Correctness and Performance Tools for OpenMP

Dirk Schmidl
IT Center, RWTH Aachen University
Member of the HPC Group
schmidl@itc.rwth-aachen.de

Christian Terboven
IT Center, RWTH Aachen University
Deputy lead of the HPC Group
terboven@itc.rwth-aachen.de

Correctness Checking

Race Condition

- **Data Race: the typical OpenMP programming error, when:**
  - → two or more threads access the same memory location, and
  - → at least one of these accesses is a write, and
  - → the accesses are not protected by locks or critical regions, and
  - → the accesses are not synchronized, e.g. by a barrier.
- **Non-deterministic occurrence: e.g. the sequence of the execution of parallel loop iterations is non-deterministic and may change from run to run**
- **In many cases *private* clauses, *barriers* or *critical regions* are missing**
- **Data races are hard to find using a traditional debugger**
  - → Use the *Intel Inspector XE*

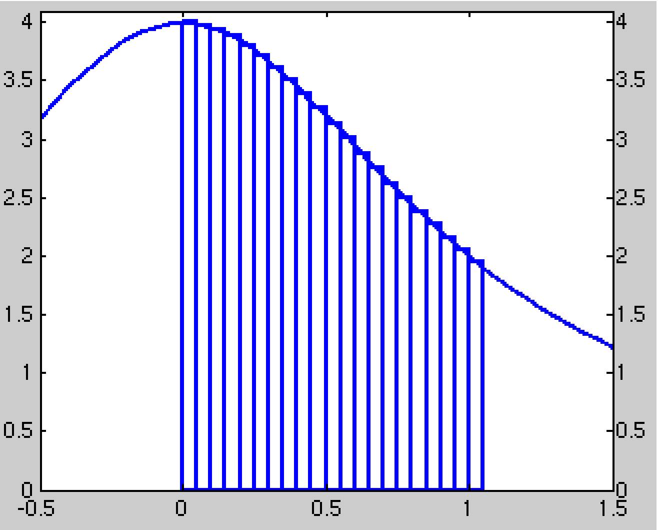
Intel Inspector XE

- **Detection of**
  - → Memory Errors
  - → Dead Locks
  - → Data Races
- **Support for**
  - → Linux (32bit and 64bit) and Windows (32bit and 64bit)
  - → WIN32-Threads, Posix-Threads, Intel Threading Building Blocks and OpenMP
- **New Features (compared to Intel Thread Checker)**
  - → Binary Instrumentation gives full functionality
  - → Independent stand-alone GUI for Windows and Linux
  - → memory error detection
  - → static security analysis (in combination with the Intel 12.X compiler)

PI Example Code

$$\pi = \int_0^1 \frac{4}{1 + x^2}$$

```
double f(double x)
{
   return (4.0 / (1.0 + x*x));
}

double CalcPi (int n)
{
   const double fH   = 1.0 / (double) n;
   double fSum = 0.0;
   double fX;
   int i;

#pragma omp parallel for private(fX,i) reduction(+:fSum)
   for (i = 0; i < n; i++)
   {
      fX = fH * ((double)i + 0.5);
      fSum += f(fX);
   }
   return fH * fSum;
}
```



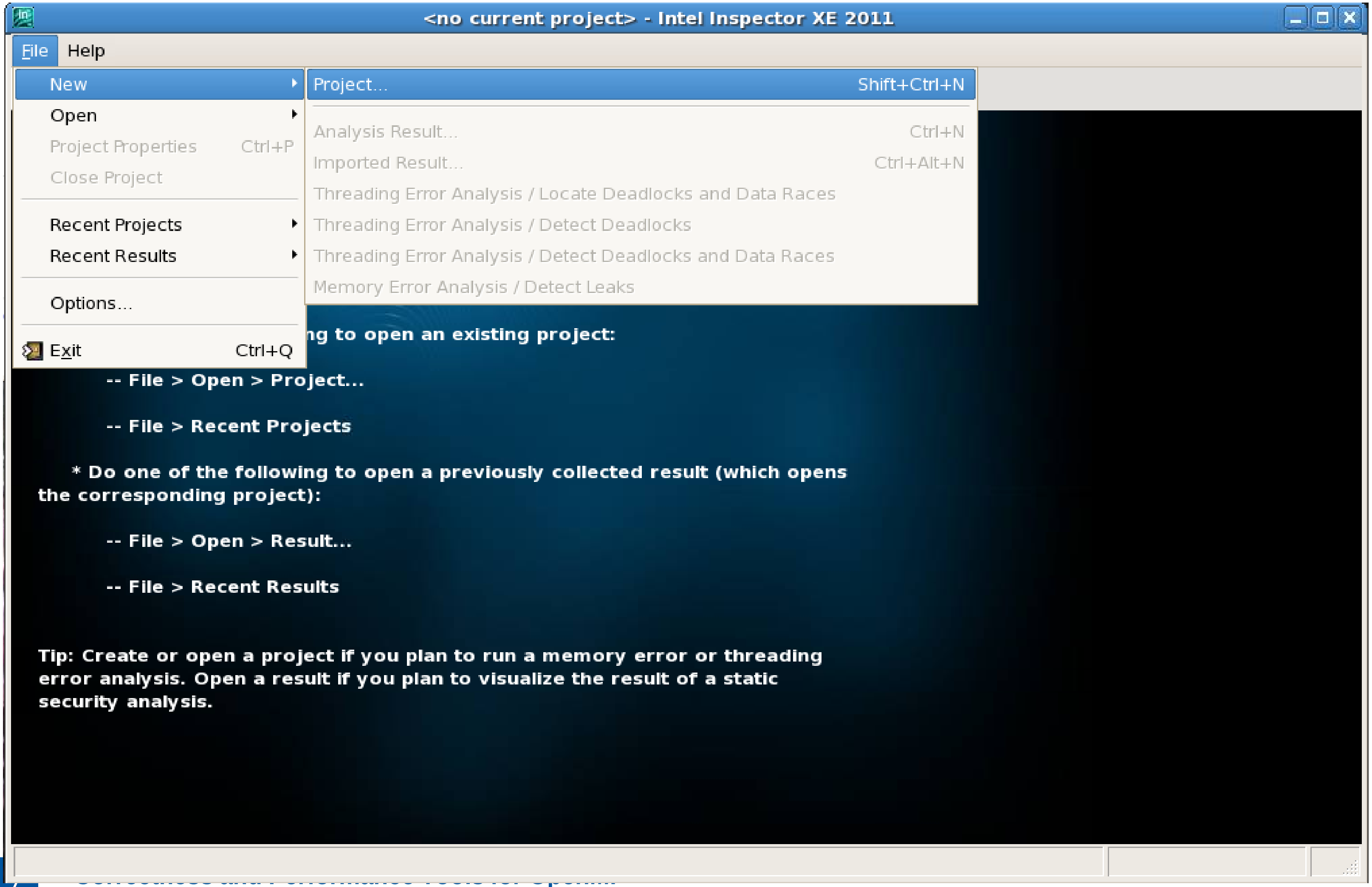
PI Example Code

```
double f(double x)
{
   return (4.0 / (1.0 + x*x));
}

double CalcPi (int n)
{
   const double fH   = 1.0 / (double) n;
   double fSum = 0.0;
   double fX;
   int i;

#pragma omp parallel for private(fX,i) reduction(+:fSum)
   for (i = 0; i < n; i++)
   {
      fX = fH * ((double)i + 0.5);
      fSum += f(fX);
   }
   return fH * fSum;
}
```

What if we would have forgotten this?

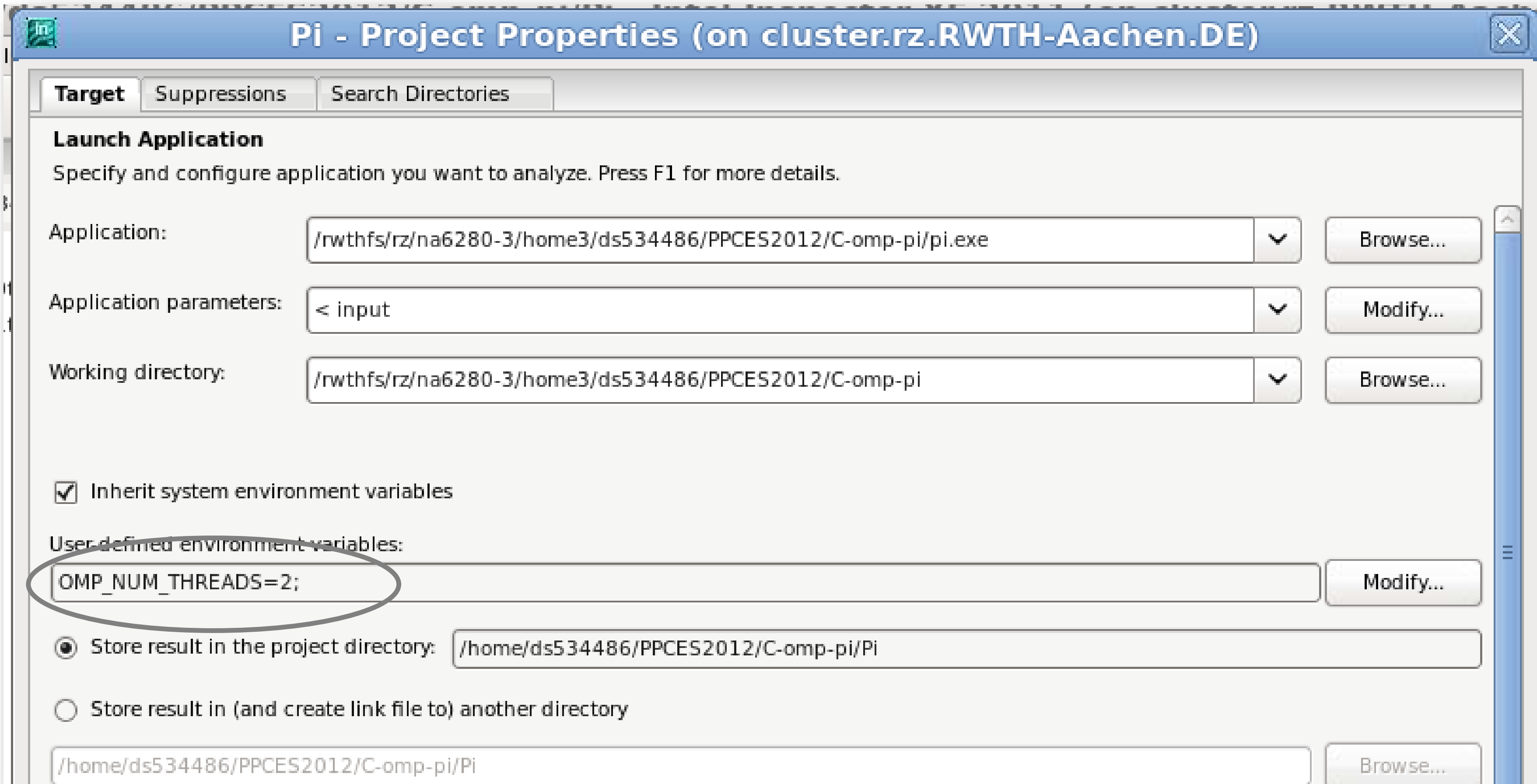
Inspector XE – Create Project

$ module load intelixe ; inspxe-gui



Inspector XE – Create Project

- ensure that multiple threads are used
- choose a real small dataset, execution time can grow 10X – 1000X

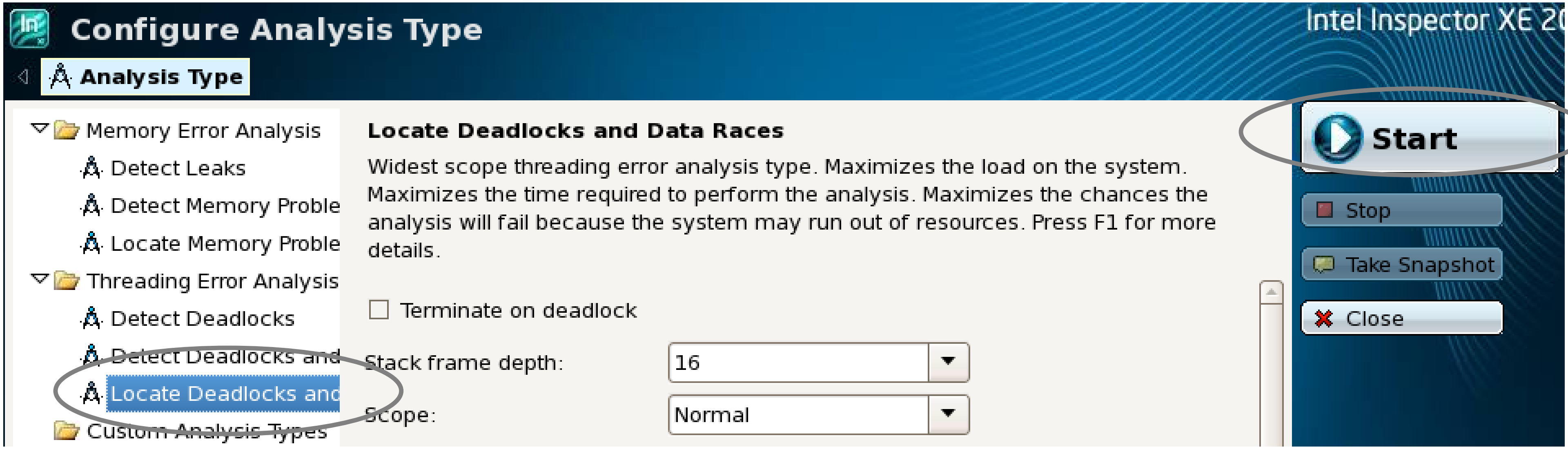


Inspector XE – Configure Analysis

Threading Error Analysis Modes

1. Detect Deadlocks
2. Detect Deadlocks and Data Races
3. Locate Deadlocks and Data Races

more details,
more overhead

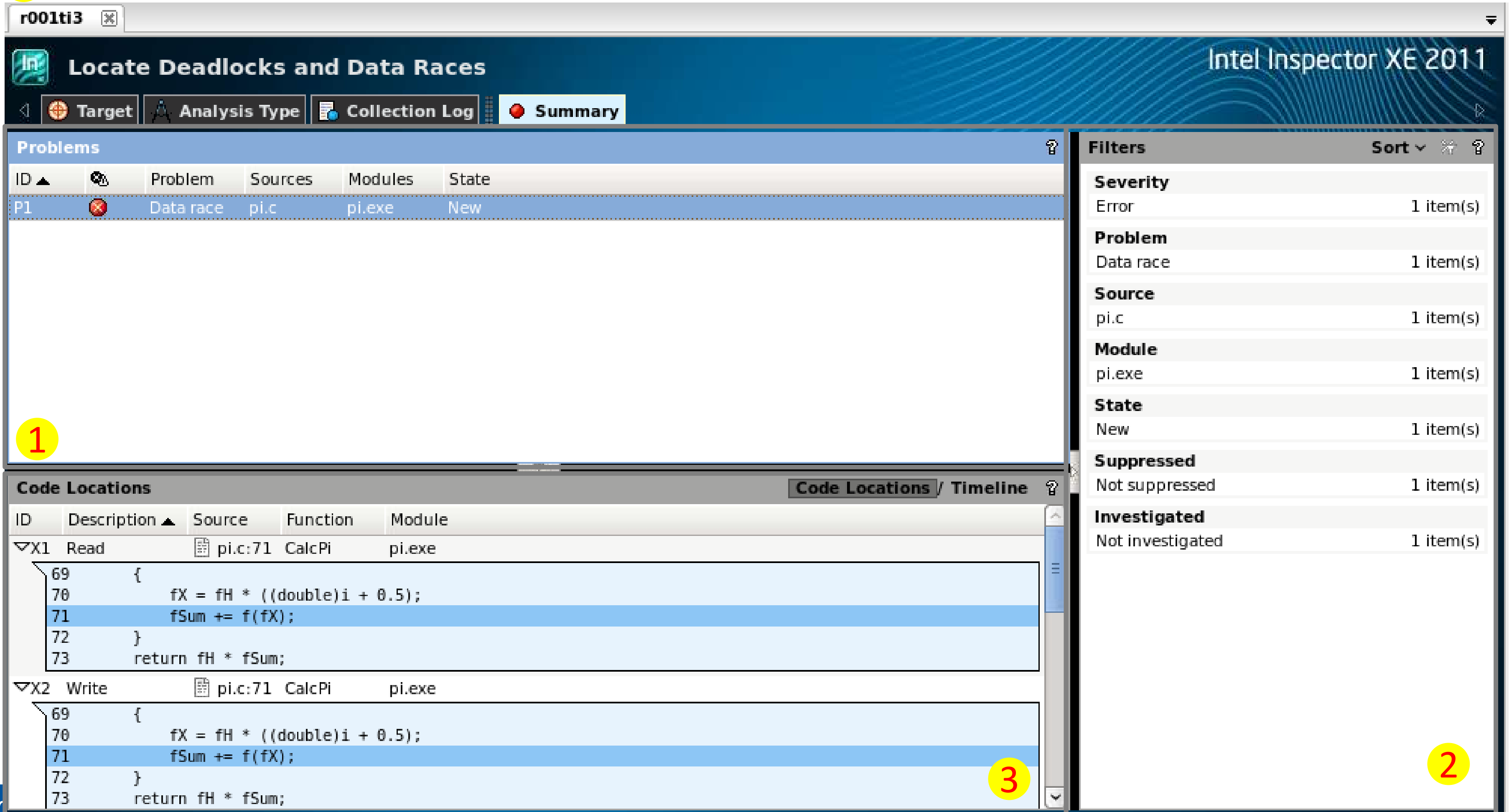


Inspector XE – Results

1 detected problems

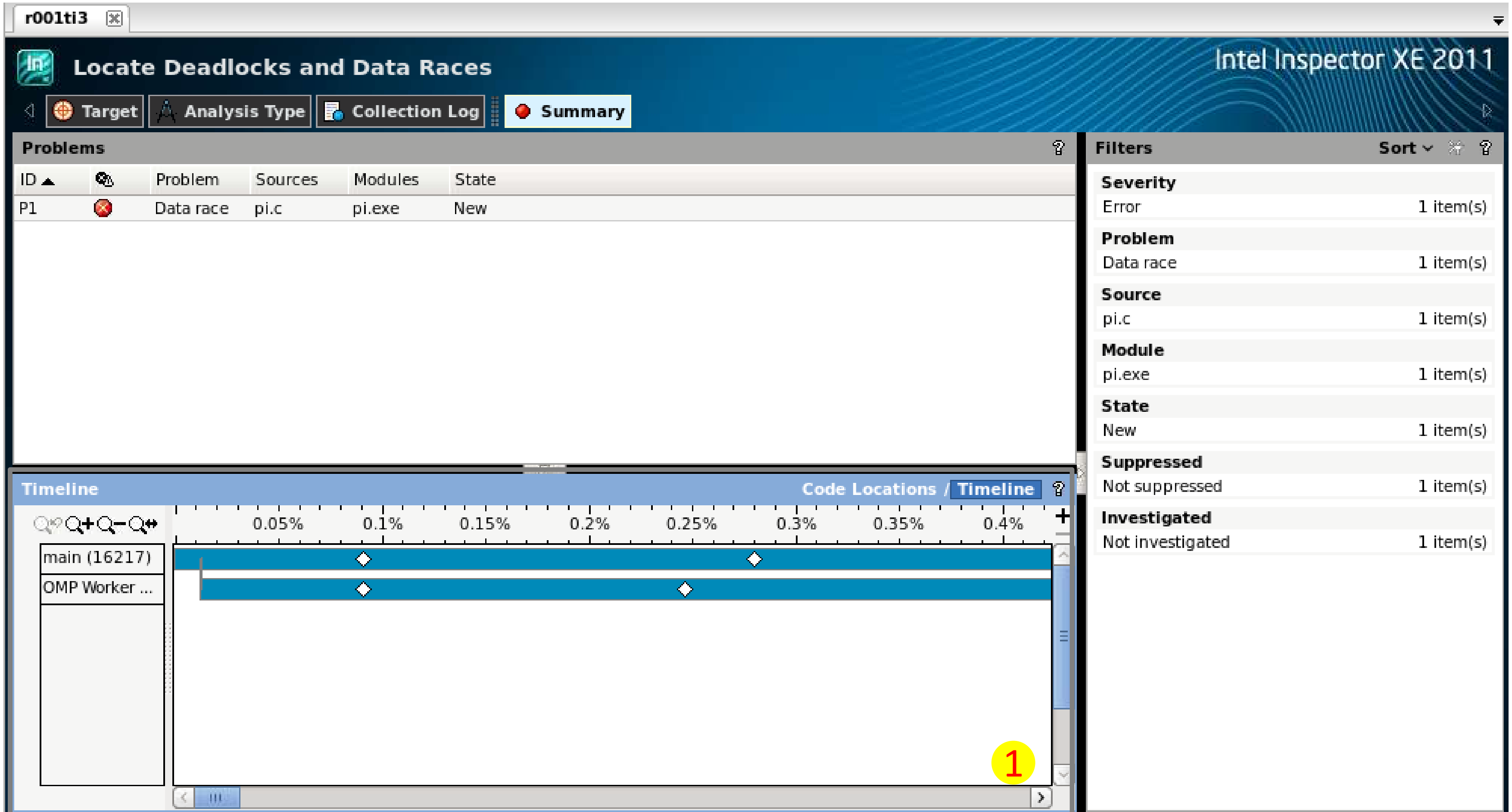
2 filters

3 code location



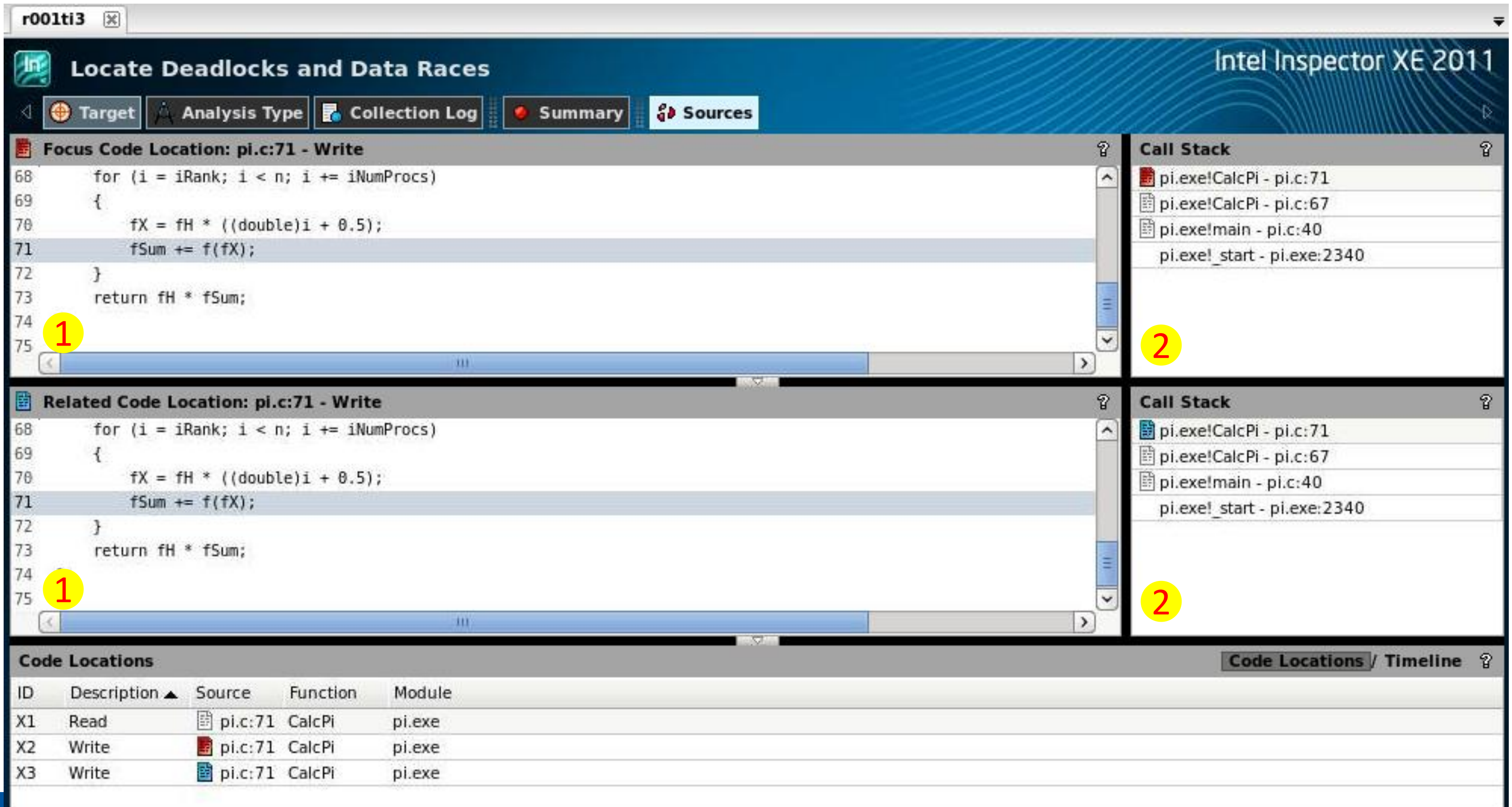
Inspector XE – Results

1 Timeline view



Inspector XE – Results

1 Source Code producing the issue – double click opens an editor
2 Corresponding Call Stack

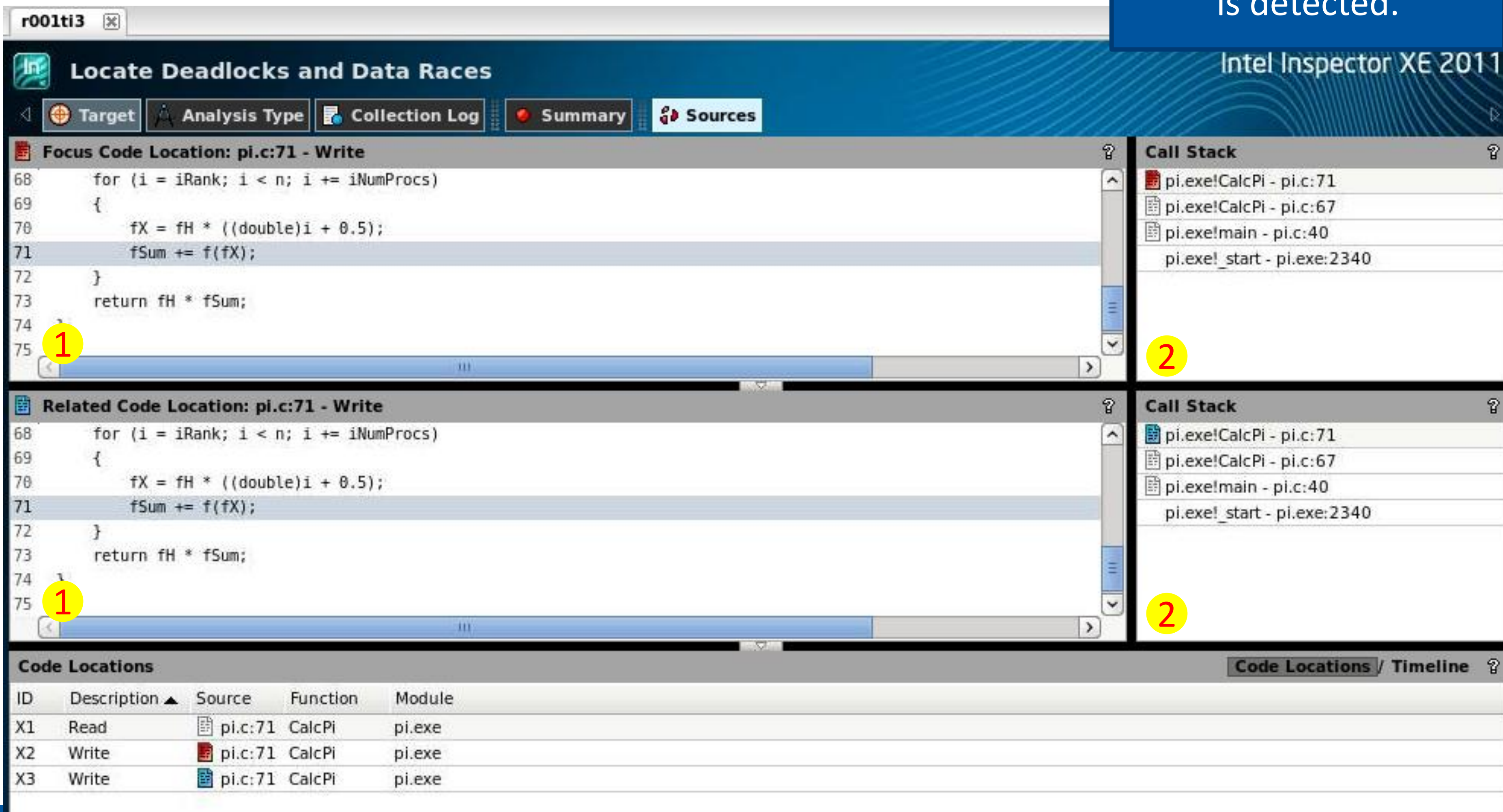


Inspector XE – Results

1 Source Code producing the issue – double click opens an editor

2 Corresponding Call Stack

The missing reduction is detected.



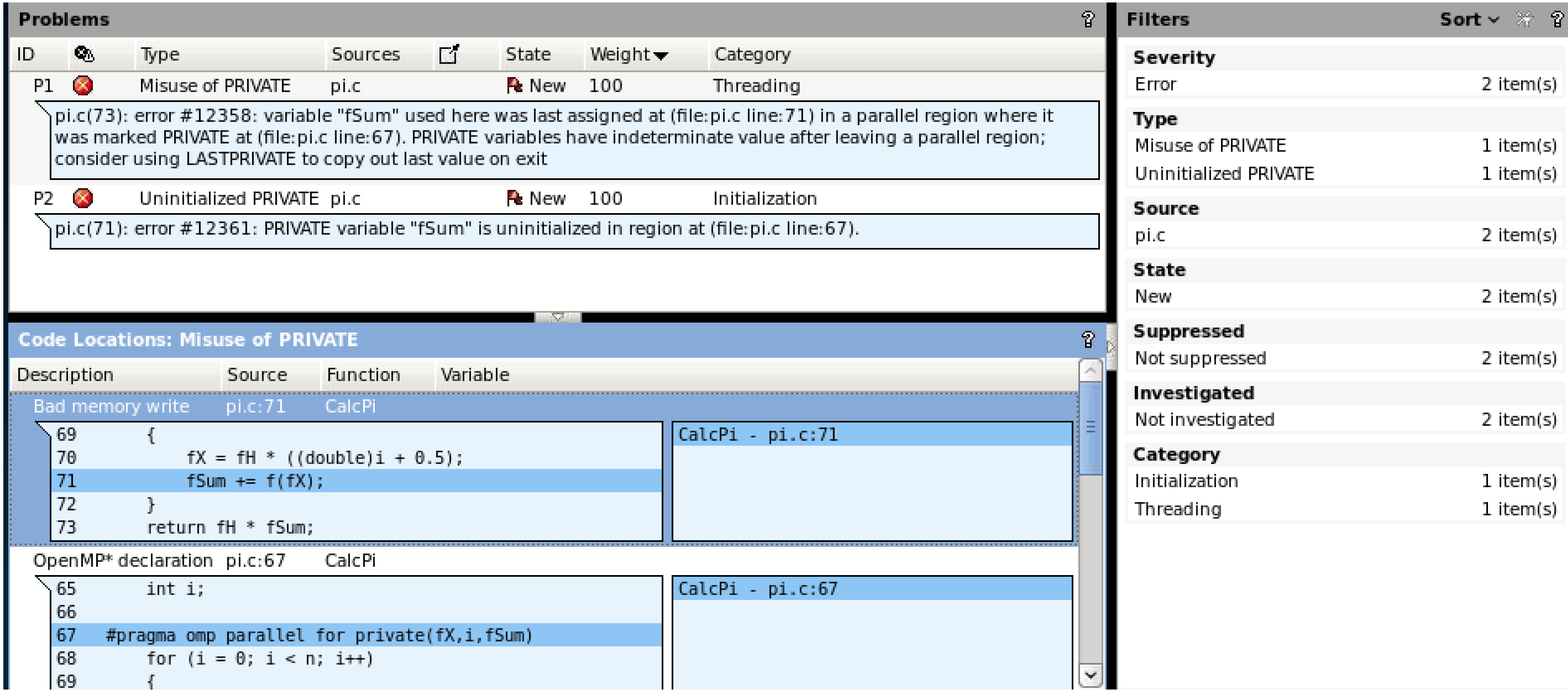
PI Example Code

```
double f(double x)
{
   return (4.0 / (1.0 + x*x));
}

double CalcPi (int n)
{
   const double fH   = 1.0 / (double) n;
   double fSum = 0.0;
   double fX;
   int i;

#pragma omp parallel for private(fX,i,fSum)
   for (i = 0; i < n; i++)
   {
      fX = fH * ((double)i + 0.5);
      fSum += f(fX);
   }
   return fH * fSum;
}
```

What if we just made the variable private?

Inspector XE – Static Security Analysis

- At runtime no Error is detected!
- Compiling with the argument “-diag-enable sc-full” delivers:

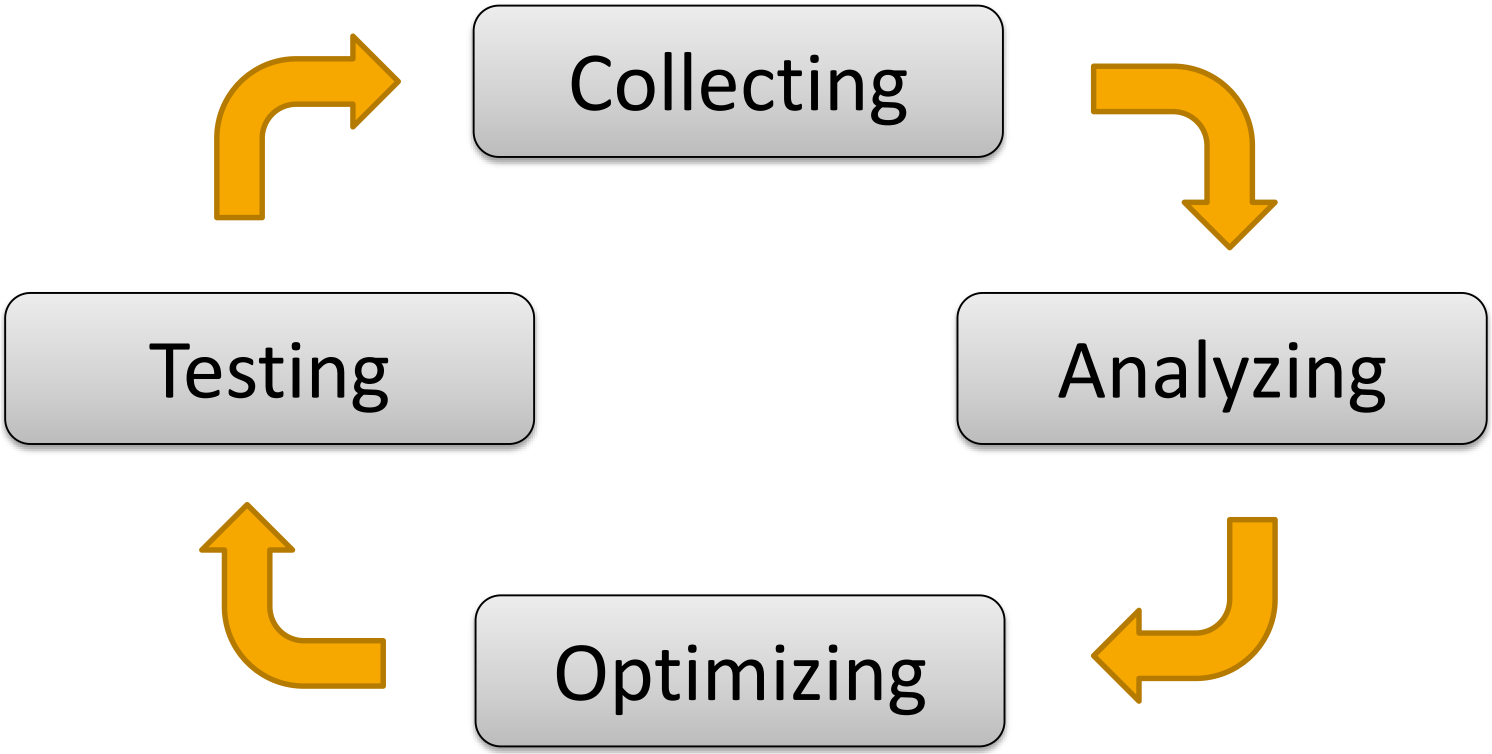


- At compile-time this error can be found!

Performance Tuning

Tuning Cycle

- **Performance Tuning aims to improve the runtime of an existing application.**



Hotspots

- **A Hotspot is a source code region where a significant part of the runtime is spent.**

90/10 law

90% of the runtime in a program is spent in 10% of the code.

- **Hotspots can indicate where to start with serial optimization or shared memory parallelization.**
- **Use a tool to identify hotspots. In many cases the results are surprising.**

Performance Tools

VTune Amplifier XE

- **Performance Analyses for**
  - → Serial Applications
  - → Shared Memory Parallel Applications
- **Sampling Based measurements**
- **Features:**
  - → Hot Spot Analysis
  - → Concurrency Analysis
  - → Wait
  - → Hardware Performance Counter Support

Stream

- Standard Benchmark to measure memory performance.
- Version is parallelized with OpenMP.

```
#pragma omp parallel for
        for (j=0; j<N; j++)
            b[j] = scalar*c[j];
```

Measures Memory bandwidth for:

y=x (copy)

y=s*x (scale)

y=x+z (add)

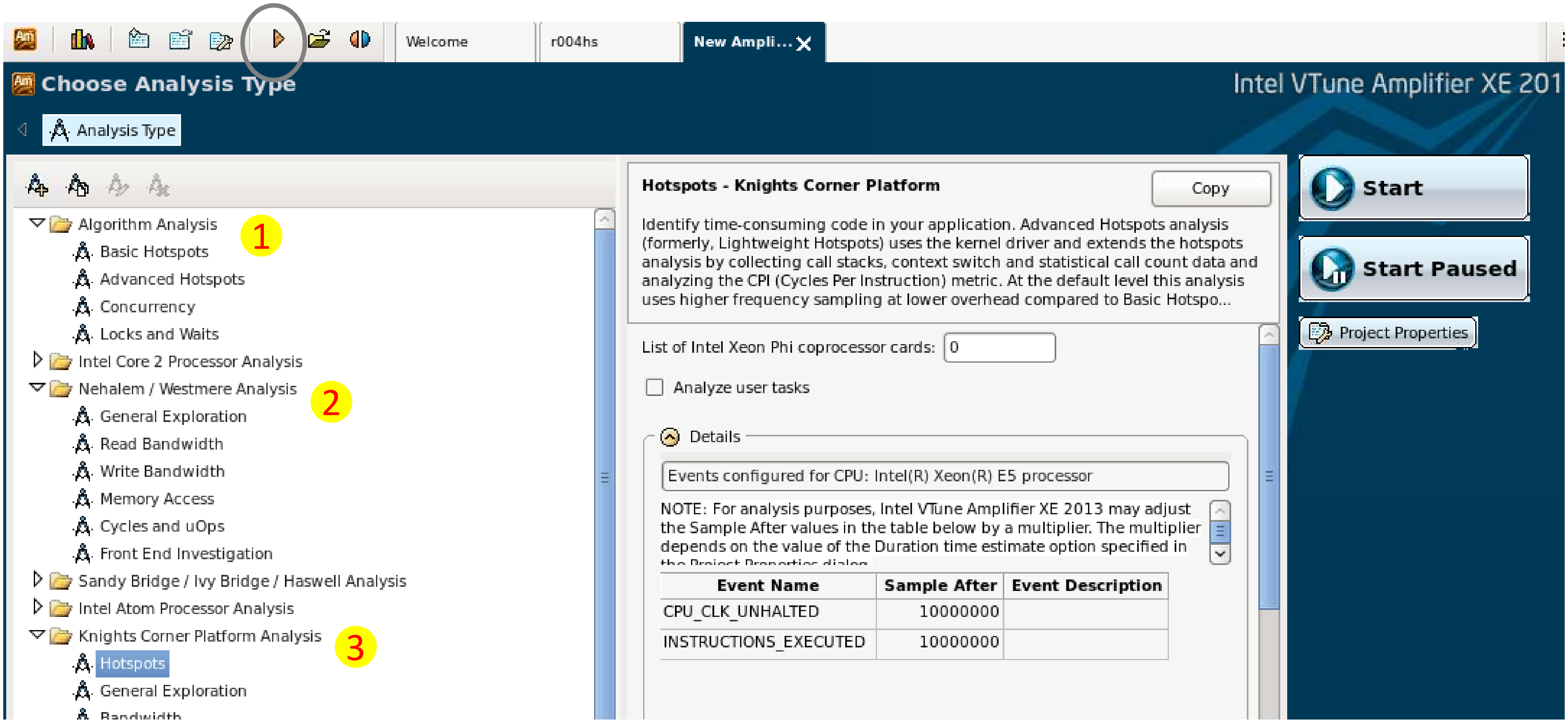
y=x+s*z (triad)

for double vectors x,y,z and scalar double value s

```
-------------------------------------------------------------
Function      Rate (MB/s)   Avg time     Min time     Max time
Copy:        33237.0185     0.0050       0.0048       0.0055
Scale:       33304.6471     0.0049       0.0048       0.0059
Add:         35456.0586     0.0070       0.0068       0.0073
Triad:       36030.9600     0.0069       0.0067       0.0072
```

Amplifier XE – Measurement Runs

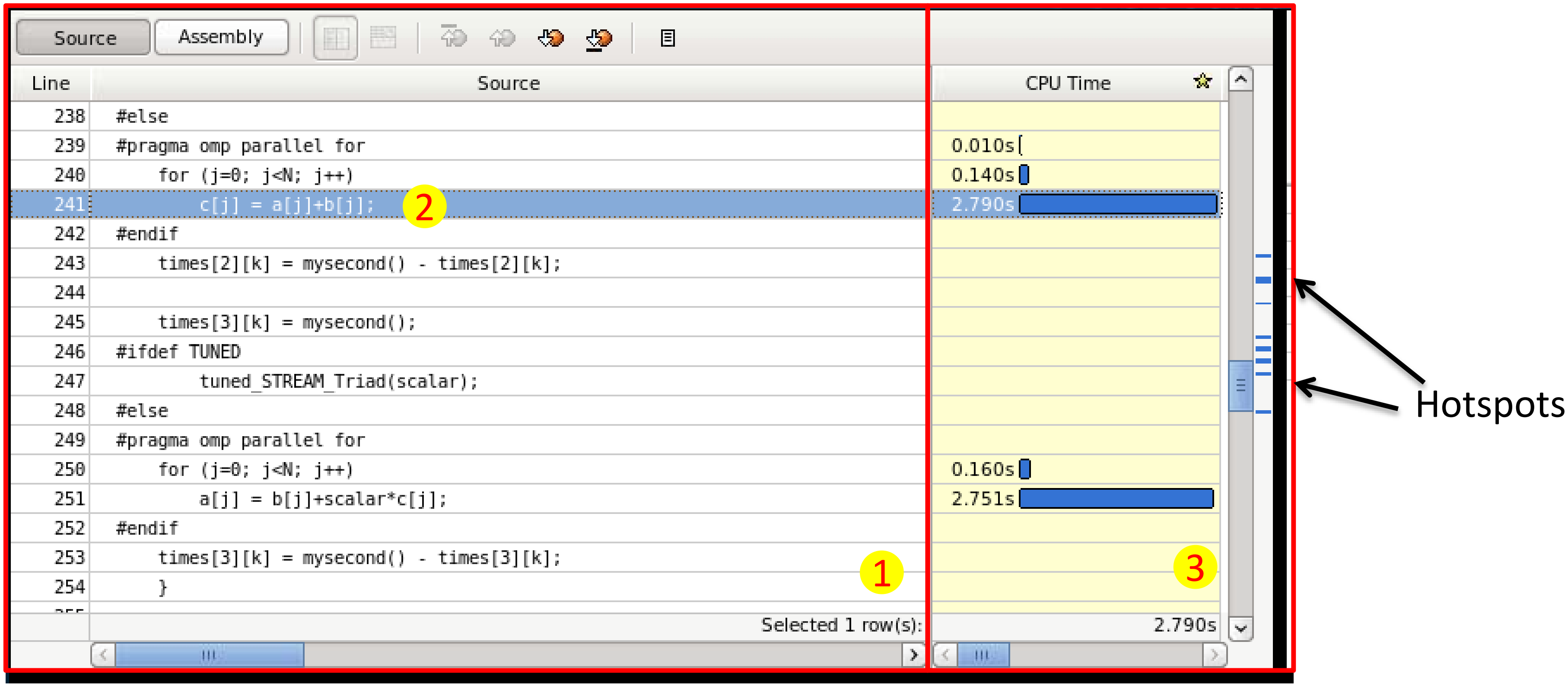
1. Basic Analysis Types
2. Hardware Counter Analysis Types, choose Nehalem Architecture, on cluster-linux-tuning.
3. Analysis for Intel Xeon Phi coprocessors, choose this for OpenMP target programs.



Amplifier XE – Hotspot Analysis

Double clicking on a function opens source code view.

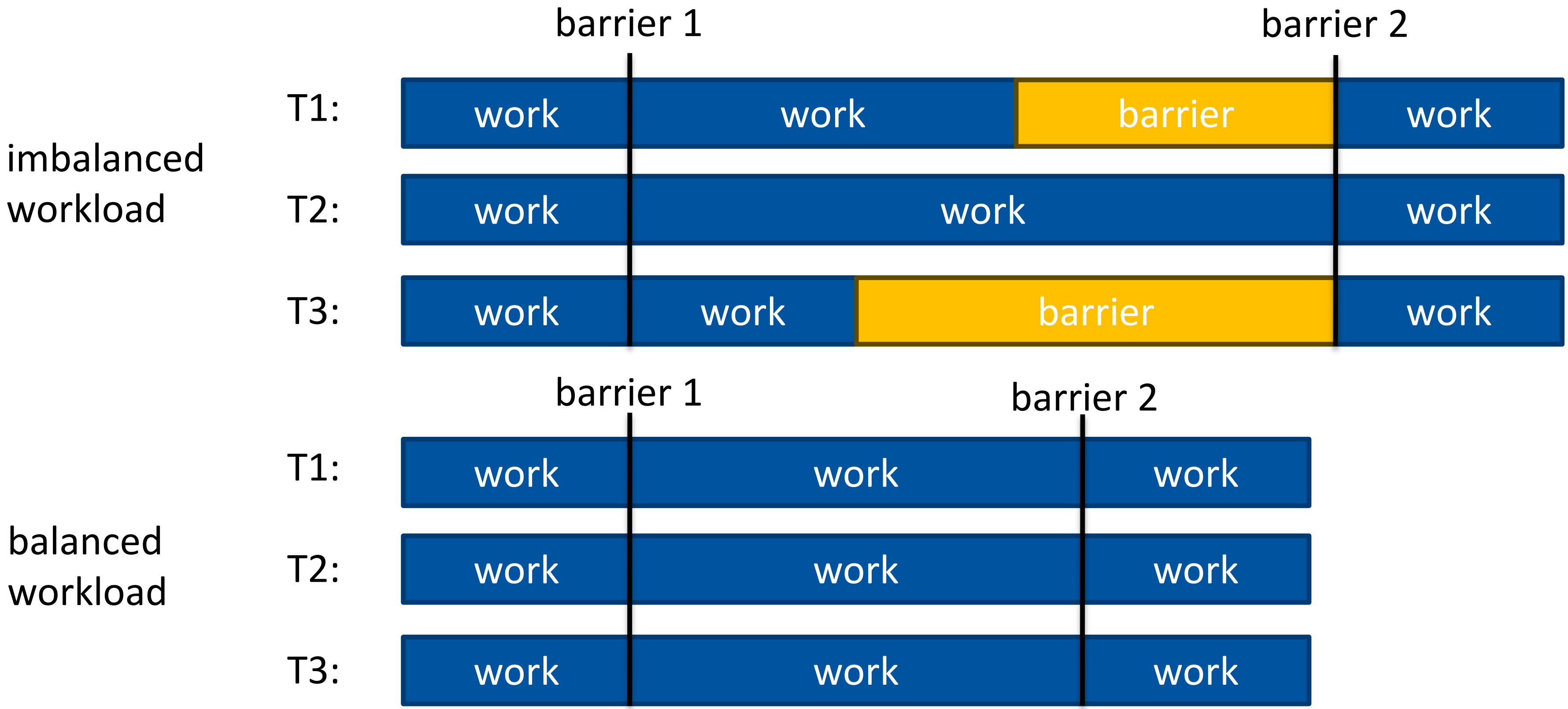
1. Source Code View (only if compiled with -g)
2. Hotspot: Add Operation of Stream
3. Metrics View



Load Balancing

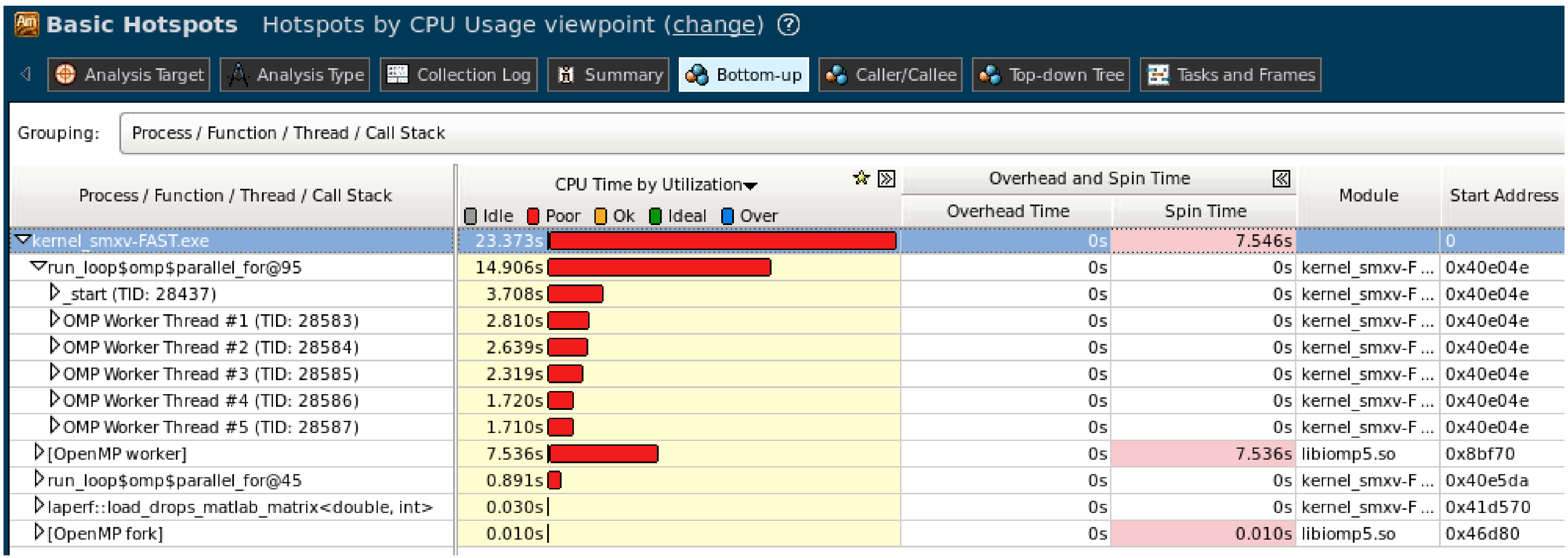
Load imbalance

- **Load imbalance occurs in a parallel program**
  - → when multiple threads synchronize at global synchronization points
  - → and these threads need a different amount of time to finish the calculation.



Load Imbalance in VTune

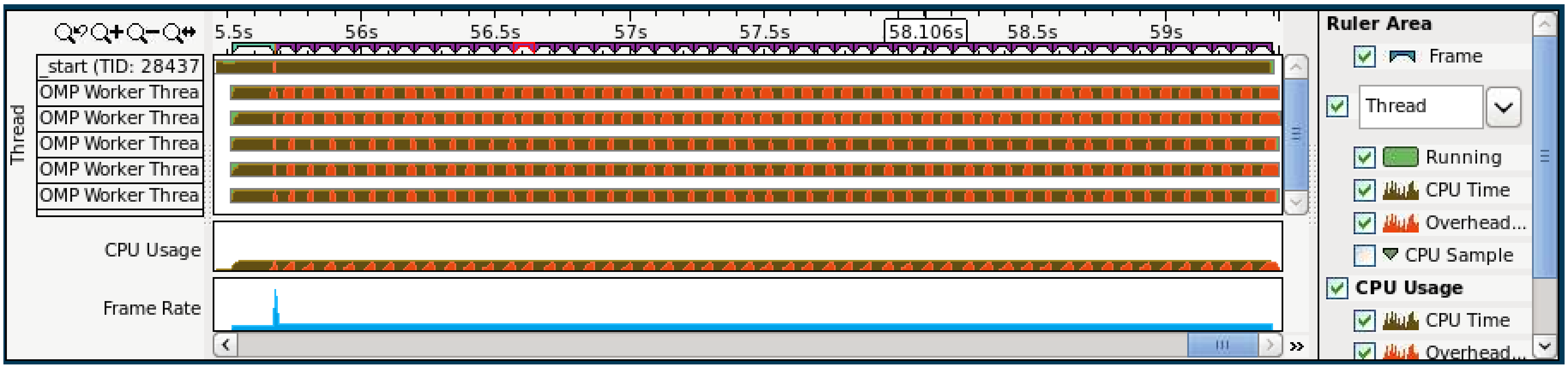
- Grouping execution time of parallel regions by threads helps to detect load imbalance.
- Significant potions of Spin Time also indicate load balance problems.
- Different loop schedules might help to avoid these problems.



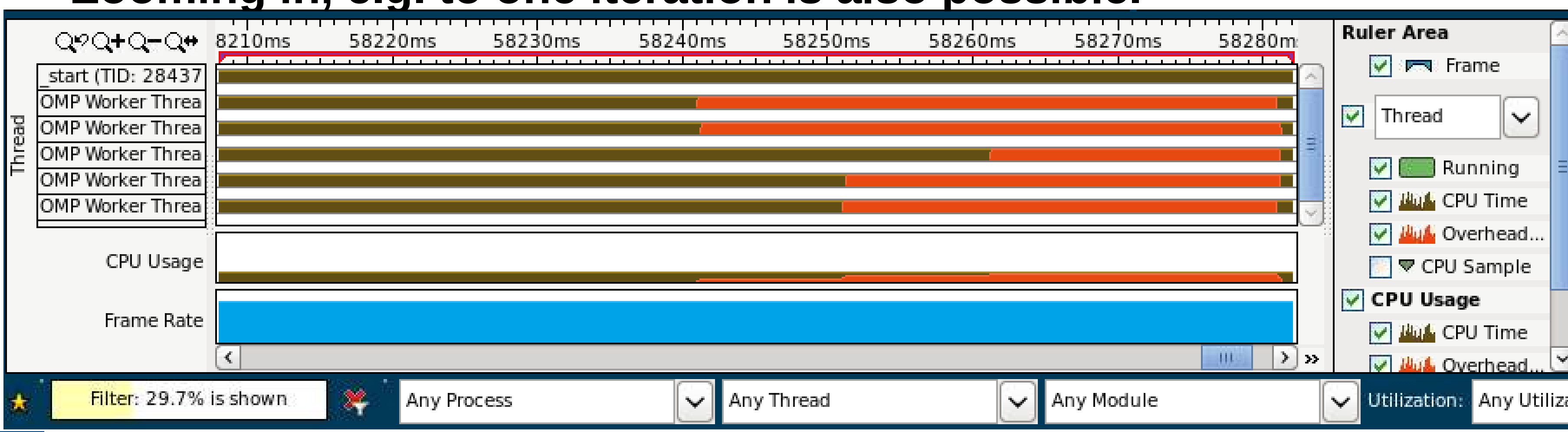
Process / Function / Thread / Call Stack	CPU Time by Utilization	Overhead and Spin Time: Overhead Time	Overhead and Spin Time: Spin Time	Module	Start Address
kernel_smxv-FAST.exe	23.373s	0s	7.546s		0
run_loopompparallel_for@95	14.906s	0s	0s	kernel_smxv-F ...	0x40e04e
_start (TID: 28437)	3.708s	0s	0s	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.



Detecting remote accesses

Hardware Counters

Definition: Hardware Performance Counters
In computers, hardware performance counters, or hardware counters are a set of special-purpose registers built into modern microprocessors to store the counts of hardware-related activities within computer systems. Advanced users often rely on those counters to conduct low-level performance analysis or tuning.
(from: http://en.wikipedia.org)

Hardware Performance Counters

Hardware Counters of our Intel Nehalem Processor:

SB_DRAIN.ANY, STORE_BLOCKS.AT_RET, STOF DAD_MISSES.PDE_MIS, DTLB_LOAD_MISSES.LARGE_W,
MEM_INST_RETIRED.LOADS, MEM_INST_RET MEM_UNCORE_RETIRED.L3_D,
MEM_UNCORE_RETIRED.OTHE, MEM_UNCOF MP_OPS_EXE.MMX, FP_COMP_OPS_EXE.SSE_FP,
FP_COMP_OPS_EXE.SSE2_INT, FP_COMP_OP NT_128.PACKED_SHIFT, SIMD_INT_128.PACK,
SIMD_INT_128.UNPACK, SIMD_INT_128.PACK B, LOAD_DISPATCH.ANY, ARITH.CYCLES_DIV_BUSY,
INST_QUEUE_WRITES, INST_DECODED.DEC0, D_HIT, L2_RQSTS.RFO_MISS, L2_RQSTS.RFOS,
L2_RQSTS.IFETCH_HIT, L2_RQSTS.IFETCH_MIS S.DEMAND.S_S, L2_DATA_RQSTS.DEMAND.E_S,
L2_DATA_RQSTS.DEMAND.M_, L2_DATA_RQS A_RQSTS.PREFETCH.M, L2_WRITE.RFO.I_STATE,
L2_WRITE.RFO.S_STATE, L2_WRITE.RFO.M_ST E, L2_WRITE.LOCK.HIT, L2_WRITE.LOCK.MESI,
L1D_WB_L2.I_STATE, L1D_WB_L2.S_STATE, L1D_WB_L2.E_STATE, L1D_WB_L2.M_STATE, L1D_WB_L2.MESI, CPU_CLK_UNHALTED.THREAD, CPU_CLK_UNHALTED.REF_P, L1D_CACHE_LD.I_STATE, L1D_CACHE_LD.S_STATE, L1D_CACHE_LD.E_STATE,
L1D_CACHE_LD.M_STATE, L1D_CACHE_LD.MESI, L1D_CACHE_ST.S_STATE, L1D_CACHE_ST.E_STATE, L1D_CACHE_ST.M_STATE, L1D_CACHE_LOCK.HIT, L1D_CACHE_LOCK.S_STATE, L1D_CACHE_LOCK.E_STATE, L1D_CACHE_LOCK.M_STATE, L1D_ALL_REF.ANY,
L1D_ALL_REF.CACHEABLE, DTLB_MISSES.ANY, DTLB_MISSES.WALK_COMPLET, DTLB_MISSES.STLB_HIT, DTLB_MISSES.PDE_MISS, DTLB_MISSES.LARGE_WALK_C, LOAD_HIT_PRE, L1D_PREFETCH.REQUESTS, L1D_PREFETCH.MISS, L1D_PREFETCH.TRIGGERS,
L1D.M_REPL, L1D.M_EVICT, L1D.M_SNOOP_EVICT, L1D_CACHE_PREFETCH_LOCK, L1D_CACHE_LOCK_FB_HIT, CACHE_LOCK_CYCLES.L1D_L2, CACHE_LOCK_CYCLES.L1D, IO_TRANSACTIONS, L1I.CYCLES_STALLED, LARGE_ITLB.HIT, ITLB_MISSES.ANY,
ITLB_MISSES.WALK_COMPLET, ILD_STALL.LCP, ILD_STALL.MRU, ILD_STALL.IQ_FULL, ILD_STALL.REGEN, ILD_STALL.ANY, BR_INST_EXEC.COND, BR_INST_EXEC.DIRECT, BR_INST_EXEC.INDIRECT_NON, BR_INST_EXEC.NON_CALLS, BR_INST_EXEC.RETURN_NEA,
BR_INST_EXEC.DIRECT_NEAR, BR_INST_EXEC.INDIRECT_NEA, BR_INST_EXEC.NEAR_CALLS, BR_INST_EXEC.TAKEN, BR_MISP_EXEC.COND, BR_MISP_EXEC.DIRECT, BR_MISP_EXEC.INDIRECT_NO, BR_MISP_EXEC.NON_CALLS, BR_MISP_EXEC.RETURN_NEA,
BR_MISP_EXEC.DIRECT_NEAR, BR_MISP_EXEC.INDIRECT_NEA, BR_MISP_EXEC.NEAR_CALLS, BR_MISP_EXEC.TAKEN, RESOURCE_STALLS.ANY, RESOURCE_STALLS.LOAD, RESOURCE_STALLS.RS_FULL, RESOURCE_STALLS.STORE, RESOURCE_STALLS.ROB_FULL,
RESOURCE_STALLS.FPCW, RESOURCE_STALLS.MXCSR, RESOURCE_STALLS.OTHER, MACRO_INSTS.FUSIONS_DECO, BACLEAR_FORCE_IQ, ITLB_FLUSH, OFFCORE_REQUESTS.L1D_WR, UOPS_EXECUTED.PORT0, UOPS_EXECUTED.PORT1,
UOPS_EXECUTED.PORT2_COR, UOPS_EXECUTED.PORT3_COR, UOPS_EXECUTED.PORT4_COR, UOPS_EXECUTED.PORT5, UOPS_EXECUTED.PORT015, UOPS_EXECUTED.PORT234, OFFCORE_REQUESTS_SQ_FUL, OFF_CORE_RESPONSE_0, SNOOP_RESPONSE.HIT,
SNOOP_RESPONSE.HITE, SNOOP_RESPONSE.HITM, OFF_CORE_RESPONSE_1, INST_RETIRED.ANY_P, INST_RETIRED.X87, INST_RETIRED.MMX, UOPS_RETIRED.ANY, UOPS_RETIRED.RETIRE_SLOTS, UOPS_RETIRED.MACRO_FUSE, MACHINE_CLEARS.CYCLES,
MACHINE_CLEARS.MEM_ORDE, MACHINE_CLEARS.SMC, BR_INST_RETIRED.ALL_BRAN, BR_INST_RETIRED.CONDITION, BR_INST_RETIRED.NEAR_CAL, BR_MISP_RETIRED.ALL_BRAN, BR_MISP_RETIRED.NEAR_CAL, SSEX_UOPS_RETIRED.PACKED,
SSEX_UOPS_RETIRED.SCALAR, SSEX_UOPS_RETIRED.PACKED, SSEX_UOPS_RETIRED.SCALAR, SSEX_UOPS_RETIRED.VECTOR, ITLB_MISS_RETIRED, MEM_LOAD_RETIRED.L1D_HIT, MEM_LOAD_RETIRED.L2_HIT, MEM_LOAD_RETIRED.L3_UNS,
MEM_LOAD_RETIRED.OTHER_, MEM_LOAD_RETIRED.L3_MISS, MEM_LOAD_RETIRED.HIT_LFB, MEM_LOAD_RETIRED.DTLB_MI, FP_MMX_TRANS.TO_FP, FP_MMX_TRANS.TO_MMX, FP_MMX_TRANS.ANY, MACRO_INSTS.DECODED, UOPS_DECODED.MS,
UOPS_DECODED.ESP_FOLDING, UOPS_DECODED.ESP_SYNC, RAT_STALLS.FLAGS, RAT_STALLS.REGISTERS, RAT_STALLS.ROB_READ_POR, RAT_STALLS.SCOREBOARD, RAT_STALLS.ANY, SEG_RENAME_STALLS, ES_REG_RENAMES, UOP_UNFUSION,
BR_INST_DECODED, BPU_MISSED_CALL_RET, BACLEAR.BAD_TARGET, BPU_CLEARS.EARLY, BPU_CLEARS.LATE, L2_TRANSACTIONS.LOAD, L2_TRANSACTIONS.RFO, L2_TRANSACTIONS.IFETCH, L2_TRANSACTIONS.PREFETCH, L2_TRANSACTIONS.L1D_WB,
L2_TRANSACTIONS.FILL, L2_TRANSACTIONS.WB, L2_TRANSACTIONS.ANY, L2_LINES_IN.S_STATE, L2_LINES_IN.E_STATE, L2_LINES_IN.ANY, L2_LINES_OUT.DEMAND_CLEA, L2_LINES_OUT.DEMAND_DIRT, L2_LINES_OUT.PREFETCH_CLE,
L2_LINES_OUT.PREFETCH_DIR, L2_LINES_OUT.ANY, SQ_MISC.SPLIT_LOCK, SQ_FULL_STALL_CYCLES, FP_ASSIST.ALL, FP_ASSIST.OUTPUT, FP_ASSIST.INPUT, SIMD_INT_64.PACKED_MPY, SIMD_INT_64.PACKED_SHIFT, SIMD_INT_64.PACK,
SIMD_INT_64.UNPACK, SIMD_INT_64.PACKED_LOGICA, CPUID, SIMD_INT_64.PACKED_ARITH, SIMD_INT_64.SHUFFLE_MOVE, UNC_GQ_CYCLES_FULL.READ_, UNC_GQ_CYCLES_FULL.WRITE, UNC_GQ_CYCLES_FULL.PEER_, UNC_GQ_CYCLES_NOT_EMPTY,
UNC_GQ_CYCLES_NOT_EMPTY, UNC_GQ_CYCLES_NOT_EMPTY, UNC_GQ_ALLOC.READ_TRACK, UNC_GQ_ALLOC.RT_L3_MISS, UNC_GQ_ALLOC.RT_TO_L3_RE, UNC_GQ_ALLOC.RT_TO_RTID_, UNC_GQ_ALLOC.WT_TO_RTID, UNC_GQ_ALLOC.WRITE_TRAC,
UNC_GQ_ALLOC.PEER_PROBE, UNC_GQ_DATA.FROM_QPI, UNC_GQ_DATA.FROM_QMC, UNC_GQ_DATA.FROM_L3, UNC_GQ_DATA.FROM_CORES_, UNC_GQ_DATA.FROM_CORES_, UNC_GQ_DATA.TO_QPI_QMC, UNC_GQ_DATA.TO_L3,
UNC_GQ_DATA.TO_CORES, UNC_SNP_RESP_ P_RESP_TO_LOCAL_H, UNC_SNP_RESP_TO_REMOTE,
UNC_SNP_RESP_TO_REMOTE, UNC_SNP_RES HITS.READ, UNC_L3_HITS.WRITE, UNC_L3_HITS.PROBE,
UNC_L3_HITS.ANY, UNC_L3_MISS.READ, UNC NC_L3_LINES_IN.F_STATE, UNC_L3_LINES_IN.ANY,
UNC_L3_LINES_OUT.M_STATE, UNC_L3_LINES QUESTS.IOH_RE, UNC_QHL_REQUESTS.IOH_WR,
UNC_QHL_REQUESTS.REMOTE, UNC_QHL_RE IL_CYCLES_FULL.LOCA, UNC_QHL_CYCLES_NOT_EMPT,
UNC_QHL_CYCLES_NOT_EMPT, UNC_QHL_CY _QHL_ADDRESS_CONFLIC, UNC_QHL_CONFLICT_CYCLES.I,
UNC_QHL_CONFLICT_CYCLES., UNC_QHL_CO MC_NORMAL_FULL.WRI, UNC_QMC_NORMAL_FULL.WRI,
UNC_QMC_NORMAL_FULL.WRI, UNC_QMC_ UNC_QMC_ISOC_FULL.WRITE.C, UNC_QMC_BUSY.READ.CH0,
UNC_QMC_BUSY.READ.CH1, UNC_QMC_BUS CUPANCY.CH1, UNC_QMC_OCCUPANCY.CH2,
UNC_QMC_ISSOC_OCCUPANCY., UNC_QMC_ C, UNC_QMC_NORMAL_READS.C,
UNC_QMC_NORMAL_READS.A, UNC_QMC_H UNC_QMC_CRITICAL_PRIORIT, UNC_QMC_CRITICAL_PRIORIT,
UNC_QMC_CRITICAL_PRIORIT, UNC_QMC_W MC_WRITES.PARTIAL.C, UNC_QMC_WRITES.PARTIAL.C,
UNC_QMC_CANCEL.CH0, UNC_QMC_CANCEL ATE, UNC_QMC_PRIORITY_UPDATE,
UNC_QHL_FRC_ACK_CNFLTS.L, UNC_QPI_TX PI_TX_STALLED_SINGL, UNC_QPI_TX_STALLED_SINGL,
UNC_QPI_TX_STALLED_SINGL, UNC_QPI_TX QPI_TX_STALLED_MULTI, UNC_QPI_TX_STALLED_MULTI,
UNC_QPI_TX_HEADER.BUSY.LI, UNC_QPI_TX_HEADER.BUSY.LI, UNC_QPI_RX_NO_PPT_CREDI, UNC_QPI_RX_NO_PPT_CREDI, UNC_DRAM_OPEN.CH0, UNC_DRAM_OPEN.CH1, UNC_DRAM_OPEN.CH2, UNC_DRAM_PAGE_CLOSE.CH0,
UNC_DRAM_PAGE_CLOSE.CH1, UNC_DRAM_PAGE_CLOSE.CH2, UNC_DRAM_PAGE_MISS.CH0, UNC_DRAM_PAGE_MISS.CH1, UNC_DRAM_PAGE_MISS.CH2, UNC_DRAM_READ_CAS.CH0, UNC_DRAM_READ_CAS.AUTO, UNC_DRAM_READ_CAS.CH1,
UNC_DRAM_READ_CAS.AUTO, UNC_DRAM_READ_CAS.CH2, UNC_DRAM_READ_CAS.AUTO, UNC_DRAM_WRITE_CAS.CH0, UNC_DRAM_WRITE_CAS.AUTO, UNC_DRAM_WRITE_CAS.CH1, UNC_DRAM_WRITE_CAS.AUTO, UNC_DRAM_WRITE_CAS.CH2,
UNC_DRAM_WRITE_CAS.AUTO, UNC_DRAM_REFRESH.CH0

L1I.HITS:
Counts all instruction fetches that hit the L1 instruction cache.

BR_MISP_EXEC.COND:
Counts the number of mispredicted conditional near branch instructions executed, but not necessarily retired.

Derived Metrics

- **Clock cycles per Instructions (CPI)**
  - → CPI indicates if the application is utilizing the CPU or not
  - → Take care: Doing “something” does not always mean doing “something useful”.
- **Floating Point Operations per second (FLOPS)**
  - → How many arithmetic operations are done per second?
  - → Floating Point operations are normally really computing and for some algorithms the number of floating point operations needed can be determined.

Hardware Performance Counters

1 **CPI rate (Clock cycles per instruction): In theory modern processors can finish 4 instructions in 1 cycle, so a CPI rate of 0.25 is possible. A value between 0.25 and 1 is often considered as good for HPC applications.**

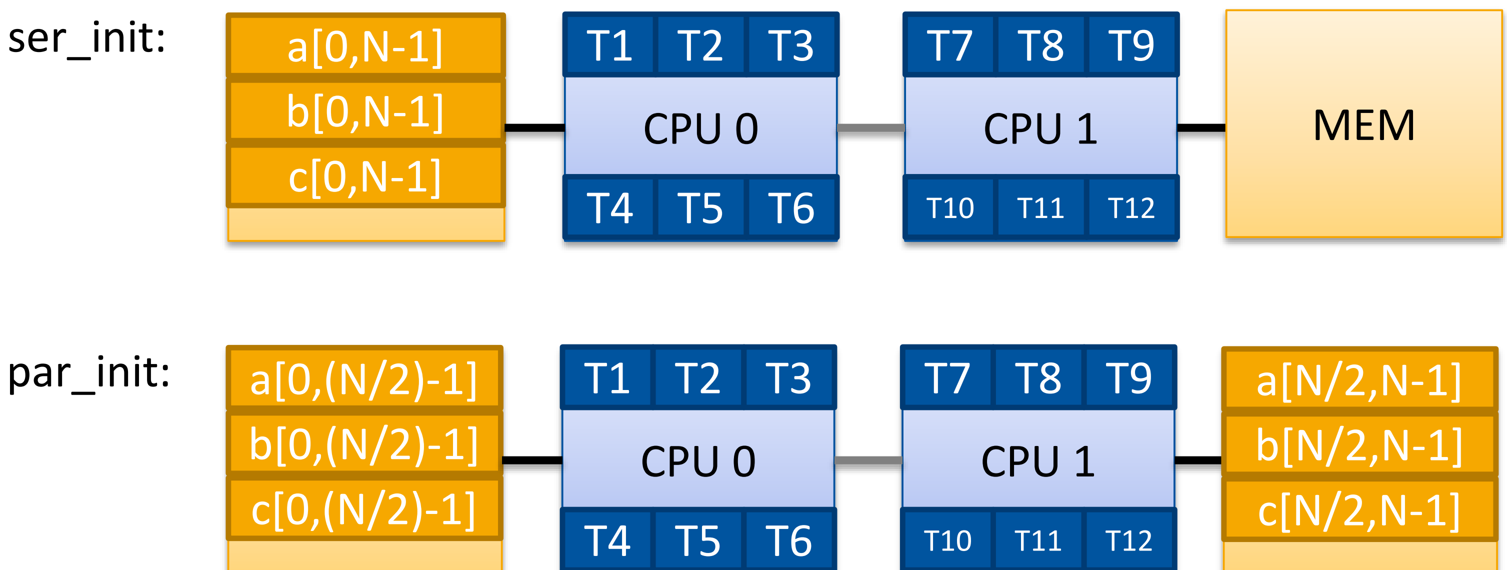
Elapsed Time: 1.872s

Hardware Event Count:	125,574,000,000
CPU_CLK_UNHALTED.THREAD:	6.3462e+10
INST_RETIRED.ANY:	6.2112e+10
1 CPI Rate:	1.022
The CPI may be too high. This could be caused by issues such as memo instructions. Explore the other hardware-related metrics to identify wha	
Retire Stalls:	0.570s
A high number of retire stalls is detected. This may result from branch n issues. Use this metric to find where you have stalled instructions. Once	
LLC Miss:	0.013s
LLC Load Misses Serviced By Remote DRAM:	0.001s
Instruction Starvation:	0.098s
Branch Mispredict:	0.001s
Execution Stalls:	0.288s

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



Counters for Remote Traffic

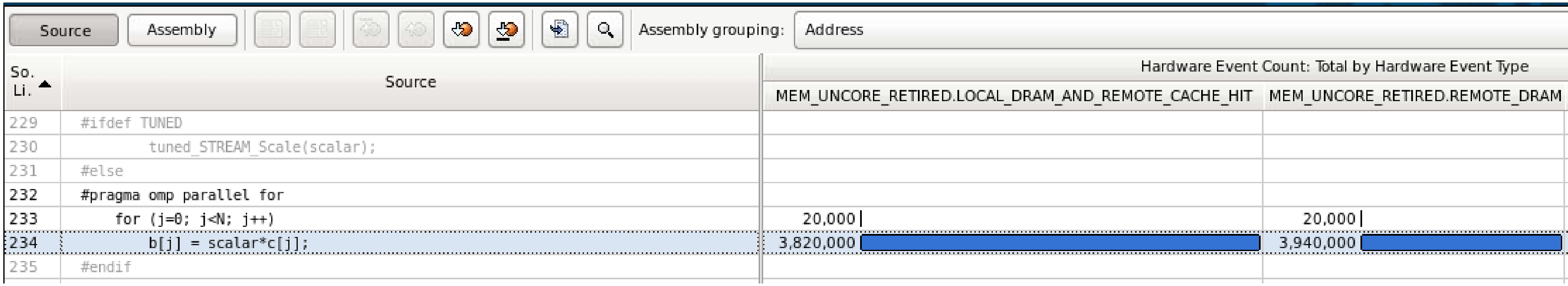
- **Hardware counters can measure local and remote memory accesses.**
  - → MEM_UNCORE_RETIRED.LOCAL_DRAM_AND_REMOTE_CACHE_HIT

    accesses to local memory
  - → MEM_UNCORE_RETIRED.REMOTE_DRAM

    accesses to remote memory

- **Absolute values are hard to interpret, but the ratio between both is useful.**

- **Detecting bad memory accesses for the stream benchmark.**



- **Ratio of remote memory accesses:**

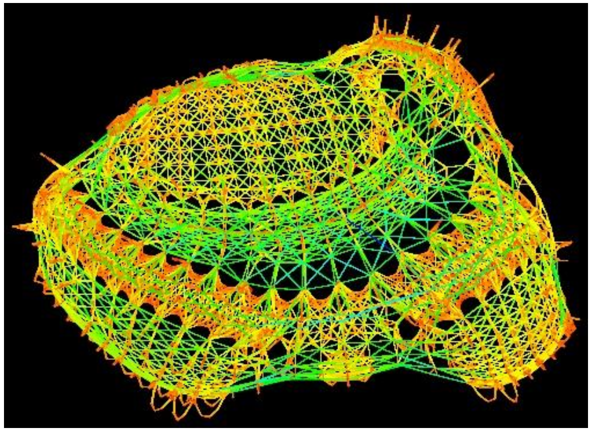
	copy	scale	add	triad
ser_init	52%	50%	50%	51%
par_init	0.5%	1.7%	0.6%	0.2%

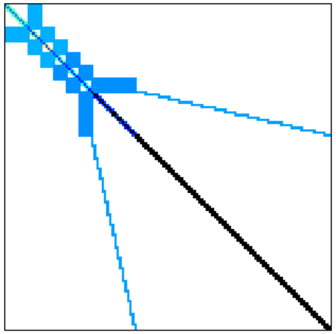
Percentage of remote accesses for ser_init and par_init stream benchmark.

CG Solver

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: 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.

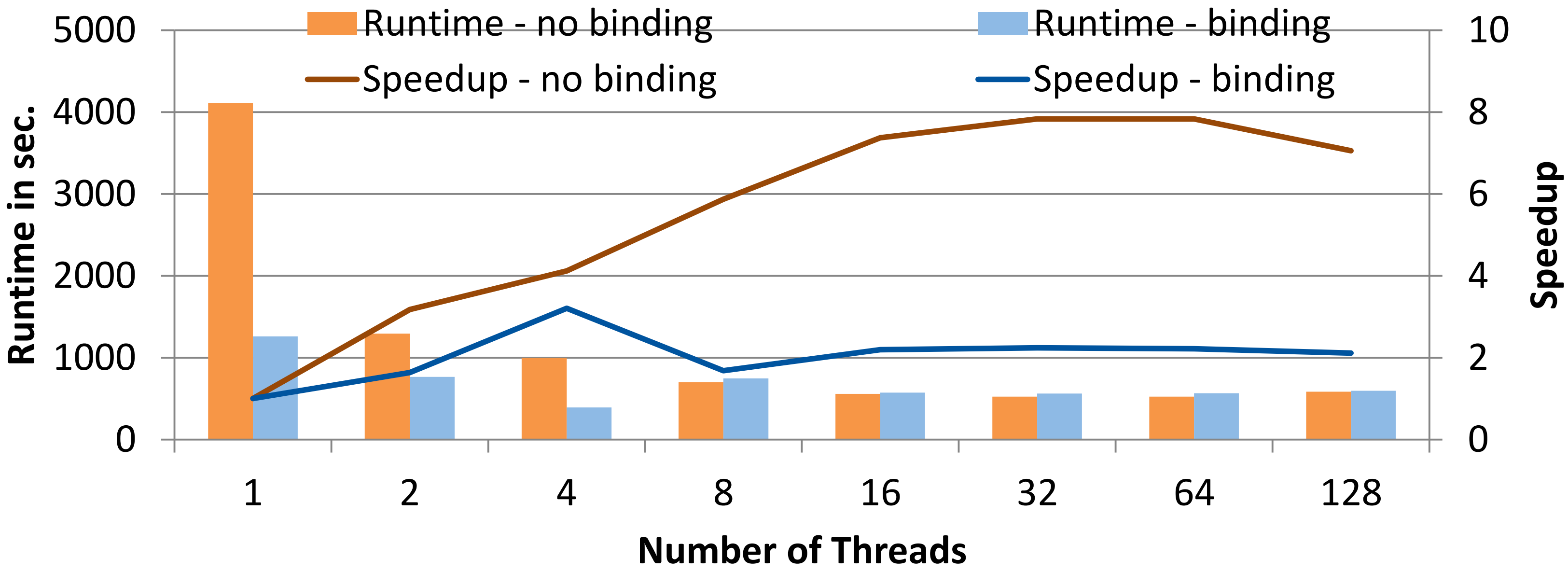
Hotspots are:

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

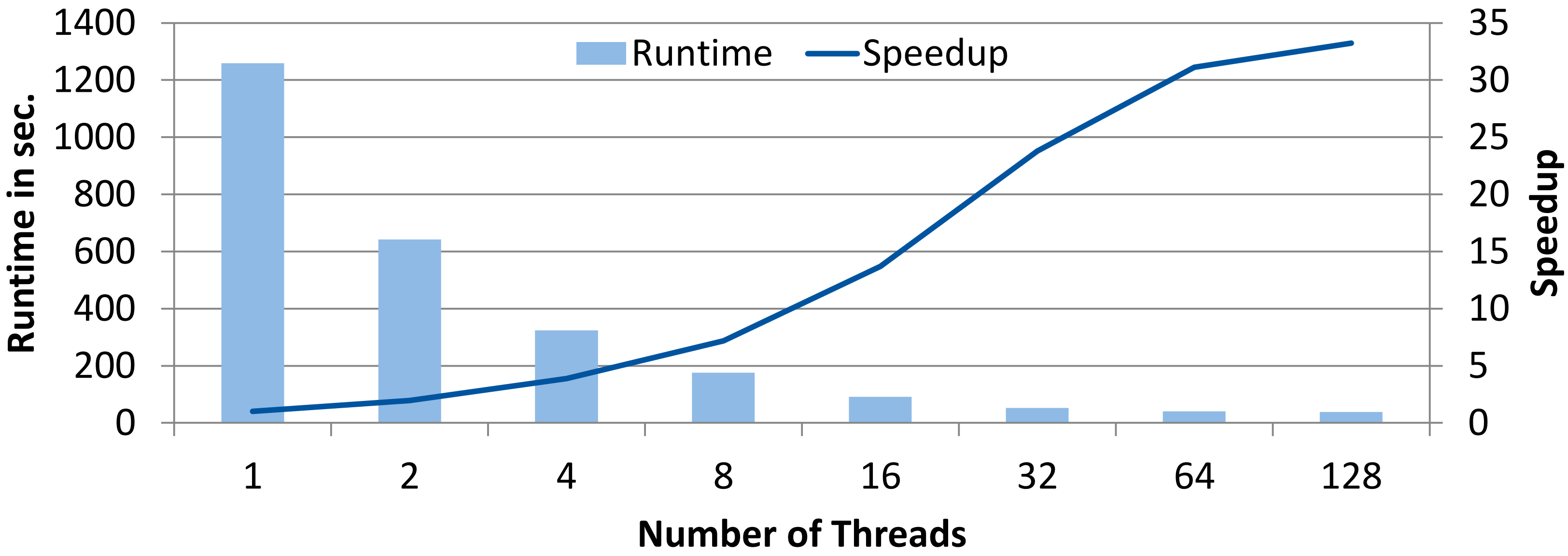
A lot of remote accesses occur in nearly all places.

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.**

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**

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;
}
```

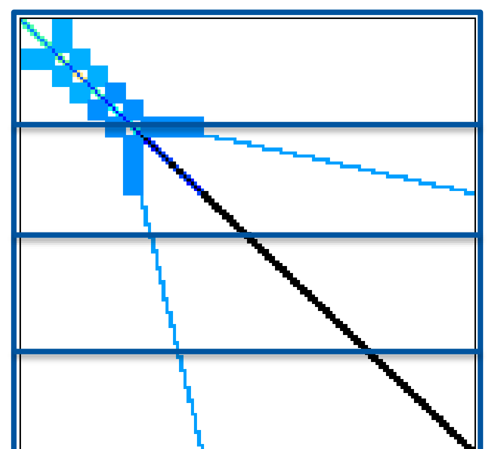
$\boxed{\vec{y} = A * \vec{x}}$

- **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)**

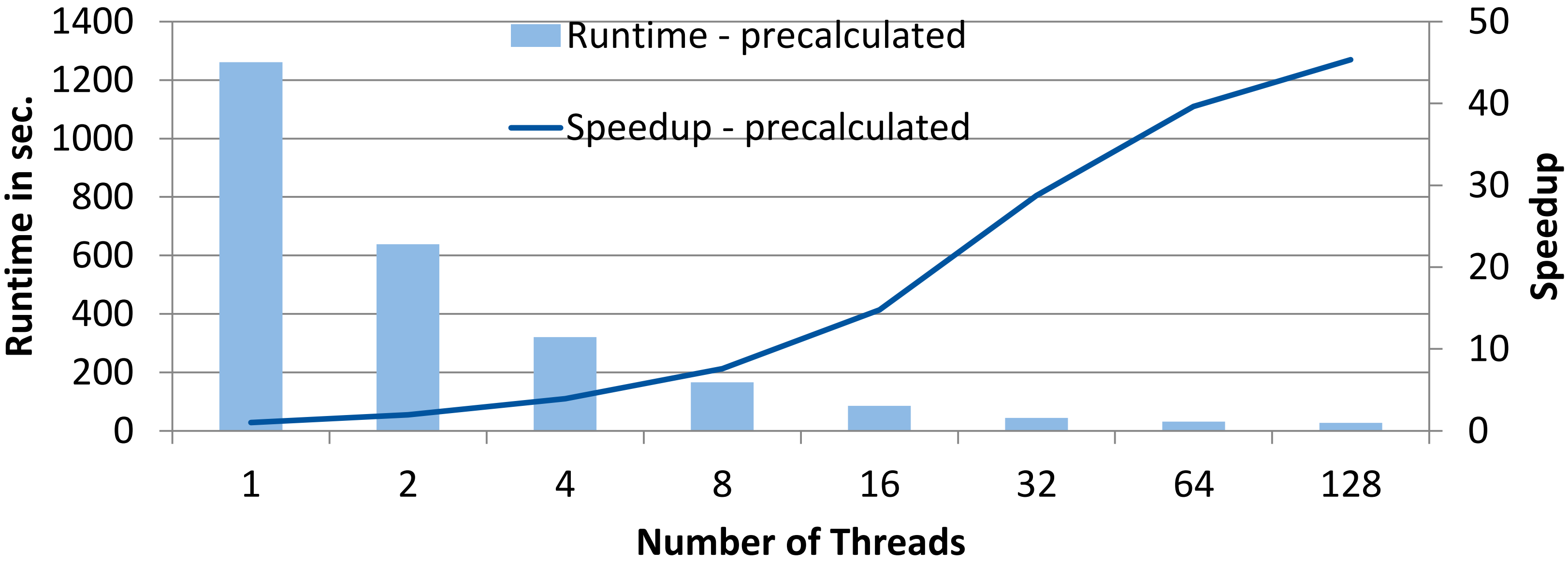
value:	1	2	2	3	4	4	4
index:	0	0	1	2	0	2	3

row:	0	1	3	4	7



Case Study CG: Step 3

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



The Roofline Model

When to stop tuning?

- **Depends on many different factors:**
  → How often is the code program used?

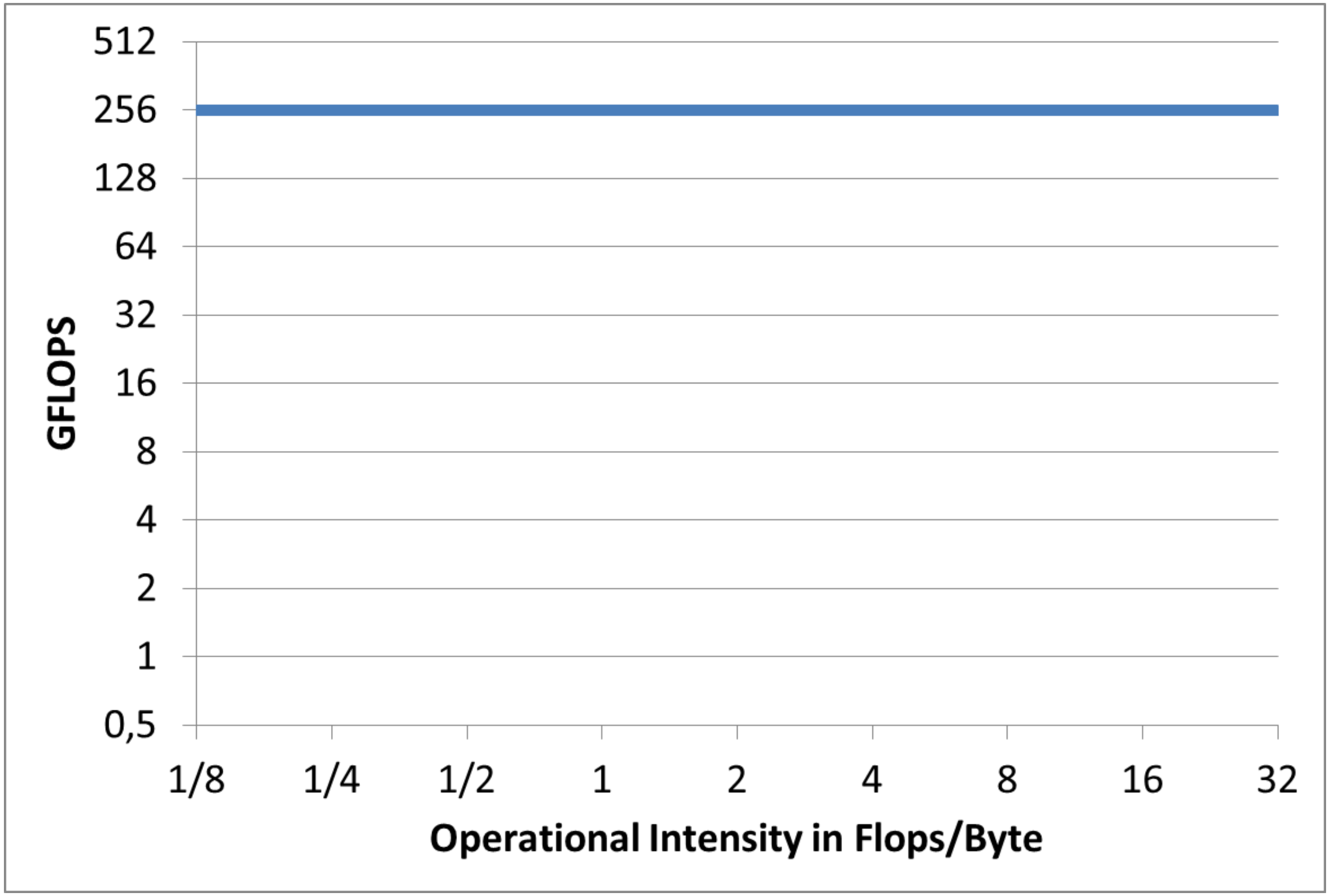
  → What are the runtime requirements?

  → Which performance can I expect?

- **Investigating kernels may help to understand larger applications.**

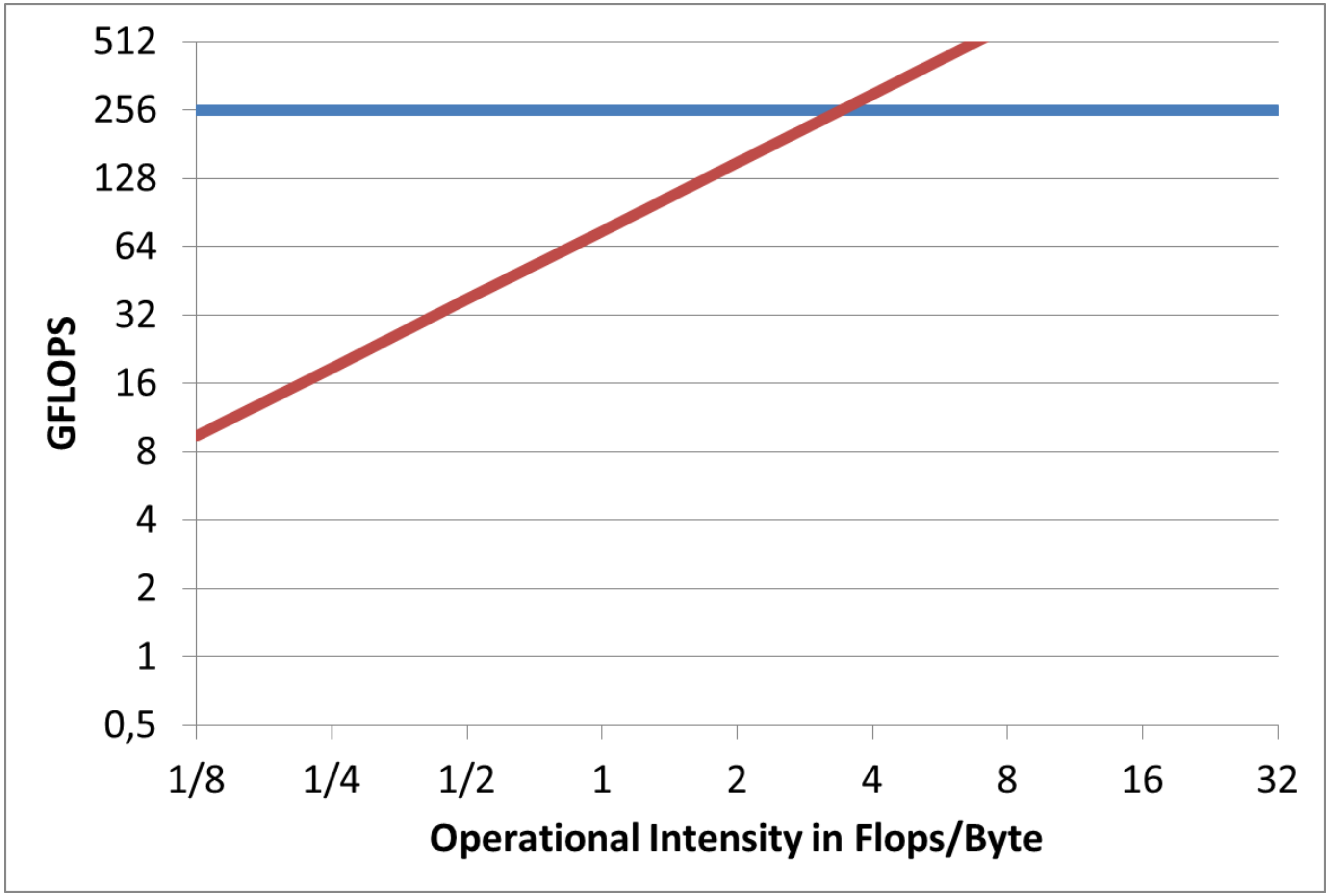
Roofline Model

- **Peak performance of a 4 socket Nehalem Server is 256 GFLOPS.**



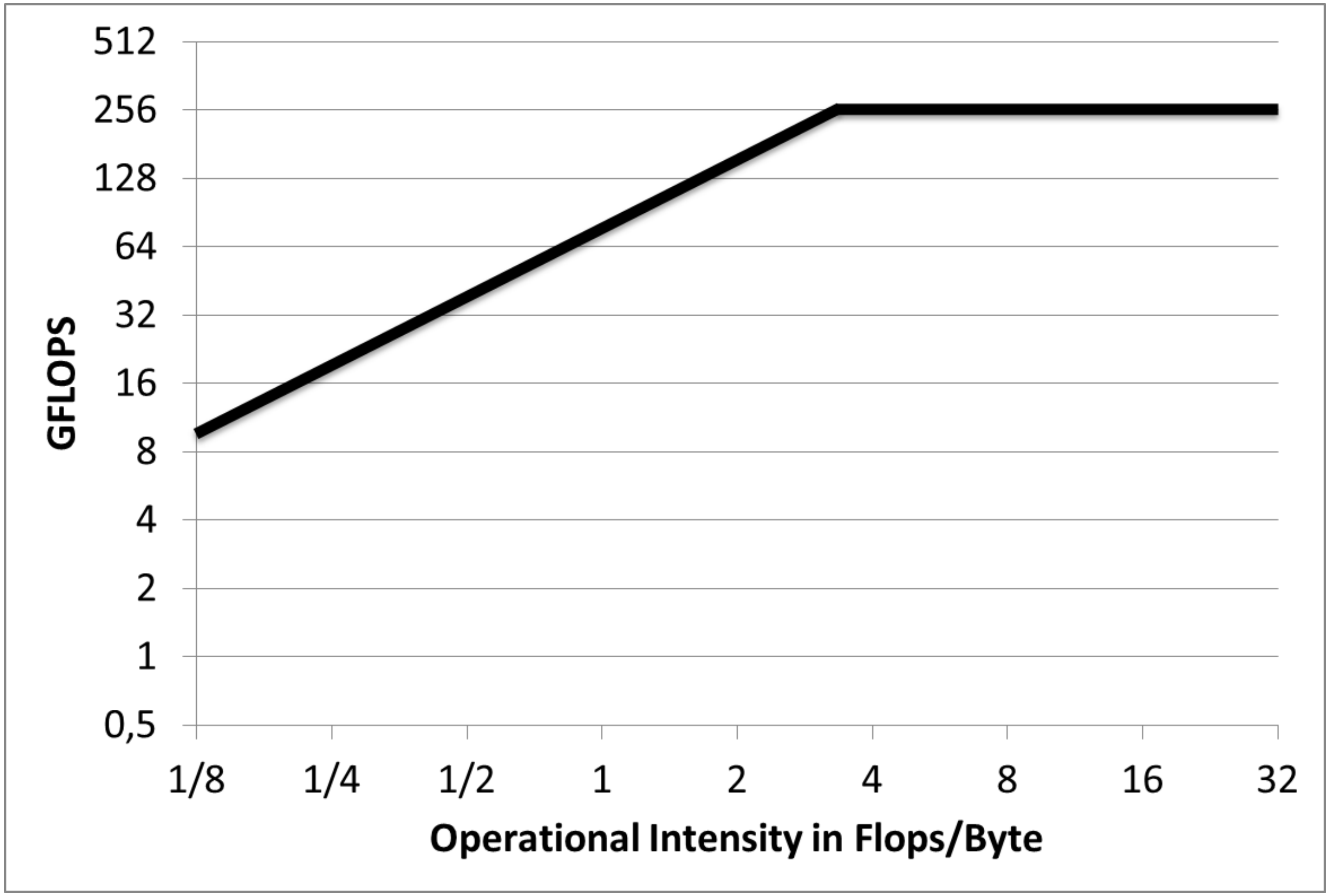
Roofline Model

- **Memory bandwidth measured with Stream benchmark is about 75 GB/s.**



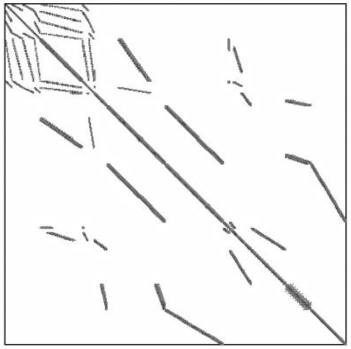
Roofline Model

- **The "Roofline" describes the peak performance the system can reach depending on the "operational intensity" of the algorithm.**



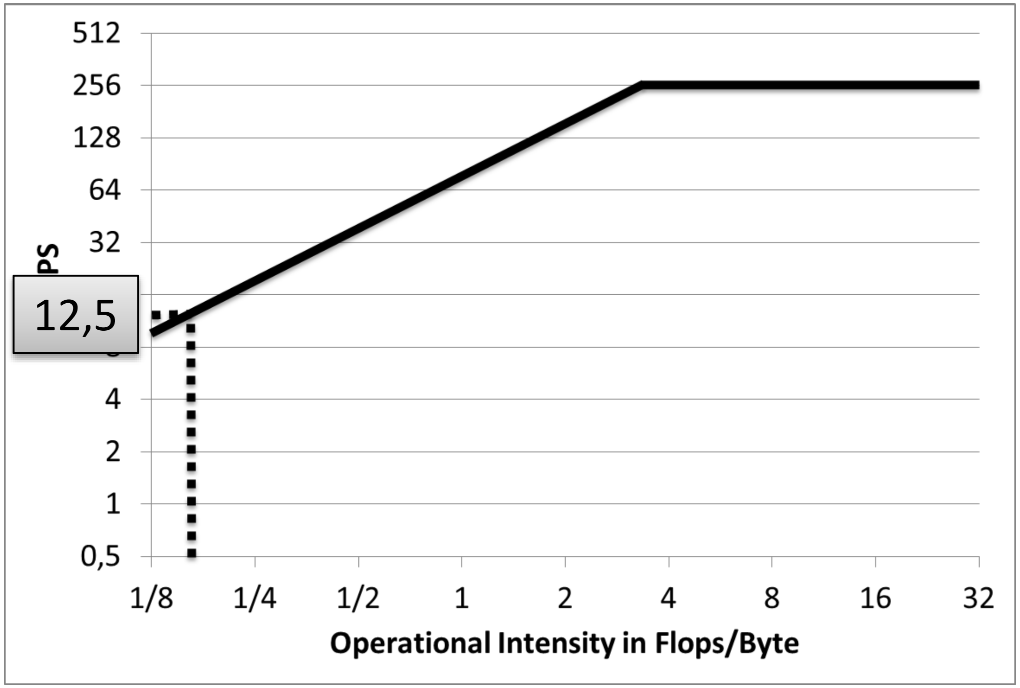
Roofline Model

Example: Sparse Matrix Vector Multiplication y=Ax



Given:

- **x and y are in the cache**
- **A is too large for the cache**
- **measured performance was 12 GFLOPS**



- 1 ADD and 1 MULT per element
- load of value (double) and index (int) per element

-> 2 Flops / 12 Byte = 1/6 Flops/Byte

Task-Analysis with Performance Tools

Intel VTune Amplifier XE

Top Hotspots

This section lists the most active functions in overall application performance.

Function	CPU Time
solve_parallelomptask@106	9.090s
CSudokuBoard::checkHorizontal	5.730s
CSudokuBoard::check	1.910s
CSudokuBoard::checkBox	1.620s
CSudokuBoard::CSudokuBoard	1.370s

The Task-region is our main Hotspot.

Call Stack	CPU Time: Total by Utilization (Idle, Poor, Ok, Ideal, Over)
[OpenMP worker]	14.620s
solve_parallelomptask@106	14.390s
solve_parallelomptask@106	11.710s
solve_parallelomptask@10	11.710s
solve_parallelomptask@1	9.020s
solve_parallelomptask@	9.020s
solve_parallelomptask(	8.220s
solve_parallelomptas	8.210s
solve_parallel	5.570s
solve_parallel	5.470s
solve_parallel$omp:	5.470s
solve_parallel$om|	5.470s
Highlighted 680 row(s):	

There is a long recursive call stack and the amount of work per level declines.

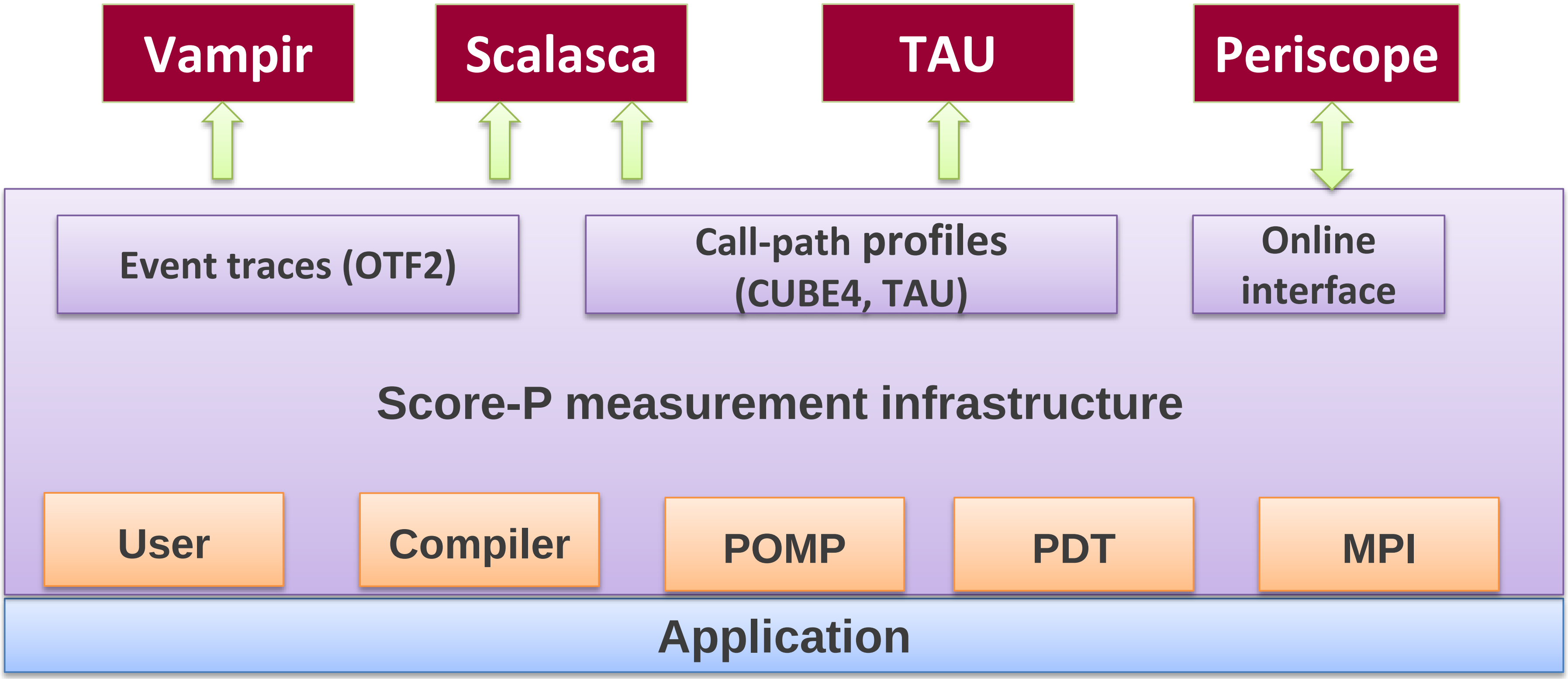
Overhead and Spin time is found for the task region …

Function / Call Stack	CPU Time by Utilization (Idle, Poor, Ok, Ideal)	Overhead and Spin Time
solve_parallelomptask@106	9.090s	1.220s

… and even for individual source code lines.

So.. Line	Source	CPU Time: Total by… (Idle, Poor, Ok, Ic)	Overhead and Spin Time: Total
128	#pragma omp taskwait	16.450s	1.220s
129			
130	sudoku->set(y, x, 0);		
131	return false;	0.050s	0s

Performance Measurements with Score-P



Score-P Usage

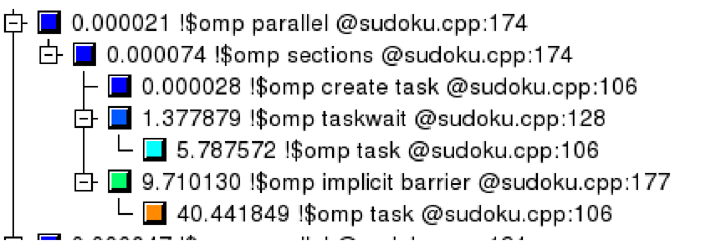
1. **Compiling and Linking:**
   - → gcc -> scorep gcc
   - → g++ -> scorep g++
   - → gfortran -> scorep gfortran
2. **Run your application:**
   - → SCOREP_ENABLE_PROFILING=true/false to enable/disable profiling
   - → SCOREP_ENABLE_TRACING=true/false to enable/disable tracing

Profiling	Tracing
Accumulated events are recorded. e.g. total time spend in foo()	Every event is recorded with a timestamp. e.g. start and end time for every call of foo()
less accurate	all information stored
reduced storage requirements (44 KB for Sudoku)	might need a lot of storage (1.2 GB for Sudoku)
Visualized e.g. in Cube Browser	Visualized e.g. in Vampir

Score-P (Profiling)/ Cube

Profiled Code with:

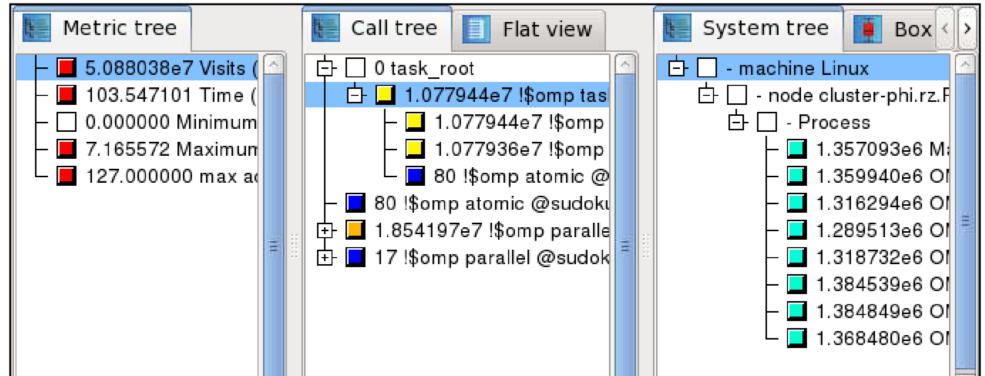
- OpenMP constructs enabled
- Function instrumentation disabled



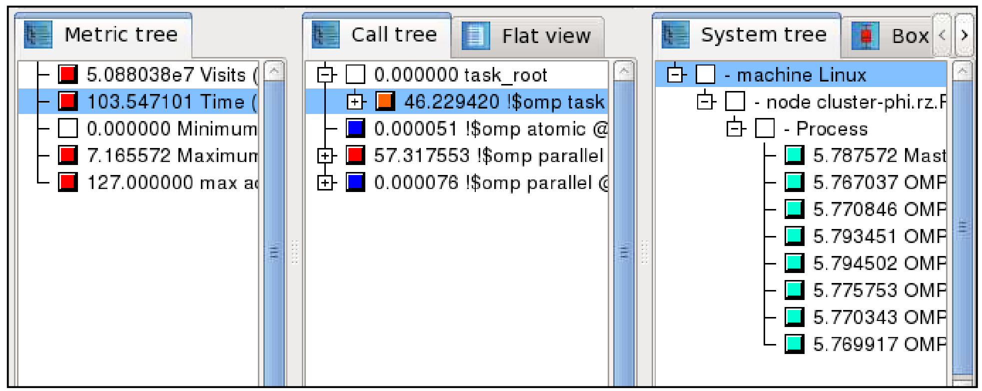
The Task-region is identified as hotspot.

No mapping to source code possible.

Profiling also gives more details about individual task-instances:



Every thread is executing ~1.3m tasks...

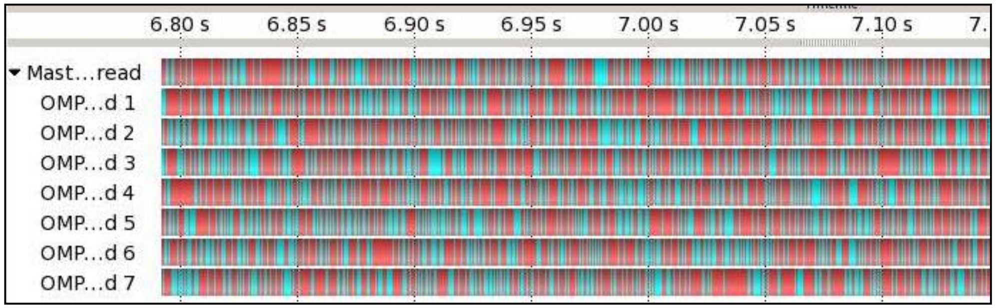


... in ~5.7 seconds.

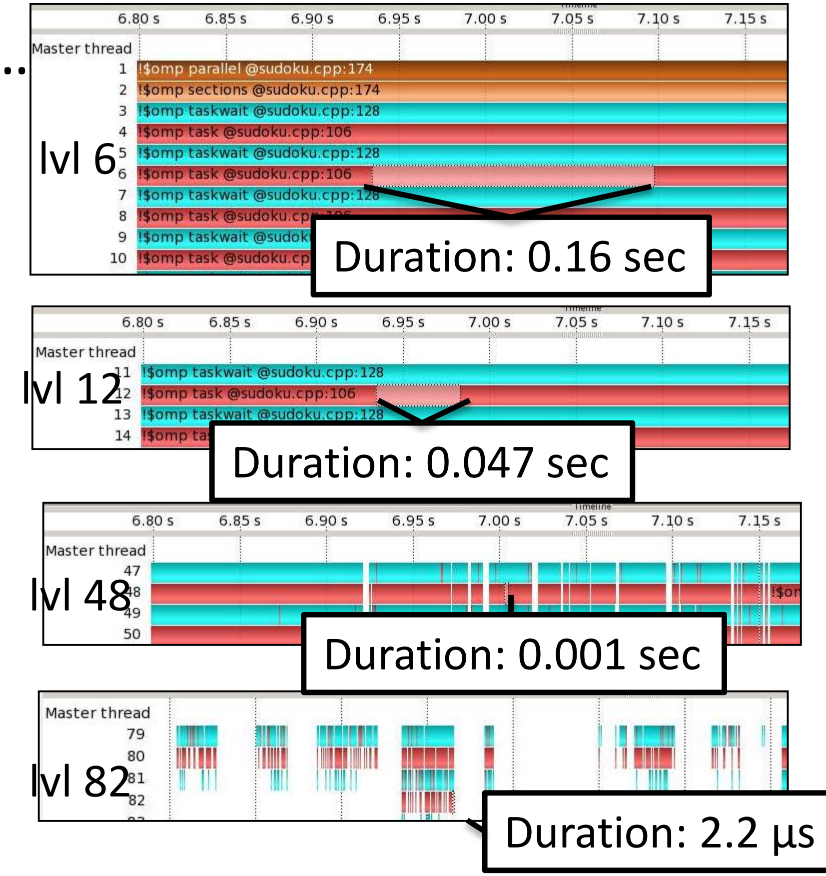
=> average duration of a task is ~4.4 µs

Score-P (Tracing)/ Vampir

Tracing gives a detailed timeline view…



… but also more detailed information on the call stack and the task durations can be shown.



Tasks get much smaller down the call stack.

Questions?