MANCHESTER
1824
The University of Manchester

EMiT
Emerging Technology
Conference
2014

intel® Look Inside™

Two examples\case studies using Intel® Xeon™ Phi

Tachyon Ray Tracing

Cloverleaf Hydrodynamics Mini-app

Stephen Blair-Chappell, Intel

Tachyon ray tracer

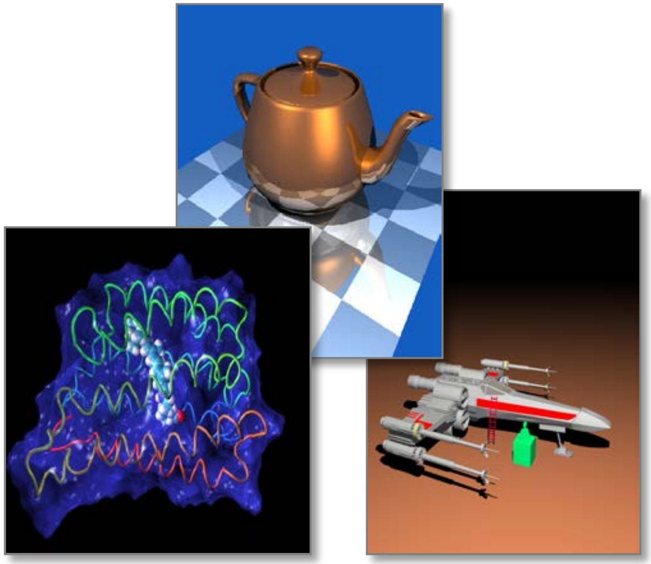
Port to Intel® Xeon Phi™ with Intel® Cluster Studio XE

Project goals

- Port to Intel® Xeon Phi™ and reach tangible performance gains vs initial Xeon-only baseline
- Test-drive Intel® Cluster Studio XE on Xeon Phi
- Create a case-study, with practical recommendations reusable in other cases

Not a goal: to create the best performing ray tracer. Refer to dedicated projects (e.g. Embree by Intel Labs)

Tachyon ray tracer



Open source ray tracing demo
(http://jedi.ks.uiuc.edu/~johns/raytracer/)

Part of SpecMPI suite

Supports parallelism (MPI + OpenMP)

Computational modes

Real-time rendering

Throughput computing

Production of *Puss in Boots*
required **69 million render hours**

Images (c) Audi, Dreamworks

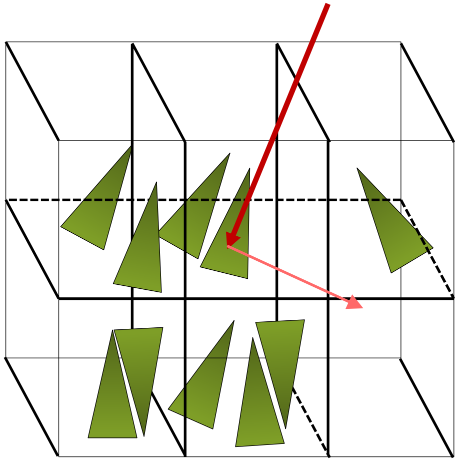
Tachyon algorithm

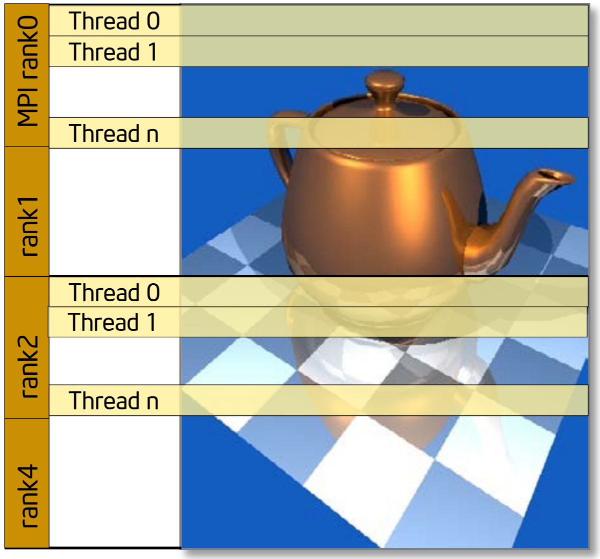
3D model is a set of primitives (e.g. triangles)

3D space is pre-divided into grid, each voxel points to list of triangles contained/crossing it

Image pixel calculated using ray intersections (lights, reflections, shadows)

Hybrid parallelism: each frame is divided into chunks processed by MPI processes, a chunk is divided into lines processed by OpenMP threads





Known issues of the algorithm

☹ **Communication profile:**

- 1 master and n workers. Workers communicate to the master only.
- Master performs same computations + processing. A bottleneck and limited scalability.
- Each frame starts after a previous one. All workers have to wait for order from the master.

☹ **Work imbalance:** lines and frames have different complexities

- ☺ Hybrid parallelism with dynamic OpenMP scheduling helps to relieve
- Static MPI scheduling still exhibits the issue across frames

Limited scalability across Xeon cluster. MPI+OpenMP hybrid better than MPI only

Extra challenge - imbalance across Xeon and Xeon Phi

Xeon and Xeon Phi have different performances

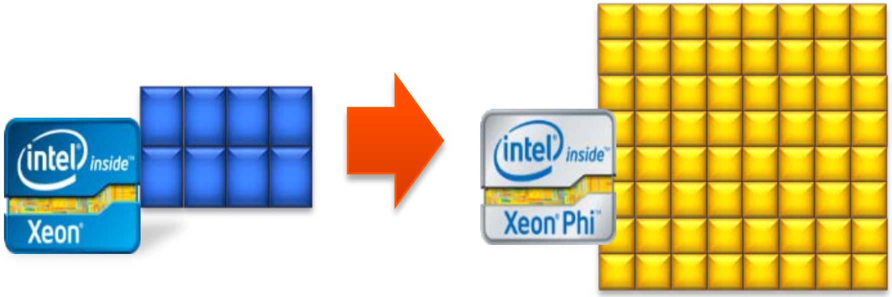
How to split up the work ?

Which execution model to choose ?

Is ray tracing good for Xeon Phi ?

Porting: Efficient apps for Xeon Phi

	Tachyon’s profile:
1. Allow massive parallelism (to load 60+cores x 4 threads)	☹ no slack: available parallel work (frame height) ~ # of threads
2. Run intensive computations (to efficiently use 512bit vectors)	☹ no vectorizable loops, only scalar computations
3. Provide memory efficiency (to meet current 4-8GB constraints)	☺

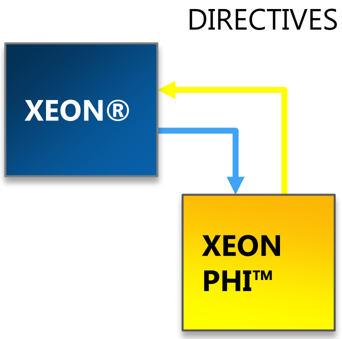


Your application needs to meet certain requirements to use Xeon Phi best

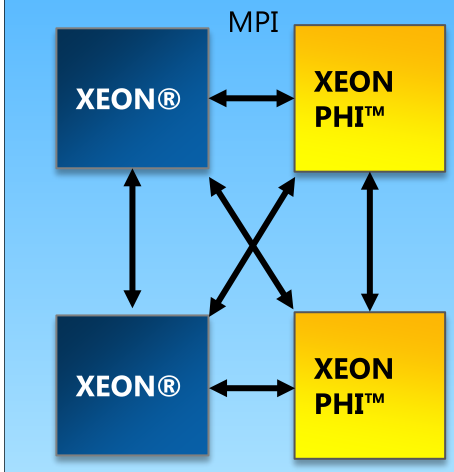
Target execution model – Symmetric MPI

XEON®
XEON® PHI

NATIVE model



OFFLOAD model



SYMMETRIC model

Most flexible. Least number of code changes.

Build for Xeon Phi

No code changes, only makefile:

-mmic	Target platform is Xeon Phi
-fp-model fast=2	Trade-off between accuracy and performance, OK for ray tracing

Very easy! Running code in a minute

Why ‘–fp-model fast=2’ ?

With default flag, a reciprocal (1/x) computation unexpectedly became a hotspot on Phi (not on Xeon):

- Compiler generated heavy-weight code for higher precision

-fp-model fast=2 is a trade-off to favor performance (precision is still fine for ray tracing)

- Reciprocal calculation time reduced by >2x

Run...

```
export I_MPI_MIC=enable

mpiexec.hydra \
    -n 2 -host mynode1 <command-line> : \
    -n 2 -host mynode2 <command-line> : \
…
    -n 2 –host mynoden <command-line> : \
    -n 2 –host mynode1-mic0 <command-line> : \
    -n 2 –host mynode1-mic1 <command-line> : \
    -n 2 –host mynode2-mic0 <command-line> : \
…
    -n 2 –host mynoden-mic1 <command-line>
```

Same syntax. A Phi card is just like another node.

First results

4 nodes x 2SNB – 102 FPS

4 nodes x 1KNC – **32 FPS** ???

4 nodes x (2SNB + 1KNC) - **38 FPS** !!!

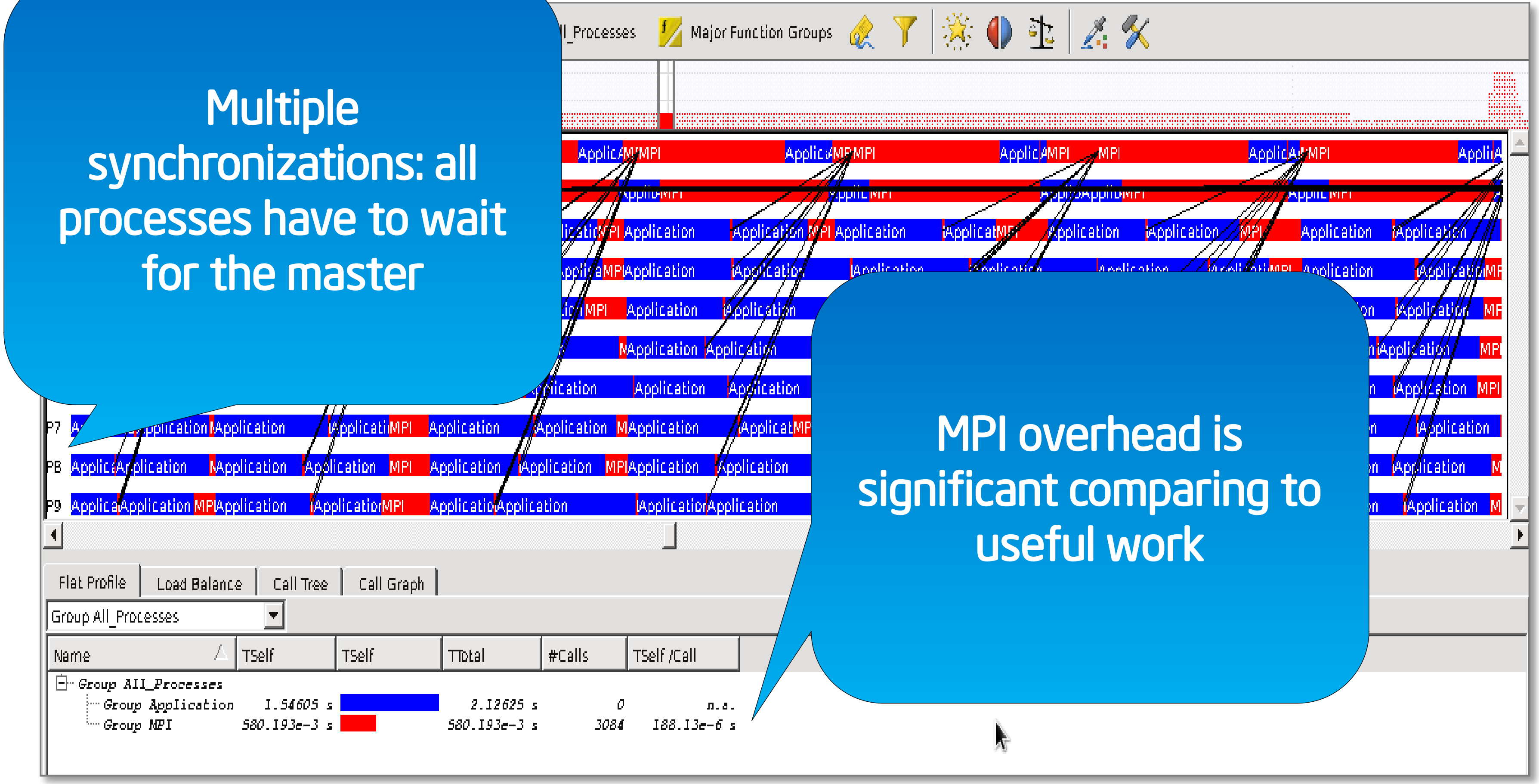
SNB – Sandy Bridge, 2nd generation Intel® Core™ processors

KNC – Knights Corner, Intel® Xeon Phi™ co-processors

Heterogeneous run slows down. Need to understand what happens

Using Intel® Trace Analyzer and Collector

Multiple synchronizations: all processes have to wait for the master



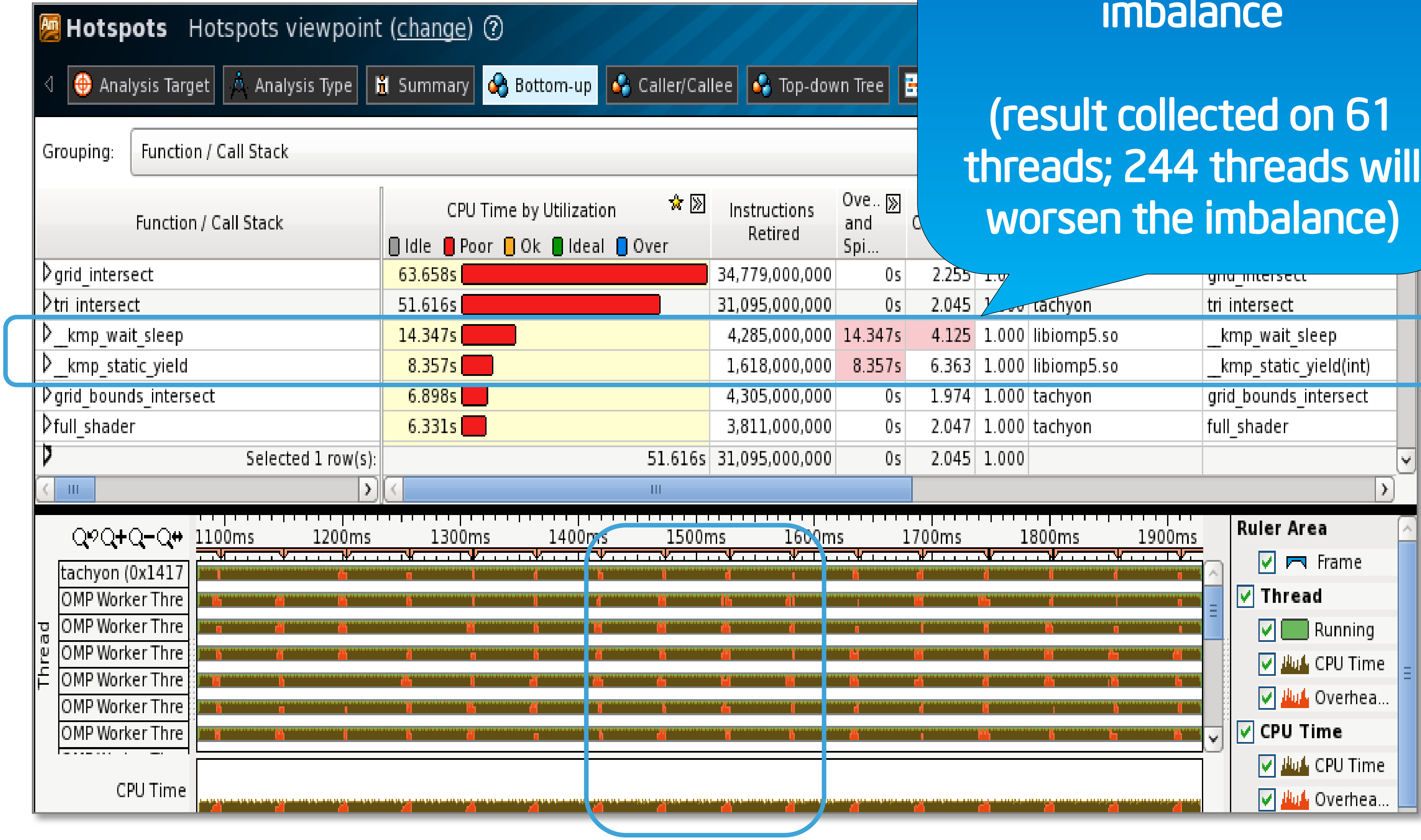
Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
Group All_Processes					
Group Application	1.54605 s	🟦	2.12625 s	0	n.a.
Group MPI	580.193e-3 s	🟥	580.193e-3 s	3084	188.13e-6 s

MPI overhead is significant comparing to useful work

Using VTune™ Amplifier XE

OpenMP overhead within each frame due to work imbalance

(result collected on 61 threads; 244 threads will worsen the imbalance)

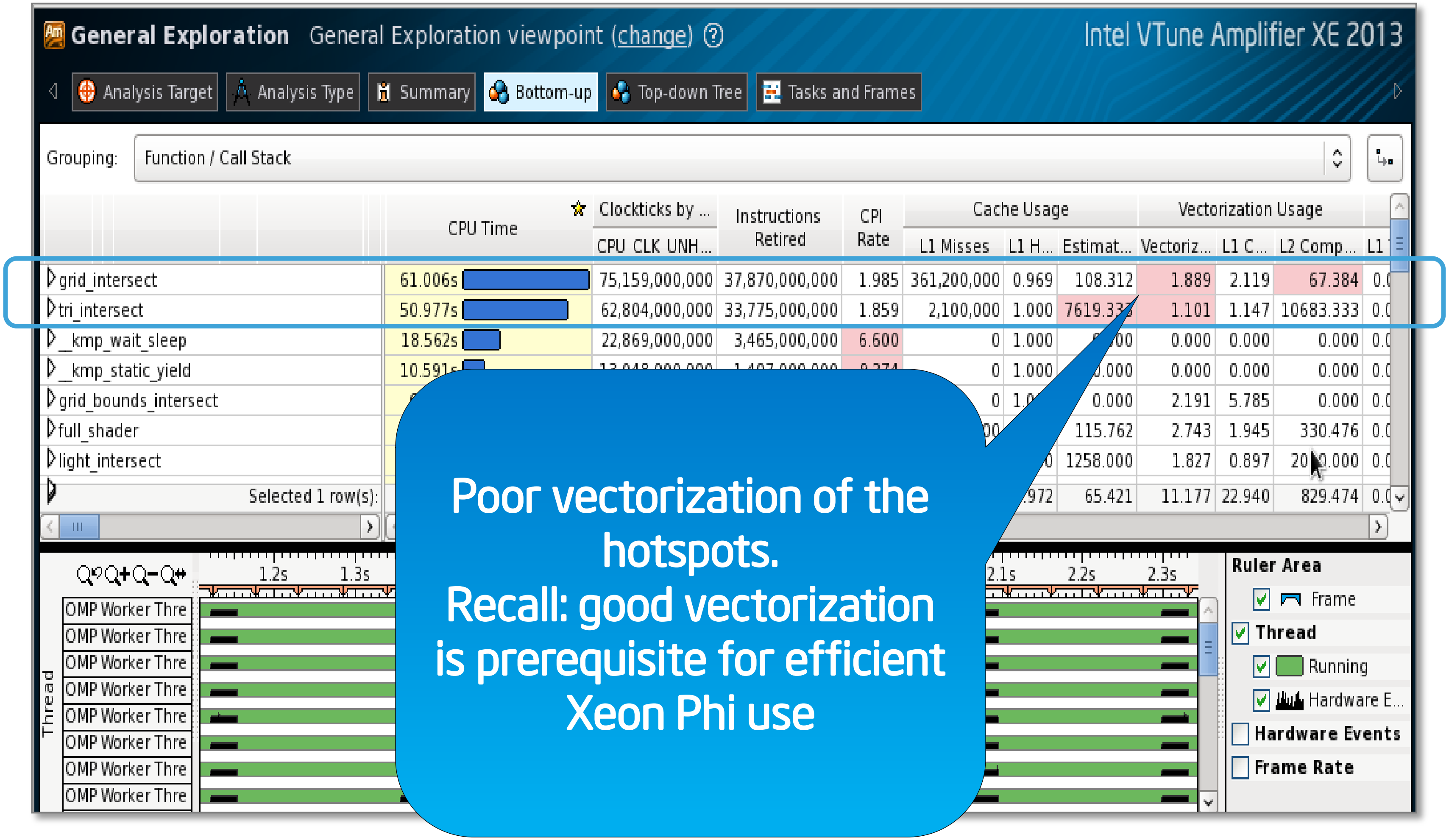


Hotspots Hotspots viewpoint (change) ?

Grouping: Function / Call Stack

Function / Call Stack	CPU Time by Utilization Idle Poor Ok Ideal Over		Instructions Retired	Ove.. and Spi...	C			
grid_intersect	63.658s		34,779,000,000	0s	2.255	1.0		grid_intersect
tri_intersect	51.616s		31,095,000,000	0s	2.045	1	tachyon	tri_intersect
__kmp_wait_sleep	14.347s		4,285,000,000	14.347s	4.125	1.000	libiomp5.so	__kmp_wait_sleep
__kmp_static_yield	8.357s		1,618,000,000	8.357s	6.363	1.000	libiomp5.so	__kmp_static_yield(int)
grid_bounds_intersect	6.898s		4,305,000,000	0s	1.974	1.000	tachyon	grid_bounds_intersect
full_shader	6.331s		3,811,000,000	0s	2.047	1.000	tachyon	full_shader
Selected 1 row(s):		51.616s	31,095,000,000	0s	2.045	1.000		

Using VTune™ Amplifier XE



Grouping: Function / Call Stack

	CPU Time	Clockticks by ...	Instructions Retired	CPI Rate	Cache Usage			Vectorization Usage			
		CPU_CLK_UNH...			L1 Misses	L1 H...	Estimat...	Vectoriz...	L1 C...	L2 Comp...	L1
grid_intersect	61.006s	75,159,000,000	37,870,000,000	1.985	361,200,000	0.969	108.312	1.889	2.119	67.384	0.0
tri_intersect	50.977s	62,804,000,000	33,775,000,000	1.859	2,100,000	1.000	7619.33[illegible]	1.101	1.147	10683.333	0.0
_kmp_wait_sleep	18.562s	22,869,000,000	3,465,000,000	6.600	0	1.000	[illegible]	0.000	0.000	0.000	0.0
_kmp_static_yield	10.591s	[illegible]	[illegible]	[illegible]	0	1.000	[illegible]	0.000	0.000	0.000	0.0
grid_bounds_intersect	[illegible]	[illegible]	[illegible]	[illegible]	0	[illegible]	0.000	2.191	5.785	0.000	0.0
full_shader	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	115.762	2.743	1.945	330.476	0.0
light_intersect	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	1258.000	1.827	0.897	20[illegible]0.000	0.0
Selected 1 row(s):	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible].972	65.421	11.177	22.940	829.474	0.0

Poor vectorization of the hotspots.
Recall: good vectorization is prerequisite for efficient Xeon Phi use

Conclusions

- No vectorization – 512 bit registers (able to hold 16 floats) are wasted
- Insufficient parallelism – 240 hyper-threads are wasted

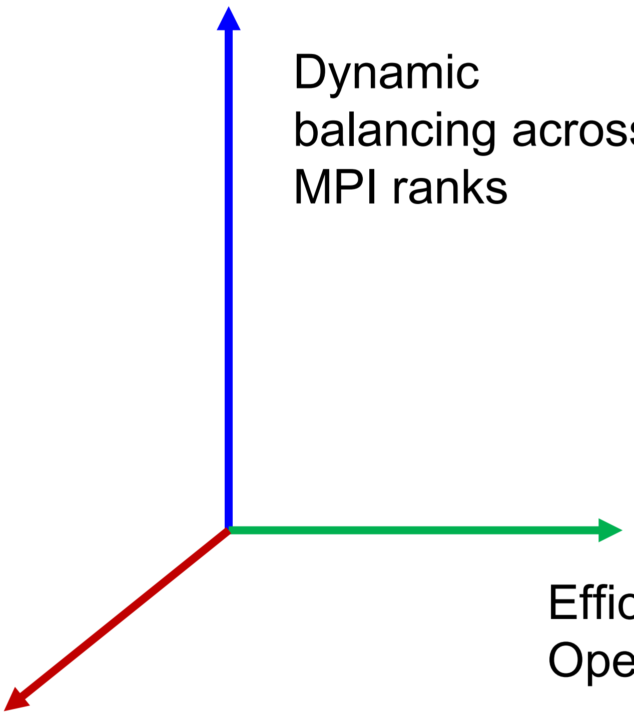
↓

- Ranks on Xeon Phi run slower than on Xeon

Due to static MPI scheduling within each frame and frame-by-frame computation, Xeon’s cannot start new frame until Xeon Phi’s complete their lines.

Total performance suffers

Improvement directions



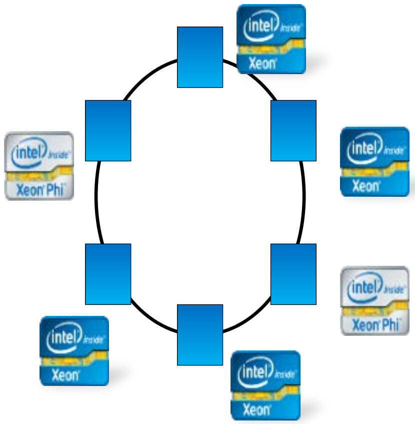
Dynamic balancing across MPI ranks

Efficient intra-process OpenMP parallelism

SIMD: exploit vectors

This works for **both** Xeon Phi and Xeon

#1 - Dynamic MPI scheduling



Each worker computes entire frame: asks a master for a frame #, computes and sends back entire frame

Master maintains a circular buffer, dispatches frame #, displays a frame. No computation by master

- Circular buffer to avoid memory growth

Significantly reduces # of communications

Reduced synchronizations: a worker does not wait for others anymore

Compensates Xeon vs Xeon Phi difference

Increases scalability

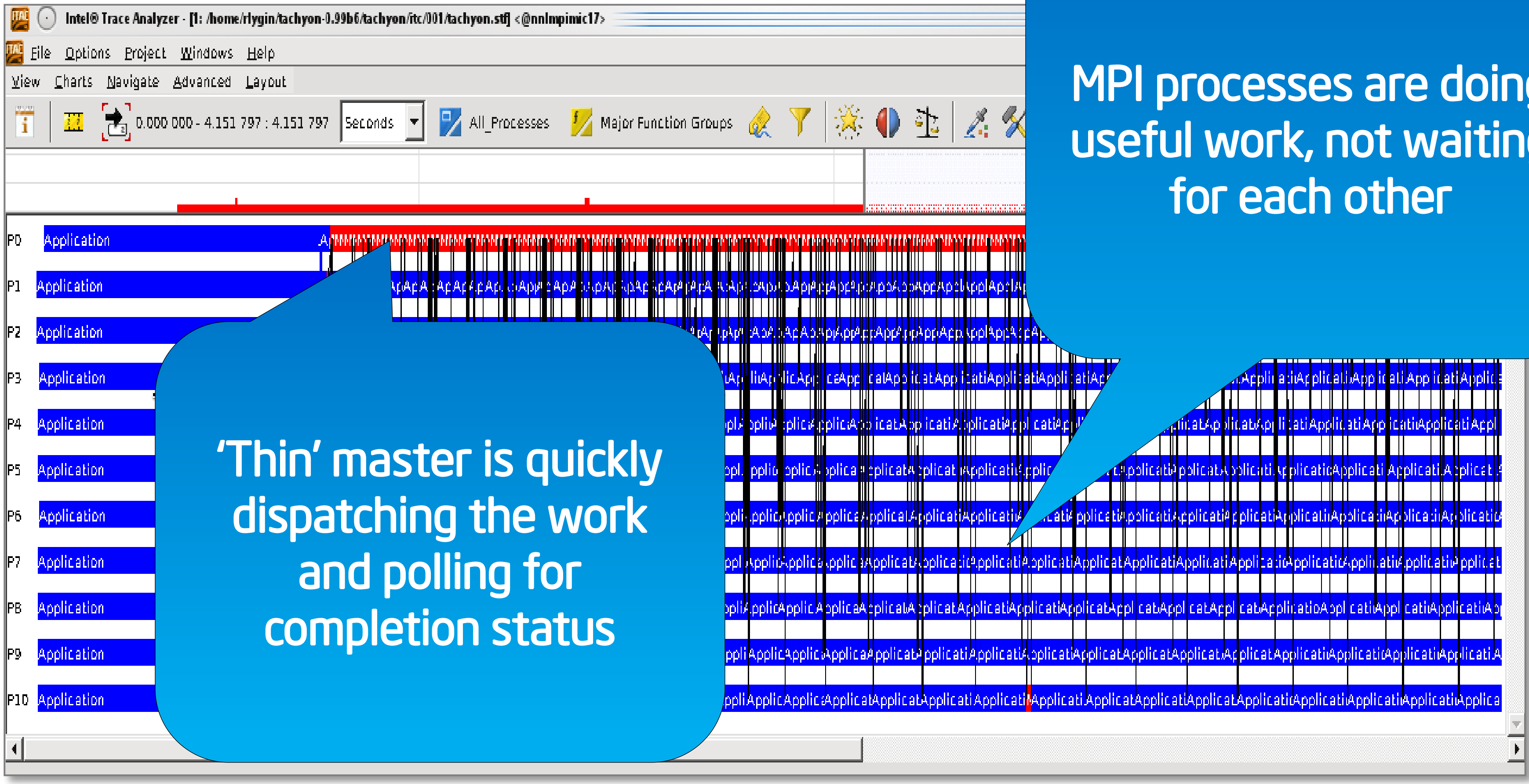
Improves both Xeon Phi and Xeon-only !

Code change

- Producer-consumer like algorithm
- New algorithm – ~250 lines in main loop
- Not Xeon Phi specific: could be implemented to address limited Xeon scalability. Xeon Phi just triggered it.
  - This is important: you optimize for Xeon, benefit everywhere!

Non-trivial but not a rocket science. Double ROI

Re-running Intel Trace Analyzer and Collector



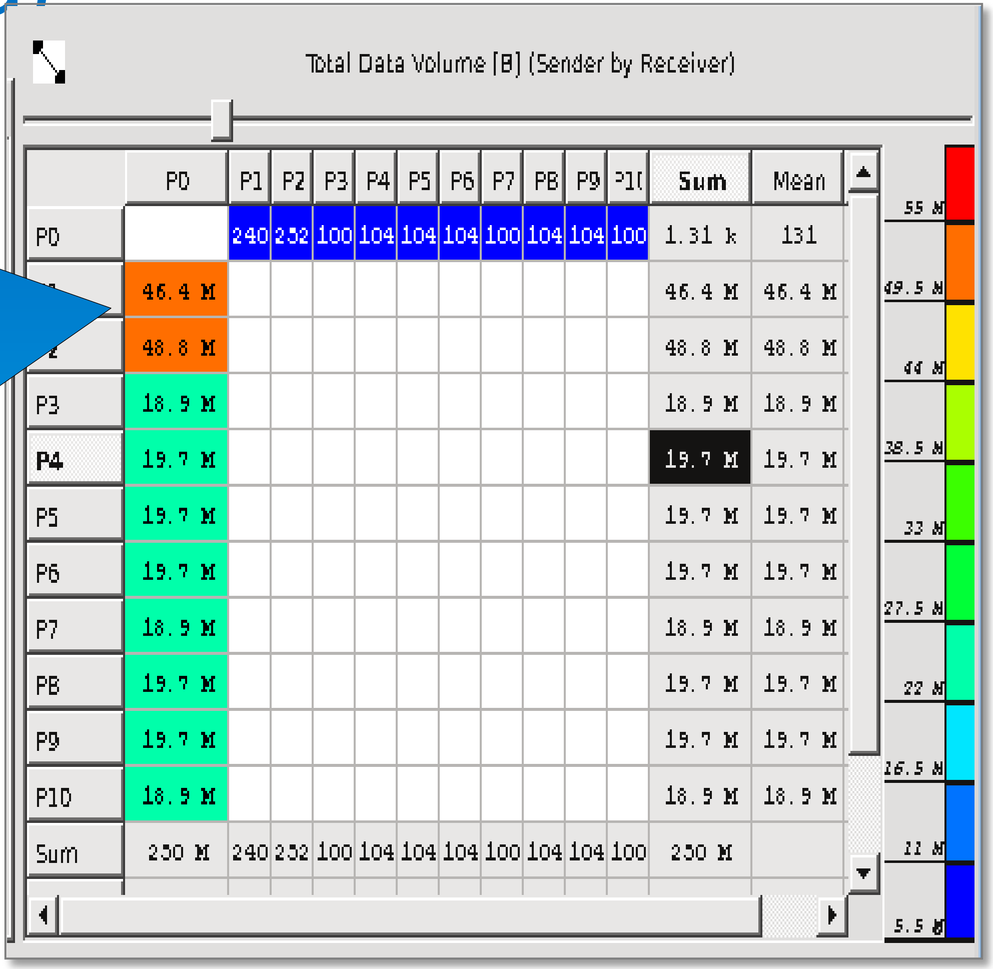
MPI processes are doing useful work, not waiting for each other

'Thin' master is quickly dispatching the work and polling for completion status

Re-running Intel Trace Analyzer and Collector (cont’ed)

Each Xeon process (P1 and P2) processes 2x data of each Xeon Phi process (P3-P10).

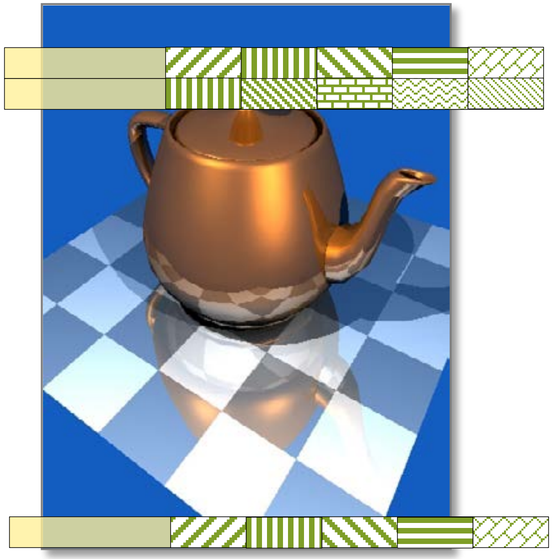
Processes are no longer gated by each other



Total Data Volume [B] (Sender by Receiver)

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P1(	Sum	Mean
P0		240	252	100	104	104	104	100	104	104	100	1.31 k	131
[illegible]	46.4 M											46.4 M	46.4 M
2	48.8 M											48.8 M	48.8 M
P3	18.9 M											18.9 M	18.9 M
P4	19.7 M											19.7 M	19.7 M
P5	19.7 M											19.7 M	19.7 M
P6	19.7 M											19.7 M	19.7 M
P7	18.9 M											18.9 M	18.9 M
P8	19.7 M											19.7 M	19.7 M
P9	19.7 M											19.7 M	19.7 M
P10	18.9 M											18.9 M	18.9 M
Sum	250 M	240	252	100	104	104	104	100	104	104	100	250 M	

#2. Improve OpenMP parallelism



Create parallel slack by reducing chunk size: from a line to a few pixels.

- >= cache line (to avoid false sharing)

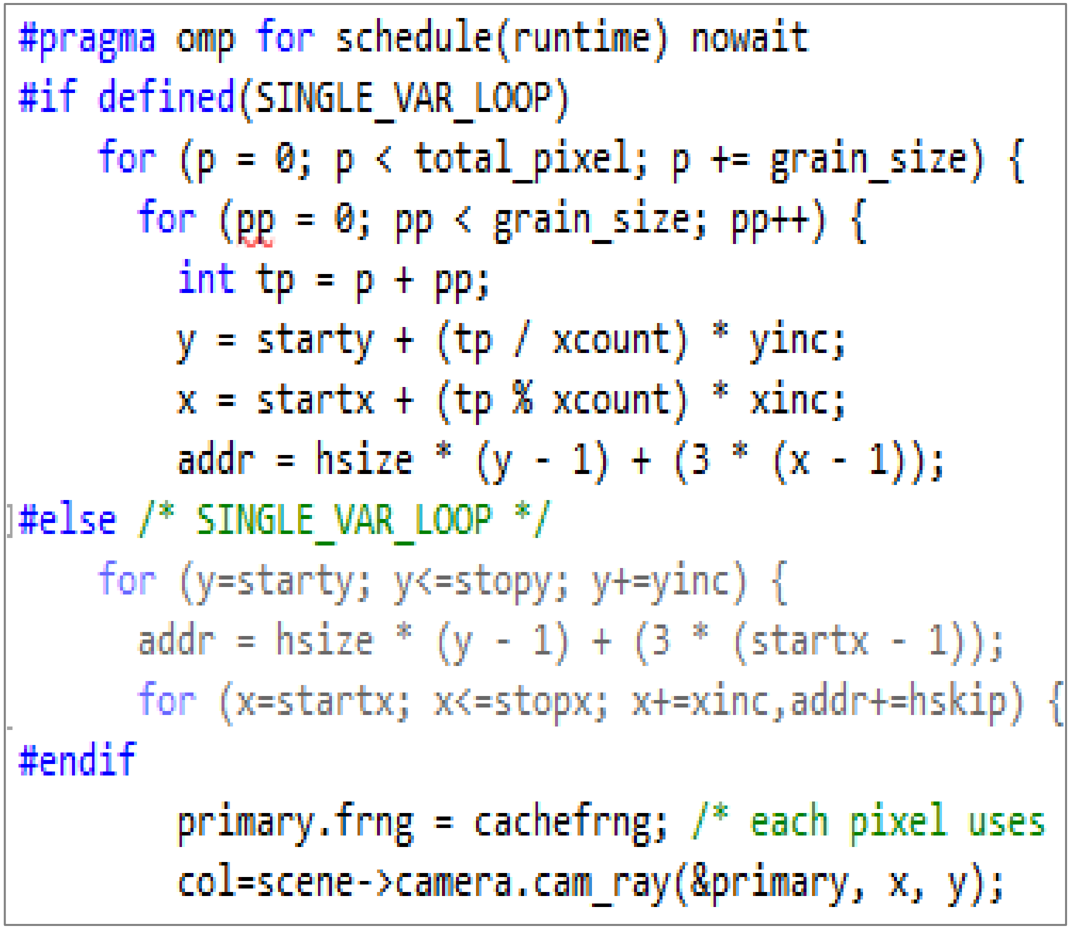
Keep dynamic scheduling (OMP_SCHEDULE=dynamic)

Enables massive parallelism (# of chunks >> HW threads)

Compensates different line complexities

- Also helps on Xeon

Code change

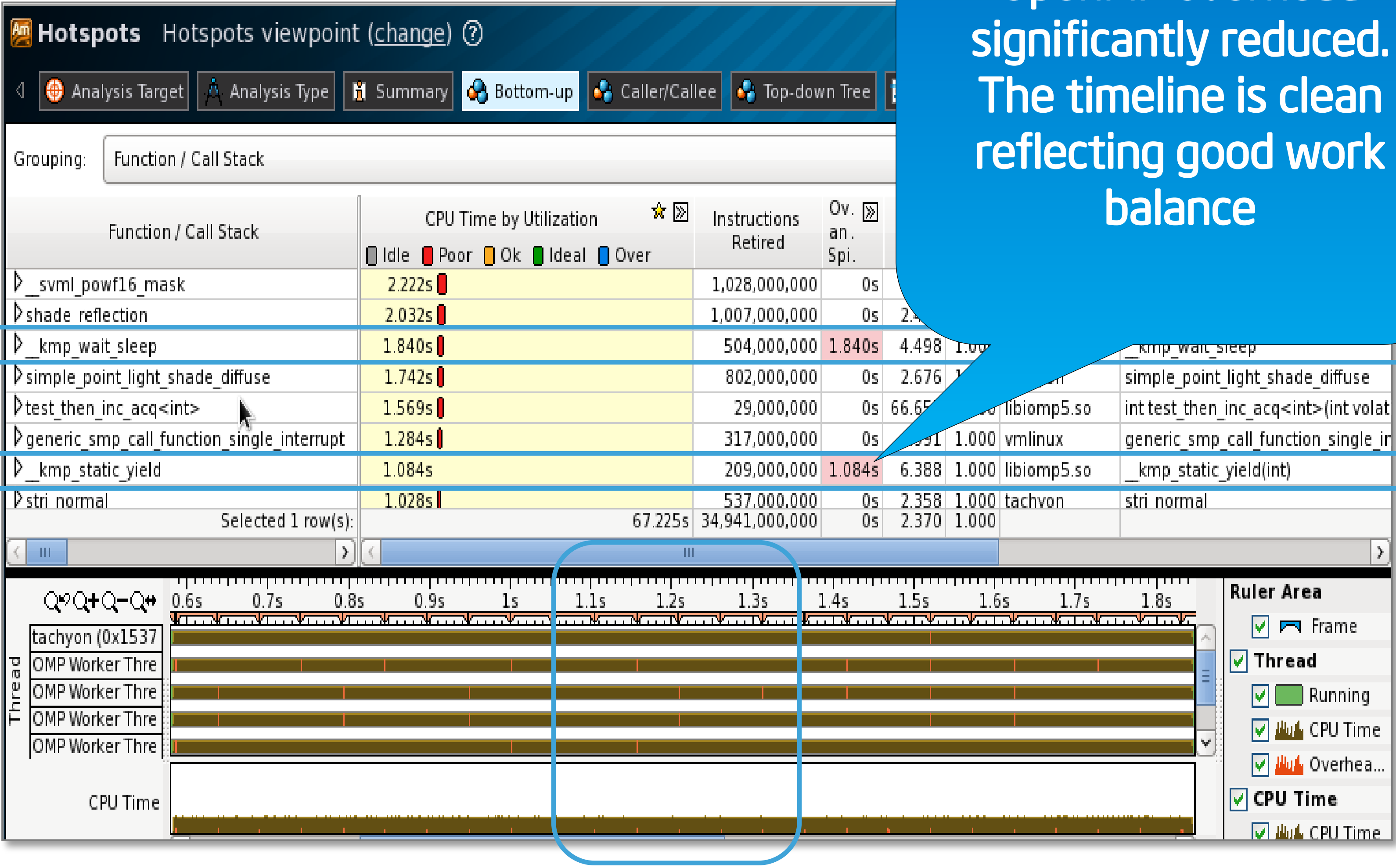


```
#pragma omp for schedule(runtime) nowait
#if defined(SINGLE_VAR_LOOP)
    for (p = 0; p < total_pixel; p += grain_size) {
      for (pp = 0; pp < grain_size; pp++) {
        int tp = p + pp;
        y = starty + (tp / xcount) * yinc;
        x = startx + (tp % xcount) * xinc;
        addr = hsize * (y - 1) + (3 * (x - 1));
#else /* SINGLE_VAR_LOOP */
    for (y=starty; y<=stopy; y+=yinc) {
      addr = hsize * (y - 1) + (3 * (startx - 1));
      for (x=startx; x<=stopx; x+=xinc,addr+=hskip) {
#endif
        primary.frng = cachefrng; /* each pixel uses
        col=scene->camera.cam_ray(&primary, x, y);
```

6 new lines – an OpenMP for-loop by pixel #, instead of by line #

Straightforward change. The same parallel model – OpenMP. Again, double ROI

Re-running with Amplifier



Hotspots Hotspots viewpoint (change) ?

Analysis Target | Analysis Type | Summary | Bottom-up | Caller/Callee | Top-down Tree

Grouping: Function / Call Stack

Function / Call Stack	CPU Time by Utilization ▮ Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over	Instructions Retired	Ov. an. Spi.				
▷ __svml_powf16_mask	2.222s	1,028,000,000	0s				
▷ shade_reflection	2.032s	1,007,000,000	0s	2.4			
▷ __kmp_wait_sleep	1.840s	504,000,000	1.840s	4.498	1.00		__kmp_wait_sleep
▷ simple_point_light_shade_diffuse	1.742s	802,000,000	0s	2.676	1		simple_point_light_shade_diffuse
▷ test_then_inc_acq<int>	1.569s	29,000,000	0s	66.65		libiomp5.so	int test_then_inc_acq<int>(int volati
▷ generic_smp_call_function_single_interrupt	1.284s	317,000,000	0s	991	1.000	vmlinux	generic_smp_call_function_single_in
▷ __kmp_static_yield	1.084s	209,000,000	1.084s	6.388	1.000	libiomp5.so	__kmp_static_yield(int)
▷ stri_normal	1.028s	537,000,000	0s	2.358	1.000	tachyon	stri_normal
Selected 1 row(s):	67.225s	34,941,000,000	0s	2.370	1.000		

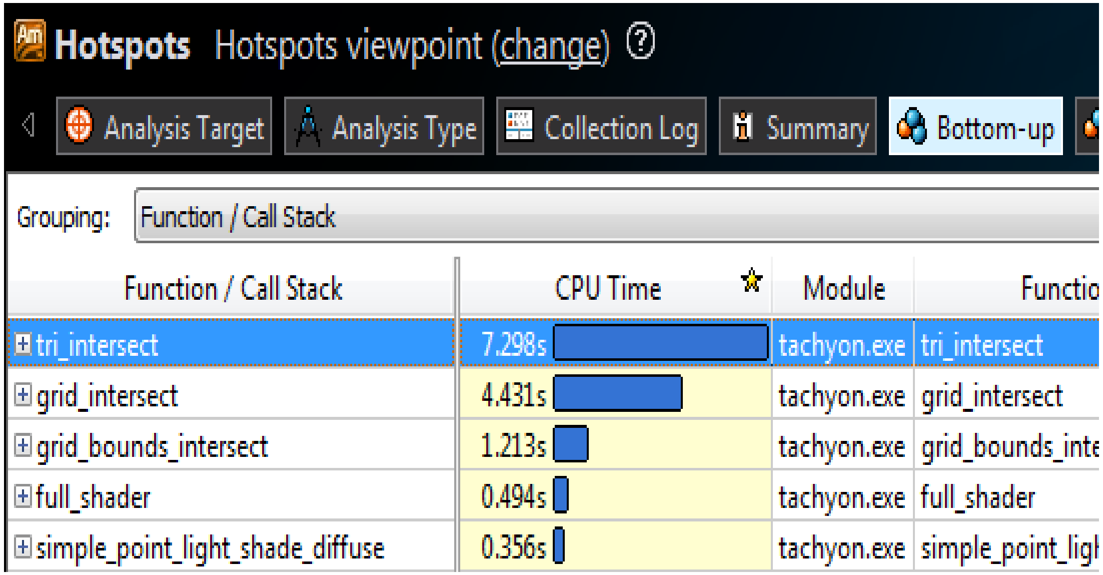
OpenMP overhead significantly reduced. The timeline is clean reflecting good work balance

Optimization Notice

#3. Exploiting SIMD (Single Instruction Multiple Data)

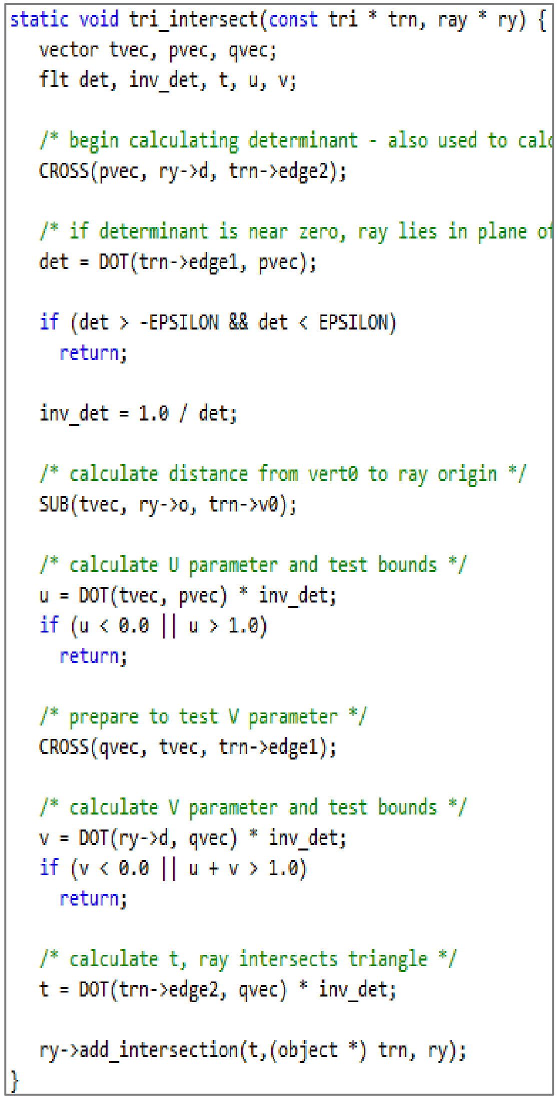
How to utilize vectorization when:

- there are no loops in a hotspot function (*tri_intersect*) ?
- the hotspot function is called on a linked list (*grid_intersect*) ?



Grouping: Function / Call Stack

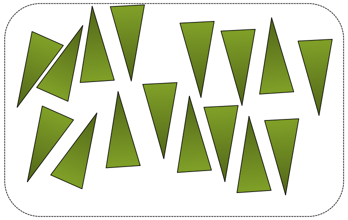
Function / Call Stack	CPU Time	Module	Functio
⊞tri_intersect	7.298s	tachyon.exe	tri_intersect
⊞grid_intersect	4.431s	tachyon.exe	grid_intersect
⊞grid_bounds_intersect	1.213s	tachyon.exe	grid_bounds_inte
⊞full_shader	0.494s	tachyon.exe	full_shader
⊞simple_point_light_shade_diffuse	0.356s	tachyon.exe	simple_point_ligh



```
static void tri_intersect(const tri * trn, ray * ry) {
   vector tvec, pvec, qvec;
   flt det, inv_det, t, u, v;

   /* begin calculating determinant - also used to calc
   CROSS(pvec, ry->d, trn->edge2);

   /* if determinant is near zero, ray lies in plane of
   det = DOT(trn->edge1, pvec);

   if (det > -EPSILON && det < EPSILON)
     return;

   inv_det = 1.0 / det;

   /* calculate distance from vert0 to ray origin */
   SUB(tvec, ry->o, trn->v0);

   /* calculate U parameter and test bounds */
   u = DOT(tvec, pvec) * inv_det;
   if (u < 0.0 || u > 1.0)
     return;

   /* prepare to test V parameter */
   CROSS(qvec, tvec, trn->edge1);

   /* calculate V parameter and test bounds */
   v = DOT(ry->d, qvec) * inv_det;
   if (v < 0.0 || u + v > 1.0)
     return;

   /* calculate t, ray intersects triangle */
   t = DOT(trn->edge2, qvec) * inv_det;

   ry->add_intersection(t,(object *) trn, ry);
}
```

Code change – new data structures

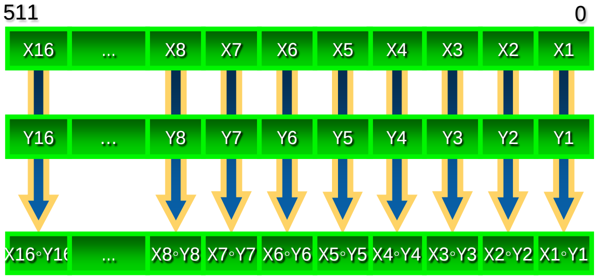
Composite triangles



V1: {x1, x2, ..., xn}
{y1, y2, ..., yn}
{z1, z2, ..., zn}

V2: ...

V3: ...



Composite triangles:

- SSE: 4 triangles, AVX: 8, Xeon Phi: 16
- Structure Of Arrays: register containing 4/8/16 float coordinates (x, y or z)
- Bit mask to describe ‘real’/‘void’ triangles

A small library of vector operations (+,-, dot-, cross-product,…) using intrinsics

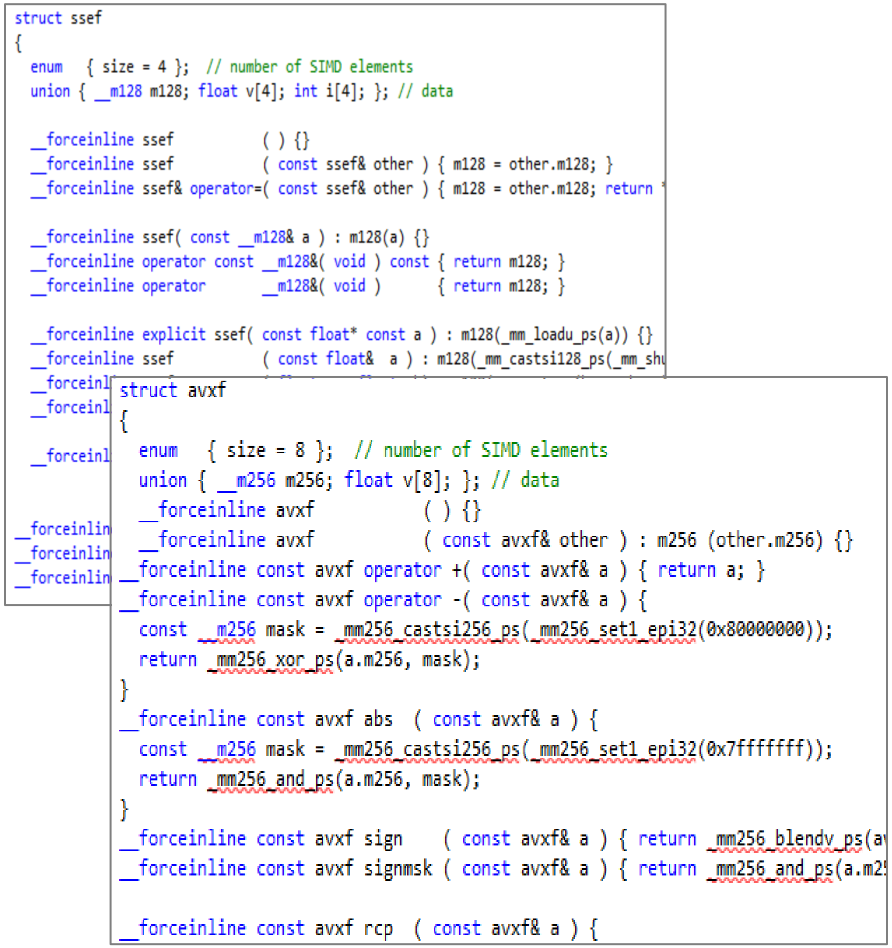
- Reused from Embree for SSE/AVX, extended for Phi

Single C++ template intersection (et al) function

- No code duplication

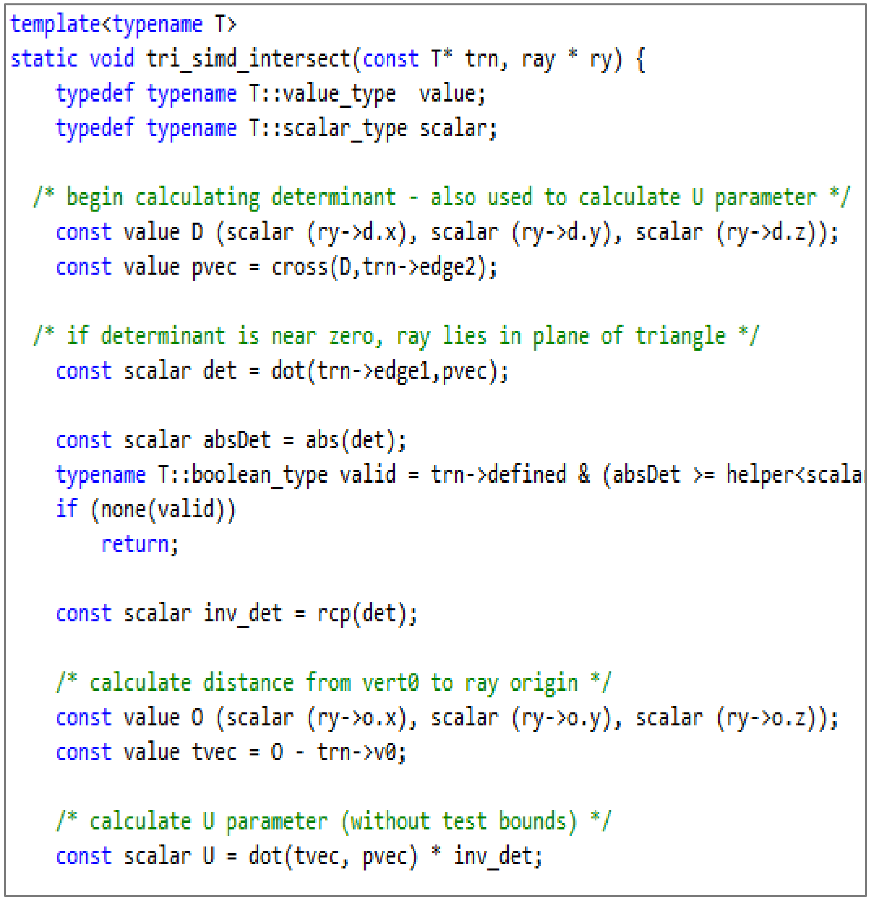
Again, double ROI - improves both Xeon Phi and Xeon !

Code change (cont'ed)



```
struct ssef
{
  enum   { size = 4 };  // number of SIMD elements
  union { __m128 m128; float v[4]; int i[4]; }; // data

  __forceinline ssef           ( ) {}
  __forceinline ssef           ( const ssef& other ) { m128 = other.m128; }
  __forceinline ssef& operator=( const ssef& other ) { m128 = other.m128; return

  __forceinline ssef( const __m128& a ) : m128(a) {}
  __forceinline operator const __m128&( void ) const { return m128; }
  __forceinline operator       __m128&( void )       { return m128; }

  __forceinline explicit ssef( const float* const a ) : m128(_mm_loadu_ps(a)) {}
  __forceinline ssef           ( const float&  a ) : m128(_mm_castsi128_ps(_mm_shu
  __forceinl
  __forceinl

  __forceinl


__forceinlin
__forceinlin
__forceinlin
```

```
struct avxf
{
  enum   { size = 8 };  // number of SIMD elements
  union { __m256 m256; float v[8]; }; // data
  __forceinline avxf           ( ) {}
  __forceinline avxf           ( const avxf& other ) : m256 (other.m256) {}
__forceinline const avxf operator +( const avxf& a ) { return a; }
__forceinline const avxf operator -( const avxf& a ) {
  const __m256 mask = _mm256_castsi256_ps(_mm256_set1_epi32(0x80000000));
  return _mm256_xor_ps(a.m256, mask);
}
__forceinline const avxf abs  ( const avxf& a ) {
  const __m256 mask = _mm256_castsi256_ps(_mm256_set1_epi32(0x7fffffff));
  return _mm256_and_ps(a.m256, mask);
}
__forceinline const avxf sign    ( const avxf& a ) { return _mm256_blendv_ps(a
__forceinline const avxf signmsk ( const avxf& a ) { return _mm256_and_ps(a.m2

__forceinline const avxf rcp  ( const avxf& a ) {
```



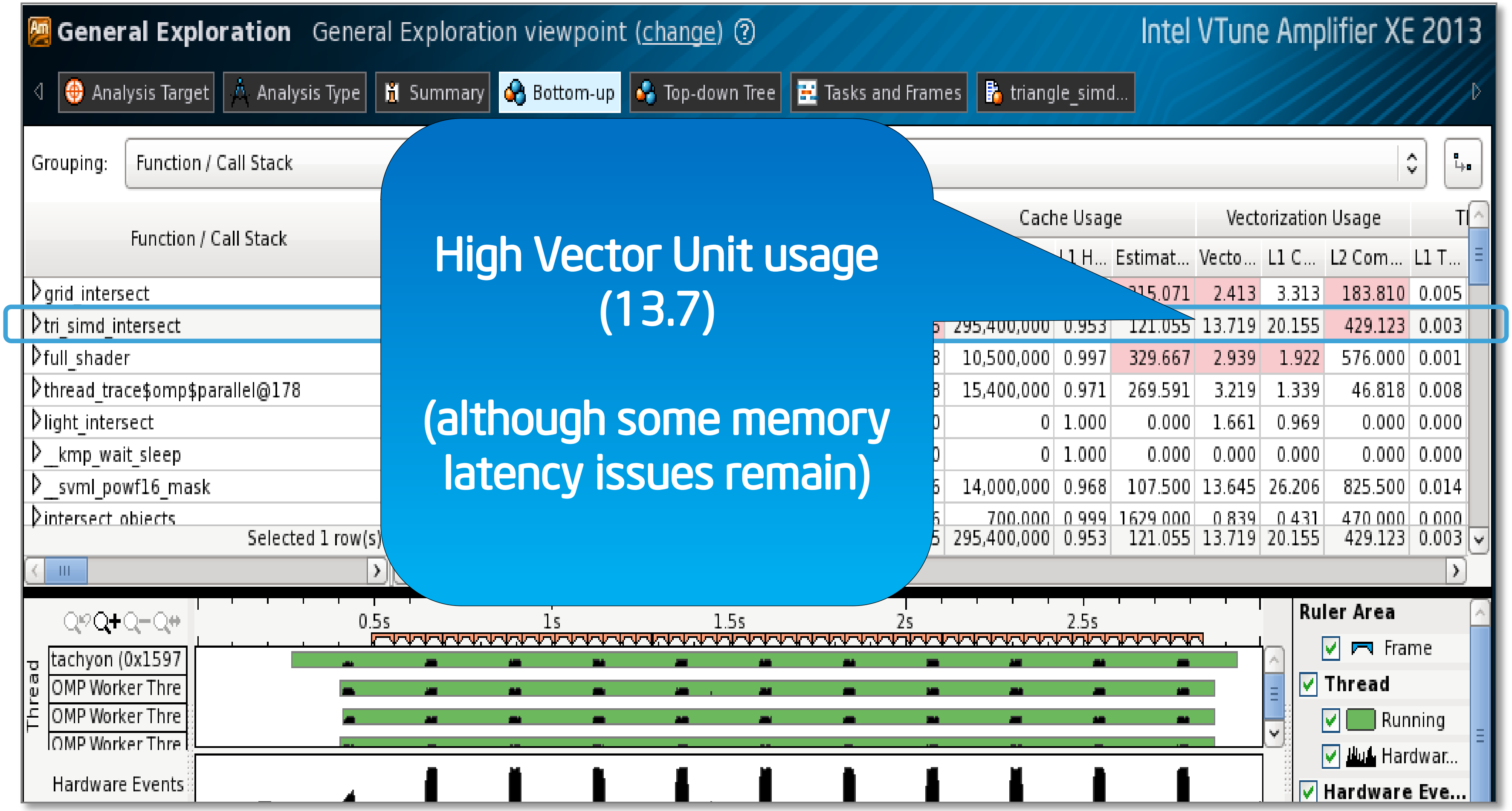
```
template<typename T>
static void tri_simd_intersect(const T* trn, ray * ry) {
    typedef typename T::value_type  value;
    typedef typename T::scalar_type scalar;

  /* begin calculating determinant - also used to calculate U parameter */
    const value D (scalar (ry->d.x), scalar (ry->d.y), scalar (ry->d.z));
    const value pvec = cross(D,trn->edge2);

  /* if determinant is near zero, ray lies in plane of triangle */
    const scalar det = dot(trn->edge1,pvec);

    const scalar absDet = abs(det);
    typename T::boolean_type valid = trn->defined & (absDet >= helper<scala
    if (none(valid))
        return;

    const scalar inv_det = rcp(det);

    /* calculate distance from vert0 to ray origin */
    const value O (scalar (ry->o.x), scalar (ry->o.y), scalar (ry->o.z));
    const value tvec = O - trn->v0;

    /* calculate U parameter (without test bounds) */
    const scalar U = dot(tvec, pvec) * inv_det;
```

SIMD benefits

- One intersection with multiple triangles at once
- Approach can be used for multi-rays intersections
  - used by Embree and Autodesk's ray tracer
- Small extra overhead during scene load (each grid cell rebuilds its list of simple triangles to composites) but benefit in heavy computations
- Intrinsics can be replaced with direct loops and compiler's auto-vectorization to improve portability

Again, double ROI - improves both Xeon Phi and Xeon !

Re-running with Amplifier XE



General Exploration General Exploration viewpoint (change) ? Intel VTune Amplifier XE 2013

Grouping: Function / Call Stack

			Cache Usage		Vectorization Usage			Tl
Function / Call Stack			L1 H...	Estimat...	Vecto...	L1 C...	L2 Com...	L1 T...
grid_intersect				[illegible]15.071	2.413	3.313	183.810	0.005
tri_simd_intersect	5	295,400,000	0.953	121.055	13.719	20.155	429.123	0.003
full_shader	8	10,500,000	0.997	329.667	2.939	1.922	576.000	0.001
thread_trace$omp$parallel@178	8	15,400,000	0.971	269.591	3.219	1.339	46.818	0.008
light_intersect	0	0	1.000	0.000	1.661	0.969	0.000	0.000
__kmp_wait_sleep	0	0	1.000	0.000	0.000	0.000	0.000	0.000
__svml_powf16_mask	6	14,000,000	0.968	107.500	13.645	26.206	825.500	0.014
intersect_objects	6	700,000	0.999	1629.000	0.839	0.431	470.000	0.000
Selected 1 row(s)	5	295,400,000	0.953	121.055	13.719	20.155	429.123	0.003

High Vector Unit usage (13.7)

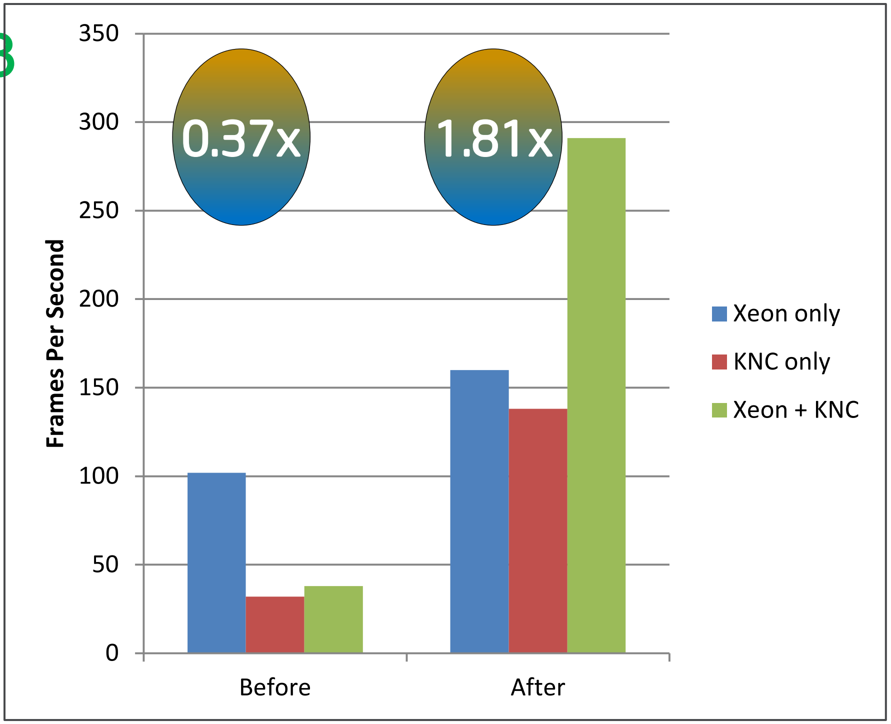
(although some memory latency issues remain)

Updated results

4 nodes x 2 SNB – 102→160 FPS

4 nodes x 1 KNC – 32→138 FPS

4 nodes x (2SNB + 1KNC) - 38→ 291 FPS



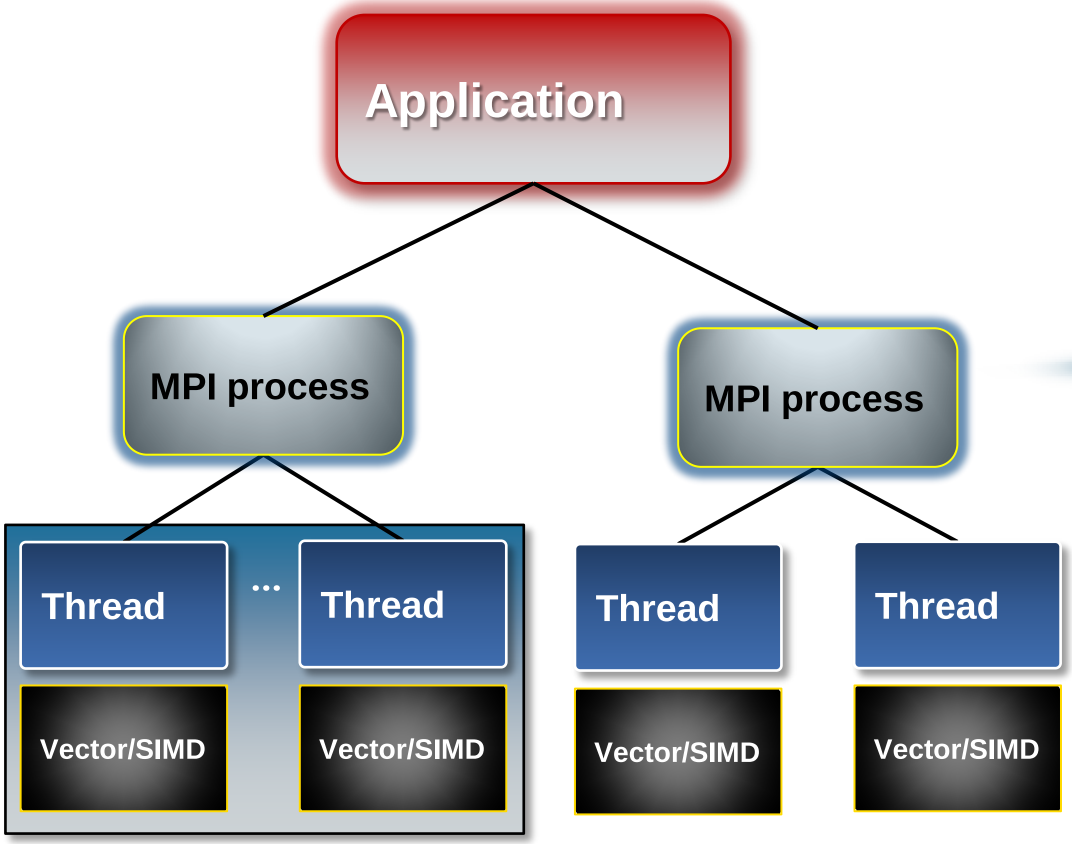
Speed up on both Xeon and Xeon Phi

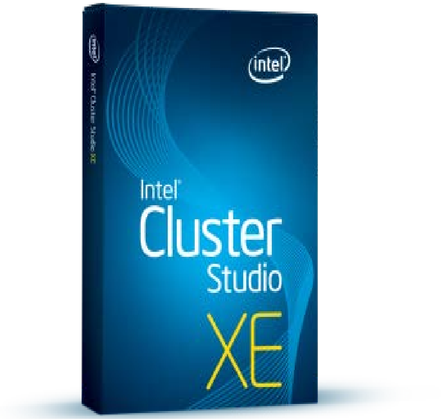
7X speed up !

Xeon and Xeon Phi add to each other

Parallel programming for Intel architecture

Intel® Xeon and Intel® Xeon Phi™





Intel Xeon E5:

- 8 cores
- 16 threads
- SIMD-256

Intel Xeon Phi:

- 60 cores
- 240 threads
- SIMD-512

Parallelism at all levels, with Intel software tools. Maximize your ROI!

Next steps

Experiment with prefetching

Replace intrinsics with plain C and rely on vectorization by compiler

Experiment with replacing linked lists with arrays

Fine-tune with affinity settings (e.g. KMP_AFFINITY=balanced)

Summary

The application must meet certain criteria to benefit from Xeon Phi

You might need to apply reasonable efforts to achieve that

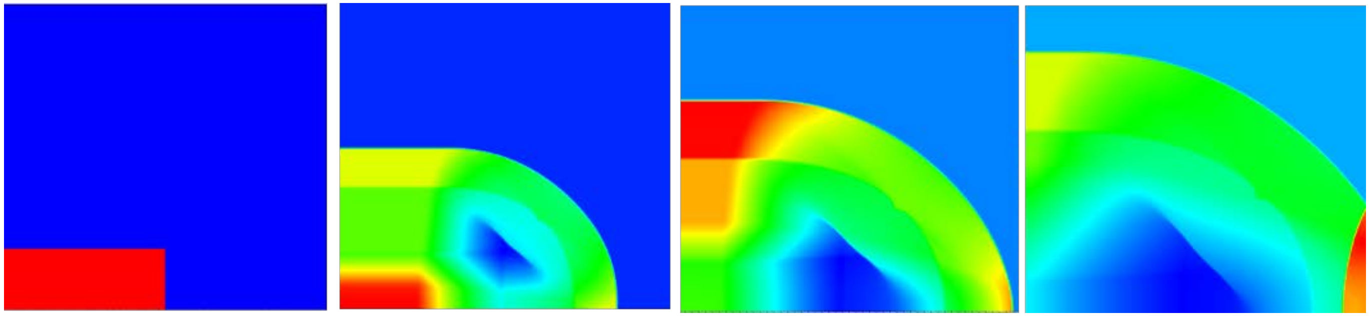
Good news:

- You can optimize for Xeon and benefit on Xeon Phi, and vice versa
- You use the same tools and programming models, same code

Cloverleaf demo

Port to Intel® Xeon Phi™ with Intel® Cluster Studio XE

What is Cloverleaf?



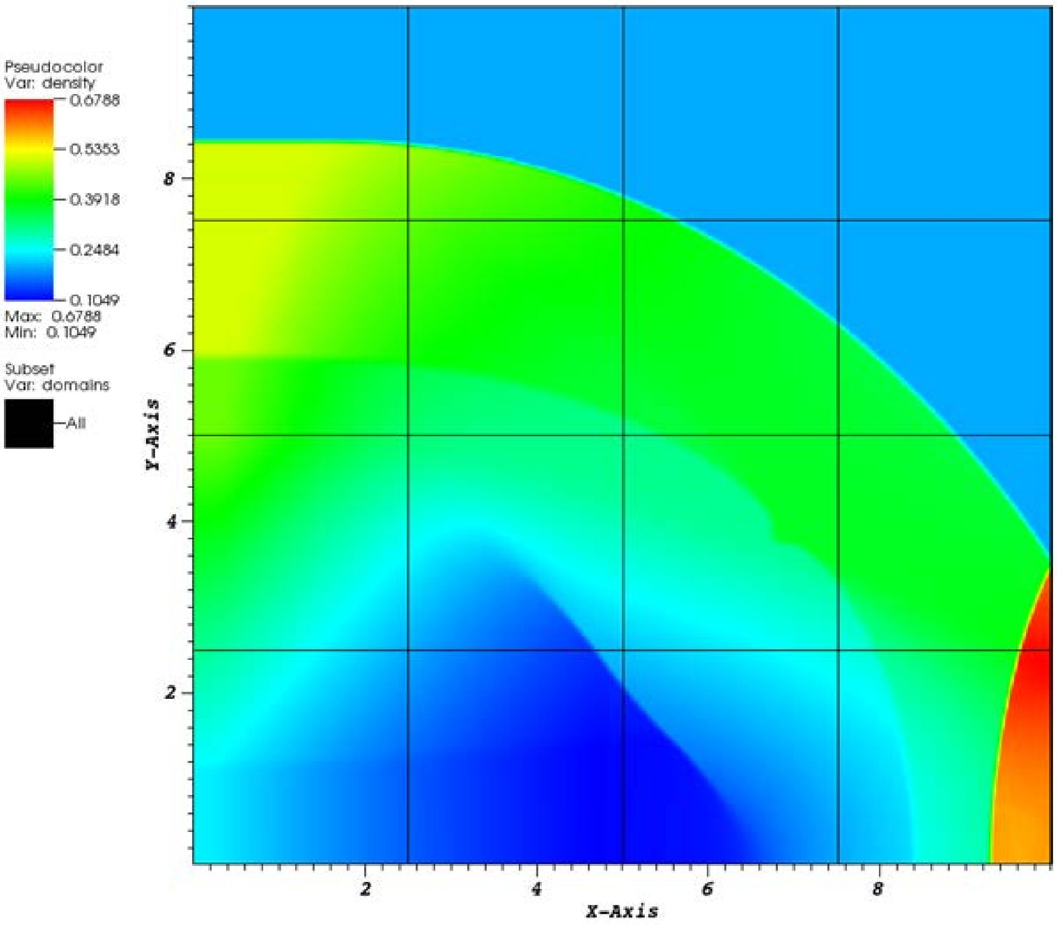
Snapshots of a Cloverleaf simulation

- Small open source benchmark
- Two dimensional compressible Euler equations across Cartesian grid
- Fortran framework
  - Fortran kernels
  - ANSI-C *kernels*

http://warwick-pcav.github.com/CloverLeaf/

- Programmer's playground
  - CUDA
  - OpenMP
  - MPI
  - OpenCL
  - OpenACC

A Partitioned Data Set



- On Xeons,
  - simulation space split into 16 regions
  - Used MPI to run on multiple cores
- On Xeon Phi
  - 60 MPI tasks
  - 4 OpenMP thread

What steps did you take to become Intel® Xeon Phi^{TM} ready?

- At first we had no access to any MIC hardware
- So, development was carried out on a regular eight-core workstation PC
- We knew that to make best use of the Intel® Xeon Phi^{TM} coprocessor, our code should be
  - well parallelized,
  - well vectorized.

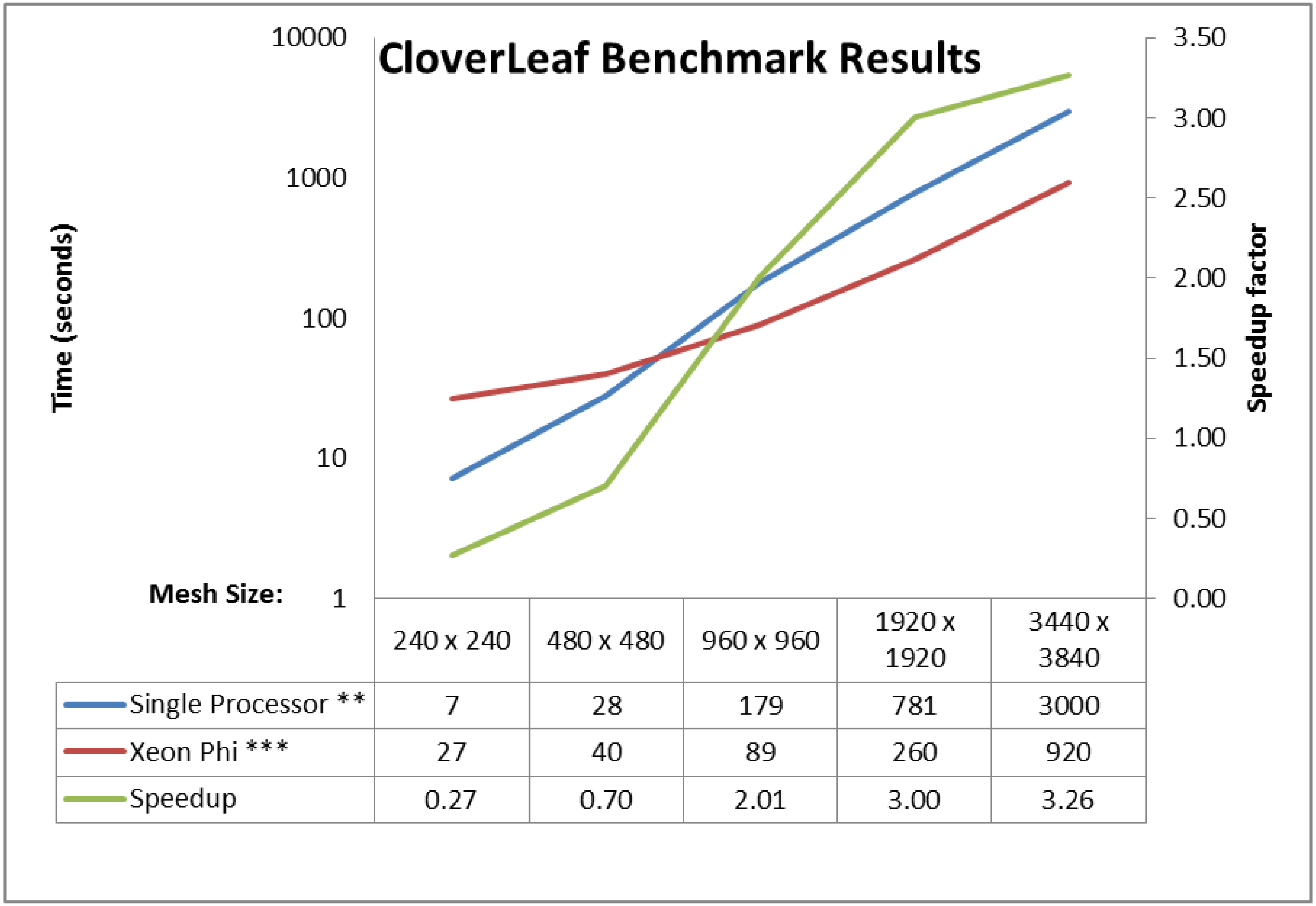
Anything went particularly well?

“We were surprised at how easy it was to get the first version of CloverLeaf running on the Intel® Xeon Phi^{TM} coprocessor. We simply recompiled the existing code using the -mmic compiler option and ran executable natively on the coprocessor”

What was the most difficult hurdle?

- Code originally not written with much consideration to how well it would vectorise
  - Compiler reports helped
  - Compiler reports were sometime confusing, with multiple message relating to one line
  - Has to resort to looking at assembler

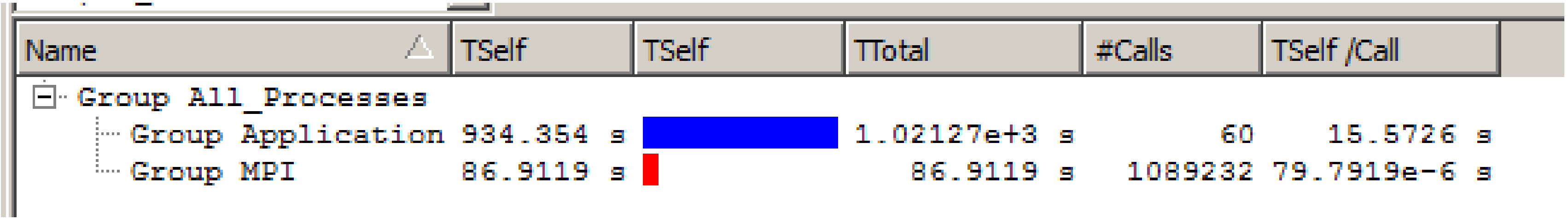
Speedup



Mesh Size:	240 x 240	480 x 480	960 x 960	1920 x 1920	3440 x 3840
🟦 Single Processor **	7	28	179	781	3000
🟥 Xeon Phi ***	27	40	89	260	920
🟩 Speedup	0.27	0.70	2.01	3.00	3.26

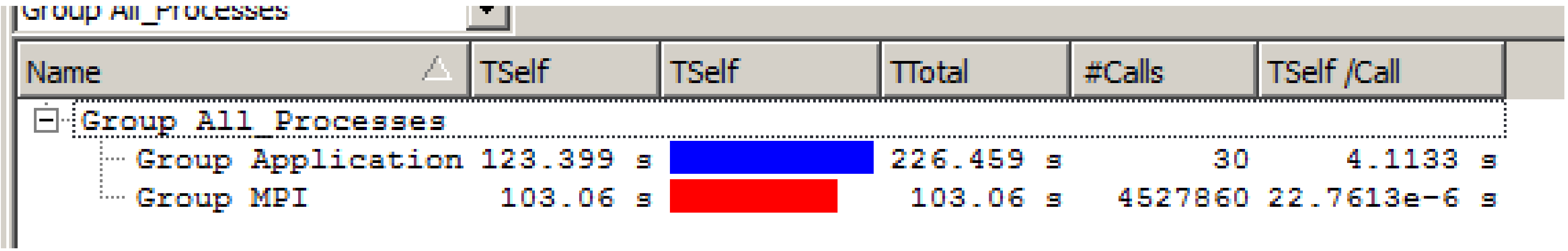
Single socket eight core Intel® Xeon® E5-2687W processor, 3.1GHz, 32GB DDR3(1333Mhz) memory, with both Turbo boost and Hyperthreading enabled. * Intel® Xeon Phi^{TM} coprocessor had 61 cores, running at 1.09GHz, with 8GB of GDDR5 (5.5 GT/s) memory.

Using Intel Trace Analyzer Collector (ITAC)



Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
Group All_Processes					
Group Application	934.354 s	🟦	1.02127e+3 s	60	15.5726 s
Group MPI	86.9119 s	🟥	86.9119 s	1089232	79.7919e-6 s

Running Cloverleaf natively on Xeon Phi with a 3840 x 3840 mesh size.

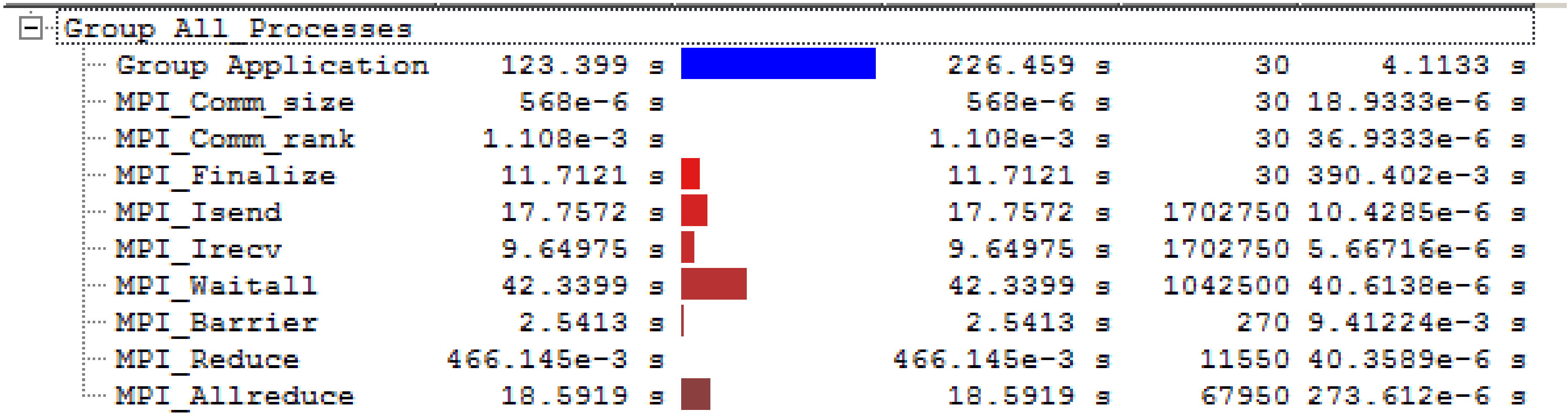


Group All_Processes

Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
Group All_Processes					
Group Application	123.399 s	🟦	226.459 s	30	4.1133 s
Group MPI	103.06 s	🟥	103.06 s	4527860	22.7613e-6 s

Running Cloverleaf natively on Xeon Phi with a 250 x 250 mesh size.

Overhead is spread over a number of MPI APIS.



Group All_Processes				
Group Application	123.399 s	226.459 s	30	4.1133 s
MPI_Comm_size	568e-6 s	568e-6 s	30	18.9333e-6 s
MPI_Comm_rank	1.108e-3 s	1.108e-3 s	30	36.9333e-6 s
MPI_Finalize	11.7121 s	11.7121 s	30	390.402e-3 s
MPI_Isend	17.7572 s	17.7572 s	1702750	10.4285e-6 s
MPI_Irecv	9.64975 s	9.64975 s	1702750	5.66716e-6 s
MPI_Waitall	42.3399 s	42.3399 s	1042500	40.6138e-6 s
MPI_Barrier	2.5413 s	2.5413 s	270	9.41224e-3 s
MPI_Reduce	466.145e-3 s	466.145e-3 s	11550	40.3589e-6 s
MPI_Allreduce	18.5919 s	18.5919 s	67950	273.612e-6 s

The breakdown of MPI calls on the 250 x 250 mesh size.

Not enough work is being done in each MPI task

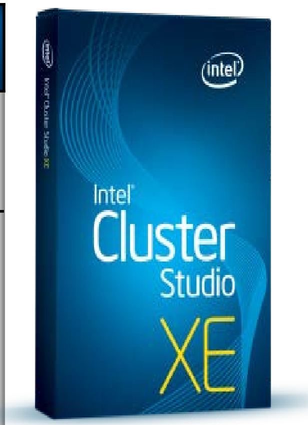
Two essential tools

Intel® **VTune** Ampifier XE

Intel® Trace analyzer and Collector **(ITAC)**

Intel® Cluster Studio XE

Phase	Product	Feature
Build	Intel® Advisor XE	Threading design assistant
	Intel® Composer XE	• C/C++ and Fortran compilers • Intel® Threading Building Blocks • Intel® Cilk™ Plus • Intel® Integrated Performance Primitives • Intel® Math Kernel Library
	Intel® MPI Library	High Performance Message Passing (MPI) Library
Verify & Tune	Intel® VTune™ Amplifier XE	Performance Profiler for optimizing application performance and scalability
	Intel® Inspector XE	Memory & threading dynamic analysis for code quality Static Analysis for code quality
	Intel® Trace Analyzer & Collector	MPI Performance Profiler for understanding application correctness & behavior



Efficiently Produce Fast, Scalable and Reliable Applications. *Including on Xeon Phi*

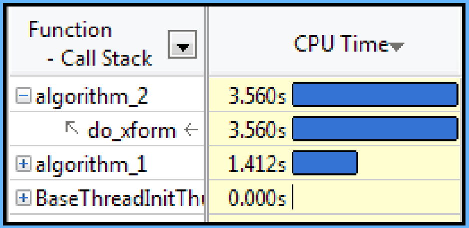
VTune

Intel® VTune™ Amplifier XE

Performance Profiler

Where is my application…

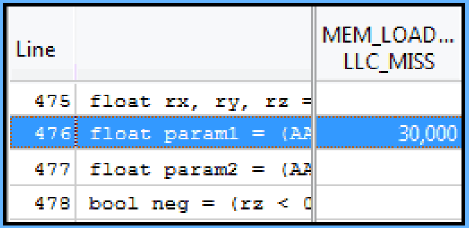
Spending Time?



Function - Call Stack	CPU Time
⊟ algorithm_2	3.560s
↖ do_xform ←	3.560s
⊞ algorithm_1	1.412s
⊞ BaseThreadInitThı	0.000s

- Focus tuning on functions taking time
- See call stacks
- See time on source

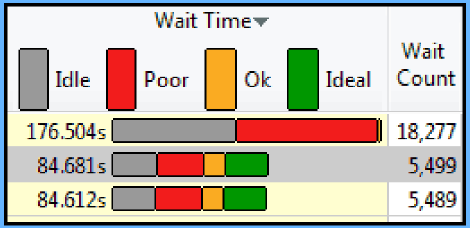
Wasting Time?



Line		MEM_LOAD… LLC_MISS
475	float rx, ry, rz =	
476	float param1 = (AA	30,000
477	float param2 = (AA	
478	bool neg = (rz < 0	

- See cache misses on your source
- See functions sorted by # of cache misses

Waiting Too Long?



Wait Time Idle Poor Ok Ideal	Wait Count
176.504s	18,277
84.681s	5,499
84.612s	5,489

- See locks by wait time
- Red/Green for CPU utilization during wait

- **Windows & Linux**
- **Low overhead**
- **No special recompiles**

Advanced Profiling For Scalable Multicore Performance

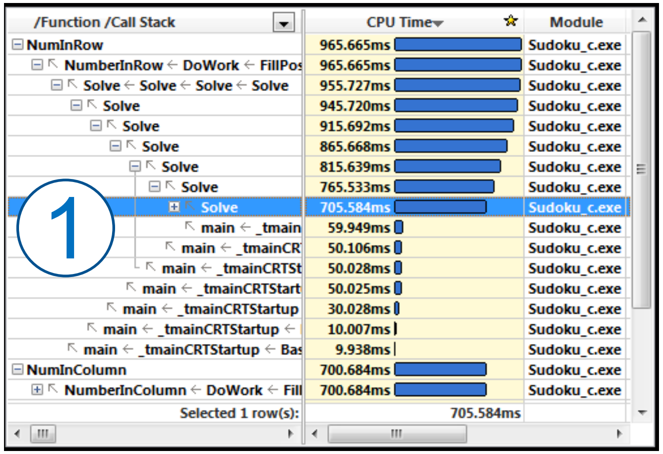
VTune Amplifier is a simple tool

Imagine you have a cool car and you want to drive a little faster or fuel effective

All what you'd need you can find here.

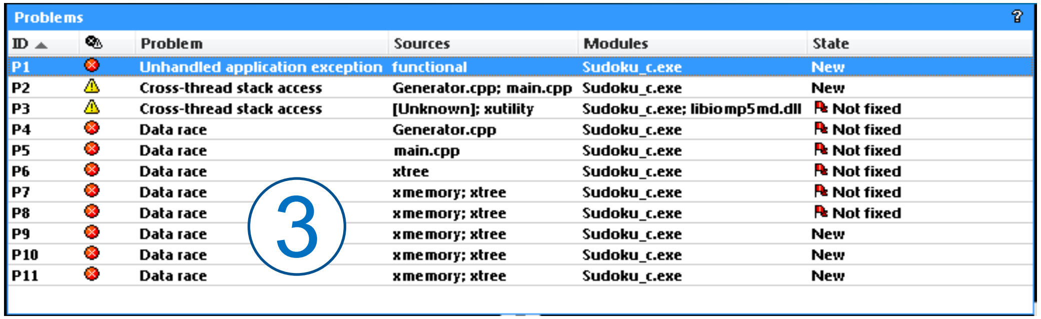
VTune as other simple tools can provide basic information on performance of your engine.

An example of user mode analysis…



1

/Function /Call Stack	CPU Time	Module
⊟ NumInRow	965.665ms	Sudoku_c.exe
⊟ ↖ NumberInRow ← DoWork ← FillPos	965.665ms	Sudoku_c.exe
⊟ ↖ Solve ← Solve ← Solve ← Solve	955.727ms	Sudoku_c.exe
⊟ ↖ Solve	945.720ms	Sudoku_c.exe
⊟ ↖ Solve	915.692ms	Sudoku_c.exe
⊟ ↖ Solve	865.668ms	Sudoku_c.exe
⊟ ↖ Solve	815.639ms	Sudoku_c.exe
⊟ ↖ Solve	765.533ms	Sudoku_c.exe
⊞ ↖ Solve	705.584ms	Sudoku_c.exe
↖ main ← _tmain	59.949ms	Sudoku_c.exe
↖ main ← _tmainCR	50.106ms	Sudoku_c.exe
↖ main ← _tmainCRTSt	50.028ms	Sudoku_c.exe
↖ main ← _tmainCRTStart	50.025ms	Sudoku_c.exe
↖ main ← _tmainCRTStartup	30.028ms	Sudoku_c.exe
↖ main ← _tmainCRTStartup ←	10.007ms	Sudoku_c.exe
↖ main ← _tmainCRTStartup ← Bas	9.938ms	Sudoku_c.exe
⊟ NumInColumn	700.684ms	Sudoku_c.exe
⊞ ↖ NumberInColumn ← DoWork ← Fill	700.684ms	Sudoku_c.exe
Selected 1 row(s):	705.584ms	

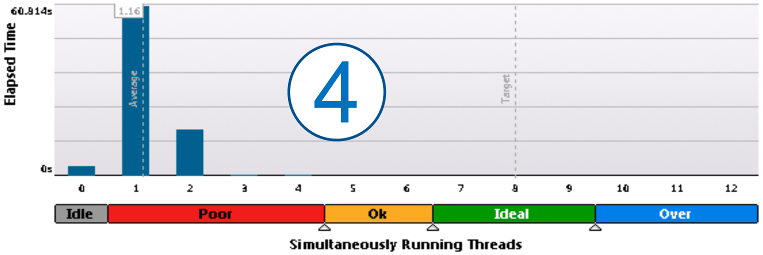


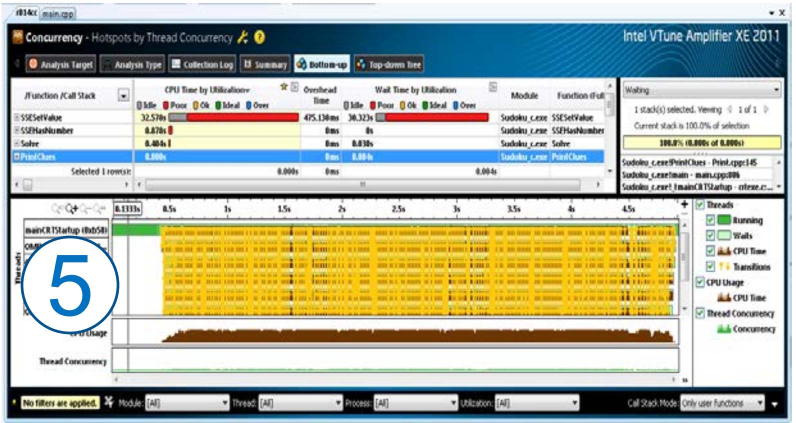
Problems

3

ID		Problem	Sources	Modules	State
P1	⊗	Unhandled application exception	functional	Sudoku_c.exe	New
P2	⚠	Cross-thread stack access	Generator.cpp; main.cpp	Sudoku_c.exe	New
P3	⚠	Cross-thread stack access	[Unknown]; xutility	Sudoku_c.exe; libiomp5md.dll	Not fixed
P4	⊗	Data race	Generator.cpp	Sudoku_c.exe	Not fixed
P5	⊗	Data race	main.cpp	Sudoku_c.exe	Not fixed
P6	⊗	Data race	xtree	Sudoku_c.exe	Not fixed
P7	⊗	Data race	xmemory; xtree	Sudoku_c.exe	Not fixed
P8	⊗	Data race	xmemory; xtree	Sudoku_c.exe	Not fixed
P9	⊗	Data race	xmemory; xtree	Sudoku_c.exe	New
P10	⊗	Data race	xmemory; xtree	Sudoku_c.exe	New
P11	⊗	Data race	xmemory; xtree	Sudoku_c.exe	New

2





5

Concurrency - Hotspots by Thread Concurrency

Intel VTune Amplifier XE 2011

/Function /Call Stack	CPU Time by Utilization	Overhead Time	Wait Time by Utilization	Module	Function (Full
SSESetValue	32.578s	475.130ms	30.323s	Sudoku_c.exe	SSESetValue
SSEHasNumber	0.878s	0ms	0s	Sudoku_c.exe	SSEHasNumber
Solve	0.404s	0ms	0.038s	Sudoku_c.exe	Solve
PrintClues	0.000s	0ms	0.004s	Sudoku_c.exe	PrintClues
Selected 1 row(s):	0.000s	0ms	0.004s		

Waiting

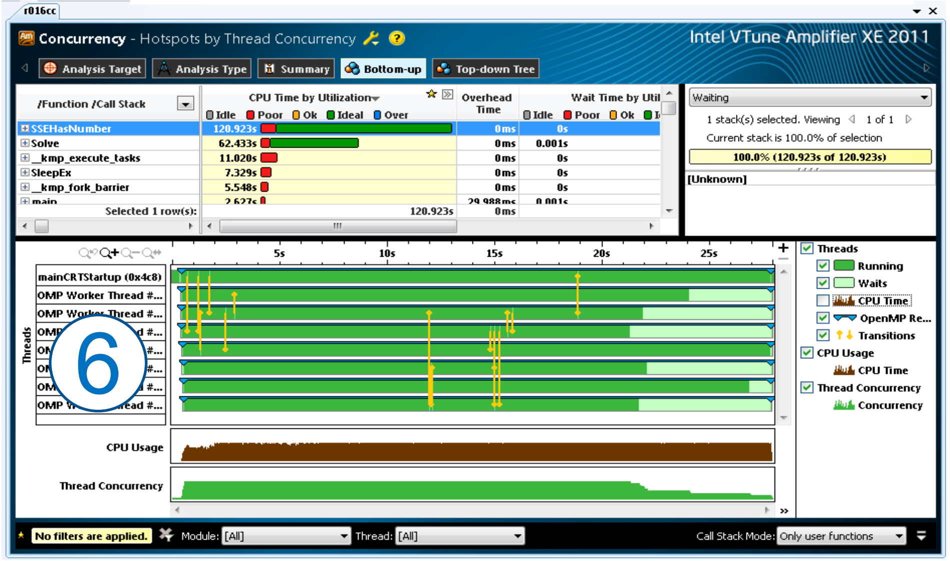
1 stack(s) selected. Viewing 1 of 1

Current stack is 100.0% of selection

100.0% (0.000s of 0.000s)

Sudoku_c.exe!PrintClues - Print.cpp:145
Sudoku_c.exe!main - main.cpp:806
Sudoku_c.exe!_tmainCRTStartup - crtexe.c...

No filters are applied. Module: [All] Thread: [All] Process: [All] Utilization: [All] Call Stack Mode: Only user functions



6

Concurrency - Hotspots by Thread Concurrency

Intel VTune Amplifier XE 2011

/Function /Call Stack	CPU Time by Utilization	Overhead Time	Wait Time by Util
⊞ SSEHasNumber	120.923s	0ms	0s
⊞ Solve	62.433s	0ms	0.001s
⊞ __kmp_execute_tasks	11.020s	0ms	0s
⊞ SleepEx	7.329s	0ms	0s
⊞ __kmp_fork_barrier	5.548s	0ms	0s
⊞ main	2.627s	29.988ms	0.001s
Selected 1 row(s):	120.923s	0ms	

Waiting

1 stack(s) selected. Viewing 1 of 1

Current stack is 100.0% of selection

100.0% (120.923s of 120.923s)

[Unknown]

No filters are applied. Module: [All] Thread: [All] Call Stack Mode: Only user functions

1. Hotspot Analysis
2. Implement
3. Find Threading Errors

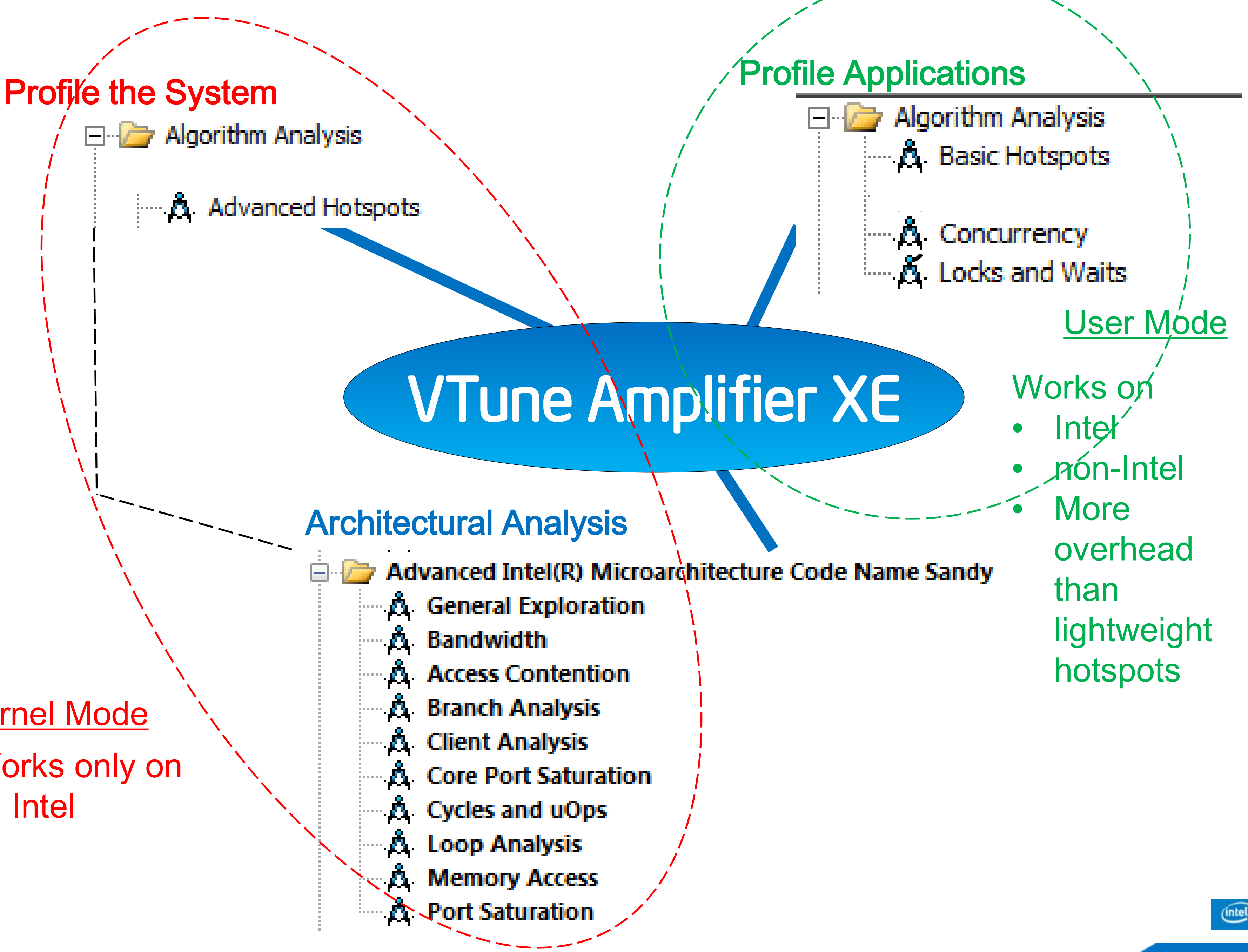
4,5,6. Tune Parallelism

VTune Amplifier is a complex tool

However, if you want your car to win a race...

Your tools set has to be much more complex to analyze all aspects of engine functioning.

You need to be more proficient in both: the tool’s functionality and the engine internals!



Profile the System

- Algorithm Analysis
  - Advanced Hotspots

Profile Applications

- Algorithm Analysis
  - Basic Hotspots
  - Concurrency
  - Locks and Waits

VTune Amplifier XE

Architectural Analysis

- Advanced Intel(R) Microarchitecture Code Name Sandy
  - General Exploration
  - Bandwidth
  - Access Contention
  - Branch Analysis
  - Client Analysis
  - Core Port Saturation
  - Cycles and uOps
  - Loop Analysis
  - Memory Access
  - Port Saturation

User Mode

Works on
- Intel
- non-Intel
- More overhead than lightweight hotspots

Kernel Mode

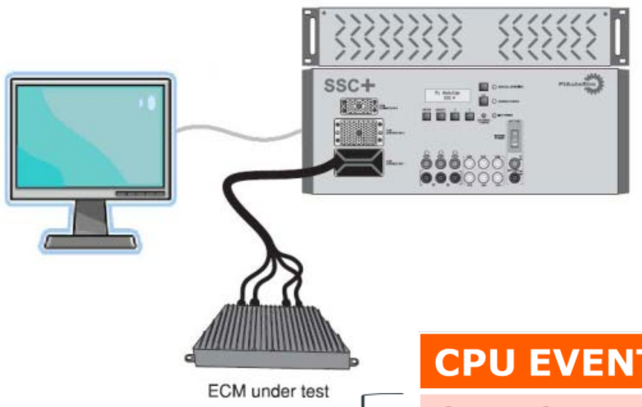
Works only on
- Intel

An example of architectural analysis

Speedup by upgrading silicon

CPU	No Auto-Vectorisation	With Auto-Vectorisation	Speedup
P4	39.344	21.9	**1.80**
Core 2	5.546	0.515	**10.77**
Speedup	**7.09**	**45.52**	**76**

Speedup by swapping compiler



Verified using VTune

CPU EVENT	Without Vect	With Vect
CPU_CLK_UNHALTED.CORE	16,641,000,448	1,548,000,000
INST_RETIRED.ANY	3,308,999,936	1,395,000,064
X87_OPS_RETIRED.ANY	250,000,000	0
SIMD_INST_RETIRED	0	763,000,000

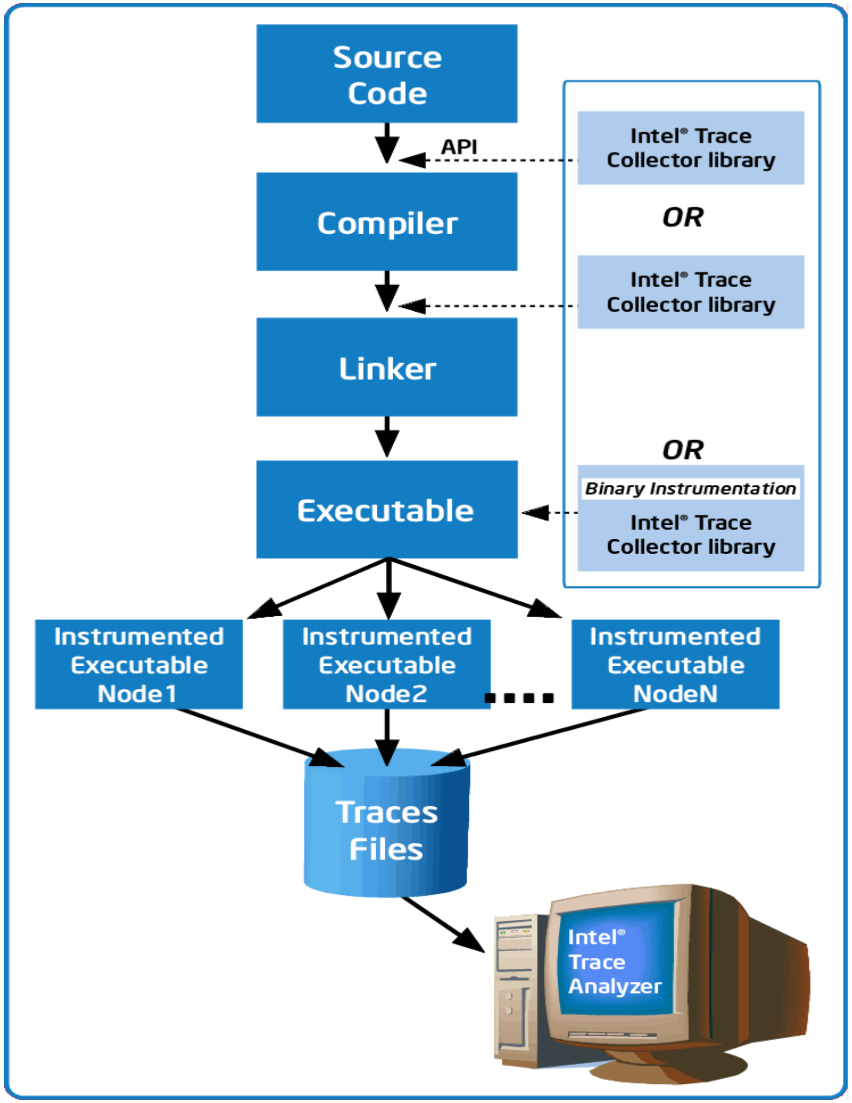
Full paper available here: http://edc.intel.com/Link.aspx?id=1045

ITAC

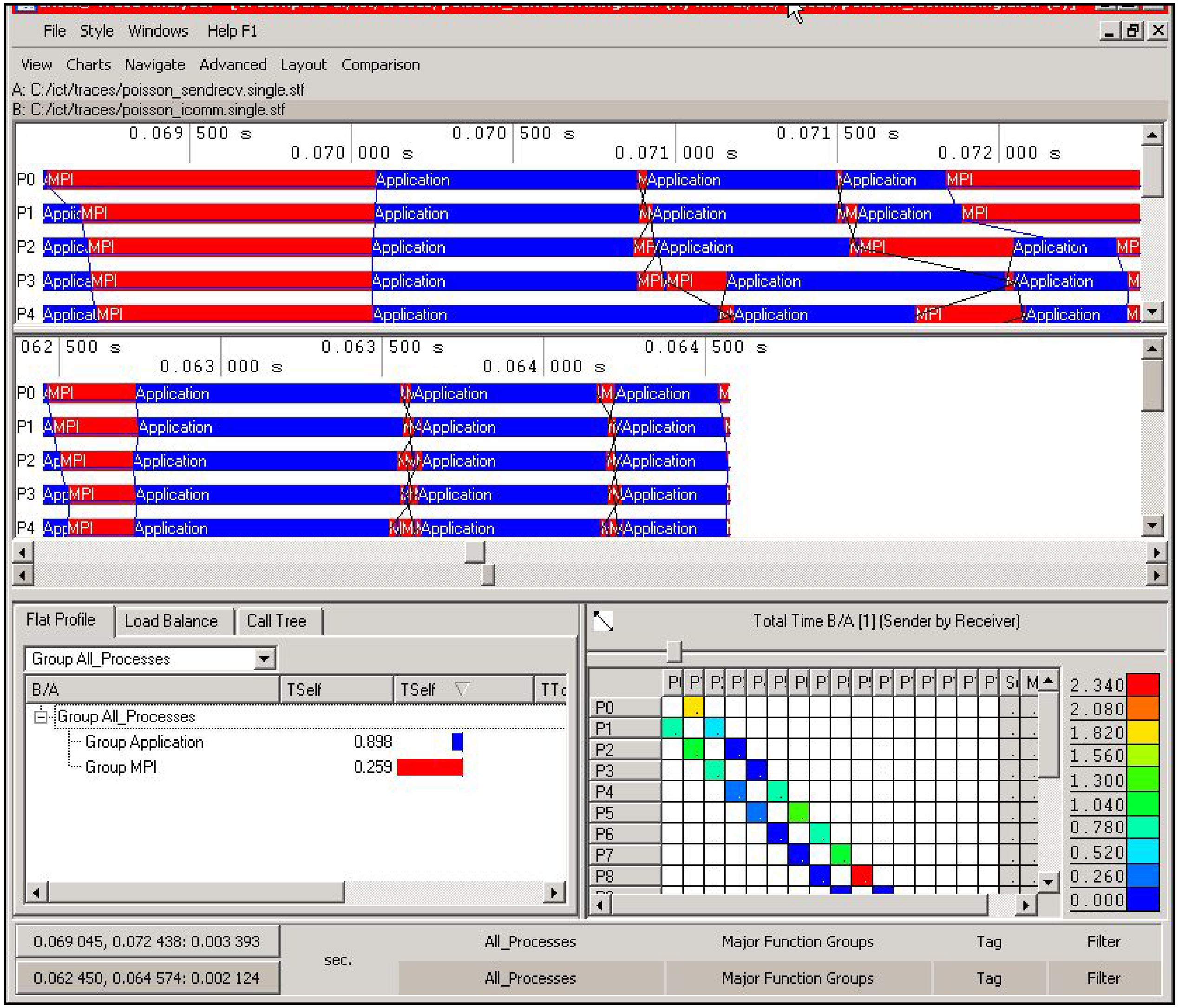
Intel® Trace Analyzer and Collector Overview

Intel® Trace Analyzer and Collector helps the developer:

- Visualize and understand parallel application behavior
- Evaluate profiling statistics and load balancing
- Identify communication hotspots


Source Code
API
Intel® Trace Collector library
OR
Compiler
Intel® Trace Collector library
Linker
OR
Executable
Binary Instrumentation
Intel® Trace Collector library
Instrumented Executable Node1
Instrumented Executable Node2
....
Instrumented Executable NodeN
Traces Files
Intel® Trace Analyzer

Intel® Trace Analyzer and Collector



B/A	TSelf	TSelf	TT
Group All_Processes			
Group Application		0.898	
Group MPI		0.259	

0.069 045, 0.072 438: 0.003 393

0.062 450, 0.064 574: 0.002 124

Compare the event timelines of two communication profiles

Blue = computation
Red = communication

Chart showing how the MPI processes interact

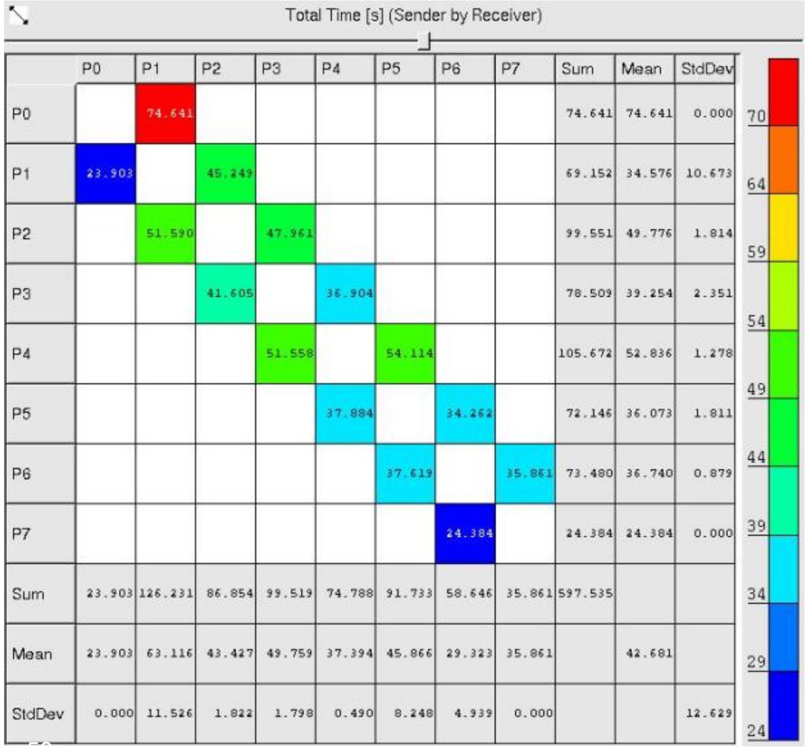
Communication Profiles

Statistics about point-to-point or collective communication

Generic matrix supports grouping by several attributes in each dimension

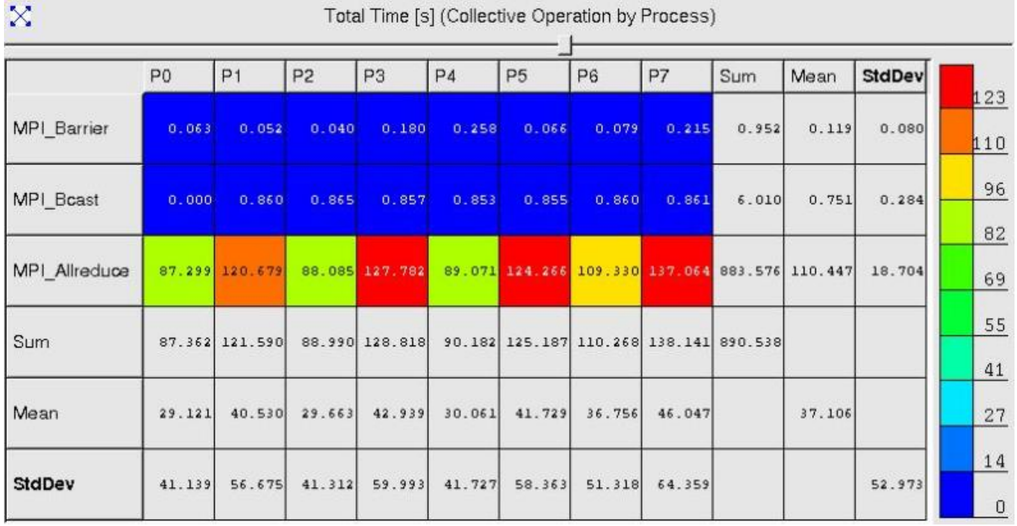
Sender, Receiver, Data volume per msg, Tag, Communicator, Type

Available attributes: Count, Bytes transferred, Time, Transfer rate



Total Time [s] (Sender by Receiver)

	P0	P1	P2	P3	P4	P5	P6	P7	Sum	Mean	StdDev
P0		74.641							74.641	74.641	0.000
P1	23.903		45.249						69.152	34.576	10.673
P2		51.590		47.961					99.551	49.776	1.814
P3			41.605		36.904				78.509	39.254	2.351
P4				51.558		54.114			105.672	52.836	1.278
P5					37.884		34.262		72.146	36.073	1.811
P6						37.619		35.861	73.480	36.740	0.879
P7							24.384		24.384	24.384	0.000
Sum	23.903	126.231	86.854	99.519	74.788	91.733	58.646	35.861	597.535		
Mean	23.903	63.116	43.427	49.759	37.394	45.866	29.323	35.861		42.681	
StdDev	0.000	11.526	1.822	1.798	0.490	8.248	4.939	0.000			12.629



Total Time [s] (Collective Operation by Process)

	P0	P1	P2	P3	P4	P5	P6	P7	Sum	Mean	StdDev
MPI_Barrier	0.063	0.052	0.040	0.180	0.258	0.066	0.079	0.215	0.952	0.119	0.080
MPI_Bcast	0.000	0.860	0.865	0.857	0.853	0.855	0.860	0.861	6.010	0.751	0.284
MPI_Allreduce	87.299	120.679	88.085	127.782	89.071	124.266	109.330	137.064	883.576	110.447	18.704
Sum	87.362	121.590	88.990	128.818	90.182	125.187	110.268	138.141	890.538		
Mean	29.121	40.530	29.663	42.939	30.061	41.729	36.756	46.047		37.106	
StdDev	41.139	56.675	41.312	59.993	41.727	58.363	51.318	64.359			52.973

Typical Hands-on Xeon Phi training agenda

25th & 26th
June 2014
Manchester

Day 1 – Getting Ready

10.00	Welcome
10.30	Two Essential Requirements
11.00	Parallelism (L)
12.30	Lunch
1.30	Vectorisation (L)
4.00	Advance Profiling (Walkthrough)
5.00	End

Day 2-Xeon Phi Programming

09.00	Start
09.15	Native & Offload Programming for Xeon Phi (L)
11.30	A Case Study
12.00	Lunch
1.00	Vectorisation on Xeon Phi (L)
1.50	Parallelism on Xeon Phi (L)
3.40	Wrap-up
4.00	End