



Multithreaded programming on hybrid parallel architectures

Silvio Stanzani , Raphael Cóbe , Rogério Iope, Jefferson Fialho

UNESP - Núcleo de Computação Científica

silvio@ncc.unesp.br , rmcobe@ncc.unesp.br ,
rogerio@ncc.unesp.br , jfialho@ncc.unesp.br

Agenda

- Hybrid Parallel Architectures
- Intel HPC Architectures
- OpenMP
- Profiling with Intel Advisor (Threading Workflow)
- Critical Sections
- Offload
- N-Body Simulation

UNESP Center for Scientific Computing

- Consolidates scientific computing resources for São Paulo State University (UNESP) researchers
 - It mainly uses Grid computing paradigm
- Main users
 - UNESP researchers, students, and software developers
 - SPRACE (São Paulo Research and Analysis Center) physicists and students
 - ❑ Caltech, Fermilab, CERN
 - ❑ São Paulo CMS Tier-2 Facility

Hybrid Parallel Architectures

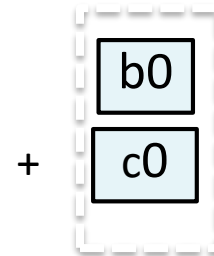
- Heterogeneous computational systems:
 - Multicore processors;
 - Multi-level memory sub-system;
- Multi-level parallelism:
 - Processing core;
 - Chip multiprocessor;
 - Computing node;
 - Computing cluster;
- Hybrid Parallel architectures
 - Coprocessors and accelerators;

Scalar and Vector Instructions

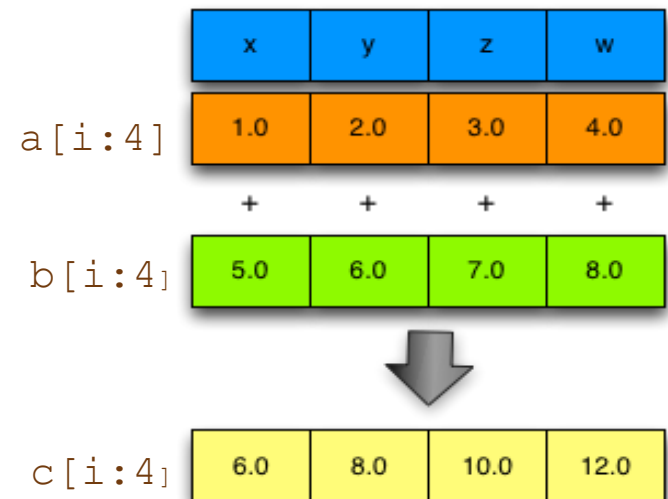
- **Scalar** Code computes this one-element at a time.
- **Vector (or SIMD)** Code computes more than one element at a time.
 - SIMD stands for **Single Instruction Multiple Data**.
- **Vectorization**
 - Loading data into cache accordingly;
 - Store elements on SIMD registers or vectors;
 - Iterations need to be independent;
 - Usually on inner loops.

```
float *A, *B, *C;  
for(i=0;i<n;i++){  
    A[i] = B[i] + C[i];  
}
```

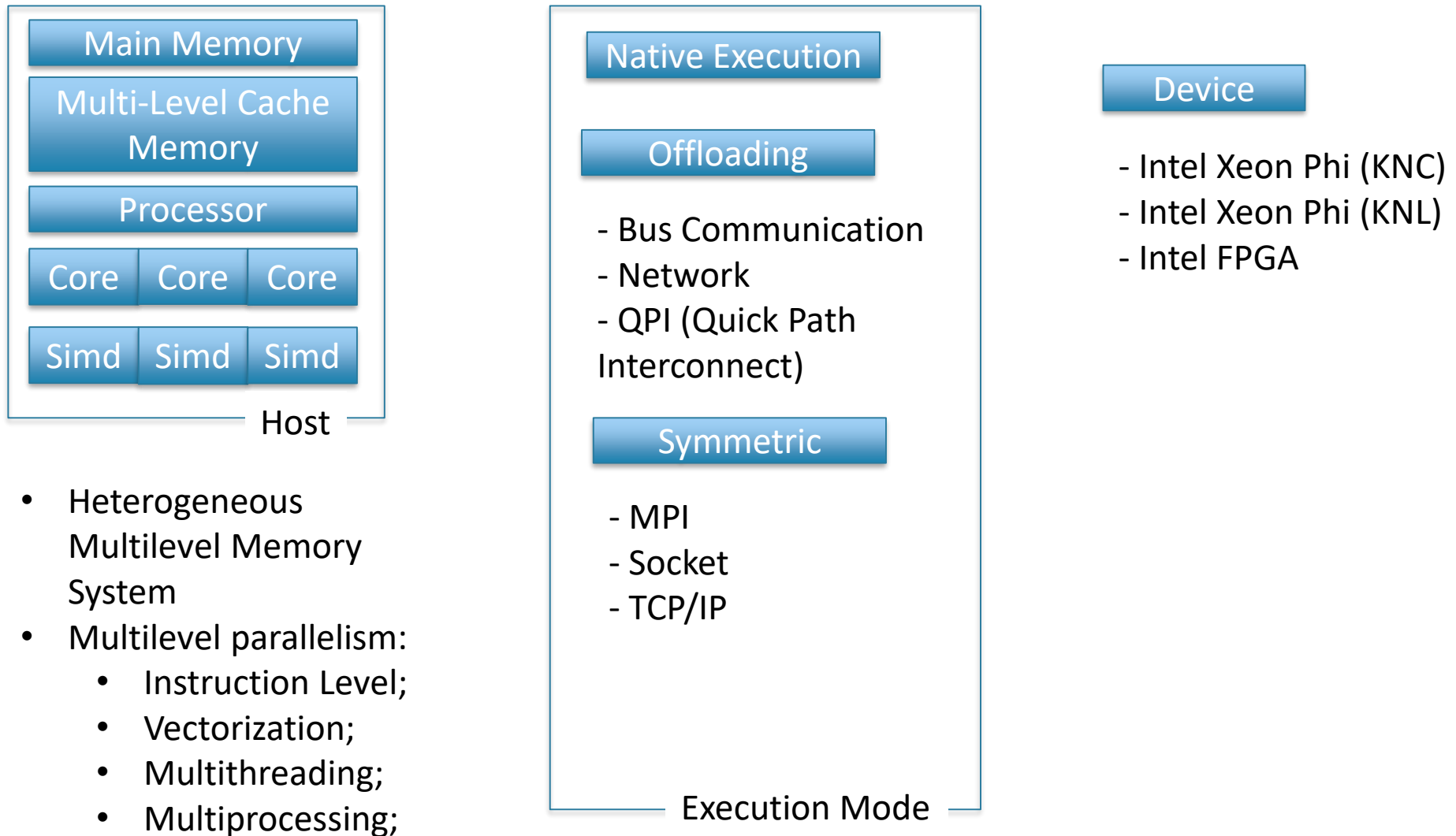
- Scalar



- SIMD



Hybrid Parallel Architectures

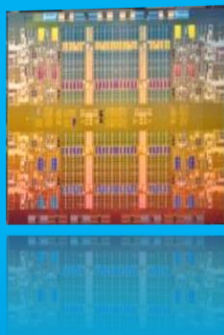


Agenda

- Hybrid Parallel Architectures
- Intel HPC Architectures
- OpenMP
- Profiling with Intel Advisor (Threading Workflow)
- Critical Sections
- Offload
- N-Body Simulation

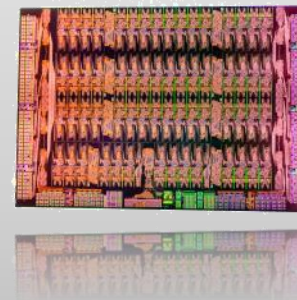
Intel Xeon and Intel® Xeon Phi™ Overview

Intel® Multicore Architecture



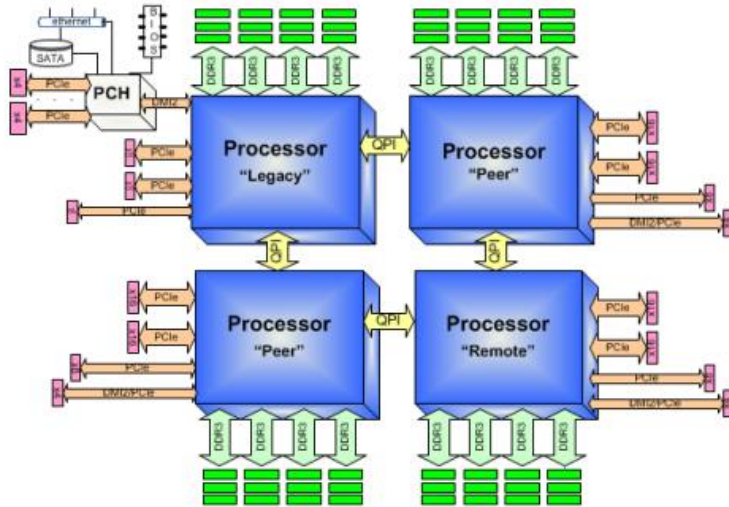
- ❖ Foundation of HPC Performance
- ❖ Suited for full scope of workloads
- ❖ Focus on fast single core/thread performance with “moderate” number of cores

Intel® Many Integrated Core Architecture

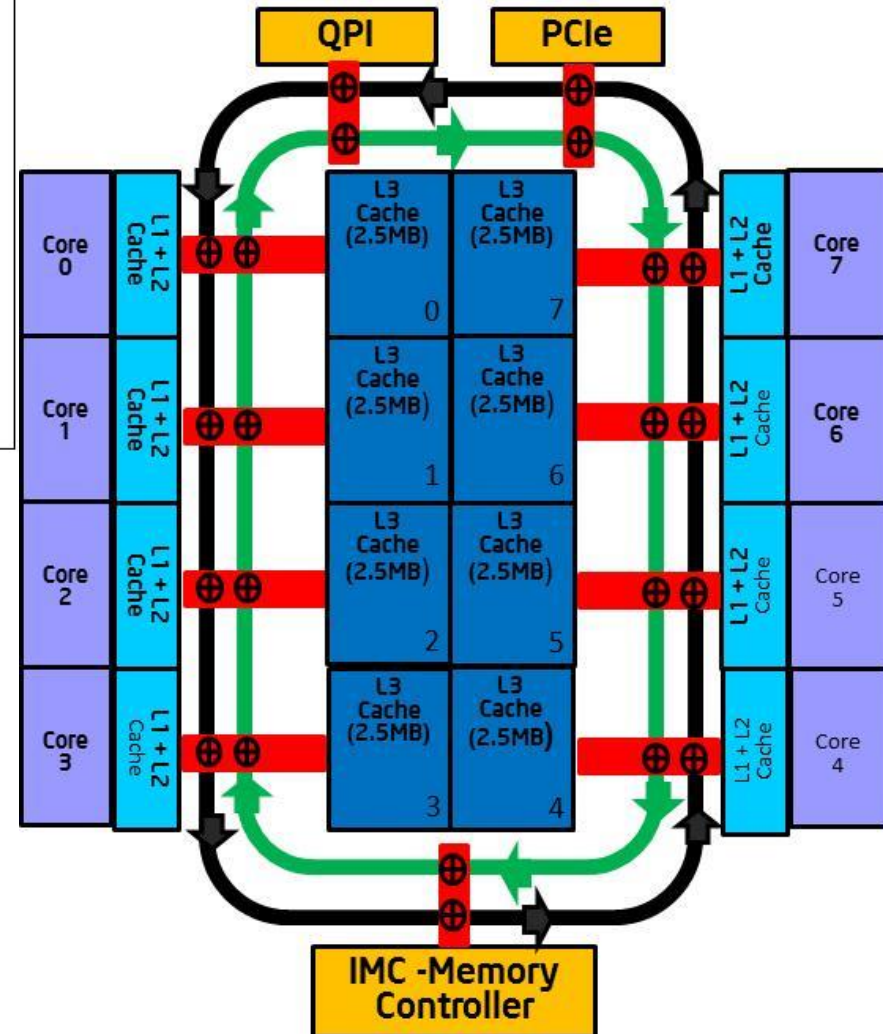


- ❖ Performance and performance/watt optimized for highly parallelized compute workloads
- ❖ IA extension to Manycore
- ❖ Many cores/threads with wide SIMD

Intel Xeon Architecture Overview

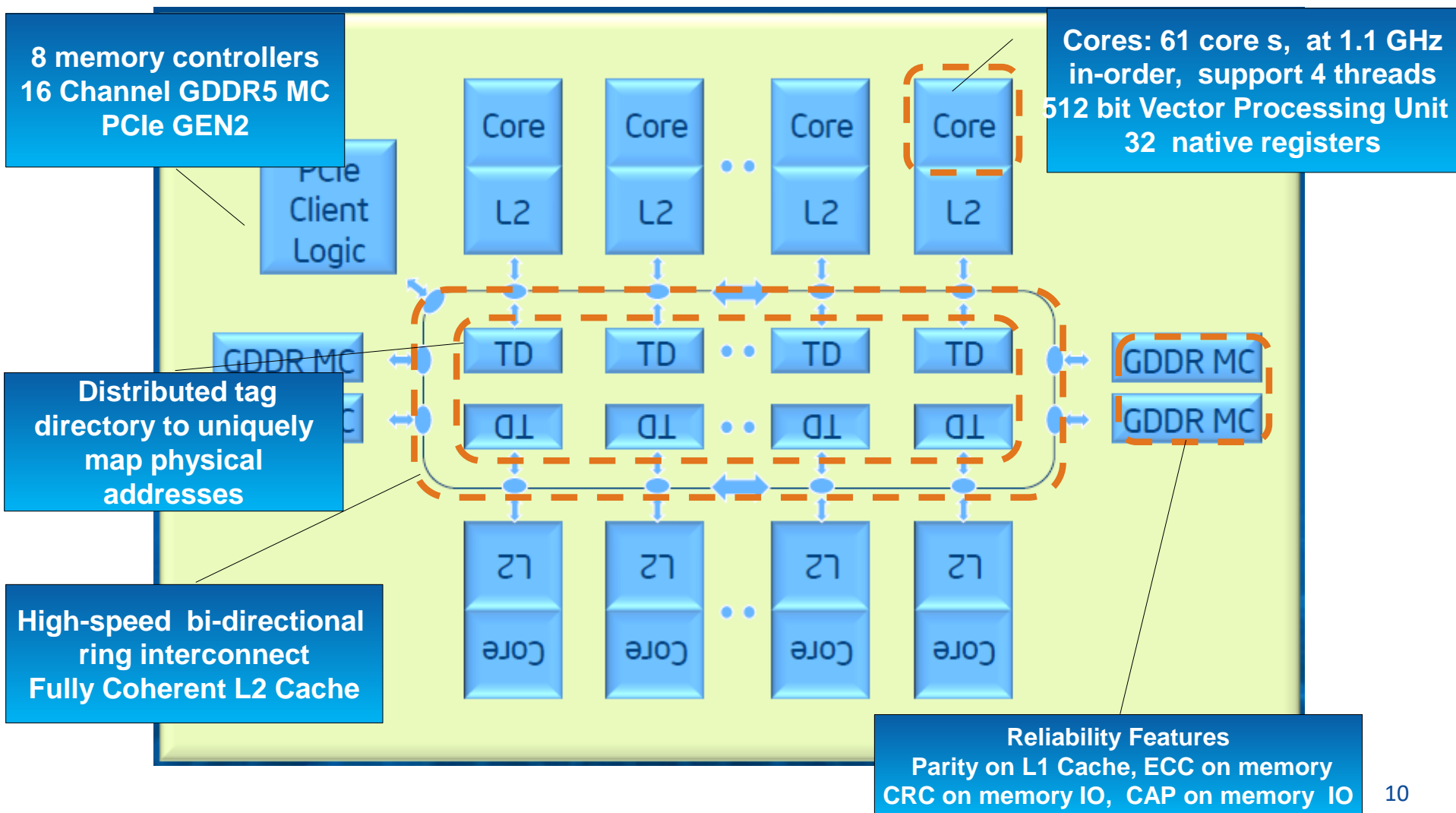


- Socket: mechanical component that provides mechanical and electrical connections between a microprocessor and a printed circuit board (PCB).
- QPI (Intel QuickPath Interconnect): high speed, packetized, point-to-point interconnection, that stitch together processors in distributed shared memory and integrated I/O platform architecture.



Intel® Xeon Phi™ Architecture Overview

- Knights Core (KNC)



- Large SMP UMA machine – a set of x86 cores
 - 4 threads
 - ❑ 32 KB L1 I/D
 - ❑ 512 KB L2 per core
 - Supports loadable kernel modules
 - VM subsystem, File I/O
- Virtual Ethernet driver
 - supports NFS mounts from Intel® Xeon Phi™ Coprocessor
 - Support bridged network

Knights Landing (KNL)

Over 3 TF DP peak
Full Xeon ISA compatibility through AVX-512
~3x single-thread vs. compared to Knights Corner

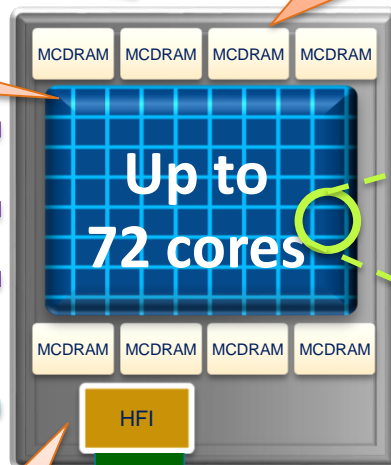
Up to 16GB high-bandwidth on-package memory (MCDRAM)
Exposed as NUMA node
~500 GB/s sustained BW

Up to 72 cores
2D mesh architecture

DDR4

DDR4

DDR4



Up to 72 cores

MCDRAM MCDRAM MCDRAM MCDRAM

MCDRAM MCDRAM MCDRAM MCDRAM

Wellsburg PCH

DMI

HFI

Connector

PCIe Gen3
x36

Twinax Cable

Twinax Cable

DDR4

DDR4

DDR4

Tile

2 VPU

HUB

2 VPU

Core

1MB L2

Core

2x 512b VPU per core
(Vector Processing Units)

6 channels
DDR4
Up to 384GB

Common with
Grantley
PCH

2 ports
Intel Omni-Path Fabric On-package
50 GB/s bi-directional

Based on Intel® Atom Silvermont processor with many HPC enhancements
Deep out-of-order buffers
Gather/scatter in hardware
Improved branch prediction
4 threads/core
High cache bandwidth
& more

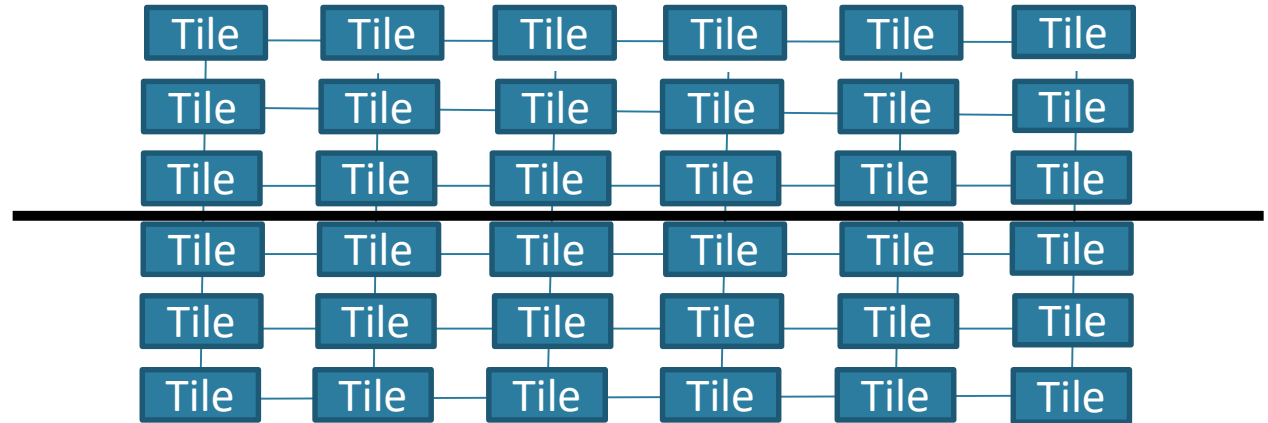
Cluster modes

One single space address

Hemisphere:

the tiles are divided into two parts called hemisphere

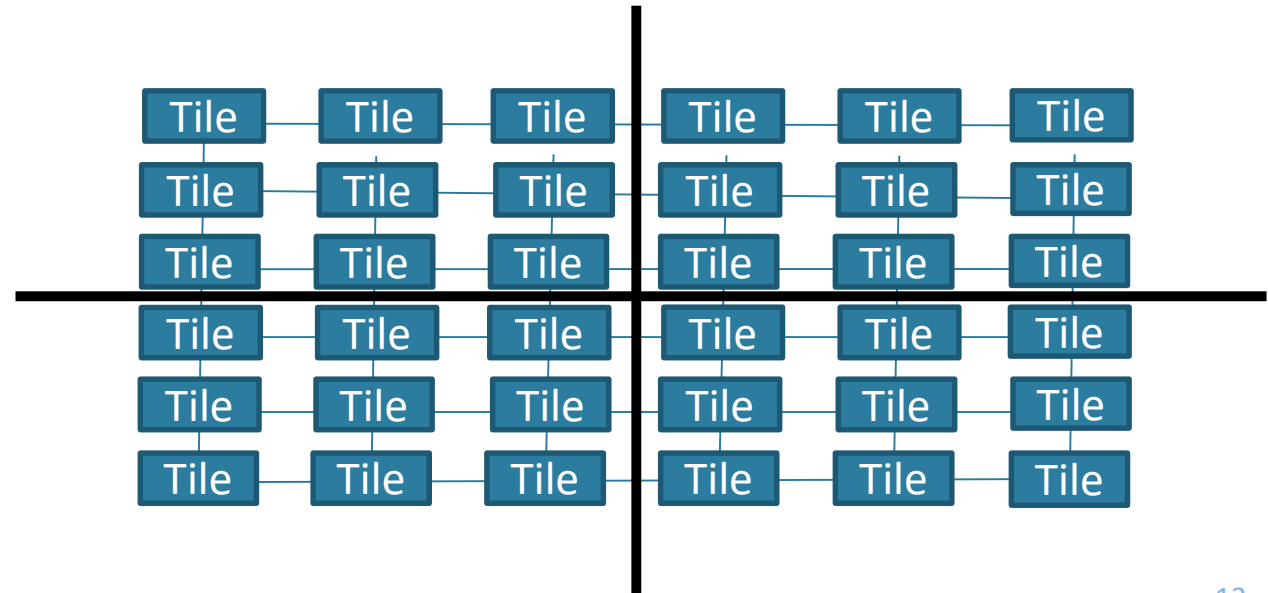
Node 0



Quadrant:

tiles are divided into two parts called hemisphere or into four parts called quadrants

Node 0

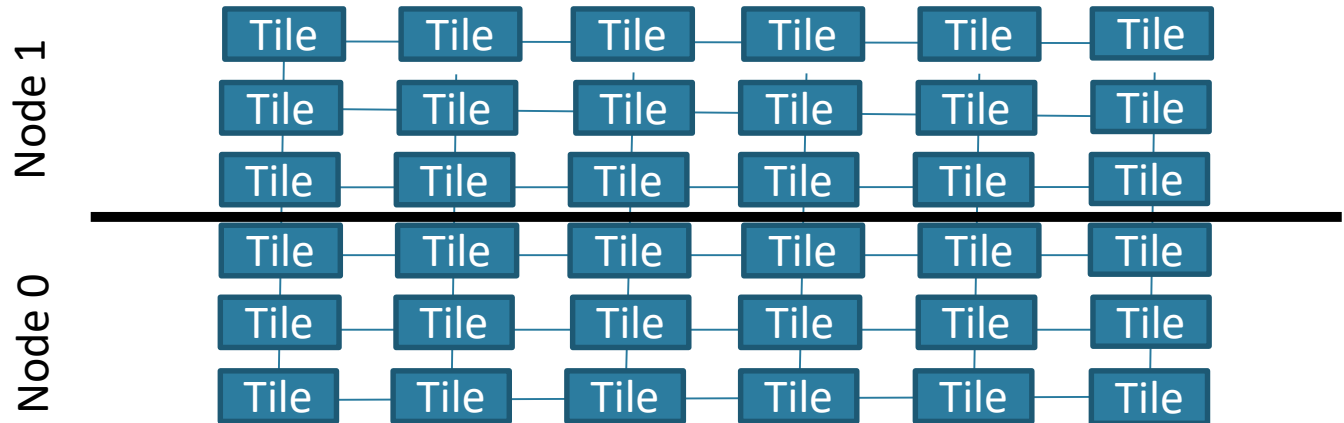


Cluster modes

Cache data are isolated in each sub numa domain

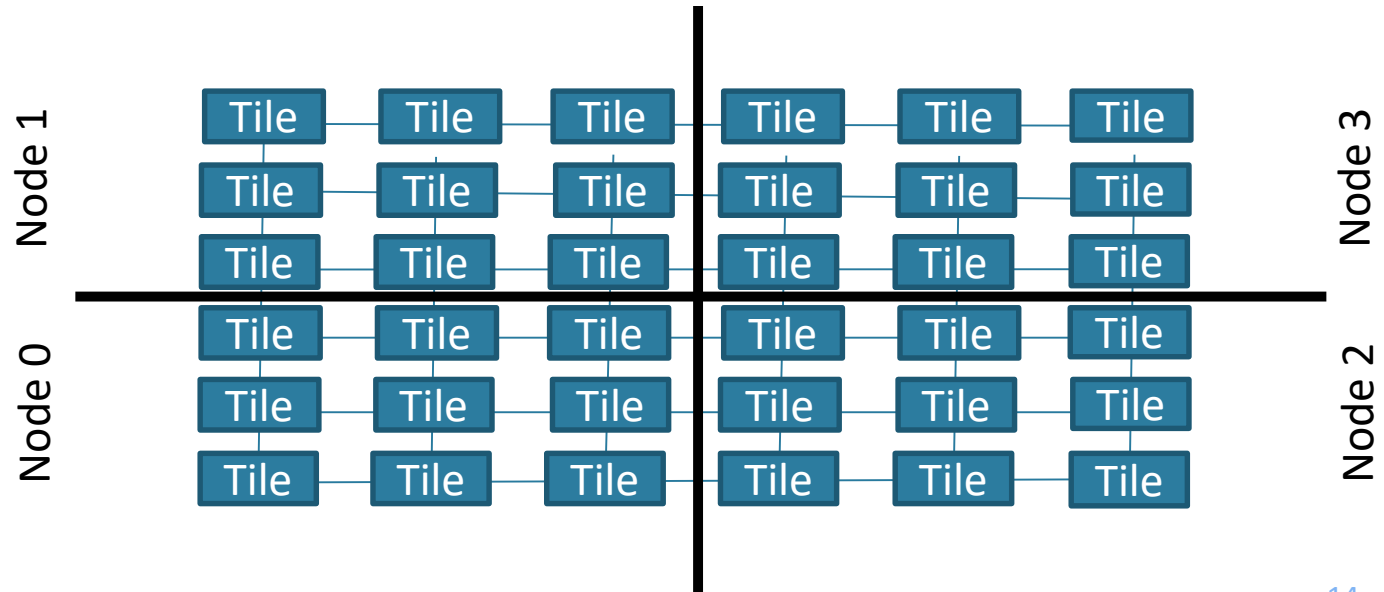
SNC-2:

the tiles are
divided into two
Numa Nodes



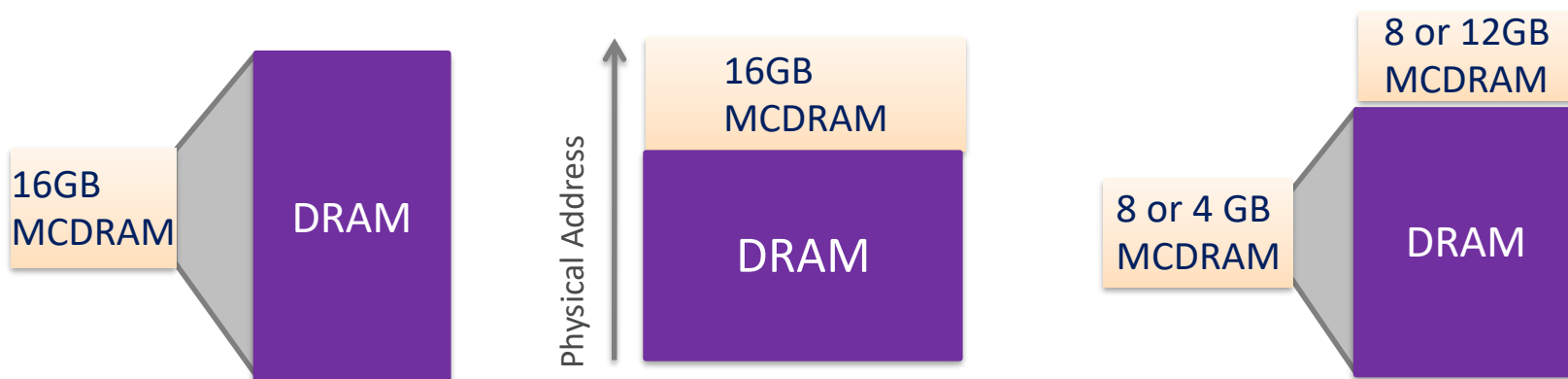
SNC-4:

the tiles are
divided into two
Numa Nodes



Integrated On-Package Memory Usage Models

Integrated On-Package Memory Usage Models



Split Options:
25/75% or 50/50%

Cache Model	Flat Model	Hybrid Model
Hardware automatically manages the MCDRAM as a “L3 cache” between CPU and ext DDR memory	Manually manage how the app uses the integrated on-package memory and external DDR for peak perf	Harness the benefits of both Cache and Flat models by segmenting the integrated on-package memory
<ul style="list-style-type: none">▪ App and/or data set is very large and will not fit into MCDRAM▪ Unknown or unstructured memory access behavior	<ul style="list-style-type: none">▪ App or portion of an app or data set that can be, or is needed to be “locked” into MCDRAM so it doesn’t get flushed out	<ul style="list-style-type: none">▪ Need to “lock” in a relatively small portion of an app or data set via the Flat model▪ Remaining MCDRAM can then be configured as Cache

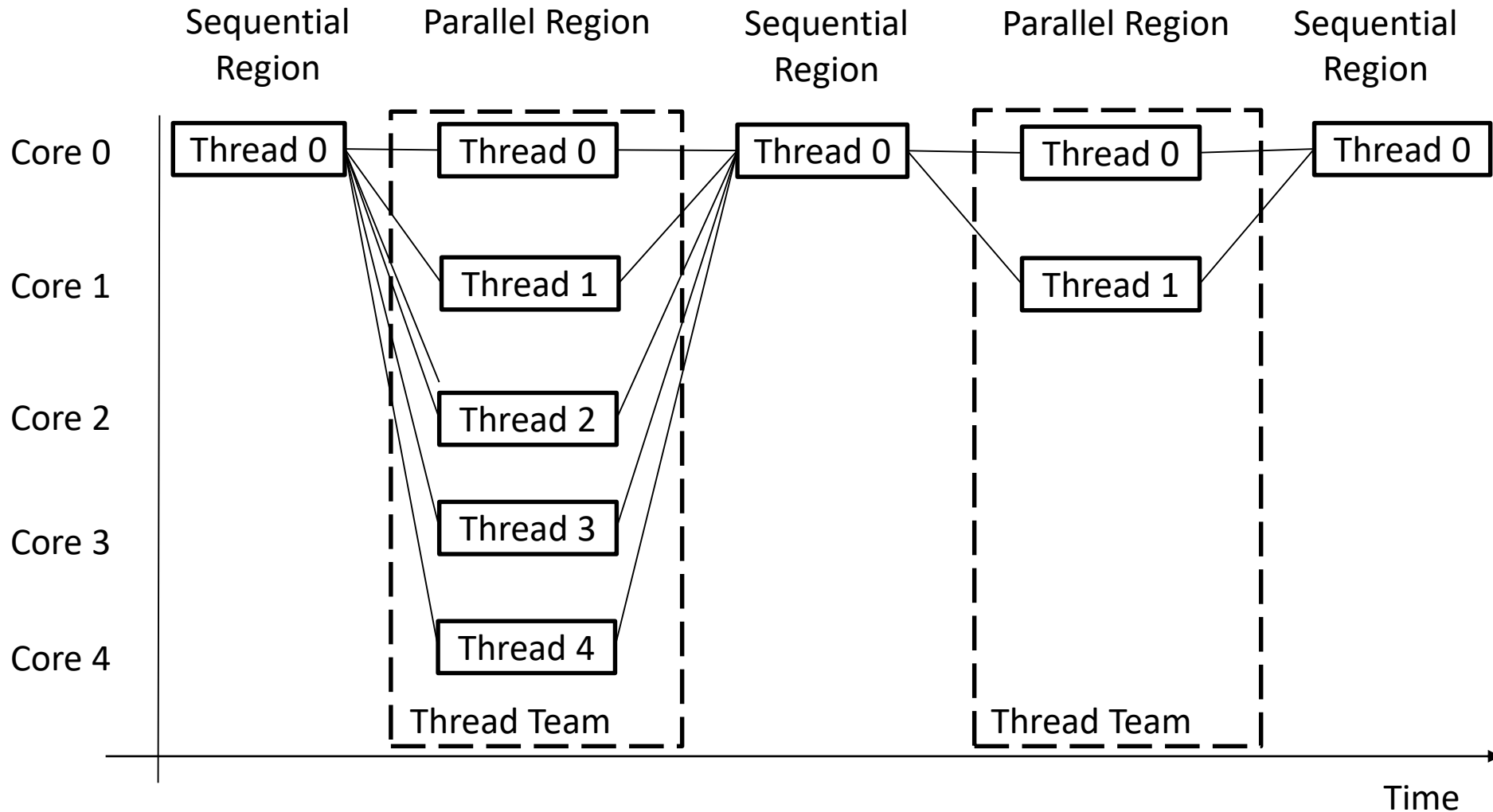
Agenda

- Hybrid Parallel Architectures
- Intel HPC Architectures
- OpenMP
- Profiling with Intel Advisor (Threading Workflow)
- Critical Sections
- Offload
- N-Body Simulation

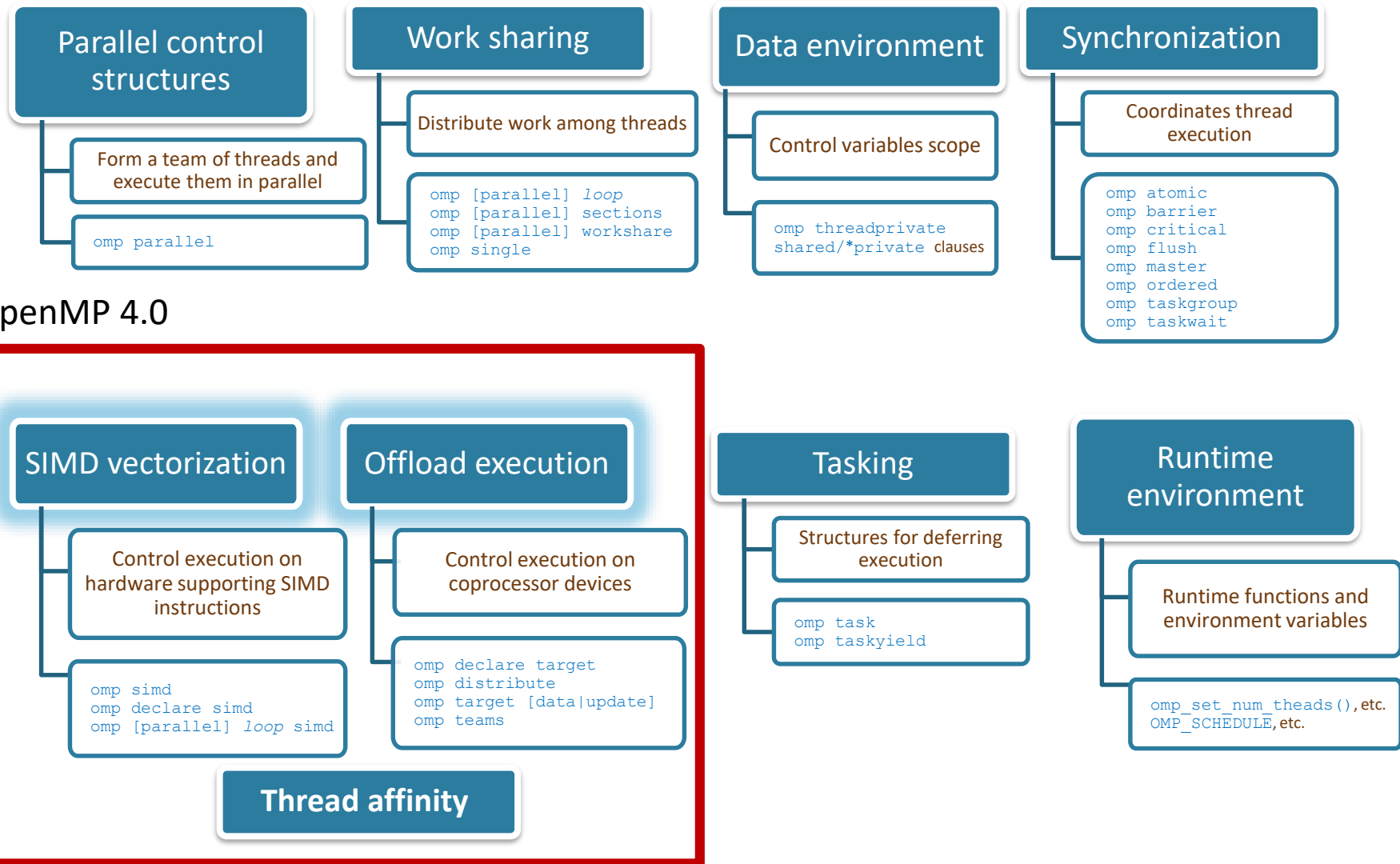
OpenMP

- OpenMP is an acronym for Open Multi-Processing
- An Application Programming Interface (API) for developing parallel programs in shared memory architectures
- Three primary components of the API are:
 - Compiler Directives
 - Runtime Library Routines
 - Environment Variables
- De facto standard - specified for C / C++ and FORTRAN
- <http://www.openmp.org/>
 - Specification, examples, tutorials and documentation

OpenMP



OpenMP - Core elements



OpenMP Sample Program

```
N=25;  
#pragma omp parallel for  
for (i=0; i<N; i++)  
    a[i] = a[i] + b;
```

	Thread 0					Thread 1					Thread 2					Thread 3					Thread 4				
i=	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24

OpenMP Sample Program

```
#include <stdio.h>

int main(){
    int r, c, i, j, *a , *b , *sum;
    char hn[600];

    #pragma omp parallel
    {
        gethostname(hn,600);
        printf("hostname %s\n",hn);
    }

    r=40000;
    c=40000;

    a = (int*)malloc(r*c*sizeof(double));
    b = (int*)malloc(r*c*sizeof(double));
    sum = (int*)malloc(r*c*sizeof(double));

    #pragma omp parallel for
    for(i=0; i<r; ++i)
        for(j=0; j<c; ++j) {
            a[i*r + j]=i+j;
            b[i*r + j]=i-j;
        }

    #pragma omp parallel for
    for(i=0;i<r;++i)
        for(j=0;j<c;++j)
            sum[i*r+j] = a[i*r+j] + b[i*r+j];

    free(a);
    free(b);
    free(sum);

    return 0;
}
```

Compiling and running an OpenMP application

#Build the application for Multicore Architecture (Xeon)

```
icc <source-code> -o <omp_binary> -fopenmp
```

#Build the application for the ManyCore Architecture (Xeon Phi)

```
icc <source-code> -o <omp_binary>.mic -fopenmp -mmic
```

#Launch the application on host

```
./omp_binary
```

#Launch the application on the device from host

```
micnativeloadex ./omp_binary.mic -e  
"LD_LIBRARY_PATH=/opt/intel/lib/mic/"
```

Compiling and running an OpenMP application

```
export OMP_NUM_THREADS=10  
./OMP-hello
```

```
hello from hostname phi02.ncc.unesp.br  
hello from hostname phi02.ncc.unesp.br  
hello from hostname phi02.ncc.unesp.br  
hello from hostname phi02.ncc.unesp.br  
hello from hostname phi02.ncc.unesp.br  
hello from hostname phi02.ncc.unesp.br  
hello from hostname phi02.ncc.unesp.br  
hello from hostname phi02.ncc.unesp.br  
hello from hostname phi02.ncc.unesp.br  
hello from hostname phi02.ncc.unesp.br  
Launch the application on the  
Coprocesor from host
```

```
micnativeloadex ./OMP-hello.mic -e  
"OMP_NUM_THREADS=10  
LD_LIBRARY_PATH=/opt/intel/lib/mic/"
```

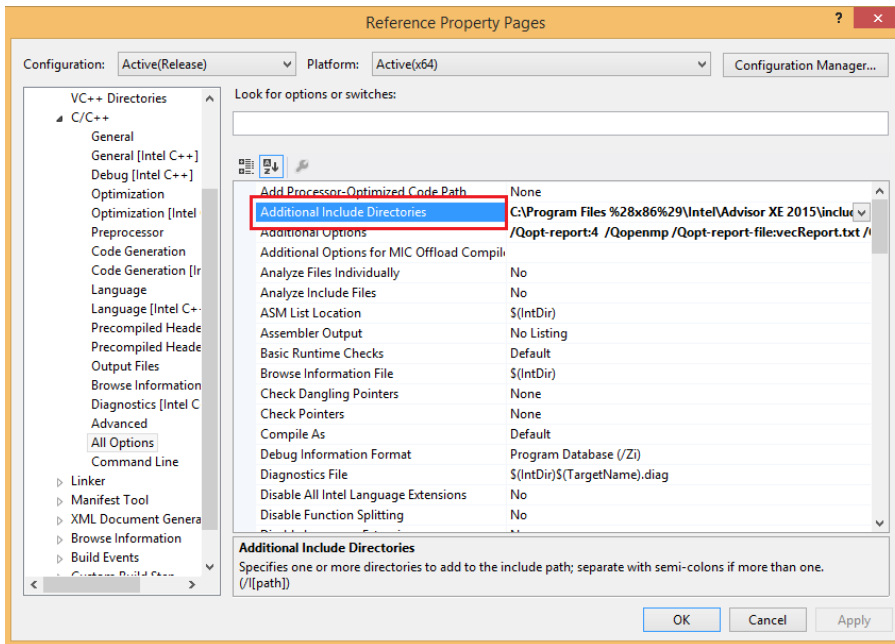
```
hello from hostname phi02-mic0.ncc.unesp.br  
hello from hostname phi02-mic0.ncc.unesp.br  
hello from hostname phi02-mic0.ncc.unesp.br  
hello from hostname phi02-mic0.ncc.unesp.br  
hello from hostname phi02-mic0.ncc.unesp.br  
hello from hostname phi02-mic0.ncc.unesp.br  
hello from hostname phi02-mic0.ncc.unesp.br  
hello from hostname phi02-mic0.ncc.unesp.br  
hello from hostname phi02-mic0.ncc.unesp.br  
hello from hostname phi02-mic0.ncc.unesp.br  
hello from hostname phi02-mic0.ncc.unesp.br  
sum of vector elements: 5789.473684
```

Agenda

- Hybrid Parallel Architectures
- Intel HPC Architectures
- OpenMP
- Profiling with Intel Advisor (Threading Workflow)
- Critical Sections
- Offload
- N-Body Simulation

Identifying Parallelization Opportunities

- Intel Advisor steps:
 - 1º - Include headers
 - `#include "advisor-annotate.h"`
 - 2º - add include reference ; link library



Linux – compiling / link with Advisor

`icpc -O2 -openmp`

`02_ReferenceVersion.cpp`

`-o 02_ReferenceVersion`

`-I/opt/intel/advisor/include/`

`-L/opt/intel/advisor/lib64/`

Identifying Parallelization Opportunities

- Intel Advisor Analysis:
 - Survey
 - ❑ Vectorization of loops: detailed information about vectorization;
 - ❑ Total Time: elapsed time in each loop considering the time involved in internal loops;
 - ❑ Self Time: elapsed time in each loop without internal loops;
 - Suitability
 - ❑ Speedup gains obtained parallelizing annotated loops;

Intel Advisor - Survey Data

The screenshot shows the Intel Advisor XE 2016 application window. The title bar indicates the path: `/home/silvio/intel/advixe/projects/TP - Intel Advisor`. The menu bar includes **File**, **View**, and **Help**. The toolbar contains various icons for file operations and analysis. The main workspace is titled **VECTORIZATION WORKFLOW** and displays a tree view of the workflow steps:

- 1. Survey Target**: Explore where to add efficient vectorization and/or threading. The **Collect** button is highlighted with a red circle.
- 1.1 Find Trip Counts**: Find how many iterations are executed.
- 2.1 Check Dependencies**: Identify and explore loop-carried dependencies for marked loops. Fix the reported problems.
- 2.2 Check Memory Access Patterns**: Identify and explore complex memory accesses for marked loops. Fix the reported problems.

At the bottom, there is a section for switching between Vectorization and Threading workflows, with a **Threading Workflow** button.

On the right side, the **Survey Report** tab is active, showing a **No Data** warning. The text states: "To collect data about your application's performance, compile your application with Release build settings and run [Survey](#) analysis."

Collect Survey Data

Intel Advisor - Survey Data

Function Call Sites and Loops	🔥	Vector Issues	Self Time▼	Total Time	Type	Why No Vectorization?
🔍 [loop in multiply3 at multiply.c:228]		💡 2 Assume...	0.170s	0.170s	Scalar	🚫 vector dependence prevents vectorization
🔍 [loop in _libc_csu_init]			0.000s	0.000s	Scalar	
🔍 [loop in _INTERNAL_16_offload_host_cpp_ad92...]			0.000s	0.000s	Scalar	
🔍 [loop in func@0x5b810]		💡 2 Data typ...	0.000s	0.000s	Scalar	
🔍 [loop in main at matrix.c:144]		💡 2 Data typ...	0.000s	0.000s	Scalar	🚫 inner loop was already vectorized
🔍 [loop in multiply3 at multiply.c:227]		💡 2 Assume...	0.000s	0.170s	Scalar	🚫 vector dependence prevents vectorization
🔍 [loop in multiply3 at multiply.c:226]		💡 2 Assume...	0.000s	0.170s	Scalar	🚫 vector dependence prevents vectorization
⊕ [loop in main at matrix.c:144]		💡 1 Data typ...	0.000s	0.000s	Vectorized (B...	

Source	Top Down	Loop Analytics	Loop Assembly	💡 Recommendations	🚫 Compiler Diagnostic Details
--------	----------	----------------	---------------	-------------------	-------------------------------

File: multiply.c:228 multiply3

Line	Source
218	void multiply3(int msize, int tid, int numt, TYPE a[][NUM], TYPE b[][NUM], TYPE c[][NUM], TYPE t[][NUM])
219	{
220	
221	#pragma omp target device(0) map(a[0:NUM][0:NUM]) \
222	//map(b[0:NUM][0:NUM]) map(c[0:NUM][0:NUM])
223	//{
224	int i,j,k;
225	// #pragma omp parallel for collapse (2) //num threads(60)
226	for(i=0; i<msize; i++) {
	🔍 [loop in multiply3 at multiply.c:226]
	Scalar loop. Not vectorized: vector dependence prevents vectorization
	No loop transformations applied
227	for(k=0; k<msize; k++) {
	🔍 [loop in multiply3 at multiply.c:227]
	Scalar loop. Not vectorized: vector dependence prevents vectorization
	Remainder loop
228	for(j=0; j<msize; j++) {
	🔍 [loop in multiply3 at multiply.c:228]
	Scalar loop. Not vectorized: vector dependence prevents vectorization
	Loop was unrolled by 2
229	c[i][j] = c[i][j] + a[i][k] * b[k][j];
230	}
231	}
232	}
233	//}
234	}
235	

Matrix Multiplication

```
for(i=0; i<msize; i++) {  
    for(j=0; j<msize; j++) {  
        for(k=0; k<msize; k++) {  
            c[i][j] = c[i][j] + a[i][k] * b[k][j];  
        }  
    }  
}
```

Intel Advisor – Check Suitability

- Inserting advisor **Annotations key words** for Check Suitability:
 - **ANNOTATE_SITE_BEGIN(id)**: before beginning of loop;
 - **ANNOTATE_ITERATION_TASK(id)**: first line inside the loop;
 - **ANNOTATE_SITE_END()**: after end of loop;

```
ANNOTATE_SITE_BEGIN( MySite1 );  
for(i=0; i<msize; i++) {  
    ANNOTATE_ITERATION_TASK( MyTask1 );  
    for(k=0; k<msize; k++)  
        for(j=0; j<msize; j++) {  
            c[i][j] = c[i][j] + a[i][k] * b[k][j];  
        }  
    ANNOTATE_SITE_END();  
}
```

Intel Advisor – Check Suitability

The screenshot displays the Intel Advisor XE 2016 application window. The title bar shows the path `/home/silvio/intel/advixe/projects/TP - Intel Advisor`. The menu bar includes **File**, **View**, and **Help**. The toolbar contains various icons, with the **Suitability Report** icon (a document with a checkmark) circled in red and pointed to by a blue arrow. Below the toolbar, the **THREADING WORKFLOW** section is visible, containing four steps: **1. Survey Target**, **1.1 Find Trip Counts**, **2. Annotate Sources**, and **3. Check Suitability**. The **Check Suitability** step is currently selected, showing a description and a **Collect** button. To the right, the **Where are the detected annotations?** section displays a table of annotations. The table has three columns: **Annotation**, **Source Location**, and **Annotation Label**. The table lists several annotations, including **Site**, **Site End**, and **Task**, with their respective source locations and labels. The **Suitability Report** icon in the toolbar is circled in red, and a blue arrow points from it to the **Suitability Data** text on the right.

Where are the detected annotations?

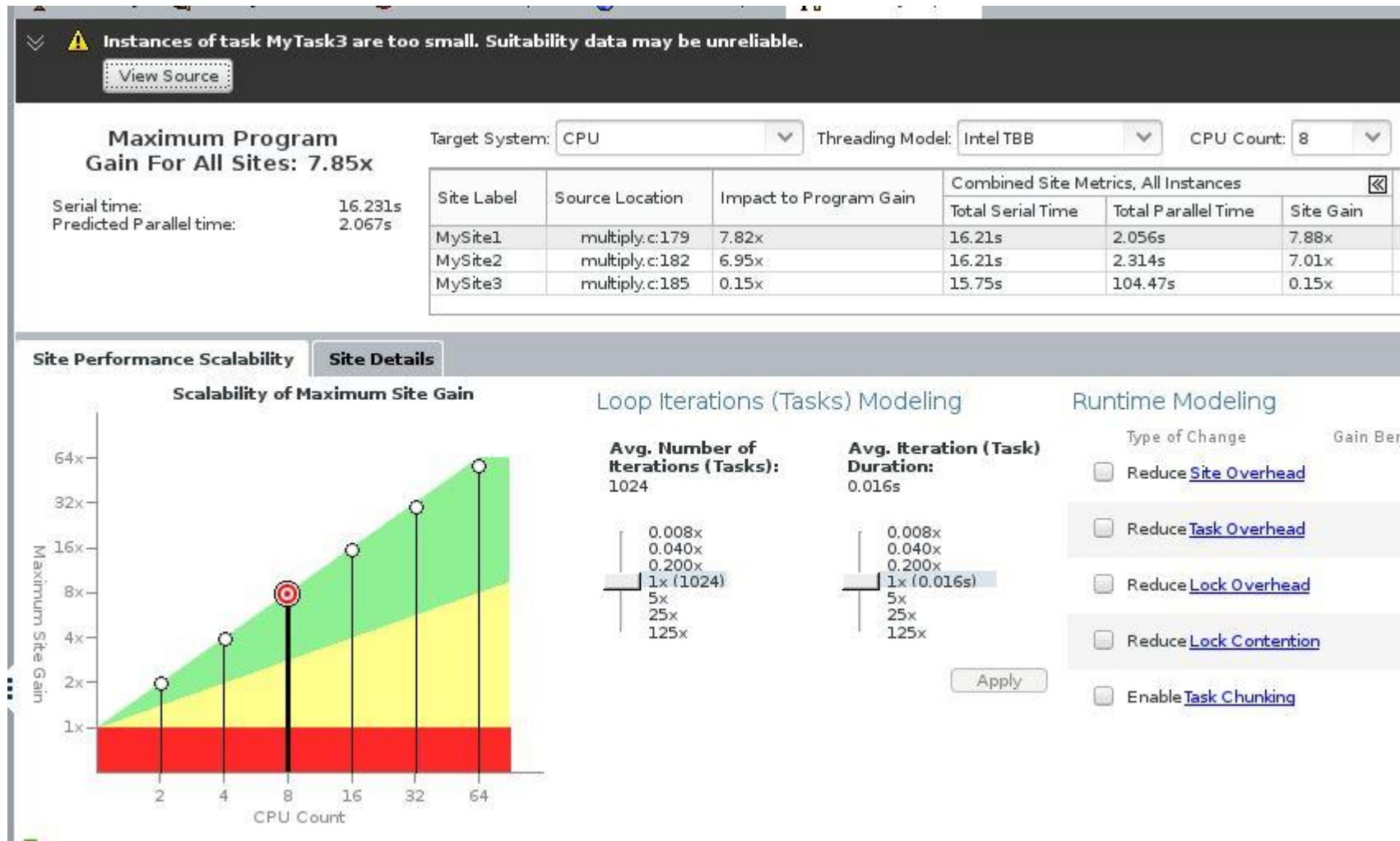
Elapsed time: 40.77s **Vectorized** **Not Vectorized** FILTER: All Modules All Sources

Summary Survey Report Refinement Reports Annotation Report Suitability Report

Annotation	Source Location	Annotation Label
+ Site	Transpose.cc:16	MySite1
+ Site	Transpose.cc:19	MySite2
+ Site End	Transpose.cc:26	-
+ Site End	Transpose.cc:28	-
+ Task	Transpose.cc:18	MyTask1
+ Task	Transpose.cc:21	MyTask2
+ Intel Advisor XE annotations definition file	Transpose.cc:13	advisor-annotate.h

Suitability Data

Intel Advisor – Check Suitability



Agenda

- Hybrid Parallel Architectures
- Intel HPC Architectures
- OpenMP
- Profiling with Intel Advisor (Threading Workflow)
- Critical Sections
- Offload
- N-Body Simulation

Critical section

- In concurrent programming, concurrent accesses to shared resources can lead to unexpected or erroneous behavior;
- Regions of code where the shared resource is accessed, has to be protected against concurrent access and is known as critical section;
- Challenge:
 - Identify critical sections;
 - Impose synchronization without loss of performance.

Intel Inspector

- Intel Inspector is a debugger tool that performs dynamic analysis. It is capable of identifying the following errors:

- **Memory Errors**

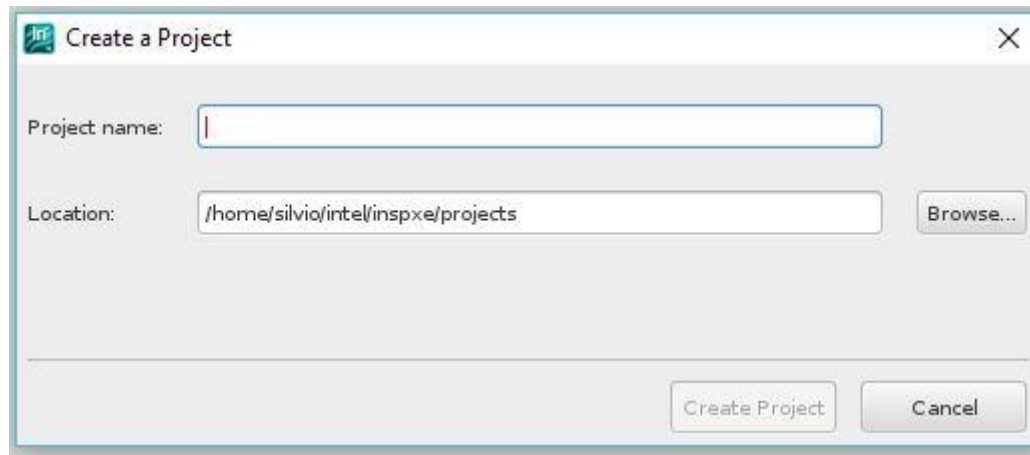
- ☐ Memory leaks;
- ☐ Memory corruption;
- ☐ Allocation / de-allocation API mismatches
- ☐ Inconsistent memory API usage;
- ☐ Illegal memory access;
- ☐ Uninitialized memory read;

- **Threading Errors**

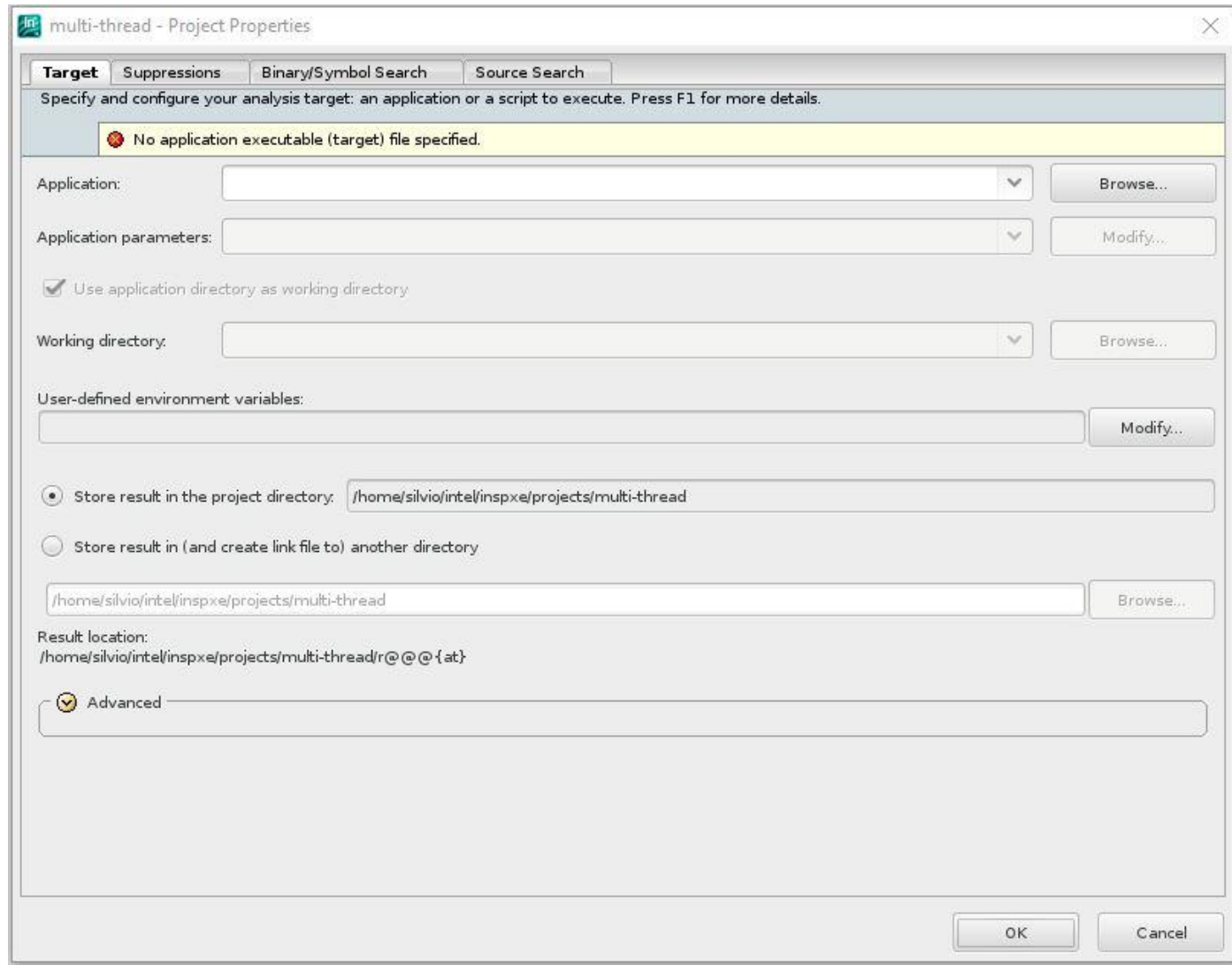
- ☐ Data races:
 - Heap races;
 - Stack races;
- ☐ Deadlocks;



Intel Inspector




Intel Inspector



Intel Inspector

r008ti3


 Collecting Data...

Target

Analysis Type

Collection Log

Summary

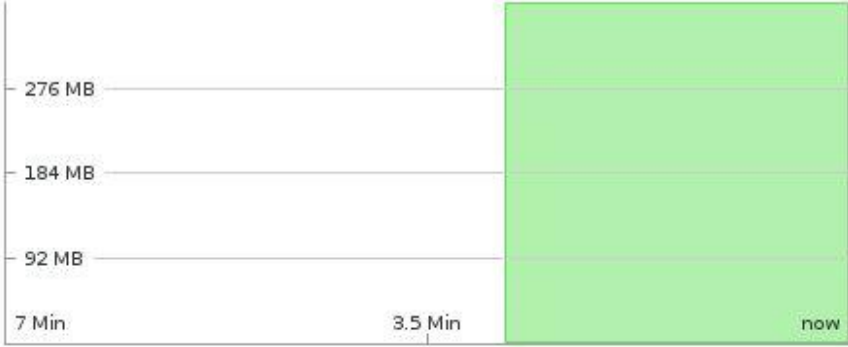


Please wait while result data is collected...

Estimated collection time may be 2 to 320 times normal target execution time. Data set size and workload have a direct impact on target execution time and analysis speed. If possible, choose small, representative data sets with runs in the seconds time range.

Memory Used by Analysis Tool and Target Application

Current memory usage (updated every second): 368 MB





Time	Memory Usage (MB)
7 Min	~368
3.5 Min	~368
now	~368


Analysis Progress and Thread Activity


Elapsed time since collection start: 00:03:06


 Show details


 Start


 Stop

 Close

 Reset Growth Tracking

 Measure Growth

 Reset Leak Tracking

 Find Leaks

Intel Inspector

r002ti3

Locate Deadlocks and Data Races

Target Analysis Type Collection Log Summary

Problems

ID	Type	Sources	Modules	State
P1	Data race	sumOMP.cpp	sum	New
	Data race	sumOMP.cpp:17	sum	New
	Data race	sumOMP.cpp:17	sum	New

1 of 6 Code Locations: Data race

Description	Source	Function	Module
Write	sumOMP.cpp:17	sum	sum
<pre>15 long unsigned int div = remainder / i; 16 long unsigned int mod = remainder % i; 17 digits[digit] += div; 18 remainder = mod * 10; 19 }</pre>			
Write	sumOMP.cpp:17	sum	sum
<pre>15 long unsigned int div = remainder / i; 16 long unsigned int mod = remainder % i; 17 digits[digit] += div; 18 remainder = mod * 10; 19 }</pre>			

sum!sum - sumOMP.cpp: 17

Tachyon

- **Tachyon:** a parallel/multiprocessor ray tracing software.
- Variable `col` is declared as global, but used by several threads;

The screenshot shows the Visual Studio IDE with the 'Locate Deadlocks and Data Races' tool open. The 'Problems' window displays a list of data race errors. The first error, labeled 'P1', is a data race in the file `find_and_fix_threading_errors.cpp` at line 105. The error is categorized as 'Data race' and 'Error'. The 'Code Locations: Data race' window shows the specific code locations where the data race occurs. It lists two threads: one reading the variable `col` at line 149 and another writing to it at line 105. The code snippets show that `col` is a global variable declared at line 80, and the threads are executing in parallel tasks.

ID	Type	Sources	Modules	State
P1	Data race	find_and_fix_threading_errors.cpp	tachyon.find_and_fix_threading_errors	New
	Data race	find_and_fix_threading_errors.cpp:105	tachyon.find_and_fix_threading_errors	New
	Data race	find_and_fix_threading_errors.cpp:105; find_a...	tachyon.find_and_fix_threading_errors	New
	Data race	find_and_fix_threading_errors.cpp:105; find_a...	tachyon.find_and_fix_threading_errors	New
	Data race	find_and_fix_threading_errors.cpp:105; find_a...	tachyon.find_and_fix_threading_errors	New
P2	Data race	xvideo.cpp	tachyon.find_and_fix_threading_errors	New
	Data race	xvideo.cpp:264	tachyon.find_and_fix_threading_errors	New
	Data race	xvideo.cpp:264	tachyon.find_and_fix_threading_errors	New

Description	Source	Function	Module
Read	find_and_fix_threading_errors.cpp:149	render_one_pixel	tachyon.find_and_fix_threading_errors
147	else if (G < 0) G = 0;		tachyon.find_and_fix_threading_errors!render_one_pixel - find_and_fix_threading_errors.cpp:149
148	B=(int) (col.b*255); //Threading Error: see comments near line 104		tachyon.find_and_fix_threading_errors!operator() - find_and_fix_threading_errors.cpp:202
149	if (B > 255) B = 255;		tachyon.find_and_fix_threading_errors!run_body - parallel_for.h:102
150	else if (B < 0) B = 0;		tachyon.find_and_fix_threading_errors!execute<tbb::interface6::internal::start_for<tbb::blocked_range2d<int, int>, parallel_task,
151			tachyon.find_and_fix_threading_errors!execute - parallel_for.h:108
Write	find_and_fix_threading_errors.cpp:105	render_one_pixel	tachyon.find_and_fix_threading_errors
103	primary.scene = &scene;		tachyon.find_and_fix_threading_errors!render_one_pixel - find_and_fix_threading_errors.cpp:105
104	col=trace(&primary); //Threading Error: col is a global variable declared at line 80		tachyon.find_and_fix_threading_errors!operator() - find_and_fix_threading_errors.cpp:202
105	//2 ways to fix this threading error		tachyon.find_and_fix_threading_errors!run_body - parallel_for.h:102
106	// 1) Make col a local variable		tachyon.find_and_fix_threading_errors!execute<tbb::interface6::internal::start_for<tbb::blocked_range2d<int, int>, parallel_task,
107			tachyon.find_and_fix_threading_errors!execute - parallel_for.h:108

Tachyon

- Solution with synchronization;
- Eliminating concurrency
 - Variable is not used outside function;
 - changing variable from global to local eliminates Synchronization;

Agenda

- Hybrid Parallel Architectures
- Intel HPC Architectures
- OpenMP
- Profiling with Intel Advisor (Threading Workflow)
- Critical Sections
- Offload
- N-Body Simulation

Pragma omp target

- Transfer control [and data] from host to device
- Syntax
 - `#pragma omp target [data] [clause[[,] clause],...]
structured-block`
- Clauses
 - `device(scalar-integer-expression) :`
 - ❑ device to offload code;
 - `map(alloc | to | from | tofrom: list) :`
 - ❑ map variables to device;
 - `if(scalar-expr) :`
 - ❑ test an expression before offload:
 - o True executes on device;
 - o False executes on host;
 - `Nowait`
 - ❑ Execute the data transfer defined in map asynchronously;

Pragma omp target

- Map clauses:
 - alloc : allocate memory on device;
 - to : transfer a variable from host to device;
 - from : transfer a variable from device to host;
 - tofrom :
 - ❑ transfer a variable from host to device before start execution;
 - ❑ transfer a variable from device to host after finish execution;



Offloading - omp target

```
Int main() {  
  Printf("begin");  
  int N=25;  
  int b =2;  
  int l = 0;
```

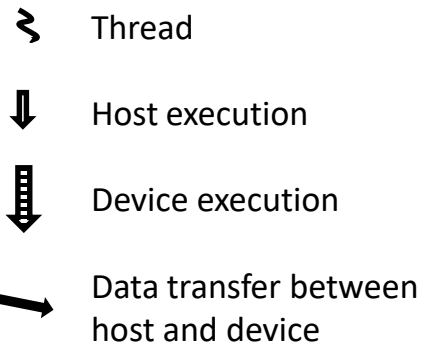
*Offload:
Copy variable:
N,b,l and **a** to device*

```
#pragma omp target map(N,b,l,a)  
{  
  for (i=0; i<N; i++) a[i] = 2;  
  for (i=0; i<N; i++) a[i] = a[i] + b;  
}
```

```
for (i=0; i<N; i++)  
  printf("%d",a[i]);  
...  
return(0);  
}
```

Host

Device



Pragma omp target example

```
#pragma omp target device(0) map(a[0:NUM][0:NUM])  
map(b[0:NUM][0:NUM]) map(c[0:NUM][0:NUM])  
{  
    #pragma omp parallel for collapse (2)  
    for(i=0; i<msize; i++) {  
        for(k=0; k<msize; k++) {  
            for(j=0; j<msize; j++) {  
                c[i][j] = c[i][j] + a[i][k] * b[k][j];  
            }  
        }  
    }  
}
```

Agenda

- Hybrid Parallel Architectures
- Intel HPC Architectures
- OpenMP
- Profiling with Intel Advisor (Threading Workflow)
- Critical Sections
- Offload
- N-Body Simulation

N-Body Simulation

- An N-body simulation [1] aims to approximate the motion of particles that interact with each other according to some physical force;
- Used to study the movement of bodies such as satellites, planets, stars, galaxies, etc., which interact with each other according to the gravitational force;
- Newton's second law of motion can be used in a N-body simulation to define the bodies' movement.

[1] AARSETH, S. J. Gravitational n-body simulations. [S.l.]: Cambridge University Press, 2003. Cambridge Books Online.

N-Body Algorithm

- Bodies struct:
 - 3 matrix represents velocity (x,y and z)
 - 3 matrix represents position (x,y and z)
 - 1 matrix represent mass
- A loop calculate temporal steps:
 - At each temporal step new velocity and position are calculated to all bodies according to a function that implements Newton's second law of motion

N-Body - Parallel version (host only)

```
function Newton(step)
{
    #pragma omp for
    for each body[x] {
        #pragma omp simd
        for each body[y]
            calc force exerted from body[y] to body[x];
        calc new velocity of body[x]
    }
    #pragma omp simd
    for each body[x]
        calc new position of body[x]
}

Main() {
    for each temporal step
        Newton(step)
}
```

N-Body - Parallel version (Load balancing)

- The temporal step loop remains sequential
- The N-bodies are divided among host and devices to be executed using Newton
- OpenMP offload pragmas are used to
 - Newton function offloading to devices
 - Transfer data (bodies) between host and devices

N-Body - Parallel version (Load balancing)

```
function Newton(step, begin_body, end_body, deviceId)
{
    #pragma omp target device (deviceId) {
        #pragma omp for
        for each body[x] from subset(begin_body, end_body) {
            #pragma omp simd
            for each body[y] from subset(begin_body, end_body)
                calc force exerted from body[y] to body[x];
            calc new velocity of body[x]
        }
        #pragma omp simd
        for each body[x]
            calc new position of body[x]
    }
}
```

N-Body - Parallel version (Load balancing)

for each temporal step

Divide the amount of bodies among host and devices;

```
#pragma omp parallel  
{  
    #pragma omp target data device ( tid ) to(bodies[begin_body:  
end_body])  
    {  
        Newton(step, begin_body, end_body, deviceId)  
        #pragma omp target update device ( tid ) (from:bodies)  
        #pragma omp barrier  
        #pragma omp target data device ( tid )  
to(bodies[begin_body: end_body])  
    }  
}
```