RWTH AACHEN UNIVERSITY

Introduction to OpenMP

Christian Terboven <terboven@itc.rwth-aachen.de>

Dirk Schmidl <schmidl@itc.rwth-aachen.de>

18.03.2015 / Aachen, Germany

Stand: 12.03.2015

Version 2.3

RWTH AACHEN UNIVERSITY

Tasking

The Single Construct

C/C++

```
#pragma omp single [clause]
... structured block ...
```

Fortran

```
!$omp single [clause]
... structured block ...
!$omp end single
```

- **The `single` construct specifies that the enclosed structured block is executed by only on thread of the team.**
  - → It is up to the runtime which thread that is.

- **Useful for:**
  - → I/O
  - → Memory allocation and deallocation, etc. (in general: setup work)
  - → Implementation of the single-creator parallel-executor pattern as we will see now…

Recursive approach to compute Fibonacci

RWTH AACHEN UNIVERSITY

```
int main(int argc,
         char* argv[])
{
   [...]
   fib(input);
   [...]
}
```

```
int fib(int n)   {
   if (n < 2) return n;
   int x = fib(n - 1);
   int y = fib(n - 2);
   return x+y;
}
```

- **On the following slides we will discuss three approaches to parallelize this recursive code with Tasking.**

The Task Construct

C/C++

```
#pragma omp task [clause]
... structured block ...
```

Fortran

```
!$omp task [clause]
... structured block ...
!$omp end task
```

- **Each encountering thread/task creates a new Task**
  - → Code and data is being packaged up
  - → Tasks can be nested
    - → Into another Task directive
    - → Into a Worksharing construct
- **Data scoping clauses:**
  - → `shared`(*list*)
  - → `private`(*list*) `firstprivate`(*list*)
  - → `default`(*shared | none*)

Tasks in OpenMP: Data Scoping

- **Some rules from *Parallel Regions* apply:**
  - → Static and Global variables are shared
  - → Automatic Storage (local) variables are private

- **If `shared` scoping is not derived by default:**
  - → Orphaned Task variables are `firstprivate` by default!
  - → Non-Orphaned Task variables inherit the `shared` attribute!
  - → Variables are `firstprivate` unless `shared` in the enclosing context

- **So far no verification tool is available to check Tasking programs for correctness!**

First version parallelized with Tasking (omp-v1)

RWTH AACHEN UNIVERSITY

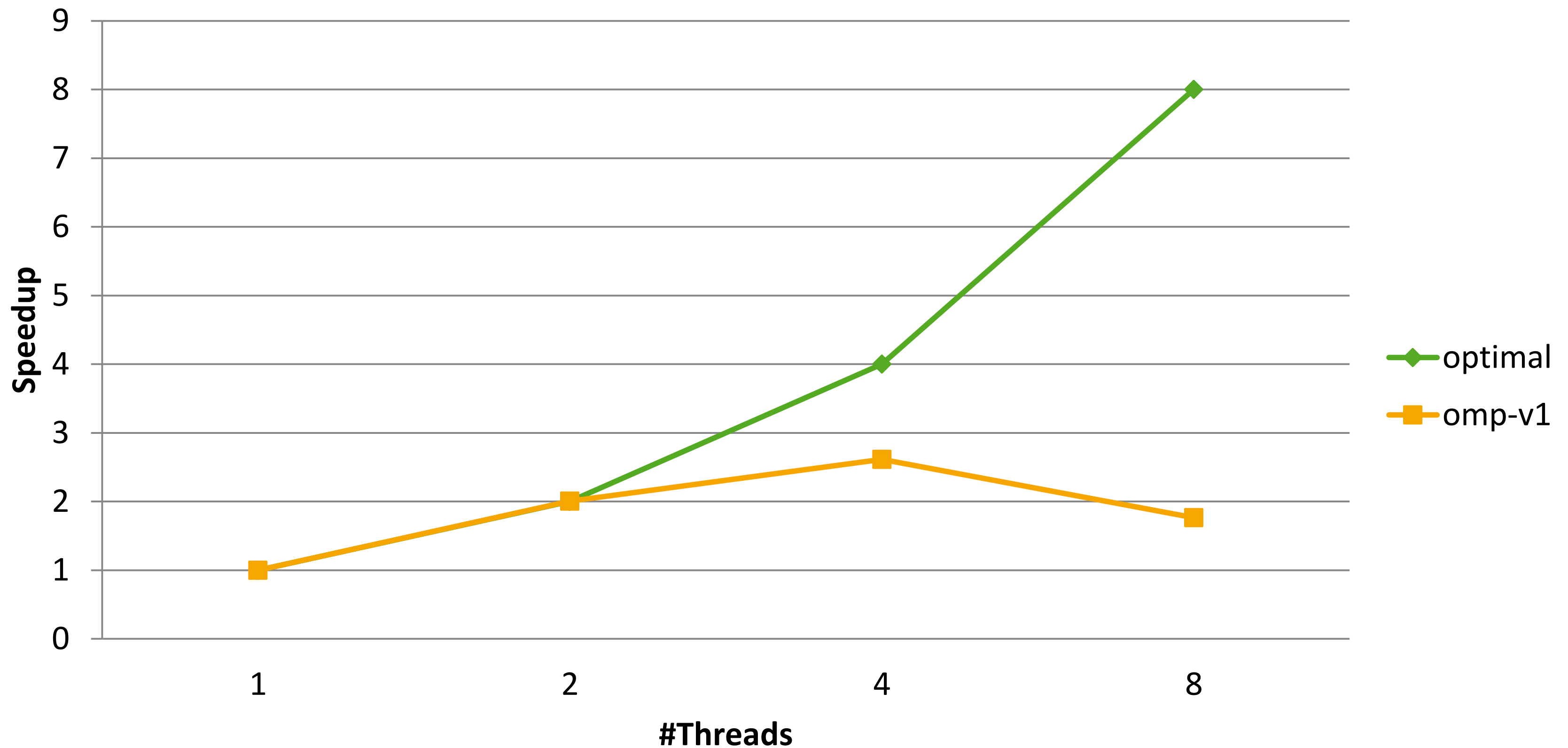
```
int main(int argc,
         char* argv[])
{
   [...]
#pragma omp parallel
{
#pragma omp single
{
   fib(input);
}
}
   [...]
}
```

```
int fib(int n)   {
   if (n < 2) return n;
int x, y;
#pragma omp task shared(x)
{
   x = fib(n - 1);
}
#pragma omp task shared(y)
{
   y = fib(n - 2);
}
#pragma omp taskwait
   return x+y;
}
```

- **Only one Task / Thread enters `fib()` from `main()`, it is responsable for creating the two initial work tasks**
- **Taskwait is required, as otherwise `x` and `y` would be lost**

Scalability measurements (1/3)

- Overhead of task creation prevents better scalability!



Improved parallelization with Tasking (omp-v2)

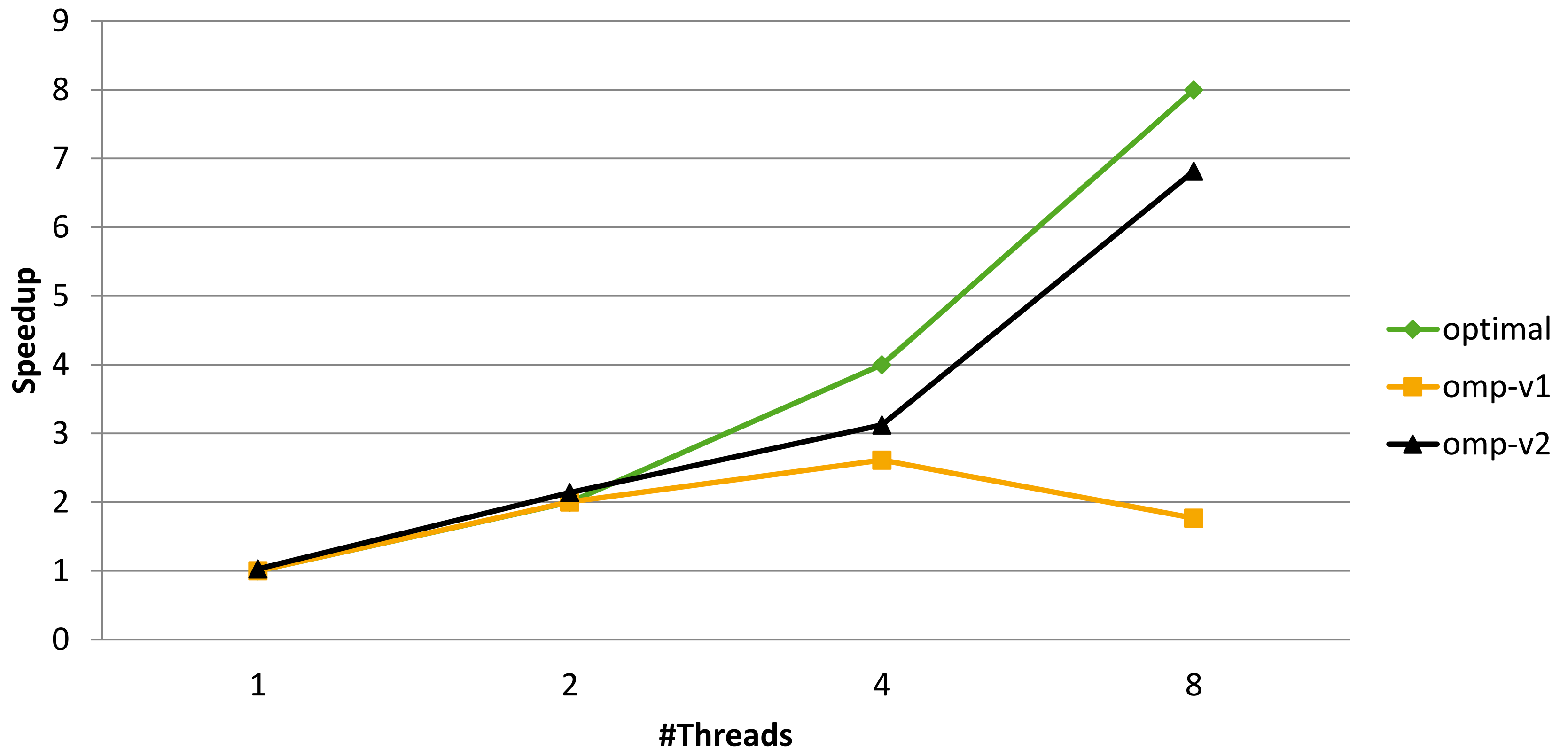
- **Improvement: Don‘t create yet another task once a certain (small enough) `n` is reached**

```
int main(int argc,
         char* argv[])
{
   [...]
#pragma omp parallel
{
#pragma omp single
{
   fib(input);
}
}
   [...]
}
```

```
int fib(int n)   {
   if (n < 2) return n;
int x, y;
#pragma omp task shared(x) \
   if(n > 30)
{
   x = fib(n - 1);
}
#pragma omp task shared(y) \
   if(n > 30)
{
   y = fib(n - 2);
}
#pragma omp taskwait
   return x+y;
}
```

Scalability measurements (2/3)

- **Speedup is ok, but we still have some overhead when running with 4 or 8 threads**



Improved parallelization with Tasking (omp-v3)

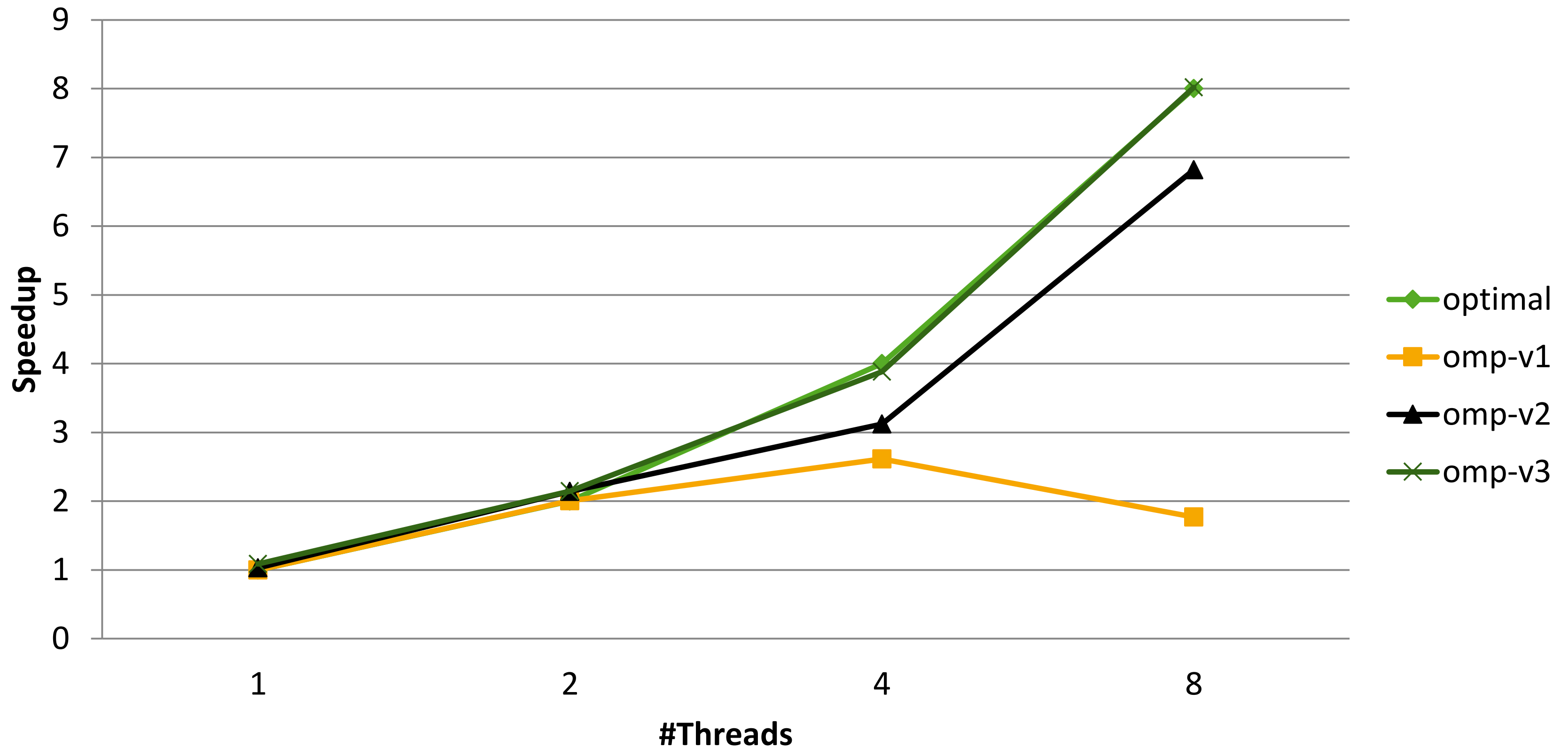
- **Improvement: Skip the OpenMP overhead once a certain n is reached (no issue w/ production compilers)**

```
int main(int argc,
         char* argv[])
{
   [...]
#pragma omp parallel
{
#pragma omp single
{
   fib(input);
}
}
   [...]
}
```

```
int fib(int n)   {
   if (n < 2) return n;
   if (n <= 30)
      return serfib(n);
int x, y;
#pragma omp task shared(x)
{
   x = fib(n - 1);
}
#pragma omp task shared(y)
{
   y = fib(n - 2);
}
#pragma omp taskwait
   return x+y;
}
```

Scalability measurements (3/3)

- Everything ok now ☺



Data Scoping Example (1/7)

RWTH AACHEN UNIVERSITY

```
int a = 1;
void foo()
{
   int b = 2, c = 3;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d = 4;
        #pragma omp task
        {
                int e = 5;

                // Scope of a:
                // Scope of b:
                // Scope of c:
                // Scope of d:
                // Scope of e:
} } }
```

Data Scoping Example (2/7)

```
int a = 1;
void foo()
{
   int b = 2, c = 3;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d = 4;
        #pragma omp task
        {
               int e = 5;

               // Scope of a: shared
               // Scope of b:
               // Scope of c:
               // Scope of d:
               // Scope of e:
} } }
```

Data Scoping Example (3/7)

```
int a = 1;
void foo()
{
   int b = 2, c = 3;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d = 4;
        #pragma omp task
        {
               int e = 5;

               // Scope of a: shared
               // Scope of b: firstprivate
               // Scope of c:
               // Scope of d:
               // Scope of e:
} } }
```

Data Scoping Example (4/7)

```
int a = 1;
void foo()
{
   int b = 2, c = 3;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d = 4;
        #pragma omp task
        {
               int e = 5;

               // Scope of a: shared
               // Scope of b: firstprivate
               // Scope of c: shared
               // Scope of d:
               // Scope of e:
} } }
```

Data Scoping Example (5/7)

RWTH AACHEN UNIVERSITY

```
int a = 1;
void foo()
{
   int b = 2, c = 3;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
       int d = 4;
       #pragma omp task
       {
              int e = 5;

              // Scope of a: shared
              // Scope of b: firstprivate
              // Scope of c: shared
              // Scope of d: firstprivate
              // Scope of e:
} } }
```

Data Scoping Example (6/7)

RWTH AACHEN UNIVERSITY

```
int a = 1;
void foo()
{
   int b = 2, c = 3;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d = 4;
        #pragma omp task
        {
               int e = 5;

               // Scope of a: shared
               // Scope of b: firstprivate
               // Scope of c: shared
               // Scope of d: firstprivate
               // Scope of e: private
} } }
```

Hint: Use default(none) to be forced to think about every variable if you do not see clear.

Data Scoping Example (7/7)

```
int a = 1;
void foo()
{
   int b = 2, c = 3;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d = 4;
        #pragma omp task
        {
               int e = 5;

               // Scope of a: shared,         value of a: 1
               // Scope of b: firstprivate,   value of b: 0 / undefined
               // Scope of c: shared,         value of c: 3
               // Scope of d: firstprivate,   value of d: 4
               // Scope of e: private,        value of e: 5
} } }
```

The Barrier and Taskwait Constructs

- **OpenMP `barrier` (implicit or explicit)**
  - → All tasks created by any thread of the current *Team* are guaranteed to be completed at barrier exit

```
C/C++
#pragma omp barrier
```

- **Task barrier: `taskwait`**
  - → Encountering Task suspends until child tasks are complete
    - → Only direct childs, not descendants!

```
C/C++
#pragma omp taskwait
```

Task Synchronization

- **Task Synchronization explained:**

```
#pragma omp parallel num_threads(np)
{
#pragma omp task
   function_A();
#pragma omp barrier
#pragma omp single
   {
#pragma omp task
      function_B();
   }
}
```

np Tasks created here, one for each thread

All Tasks guaranteed to be completed here

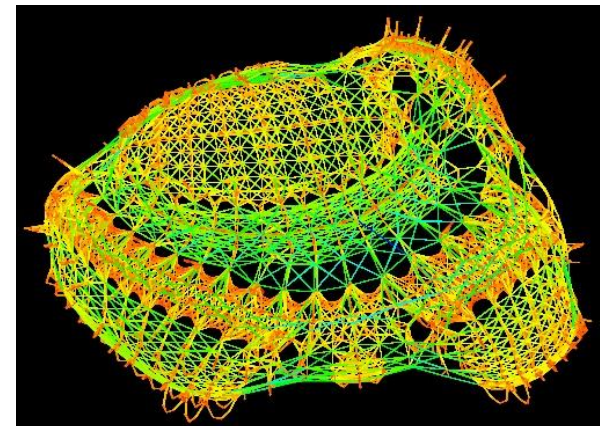
1 Task created here

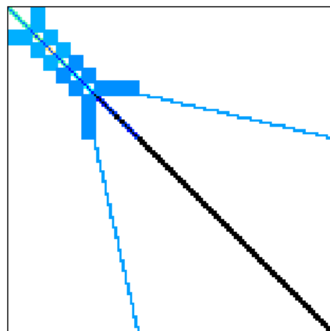
B-Task guaranteed to be completed here

Load Balancing

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

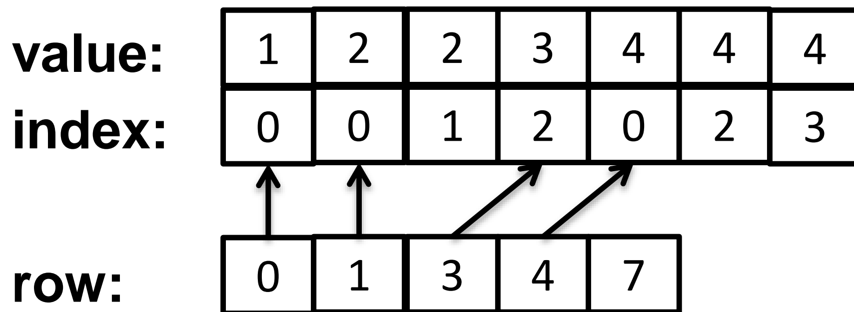
(Quelle: Beijing Botanical Garden and University of Florida, Sparse Matrix Collection)

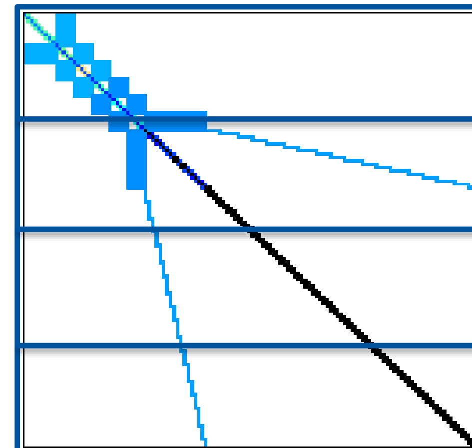
Case Study: CG

RWTH AACHEN UNIVERSITY

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

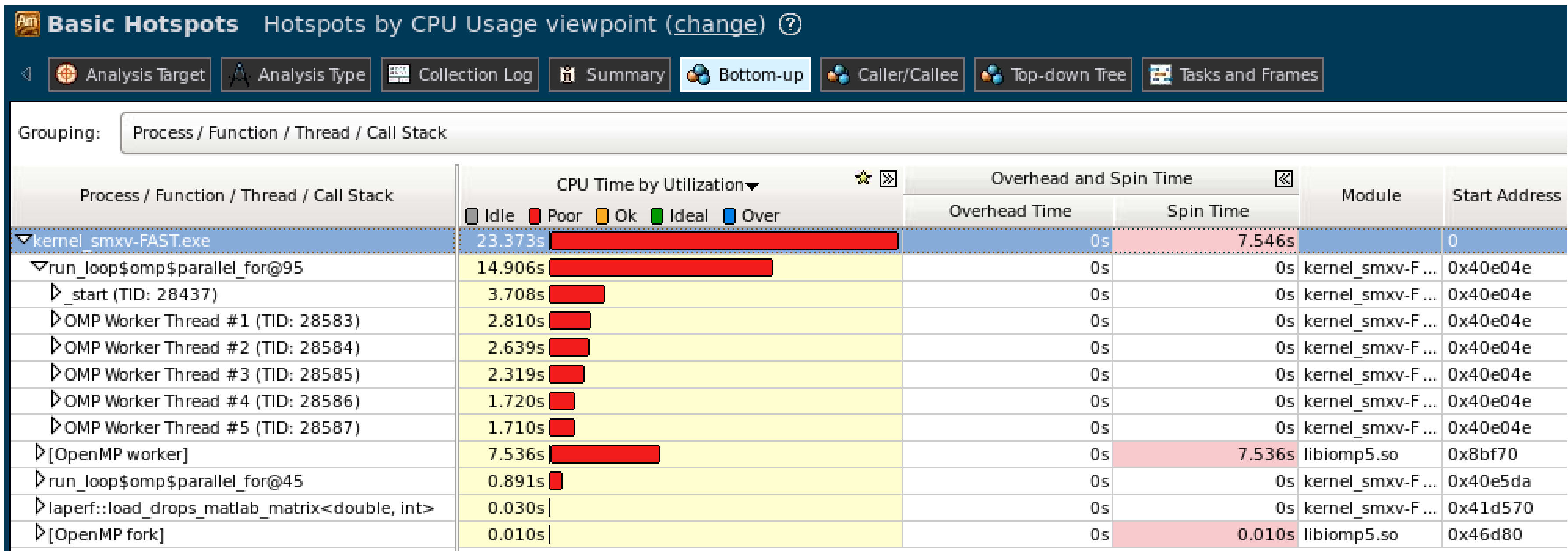
- $A = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 2 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 4 & 0 & 4 & 4 \end{pmatrix}$
- **Format: compressed row storage**
- **store all values and columns in arrays (length nnz)**
- **store beginning of a new row in a third array (length n+1)**





Load Imbalance in VTune

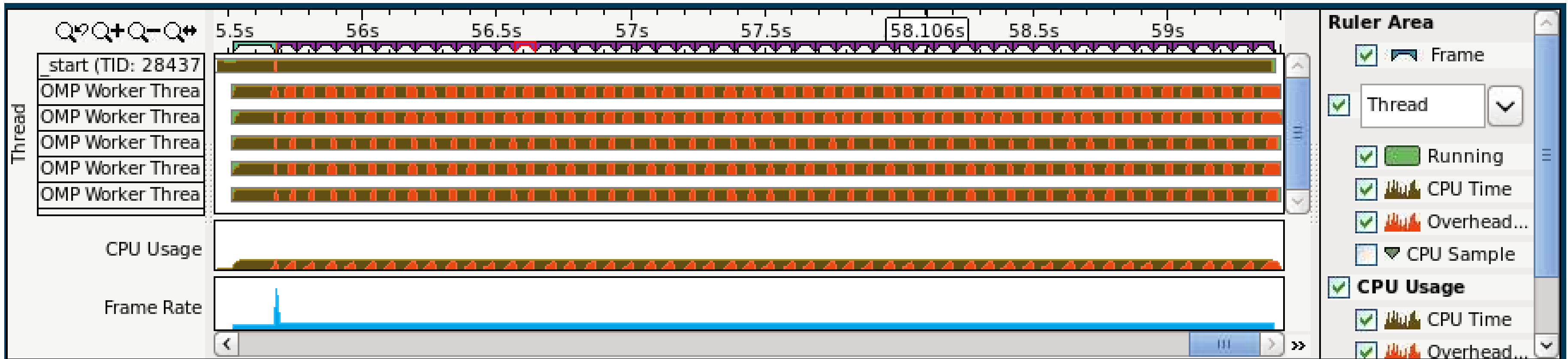
- Grouping execution time of parallel regions by threads helps to detect load imbalance.
- Significant potions of Spin Time also indicate load balance problems.
- Different loop schedules might help to avoid these problems.



Process / Function / Thread / Call Stack	CPU Time by Utilization (Idle, Poor, Ok, Ideal, Over)	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.

Parallel Loop Scheduling

Influencing the For Loop Scheduling

- ***for*-construct: OpenMP allows to influence how the iterations are scheduled among the threads of the team, via the *schedule* clause:**
  - → `schedule(static [, chunk])`: Iteration space divided into blocks of chunk size, blocks are assigned to threads in a round-robin fashion. If chunk is not specified: #threads blocks.
  - → `schedule(dynamic [, chunk])`: Iteration space divided into blocks of chunk (not specified: 1) size, blocks are scheduled to threads in the order in which threads finish previous blocks.
  - → `schedule(guided [, chunk])`: Similar to dynamic, but block size starts with implementation-defined value, then is decreased exponentially down to chunk.
- **Default on most implementations is `schedule(static)`.**

RWTH AACHEN UNIVERSITY

Exercises 3 and 5