



An introduction to OpenMP

Ioannis E. Venetis

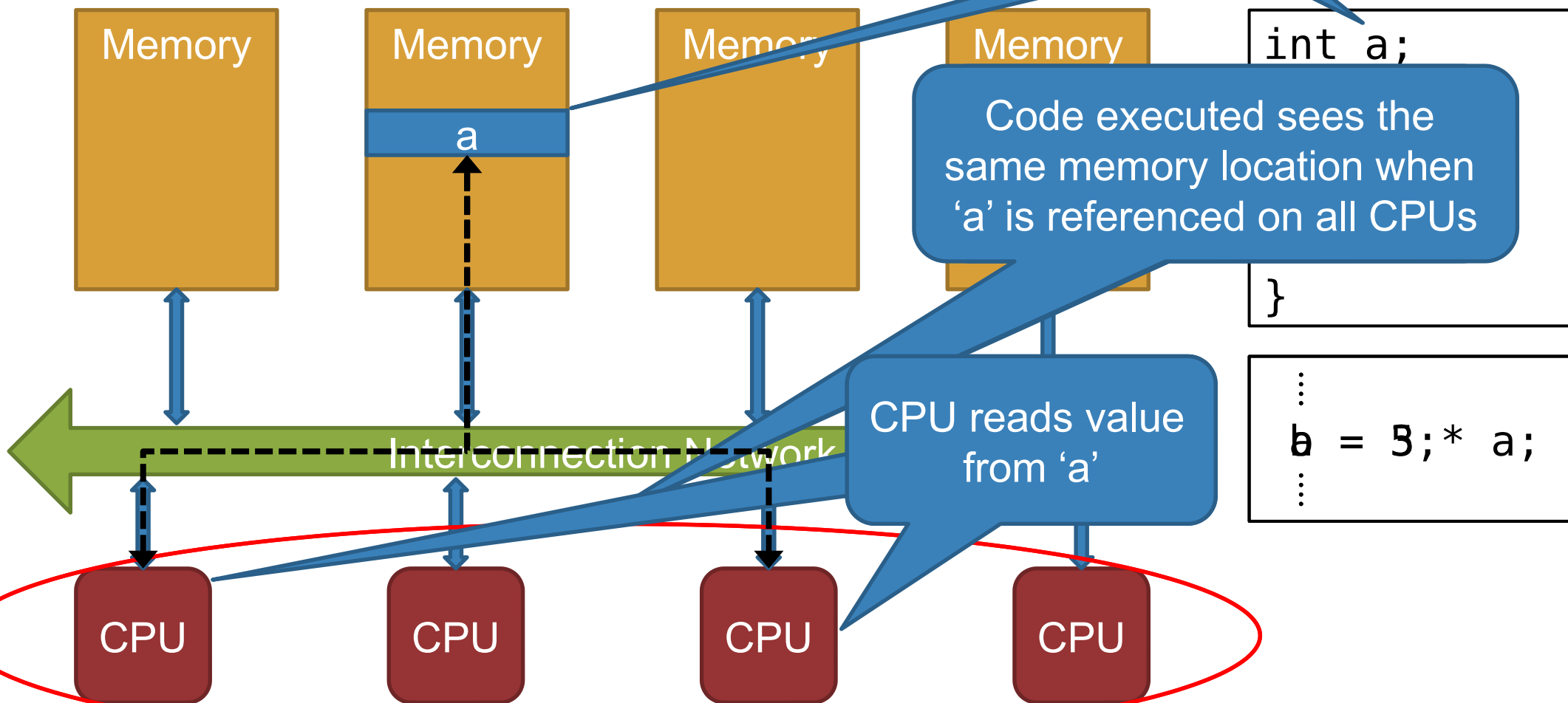
University of Patras



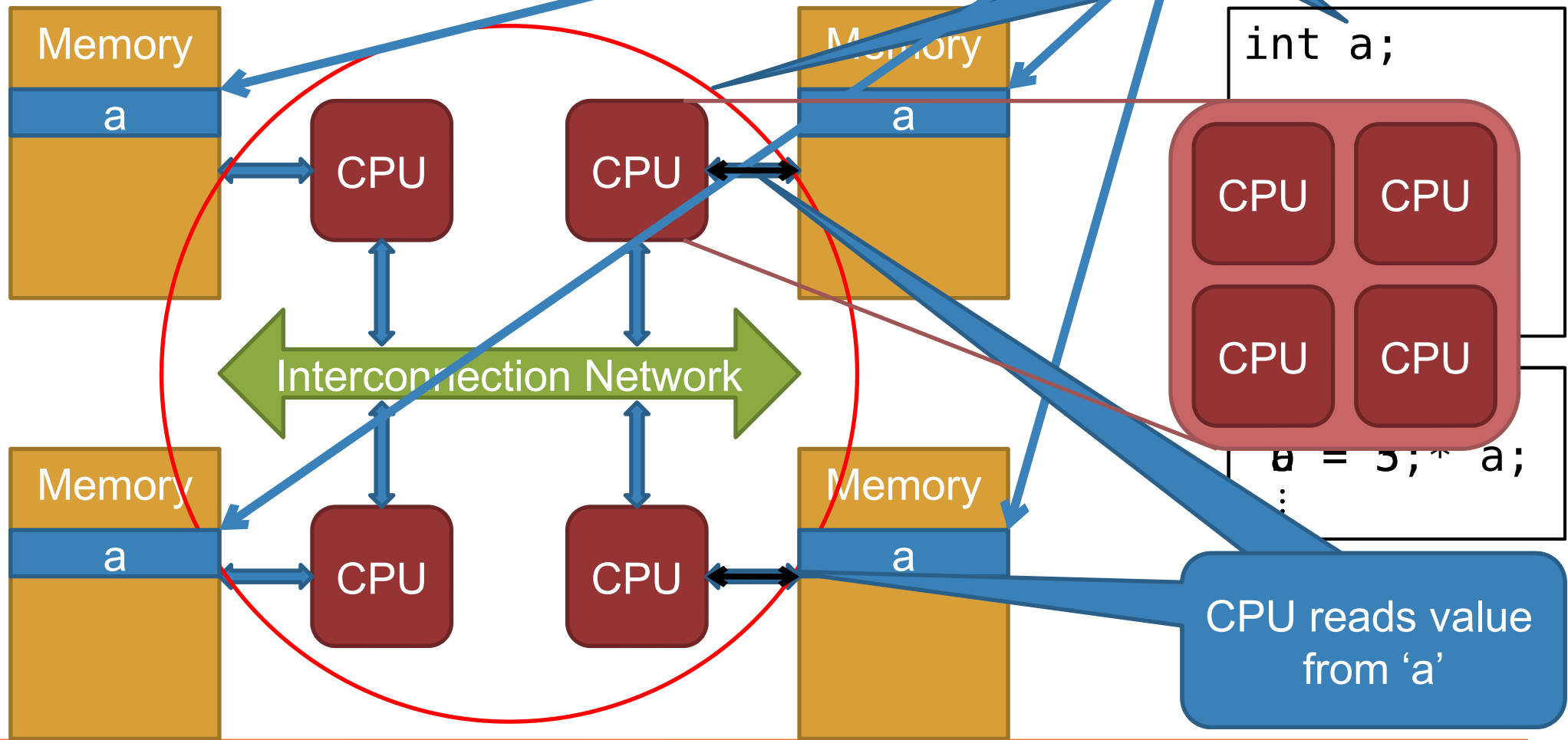
Classification of parallel architectures

- ▶ There are many classification schemes for parallel architectures
- ▶ One of the most useful ones is according to the memory architecture of the system
 - ▶ Shared memory
 - ▶ Distributed memory
 - ▶ Distributed shared memory
- ▶ Largely determines the **programming models** that can be used to program the specific architecture

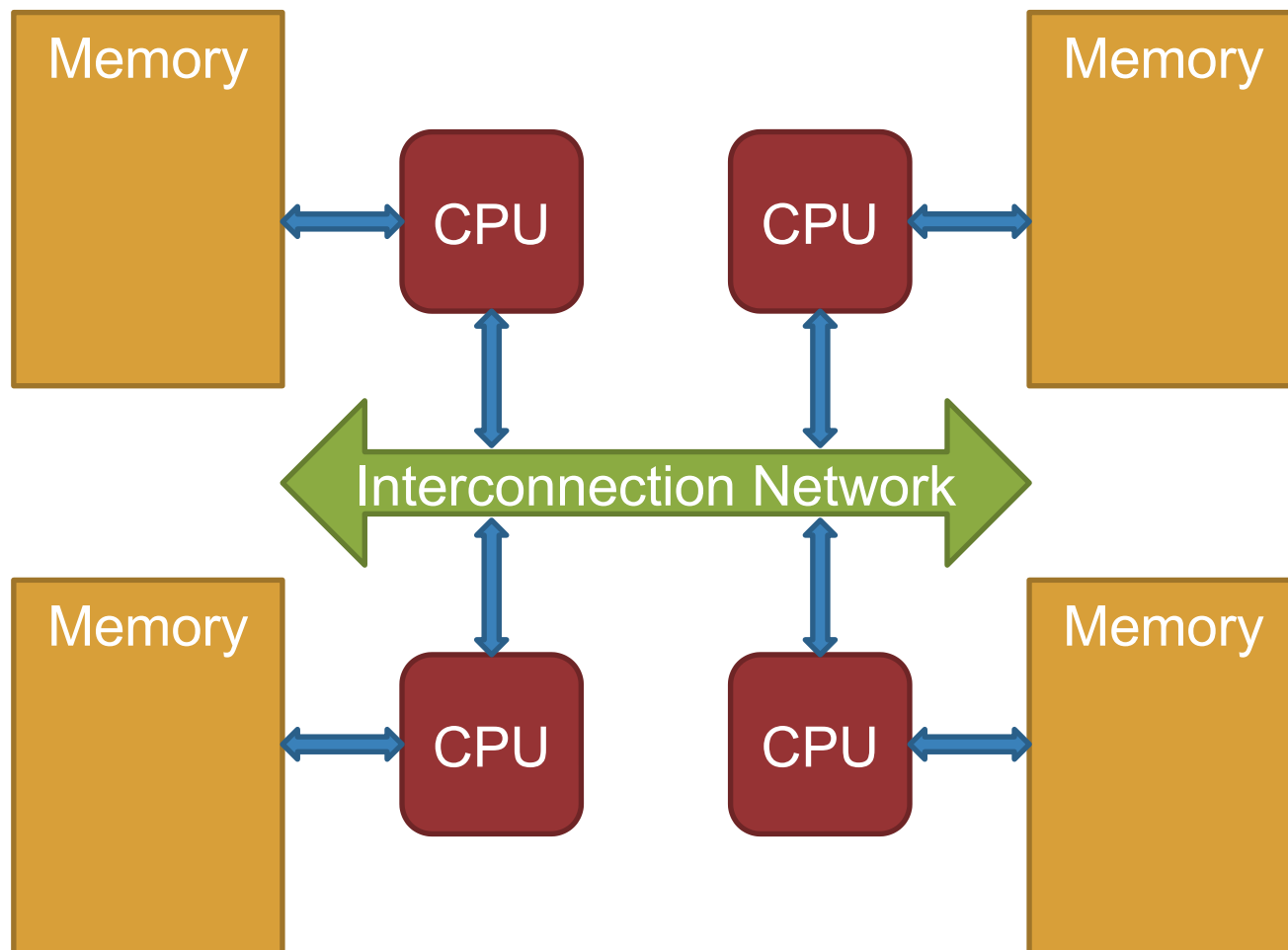
Shared memory parallel architectures



Distributed memory parallel architectures



Distributed shared memory parallel architectures



- ▶ Same organization as distributed memory systems
- ▶ But memory behaves as in shared memory systems
 - ▶ Hardware ensures this kind of behavior



Programming models for shared memory parallel architectures

- ▶ OpenMP
- ▶ Threads
 - ▶ POSIX Threads as well as other threading libraries
- ▶ Cilk
- ▶ Cilk Plus
- ▶ Threading Building Blocks
- ▶ ...



Programming models for distributed memory parallel architectures

- ▶ MPI
- ▶ PVM
- ▶ ...

Ultimate winner!



What is the goal when using a parallel programming model?

- ▶ Problem to be solved is large
 - ▶ Requires too much time to run on single CPU
- ▶ Divide input data into smaller chunks
 - ▶ Smaller size of input \Rightarrow Faster calculation
- ▶ Assign every chunk to a CPU
 - ▶ Using appropriate software abstraction (thread or process)
- ▶ Every CPU processes its chunk and creates part of the solution
- ▶ Combine partial solutions into global solution



Threads to express parallelism

- ▶ Typically a specific series of steps is followed by the programmer
 - ▶ Identification of code that can execute in parallel
 - ▶ Put code into separate function
 - ▶ Pass (typically) one parameter
 - ▶ *An identification number for the thread (ID)*
 - ▶ Distribution of workload using the above parameter
 - ▶ *Each thread can determine on its own the part of the workload it will process*
 - ▶ *How this part of the workload is calculated is usually the same or similar no matter the specific problem being solved*
 - ▶ Addition of mutual exclusion/synchronization



The main idea of OpenMP

- ▶ The steps described in the previous slide:
 - ▶ Are well defined
 - ▶ Repeat in many problems
- ▶ The real problem for the programmer is to find a good way to parallelize the application
 - ▶ The implementation of the solution can be achieved to an large degree in an automated way
- ▶ Can the compiler undertake the automated part of programming?



Directive based programming models

- ▶ There have been several proposals and implementations
 - ▶ Sequent Fortran
 - ▶ Cedar Fortran
 - ▶ PCF Fortran
 - ▶ SGI
 - ▶ ANSI X3H5
- ▶ Most failed because:
 - ▶ They were not general enough
 - ▶ *Could not support enough parallelization approaches of applications*
 - ▶ They were overlapping
 - ▶ They were proposed in the wrong point in time



OpenMP

- ▶ Why do we need another standard?
 - ▶ Previously proposed models were in reality never standardized
 - ▶ *Many companies provided their own directives for parallelization*
 - ▶ *Similar syntax and meaning*
 - ▶ *But different names for the directive*
 - ▶ *Created portability issues*
- ▶ OpenMP unifies all previous attempts and extends them:
 - ▶ A unique set of directives with a uniquely defined syntax and meaning
 - ▶ Portability of source code for programming shared memory systems

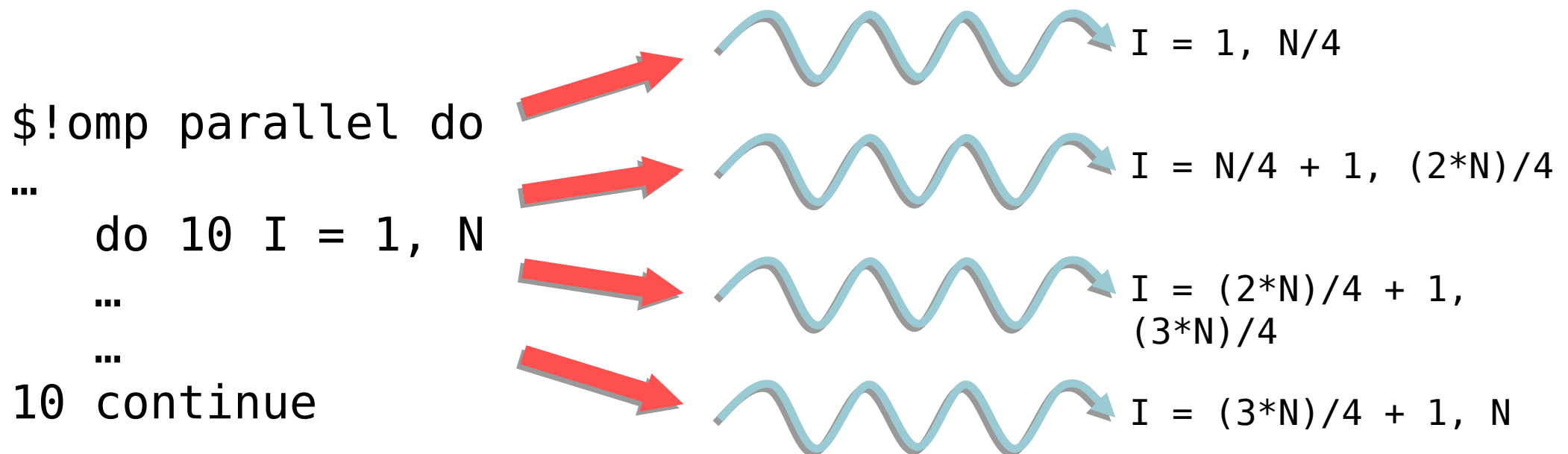


What is then OpenMP?

- ▶ It is a specification
 - ▶ Not a specific implementation
- ▶ It provides an Application Programming Interface (API) for programming shared memory systems:
 - ▶ Directives to the compiler
 - ▶ Run-Time Library
 - ▶ Environment Variables
- ▶ It supports C, C++ and Fortran
- ▶ Programs written in OpenMP can also be compiled and run as serial programs

How does OpenMP work?

- ▶ The programmer directs the compiler **where and how** it should parallelize code
- ▶ The compiler creates parallelism





Who does support OpenMP?

- ▶ GNU GCC ≥ 4.2
- ▶ IBM XL Compilers
- ▶ Sun Microsystems Sun Studio Compilers
- ▶ Intel Compilers
- ▶ Portland Group Compilers
- ▶ Absoft Pro Compilers
 - ▶ Only Fortran
- ▶ Lahey/Fujitsu Compilers
- ▶ PathScale Compilers
- ▶ Microsoft Visual Studio
- ▶ ...

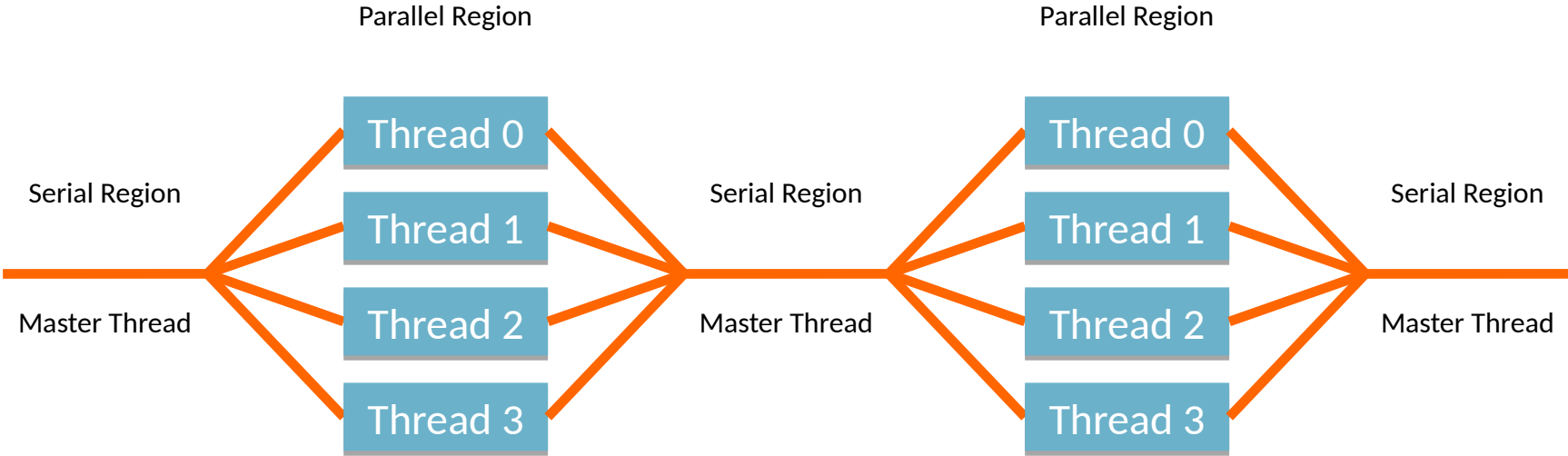


More information about OpenMP

- ▶ Current specification
 - ▶ OpenMP 4.5, November 2015
- ▶ Official web site
 - ▶ <http://www.openmp.org>
- ▶ OpenMP user community
 - ▶ <http://www.compunity.org>

Execution model of OpenMP (1/4)

- ▶ Based on threads and tasks
- ▶ Follows the fork-join model





Execution model of OpenMP (2/4)

- ▶ When a thread encounters a “parallel” directive
 - ▶ The thread creates a team of threads that includes itself and zero or more additional threads
 - ▶ *The thread becomes the “master thread”*
 - ▶ A set of tasks is created
 - ▶ *The code that each task will execute is determined by the code within the “parallel” construct*
 - ▶ Each task is assigned to a thread and is bound to it
 - ▶ Execution of the current task is suspended
 - ▶ Tasks of the newly created team are executed



Execution model of OpenMP (3/4)

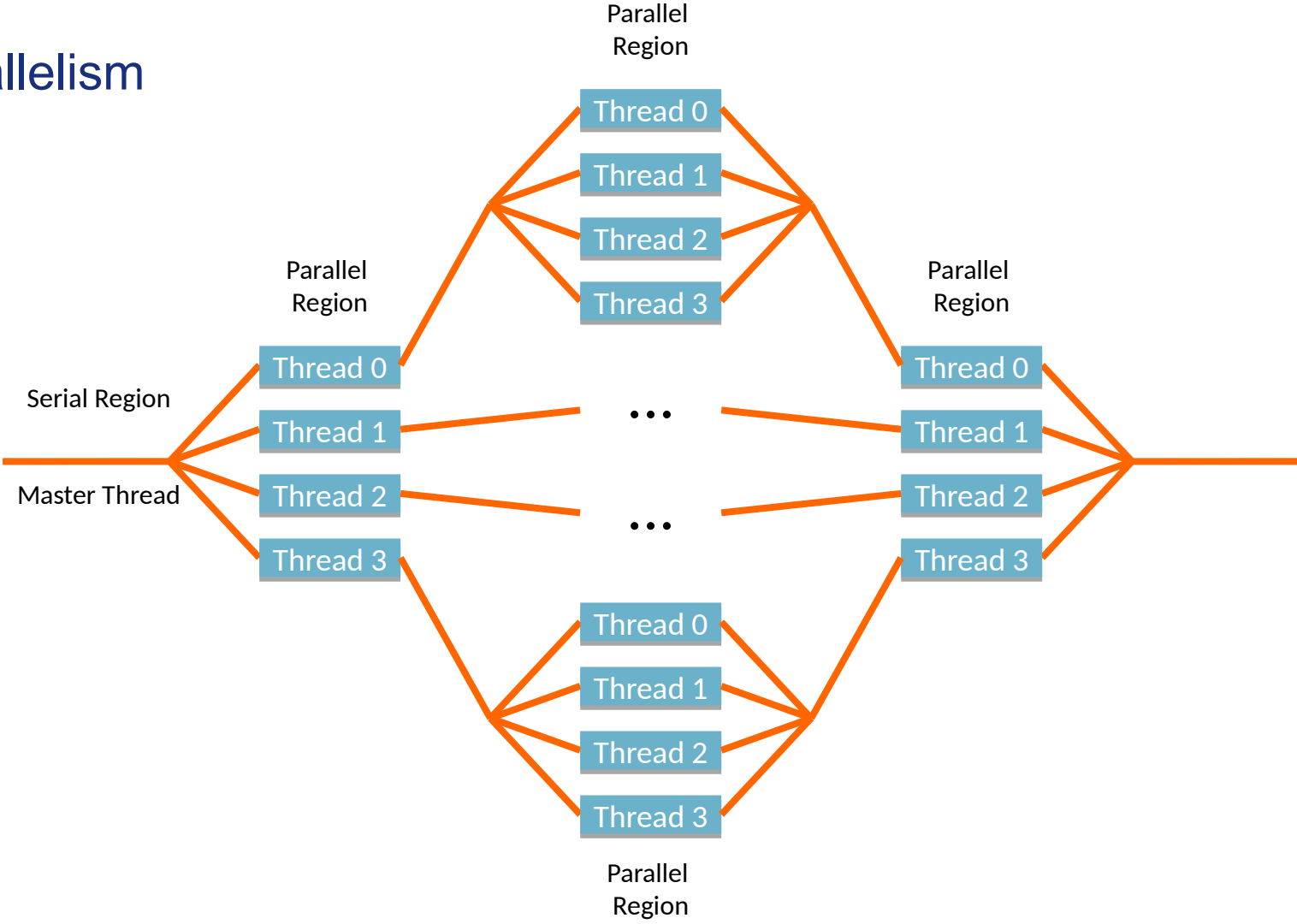
- ▶ At the end of each parallel region a barrier exists
 - ▶ A method to synchronize threads
 - ▶ All threads need to reach the barrier
 - ▶ *As soon as the last thread reaches the barrier, all threads can pass the barrier and continue their execution*
- ▶ After the end of a parallel region
 - ▶ Only the master thread continues execution



Execution model of OpenMP (4/4)

- ▶ OpenMP allows optional support for nested parallelism
 - ▶ Threads in a parallel region may themselves create new threads
- ▶ The implementers of OpenMP in a specific compiler decide whether they will support it or not
 - ▶ If not supported every nested parallel construct is ignored
 - ▶ *The new team of threads that is created when a parallel construct is encountered is composed only of the encountering thread*

Nested parallelism

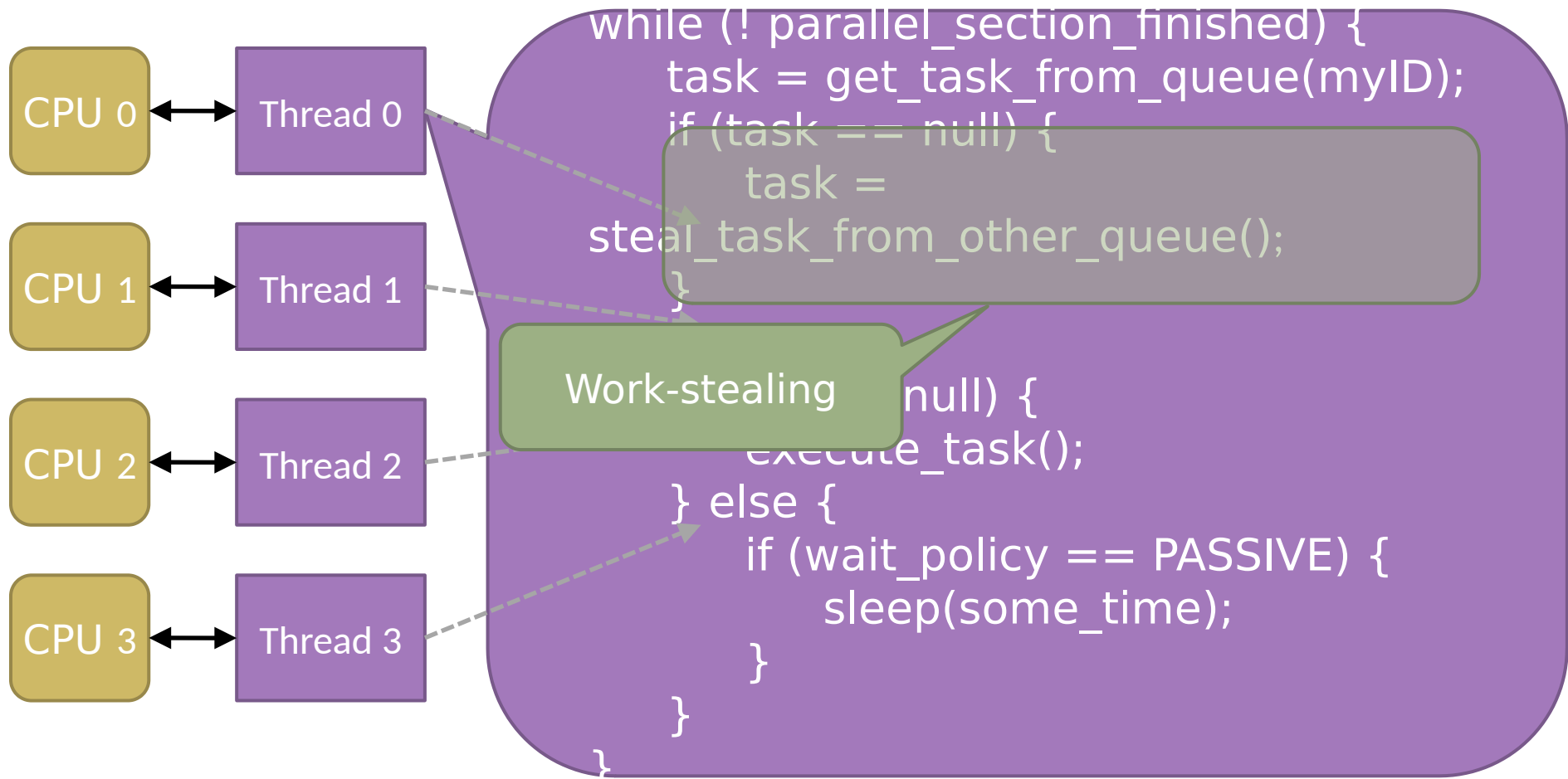




Why tasks in addition to threads?

- ▶ When a thread is created
 - ▶ We determine the code it should execute
 - ▶ We define its arguments
 - ▶ It constitutes a **static** way to assign work
- ▶ A task only describes the work it should perform
 - ▶ How is a task selected to execute?

User-level scheduler





Implementation issues

- ▶ Many issues mentioned in the previous slide are **not** required by the OpenMP specification
- ▶ They depend on the specific implementation
 - ▶ Will work-stealing be supported or not?
 - ▶ *How does it affect performance?*
 - ▶ What happens with threads when a parallel region ends?
 - ▶ *Are they destroyed?*
 - ▶ *Are they kept inactive and reused when the next parallel region is encountered?*
- ▶ As a result, the implementation we discussed is only one possibility among many
 - ▶ Although it already includes some advanced features



Advantages of tasks

- ▶ Tasks are suitable for parallelizing irregular problems
 - ▶ Loops with unknown limits
 - ▶ Recursive algorithms
 - ▶ Producer/Consumer problems
 - ▶ Trees
 - ▶ Graphs
- ▶ The most important addition in version 3.0 of OpenMP



OpenMP directives for C/C++

- ▶ The “#pragma” mechanism is used
 - ▶ Directive to the preprocessor
- ▶ Every OpenMP directive starts with “#pragma omp”
 - ▶ Directives are case-sensitive
- ▶ Every OpenMP directive applies to the structured block that immediately follows the directive
- ▶ General form:
`#pragma omp directive-name [clause [,] clause]... new-line`



A first example

```
#include <omp.h>
#include <stdio.h>

int main(void){
    #pragma omp parallel
    {
        printf( "Hello World from thread %i of %i\n",
            omp_get_thread_num(),
            omp_get_num_threads());
    }
    return 0;
}
```



How many threads?

- ▶ We didn't define the number of threads to be created
- ▶ There are 4 ways to do it:
 - ▶ We add the clause “`num_threads(integer-expression)`” to the “parallel” directive
 - ▶ The source code calls the function “`omp_set_num_threads(integer-expression)`” from the OpenMP Run-Time Library
 - ▶ *Before encountering the “parallel” construct*
 - ▶ Before executing the application we set the environment variable “OMP_NUM_THREADS” in the shell
 - ▶ Default number of threads for the specific implementation
 - ▶ *Usually the total number of available processors/cores*



Conditional compilation

- ▶ For OpenMP implementations where a preprocessor is available
 - ▶ The macro “_OPENMP” is defined
 - ▶ *The value of the macro is of the form “yyyymm”*
 - ▶ *“yyyy” and “mm”: The year and the month when the implemented version of OpenMP was officially accepted*
- ▶ Useful when functions from the Run-Time Library of OpenMP are called
 - ▶ If a compiler does not support OpenMP, these functions are not known
 - ▶ *Error during the link stage: **Undefined reference to `...`***
 - ▶ What would happen in the previous example if would try to compile it with a compiler that does not support OpenMP?



Example

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
#ifdef _OPENMP
```

```
    printf("Compiled by an OpenMP-compliant  
implementation.\n");
```

```
#endif
```

```
    return 0;
```

```
}
```



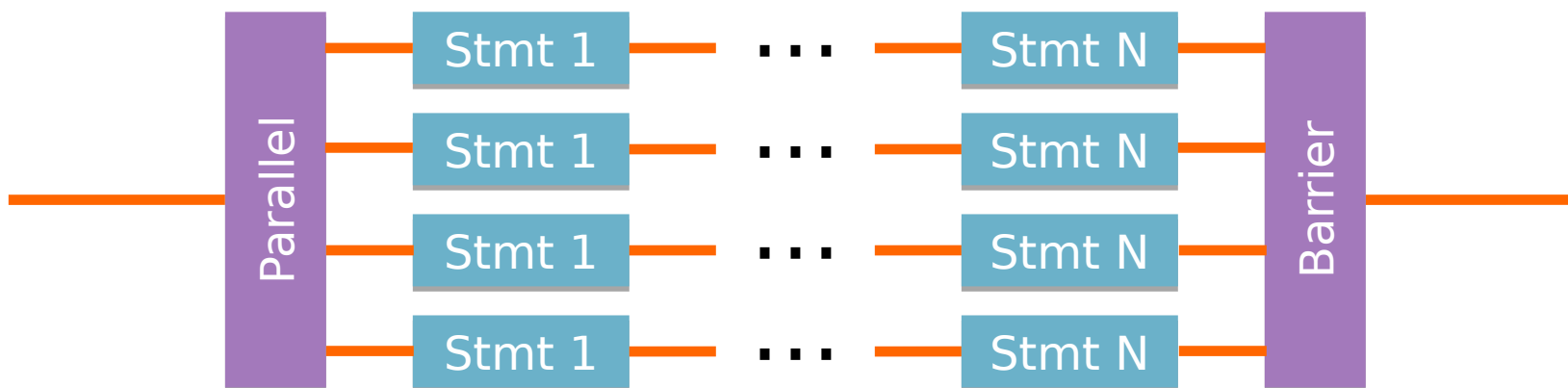
The “parallel” directive

```
#pragma omp parallel [clause [,] clause]... new-line  
    structured-block
```

- ▶ “clause” can be one or more of the following:
 - ▶ if (scalar-expression)
 - ▶ num_threads (integer-expression)
 - ▶ default(shared | none)
 - ▶ private(list)
 - ▶ firstprivate(list)
 - ▶ shared(list)
 - ▶ copyin(list)
 - ▶ reduction(operator : list)

Description

- ▶ The “parallel” construct is the most basic construct in OpenMP
 - ▶ Every thread in the parallel region will execute the “*structured-block*”





Description

- ▶ “if (*scalar-expression*)”
 - ▶ If the expression is true, the “*structured-block*” will be executed by a group of threads
 - ▶ If the expression is false, the “*structured-block*” will be executed serially
- ▶ “num_threads (*integer-expression*)”
 - ▶ Defines how many threads will be created to execute the parallel region
 - ▶ Determining the actual number of threads to be used is quite complicated
 - ▶ *Depends on a multitude of parameters*
 - ▶ *OpenMP 4.5 Specification Document, pages 50-51*



Description

- ▶ “default(shared | none)”
 - ▶ “shared”: All variables referenced in the “*structured-block*” for which no scope has been explicitly defined will be considered to be shared among threads
 - ▶ “none”: The scope for each variable referenced in “*structured-block*” must be specified explicitly
 - ▶ *Shared among threads or private per thread*
- ▶ “shared(*list*)”
 - ▶ All variables referenced in “*list*” become shared among threads



Description

- ▶ “private(*list*)”
 - ▶ All variables referenced in “list” become private per thread
- ▶ “firstprivate(*list*)”
 - ▶ All variables referenced in “list” become private per thread
 - ▶ All local copies of the initial variables are initialized with the last value contained in the corresponding variable before encountering the parallel region



Description

- ▶ “reduction(*operator* : *list*)”
 - ▶ All variables referenced in “list” become private per thread
 - ▶ All local copies of the initial variables are initialized with an appropriate value (see next slide)
 - ▶ At the end of the parallel region each of the initial variables is updated according to the selected operator and the values of the corresponding local copies

Operators for the “reduction” clause

Operator	Initial value
+	0
*	1
-	0
&	~0
	0
^	0
&&	1
	0



Serial addition of vectors (1/2)

```
int *A, *B, *C;
int N;

int main(int argc, char *argv[]) {
    int i;

    if (argc != 2) {
        printf("Provide the problem size.\n");
        exit(0);
    }

    N = atoi(argv[1]);

    A = (int *)malloc(N * sizeof(int));
    B = (int *)malloc(N * sizeof(int));
    C = (int *)malloc(N * sizeof(int));
    if ((A == NULL) || (B == NULL) || (C == NULL)) {
        printf("Could not allocate memory.\n");
        exit(0);
    }
}
```



Serial addition of vectors (2/2)

```
for (i = 0; i < N; i++) {
    C[i] = A[i] + B[i];
}

printf("C = ");
for (i = 0; i < N; i++) {
    printf("%d ", C[i]);
}

return(0);
} /* main() ends here */
```



Addition of vectors with OpenMP (1/3)

```
int *A, *B, *C;
int N;

int main(int argc, char *argv[]) {

    if (argc != 2) {
        printf("Provide the problem size.\n");
        exit(0);
    }

    N = atoi(argv[1]);

    A = (int *)malloc(N * sizeof(int));
    B = (int *)malloc(N * sizeof(int));
    C = (int *)malloc(N * sizeof(int));
    if ((A == NULL) || (B == NULL) || (C == NULL)) {
        printf("Could not allocate memory.\n");
        exit(0);
    }
}
```




Addition of vectors with OpenMP (2/3)

```
#pragma omp parallel
{
    int i, start, end, numOfThreads, numOfElements, id;

    id = omp_get_thread_num();
    numOfThreads = omp_get_num_threads();

    numOfElements = N / numOfThreads;
    start = numOfElements * id;

    if (id != numOfThreads - 1) {
        end = start + numOfElements;
    } else {
        end = N;
    }

    for (i = start; i < end; i++) {
        C[i] = A[i] + B[i];
    }
} /* #pragma omp parallel ends here */
```

Επιστρέφει το ID του νήματος
μικρές μεταβλητές

Επιστρέφει το πλήθος των νημάτων που δημιουργήθηκαν στην παράλληλη περιοχή



Addition of vectors with OpenMP (3/3)

```
printf("C = ");  
for (i = 0; i < N; i++) {  
    printf("%d ", C[i]);  
}  
  
return(0);  
} /* main() ends here */
```



Example (Compilation/Execution)

```
$ gcc -O3 -Wall -fopenmp -o my_prog my_prog.c
```

```
$ ./my_prog
```

12

```
C = ...
```

```
$
```

Size of vectors



Variation of vector addition (1/3)

```
int *A, *B, *C;
```

```
int main(int argc, char *argv[]) {  
    int i, N, start, end, numOfThreads, numOfElements, id;
```

```
    if (argc != 2) {  
        printf("Provide the problem size.\n");  
        exit(0);  
    }
```

```
    N = atoi(argv[1]);
```

```
    A = (int *)malloc(N * sizeof(int));  
    B = (int *)malloc(N * sizeof(int));  
    C = (int *)malloc(N * sizeof(int));  
    if ((A == NULL) || (B == NULL) || (C == NULL)) {  
        printf("Could not allocate memory.\n");  
        exit(0);  
    }
```



Variation of vector addition (2/3)

```
#pragma omp parallel shared(N) private(i, id, numOfThreads, numOfElements,  
    start, end)  
{  
    id = omp_get_thread_num();  
    numOfThreads = omp_get_num_threads();  
  
    numOfElements = N / numOfThreads;  
    start = numOfElements * id;  
  
    if (id != numOfThreads - 1) {  
        end = start + numOfElements;  
    } else {  
        end = N;  
    }  
  
    for (i = start; i < end; i++) {  
        C[i] = A[i] + B[i];  
    }  
} /* #pragma omp parallel ends here */
```



Variation of vector addition (3/3)

```
printf("C = ");  
for (i = 0; i < N; i++) {  
    printf("%d ", C[i]);  
}  
  
return(0);  
} /* main() ends here */
```



Why would we prefer the second version?

- ▶ Think how you typically write a serial program
 - ▶ All variables declared at the beginning of a function
 - ▶ No one would really create a new block of code using curly braces to declare the variables inside that block
- ▶ OpenMP is designed to easily extend serial programs



Serial addition of matrices (1/2)

```
int *A, *B, *C;
int N;

int main(int argc, char *argv[]) {
    int i, j;

    if (argc != 2) {
        printf("Provide the problem size.\n");
        exit(0);
    }

    N = atoi(argv[1]);

    A = (int *)malloc(N * N * sizeof(int));
    B = (int *)malloc(N * N * sizeof(int));
    C = (int *)malloc(N * N * sizeof(int));
    if ((A == NULL) || (B == NULL) || (C == NULL)) {
        printf("Could not allocate memory.\n");
        exit(0);
    }
}
```




Serial addition of matrices (2/2)

```
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        C[i * N + j] = A[i * N + j] + B[i * N + j];
    }
}

printf("C = ");
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        printf("%d ", C[i * N + j]);
    }
    printf("\n");
}

return(0);
} /* main() ends here */
```



Addition of matrices using OpenMP (1/3)

```
int *A, *B, *C;
```

```
int main(int argc, char *argv[]) {  
    int i, j, N, startLine, endLine, numOfThreads, numOfLines, id;
```

```
    if (argc != 2) {  
        printf("Provide the problem size.\n");  
        exit(0);  
    }
```

```
    N = atoi(argv[1]);
```

```
    A = (int *)malloc(N * N * sizeof(int));  
    B = (int *)malloc(N * N * sizeof(int));  
    C = (int *)malloc(N * N * sizeof(int));  
    if ((A == NULL) || (B == NULL) || (C == NULL)) {  
        printf("Could not allocate memory.\n");  
        exit(0);  
    }
```



Addition of matrices using OpenMP (2/3)

```
#pragma omp parallel shared(N) private(i, j, id, numOfThreads, numOfLines, startLine,
    endLine)
{
    id = omp_get_thread_num();
    numOfThreads = omp_get_num_threads();

    numOfLines = N / numOfThreads;
    startLine = numOfLines * id;

    if (id != numOfThreads - 1) {
        endLine = startLine + numOfLines;
    } else {
        endLine = N;
    }

    for (i = startLine; i < endLine; i++) {
        for (j = 0; j < N; j++) {
            C[i * N + j] = A[i * N + j] + B[j * N + j];
        }
    }
} /* #pragma omp parallel ends here */
```



Addition of matrices using OpenMP (3/3)

```
printf("C = ");
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        printf("%d ", C[i * N + j]);
    }
    printf("\n");
}

return(0);
} /* main() ends here */
```



Parallel initialization of vector

```
#include <omp.h>

void subdomain(float *x, int istart, int
ipoints)
{
    int i;
    for (i = 0; i < ipoints; i++)
        x[istart+i] = 123.456;
}

void sub(float *x, int npoints)
{
    int iam, nt, ipoints, istart;

    #pragma omp parallel default(shared)
        private(iam,nt,ipoints,istart)
    {
        iam = omp_get_thread_num();
        nt = omp_get_num_threads();

        /* size of partition */
        ipoints = npoints / nt;

        /* starting array index */
        istart = iam * ipoints;

        /* last thread may do more */
        if (iam == nt-1)
            ipoints = npoints - istart;

        subdomain(x, istart, ipoints);
    }
}
```

```
int main()
{
    float array[10000];

    sub(array, 10000);

    return 0;
}
```



Work sharing constructs

- ▶ The “parallel” construct assigns the same code to all threads
- ▶ Usually we need to distribute work among more threads:
 - ▶ Distribute iterations of a loop
 - ▶ Parallel execution of different parts of the code
- ▶ Available constructs in OpenMP
 - ▶ Work sharing construct for loops
 - ▶ “sections”



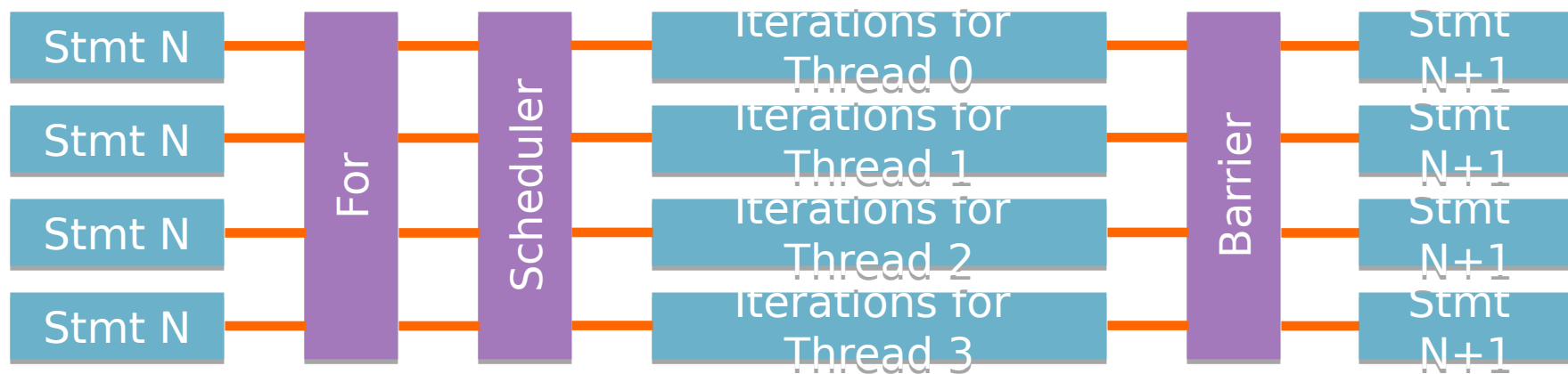
Work sharing construct for loops

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

- ▶ “clause” can be one or more of the following:
 - ▶ `private(list)`
 - ▶ `firstprivate(list)`
 - ▶ `lastprivate(list)`
 - ▶ `reduction(operator : list)`
 - ▶ `schedule(kind[, chunk_size])`
 - ▶ `collapse(n)`
 - ▶ `ordered`
 - ▶ `nowait`

Description

- ▶ The “for” construct is probably the second most used directive in OpenMP
 - ▶ The iterations of the loop are distributed among preexisting threads
 - ▶ *The “for” construct must be inside a parallel construct for parallel execution*





Description

- ▶ “lastprivate(*list*)”
 - ▶ All variables referenced in “list” become private variables per thread
 - ▶ At the end of the “for” construct, each variable is updated with the value that results from the execution of the last iteration of the loop
- ▶ “nowait”
 - ▶ Threads will not wait at the implied barrier at the end of the “for” construct



Description

- ▶ “ordered”
 - ▶ Declares that the iterations of the loop must be executed in the order that the serial code dictates
- ▶ “collapse(n)”
 - ▶ Defines how many levels of a nested for-loop construct will be combined and distributed among the available threads
 - ▶ How are iterations distributed if more than one levels are requested to be collapsed?
 - ▶ *OpenMP 4.5 Specification Document, pages 58-59*



Description

- ▶ “`schedule(kind[, chunk_size])`”
 - ▶ Defines how iterations of a loop are distributed among available threads of a parallel region
 - ▶ “*static, chunk_size*”: Iterations are divided into smaller parts of size “*chunk_size*” (except the last one which can have less) and are assigned to threads in a round-robin fashion
 - ▶ “*dynamic, chunk_size*”: Iterations are divided into smaller parts of size “*chunk_size*” (except the last one which can have less) and threads grab a new part dynamically, as soon as they finish with the previous part
 - ▶ “*guided, chunk_size*”: Iterations are divided into smaller parts. The size of each part equals the number of unassigned iterations divided by the number of available threads. Each part is not allowed to have less than “*chunk_size*” iterations. The assignment of parts to threads is performed dynamically.
 - ▶ “*auto*” or “*runtime*”



Serial addition of vectors (1/2)

```
int *A, *B, *C;
int N;

int main(int argc, char *argv[]) {
    int i;

    if (argc != 2) {
        printf("Provide the problem size.\n");
        exit(0);
    }

    N = atoi(argv[1]);

    A = (int *)malloc(N * sizeof(int));
    B = (int *)malloc(N * sizeof(int));
    C = (int *)malloc(N * sizeof(int));
    if ((A == NULL) || (B == NULL) || (C == NULL)) {
        printf("Could not allocate memory.\n");
        exit(0);
    }
}
```



Serial addition of vectors (2/2)

```
for (i = 0; i < N; i++) {
    C[i] = A[i] + B[i];
}

printf("C = ");
for (i = 0; i < N; i++) {
    printf("%d ", C[i]);
}

return(0);
} /* main() ends here */
```



Addition of vectors with OpenMP (1/2)

```
int *A, *B, *C;
int N;

int main(int argc, char *argv[]) {
    int i;

    if (argc != 2) {
        printf("Provide the problem size.\n");
        exit(0);
    }

    N = atoi(argv[1]);

    A = (int *)malloc(N * sizeof(int));
    B = (int *)malloc(N * sizeof(int));
    C = (int *)malloc(N * sizeof(int));
    if ((A == NULL) || (B == NULL) || (C == NULL)) {
        printf("Could not allocate memory.\n");
        exit(0);
    }
}
```



Addition of vectors with OpenMP (2/2)

```
#pragma omp parallel
{
#pragma omp for
for (i = 0; i < N; i++) {
    C[i] = A[i] + B[i];
}
}

printf("C = ");
for (i = 0; i < N; i++) {
    printf("%d ", C[i]);
}

return(0);
} /* main() ends here */
```



Serial addition of matrices (1/2)

```
int *A, *B, *C;  
int N;
```

```
int main(int argc, char *argv[]) {  
    int i, j;  
  
    if (argc != 2) {  
        printf("Provide the problem size.\n");  
        exit(0);  
    }  
  
    N = atoi(argv[1]);  
  
    A = (int *)malloc(N * N * sizeof(int));  
    B = (int *)malloc(N * N * sizeof(int));  
    C = (int *)malloc(N * N * sizeof(int));  
    if ((A == NULL) || (B == NULL) || (C == NULL)) {  
        printf("Could not allocate memory.\n");  
        exit(0);  
    }  
}
```




Serial addition of matrices (2/2)

```
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        C[i * N + j] = A[i * N + j] + B[i * N + j];
    }
}

printf("C = ");
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        printf("%d ", C[i * N + j]);
    }
    printf("\n");
}

return(0);
} /* main() ends here */
```



Addition of matrices with OpenMP (1/2)

```
int *A, *B, *C;
int N;

int main(int argc, char *argv[]) {
    int i, j;

    if (argc != 2) {
        printf("Provide the problem size.\n");
        exit(0);
    }

    N = atoi(argv[1]);

    A = (int *)malloc(N * N * sizeof(int));
    B = (int *)malloc(N * N * sizeof(int));
    C = (int *)malloc(N * N * sizeof(int));
    if ((A == NULL) || (B == NULL) || (C == NULL)) {
        printf("Could not allocate memory.\n");
        exit(0);
    }
}
```



Addition of matrices with OpenMP (2/2)

```
#pragma omp parallel private(j)
#pragma omp for
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        C[i * N + j] = A[i * N + j] + B[i * N + j];
    }
}

printf("C = ");
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        printf("%d ", C[i * N + j]);
    }
    printf("\n");
}

return(0);
} /* main() ends here */
```



Variation of matrix addition (1/2)

```
int *A, *B, *C;  
int N;
```

```
int main(int argc, char *argv[]) {  
    int i, j;  
  
    if (argc != 2) {  
        printf("Provide the problem size.\n");  
        exit(0);  
    }  
  
    N = atoi(argv[1]);  
  
    A = (int *)malloc(N * N * sizeof(int));  
    B = (int *)malloc(N * N * sizeof(int));  
    C = (int *)malloc(N * N * sizeof(int));  
    if ((A == NULL) || (B == NULL) || (C == NULL)) {  
        printf("Could not allocate memory.\n");  
        exit(0);  
    }  
}
```



Variation of matrix addition (2/2)

```
for (i = 0; i < N; i++) {
    #pragma omp parallel
    #pragma omp for
    for (j = 0; j < N; j++) {
        C[i * N + j] = A[i * N + j] + B[i * N + j];
    }
}

printf("C = ");
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        printf("%d ", C[i * N + j]);
    }
    printf("\n");
}

return(0);
} /* main() ends here */
```



Which version is better?

- ▶ The first one divides matrices by rows
- ▶ The second one divides matrices by columns

- ▶ How many times are threads created in each case?
 - ▶ How many times are threads joined?
 - ▶ How many barriers are executed?



Lower triangular matrix x vector (1/2)

```
int *A, *B, *C;
int N, chunk;

int main(int argc, char *argv[]) {
    int i, j;

    if (argc != 3) {
        printf("Provide the number of threads to create and the problem size.\n");
        exit(0);
    }

    N = atoi(argv[1]);
    chunk = atoi(argv[2]);

    A = (int *)malloc(N * N * sizeof(int));
    B = (int *)malloc(N * sizeof(int));
    C = (int *)malloc(N * sizeof(int));
    if ((A == NULL) || (B == NULL) || (C == NULL)) {
        printf("Could not allocate memory.\n");
        exit(0);
    }
}
```



Lower triangular matrix x vector (2/2)

```
/*  
 * For all rows in the chunk.  
 */  
#pragma omp parallel private(j)  
#pragma omp for schedule(dynamic, chunk)  
for (i = 0; i < N; i++) {  
    C[i] = 0;  
    /*  
     * For all elements in the current row.  
     */  
    for (j = 0; j < i + 1; j++) {  
        C[i] += (A[j] * B[j]);  
    }  
}  
  
return(0);  
} /* main() ends here */
```


Examples of “collapse”, “nowait” and “schedule” clauses

```
void sub()
{
    int i, j, k;

    #pragma omp for collapse(2)
    private(i,j,k)
    for (k = kl; k <= ku; k += ks)
        for (j = jl; j <= ju; j += js)
            for (i = il; i <= iu; i +=
is)
                bar(a, i, j, k)
}
```

```
#include <math.h>

void a92(int n, float *a, float *b, float *c,
float *y, float *z)
{
    int i;
    #pragma omp parallel
    {
        #pragma omp for schedule(static)
    nowait
        for (i=0; i<n; i++)
            c[i] = (a[i] + b[i]) / 2.0;
        #pragma omp for schedule(static)
    nowait
        for (i=0; i<n; i++)
            z[i] = sqrt(c[i]);
        #pragma omp for schedule(static)
    nowait
        for (i=1; i<=n; i++)
            y[i] = z[i-1] + a[i];
    }
}
```

```
#include <math.h>

void a9(int n, int m, float *a, float *b,
float *y, float *z)
{
    int i;
    #pragma omp parallel
    {
        #pragma omp for nowait
        for (i=1; i<n; i++)
            b[i] = (a[i] + a[i-1]) / 2.0;

        #pragma omp for nowait
        for (i=0; i<m; i++)
            y[i] = sqrt(z[i]);
    }
}
```



Reading and writing ICVs

Variable	Scope	Write	Read	Initial value
dyn-var	Task	OMP_DYNAMIC omp_set_dynamic()	omp_get_dynamic()	If supported, then implementation dependent, otherwise false
nest-var	Task	OMP_NESTED omp_set_nested()	omp_get_nested()	false
nthreads-var	Task	OMP_NUM_THREADS omp_set_num_threads()	omp_get_max_threads()	Implementation dependent
thread-limit-var	Global	OMP_THREAD_LIMIT	omp_get_thread_limit()	Implementation dependent
max-active-levels-var	Global	OMP_MAX_ACTIVE_LEVELS omp_set_max_active_levels()	omp_get_max_active_levels()	Implementation dependent
run-sched-var	Task	OMP_SCHEDULE omp_set_schedule()	omp_get_schedule()	Implementation dependent
def-sched-var	Global	Not supported	Not supported	Implementation dependent
stacksize-var	Global	OMP_STACKSIZE	Not supported	Implementation dependent
wait-policy-var	Global	OMP_WAIT_POLICY	Not supported	Implementation dependent



Precedence rules

Clause...	Which overtakes call to...	which overtakes environment variable...	which overtakes initial value
Not supported	<code>omp_set_dynamic()</code>	<code>OMP_DYNAMIC</code>	<code>dyn-var</code>
Not supported	<code>omp_set_nested()</code>	<code>OMP_NESTED</code>	<code>nest-var</code>
<code>num_threads</code>	<code>omp_set_num_threads()</code>	<code>OMP_NUM_THREADS</code>	<code>nthreads-var</code>
Not supported	Not supported	<code>OMP_THREAD_LIMIT</code>	<code>thread-limit-var</code>
Not supported	<code>omp_set_max_active_levels()</code>	<code>OMP_MAX_ACTIVE_LEVELS</code>	<code>max-active-levels-var</code>
<code>schedule</code>	<code>omp_set_schedule()</code>	<code>OMP_SCHEDULE</code>	<code>run-sched-var</code>
<code>schedule</code>	Not supported	Not supported	<code>def-sched-var</code>
Not supported	Not supported	<code>OMP_STACKSIZE</code>	<code>stacksize-var</code>
Not supported	Not supported	<code>OMP_WAIT_POLICY</code>	<code>wait-policy-var</code>



The “sections” construct

- ▶ Syntax of the “sections” construct

```
#pragma omp sections [clause[[, clause] ...] new-line
```

```
{
```

```
  [#pragma omp section new-line]
```

```
    structured-block
```

```
  [#pragma omp section new-line]
```

```
    structured-block]
```

```
}
```



The “sections” construct

- ▶ “clause” can be one or more of the following:
 - ▶ private(list)
 - ▶ firstprivate(list)
 - ▶ lastprivate(list)
 - ▶ reduction(operator : list)
 - ▶ nowait
- ▶ Composed of a set of “structured-blocks”
 - ▶ Each one of them executes on a single thread



Example (1/2)

```
int *A, *B, *C, *D;  
int N;
```

```
int main(int argc, char *argv[]) {  
    int i, j;  
  
    if (argc != 2) {  
        printf("Provide the problem size.\n");  
        exit(0);  
    }  
  
    N = atoi(argv[1]);  
  
    A = (int *)malloc(N * sizeof(int));  
    B = (int *)malloc(N * sizeof(int));  
    C = (int *)malloc(N * sizeof(int));  
    D = (int *)malloc(N * sizeof(int));  
    if ((A == NULL) || (B == NULL) || (C == NULL) || (D == NULL)) {  
        printf("Could not allocate memory.\n");  
        exit(0);  
    }  
}
```



Example (2/2)

```
#pragma omp parallel shared(A, B, C, D) private(i)
{
    #pragma omp sections nowait
    {

        #pragma omp section
        for (i = 0; i < N; i++)
            C[i] = A[i] + B[i];

        #pragma omp section
        for (i = 0; i < N; i++)
            D[i] = A[i] * B[i];

    }
} /* #pragma omp parallel ends here */
} /* main() ends here */
```

One thread
executes this loop

One thread
executes this loop



Example

```
void XAXIS();
void YAXIS();
void ZAXIS();

void all()
{
  #pragma omp parallel
    #pragma omp sections
    {
      #pragma omp
section
      XAXIS();

      #pragma omp
section
      YAXIS();

      #pragma omp
section
      ZAXIS();
    }
}
```




Combined parallelization and work sharing constructs

- ▶ Shortcuts to define a “parallel” and a work sharing construct together
 - ▶ The meaning is the same like having a “parallel” construct followed immediately by a work sharing construct
- ▶ Allowed clauses are the union of the clauses of the two constructs
 - ▶ With a few exceptions



The “parallel for” construct

```
#pragma omp parallel for [clause [[,] clause]...] new-line  
for-loop
```

- ▶ “clause” can be any clause of the “parallel” or the “for” construct
 - ▶ With the exception of “nowait”



Serial addition of vectors (1/2)

```
int *A, *B, *C;  
int N;
```

```
int main(int argc, char *argv[]) {  
    int i;  
  
    if (argc != 2) {  
        printf("Provide the problem size.\n");  
        exit(0);  
    }  
  
    N = atoi(argv[1]);  
  
    A = (int *)malloc(N * sizeof(int));  
    B = (int *)malloc(N * sizeof(int));  
    C = (int *)malloc(N * sizeof(int));  
    if ((A == NULL) || (B == NULL) || (C == NULL)) {  
        printf("Could not allocate memory.\n");  
        exit(0);  
    }  
}
```



Serial addition of vectors (2/2)

```
for (i = 0; i < N; i++) {
    C[i] = A[i] + B[i];
}

printf("C = ");
for (i = 0; i < N; i++) {
    printf("%d ", C[i]);
}

return(0);
} /* main() ends here */
```



Addition of vectors with OpenMP (1/2)

```
int *A, *B, *C;
int N;

int main(int argc, char *argv[]) {
    int i;

    if (argc != 2) {
        printf("Provide the problem size.\n");
        exit(0);
    }

    N = atoi(argv[1]);

    A = (int *)malloc(N * sizeof(int));
    B = (int *)malloc(N * sizeof(int));
    C = (int *)malloc(N * sizeof(int));
    if ((A == NULL) || (B == NULL) || (C == NULL)) {
        printf("Could not allocate memory.\n");
        exit(0);
    }
}
```



Addition of vectors with OpenMP (2/2)

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
    C[i] = A[i] + B[i];
}

printf("C = ");
for (i = 0; i < N; i++) {
    printf("%d ", C[i]);
}

return(0);
} /* main() ends here */
```



Serial addition of matrices (1/2)

```
int *A, *B, *C;
int N;

int main(int argc, char *argv[]) {
    int i, j;

    if (argc != 2) {
        printf("Provide the problem size.\n");
        exit(0);
    }

    N = atoi(argv[1]);

    A = (int *)malloc(N * N * sizeof(int));
    B = (int *)malloc(N * N * sizeof(int));
    C = (int *)malloc(N * N * sizeof(int));
    if ((A == NULL) || (B == NULL) || (C == NULL)) {
        printf("Could not allocate memory.\n");
        exit(0);
    }
}
```



Serial addition of matrices (2/2)

```
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        C[i * N + j] = A[i * N + j] + B[i * N + j];
    }
}

printf("C = ");
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        printf("%d ", C[i * N + j]);
    }
    printf("\n");
}

return(0);
} /* main() ends here */
```




Addition of matrices with OpenMP (1/2)

```
int *A, *B, *C;
int N;

int main(int argc, char *argv[]) {
    int i, j;

    if (argc != 2) {
        printf("Provide the problem size.\n");
        exit(0);
    }

    N = atoi(argv[1]);

    A = (int *)malloc(N * N * sizeof(int));
    B = (int *)malloc(N * N * sizeof(int));
    C = (int *)malloc(N * N * sizeof(int));
    if ((A == NULL) || (B == NULL) || (C == NULL)) {
        printf("Could not allocate memory.\n");
        exit(0);
    }
}
```



Addition of matrices with OpenMP (2/2)

```
#pragma omp parallel for private(j)
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        C[i * N + j] = A[i * N + j] + B[i * N + j];
    }
}

printf("C = ");
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        printf("%d ", C[i * N + j]);
    }
    printf("\n");
}

return(0);
} /* main() ends here */
```



The “parallel sections” construct

- ▶ Syntax of the “parallel sections” construct

```
#pragma omp parallel sections [clause[[, clause] ...] new-line
```

```
{
```

```
  [#pragma omp section new-line]
```

```
    structured-block
```

```
  [#pragma omp section new-line]
```

```
    structured-block]
```

```
}
```

- ▶ “clause” can be any clause of the “parallel” or the “sections” construct
 - ▶ With the exception of “nowait”



Example (1/2)

```
int *A, *B, *C, *D;  
int N;
```

```
int main(int argc, char *argv[]) {  
    int i, j;  
  
    if (argc != 2) {  
        printf("Provide the problem size.\n");  
        exit(0);  
    }  
  
    N = atoi(argv[1]);  
  
    A = (int *)malloc(N * sizeof(int));  
    B = (int *)malloc(N * sizeof(int));  
    C = (int *)malloc(N * sizeof(int));  
    D = (int *)malloc(N * sizeof(int));  
    if ((A == NULL) || (B == NULL) || (C == NULL) || (D == NULL)) {  
        printf("Could not allocate memory.\n");  
        exit(0);  
    }  
}
```



Example (2/2)

```
#pragma omp parallel sections shared(A, B, C, D) private(i)
{

    #pragma omp section
    for (i = 0; i < N; i++)
        C[i] = A[i] + B[i];

    #pragma omp section
    for (i = 0; i < N; i++)
        D[i] = A[i] * B[i];

} /* #pragma omp parallel sections ends here */
} /* main() ends here */
```



Example

```
void XAXIS();  
void YAXIS();  
void ZAXIS();  
  
void a11()  
{  
#pragma omp parallel  
sections  
{  
    #pragma omp section  
    XAXIS();  
  
    #pragma omp section  
    YAXIS();  
  
    #pragma omp section  
    ZAXIS();  
}  
}
```



The “master” construct

```
#pragma omp master new-line  
    structured-block
```

- ▶ Only the master thread executes the “structured-block”
- ▶ If nested parallelism is supported
 - ▶ It applies to the innermost team of threads that has been created
- ▶ There is **no barrier** at the end of the construct
 - ▶ In contrast to the “single” construct

Example

```
#include <stdio.h>

extern float average(float, float, float);

void a15( float* x, float* xold, int n, float
tol )
{
    int      c, i, toobig;
    float    error, y;

    c = 0;
    #pragma omp parallel
    {
        do {
            #pragma omp for private(i)
            for( i = 1; i < n-1; ++i ) {
                xold[i] = x[i];
            }

            #pragma omp single
            {
                toobig = 0;
            }
        }
    }
}
```

```
#pragma omp for private(i,y,error) \
reduction(+:toobig)
for( i = 1; i < n-1; ++i ) {
    y = x[i];
    x[i] = average( xold[i-1], x[i],
xold[i+1] );
    error = y - x[i];
    if( error > tol || error < -tol )
        ++toobig;
}

#pragma omp master
{
    ++c;
    printf( "iteration %d, toobig=%d\
n",
c, toobig );
}
} while( toobig > 0 );
}
```




The “single” construct

```
#pragma omp single [clause[[,] clause] ...] new-line  
structured-block
```

- ▶ “clause” can be one or more of the following:
 - ▶ private(list)
 - ▶ firstprivate(list)
 - ▶ copyprivate(list)
 - ▶ nowait



Description

- ▶ The “single” construct defines that the corresponding “*structured-block*” will be executed only by one thread of the team
 - ▶ Not necessarily the master thread
- ▶ The rest of the threads of the team wait on a barrier at the end of the “single” construct
 - ▶ Except if the clause “nowait” has been used



Example

```
#include <stdio.h>

void work1() {}
void work2() {}

void a12()
{
    #pragma omp parallel
    {
        #pragma omp single
        printf("Beginning work1.\n");

        work1();

        #pragma omp single
        printf("Finishing work1.\n");

        #pragma omp single nowait
        printf("Finished work1 and beginning work2.\n");

        work2();
    }
}
```



The “critical” construct

```
#pragma omp critical [(name)] new-line  
    structured-block
```

- ▶ The “structured-block” is executed only by a single thread at any point in time
 - ▶ But all threads encountering the construct will eventually execute it
 - ▶ It does not matter to which team each thread belongs
- ▶ An optional name can be given to each “critical” construct
 - ▶ Only threads that use the same name participate in the mutual exclusion
 - ▶ All threads that don’t use a name participate in a default “critical” construct
 - ▶ *This construct does not have a name*



Multiplication of vectors (1/2)

```
int *A, *B, res = 0;
int N;

int main(int argc, char *argv[]) {
    int i;

    if (argc != 2) {
        printf("Provide the problem size.\n");
        exit(0);
    }

    N = atoi(argv[1]);

    A = (int *)malloc(N * sizeof(int));
    B = (int *)malloc(N * sizeof(int));
    if ((A == NULL) || (B == NULL)) {
        printf("Could not allocate memory.\n");
        exit(0);
    }
}
```



Multiplication of vectors (2/2)

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
    #pragma omp critical
    res += (A[i] * B[i]);
}

printf("Result is %d\n", res);

return(0);
} /* main() ends here */
```



Example

```
int dequeue(float *a);
void work(int i, float *a);

void a16(float *x, float *y)
{
    int ix_next, iy_next;
    #pragma omp parallel shared(x, y) private(ix_next,
iy_next)
    {
        #pragma omp critical(xaxis)
        ix_next = dequeue(x);
        work(ix_next, x);

        #pragma omp critical(yaxis)
        iy_next = dequeue(y);
        work(iy_next, y);
    }
}
```



The “atomic” construct

```
#pragma omp atomic new-line  
expression-stmt
```

- ▶ Declares that a memory location must be updated atomically
- ▶ “expression-stmt” can be:
 - ▶ $x++$, $++x$, $x--$, $--x$
 - ▶ $x \text{ binop} = \text{expr}$
 - ▶ binop : $+$, $-$, $*$, $/$, $\&$, $|$, \wedge , \ll , \gg
 - ▶ *Cannot be an overloaded operator (C++)*
- ▶ Only the update of the memory location is atomic
 - ▶ The calculation of “ expr ” is not atomic



Multiplication of vectors (1/2)

```
int *A, *B, res = 0;
int N;

int main(int argc, char *argv[]) {
    int i;

    if (argc != 2) {
        printf("Provide the problem size.\n");
        exit(0);
    }

    N = atoi(argv[1]);

    A = (int *)malloc(N * sizeof(int));
    B = (int *)malloc(N * sizeof(int));
    if ((A == NULL) || (B == NULL)) {
        printf("Could not allocate memory.\n");
        exit(0);
    }
}
```



Multiplication of vectors (2/2)

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
    #pragma omp atomic
    res += (A[i] * B[i]);
}

printf("Result is %d\n", res);

return(0);
} /* main() ends here */
```



An even better solution...

```
#pragma omp parallel for reduction(+:res)
for (i = 0; i < N; i++) {
    res += (A[i] * B[i]);
}

printf("Result is %d\n", res);

return(0);
} /* main() ends here */
```



Example

```
float work1(int i)
{
    return 1.0 * i;
}

float work2(int i)
{
    return 2.0 * i;
}

void a19(float *x, float *y, int *index, int n)
{
    int i;

    #pragma omp parallel for shared(x, y,
index, n)
    for (i=0; i<n; i++) {
        #pragma omp atomic
        x[index[i]] += work1(i);

        y[i] += work2(i);
    }
}
```

```
int main()
{
    float    x[1000];
    float    y[10000];
    int      index[10000];
    int      i;

    for (i = 0; i < 10000; i++) {
        index[i] = i % 1000;
        y[i]=0.0;
    }

    for (i = 0; i < 1000; i++)
        x[i] = 0.0;

    a19(x, y, index, 10000);

    return 0;
}
```



The “barrier” construct

`#pragma omp barrier new-line`

- ▶ Declares a barrier at the point where the construct appears
- ▶ It operates on the threads of the current team
- ▶ If nested parallelism is supported
 - ▶ It refers to the innermost team of threads that has been created



Example

```
void work(int n) {}

void sub3(int n)
{
    work(n);
    #pragma omp barrier
    work(n);
}

void sub2(int k)
{
    #pragma omp parallel shared(k)
    sub3(k);
}

void sub1(int n)
{
    int i;

    #pragma omp parallel private(i) shared(n)
    {
        #pragma omp for
        for (i=0; i<n; i++)
            sub2(i);
    }
}
```

```
int main()
{
    sub1(2);
    sub2(2);
    sub3(2);

    return 0;
}
```



Handling data dependencies



Data dependencies in loops

- ▶ There is a data dependence between two instructions S_1 and S_2 when they access the same memory location
 - ▶ In the same iteration of the loop
 - ▶ *Loop-Independent Dependencies*
 - ▶ During different iterations of the loop
 - ▶ *Loop-Carried Dependencies*



Parallelization

- ▶ A loop can be easily parallelized when there are no data dependencies
 - ▶ Typical use of parallelization constructs, like `#pragma omp parallel for`
- ▶ Otherwise it can be parallelized:
 - ▶ If it can be converted into a form where parallelization is possible
 - ▶ Using another parallelization approach
 - ▶ Using tasks where data dependencies can be expressed (see “task” struct using clause `depend(in/out/inout)` in OpenMP ≥ 4.0)



Example

```
#define N 1000
```

```
void testParallelization() {  
    double V[N];  
    int i;
```

```
    #pragma omp parallel for shared(V) private(i) schedule(static)
```

```
    for (i = 0; i < N - 1; i++) {  
        V[i] = (V[i] + V[i+1]) / 2.0;
```

```
    }
```

```
}
```





First correct approach

```
#define N 1000
```

```
void testParallelization() {  
    double V[N], oldV[N];  
    int i;  
  
    #pragma omp parallel shared(V, oldV) private(i) schedule(static)  
    {  
        #pragma omp for  
        for (i = 0; i < N; i++) {  
            oldV[i] = V[i];  
        }  
  
        #pragma omp for  
        for (i = 0; i < N - 1; i++) {  
            V[i] = (V[i] + oldV[i+1]) / 2.0;  
        }  
    }  
}
```



A better solution (1/2)

```
#define N 1000
```

```
void testParallelization() {  
    double V[N], border;  
    int i, s, id, nt, limitL, limitR;  
  
    #pragma omp parallel shared(V) private(i,s,id,nt,limitL,limitR,border)  
    {  
        id = omp_get_thread_num();  
        nt = omp_get_num_threads();  
        s = N / nt; /* We assume that they divide exactly. */  
  
        limitL = id * s;  
        limitR = limitL + s - 1;  
  
        if (id != nt - 1) {  
            border = V[limitR+1];  
        }  
  
        #pragma omp barrier  
    }  
}
```



A better solution (2/2)

```
    for (i = limitL; i < limitR; i++) {  
        V[i] = (V[i] + V[i+1]) / 2.0;  
    }  
  
    if (id != nt - 1) {  
        V[limitR] = (V[limitR] + border) / 2.0;  
    }  
}
```



What happens in this case?

```
#define N 1000
```

```
void testParallelization() {  
    double V[N];  
    int i;  
  
    for (i = 1; i < N; i++) {  
        V[i] = (V[i-1] + V[i]) / 2.0;  
    }  
}
```



Functions of the Run-Time Library



Run-Time Library Routines

- ▶ Prototypes of functions are declared in `<omp.h>`
 - ▶ Furthermore definitions of data types
 - ▶ “`omp_lock_t`”
 - ▶ *To declare variables for mutual exclusion*
 - ▶ “`omp_nest_lock_t`”
 - ▶ *To declare variables for mutual exclusion that support nesting*
 - ▶ “`omp_sched_t`”
 - ▶ *To declare variables that control how iterations of a loop are distributed among threads*



Number of threads and thread identification

- ▶ `void omp_set_num_threads(int num_threads);`
 - ▶ Defines the number of threads to be created in following “parallel” constructs
 - ▶ *If these don't use the “num_threads” clause*
- ▶ `int omp_get_num_threads(void);`
 - ▶ Returns the number of threads that compose the current team
- ▶ `int omp_get_thread_num(void);`
 - ▶ Returns the ID of the thread within a team
 - ▶ IDs are in the range from 0 up to and including `omp_get_num_threads() - 1`
- ▶ `int omp_get_max_threads(void);`
 - ▶ Returns the maximum allowed number of threads to be created in a “parallel” construct



Dynamic adjustment and nesting of parallelism

- ▶ `void omp_set_dynamic(int dynamic_threads);`
 - ▶ Enables or disables dynamic adjustment of the number of threads available for the execution of subsequent parallel regions
 - ▶ *Sets the ICV dyn-var to either true or false*
- ▶ `int omp_get_dynamic(void);`
 - ▶ Returns the current value of the ICV *dyn-var*
- ▶ `void omp_set_nested(int nested);`
 - ▶ Enables or disables nested parallelism
 - ▶ *Sets the ICV nest-var to either true or false*
- ▶ `int omp_get_nested(void);`
 - ▶ Returns the current value of the ICV *nest-var*



Retrieving information about nested parallelism

- ▶ `void omp_set_max_active_levels(int max_levels);`
 - ▶ Sets the maximum allowed depth for creating nested parallelism
- ▶ `int omp_get_max_active_levels(void);`
 - ▶ Returns the maximum allowed depth for creating nested parallelism
- ▶ `int omp_get_level(void);`
 - ▶ Returns the number of nested “parallel” constructs that the calling task has encountered
 - ▶ *Constructs might be active or inactive*
- ▶ `int omp_get_active_level(void);`
 - ▶ Returns the number of nested “parallel” constructs that the calling task has encountered
 - ▶ *Counts only active levels*



Retrieving information about parallel constructs

- ▶ `int omp_in_parallel(void);`
 - ▶ Returns true if the routine is called within a parallel region
 - ▶ Otherwise it returns false
- ▶ `int omp_get_team_size(int level);`
 - ▶ Returns the number of threads that compose a team at level “*level*” of nested parallel regions
- ▶ `int omp_get_ancestor_thread_num(int level);`
 - ▶ Returns the ID of the ancestor thread at level “*level*” of nested parallel regions



Other routines

- ▶ `void omp_set_schedule(omp_sched_t kind, int modifier);`
 - ▶ Sets the algorithm for distributing work among threads
 - ▶ *static, dynamic, guided: modifier defines the size of chunk*
 - ▶ *auto: modifier is ignored*
- ▶ `void omp_get_schedule(omp_sched_t *kind, int *modifier);`
 - ▶ Returns the currently used algorithm for distributing work among threads
- ▶ `int omp_get_num_procs(void);`
 - ▶ Returns the number of processors available for executing the application
- ▶ `int omp_get_thread_limit(void);`
 - ▶ Returns the maximum number of threads that an OpenMP application is allowed to create



Lock routines

- ▶ The Run-Time Library includes a set of routines for handling mutual exclusion and synchronization
 - ▶ They operate on synchronization variables
- ▶ Allowed states of synchronization variables
 - ▶ Uninitialized
 - ▶ Unlocked
 - ▶ Locked



Lock routines

- ▶ There are two types of synchronization variables
 - ▶ Simple locks
 - ▶ *If a thread locks a variable, the variable cannot be locked by any other thread*
 - ▶ Nestable locks
 - ▶ *If a thread locks a variable, the variable can be locked again multiple times by the same thread*
 - ▶ *But not by another thread*



Initialization of synchronization variables

- ▶ `void omp_init_lock(omp_lock_t *lock);`
- ▶ `void omp_init_nest_lock(omp_nest_lock_t *lock);`



Destruction of synchronization variables

- ▶ `void omp_destroy_lock(omp_lock_t *lock);`
- ▶ `void omp_destroy_nest_lock(omp_nest_lock_t *lock);`
- ▶ These routines ensure that the synchronization variable is set to the “Uninitialized” state



Locking of synchronization variables

- ▶ `void omp_set_lock(omp_lock_t *lock);`
- ▶ `void omp_set_nest_lock(omp_nest_lock_t *lock);`
- ▶ If the case of a simple synchronization variable
 - ▶ The variable becomes “Locked” only if it previously was “Unlocked”
- ▶ If the case of a nestable synchronization variable
 - ▶ The variable becomes “Locked” if it previously was “Unlocked”
 - ▶ The variable remains “Locked” if it previously was “Locked” and the routine is executed by the same task
 - ▶ *Nesting level is increased by one*



Unlocking of synchronization variables

- ▶ `void omp_unset_lock(omp_lock_t *lock);`
- ▶ `void omp_unset_nest_lock(omp_nest_lock_t *lock);`
- ▶ If the case of a simple synchronization variable
 - ▶ The variable becomes “Unlocked”
- ▶ If the case of a nestable synchronization variable
 - ▶ Nesting level is decreased by one
 - ▶ If it reaches zero, the variable becomes “Unlocked”



Trying to lock and testing

- ▶ `int omp_test_lock(omp_lock_t *lock);`
- ▶ `int omp_test_nest_lock(omp_nest_lock_t *lock);`
- ▶ These routines attempt to set an OpenMP lock
 - ▶ They do not suspend execution of the task executing the routine
 - ▶ *omp_test_lock(): Returns “true” if the call actually locks the variable, otherwise “false”*
 - ▶ *omp_test_nest_lock(): Returns the current level of nesting if the call actually locks the variable, otherwise 0*



Example

```
#include <omp.h>

typedef struct {
    int          a, b;
    omp_nest_lock_t lck;
} pair;

int work1();
int work2();
int work3();

void incr_a(pair *p, int a)
{
    /* Called only from incr_pair, no lock. */
    p->a += a;
}

void incr_b(pair *p, int b)
{
    /* Called both from incr_pair and
    elsewhere, */
    /* so need a nestable lock. */
    omp_set_nest_lock(&p->lck);
    p->b += b;
    omp_unset_nest_lock(&p->lck);
}
```

```
void incr_pair(pair *p, int a, int b)
{
    omp_set_nest_lock(&p->lck);
    incr_a(p, a);
    incr_b(p, b);
    omp_unset_nest_lock(&p->lck);
}

void a45(pair *p)
{
    #pragma omp parallel sections
    {
        #pragma omp section
        incr_pair(p, work1(), work2());

        #pragma omp section
        incr_b(p, work3());
    }
}
```



Timing routines

- ▶ `double omp_get_wtime(void);`
 - ▶ Returns elapsed wall clock time in seconds
- ▶ `double omp_get_wtick(void);`
 - ▶ Returns the precision of the timer used by `omp_get_wtime()`



Environment variables



Environment variables

- ▶ They affect the values of ICVs
- ▶ Changes to environment variables after execution of an OpenMP application has started are ignored
 - ▶ Even if the changes are performed by the application itself



OMP_SCHEDULE

- ▶ Ελέγχει τον τρόπο διαμοίρασης επαναλήψεων σε νήματα και το μέγεθος του chunk
- ▶ Η τιμή της μεταβλητής έχει την μορφή
 type[,chunk]
- ▶ “type” μπορεί να είναι:
 - ▶ static, dynamic, guided, auto
- ▶ “chunk” είναι προαιρετικό
 - ▶ Καθορίζει το μέγεθος κάθε chunk
- ▶ Παράδειγμα χρήσης
 - ▶ export OMP_SCHEDULE=“guided,4”
 - ▶ export OMP_SCHEDULE=“dynamic”



OMP_NUM_THREADS

- ▶ Controls the number of threads that will be created in a “parallel” construct
- ▶ The value of the variable must be a positive integer
- ▶ Usage example
 - ▶ `export OMP_NUM_THREADS=16`



OMP_DYNAMIC

- ▶ controls dynamic adjustment of the number of threads to use for executing parallel regions
 - ▶ The value of the variable must be either “true” or “false”
 - ▶ If “true”
 - ▶ *The OpenMP implementation may adjust the number of threads to use for executing parallel regions in order to optimize the use of system resources*
- ▶ Usage example
 - ▶ `export OMP_DYNAMIC=true`



OMP_NESTED

- ▶ Controls whether actual creation of nested parallelism is allowed
 - ▶ The value of the variable must be either “true” or “false”
- ▶ Usage example
 - ▶ `export OMP_NESTED=true`



OMP_STACKSIZE

- ▶ Controls the size of the stack for threads created by the OpenMP implementation
 - ▶ The value of the variable can be:
size | size**B** | size**K** | size**M** | size**G**
 - ▶ If no suffix is used, the default size is in Kbytes
- ▶ Usage examples
 - ▶ export OMP_STACKSIZE=2000500B
 - ▶ export OMP_STACKSIZE=3000k
 - ▶ export OMP_STACKSIZE="10 M"
 - ▶ export OMP_STACKSIZE=20000



OMP_WAIT_POLICY

- ▶ Provides a hint to an OpenMP implementation about the desired behavior of waiting threads
 - ▶ The value of the variable can be either “ACTIVE” or “PASSIVE”
 - ▶ “ACTIVE”: *Specifies that waiting threads should mostly be active, consuming processor cycles, while waiting.*
 - ▶ “PASSIVE”: *Specifies that waiting threads should mostly be passive, not consuming processor cycles, while waiting. For example, an OpenMP implementation may make waiting threads yield the processor to other threads or go to sleep.*
- ▶ Usage examples
 - ▶ `export OMP_WAIT_POLICY=ACTIVE`
 - ▶ `export OMP_WAIT_POLICY=active`
 - ▶ `export OMP_WAIT_POLICY=PASSIVE`
 - ▶ `export OMP_WAIT_POLICY=passive`



OMP_MAX_ACTIVE_LEVEL

- ▶ controls the maximum number of nested active parallel regions
 - ▶ The value of the variable must be a positive integer
- ▶ Usage example
 - ▶ `export OMP_MAX_ACTIVE_LEVEL=4`



OMP_THREAD_LIMIT

- ▶ Sets the maximum number of OpenMP threads to use in a contention group
 - ▶ Contention group: An initial thread and its descendent threads
 - ▶ The value of the variable must be a positive integer
- ▶ Usage example
 - ▶ `export OMP_THREAD_LIMIT=200`



Advanced features of OpenMP



The “task” struct

```
#pragma omp task [clause [[,] clause]...] new-line  
structured-block
```

- ▶ “clause” can be one or more of the following:
 - ▶ if (scalar-expression)
 - ▶ untied
 - ▶ default(shared | none)
 - ▶ private(list)
 - ▶ firstprivate(list)
 - ▶ shared(list)



Description

- ▶ A new task is created
 - ▶ From the code of the “*structured-block*”
- ▶ Execution of the task can:
 - ▶ Take place immediately
 - ▶ *On the thread that encountered the “task” struct*
 - ▶ Be postponed
 - ▶ *Any thread of the team can execute the task*
- ▶ It is possible to create nested tasks
 - ▶ However, the newly created task is not part of the task that created it



Description

- ▶ When an “if” clause exists
 - ▶ If “*scalar-expression*” evaluates to false
 - ▶ *The thread that encounters the “task” struct immediately suspends its execution*
 - ▶ *The new task immediately starts its execution*
 - ▶ *The task that was suspended cannot resume execution until the new task finishes*



Description

- ▶ A thread can suspend execution of a task
 - ▶ When it reaches a scheduling point
- ▶ A suspended task will continue its execution at a later point in time on the same thread that suspended it
 - ▶ Except if the “untied” clause has been specified
 - ▶ *Any thread of the team is allowed to select the suspended task and resume its execution*



Tree traversal (1/3)

```
struct node {
    /*
     * Here exist elements of the struct that
     * represent the data that each node stores.
     */
    struct node *left; /* NULL if there is no left child. */
    struct node *right; /* NULL if there is no right child. */
};

/*****/

struct node *build_tree() {
    struct node *root; /* Root node of the tree. */
    /*
     * Build a tree.
     * Return the root node to the caller.
     */

    return(root);
}
```



Tree traversal (2/3)

```
void process_node(struct node *curNode) {
    /*
     * Do cool stuff here with the data of the node.
     */
}

/*****

void traverse(struct node *p) {
    if (p->left != NULL) {
        #pragma omp task /* p is firstprivate by default */
        traverse(p->left);
    }

    if (p->right != NULL) {
        #pragma omp task /* p is firstprivate by default */
        traverse(p->right);
    }

    process_node(p);
}
```



Tree traversal (3/3)

```
void main(int argc, char *argv[]) {
    struct node *root;

    root = build_tree();

    #pragma omp parallel
    {
        #pragma omp single
        {
            traverse(root);
        }
    }

    return(0);
} /* main() ends here */
```




The “taskwait” construct

`#pragma omp taskwait new-line`

- ▶ Forces a task that has created more tasks to wait for the completion of all of them



Addition of data in a tree (1/3)

```
struct node {
    int      nodeData;
    struct node *left; /* NULL if there is no left child. */
    struct node *right; /* NULL if there is no right child. */
};

/*****

struct node *build_tree() {
    struct node *root; /* Root node of the tree. */
    /*
    * Build a tree.
    * Return the root node to the caller.
    */

    return(root);
}
```



Addition of data in a tree (2/3)

```
int traverse(struct node *p) {
    int localSum, sumFromLeft = 0, sumFromRight = 0;

    if (p->left != NULL) {
        #pragma omp task /* p is firstprivate by default */
        sumFromLeft = traverse(p->left);
    }

    if (p->right != NULL) {
        #pragma omp task /* p is firstprivate by default */
        sumFromRight = traverse(p->right);
    }

    #pragma omp taskwait

    localSum = sumFromLeft + sumFromRight + p->nodeData;

    return(localSum);
}
```



Addition of data in a tree (3/3)

```
void main(int argc, char *argv[]) {
    struct node *root;
    int      sum;

    root = build_tree();

    #pragma omp parallel
    {
        #pragma omp single
        {
            sum = traverse(root);
        }
    }

    printf("Sum = %d\n", sum);

    return(0);
} /* main() ends here */
```



The “taskgroup” construct

`#pragma omp taskgroup new-line`
`structured-block`

- ▶ Forces a task that has created more tasks to wait for the completion of all of them
 - ▶ As well as the completion of all tasks that have been created by its child tasks
 - ▶ OpenMP ≥ 4.0
- ▶ Notice: The “taskwait” construct waits for completion only of its immediate “child-tasks” that it created



THANK YOU FOR YOUR ATTENTION

www.prace-ri.eu