Parallelware Tool Workshop

Learning parallelization of real applications from the ground-up

Manuel Arenaz | October 17, 2019

Agenda

8:15 - 8:45	*Morning refreshment and coffee*
8:45 - 9:00	*Welcome and introductions*
9:00 - 9:30	Lecture 1: An introduction to OpenMP/OpenACC optimizations for CPUs/GPUs
9:30 - 10:15	**Lecture 2: Patterns to minimize data transfers, optimize memory usage and exploit massive parallelism**
10:15 - 10:30	Break
10:30 - 11:00	Lecture 3: Minimizing data transfers
11:00 - 11:30	Lecture 4: Optimizing memory usage
11:30 - 12:00	Lecture 5: Exploiting massive parallelism
12:00 - 13:00	Working lunch (hands-on activities)
13:00 - 14:00	Practical 5A: Parallelizing the calculation of HEAT
14:00 - 17:00	Hands-on time with your code
17:00 pm	*Close*

Use cases: Performance optimization on CPU/GPU

Use case #1: Minimizing data transfers

- **Code patterns: flow patterns (eg. convergence loop)**
- On GPUs: Transfer data from CPU to GPU and reuse it!
- On CPUs: Create threads and reuse them!

Use case #2: Optimizing memory usage

- **Code patterns: memory patterns (eg. data structure design)**
- On GPUs: Watch your data structure design as it may break your code!
- On CPUs: Hardware keeps memory consistency, so focus mostly on locality!

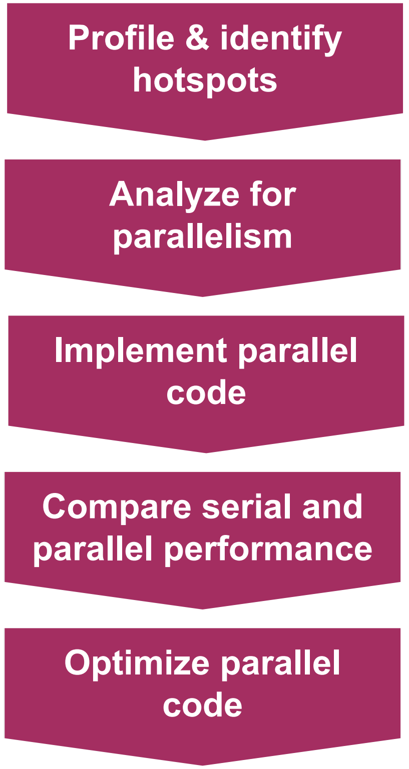
Use case #3: Exploiting massive parallelism

- **Code patterns: computation patterns (eg. collapsible nested loops)**
- On GPUs: Scale-up to thousands of threads!
- On CPUs: Limited number of threads, so not so important as on GPUs!

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



Identify hotspots

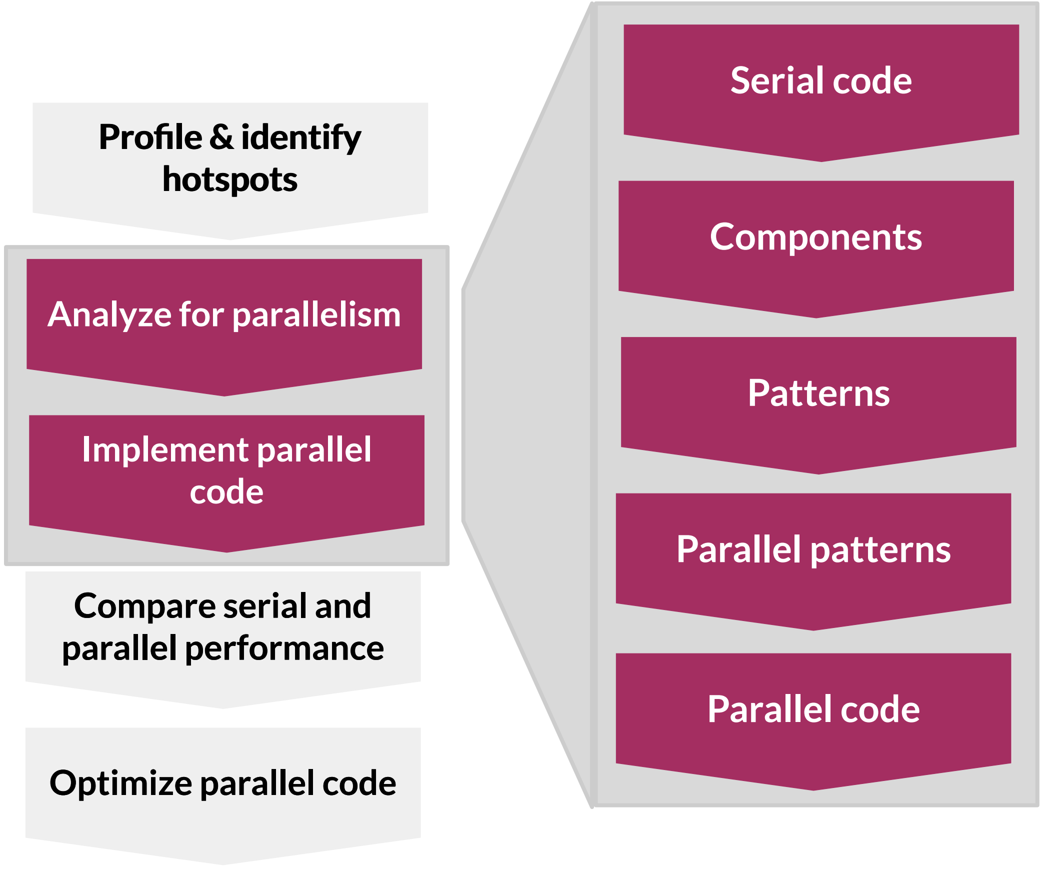
Analyze loops
- Understand code components
- What patterns are present?

Implement parallelism by adding directives

Benchmark performance

Optimize
- Improve data locality
- Minimize data transfers

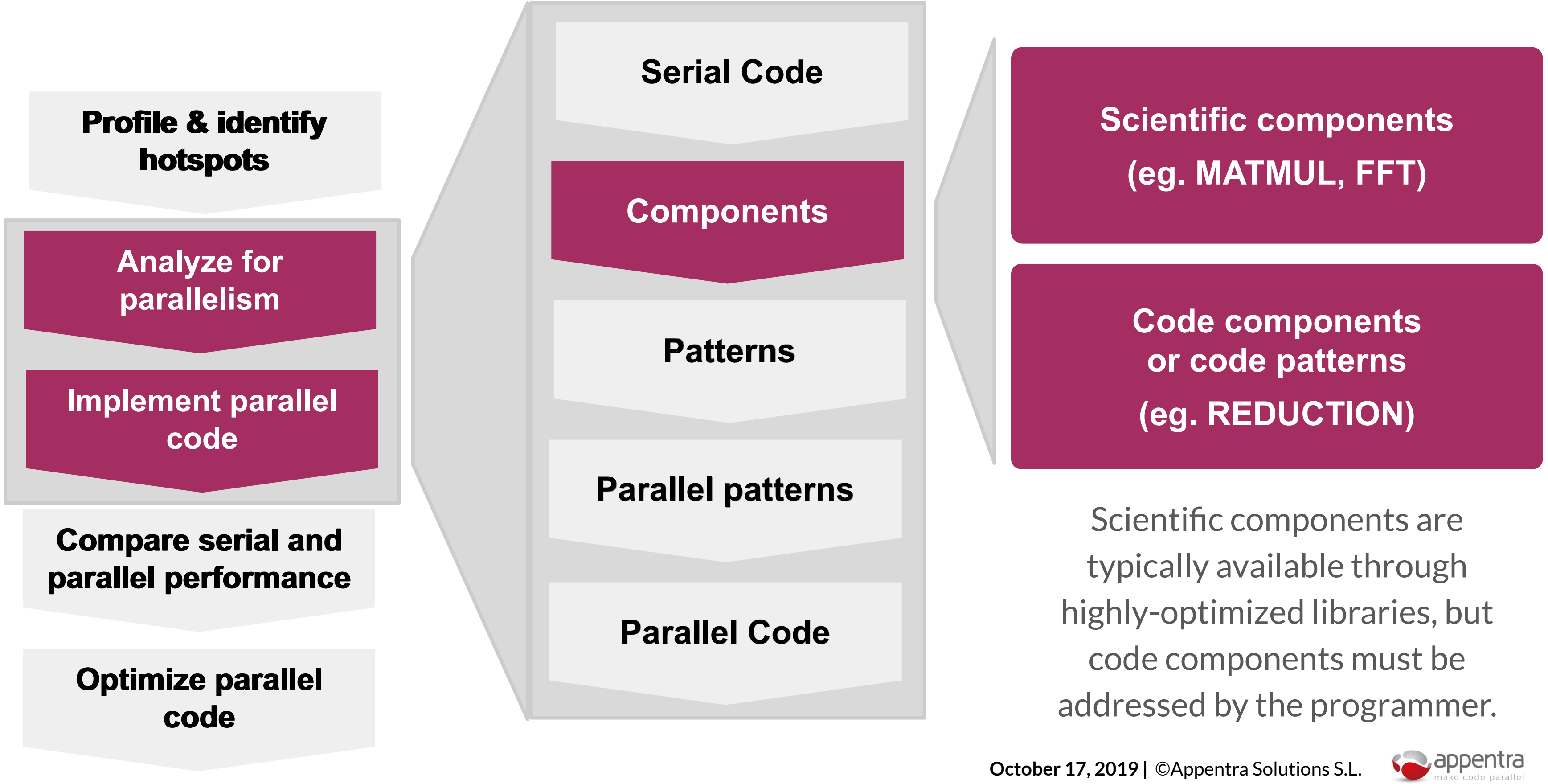
Decomposing your code into components



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

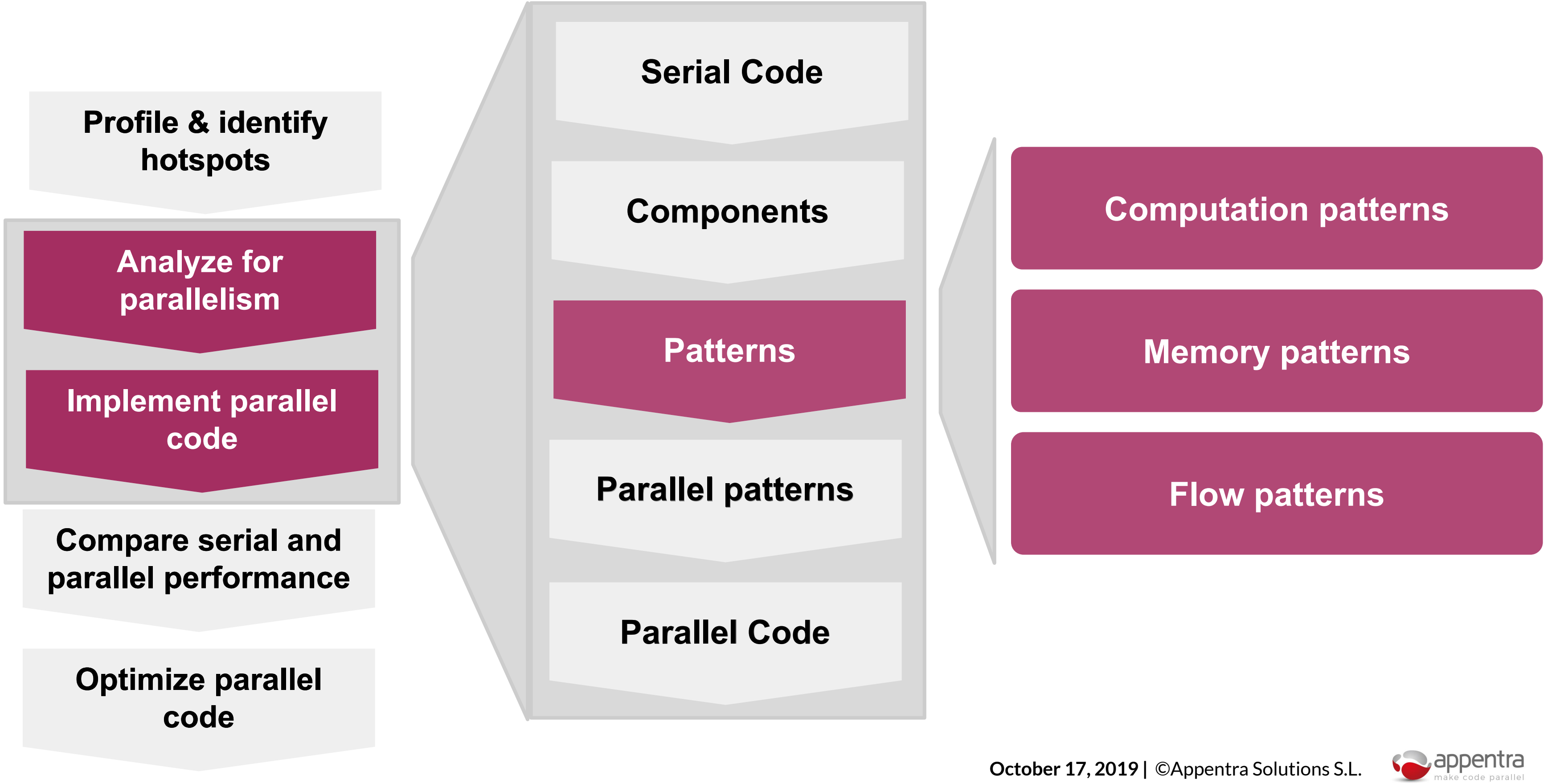
Step 3: For each routine coded by the programmer that matches a routine contained in external library

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

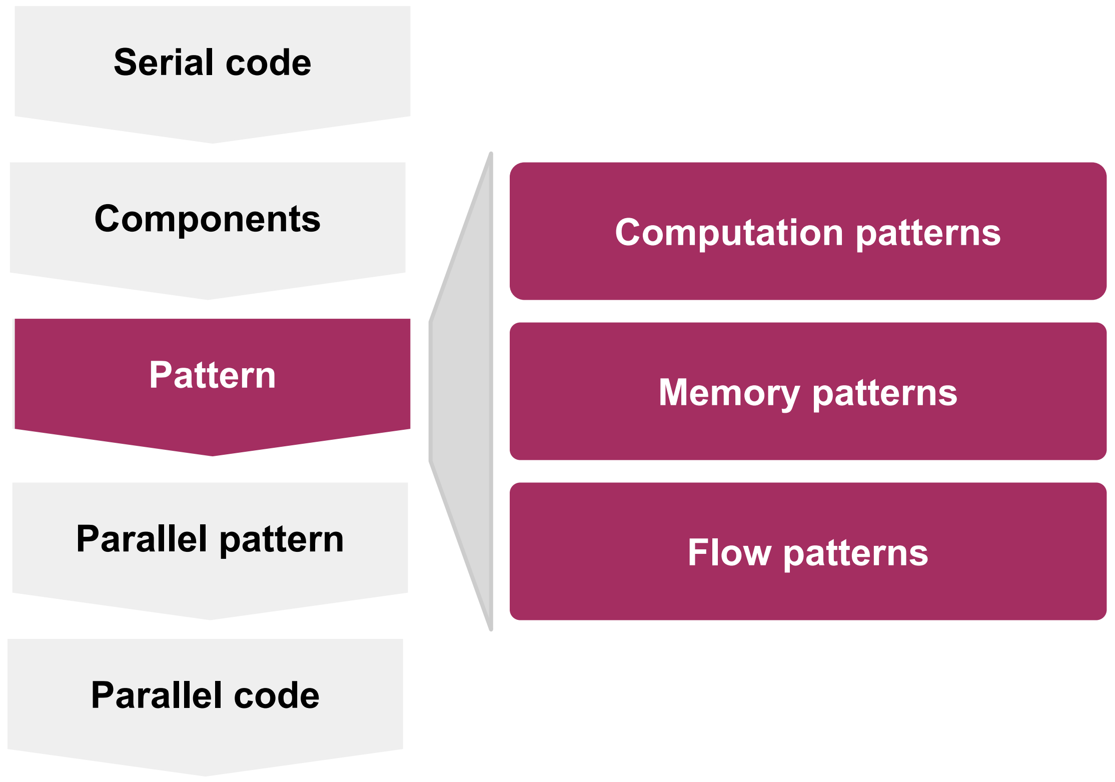
Step 4: For the remaining user-defined routines

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

Types of code patterns



Types of code 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*".

parallel forall

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

Identifying opportunities for parallelization

Forall

Parallel Loop

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

Sparse forall

Parallel Loop w/ Explicit Privatization

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

Memory Patterns

data structure design patterns

(e.g. array 1D, multi-dimensional array, array-of-structs/struct-of-arrays)

data access patterns

(e.g. linear, strided, irregular, stencil)

Why using memory patterns?

1: Memory patterns enable to exploit locality during the execution of parallel code

- By exploiting data locality in a parallel code, each thread will have very fast access to the data needed for the computations; and this is key for performance in modern hardware systems (e.g. stride one memory access).

2: Memory patterns provide rules to re-code the data structure of the variables

- The data structure of the variables dictates the memory layout of the data of our programs.
- It must be designed with memory layout in mind (e.g. AoS -Struct of Arrays- not AoS -Array of Structs).

3: Memory patterns enable the detection of errors/bugs in the parallel code

- Data structures typically allocate memory not only for the actual data, but also for auxiliary data structures that contain pointers to enable seamless access to the actual data.
- It may lead to incorrect memory accesses when data needs to be moved around different memory systems, as it is the case of moving data between CPU memory and GPU memory.

Flow Patterns

convergence loop

```
for(iter=0, err = tol; err >= tol && iter < iter_max; iter++){
  ...
}
```

propagation loop

```
for(iter=0; iter < iter_max; iter++){
  ...
}
```

Why using flow patterns?

1: Flow patterns provide a deeper understanding of the reuse of data during the execution of the parallel code

- Typically, scientific and engineering codes perform simulations over time where many program inputs are read-only data that does not change at run-time. In GPU programming it is recommended to transfer such data to GPU memory only once at the beginning of the program.

2: Flow patterns enable to the detection of loops that cannot be parallelized

- Time-step loops that dictate the progress in time during the execution of the code cannot be parallelized because, given an initial state of a variable, such variable is updated in each time-step using as inputs the values computed in the previous iteration(s). Thus, all the threads either on the CPU or on the GPU must go through all the time-steps and synchronize at the beginning/end of each time-step iteration.

Use cases: Performance optimization on CPU/GPU

Use case #1: Minimizing data transfers

- **Code patterns: flow patterns (eg. convergence loop)**
- On GPUs: Transfer data from CPU to GPU and reuse it!
- On CPUs: Create threads and reuse them!

Use case #2: Optimizing memory usage

- **Code patterns: memory patterns (eg. data structure design)**
- On GPUs: Watch your data structure design as it may break your code!
- On CPUs: Hardware keeps memory consistency, so focus mostly on locality!

Use case #3: Exploiting massive parallelism

- **Code patterns: computation patterns (eg. collapsible nested loops)**
- On GPUs: Scale-up to thousands of threads!
- On CPUs: Limited number of threads, so not so important as on GPUs!