



Introdução à Programação Multithreaded: explorando arquiteturas heterogêneas e vetorização com OpenMP 4.0

Silvio Luiz Stanzani , Raphael Mendes de Oliveira Cóbe , Rogério
Luiz Iope

NCC/UNESP

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

Agenda

- NCC Presentation
- Parallel architectures
- OpenMP
- Vectorization
- Offload
- Thread league
- N-body

Material

- **Source-code, Slides and Book Chapter:**
- <https://github.com/intel-unesp-mcp/talks-source-code/tree/master/OpenMP4>

UNESP Center for Scientific Computing

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

UNESP Center for Scientific Computing



SPRACE - CMS Tier2 Facility

- 144 worker nodes
 - Physical/Logical CPUs: 288/1088
 - HEPSpec06: 13698
- 02 head nodes
- 04 auxiliary servers
- 12 storage servers
 - 1 PB (raw), 0.85 PB (effective): 81% usage
- CSC Network
 - LAN: 1 Gbps & 10 Gbps
 - MAN: 10 Gbps & 100 Gbps
 - WAN: 4x10Gbps & 100 Gbps

Intel[®] Partnership

- IPCC (Intel Parallel Computing Center)
 - Vectorization of Geant (**GE**ometry **ANd** Tracking)
- Intel Modern Code
 - Workshops and Tutorials
 - ❑ High Performance Computing (HPC)
 - ❑ Big Data
 - HPC Consultancy
 - <http://modern-code.ncc.unesp.br/>



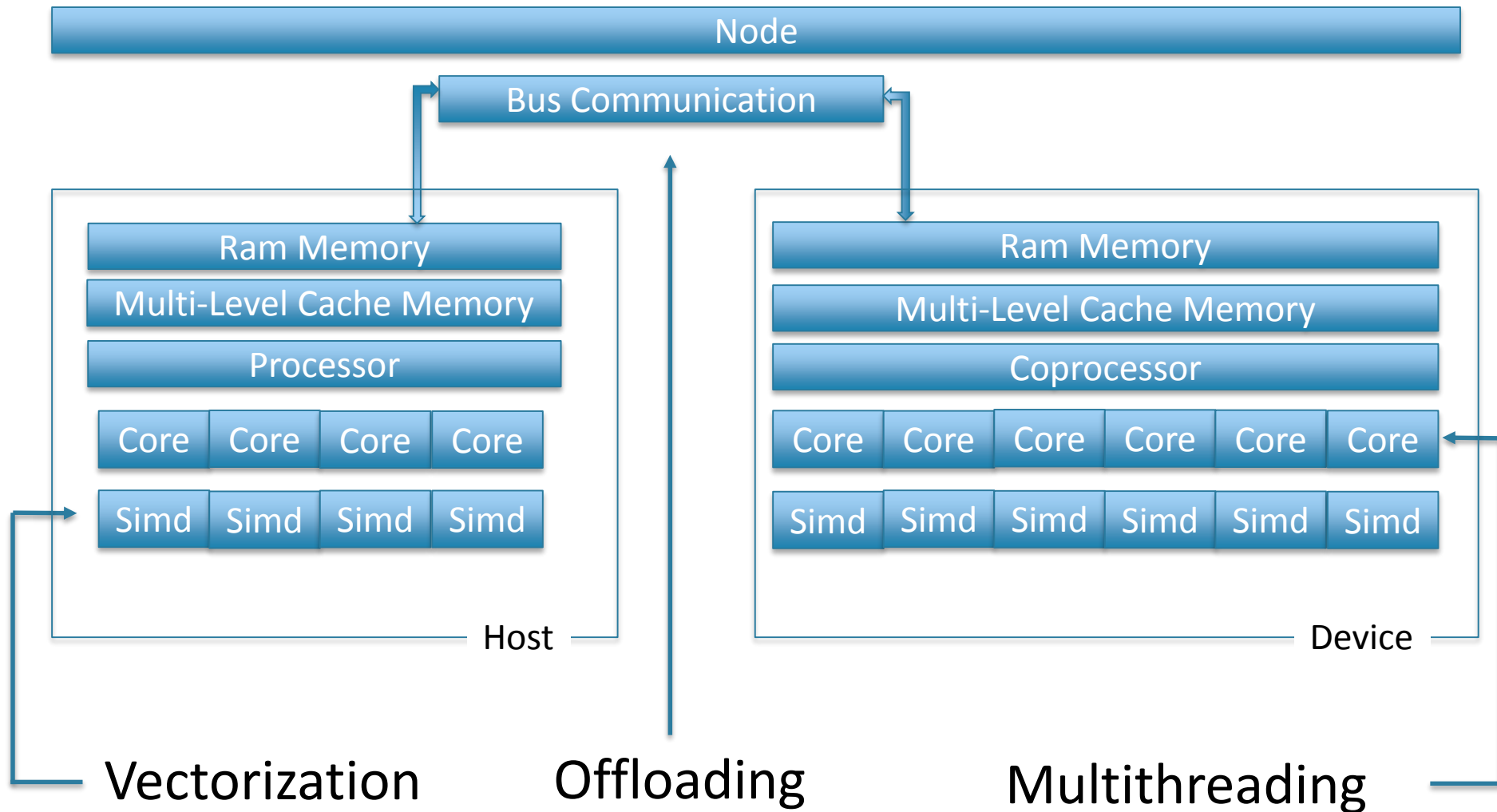
Agenda

- NCC Presentation 5 min
- Parallel architectures 12 min
- OpenMP 15 min
- Vectorization 25 min
- Offload 25 min
- Thread league 20 min
- N-body 20 min

Parallel Architectures

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

Hybrid Parallel Architectures



Hybrid Parallel Architectures

- Exploring parallelism in hybrid parallel architectures
 - Multithreading
 - Vectorization
 - ❑ Auto vectorization;
 - ❑ Semi-auto vectorization;
 - ❑ Explicit vectorization;
 - Offloading
 - ❑ Offloading code to device;
- OpenMP 4.0;
 - Supports vectorization and offloading on hybrid parallel architectures;

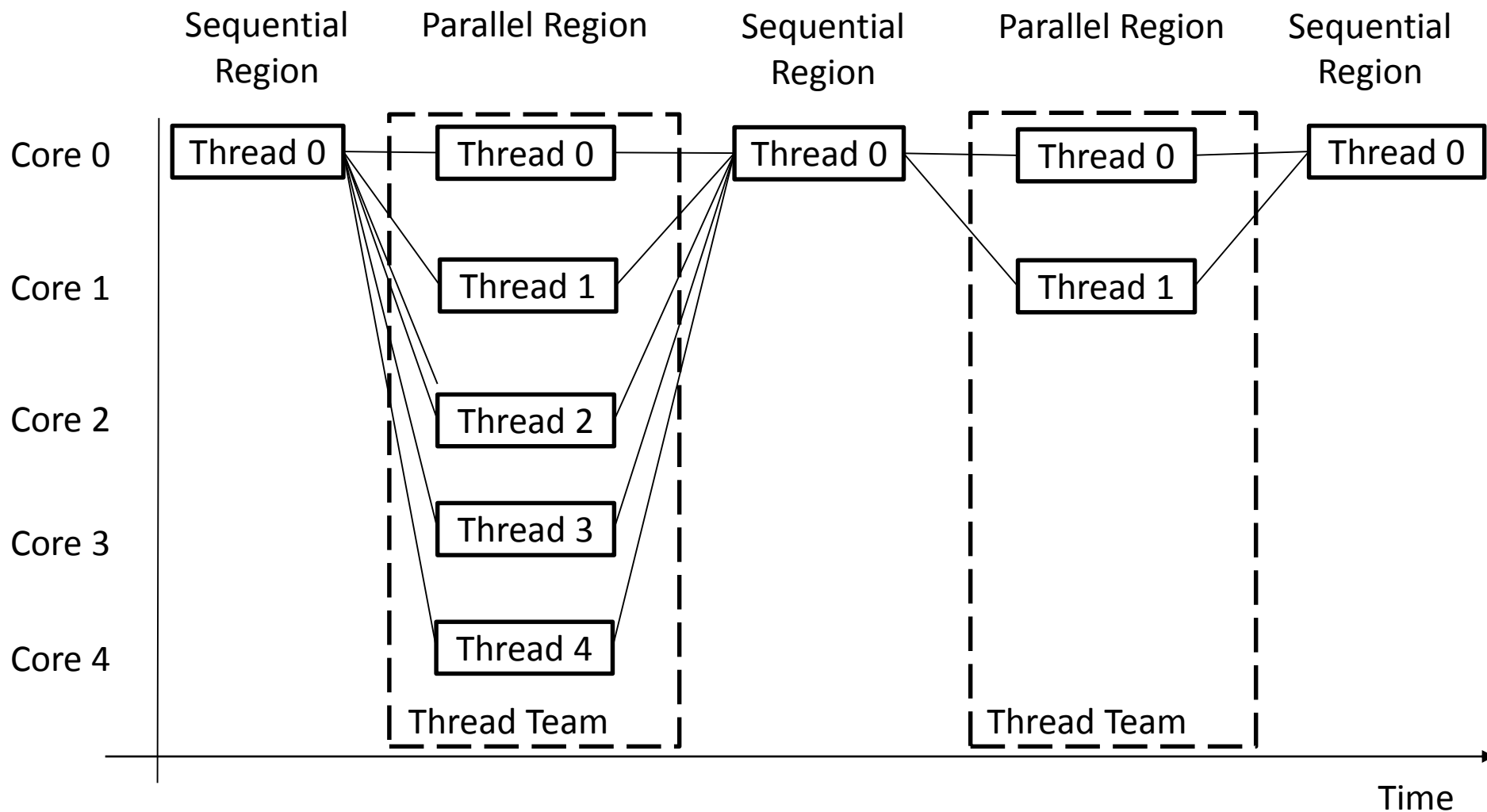
Agenda

- NCC Presentation
- Hybrid Parallel Architectures
- OpenMP
- Vectorization
- Offload
- Thread League
- N-Body

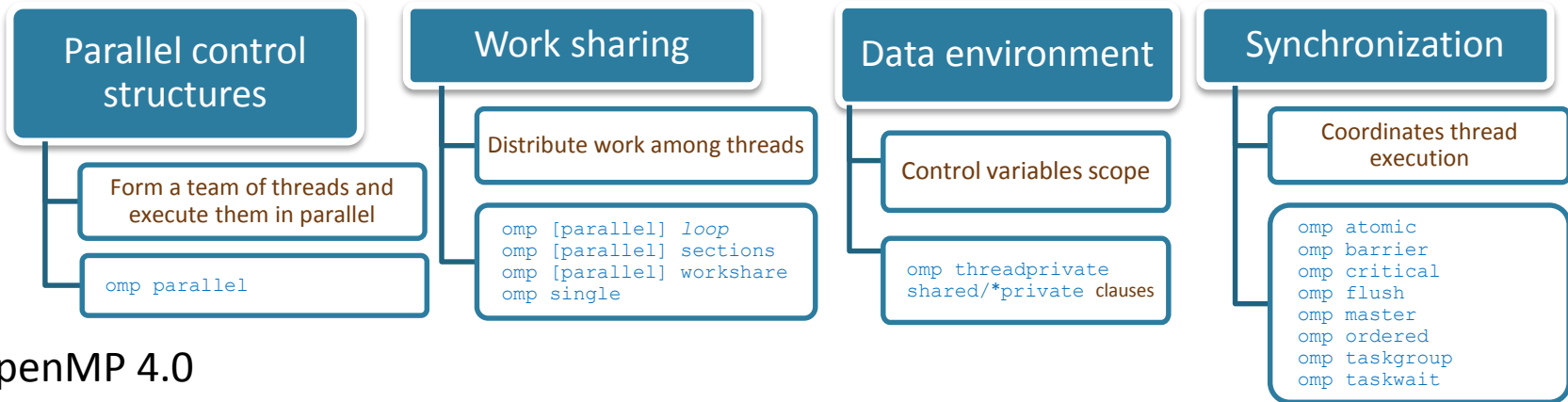
OpenMP

- 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

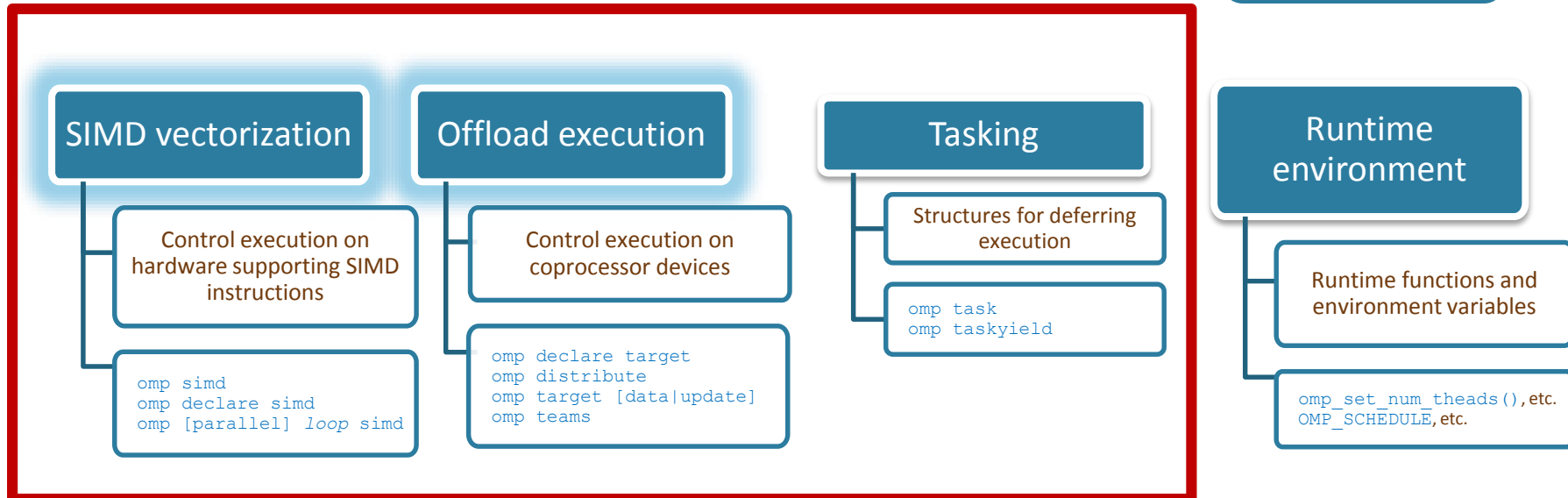
OpenMP



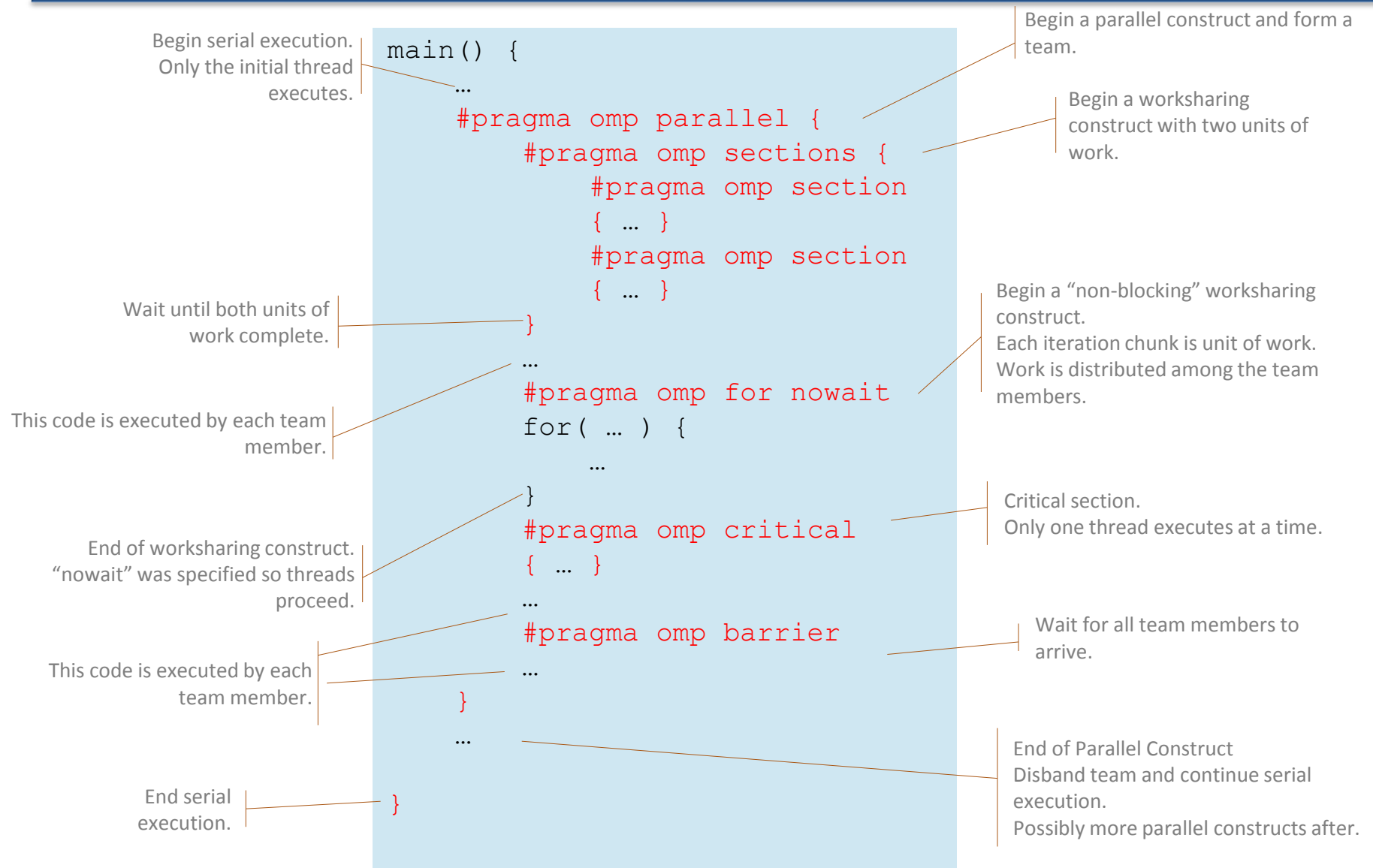
OpenMP: the core elements



OpenMP 4.0



OpenMP parallel processing model



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>
#include <stdlib.h>
#include <omp.h>
#include <unistd.h>

int main() {
    int thid; char hn[600], i;
    double res, p[100];

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

```
    res = 0;

    #pragma omp for
    for ( i = 0 ; i < 100 ; i++ ) {
        p[i] = i/0.855;
    }

    #pragma omp for
    for ( i = 0 ; i < 100 ; i++ ) {
        res = res + p[i];
    }

    printf("sum: %f", res);
}
```

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/"
```

Agenda

- NCC Presentation
- Parallel architectures
- OpenMP
- Vectorization
- Offload
- Thread league
- N-body

Intel Advisor

- Evaluate multi-threading parallelization
- Intel® Advisor XE
 - ❑ Performance modeling using several frameworks for multi-threading in processors and co-processors:
 - OpenMP, Intel® Cilk™ Plus, Intel® Threading Building Blocks
 - C, C++, Fortran (OpenMP only) e C# (Microsoft TPL)
 - ❑ Identify parallel opportunities
 - ❑ Scalability prediction: amount of threads/performance gains
 - ❑ Correctness (deadlocks, race condition)



Intel Advisor

- Intel Advisor Analysis:
 - Survey
 - ❑ Vectorization of loops: detailed information about vectorization;
 - ❑ Total Time: elapsed time on each loop considering the time involved in internal loops;
 - ❑ Self Time: elapsed time on each loop disconsidering the time involved in internal loops;
 - Suitability
 - ❑ Speedup gains that may be 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 threading.
 - Collect** (highlighted with a red circle)
 - Command Line**
- 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.
 - Collect**
 - Command Line**
- 2.2 Check Memory Access Patterns**
 - Identify and explore complex memory accesses for marked loops. Fix the reported problems.
 - Collect**
 - Command Line**

Below the workflow steps, there are status indicators: **-- Nothing to analyze --** for each step.

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

On the right side of the interface, a banner reads **Where should I add vectorization and/or threading parallelism?** with a camera icon. Below this, the **Survey Report** tab is selected, showing a **No Data** warning icon and the text: "To collect data about your application's performance, compile your application with Release build settings and run [Survey](#) analysis."

A large arrow points from the **Collect** button in the **Survey Target** step to the text **Collect Survey Data** at the bottom of the slide.

Vectorization

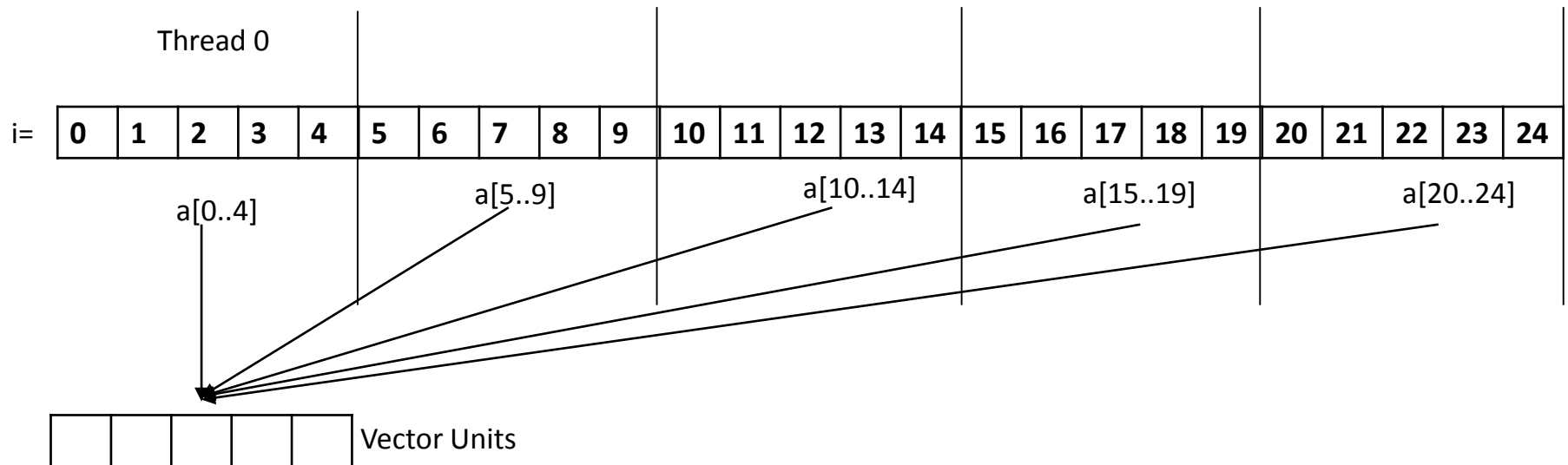
- Instructs the compiler to enforce vectorization of loops
(Semi-auto vectorization)
- `omp simd`
 - marks a loop to be vectorized by the compiler
- `omp declare simd`
 - marks a function that can be called from a SIMD loop to be vectorized by the compiler
- `omp parallel for simd`
 - marks a loop for thread work-sharing as well as SIMDing

OMP SIMD

- Vectorize a loop nest
 - Cut loop into chunks that fit a SIMD vector register
 - No parallelization of the loop body
- Syntax

```
#pragma omp simd [clause[[, clause],...]
for-loops
```

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



Data Sharing Clauses

- Specifies that each thread has its own instance of a variable:
 - `private(var-list)`: uninitialized vectors for variables in *var-list*
 - `firstprivate(var-list)`: Initialized vectors for variables in *var-list*
 - `lastprivate(var-list)`:
 - ❑ similar to private clause
 - ❑ Private copy of last iteration is copied to the original variable
 - `reduction(op:var-list)`: create private variables for *var-list* and apply reduction operator *op* at the end of the construct

SIMD Loop Clauses

- `safelen (length)`
 - Maximum number of iterations that can run concurrently without breaking a dependence
- `linear (list[:linear-step])`
 - The variable's value is in relationship with the iteration number
$$x_i = x_{\text{orig}} + i * \text{linear-step}$$
- `aligned (list[:alignment])`
 - Specifies that the list items have a given alignment
 - Default is alignment for the architecture
- `collapse (n)`
 - Groups two or more loops into a single loop

Pragma OMP simd Example

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

OMP SIMD – vectorization Report

Intel Advisor XE 2016

Where should I add vectorization and/or threading parallelism?

Elapsed time: 62.03s Vectorized Not Vectorized FILTER: All Modules All Sources

Summary Survey Report Refinement Reports Annotation Report

Some target modules do not contain debug information
Suggestion: enable debug information for relevant modules.

Loops	Vector Issues	Self Time	Total Time	Loop Type	Why No Vectorization?	Vectorized Loops	Gain Estim...	VL (Vector Length)	Traits	Data Types	
[loop in multiply3\$omp\$parallel_for@225 at multiply.c:229]	Assumed dependency present	1984.812s	1984.812s	Scalar	vector dependence: assumed de...						
[loop in __kmp_launch_thread at kmp_runtime.c:5900]		8.699s	1939.702s	Scalar							
[loop in multiply3\$omp\$parallel_for@225 at multiply.c:226]	Assumed dependency present	1.140s	1985.952s	Scalar	vector dependence: assumed de...				Divisions	Float64	
[loop in __libc_start_main]		0.000s	0.021s	Scalar							
[loop in __libc_start_main]		0.000s	56.039s	Scalar							
[loop in start_thread]		0.000s	1939.702s	Scalar							
[loop in _INTERNAL_16_offload_host_cpp_ad9271c5:___offload_init_library_once]		0.000s	0.021s	Scalar							
[loop in func@0x5b810]	Data type conversions present	0.000s	0.010s	Scalar					Type Conversions	Float64	
[loop in func@0x5b740]	System function call(s) present	0.000s	0.010s	Scalar							
[loop in func@0x5b740]		0.000s	0.010s	Scalar							
[loop in func@0x5b740]		0.000s	0.010s	Scalar							
[loop in func@0x54bf0]	System function call(s) present	0.000s	0.011s	Scalar							
[loop in func@0x54bf0]		0.000s	0.011s	Scalar							
[loop in main at matrix.c:144]	Data type conversions present	0.000s	0.619s	Scalar	inner loop was already vectorized				Type Conversions; ...	Float32; Float64; Ir	
[loop in [OpenMP worker] at z_Linux_util.c:786]		0.000s	1939.702s	Scalar							
[loop in main at matrix.c:144]	Data type conversions present	0.000s	0.619s	Vectorized (Body)		SSE2	100%	3.59x	2	Type Conversions	Float32; Float64; Ir

1. Survey Target
Explore where to add efficient vectorization and/or threading.
Collect Command Line

1.1 Find Trip Counts
Find how many iterations are executed.
Collect Command Line

Mark Loops for Deeper Analysis
Select loops in the Survey Report for Dependencies and/or Memory Access Patterns analysis.
-- There are no marked loops --

2.1 Check Dependencies
Identify and explore loop-carried dependencies for marked loops. Fix the reported problems.
Collect Command Line

2.2 Check Memory Access Patterns
Identify and explore complex memory accesses for marked loops. Fix the reported problems.
Collect Command Line

Switch between Vectorization and Threading workflows
Threading Workflow

Files: multiply.c:229 multiply3\$omp\$parallel_for@225

Line	Source	Total Time	%	Loop Time	%	Traits
214	}					
215	}					
216	}					
217	}					
218	void multiply3(int msize, int tidz, 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)	55.399s				
226	for(i=0; i<msize; i++) {	0.880s		1.985.950s		Divisions
227	for(k=0; k<msize; k++) {	0.030s				
228	//pragma omp simd					
229	for(j=0; j<msize; j++) {	181.923s		1.984.810s		
230	c[i][j] = c[i][j] + a[i][k] * b[k][j];	1,809.120s				
231	}					
232	}					
233	}					
234	//}					
235	}					
236						
237	void multiply4(int msize, int tidz, int numt, TYPE a[][NUM], TYPE b[][NUM], TYPE c[][NUM], TYPE t[][NUM])					
238	{					
239	//loop vectorization with pragma omp simd					
240						
241	int i,j,k;					
242	#pragma omp parallel for collapse (2) //num threads(60)					
243	for(i=0; i<msize; i++) {					

Selected (Total Time): 181.923s

SIMD Function Vectorization

- Declare one or more functions to be compiled for calls from a SIMD-parallel loop
- Syntax (C/C++):

```
#pragma omp declare simd [clause[,] clause],...  
[#pragma omp declare simd [clause[,] clause],...]  
[...]  
function-definition-or-declaration
```

SIMD Function Vectorization

- `simdlen` (*length*)
 - generate function to support a given vector length
- `uniform` (*argument-list*)
 - argument has a constant value between the iterations of a given loop
- `inbranch`
 - function always called from inside an if statement
- `notinbranch`
 - function never called from inside an if statement
- `linear` (*argument-list[:linear-step]*)
- `aligned` (*argument-list[:alignment]*)
- `reduction` (*operator:list*)

pragma OMP declare simd

```
#pragma omp declare simdlen (SIMD_LEN)
int FindPosition(double x) {
    return (int)(log(exp(x*steps)));
}
```

```
#pragma omp declare simd uniform (vals)
double Interpolate(double x, const point*
vals)
{
    int ind = FindPosition(x);
    ...

    return res;
}
```

```
int main ( int argc , char argv [] )
{
    ...
    #pragma omp simd
    for ( i=0; i <ARRAY_SIZE;++ i ) {
        dst[i] = Interpolate( src[i], vals ) ;
    }
    ...
}
```


OMP Declare Simd Vectorization Report

The screenshot displays the Intel Advisor XE 2016 interface. The top bar shows the project path: `/home/silvio/intel/advise/projects/interpolate - Intel Advisor`. The left sidebar contains a 'Project Navigator' with a tree view showing the project structure: `interpolate`, `e000`, `matriceval`, `nbody`, and `proftrager`. The main window is titled 'Where should I add vectorization and/or threading parallelism?' and shows a 'VECTORIZATION WORKFLOW' on the left. The workflow includes steps like '1. Survey Target', '1.1 Find Trip Counts', '2.1 Check Dependencies', and '2.2 Check Memory Access Patterns'. The main area displays a table of loops and their vectorization status. A warning message states: 'Higher instruction set architecture (ISA) available. Consider recompiling your application using a higher ISA.' Below this, a table lists loops and their characteristics.

Loops	Vector Issues	Self Time	Total Time	Loop Type	Why No Vectorization?	Vectorized Loops	Instruction Set Analysis	Advanced	Location						
						Vec...	Gal...	VL ...	Com...	Traits	Data...	Vector Widths	Instruction Sets		
loop in main at main.c:86	Assum...	0.240s	61.629s	Scalar	vector dependence prevents vector...										main.c:86
loop in main at main.c:81	Assum...	0.000s	61.629s	Scalar	vector dependence prevents vectorization										main.c:81
loop in _libc_start_main		0.000s	61.629s	Scalar											

Below the main table, there is a 'Function Call Sites and Loops' section with a table showing the total time percentage, total time, self time, loop type, and why no vectorization for various function call sites and loops.

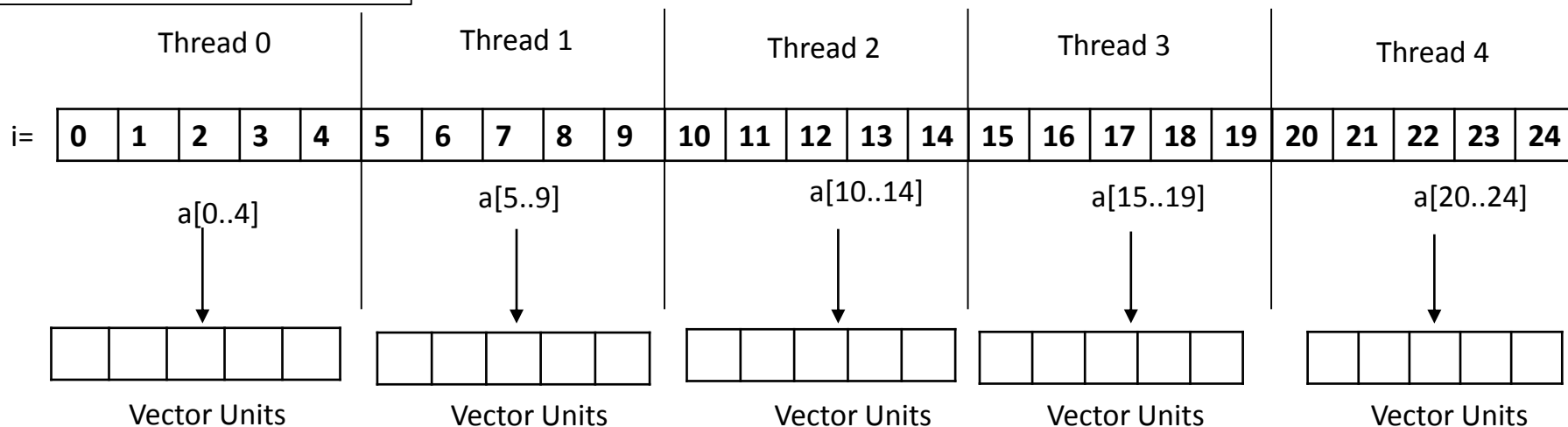
Function Call Sites and Loops	Total Time %	Total Time	Self Time	Loop Type	Why No Vectorization?	Vectoriz...	Instruction S...	Advanced	Location		
						Vec...	VL ...	Traits	Data ...		
Total	100.0%	61.629s	0s								
_libc_start_main	100.0%	61.629s	0s								
loop in _libc_start_main	100.0%	61.629s	0s	Scalar							
main	100.0%	61.629s	0s								
loop in main at main.c:81	100.0%	61.629s	0s	Scalar	vector dependence ...						
loop in main at main.c:86	100.0%	61.629s	0.2400s	Scalar	vector dependence ...					Float64	

pragma OMP for simd

- Parallelize and vectorize a loop nest
 - Distribute a loop's iteration space across a thread team
 - Subdivide loop chunks to fit a SIMD vector register
- Syntax

```
#pragma omp for simd [clause[[,] clause],...]  
for-loops
```

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



pragma OMP for simd

```
#pragma omp parallel for simd
for(i=0; i<msize; i++) {
    a[i][j] = distsq(a[i][j], b[i][j])-auxrand;
    b[i][j] += min(a[i][j], b[i][j])+auxrand;
    c[i][j] = (min(distsq(a[i][j], b[i][j]), a[i][j]))/auxrand;
}
```

Agenda

- NCC Presentation
- Parallel architectures
- OpenMP
- Vectorization
- Offload
- Thread league
- N-body

OpenMP 4.0 Offload

- **target:** transfers the control flow to the target device
 - Transfer is sequential and synchronous
 - Transfer clauses control data flow
- **target data:** creates a scoped device data environment
 - Does not include a transfer of control
 - Transfer clauses control data flow
 - The device data environment is valid through the lifetime of the target data region
- **target update:** request data transfers from within a target data region
- **omp declare target:** creates a structured-block of functions that can be offloaded.

OpenMP 4.0 Offload Report

- **OFFLOAD REPORT:**
 - Measures the amount of time it takes to execute an offload region of code;
 - Measures the amount of data transferred during the execution of the offload region;
 - Turn on the report: Export OFFLOAD_REPORT=2
- **[Var]** The name of a variable transferred and the direction(s) of transfer.
- **[CPU Time]** The total time measured for that offload directive on the host.
- **[MIC Time]** The total time measured for executing the offload on the target.
- **[CPU->MIC Data]** The number of bytes of data transferred from the host to the target.
- **[MIC->CPU Data]** The number of bytes of data transferred from the target to the host.

pragma OMP target

- Transfer control [and data] from the host to the 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;

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;
```

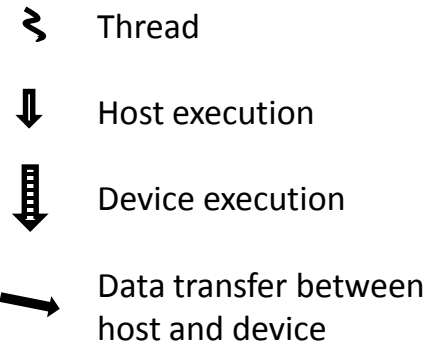
*Offload:
Copy variable:
N,b,l and **a** to device*

```
#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

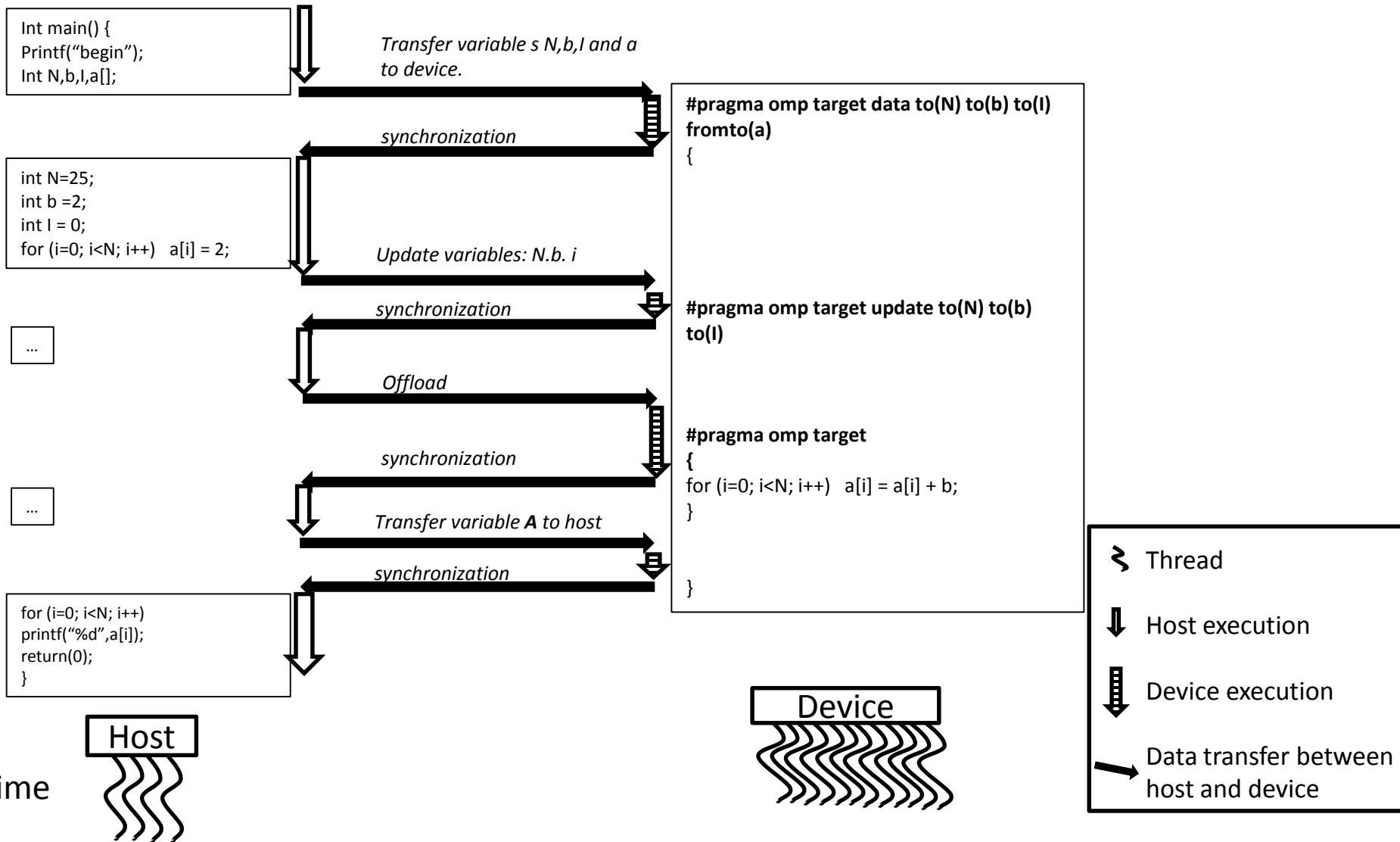
Device



pragma OMP target

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

Offloading – OMP Target Data



pragma OMP target data

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

pragma OMP target update

- Update Data between host and device

- Syntax

```
#pragma omp target update [clause[[,  
clause],...]  
structured-block
```

- Clauses

```
device(scalar-integer-expression)  
map(alloc | to | from | tofrom: list)  
if(scalar-expr)
```

pragma OMP target update

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

Agenda

- NCC Presentation
- Parallel architectures
- OpenMP
- Vectorization
- Offload
- Thread league
- N-body

Thread League

- **omp teams:** creates a league of thread teams
 - #pragma omp teams [clause [[,] clause] . . .]
 - ❑ num_teams(amount) : define the amount of thread teams
 - ❑ thread_limit(limit) : define the highest amount of threads that can be created in each team;
- **omp distribute:** distributes a loop over the teams in the league
 - #pragma omp distribute [clause [[,] clause] . . .]
 - ❑ dist_schedule (static[block size]):

Thread League

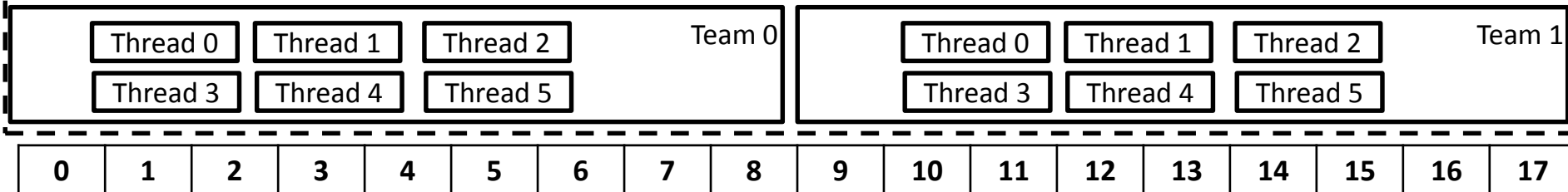
```
#pragma omp target teams num_teams (2) thread_limit (6)
{
  int i , N, teams , idteam , idthread ; int sum; N=20;
  #pragma omp distribute parallel for reduction (+:sum)
  for ( i =0; i<N; i ++ ) sum += i ;
}
```

Example1

```
#pragma omp target teams num_teams (3) thread_limit (3)
{
  int i , N, teams , idteam , idthread ; int sum; N=20;
  #pragma omp distribute parallel for reduction (+:sum)
  for ( i =0; i<N; i ++ ) sum += i ;
}
```

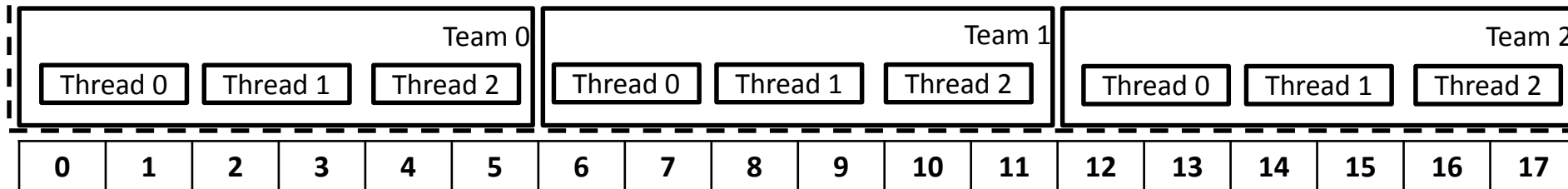
Example2

Thread League– Example 1



Logical Cores

Thread League– Example 2



Logical Cores

Thread League – Example 1

```
#pragma omp target teams num_teams (2) thread_limit( 3 )
{
    int i, N, teams, idteam , idthread;
    int sum;
    N=20;

    #pragma omp distribute parallel for reduction (+: sum)
    for ( i =0; i <N; i ++ ) {
        sum += i ;
        idthread = omp_get_thread_num ();
        idteam = omp_get_team_num () ;
        teams = omp_get_num_teams () ;
        printf("i %d n %d idteam %d idthread %d teams %d \ n" , i ,N, idteam ,
idthread , teams ) ;
    }
}
```

Thread League – Example 2

```
#pragma omp target data device (0) map (i , j , k) map ( to : a[0:NUM]
[0:NUM] ) map ( to : b [ 0 :NUM] [ 0 :NUM] ) map ( tofrom : c [ 0 :NUM] [ 0
:NUM] )
{
    #pragma omp target teams distribute parallel for collapse (2) num_teams
(2) thread_limit (30)
    for ( i =0; i <NUM; i ++ ) {
        for ( k =0; k<NUM; k++) {
            #pragma omp simd
            for ( j =0; j <NUM; j ++ ) {
                c[i][j] = c[i][j] + a [i][k] b[k][j] ;
            }
        }
    }
}
```

Agenda

- NCC Presentation
- Parallel architectures
- OpenMP
- Vectorization
- Offload
- Thread league
- N-body

N-Body

- An N-body simulation [1] aims at 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 gravity force;
- Newton's second law of motion can be used in in a N-body simulation to define 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:
 - In 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 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])  
    }  
}
```

