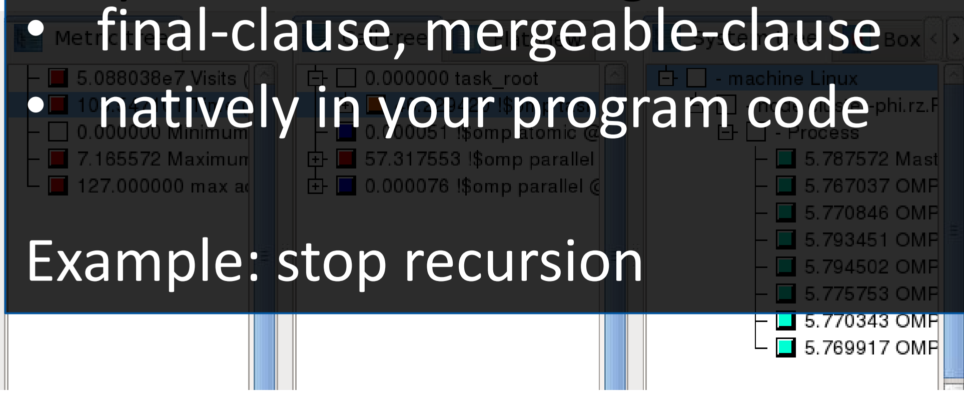


Performance and Scalability Tuning Idea: If you have created sufficiently many tasks to make you cores busy, stop creating more tasks!

- if-clause
- final-clause, mergeable-clause
- natively in your program code

Example: stop recursion

Every thread is executing ~1.3m tasks…

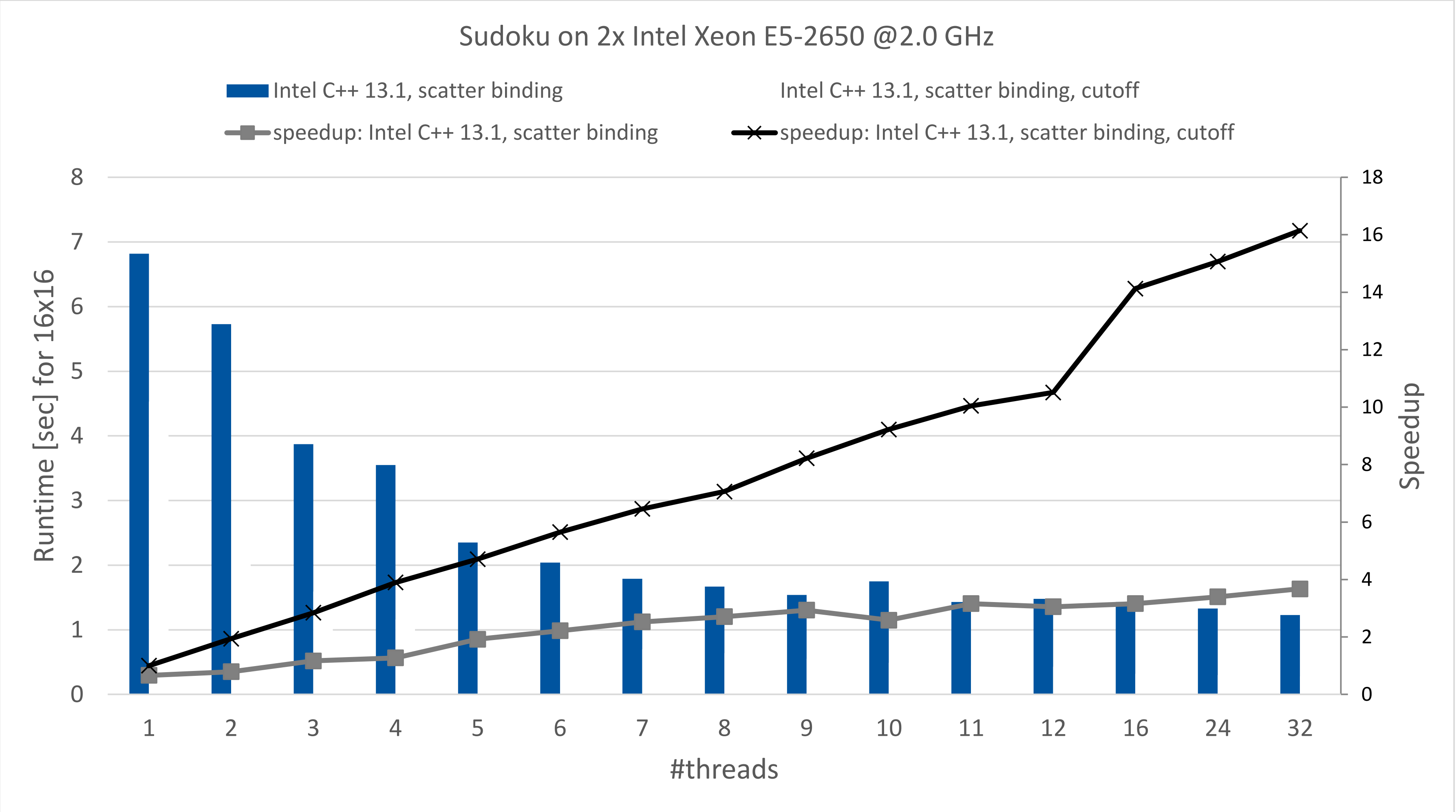


… in ~5.7 seconds.

=> average duration of a task is ~4.4 µs

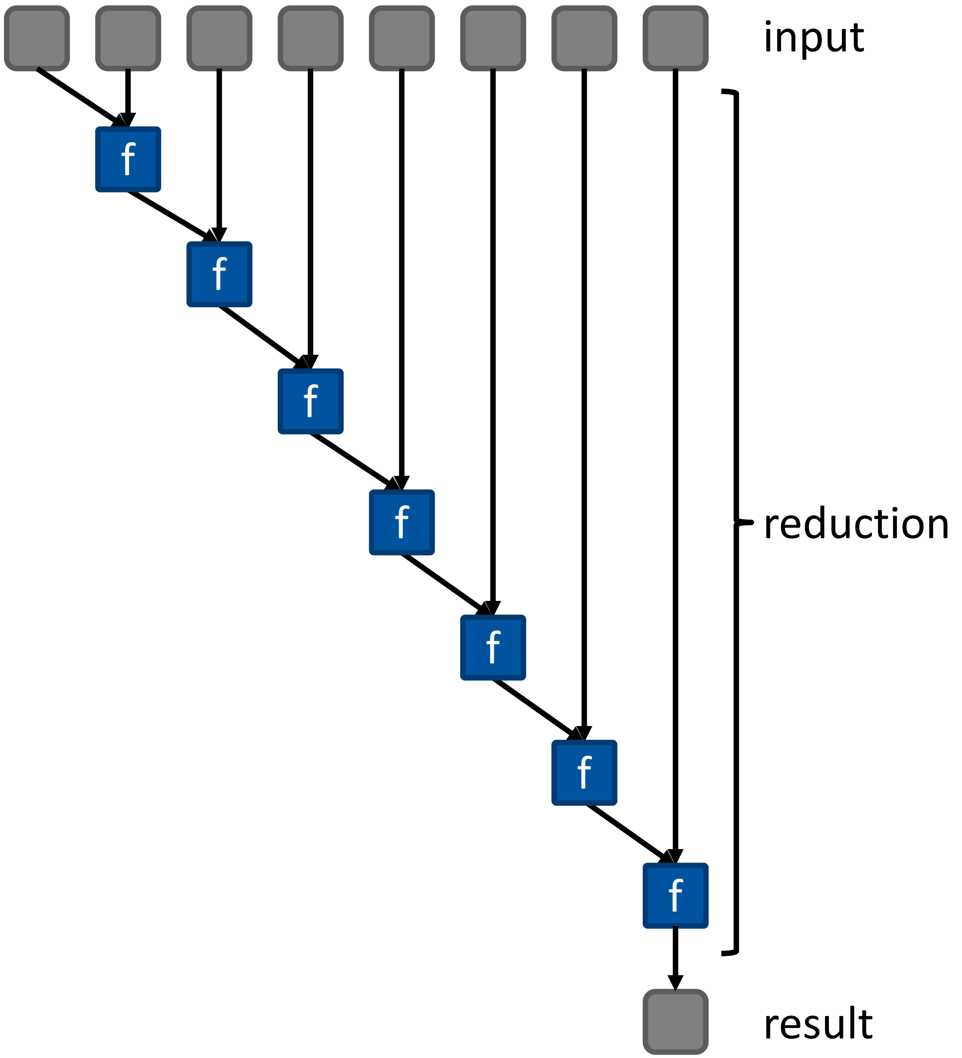
Tasks get much smaller down the call-stack.

Performance Evaluation



The Reduction Pattern

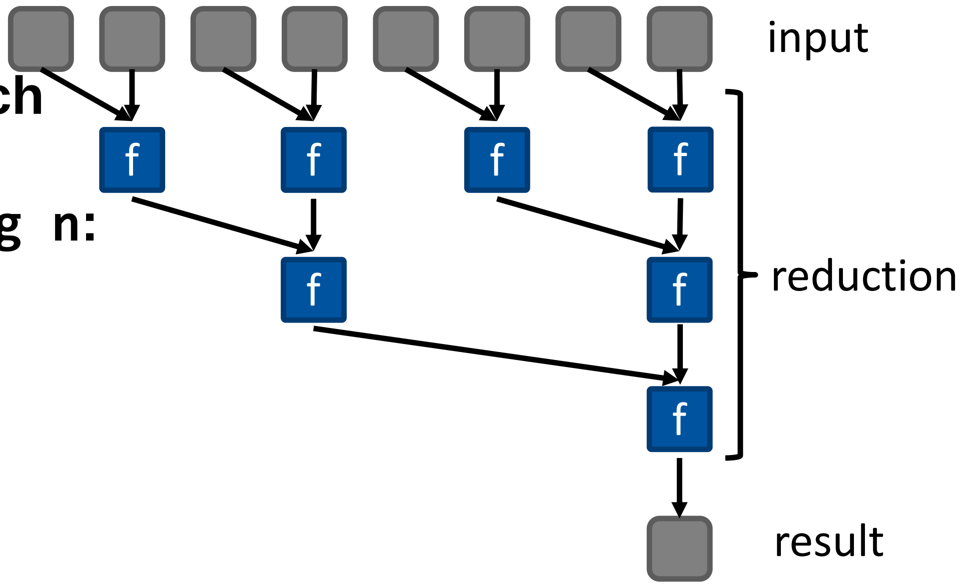
- **Combines every element in a collection into a single result**
  - → employs an associative combiner function
  - → consequently, different ordering are possible



The Reduction Pattern

- **Combines every element in a collection into a single result**
  - → employs an associative combiner function
  - → consequently, different ordering are possible

- **Natural parallelization approach is to perform a tree reduction, delivering a speedup of `n / lg n`:**



Reduction: Summary

- **Reduction allows for logarithmic runtime implementation**

Intel® Cilk™ Plus

```
cilk::reducer<sum_monoid> sum;
cilk_for (int i = 1; i < 100; i++)
   *sum += i;
std::cout << "sum 1 .. 99 is " << *sum << std::endl;
```

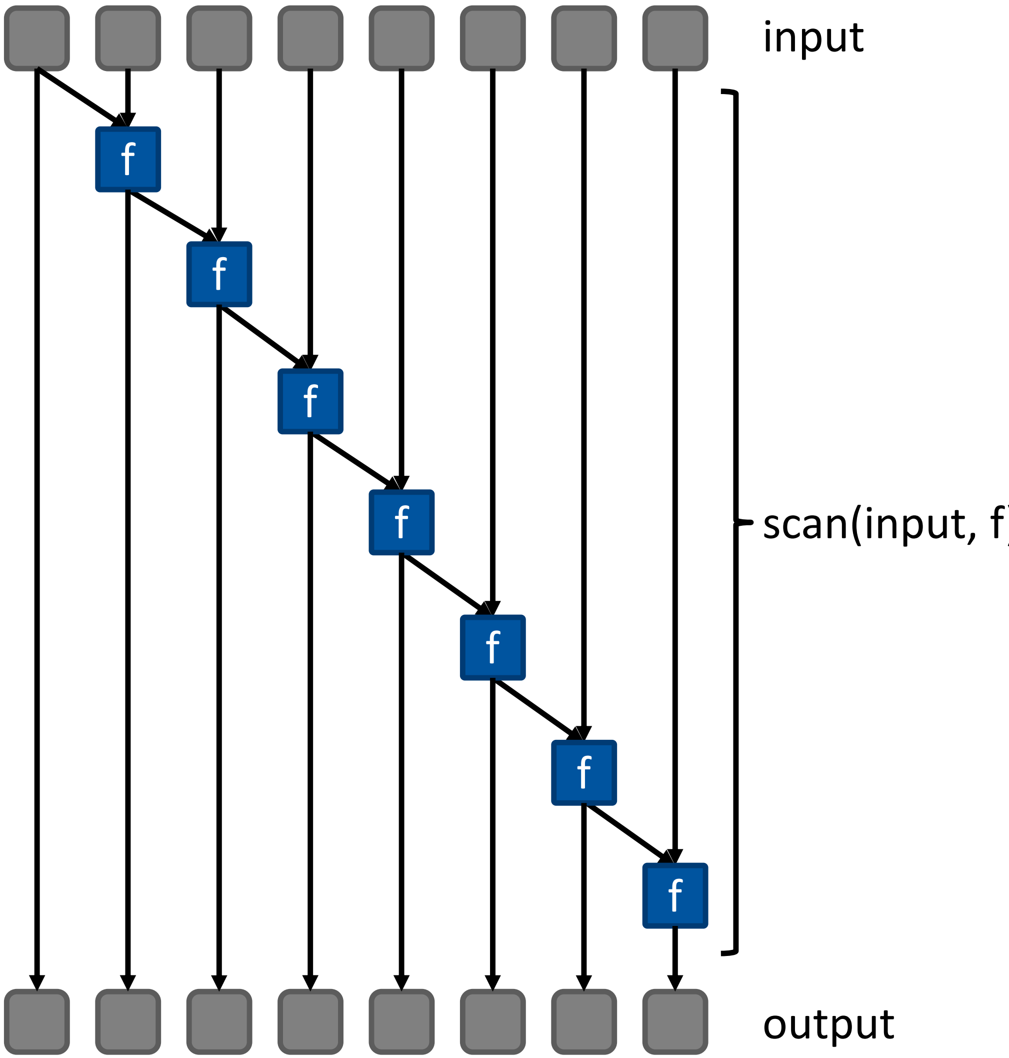
OpenMP*

```
double sum = 0.0;
#pragma omp parallel for reduction(+:sum)
for (int i = 1; i < 100; i++)
  sum += i;
```

The Scan Pattern

- **Produce all partial reductions of an input index set**
- **Function *f* needs to be associative**

```
template<typename T, typename C>
void incl_scan(size_t n,
               const T in[],
               T out[],
               C f,
               T initial)
{
  for (size_t i=0; i<n; ++i) {
    initial = f(initial, in[i]);
    out[i] = initial;
  }
}
```



Thinking about Scan

Simple example:

```
for (int i=1; i<N; i++)
{
   a[i] = f(a[i-1],b[i]);
}
```

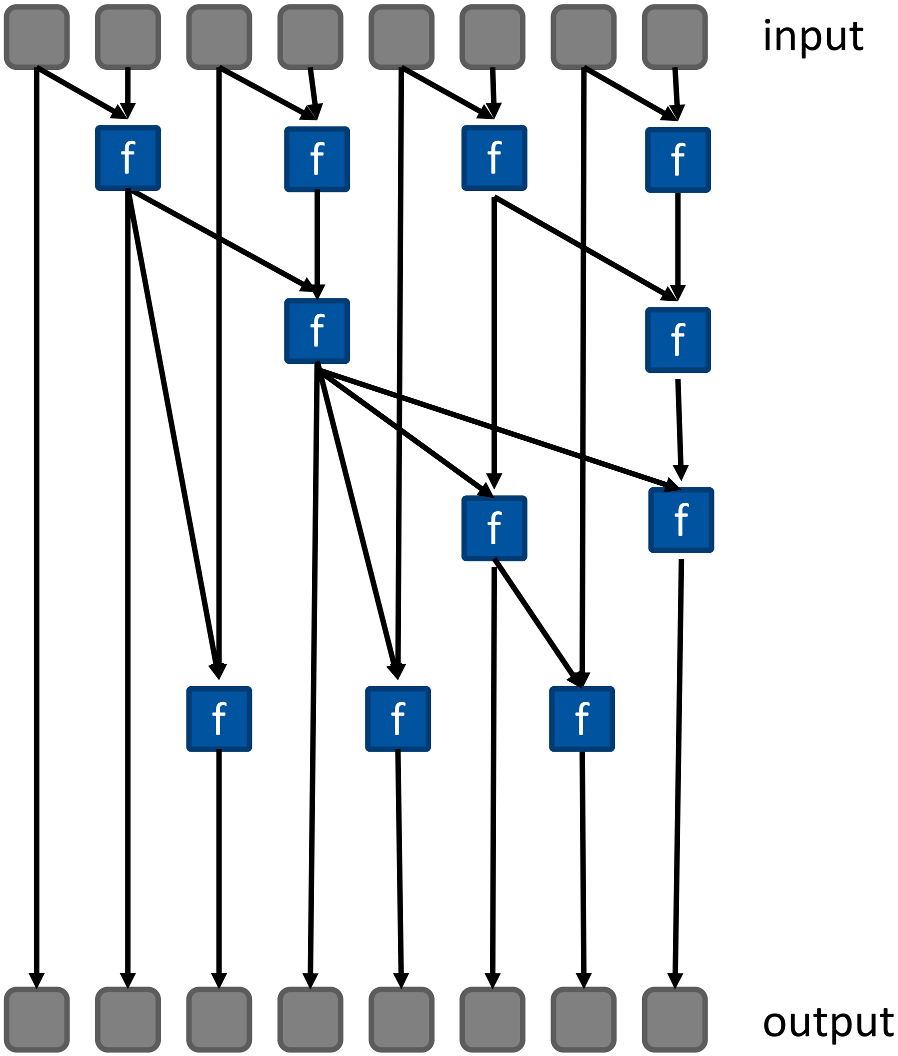
Associativity:

```
F(a, F(b, c)) = F(F(a, b), c)
```

Tree-wandering:

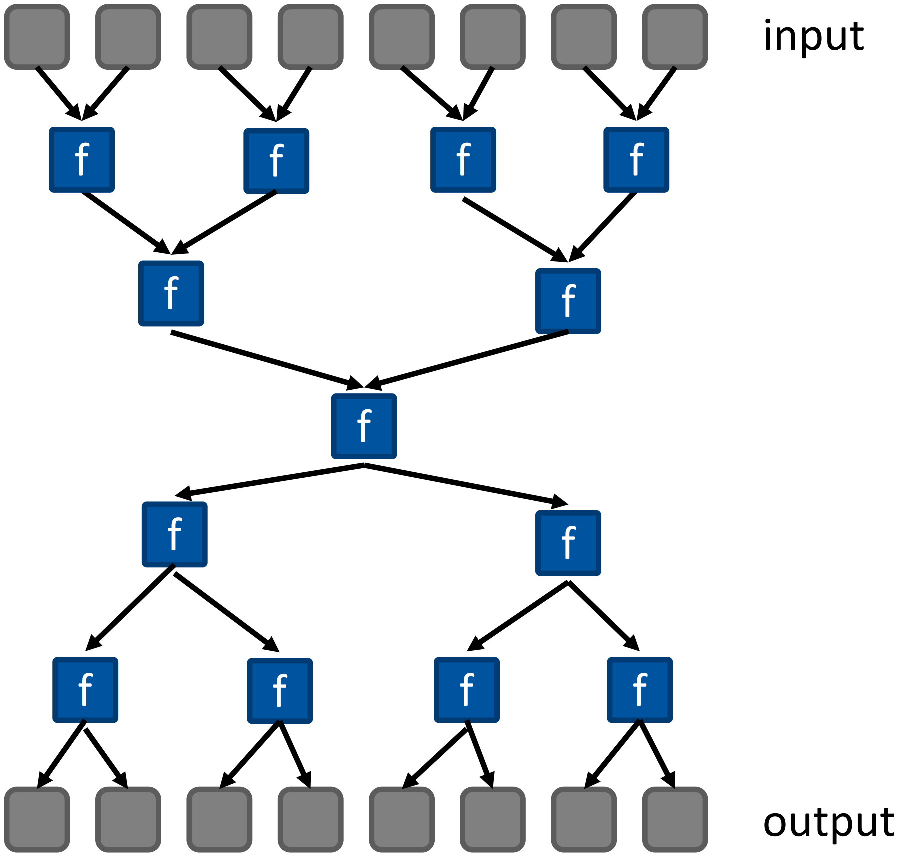
```
O(N * log N)
```

The Scan Pattern: Implementation 1



The Scan Pattern: Implementation 2

Upward sweep:
- Get values L and R from left and right child
- Save L in local variable Mine
- Compute Tmp = L + R and pass to parent

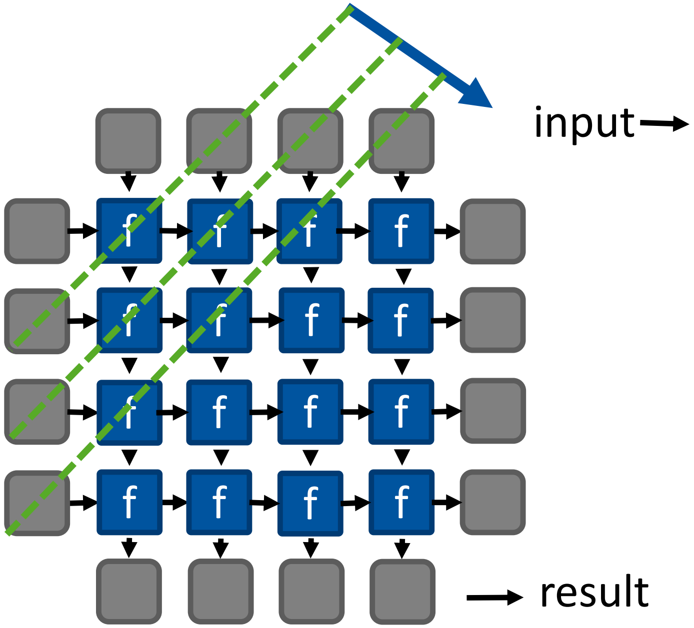


Downward sweep:
- Get value Tmp from parent
- Send Tmp to left child
- Send Tmp + Mine to right child

The Recurrence Pattern

- **In *map*, all loop iterations are independent. In *recurrence*, there are some dependencies:**
  - → 1D recurrence: computation in the element is associative
    - →has already been discussed with the *scan* pattern
  - → n-D recurrence: nested loop body

```
int i,j;
for (i=1; i<N; ++i)
  for (j=1; j<N; ++j)
    b[i][j] = f(b[i-1][j],
                b[i][j-1], a[i][j])
```



- **Leslie Lamport‘s theorem: hyperplane separation can be found if the dependencies are constant offsets**

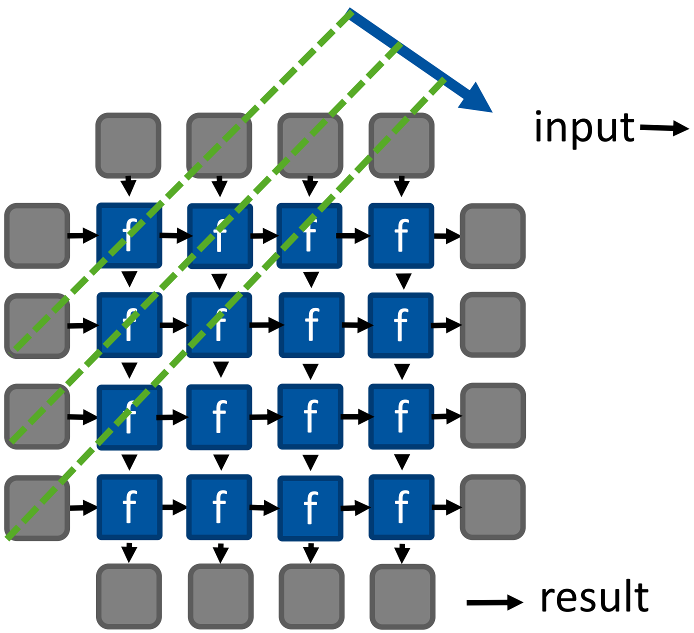
Recurrence: Example

- **Loop iteration space to hyperplane separation**

```c
int i,j;
for (i=1; i<N; ++i)
  for (j=1; j<N; ++j)
    b[i][j] = f(b[i-1][j],
                b[i][j-1], a[i][j])
```

OpenMP*

```c
int i,j;
#pragma omp parallel private(i,j)
for (i=1; i<N; ++i)
#pragma omp for
  for (j=i+1; j<i+N; ++j)
    b[i][j-i] = f(b[i-1][j-i],
                  b[i][j-i-1], a[i][j])
```



Organization

- **Schedule**

KW	Date	Type			Date	Type	
03	16.01.2017	V	**Parallel Algorithms I**		17.01.2017	V	**Parallel Algorithms II**
04	23.01.2017	V	**GPGPU**		24.01.2017	V	**GPGPU/ Manycore Architectures**
05	30.01.2017	Ü	**Hybrid Programming/ Accelerators**		31.01.2017	V	**Energy Efficiency I**
06	06.02.2017	V	**Energy Efficiency II/ Summary/ Outlook**		07.02.2017	F	**Question Time**

KW: week, V: lecture, Ü: exercise, F: Fragenstunde

- **Exams (February 13th and April 3rd 12:15 – 15:45)**

  → 120 minutes, no additional materials allowed

Online Registration for Seminars and Practical Courses (Praktika) in Summer Term 2017

Who?

Students of:
- Master Courses
- Bachelor Informatik (~~Pro~~Seminar!)

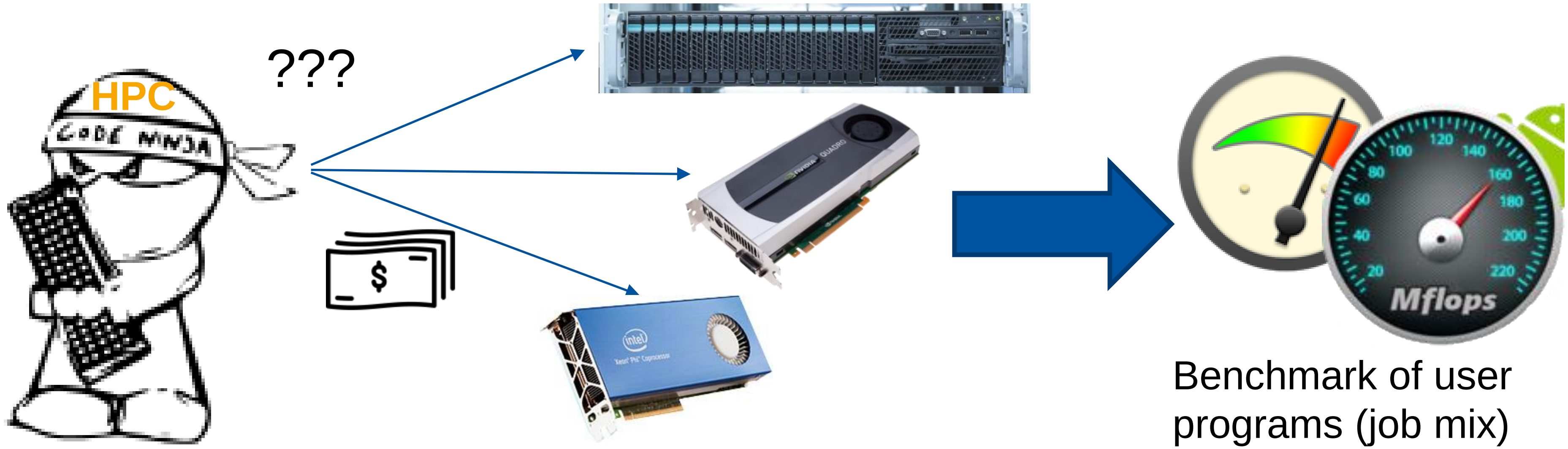
Where?

www.graphics.rwth-aachen.de/apse

When?

13.01.2017 – 29.01.2017

Master Thesis: Representation of a User Jobmix as a Parameterization of Standard Benchmarks



- **Goal: Representation of the investigated programs by means of parametrization of standard benchmarks**
  - → Characterization of the job mix by a set of metrics
  - → Comparison to common benchmarks with known characteristics
  - → Fitting a weighted mix of the benchmarks to the programs of the job mix

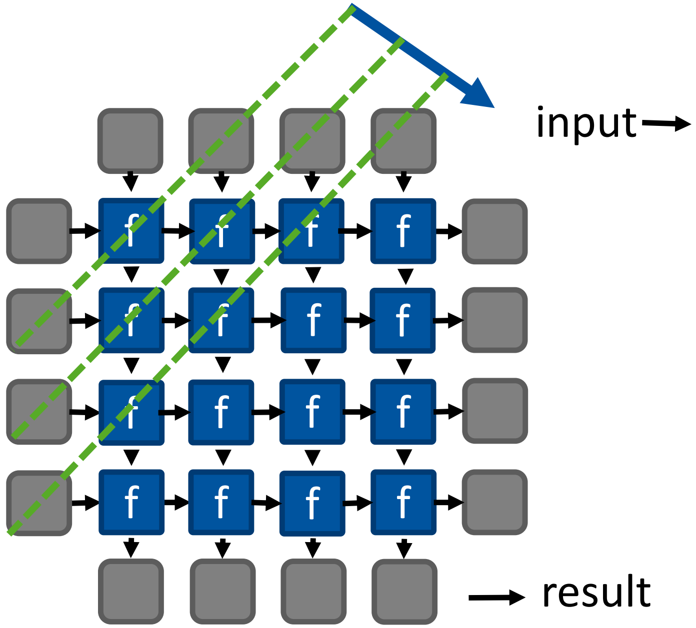
Recurrence: Example

- **Loop iteration space to hyperplane separation**

```
int i,j;
for (i=1; i<N; ++i)
  for (j=1; j<N; ++j)
    b[i][j] = f(b[i-1][j],
                b[i][j-1], a[i][j])
```

OpenMP*

```
int i,j;
#pragma omp parallel private(i,j)
for (i=1; i<N; ++i)
#pragma omp for
  for (j=i+1; j<i+N; ++j)
    b[i][j-i] = f(b[i-1][j-i],
                b[i][j-i-1], a[i][j])
```



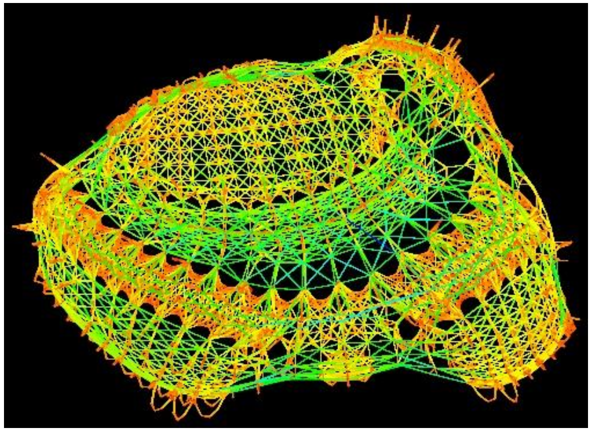
Case Study:

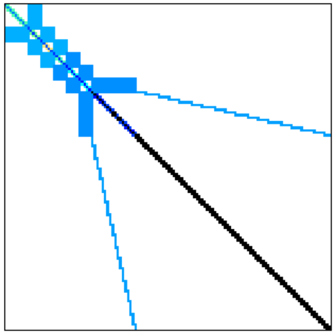
CG

Case Study: CG

Sparse Linear Algebra

- → Sparse Linear Equation Systems occur in many scientific disciplines.
- → Sparse matrix-vector multiplications (SpMxV) are the dominant part in many iterative solvers (like the CG) for such systems.
- → number of non-zeros << n*n





Beijing Botanical Garden

Oben Rechts:	Orginal Gebäude
Unten Rechts:	Modell
Unten Links:	Matrix

(Quelle: Beijing Botanical Garden and University of Florida, Sparse Matrix Collection)

Case Study: CG

- $A = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 2 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 4 & 0 & 4 & 4 \end{pmatrix}$

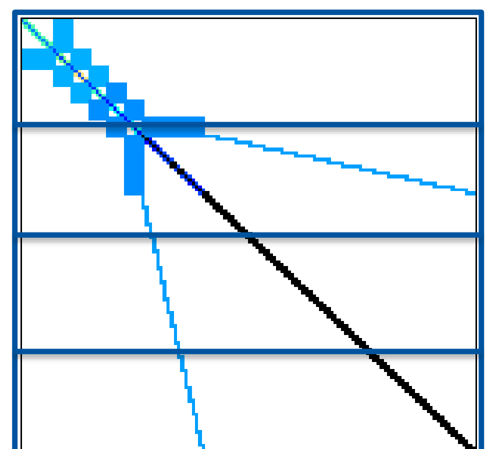
```
for (i = 0; i < A.num_rows; i++){
  sum = 0.0;
  for (nz=A.row[i]; nz<A.row[i+1]; ++nz){
    sum+= A.value[nz]*x[A.index[nz]];
  }
  y[i] = sum;
}
```

$$\boxed{\vec{y} = A * \vec{x}}$$

- **Format: compressed row storage**

- **store all values and columns in arrays (length nnz)**
- **store beginning of a new row in a third array (length n+1)**

value:	1	2	2	3	4	4	4
index:	0	0	1	2	0	2	3
row:	0	1	3	4	7		



Case Study CG: Step 1

Hotspot analysis of the serial code:

Call Stack	CPU Time: Total by Utilization (Idle, Poor, Ok, Ideal, Over)	
cg	46.7%	
matvec	40.8%	1.
xpay	1.4%	2.
axpy	1.4%	2.
vectorDot	1.2%	3.
axpy	1.1%	2.
vectorDot	0.6%	3.

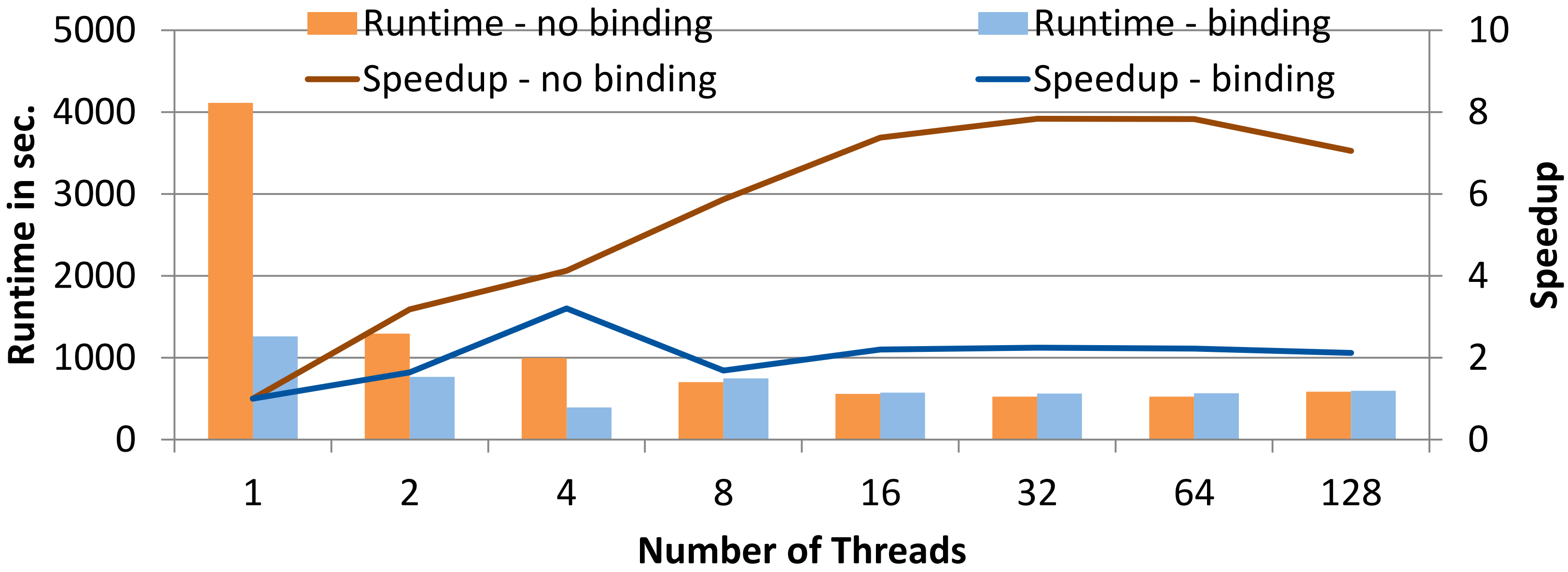
Hotspots are:

1. matrix-vector multiplication
2. scaled vector additions
3. dot product

Case Study CG: Step 1

Tuning:

- **parallelize all hotspots with a parallel for construct**
- **use a reduction for the dot-product**
- **activate thread binding**



Case Study CG: Step 2

Hotspot analysis of naive parallel version:

Event Name
MEM_UNCORE_RETIRED.LOCAL_DRAM_AND_REMOTE_CACHE_HIT
MEM_UNCORE_RETIRED.REMOTE_DRAM

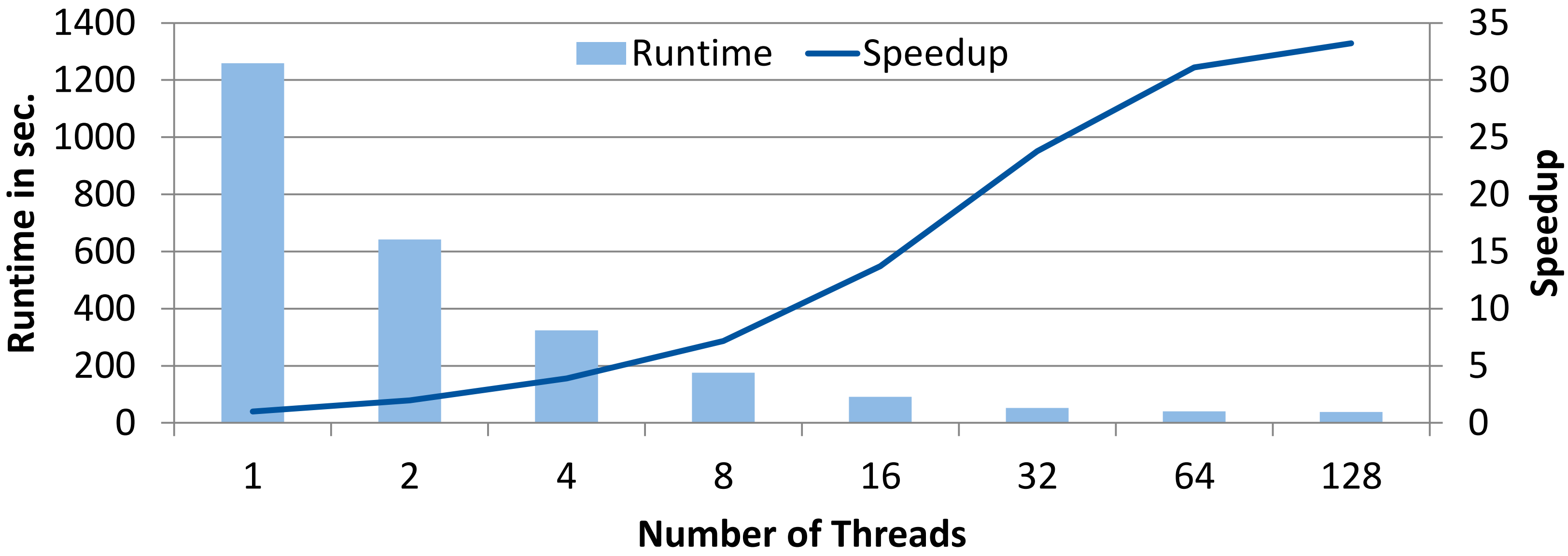
A lot of remote accesses occur in nearly all places.

Source	MEM_UNCORE_RETIRED.LOCAL_...	MEM_UNCORE_RETIRED.REMOTE...
`void matvec(const int n, const int`		
`int i,j;`		
`#pragma omp parallel for private(j)`	20,000	0
`for(i=0; i<n; i++){`	0	0
`y[i]=0;`	0	0
`for(j=ptr[i]; j<ptr[i+1]; j`	6,740,000	3,720,000
`y[i]+=value[j]*x[index[`	17,580,000	6,680,000
`}`		
`}`		

Case Study CG: Step 2

Tuning:

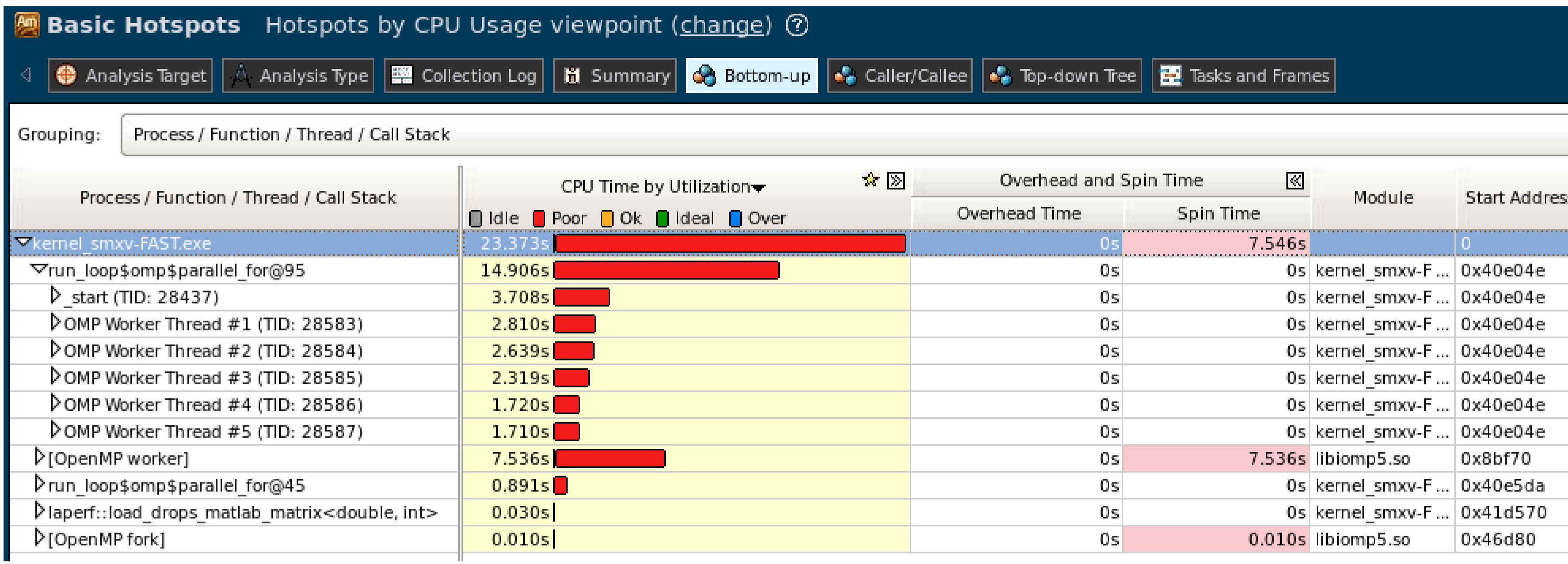
- **Initialize the data in parallel**
- **Add parallel for constructs to all initialization loops**



- **Scalability improved a lot by this tuning on the large machine.**

Load Imbalance in VTune

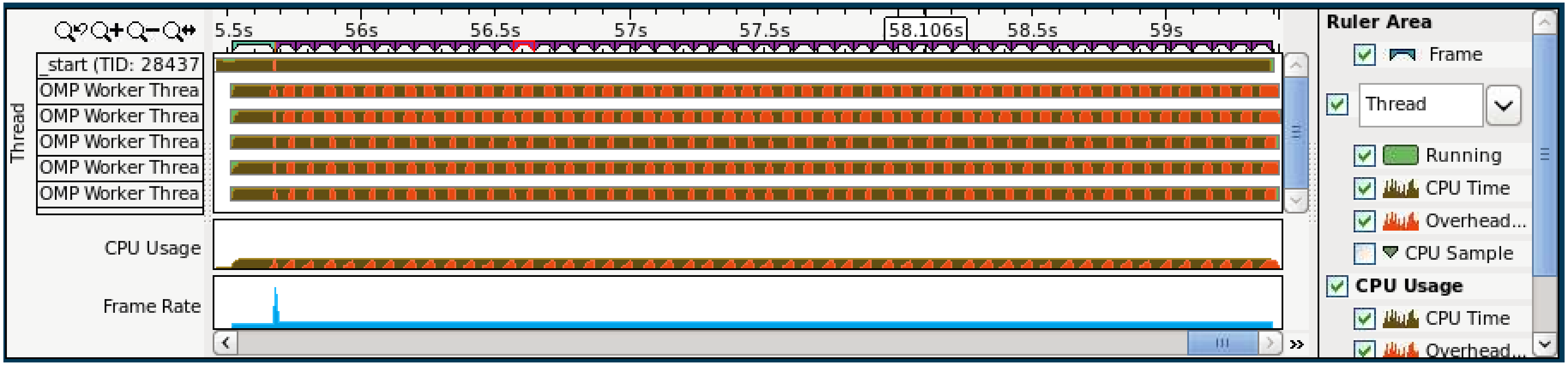
- **Grouping execution time of parallel regions by threads helps to detect load imbalance.**
- **Significant potions of Spin Time also indicate load balance problems.**
- **Different loop schedules might help to avoid these problems.**



Process / Function / Thread / Call Stack	CPU Time by Utilization	Overhead and Spin Time: Overhead Time	Overhead and Spin Time: Spin Time	Module	Start Address
kernel_smxv-FAST.exe	23.373s	0s	7.546s		0
run_loopompparallel_for@95	14.906s	0s	0s	kernel_smxv-F ...	0x40e04e
_start (TID: 28437)	3.708s	0s	0s	kernel_smxv-F ...	0x40e04e
OMP Worker Thread #1 (TID: 28583)	2.810s	0s	0s	kernel_smxv-F ...	0x40e04e
OMP Worker Thread #2 (TID: 28584)	2.639s	0s	0s	kernel_smxv-F ...	0x40e04e
OMP Worker Thread #3 (TID: 28585)	2.319s	0s	0s	kernel_smxv-F ...	0x40e04e
OMP Worker Thread #4 (TID: 28586)	1.720s	0s	0s	kernel_smxv-F ...	0x40e04e
OMP Worker Thread #5 (TID: 28587)	1.710s	0s	0s	kernel_smxv-F ...	0x40e04e
[OpenMP worker]	7.536s	0s	7.536s	libiomp5.so	0x8bf70
run_loopompparallel_for@45	0.891s	0s	0s	kernel_smxv-F ...	0x40e5da
laperf::load_drops_matlab_matrix<double, int>	0.030s	0s	0s	kernel_smxv-F ...	0x41d570
[OpenMP fork]	0.010s	0s	0.010s	libiomp5.so	0x46d80

Load Imbalance in VTune

- The Timeline can help to investigate the problem further.



- Zooming in, e.g. to one iteration is also possible.

Case Study CG: Step 3

- **Analyzing load imbalance in the concurrency view:**

So.. Line	Source	CPU Time: Total by... Idle Poor Ok Id	Ove... and...
49	void matvec(const int n, const int nnz, .		
50	int i,j;		
51	#pragma omp parallel for private(j)	22.462s	10.612s
52	for(i=0; i<n; i++){	0.050s	0s
53	y[i]=0;	0.060s	0s
54	for(j=ptr[i]; j<ptr[i+1]; j++){	1.741s	0s
55	y[i]+=value[j]*x[index[j]];	9.998s	0s

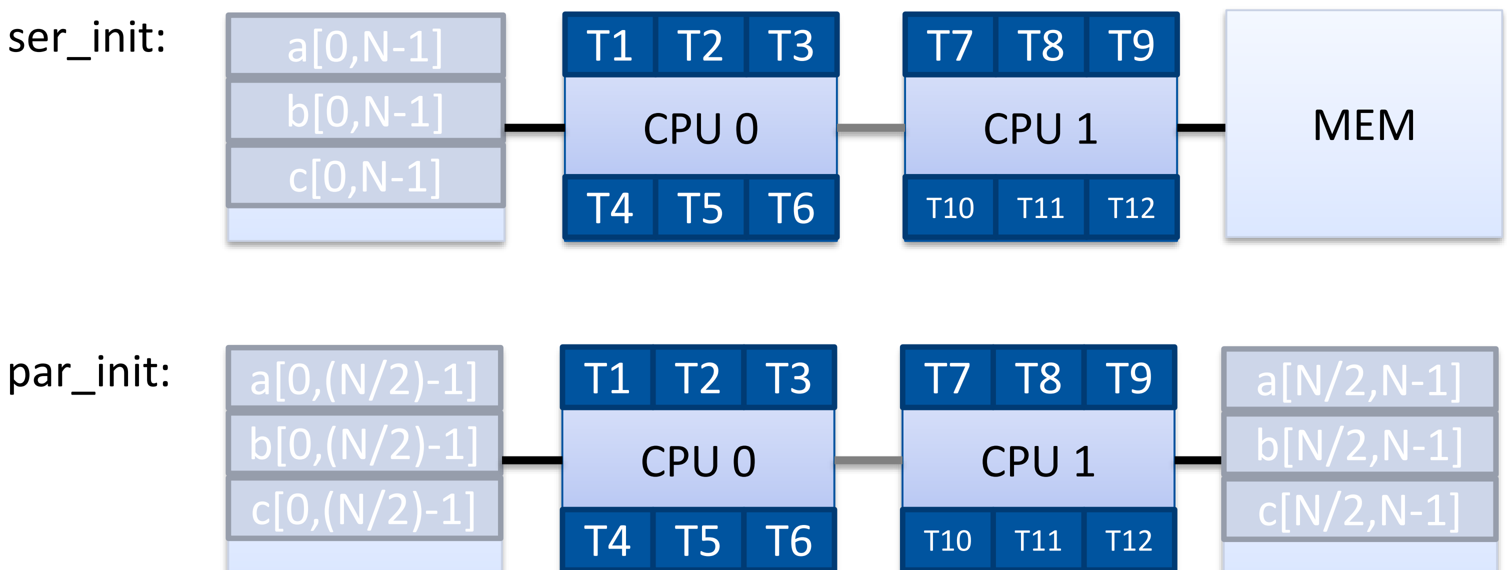
- **10 seconds out of ~35 seconds are overhead time**
- **other parallel regions which are called the same amount of time only produce 1 second of overhead**

Counters for Remote Traffic

- **Stream example ($\vec{a} = \vec{b} + s * \vec{c}$) with and without parallel initialization.**

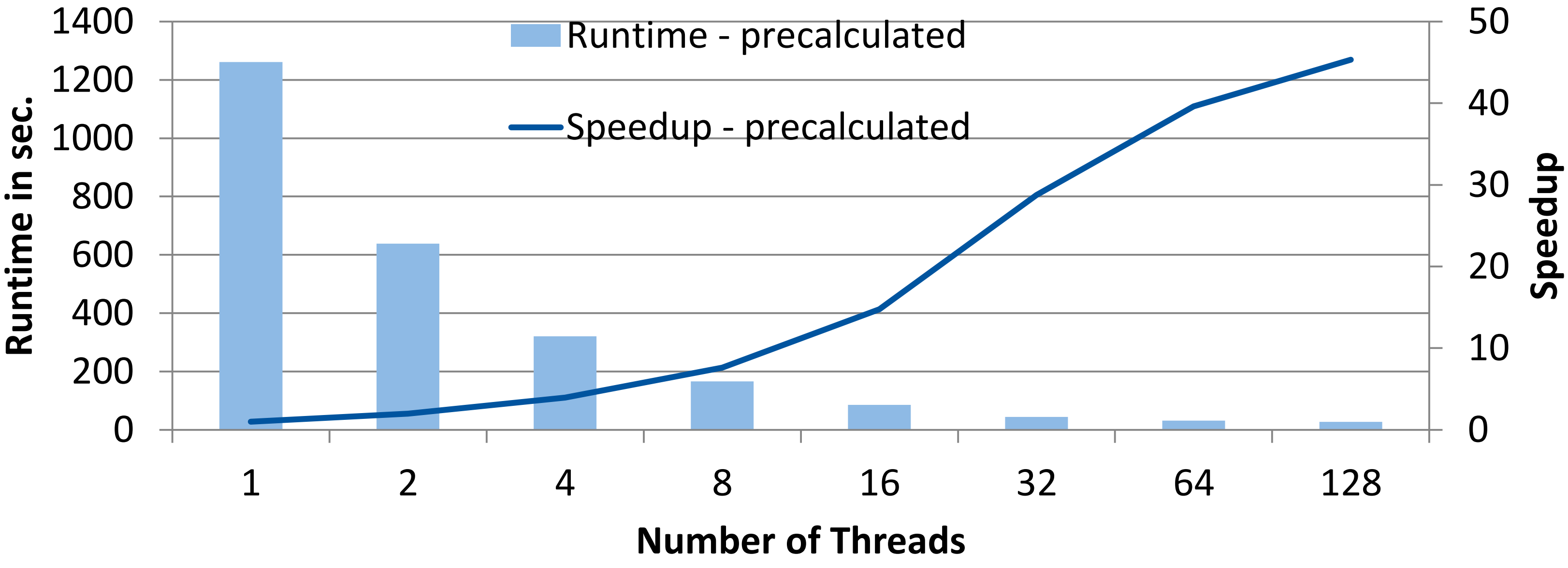
  → 2 socket sytem with Xeon X5675 processors, 12 OpenMP threads

	copy	scale	add	triad
ser_init	18.8 GB/s	18.5 GB/s	18.1 GB/s	18.2 GB/s
par_init	41.3 GB/s	39.3 GB/s	40.3 GB/s	40.4 GB/s



Case Study CG: Step 3

- **Tuning:**
  - → pre-calculate a schedule for the matrix-vector multiplication, so that the non-zeros are distributed evenly instead of the rows



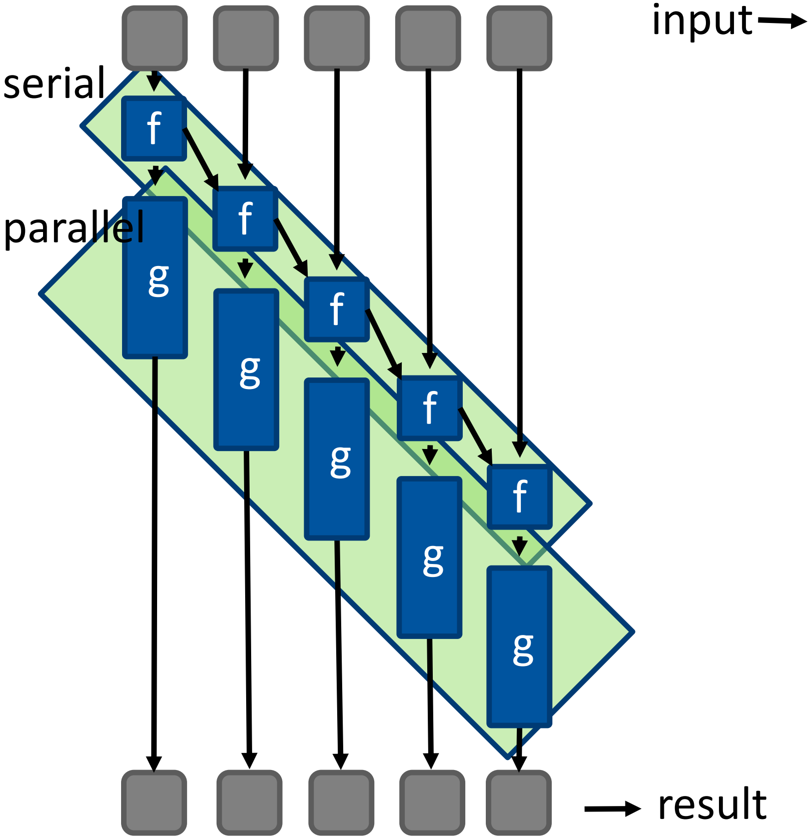
Patterns:
Data Management

The Pipeline Pattern

- **A *pipeline* is a linear sequence of stages.**

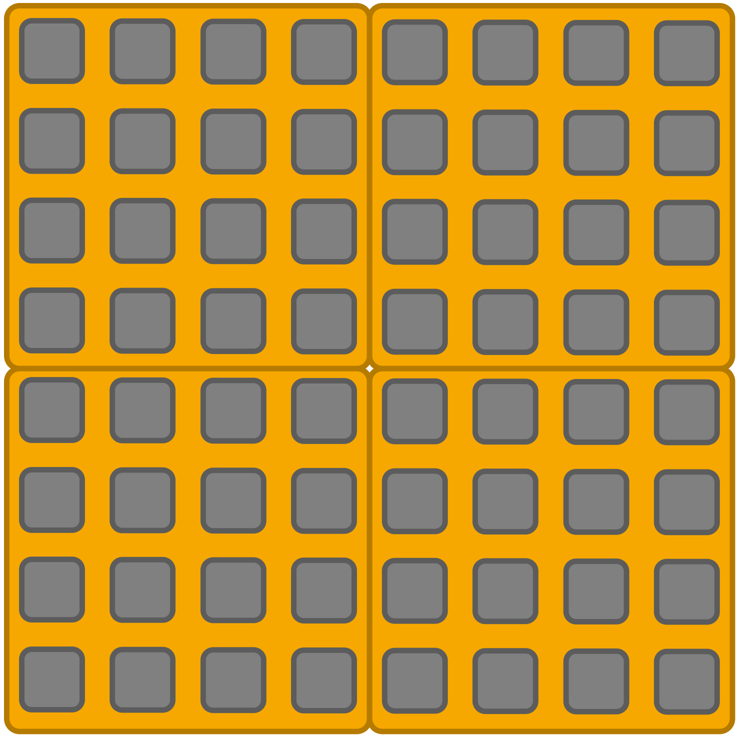
- **Parallelization approaches:**
  - → run different stages in parallel
  - → run multiple copies of stateless stages in parallel (reorder output)

- **Possibly need to manage buffering between stages.**



Geometric Decomposition

- **Break an index set into sub-index sets**
- ***Geometric partitioning* is a special case for non-overlapping sub-index sets**
- **Does not reorganize data,**
  only provides a different view on it

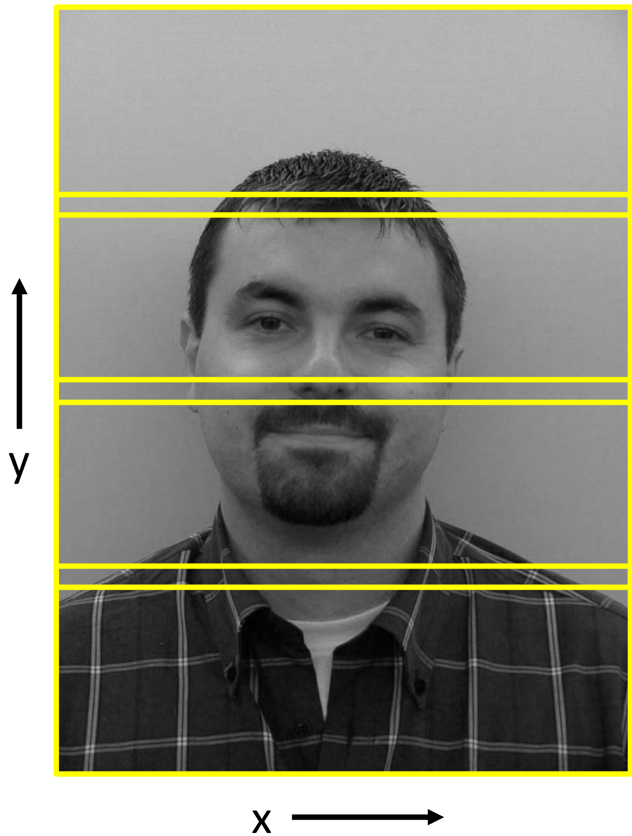


- **Requirements:**
  - → Computation independent for elements
  - → Regular data structure

- **Irregularly structured data (e.g. graph) can also be partitioned**

Example for Geometric Decomposition

Original photo:



Blurred photo:

Example for Geometric Decomposition

Original photo: 

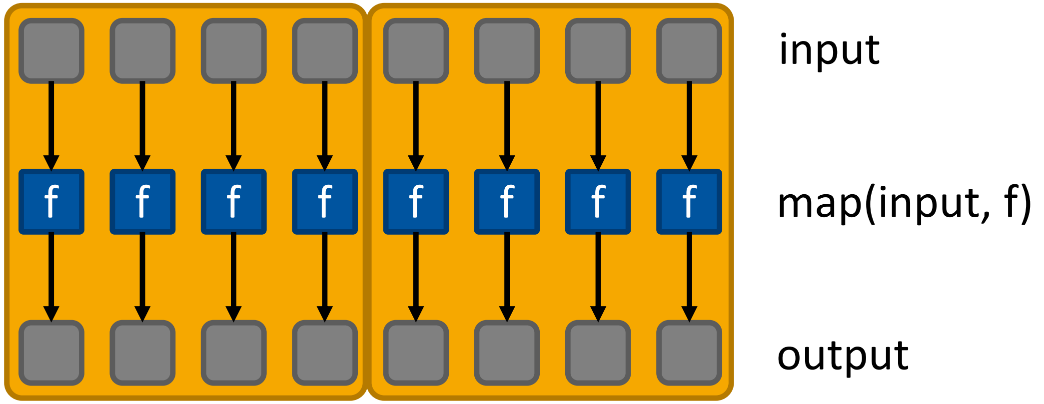
Blurred photo: 

OpenMP*

```
void blur(const picture &input, picture &output) {
  // traverse the picture
#pragma omp parallel for shared(input) shared(output)
  for (size_t y = 0; y < input.num_rows(); ++y) {
    for (size_t x = 0; x < input.num_cols(); ++x) {
      float pixel = 0.0;
      // apply the stencil
      for (size_t sy = STENCIL_WIDTH; sy < STENCIL_WIDTH; ++sx) {
        for (size_t sx = STENCIL_WIDTH; sx < STENCIL_WIDTH; ++sx) {
          // wrap around picture to avoid bogus accesses
          size_t py = wrap_around(input.num_rows(), sy - y);
          size_t px = wrap_around(input.num_cols(), sx - x);
          pixel += gauss_coeffs[sy][sx] * input.get(px,py);
      } }
      output.set(x,y,sum);
} } }
```

The Map Pattern - Revisited

- **To increase efficiency, the Map pattern can be fused with the Geometric Decomposition pattern**
  - → Useful for traditional OS-level threads
  - → Potentially having kazillions of threads executing only one instance of a function might be too fine-grained

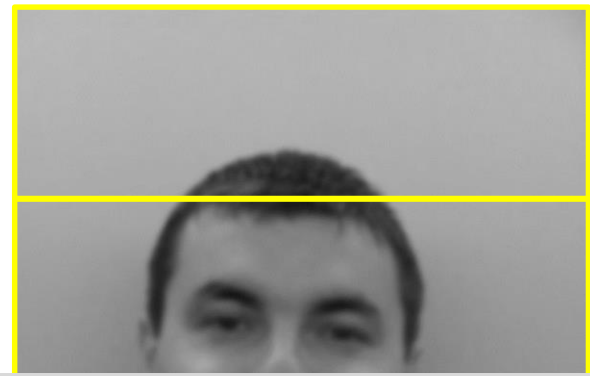


- **Have multiple partitions to work on in parallel, invoke function sequentially within each partition**

Fusing Geometric Decomposition and Map

Original photo:

Blurred photo:



```
void blur(const picture &input, picture &output) {
  // traverse over the picture
#pragma omp parallel for shared(input) shared(output)
  for (size_t y = 0; y < input.num_rows(); ++y) {
    for (size_t x = 0; x < input.num_cols(); ++x) {
      float pixel;
      // "map" the gauss stencil for current (x,y) position
      pixel = gauss_stencil(input.num_rows(), input.num_cols(), y, x);
      output.set(x,y,pixel);
} } }
```

OpenMP*

Case Study:

Karazuba

Polynomial Multiplication

				x^2	+	$2x$	+	3		**b**
				x^2	+	$4x$	+	5		**a**
				$5x^2$	+	$10x$	+	15		
		$4x^3$	+	$8x^2$	+	$12x$				
x^4	+	$2x^3$	+	$3x^2$						
x^4	+	$6x^3$	+	$16x^2$	+	$22x$	+	15		**c**

Storage Scheme:

		b[2]	b[1]	b[0]	**b**
		a[2]	a[1]	a[0]	**a**
c[4]	c[3]	c[2]	c[1]	c[0]	**c**

School Method

- **O(n^2) work:**

vector initialization

```
void simple_mul( T c[], const T a[], const T b[], size_t n ) {
  c[0:2*n-1] = 0;
  for (size_t i=0; i<n; ++i)
    c[i:n] += a[i]*b[0:n];
}
```

vector addition

- **Can we do any better?**

  → Yes. There are several methods, but Karatsuba is the one with the nicest name!

Karatsuba: Divide and Conquer

- **Suppose polynomials a and b have degree n**

  → let $K=x^{\lfloor n/2 \rfloor}$

  $a = a_1K+a_0$

  $b = b_1K+b_0$

- **Compute:**

  $t_0 = a_0 \cdot b_0$

  $t_1 = (a_0+a_1) \cdot (b_0+b_1)$

  $t_2 = a_1 \cdot b_1$

- **Then**

  $a \cdot b \equiv t_2K^2 + (t_1-t_0-t_2)K + t_0$

Partition coefficients.

3 half-sized multiplications. Do these recursively.

Sum products, shifted by multiples of K.

Save one (half) multiplication.

Cilk-parallel Karatsuba

```
void karatsuba( T c[], const T a[], const T b[], size_t n ) {




    size_t m = n/2;
    cilk_spawn karatsuba( c, a, b, m );                 // t0 = a0×b0
    cilk_spawn karatsuba( c+2*m, a+m, b+m, n-m );       // t2 = a1×b1
    temp_space<T> s(4*(n-m));
    T *a_=s.data(), *b_=a_+(n-m), *t=b_+(n-m);
    a_[0:m] = a[0:m]+a[m:m];                            // a_ = (a0+a1)
    b_[0:m] = b[0:m]+b[m:m];                            // b_ = (b0+b1)
    karatsuba( t, a_, b_, n-m );                        // t1 = (a0+a1)×(b0+b1)
    cilk_sync;
    t[0:2*m-1] -= c[0:2*m-1] + c[2*m:2*m-1];            // t = t1-t0-t2
    c[2*m-1] = 0;
    c[m:2*m-1] += t[0:2*m-1];                           // c = t2K^2+(t1-t0-t2)K+t0

}
```

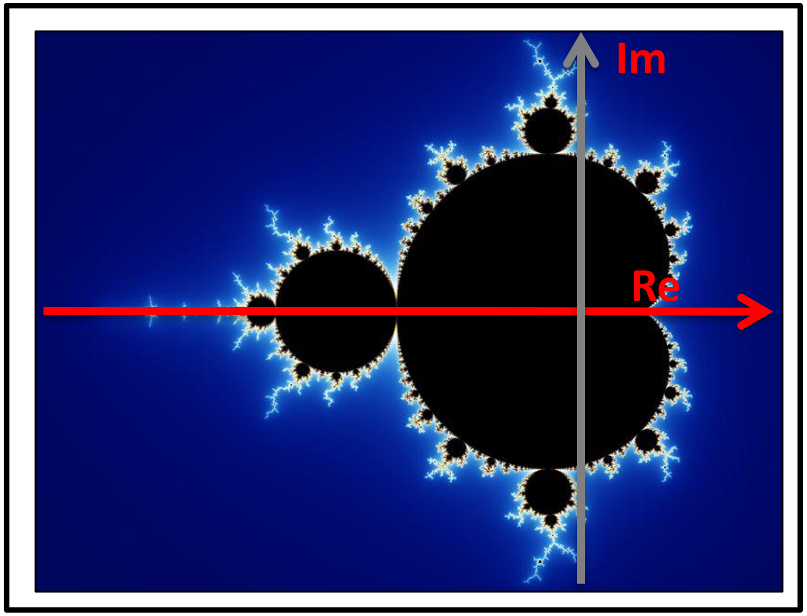
Cilk-parallel Karatsuba (better!)

```
void karatsuba( T c[], const T a[], const T b[], size_t n ) {
   if( n<=CutOff ) {
      simple_mul( c, a, b, n );
   } else {
      size_t m = n/2;
      cilk_spawn karatsuba( c, a, b, m );                    // t0 = a0×b0
      cilk_spawn karatsuba( c+2*m, a+m, b+m, n-m );          // t2 = a1×b1
      temp_space<T> s(4*(n-m));
      T *a_=s.data(), *b_=a_+(n-m), *t=b_+(n-m);
      a_[0:m] = a[0:m]+a[m:m];                               // a_ = (a0+a1)
      b_[0:m] = b[0:m]+b[m:m];                               // b_ = (b0+b1)
      karatsuba( t, a_, b_, n-m );                           // t1 = (a0+a1)×(b0+b1)
      cilk_sync;
      t[0:2*m-1] -= c[0:2*m-1] + c[2*m:2*m-1];               // t = t1-t0-t2
      c[2*m-1] = 0;
      c[m:2*m-1] += t[0:2*m-1];                              // c = t2K^2+(t1-t0-t2)K+t0
   }
}
```

Case Study:
Mandelbrot set
w/ Master-Worker

Case Study: Mandelbrot Set

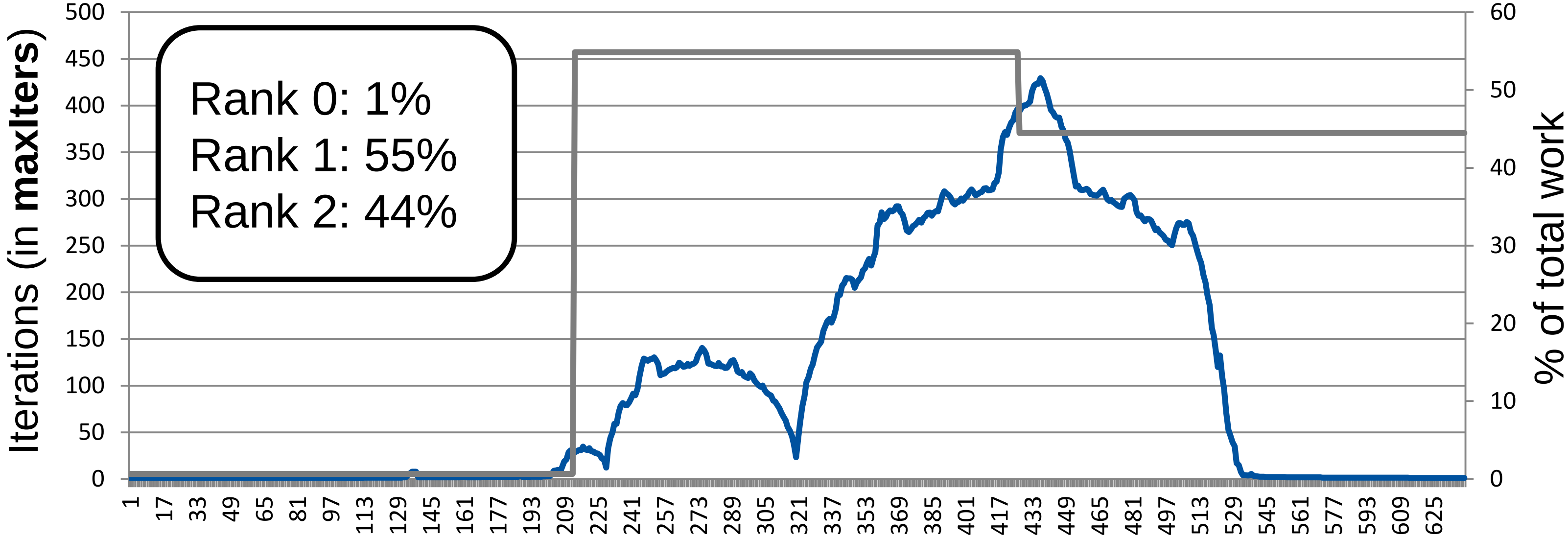
- **Set of complex numbers c for which**
  $z_0 = 0;\ z_{n+1} = {z_n}^2 + c \quad z, c \in \mathbb{C}$
  does not diverge.

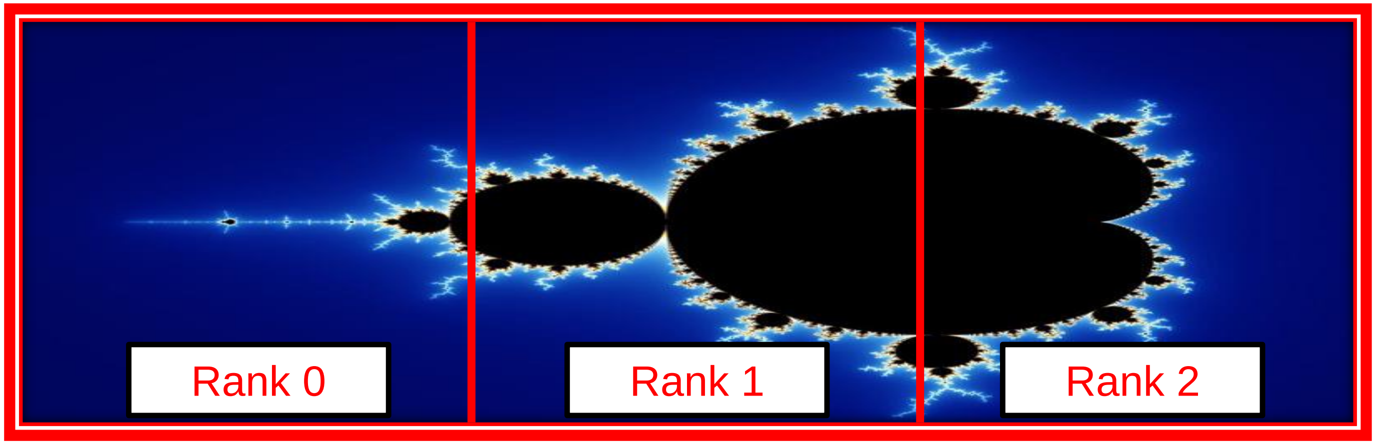



```
// For each image pixel (x, y):
// (x, y) - scaled pixel cords
c = x + i*y
z = 0
iteration = 0
maxIters = 1000
while (|z|² <= 2²
       && iteration < maxIters)
{
  z = z² + c
  iteration = iteration + 1
}
if (iteration == maxIters)
  color = black
else
  color = iteration
plot(x, y, color)
```

Case Study: Mandelbrot Set
Work Imbalance

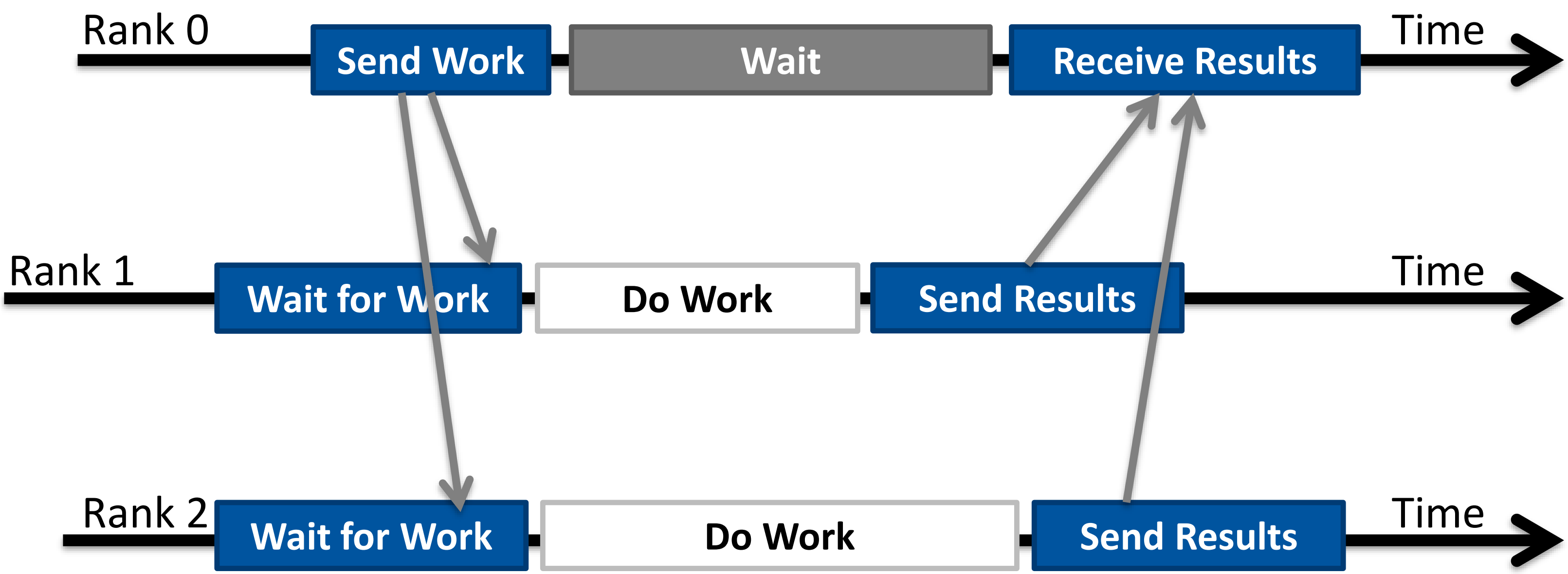
- Computation complexity mapped to ranks





Case Study: Mandelbrot Set
Master – Worker

- **One process (master) manages the work.**
  - → Work is split into many relatively small work items.
- **Many other processes (workers) compute over the work items:**

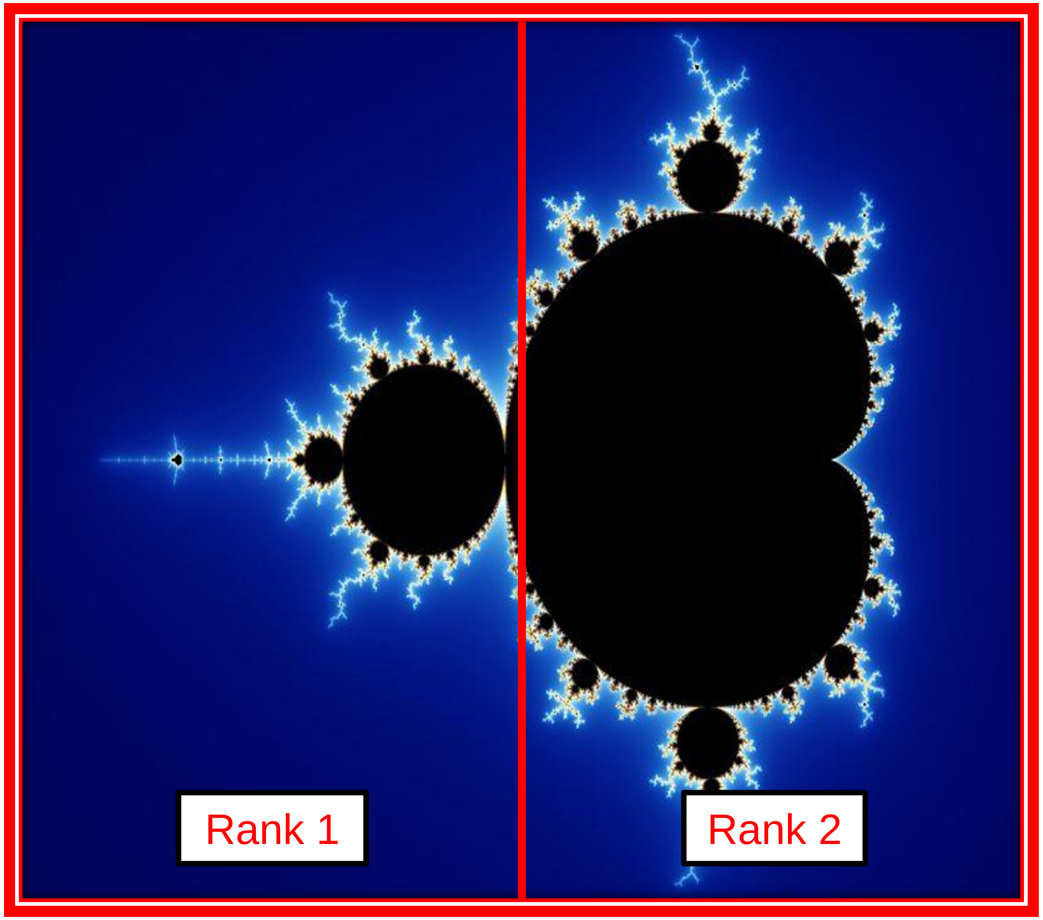


- **Above steps are repeated until all work items are processed.**
- **Sometimes called “bag of jobs” pattern.**

Case Study: Mandelbrot Set
Master – Worker

- **The algorithm:**

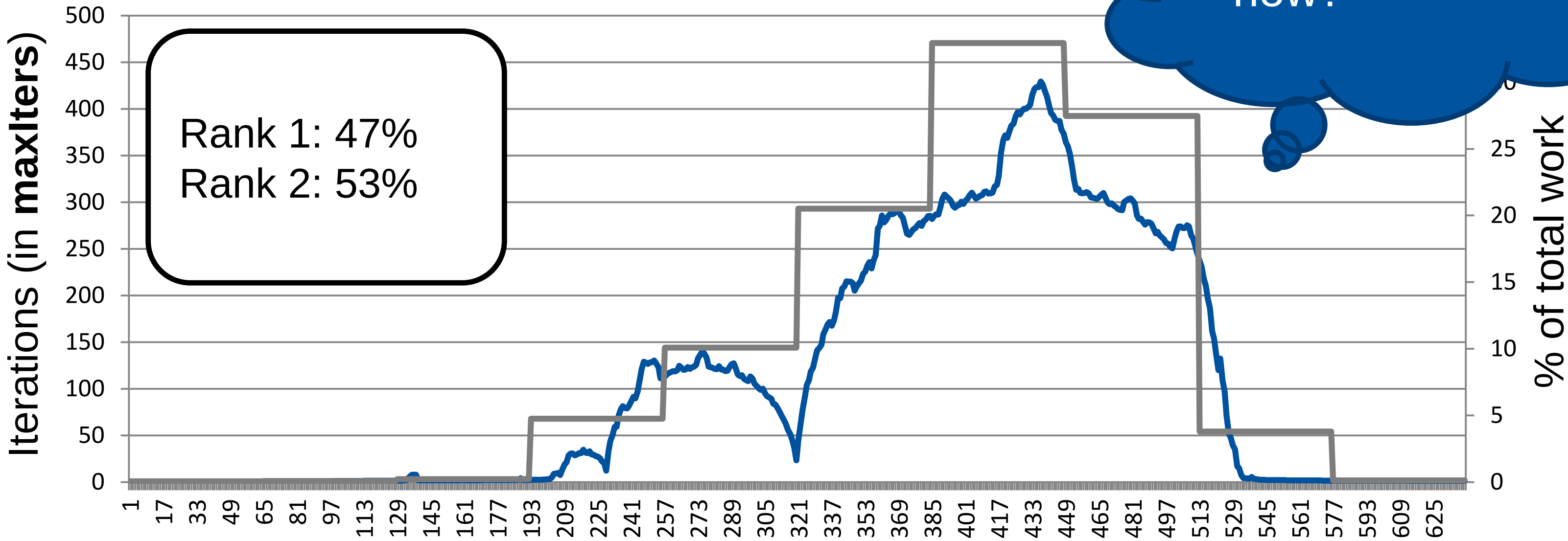
```
if (rank == 0) {  // Master
  splitDomain;
  sendWorkItems;
  receivePartialResults;
  assembleResult;
  output;
} else {          // Worker
  receiveWorkItems;
  processWorkItems;
  sendPartialResults;
}
```

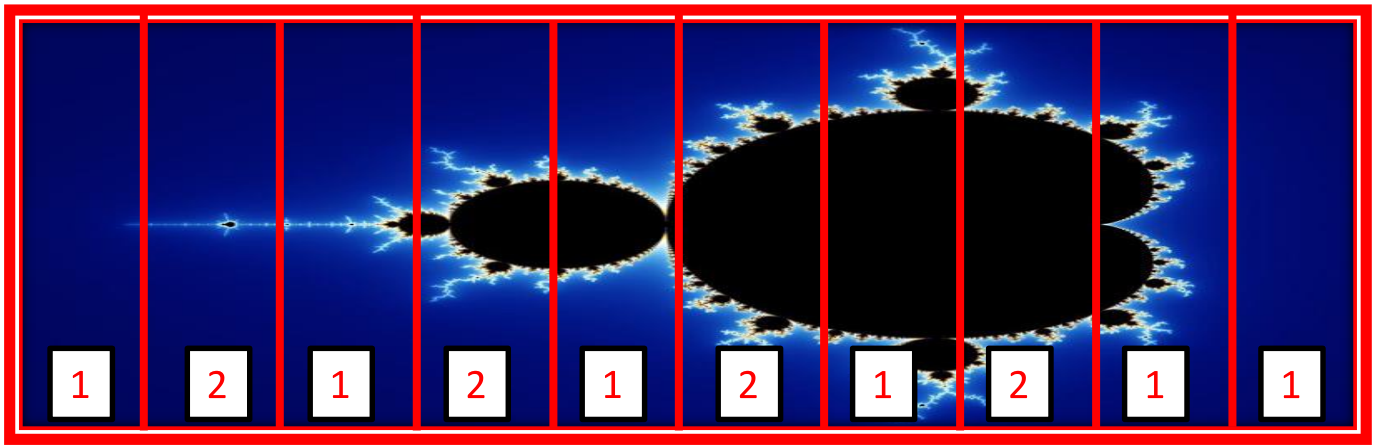


Case Study: Mandelbrot Set
Work Balancing

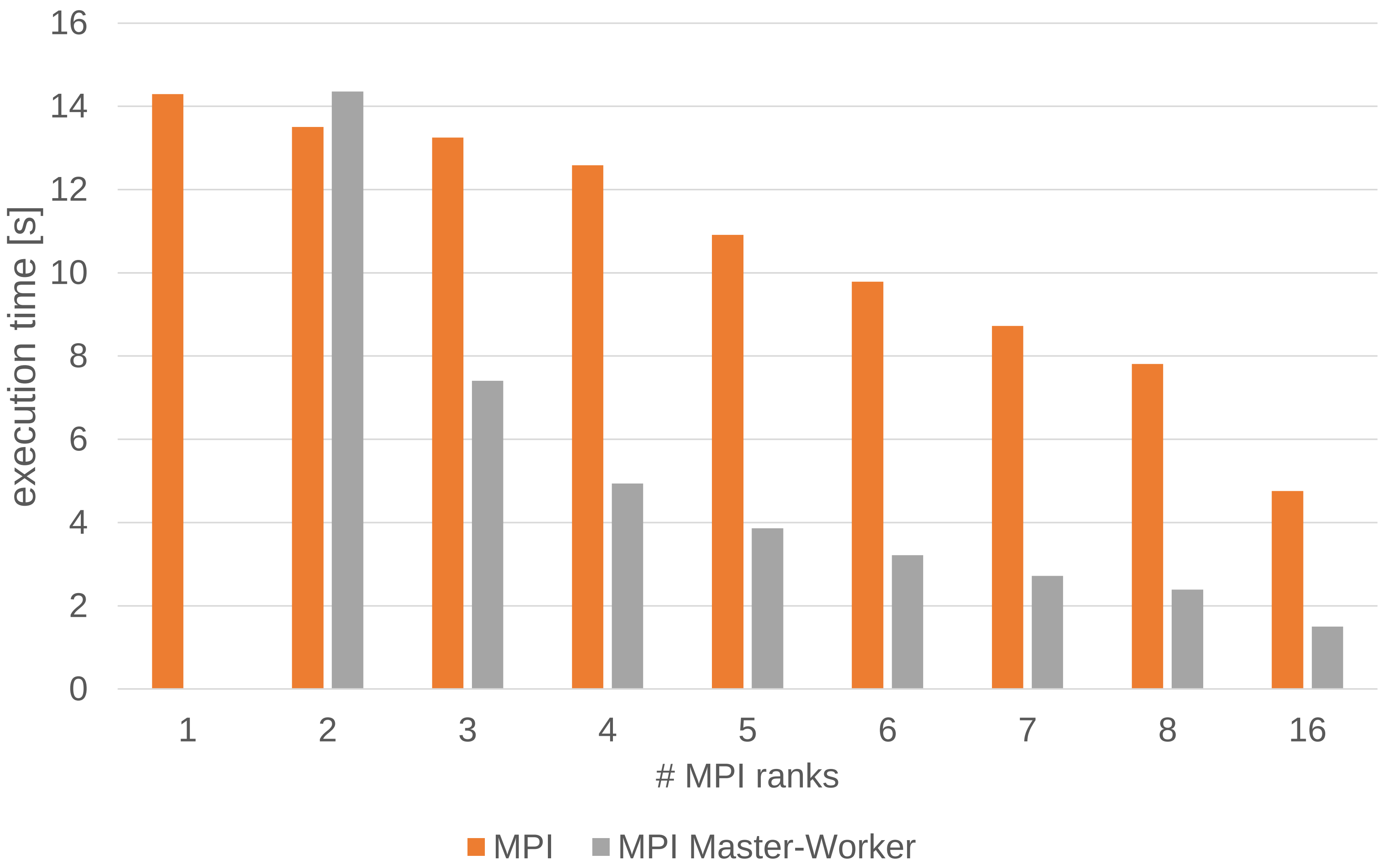
- **Computational complexity mapped to ranks**

Is it any better now?





Case Study: Mandelbrot Set
Results



Quiz

Quiz (1/3)

- **We wanted to make you think parallel: were these two lectures useful?**
  a) Yes
  b) No
  c) Yes, at least I have learned how to build a house
  d) -

Quiz (2/3)

- **Throughout this course we offered you to get experienced in using the HPC Cluster at RWTH Aachen University. Did you ever login?**
  a) Yes, several times
  b) Once to see how the environment looks like
  c) Never
  d) What is an HPC cluster?

Quiz (3/3)

- **In case you are interested in learning more about HPC: would you attend a dedicated lecture on Parallel and Data-centric Programming Models?**
  a) Yes
  b) Maybe if it fits my schedule
  c) No interest whatsoever
  d) –

Summary

Parallel Software Engineering

- **Problem: Amdahl's Law notes that scaling will be limited by the serial fraction of your program.**
- ***Solution: go parallel "as far as you can", exploit all dimensions of parallelism.***

- **Problem: Locking, access to data (memory and communication), and overhead will strangle scaling.**
- ***Solution: use programming approaches with good data locality and low overhead, and avoid locks.***

- **Problem: Parallelism introduces new debugging challenges: deadlocks and race conditions.**
- ***Solution: use structured programming strategies to avoid these by design, improving maintainability.***

Question Time…