OpenMP Programming Workshop @LRZ

The OpenMP Enabling HPC since 1997 Common Core and Beyond

Manuel Arenaz | February 11-13, 2020

©Appentra Solutions S.L.

Agenda

8:30 - 9:00	*Setup and welcome participants*
9:00 - 9:15	Overview
9:15 - 10:30	**The OpenMP Common Core** **Decomposing code into patterns for parallelization** **Using Parallelware Trainer: A walk-through with PI example**
10:30 - 11:00	Coffee
11:00 - 12:40	Practicals: Examples codes PI, MANDELBROT, HEAT and LULESHmk Worksheet: Parallelizing PI and LULESHmk with OpenMP
12:40 - 13:00	*Close*

Why use patterns to parallelize code?

- **The OpenACC Application Programming Interface. Version 2.7 (November 2018)** 🔗
  - "does **not describe automatic detection of parallel regions or automatic offloading of regions of code to an accelerator by a compiler or other tool**."
  - "if one thread updates a memory location and another reads the same location, or two threads store a value to the same location, **the hardware may not guarantee the same result** for each execution."
  - "it is (...) **possible to write a compute region that produces inconsistent numerical results**."
  - "**Programmers need to be very careful that the program uses appropriate synchronization** to ensure that an assignment or modification by a thread on any device to data in shared memory is complete and available before that data is used by another thread on the same or another device."
- **Programmers are responsible for making good use of OpenACC**
- **Decomposition of codes into patterns**
  - Helps to make good use of OpenACC and OpenMP
  - Speeds up the parallelization process
  - Is more likely to result in good performance

Accelerating code with OpenMP/OpenACC

Profile & identify hotspots	Identify hotspots
Analyze for parallelism	Analyze loops ○ Understand code components ○ What patterns are present?
Implement parallel code	Implement parallelism by adding directives
Compare serial and parallel performance	Benchmark performance
Optimize parallel code	Optimize ○ Improve data locality ○ Minimize data transfers

Decomposing your code into components

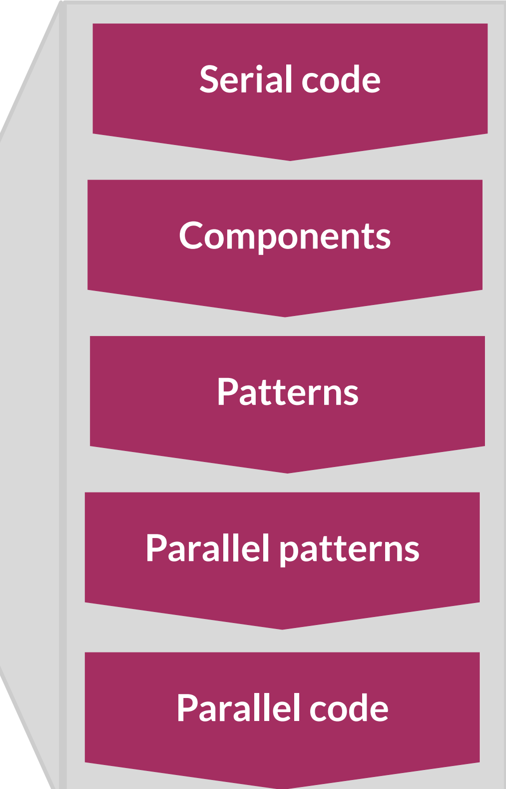
Profile & identify hotspots

Analyze for parallelism

Implement parallel code

Compare serial and parallel performance

Optimize parallel code



How does it fit into the classical parallelization workflow?

High-productivity approach independent of OpenMP/OpenACC

Decomposing your code into components

Profile & identify hotspots

Analyze for parallelism

Implement parallel code

Compare serial and parallel performance

Optimize parallel code

Serial Code

Components

Patterns

Parallel patterns

Parallel Code

Scientific components
(eg. MATMUL, FFT)

Code components or code patterns
(eg. REDUCTION)

Scientific components are typically available through highly-optimized libraries, but code components must be addressed by the programmer.

Decomposing your code into components

Step 1: **Use your profiling to**

- Identify calls, routines, functions or loops that consume most of the runtime

Step 2: **For each routine contained in an external library**

- Scientific components: kernels available as external libraries, including but not limited to dense/sparse linear algebra and spectral methods.
- Consider using a highly optimized version of the routine available in the target platform

Step 3: **For each routine coded by the programmer that matches a routine contained in external library**

- Consider replacing the corresponding routines with highly-optimized version in your platform

Step 4: **For the remaining user-defined routines**

- Understand the code patterns you have in your code and use them as a guide for parallelization

Types of code patterns

Profile & identify hotspots

Analyze for parallelism

Implement parallel code

Compare serial and parallel performance

Optimize parallel code

Serial Code

Components

Patterns

Parallel patterns

Parallel Code

Computation patterns

Memory patterns

Flow patterns

Types of code patterns

Serial code

Components

Pattern

Parallel pattern

Parallel code

- Computation patterns
- Memory patterns
- Flow patterns

Computation Patterns

parallel forall

```
for (j=0; j<n; j++ ) {
   A[j] = B[j];
}
```

parallel scalar reduction

```
for (j=0; j<n; j++ ) {
   A += B[j];
}
```

parallel sparse reduction

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

parallel sparse forall

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

Why using computation patterns?

1: **Computation patterns enable to ensure correct variable management in the parallel code**

- Each pattern has one output variable that is computed in the code.
- The pattern dictates the correct data scoping of the output variable (e.g. shared, private, reduction).

2: **Computation patterns provide algorithmic rules to re-code sequential code into a parallel-equivalent code**

- Patterns provide information about the type of computations that are associated with a variable of the code. And this type of computations dictates what codes can be parallelized (e.g. reduction).

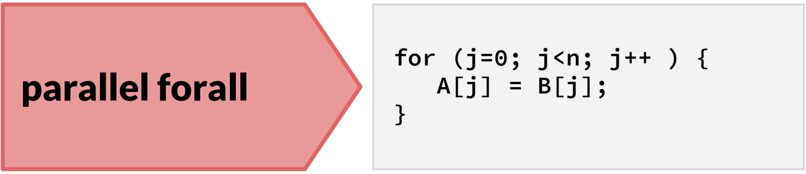
3: **Computation patterns enable to code parallel versions for several standards and platforms**

- Each pattern provides code rewriting rules for OpenMP/OpenACC and CPU/GPU.

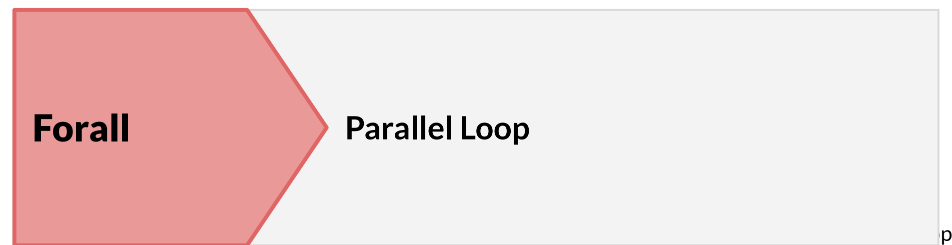
Forall

Understanding the sequential code

- A loop that updates the elements of an array.
- Each iteration updates a different element of the array.
- The result of computing this pattern is an array that is the "*output variable*".



Identifying opportunities for parallelization



Scalar reduction

Understanding the sequential code

- Combine multiple values into one single element (the scalar reduction variable) by applying an associative, commutative operator.
- Most frequently in a loop
- The result of computing this pattern is a scalar that is the “reduction variable”.

parallel scalar reduction

```
for (j=0; j<n; j++ ) {
    A += B[j];
}
```

Identifying opportunities for parallelization

Scalar reduction

Parallel Loop w/ Built-in reduction
Parallel Loop w/ Atomic
Parallel Loop w/ Explicit Privatization

Sparse reduction

Understanding the sequential code

- A sparse or irregular reduction combines a set of values from a subset of the elements of a vector or array with an associative, commutative operator.
- The set of array elements used cannot be determined until runtime due to the use of subscript array to provide these values.
- The result of computing this pattern is an array that is the “reduction variable”.

parallel sparse reduction

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

Identifying opportunities for parallelization

Sparse reduction

Parallel Loop w/ Built-in reduction
Parallel Loop w/ Atomic
Parallel Loop w/ Explicit Privatization

Sparse forall

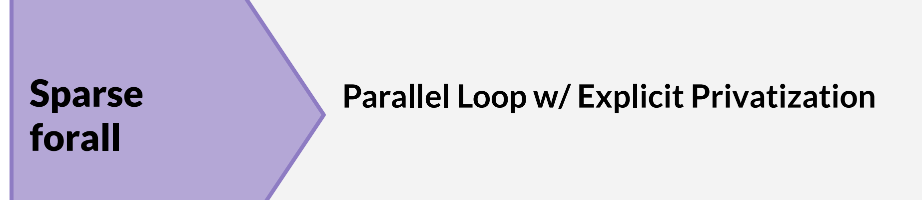
Understanding the sequential code

- A loop that updates the elements of an array.
- The set of array elements used cannot be determined until runtime due to the use of subscript array to provide these values.
- The result of computing this pattern is an array that is the "*output variable*".

parallel sparse forall

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

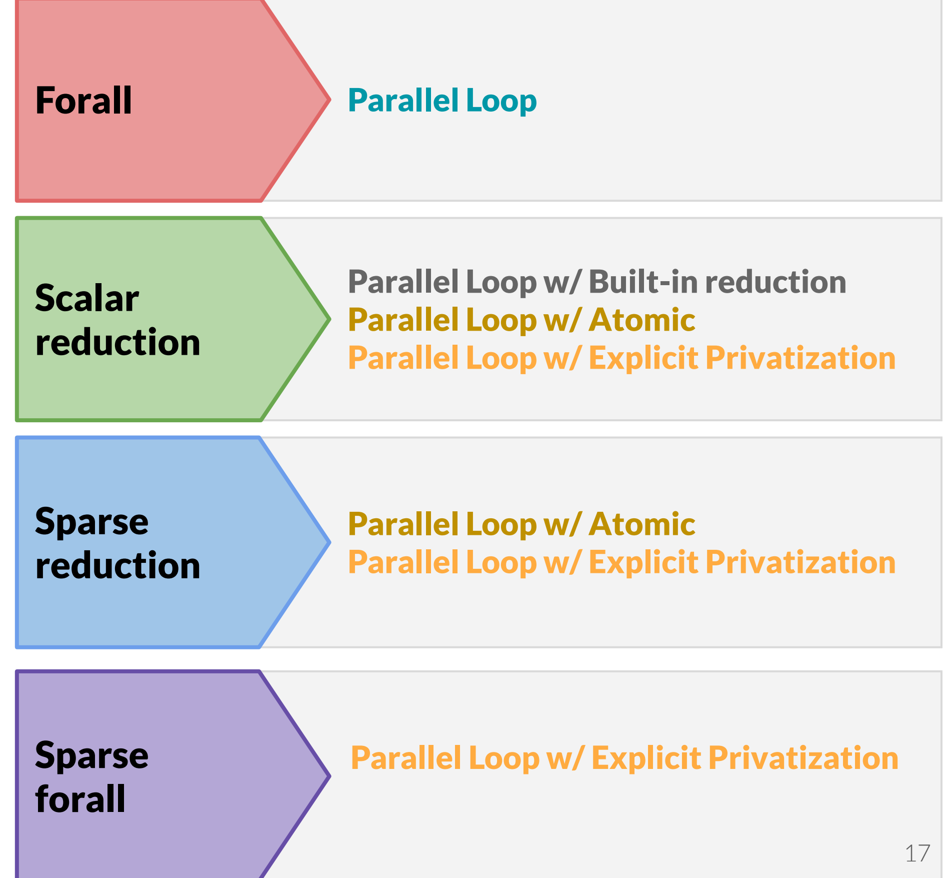
Identifying opportunities for parallelization



Parallelization strategies

Patterns and parallelization strategies

Parallelization Strategies



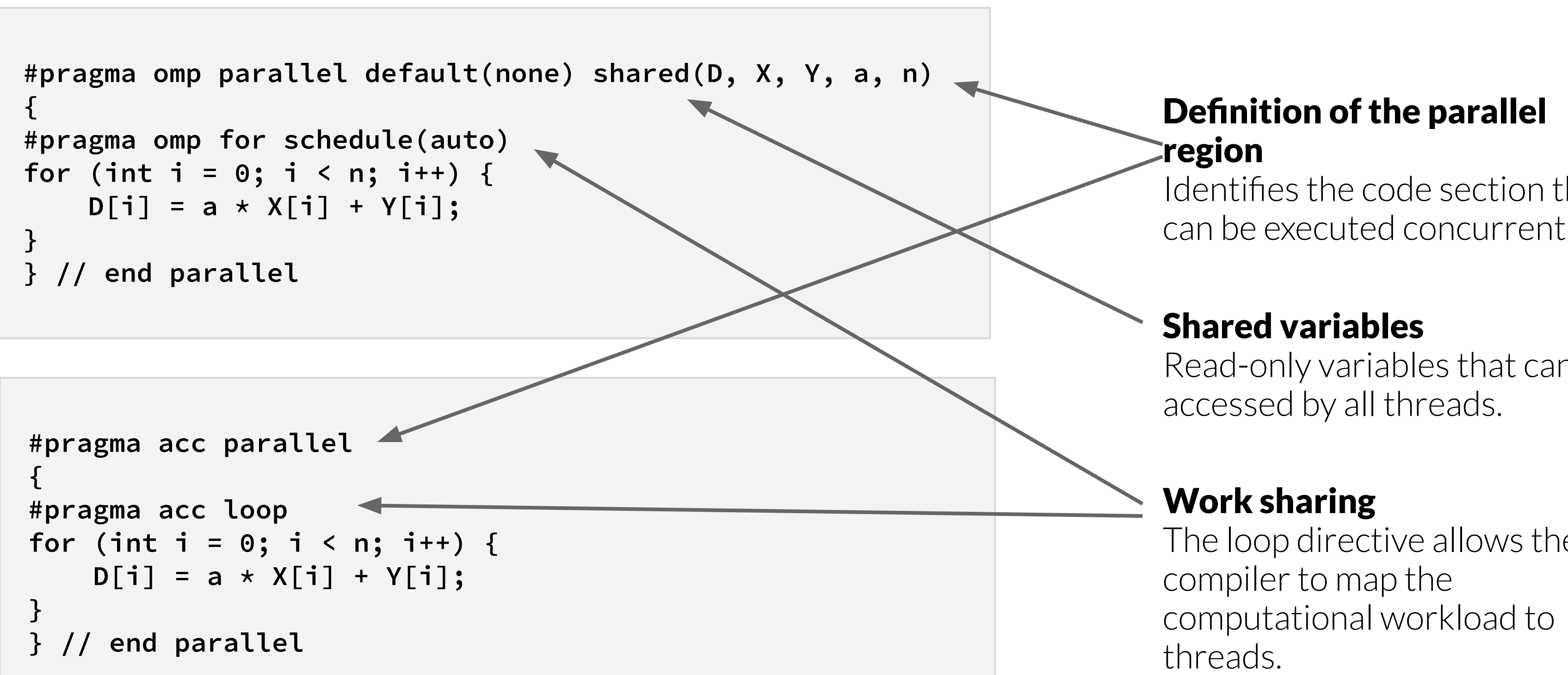
Mapping parallelization strategies to patterns

		Parallelization Strategy			
		Parallel Loop	Parallel Loop w/ Built-in reduction	Parallel Loop w/ Atomic	Parallel Loop w/ Explicit Privatization
Multithreading on CPU					
Parallel Pattern	Forall	✓			
	Scalar Reduction		✓	✓	✓
	Sparse Reduction			✓	✓
	Sparse forall				upcoming
Offloading to GPU					
Parallel Pattern	Forall	✓			
	Scalar Reduction		✓	✓	
	Sparse Reduction			✓	
	Sparse forall				

“Parallel Loop”

		Parallelization Strategy			
		Parallel Loop	Parallel Loop w/ Built-in reduction	Parallel Loop w/ Atomic	Parallel Loop w/ Explicit Privatization
Multithreading on CPU					
Parallel Pattern	Forall	✓			
	Scalar Reduction		✓	✓	✓
	Sparse Reduction			✓	✓
	Sparse forall				upcoming
Offloading to GPU					
Parallel Pattern	Forall	✓			
	Scalar Reduction		✓	✓	
	Sparse Reduction			✓	
	Sparse forall				

“Parallel Loop”: Implementation in OpenMP/OpenACC



Definition of the parallel region
Identifies the code section that can be executed concurrently.

Shared variables
Read-only variables that can be accessed by all threads.

Work sharing
The loop directive allows the compiler to map the computational workload to threads.

“Parallel Loop w/ Built-in Reduction”

		Parallelization Strategy			
		Parallel Loop	Parallel Loop w/ Built-in reduction	Parallel Loop w/ Atomic	Parallel Loop w/ Explicit Privatization
Multithreading on CPU					
Parallel Pattern	Forall	✓			
	Scalar Reduction		✓	✓	✓
	Sparse Reduction			✓	✓
	Sparse forall				upcoming
Offloading to GPU					
Parallel Pattern	Forall	✓			
	Scalar Reduction		✓	✓	
	Sparse Reduction			✓	
	Sparse forall				

“Parallel Loop w/ Built-in Reduction”: Implementation

```
double sum = 0.0;

#pragma omp parallel default(none) shared(N, sum)
{
#pragma omp for reduction(+: sum) schedule(auto)
for (int i = 0; i < N; i++) {
    double x = (i + 0.5) / N;
    sum += sqrt(1 - x * x);
}
} // end parallel
```

```
double sum = 0.0;

#pragma acc parallel
{
#pragma acc loop reduction(+: sum)
for (int i = 0; i < N; i++) {
    double x = (i + 0.5) / N;
    sum += sqrt(1 - x * x);
}
} // end parallel
```

Definition of the parallel region
Identifies the code section that can be executed concurrently.

Shared variables
Read-only variables that can be accessed by all threads.

Work sharing
The loop/for directive allows the compiler to map the computational workload to threads.

Reduction
Identifies the loop as a reduction, and identifies the subject of the reduction (i.e. sum) and the reduction operator (i.e. ‘+’)

“Parallel Loop w/ Atomic”

		Parallelization Strategy			
		Parallel Loop	Parallel Loop w/ Built-in reduction	Parallel Loop w/ Atomic	Parallel Loop w/ Explicit Privatization
Multithreading on CPU					
Parallel Pattern	Forall	✓			
	Scalar Reduction		✓	✓	✓
	Sparse Reduction			✓	✓
	Sparse forall				upcoming
Offloading to GPU					
Parallel Pattern	Forall	✓			
	Scalar Reduction		✓	✓	
	Sparse Reduction			✓	
	Sparse forall				

"Parallel Loop w/ Atomic": Implementation

Shared variable, *S*, is the 'reduction' variable. No private data.

Access to the variable *S*, is controlled by the 'atomic' directive: i.e. only one thread can read/write the variable at any one time.

In each atomic access of *S*, the thread adds part of the contribution to the total reduction value. In this instance, the reduction operation is an addition.

SHARED MEMORY

0 *S*

Thread 0	Thread 1	Thread 2
Private data	Private data	Private data
#atomic *S+= …* *#atomic* *S+= …* *#atomic* *S+= …* …	*#atomic* *S+= …* *#atomic* *S+= …* *#atomic* *S+= …* …	*#atomic* *S+= …* *#atomic* *S+= …* *#atomic* *S+= …* …

“Parallel Loop w/ Atomic”: Implementation

```
double sum = 0.0;
#pragma omp parallel default(none) shared(N, sum)
{
#pragma omp for schedule(auto)
for (int i = 0; i < N; i++) {
    double x = (i + 0.5) / N;
    #pragma omp atomic update
    sum += sqrt(1 - x * x);
}
} // end parallel
```

```
double sum = 0.0;

#pragma acc parallel
{
#pragma acc loop
for (int i = 0; i < N; i++) {
    double x = (i + 0.5) / N;
    #pragma acc atomic update
    sum += sqrt(1 - x * x);
}
} // end parallel
```

Definition of the parallel region
Identifies the code section that can be executed concurrently.

Shared variables
Read-only variables that can be accessed by all threads.

Work sharing
The loop directive allows the compiler to map the computational workload to threads.

Atomic update
Only one thread can read/write the variable at any one time.

“Parallel Loop w/ Explicit Privatization”

		Parallelization Strategy			
		Parallel Loop	Parallel Loop w/ Built-in reduction	Parallel Loop w/ Atomic	Parallel Loop w/ Explicit Privatization
Multithreading on CPU					
Parallel Pattern	Forall	✓			
	Scalar Reduction		✓	✓	✓
	Sparse Reduction			✓	✓
	Sparse forall				upcoming
Offloading to GPU					
Parallel Pattern	Forall	✓			
	Scalar Reduction		✓	✓	
	Sparse Reduction			✓	
	Sparse forall				

“Parallel Loop w/ Explicit Privatization”: Implementation

SHARED MEMORY

0 S

Thread 0 | **Thread 1** | **Thread 2**

Create private copies $S_0 \ldots S_{p-1}$ of the shared variable S. Initialize the private variables to 0.

Thread 0	Thread 1	Thread 2
Private data: 0 S_0	Private data: 0 S_1	Private data: 0 S_2

Each thread computes a partial sum using its private copy only. No synchronization with other threads.

Thread 0	Thread 1	Thread 2
S_0+= …	S_1+= …	S_2+= …
S_0+= …	S_1+= …	S_2+= …
S_0+= …	S_1+= …	S_2+= …
…	…	…

Each thread adds its partial sum to the global sum. Using *atomic* guarantees exclusive access to the reduction variable.

Thread 0	Thread 1	Thread 2
#atomic	*#atomic*	*#atomic*
S += S_0	S += S_1	S += S_2

“Parallel Loop w/ Explicit Privatization”: Implementation

Create private, local copies
Create thread-local copies of the reduction variable and initialize the local copies to 0.

Use atomic to contribute to global value
To complete the calculation each thread adds its contribution to the global shared using *atomic*.

```
double sum = 0.0;
#pragma omp parallel default(none) shared(N, sum)
{
// preamble
double sum_private = 0;
// end preamble
#pragma omp for schedule(auto)
for (int i = 0; i < N; i++) {
    double x = (i + 0.5) / N;
    sum_private += sqrt(1 - x * x);
}
// postamble
#pragma omp atomic update
sum += sum_private;
// end postamble
} // end parallel
```

```
#pragma omp parallel default(none) shared(col_ind, n, row_ptr, val, x, y)
{
// preamble
unsigned int y_length = 0 + n;
double *y_private = (double *) malloc(sizeof(double) * y_length);
for (int i = 0; i < y_length; ++i) {
  y_private[i] = 0;
}
// end preamble
#pragma omp for schedule(auto)
for (int i = 0; i < n; i++) {
    for (int k = row_ptr[i]; k < row_ptr[i + 1]; k++) {
        y_private[col_ind[k]] = y_private[col_ind[k]] + x[i] * val[k];
    }
}
// postamble
#pragma omp critical
for(int i = 0; i < y_length; ++i) {
  y[i] += y_private[i];
}
free(y_private);
// end postamble
} // end parallel
```

Explicit privatization
Each thread performs a thread-local computation on the private copy.

Mapping strategies to patterns for Tasking

		Parallelization Strategy			
		Parallel Loop	**Parallel Loop w/ Built-in reduction**	**Parallel Loop w/ Atomic**	**Parallel Loop w/ Explicit Privatization**
Fine-grain tasking on CPU (OpenMP 3.0 task/taskwait; OpenMP 4.5 taskloop -implementation dependent-)					
Parallel Pattern	Forall	✓			
	Scalar Reduction		upcoming	✓	upcoming
	Sparse Reduction			✓	upcoming
	Sparse forall				upcoming
Coarse-grain tasking on CPU (OpenMP 3.0: task/taskwait + loop stripmining; OpenMP 4.5 taskloop grainsize/numtasks)					
Parallel Pattern	Forall	upcoming			
	Scalar Reduction		upcoming	upcoming	upcoming
	Sparse Reduction			upcoming	upcoming
	Sparse forall				

“Parallel Loop w/ Atomic”: Impl. w/ Tasking

OpenMP 3.0: task/taskwait

```
double sum = 0.0;
#pragma omp parallel default(none) shared(N, sum)
#pragma omp master
{
for (int i = 0; i < N; i++) {
#pragma omp task shared(sum)
   {
    double x = (i + 0.5) / N;
    #pragma omp atomic update
    sum += sqrt(1 - x * x);
   }
}
#pragma omp taskwait
} // end parallel master
```

OpenMP 4.5: taskloop

```
double sum = 0.0;
#pragma omp parallel default(none) shared(N, sum)
#pragma omp single
{
    #pragma omp taskloop
    for (int i = 0; i < N; i++) {
        double x = (i + 0.5) / N;
        #pragma omp atomic update
        sum += sqrt(1 - x * x);
    }
} // end parallel
```

Parallelization strategies
Pros & Cons

Parallelization strategies for computation patterns

		Parallelization Strategy			
		Parallel Loop	Parallel Loop w/ Built-in reduction	Parallel Loop w/ Atomic	Parallel Loop w/ Explicit Privatization
Multithreading on CPU					
Parallel Pattern	Forall	✓			
	Scalar Reduction		✓	✓	✓
	Sparse Reduction			✓	✓
	Sparse forall				upcoming
Offloading to GPU					
Parallel Pattern	Forall	✓			
	Scalar Reduction		✓	✓	
	Sparse Reduction			✓	
	Sparse forall				

Parallelization strategies for computation patterns

		Parallelization Strategy			
		Parallel Loop	Parallel Loop w/ Built-in reduction	Parallel Loop w/ Atomic	Parallel Loop w/ Explicit Privatization
Fine-grain tasking on CPU (OpenMP 3.5 task/taskwait; OpenMP 4.5 taskloop -implementation dependent-)					
Parallel Pattern	Forall	✓			
	Scalar Reduction		upcoming	✓	upcoming
	Sparse Reduction			✓	upcoming
	Sparse forall				upcoming
Coarse-grain tasking on CPU (OpenMP 3.5: task/taskwait + loop stripmining; OpenMP 4.5 taskloop grainsize/numtasks)					
Parallel Pattern	Forall	upcoming			
	Scalar Reduction		upcoming	upcoming	upcoming
	Sparse Reduction			upcoming	upcoming
	Sparse forall				

Strategy	Pros	Cons
Parallel Loop	- Easy to implement - No synchronization overhead within the loop	- Limited applicability: only works when each loop iteration is entirely independent
Parallel Loop w/ Built-in Reduction	- Scales with threads/core counts, not the problem size - Offers speedup even for codes with low arithmetic intensity - Complexity handled by the compiler - Potential for highly optimized implementation (compiler/platform dependent)	- Can only be used for supported reduction operators
Parallel Loop w/ Atomic Protection	- Easy to understand - Provides speedup for codes with high arithmetic intensity - Solution for reduction patterns where operator is not supported by build-in reduction clause	- Synchronization overhead scales with the number of threads - Poor performance for codes with low arithmetic intensity
Parallel Loop w/ Explicit Privatization	- Possible to achieve speedup similar to Built-in Reductions - Programmer has full control of the parallel implementation	- Significant programmer effort - Not suitable for GPUs due to memory requirements