Parallelware Tool Workshop

Learning parallelization of real applications from the ground-up

Manuel Arenaz | October 17, 2019

NERSC

appentra
make code parallel

OpenACC
More Science, Less Programming

OpenMP
Enabling HPC since 1997

Expected workshop learning outcomes

- **Why are you attending a course about parallel computing?**
  - You get “competitive advantage” by developing high-performance applications
- **In the introductory Parallelware workshop in June 2019...**
  - Learn a practical step-by-step approach to parallelization based on code patterns
  - Learn how to decompose real codes into code patterns: hydrodynamics CORAL LULESHmk
  - Focus on the hotspots of the code by using “computation patterns”
- **What is new in this new intermediate-level course in October 2019?**
  - First, analysis of the source code to decide how to parallelize it
    - Understanding of the code as a whole, not just the hotspots isolatedly
    - Cover wider set of patterns: computation patterns, memory patterns and flow patterns
  - Second, implementation of parallel version of the code for CPU and GPU
    - Learn best practices for parallel programming using OpenMP/OpenACC for CPUs and GPUs
    - Quickly develop parallel versions using OpenMP/OpenACC for CPU/GPU
    - Learn the commonalities and differences between both parallel computing environments

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: A wider set of code patterns: compute patterns, memory patterns and flow patterns
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 6A-6B: Parallelizing the calculation of HEAT and MATMUL
14:00 - 17:00	Hands-on time with your code and LULESHmk, inspired in the CORAL benchmark LULESH
17:00 pm	*Close*

Parallelware Tool Workshop

Learning parallelization of real applications from the ground-up

Manuel Arenaz | October 17, 2019

NERSC

appentra
make code parallel

OpenACC
More Science, Less Programming

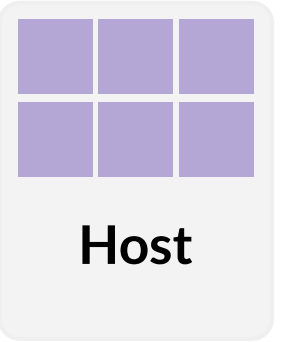
OpenMP
Enabling HPC since 1997

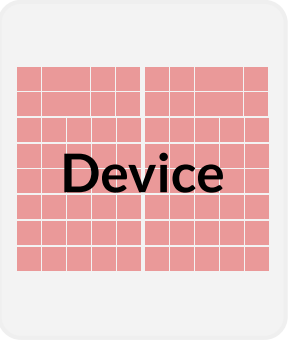
Agenda

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: A wider set of code patterns: compute patterns, memory patterns and flow patterns
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*

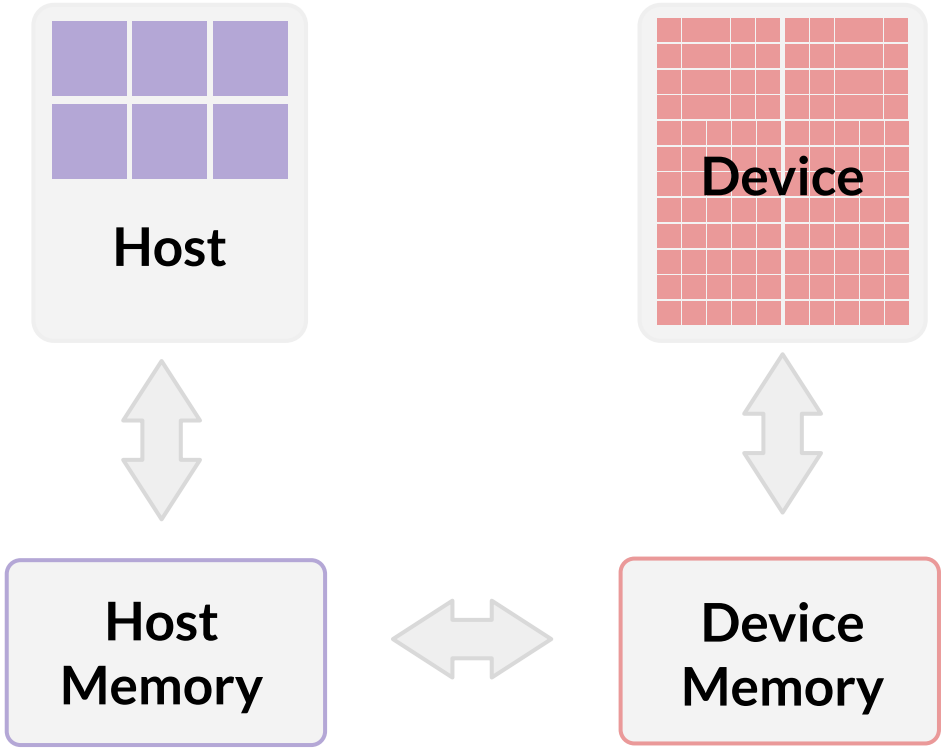
What are the differences between CPUs and GPUs?

- First, the number of cores available in the hardware
  - GPUs have many many more cores than CPUs
- Second, the grouping of the threads at the hardware level
  - In CPUs, the threads are not grouped and all the threads are executed at the same time
  - In GPUs, the threads are grouped and all the threads in a group are executed at the same time.
- Third, the complexity of the memory design
  - In CPUs, all the threads access to all the memory
  - In GPUs, there are constraints in the memory that can be accessed by the threads (e.g., cache, texture, scratchpad, global).
- Fourth, execution of instructions in vector mode
  - Both CPUs and GPUs exploit vector processing, although different “flavours” of it.





The GPU Accelerator Model



The GPU Execution Model

- Use of a host-driven execution model.
- Sequential code runs on a conventional processor.
- Computationally intensive parallel pieces of code (kernels) run on an accelerator such as a GPU.

- To maximize performance, high-performance applications generally conform to the following three rules of accelerator programming:
  - Transfer the data onto the device and keep it there.
  - Give the device enough work to do.
  - Focus on data reuse within the device(s) to avoid memory bandwidth bottlenecks

Why OpenMP/OpenACC for GPU programming?

- **GPUs have a reputation for being difficult to use because of programming**
  - OpenMP/OpenACC aim to change that!
  - OpenMP/OpenACC offer acceleration of scientific computing without significant programming effort.
- **OpenMP/OpenACC improve portability and readability of the code compared to other methods for using accelerators**
  - OpenMP/OpenACC codes use directives, which are added to the original code and can be safely ignored by compilers not supporting them.
- **Don't need to fully understand the specifics of the hardware you are using**
  - A limited understanding is still helpful to understand performance
- **Minimizes the need for code refactoring**
  - Though some refactoring may help with performance
- **Support for C/C++ and Fortran**
  - Note that our examples are in C! But the same methods apply to Fortran code

What are OpenMP/OpenACC?

- **Method for using multicore CPUs and GPUs**
  - Extension for C, C++ and Fortran

- **Uses compiler directives:**
  - Specify loops/regions to be offloaded to the GPU
  - Also runtime library functions and environment variables

- **Host/Accelerator programming model**
  - The CPU controls the accelerator
  - Uses the concepts of threads and tasks

- **OpenMP/OpenACC are focused on portability**

OpenMP/OpenACC Directive Syntax

- The **`omp/acc`** sentinel:
  - Tells the compiler that the text that follows is OpenMP/OpenACC, a directive

C and C++:

```
#pragma omp directive-name [clause [[,] clause]…]
#pragma acc directive-name [clause [[,] clause]…]
```

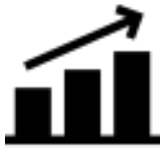
Fortran:

```
!$omp directive-name [clause [[,] clause]…]
!$acc directive-name [clause [[,] clause]…]
```

Available OpenMP/OpenACC compilers

	OpenMP Enabling HPC since 1997	OpenACC More Science, Less Programming
PGI 18.4	4.5	2.6
GCC 9	4.5	2.5
CCE 8.7	4.5	2.0
Clang 10	4.5	-
Intel Compiler 17	4.5	-

Benefits and Limitations of OpenMP/OpenACC



Benefits

- High-level platform independent
- Simple
- Portable

Limitations

- Cannot represent architectural specifics of devices without making the code less portable
- Some optimizations cannot be coded in these high-level APIs, and are only possible in lower-level APIs (e.g. CUDA, OpenCL)
- And sometimes this comes at a cost of performance

Challenge: Performance optimization on CPU/GPU

- **Typical optimizations for high-performance on the CPU**
  - Programming aware of low-level CPU features
    - Small number of threads available in the hardware
    - Instruction Set Architecture (ISA): eg. vector/simd instructions
    - Cache memory: space locality and time locality in cache lines in direct/associative caches
  - Code transformations for CPU-aware programming:
    - Loop transformations: fusion, fision, streap-mining, tiling/blocking
    - Data-layout transformations: array flattening, convert AoS to SoA, array padding

- **Typical optimizations for high-performance on the GPU**
  - Programming aware of low-level GPU features
    - Large number of threads available in the hardware
    - Warps are sets of threads that execute the same instruction, even doing nothing if needed
    - Complex memory system: scratch pad, texture memory, cache memory, global memory
  - Code transformations for GPU-aware programming:
    - Avoid thread divergence: prefer recomputation to conditional execution
    - Data-layout transformations: array flattening, convert AoS to SoA

Use cases: Performance optimization on CPU/GPU

Use case #1: Minimizing data transfers

- On GPUs: Transfer data from CPU to GPU and reuse it!
- On CPUs: Create threads and reuse them!

Use case #2: Optimizing memory usage

- 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

- On GPUs: Scale-up to thousands of threads!
- On CPUs: Limited number of threads, so not so important as on GPUs!

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!

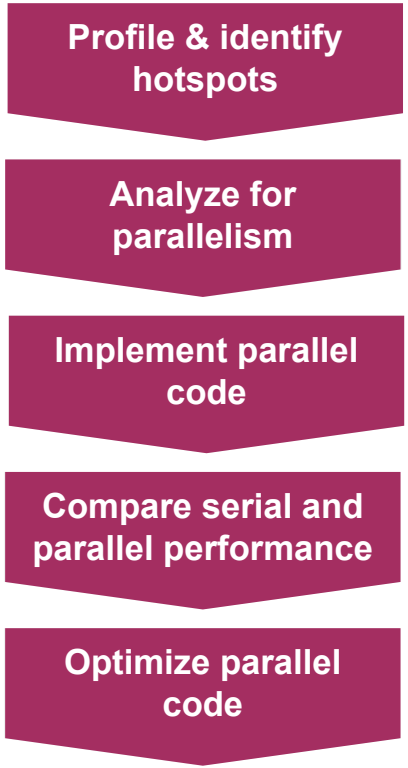
Use case #3: Exploiting massive parallelism

- **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?

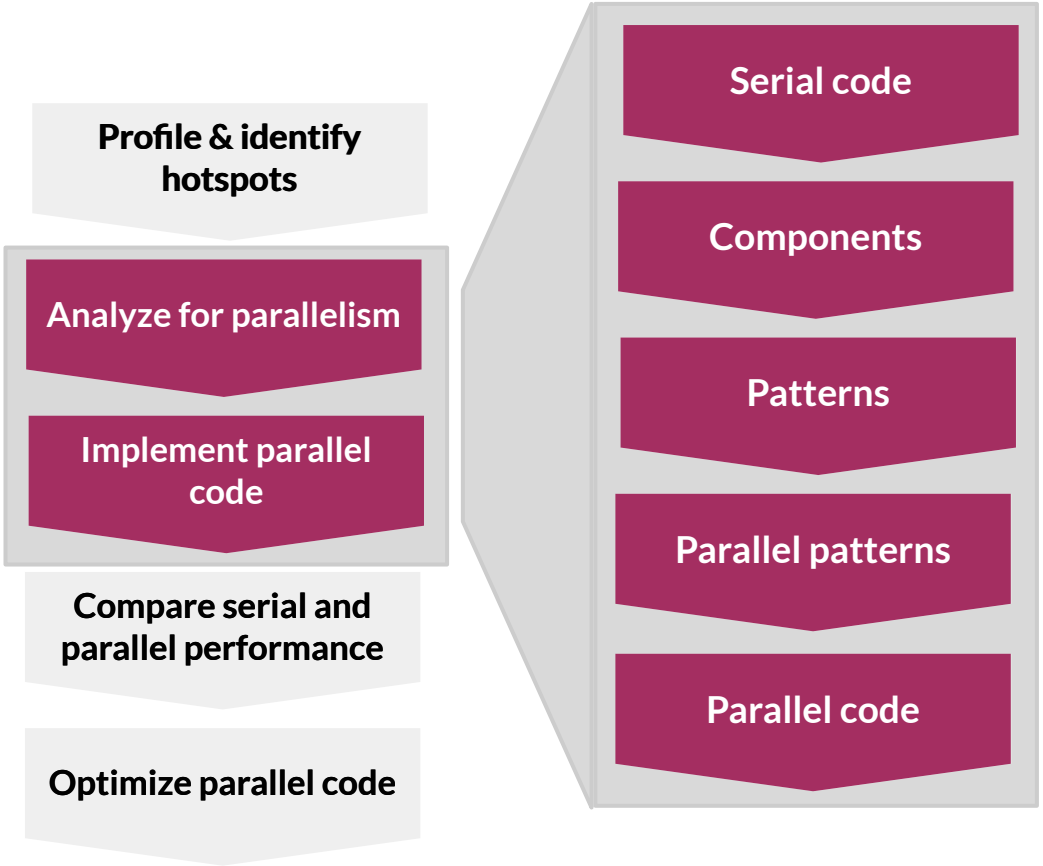
Implement parallelism by adding directives

Benchmark performance

Optimize

- Improve data locality
- Minimize data transfers

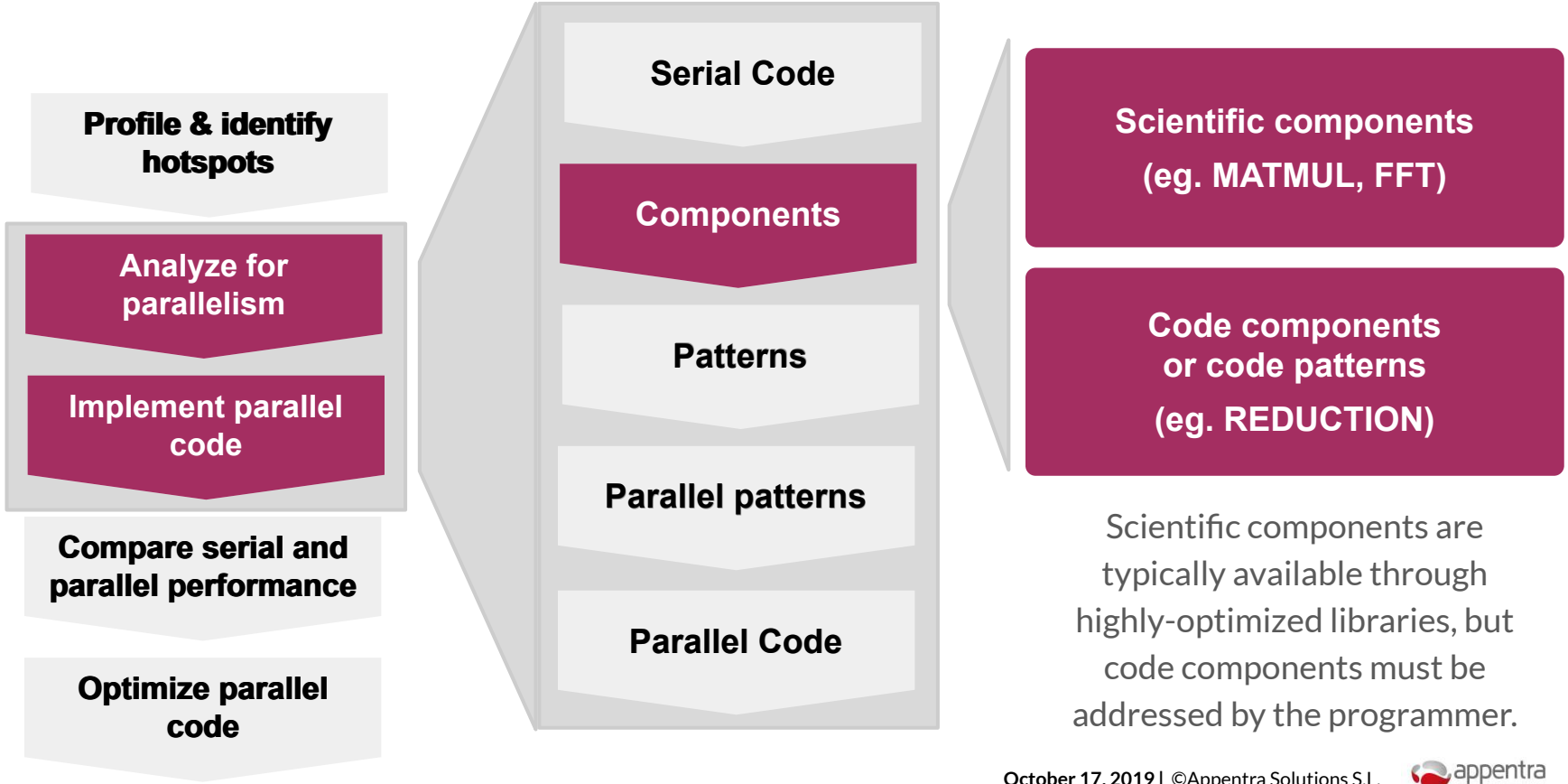
Decomposing your code into components



How does it fit into the classical parallelization workflow?

High-productivity approach independent of OpenMP/OpenACC

Decomposing your code into components



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

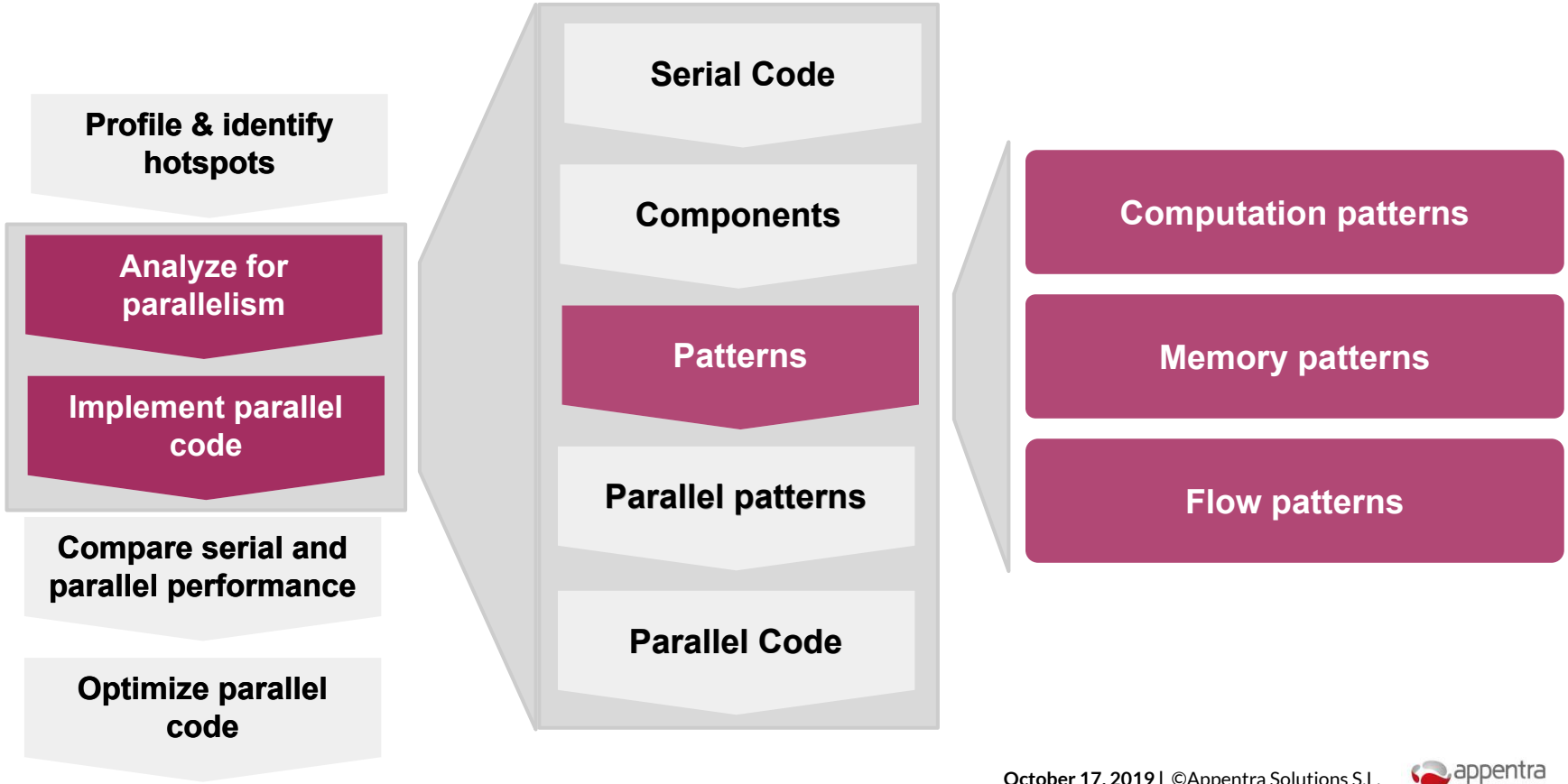
Step 3: **For each routine coded by the programmer that matches a routine contained in external library**

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

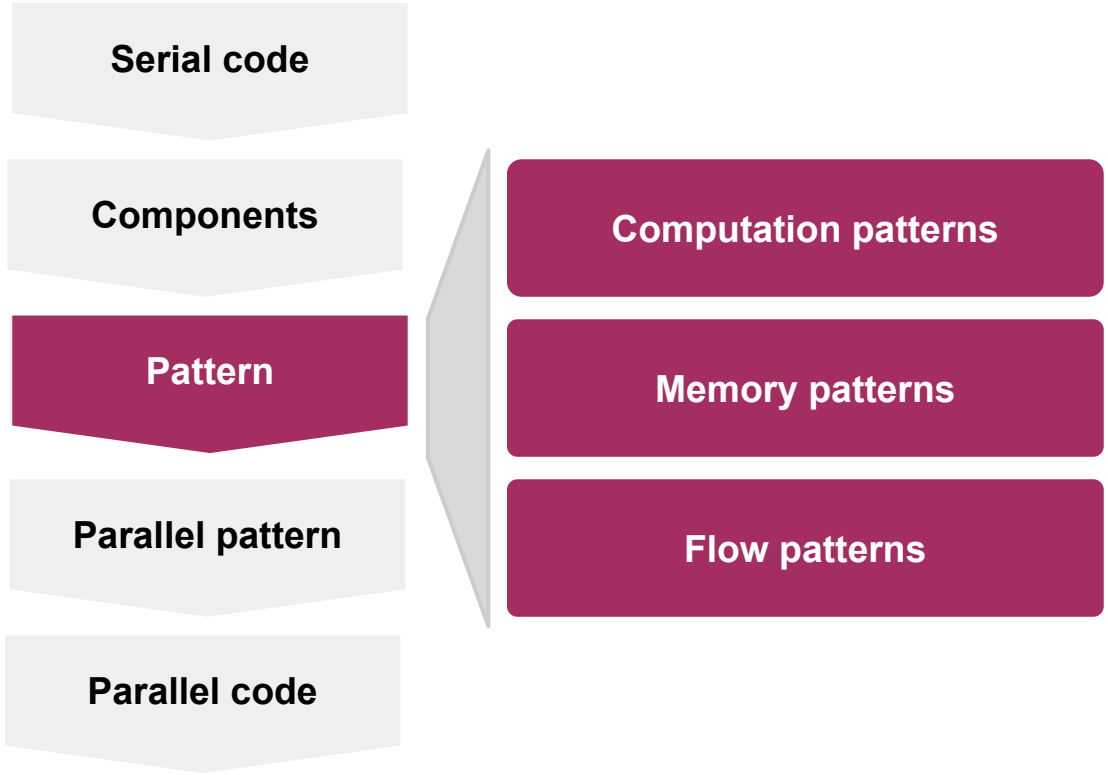
Step 4: **For the remaining user-defined routines**

- 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!

Parallelware Tool Workshop

Learning parallelization of real applications from the ground-up

Manuel Arenaz | October 17, 2019

NERSC

appentra
make code parallel

OpenACC
More Science, Less Programming

OpenMP
Enabling HPC since 1997

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 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 synch at the beginning/end of each time-step iteration.

Flow Patterns

convergence loop

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

propagation loop

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

Convergence loop

Understanding the sequential code

- A time-step loop which stops when a fixed maximum number of loop iterations is achieved or when the value of a numerical error metric is less than a fixed threshold.
- At a given step of a convergence loop, the numerical error is computed by combining the solution in the current loop iteration with the solution in previous iterations (typically 1-2 previous iterations).
- As a result, the convergence loop cannot be parallelized because there are dependencies between consecutive loop iterations.

```
C:
int iter = 0; double err;
for(iter=0, err = tol; err >= tol && iter < iter_max; iter++){
  // compute new solution A_iter using as input A_previous_iter
  // compute numerical error between A_iter and A_previous_iter
  // Copy contents of A_iter into A_previous_iter to prepare for next iteration
}

Fortran:
integer iter = 0
real err = 0
do while (iter < iter_max .and. err > tol)
  // compute new solution A_iter using as input A_previous_iter
  // compute numerical error between A_iter and A_previous_iter
  // Copy contents of A_iter into A_previous_iter to prepare for next iteration
end do
```

Propagation loop

Understanding the sequential code

- A simplified version of convergence loops also typically iterates until a maximum number of iterations is achieved.
- Although no numerical error threshold is checked, the solution in the current loop iteration is computed using the solution in previous iterations (typically 1-2 previous iterations).
- As a result, a propagation loop cannot be parallelized because there are dependencies between consecutive loop iterations.

```
C:
int iter = 0;
for(iter=0; iter < iter_max; iter++){
  // compute new solution A_iter using as input A_previous_iter
  // Copy contents of A_iter into A_previous_iter to prepare for next iteration
}


Fortran:
integer iter = 0
do while (iter < iter_max)
  // compute new solution A_iter using as input A_previous_iter
  // Copy contents of A_iter into A_previous_iter to prepare for next iteration
end do
```

Parallelizing the calculation of HEAT on the GPU with OpenMP/OpenACC

Walkthrough:

- Using Parallelware Trainer in function *compute()*:
  - Generate two separate *data* directives for two consecutive loops.
  - Generate one single data directive that covers two consecutive loops.
  - Open the solution with one joined *data* directive with array shapes.
- Using Parallelware trainer in function *cfd_heat_diffusion()*:
  - Open the solution with one single data directive that covers the convergence loop.

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 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 to 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.

Shaping Arrays in OpenMP/OpenACC

- **Provide the compiler with information about array size and array ranges.**
- **Helps the compiler ensure correct memory allocation on the device**
- **Add the shape specification to the data clauses**, e.g.:

```
x[start:count]
```

  where `start` is the first element to be copied and `count` is the number of elements to copy.

- **Allows storing of only part of the array on the device**

```
#pragma acc data create(x[0:N]) copyout(y[0:N])

!$acc data create(x(0:N)) copyout(y(0:N))
```

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)

Shaping Arrays 1D in OpenMP/OpenACC

- Vectors are typically implemented as arrays 1D.
- Developer can choose between static and dynamic memory allocation.
  - Static arrays are allocated on the stack, which is limited.
  - As a result, large arrays can make the application crash.
- Actual data is stored in consecutive memory locations, which triggers compiler optimizations.

VECTOR size 5

```
double *A = malloc(...)
for(i) {
  … A[i] …
}
```


```
double A[9]
for(i) {
  … A[i] …
}
```

Shaping Arrays 2D in OpenMP/OpenACC

- Matrices are typically implemented as “arrays 2D”, but what is the actual memory layout?
  - It depends on the programming language: row-major in C/C++ and column-major in Fortran.
- Developer can choose between static and dynamic memory allocation.
- Actual data MAY NOT be stored in consecutive memory locations, disabling compiler optimizations.

1	2	3
4	5	0
0	6	0

MATRIX 3x3

Shaping Arrays 2D in OpenMP/OpenACC

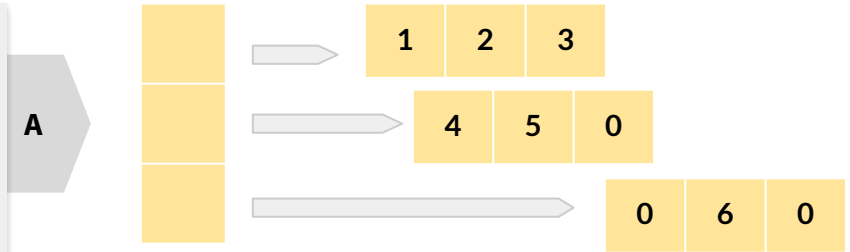
- **Statically allocated arrays** guarantee that actual data is stored in consecutive memory locations. Note that C/C++ and Fortran defer in the order of the data inside de consecutive memory region.

```
double A[3][3]
for(i) {
  for(j) {
    … A[i][j] …
  }
}
```

Shaping Arrays 2D in OpenMP/OpenACC

- **Dynamically allocated arrays** are controlled by the programmer, who is responsible for memory allocation and initialization. How can the programmer guarantee consecutive memory allocation?
- **Dynamically allocated arrays without consecutive memory:**

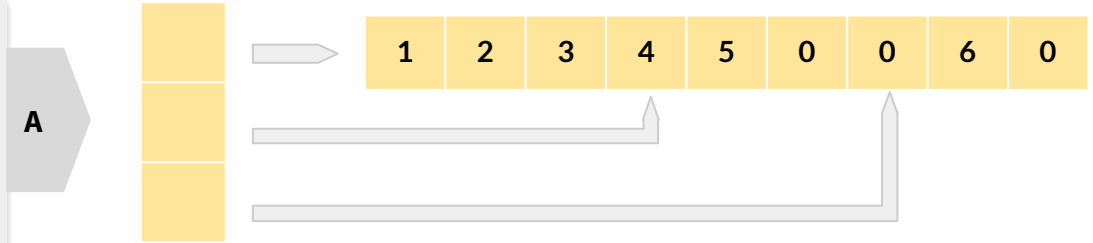
```
double **A = malloc(3)
for(i) {
  A[i] = malloc(3)
}
for(i) {
  for(j) {
    … A[i][j] …
  }
}
```



Shaping Arrays 2D in OpenMP/OpenACC

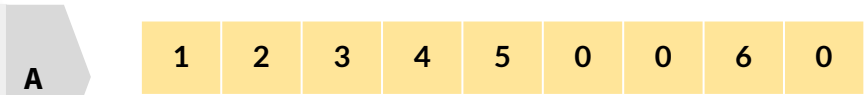
- **Dynamically allocated arrays with consecutive memory**: Several options...

```
double **A = malloc(3)
double *Aaux = malloc(3x3)
for(i) {
  A[i] = Aaux + i * 3
}
for(i) {
  for(j) {
    … A[i][j] …
  }
}
```



In this design, the programmer needs to allocate two separate arrays and initialize the pointers as offset with respect to the beginning of the consecutive memory region.

```
double *A = malloc(3x3)
for(i) {
  for(j) {
    … A[i*3+j] …
  }
}
```



And in this design, the programmer minimizes the memory consumption but must change all of the accesses in the code.

Shaping Arrays 2D in OpenMP/OpenACC

- **Dynamically allocated arrays for sparse matrices**: Sparse storage format uses several auxiliary arrays to avoid storing the elements with value equal to zero.

```
double *A
int *rawIdx
int *cdIdx
```

```
for(ij) {
  … A[rawIdx(ij)*3+colIdx(ij)] …
}
```

A

1	2	3	4	5	0	0	6	0

rawIdx

0	0	0	1	1	1	2	2	2

colIdx

0	1	2	0	1	2	0	1	2

```
for(i) {
  for(j=rawIdx(i),rawIdx(i+1)-1) {
    … A[j] …
  }
}
```

A

1	2	3	4	5	6

colIdx

0	1	2	0	1	1

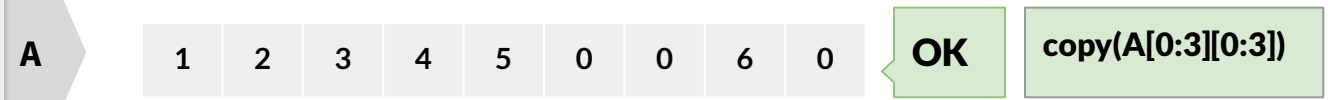
rawIdx

0	3	5	7

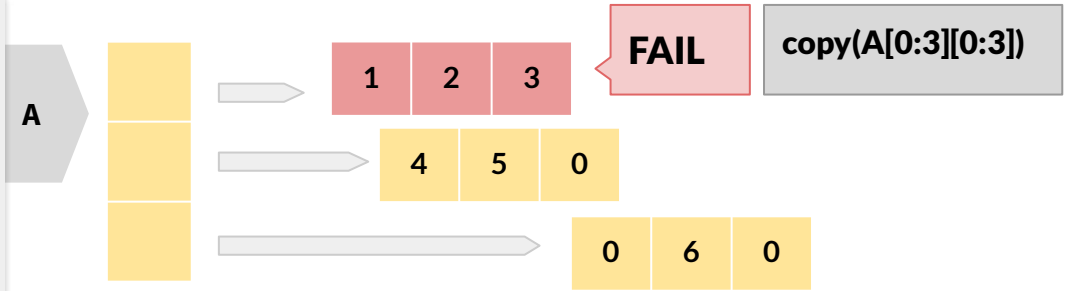
How array shaping affects in OpenMP/OpenACC?

- Array shaping in OpenMP/OpenACC clauses tells the compiler what data must be transferred between CPU memory and GPU memory.

```
double A[3][3]
for(i) {
  for(j) {
    … A[i][j] …
  }
}
```




```
double **A = malloc(3)
for(i) {
  A[i] = malloc(3)
}
for(i) {
  for(j) {
    … A[i][j] …
  }
}
```



Parallelizing MATrix MULtiplication on the GPU with OpenMP/OpenACC

Walkthrough:

- Using Parallelware Trainer in file version *matmul_v1*, function *matmul()*, first loop:
  - Generate OpenACC code, remove the array shape of *C* in *copyout* clause, and watch the defect.
- Using Parallelware Trainer in file version *matmul_v2*, function *matmul()*, first loop:
  - Generate OpenMP code, remove the array shape of *C* in *map* clause, and watch the defect.
- Using Parallelware Trainer in file version *matmul_v3*, function *matmul()*, first loop:
  - Generate OpenACC code, remove the (incomplete) array shape of *C* in *map* clause, and watch the defect.

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!

Computation pattern Forall in nested loops

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 in different source code variants

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

Simple loop

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

Two perfectly nested loops

```
for (j=0; j<n; j++ ) {
  C[j] = 0;
  for (k=0; k<n; k++ ) {
     A[j][k] = B[j][k] + A[j][k-1];
  }
}
```

Not perfectly nested loops: dependencies between iterations

Clause: `collapse`

- Collapses multiple loops into one “larger” loop.
- Use case: when one or more loops are perfectly nested and there are not dependencies that prevent the parallel execution of all of the iterations of all of the loops at the same time.

```
#pragma acc parallel loop collapse(2)
for (j=0; j<n; j++ ) {
  for (k=0; k<n; k++ ) {
     A[j][k] = B[j][k];
  }
}
```

Identifying opportunities for parallelization in different source code variants

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

Simple loop

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

OK

collapse(2)

Two perfectly nested loops

```
for (j=0; j<n; j++ ) {
  C[j] = 0;
  for (k=0; k<n; k++ ) {
     A[j][k] = B[j][k] + A[j][k-1];
  }
}
```

FAIL

collapse(2)

Not perfectly nested loops: dependencies between iterations

Parallelizing MATrix MULtiplication on the GPU with OpenMP/OpenACC

Walkthrough:

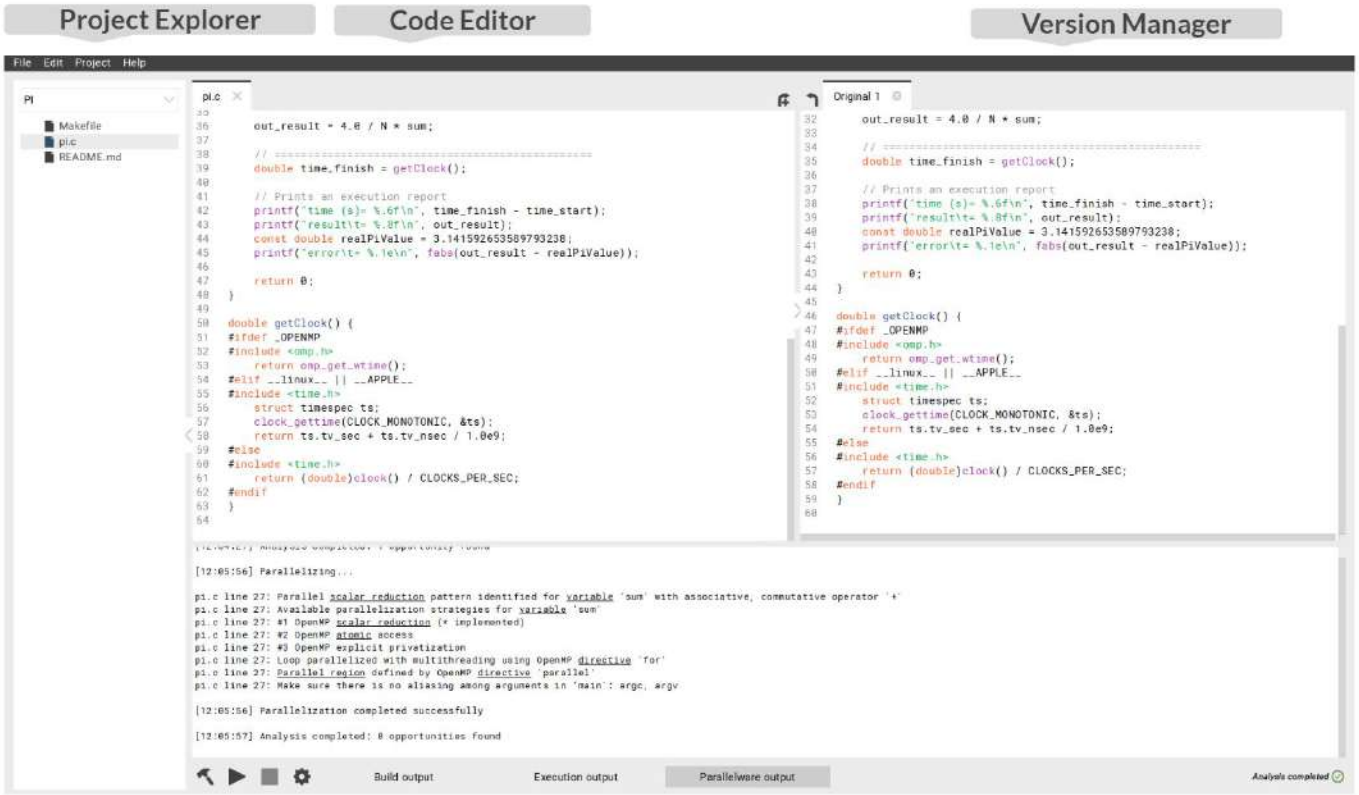
- Using Parallelware Trainer in file version *matmul_v1*, function *matmul()*:
  - Generate one single *data* directive that covers two consecutive loops.
  - In each *loop* directive, add *collapse(2)* clause to exploit parallelism across the two nested loops at the same time.
  - Finally, add an incorrect *collapse(3)* clause for the three nested loops.

Parallelizing the calculation of HEAT on the GPU with OpenMP/OpenACC

Goals:

- Build & run an OpenMP code on the GPU (for problem size N=100000)
- Build & run an OpenACC code on the GPU (for problem size N=100000)
  - Using OpenACC *kernels* directive for automatic parallelization (manually, not supported in Parallelware Trainer)
  - Using OpenACC *loop* directives for finer control
  - Using a single *data* directive enclosing all *loop* ones
  - Using a single *data* directive for the program

Parallelware Trainer



Run Parallelware Trainer on CORI

Step 1: Log in to CORI enabling X11 forwarding

```
$ ssh -X <your_login>@cori.nersc.gov
```

Step 2: Copy the sample codes to be used during the course

```
$ cp -a /global/common/cori_cle7/software/pwtrainer/Workshop-Oct19-examples.zip ~
```

Step 3: Prepare the environment by loading the appropriate modules

```
$ module purge && module load esslurm cuda gcc/8.1.1-openacc-gcc-8-branch-20190215 pgi/19.9 pwtrainer
```

Step 4 (*): Open an interactive session in a GPU node

```
$ salloc -t 60 -N 1 -c 8 -C gpu --gres=gpu:1 --mem=32GB --reservation=trainer -A nintern
```

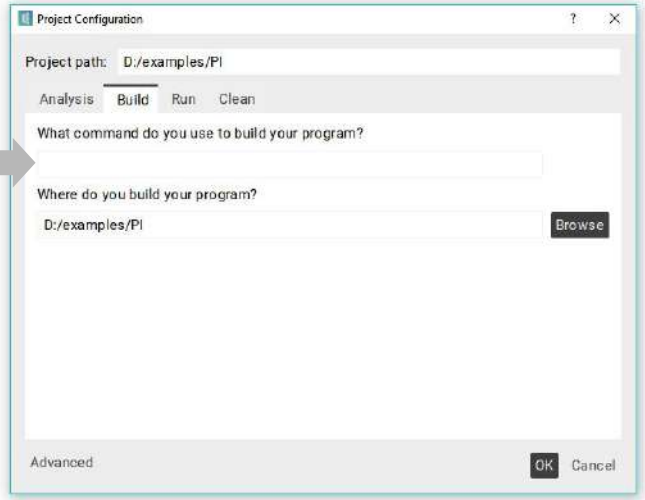
(*) For CPU nodes, use --reservation=trainer_knl for the training
(*) Some people might need to use -A nstaff or -A m1759 instead of -A nintern

Step 5: Run the Parallelware Trainer tool from the GPU node

```
$ pwtrainer &
```

Build in Parallelware Trainer

	OpenMP Enabling HPC since 1997	OpenACC More Science, Less Programming
GCC	`gcc -fopenmp -foffload=nvptx-none="-Ofast -lm -misa=sm_35" -lm -o heat heat.c`	`gcc -fopenacc -foffload=nvptx-none="-Ofast -lm -misa=sm_35" -lm -D_OPENMP -o heat heat.c`
PGI	`pgcc -mp -ta=tesla -Minfo=accel -lm -o heat heat.c`	`pgcc -acc -ta=tesla -Minfo=accel -lm -D_OPENMP -o heat heat.c`

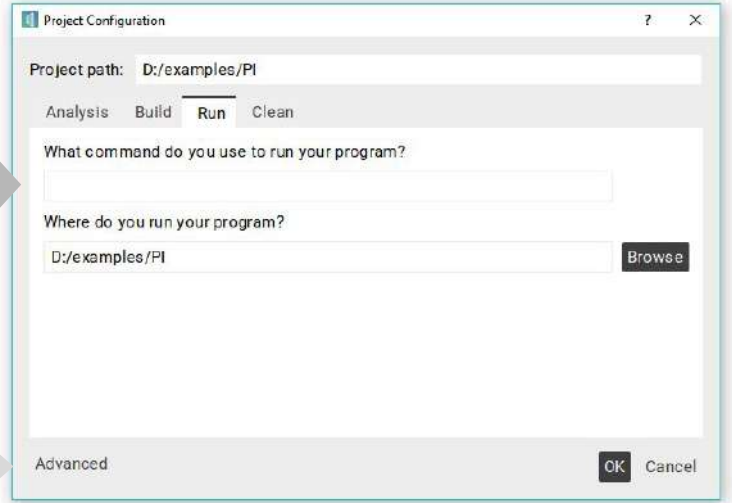


N.B. For Macs, check that your compiler supports OpenMP/OpenACC, and update with the appropriate compiler (e.g. gcc-mp-7)

Run in Parallelware Trainer

OpenMP Enabling HPC since 1997	`srun env OMP_NUM_THREADS=4 ./heat 100000`
OpenACC More Science, Less Programming	`srun env ACC_DEVICE_TYPE=nvidia ./heat 100000`

N.B. Instead of using the **env** command, you can add this variable by clicking the Advanced button.



Decomposition in parallel patterns

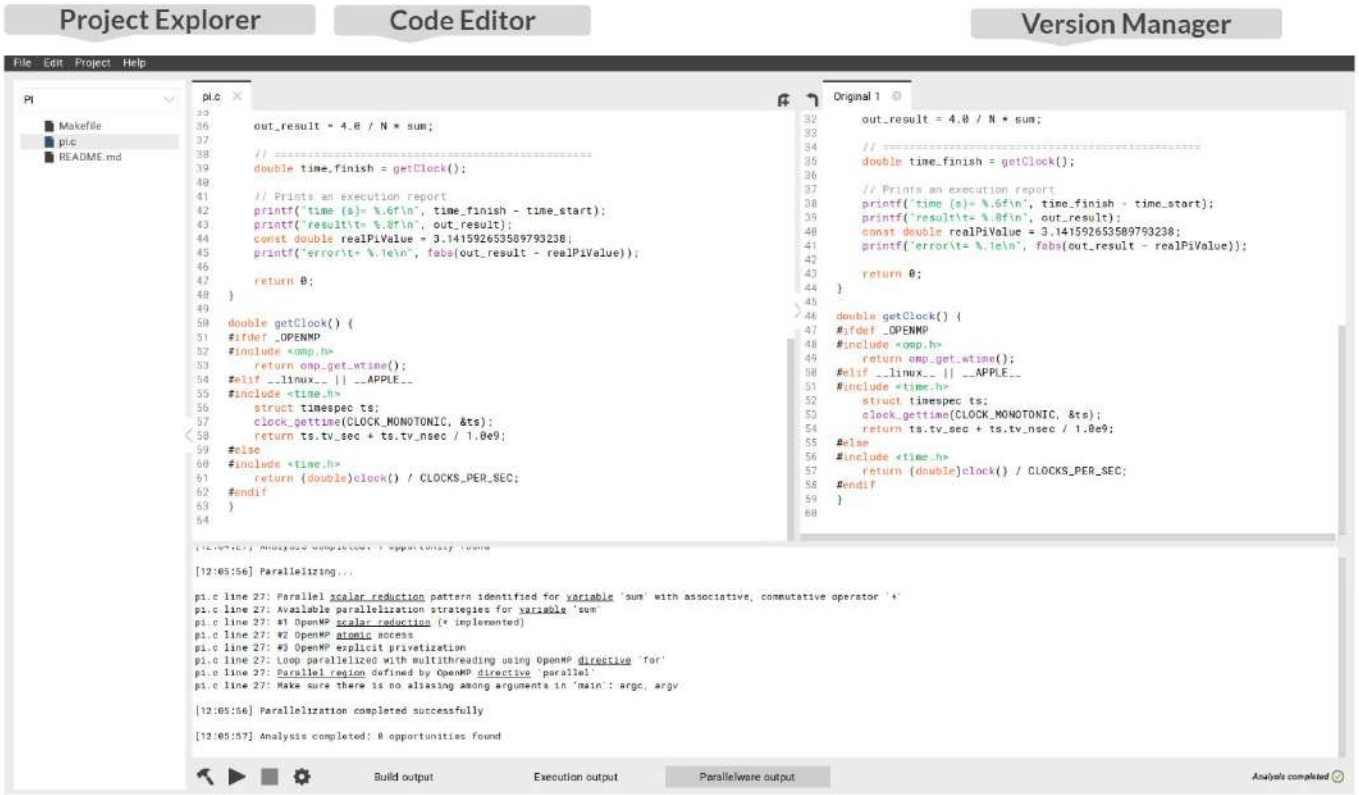
Code file "heat.c"		Pattern			
Function	**Line**	**Forall**	**Scalar reduction**	**Sparse reduction**	**Convergence loop**
compute()	25	T			
	28			T	
	33		maxdiff		
	39	uk			
initialize_uk()	49				
initialize_T()	54				
	56				
cfd_heat_diffusion()	73				
main()					
calculate_checksum()	133				

Parallelizing MATrix MULtiplication on the GPU with OpenMP/OpenACC

Goals:

- Build & run an OpenMP code on the GPU (for problem size N=5000)
- Build & run an OpenACC code on the GPU (for problem size N=5000)
  - Increase parallelism by collapsing loops
  - Choose the better memory layout for optimized memory transfers to the GPU

Parallelware Trainer



Run Parallelware Trainer on CORI

Step 1: Log in to CORI enabling X11 forwarding

```
$ ssh -X <your_login>@cori.nersc.gov
```

Step 2: Copy the sample codes to be used during the course

```
$ cp -a /global/common/cori_cle7/software/pwtrainer/Workshop-Oct19-examples.zip ~
```

Step 3: Prepare the environment by loading the appropriate modules

```
$ module purge && module load esslurm cuda gcc/8.1.1-openacc-gcc-8-branch-20190215 pgi/19.9 pwtrainer
```

Step 4 (*): Open an interactive session in a GPU node

```
$ salloc -t 60 -N 1 -c 8 -C gpu --gres=gpu:1 --mem=32GB --reservation=trainer -A nintern
```

(*) For CPU nodes, use --reservation=trainer_knl for the training
(*) Some people might need to use -A nstaff or -A m1759 instead of -A nintern

Step 5: Run the Parallelware Trainer tool from the GPU node

```
$ pwtrainer &
```

Memory layout versions

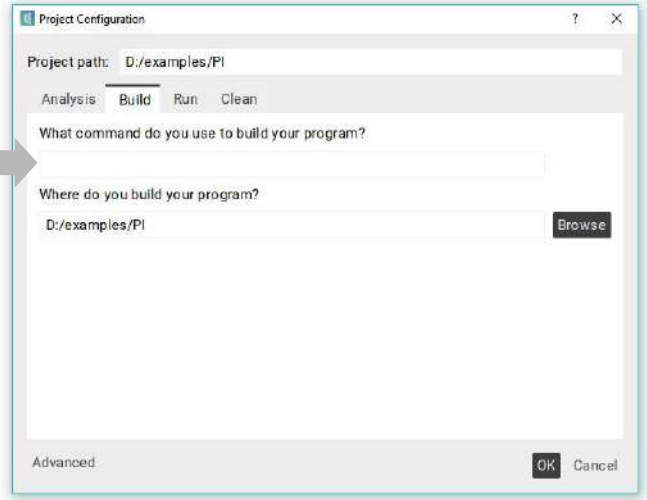
Three MATMUL versions differing in how 2D matrices are implemented using dynamic memory

Dynamic memory is preferred because it uses the **heap**,
unlike static memory which may consume the limited **stack** memory!

Version	Data structure	Memory layout	Array syntax
matmul_v1_dynarray2d_noncont.c	Dynamic array of pointers to dynamic 1D arrays (double**)	NON contiguous	A[i][j]
matmul_v2_dynarray2d.c	Dynamic array of pointers to locations of a dynamic 1D array containing the linearized data (double**)	Contiguous	A[i][j]
matmul_v3_dynarray1d.c	Dynamic 1D array containing the linearized data (double*)	Contiguous	A[i * cols + j]

Build in Parallelware Trainer

	OpenMP Enabling HPC since 1997	OpenACC More Science, Less Programming
GCC	`gcc -fopenmp -foffload=nvptx-none="-Ofast -lm -misa=sm_35" -o matmul matmul_v3_dynarray1d.c`	`gcc -fopenacc -foffload=nvptx-none="-Ofast -lm -misa=sm_35"-D_OPENMP -o matmul matmul_v3_dynarray1d.c`
PGI	`pgcc -mp -ta=tesla -Minfo=accel -o matmul matmul_v3_dynarray1d.c`	`pgcc -acc -ta=tesla -Minfo=accel -D_OPENMP -o matmul matmul_v3_dynarray1d.c`

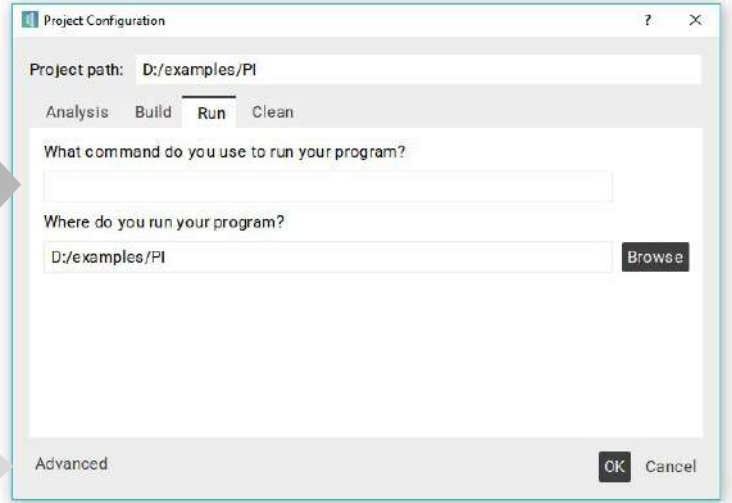


N.B. For Macs, check that your compiler supports OpenMP/OpenACC, and update with the appropriate compiler (e.g. gcc-mp-7)

Run in Parallelware Trainer

OpenMP Enabling HPC since 1997	`srun env OMP_NUM_THREADS=4 ./matmul 5000`
OpenACC More Science, Less Programming	`srun env ACC_DEVICE_TYPE=nvidia ./matmul 5000`

N.B. Instead of using the **env** command, you can add this variable by clicking the Advanced button.



Decomposition in parallel patterns

Code file "matmul_v3_dynarray1d.c"		**Pattern**			
Function	**Line**	**Forall**	**Scalar reduction**	**Sparse reduction**	**Convergence loop**
matmul()	22	C			
	32	C			
main()					

Parallelizing the CORAL LULESH microkernel

Goals:

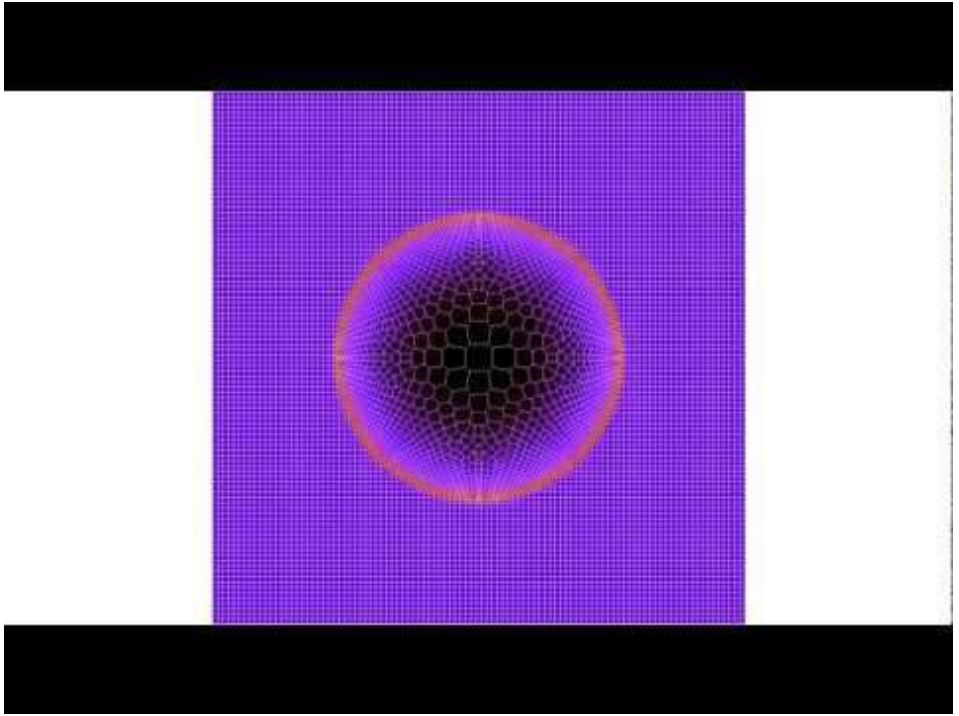
- Understand the code components
- Parallelize each component, including comparing performance of different implementations
- Combine parallel regions as required to improve performance
- Compare CPU and GPU versions
- Understand the data structure design used in the code and analyze pros/cons
- Minimize data transfers to the GPU

CORAL Benchmarks: LULESH

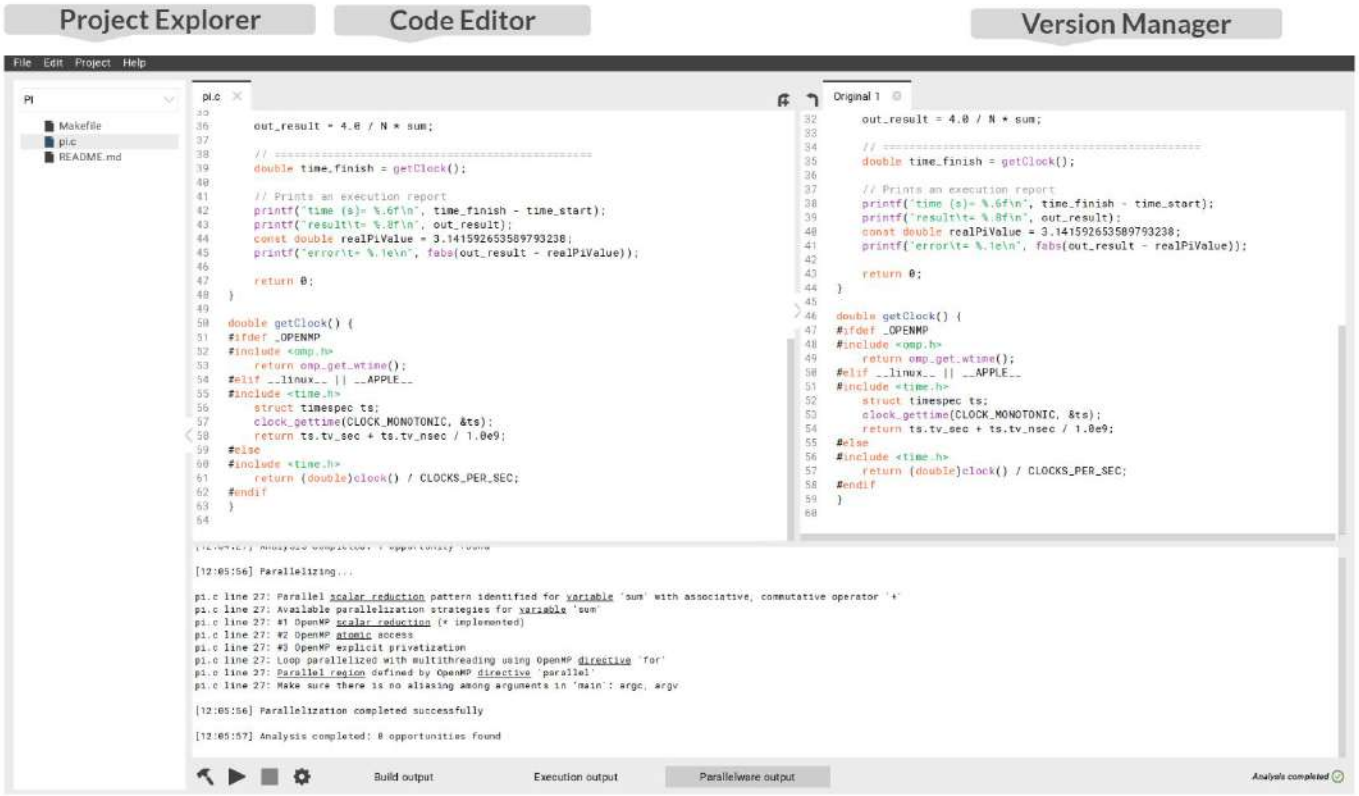
Livermore **U**nstructured **L**agrange
Explicit **S**hock **H**ydrodynamics

Part of a Physics Simulation software (ALE3D)

Models the propagation of a Sedov blast wave using Lagrangian hydrodynamics



Parallelware Trainer



Run Parallelware Trainer on CORI

Step 1: Log in to CORI enabling X11 forwarding

```
$ ssh -X <your_login>@cori.nersc.gov
```

Step 2: Copy the sample codes to be used during the course

```
$ cp -a /global/common/cori_cle7/software/pwtrainer/Workshop-Oct19-examples.zip ~
```

Step 3: Prepare the environment by loading the appropriate modules

```
$ module purge && module load esslurm cuda gcc/8.1.1-openacc-gcc-8-branch-20190215 pgi/19.9 pwtrainer
```

Step 4: (*) Open an interactive session in a GPU node

```
$ salloc -t 60 -N 1 -c 8 -C gpu --gres=gpu:1 --mem=32GB --reservation=trainer -A nintern
```

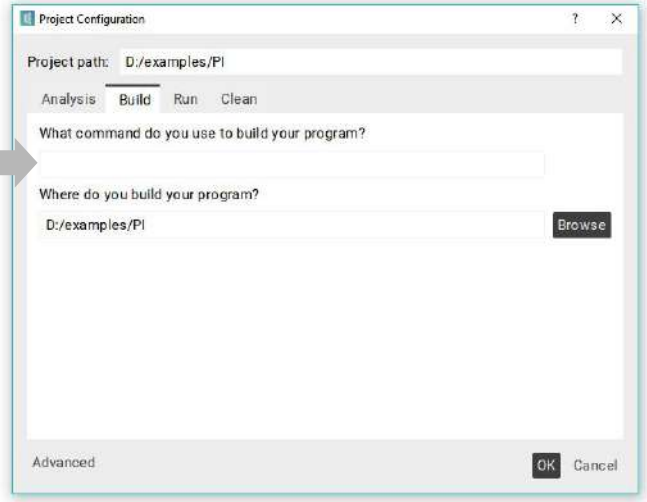
(*) For CPU nodes, use --reservation=trainer_knl for the training
(*) Some people might need to use -A nstaff or -A m1759 instead of -A nintern

Step 5: Run the Parallelware Trainer tool from the GPU node

```
$ pwtrainer &
```

Build in Parallelware Trainer

	OpenMP Enabling HPC since 1997	OpenACC More Science, Less Programming
GCC	`gcc -fopenmp -foffload=nvptx-none="-Ofast -lm -misa=sm_35" -lm -o luleshmk luleshmk.c`	`gcc -fopenacc -foffload=nvptx-none="-Ofast -lm -misa=sm_35" -lm -D_OPENMP -o luleshmk luleshmk.c`
PGI	`pgcc -mp -ta=tesla -Minfo=accel -lm -o luleshmk luleshmk.c`	`pgcc -acc -ta=tesla -Minfo=accel -lm -D_OPENMP -o luleshmk luleshmk.c`

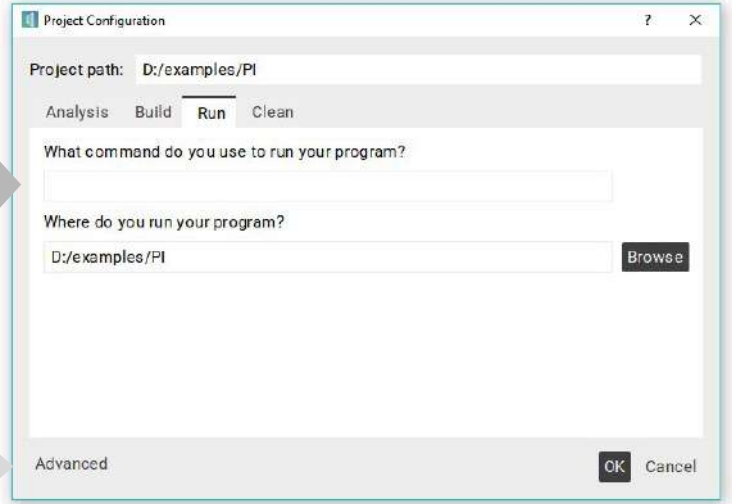


N.B. For Macs, check that your compiler supports OpenMP/OpenACC, and update with the appropriate compiler (e.g. gcc-mp-7)

Run in Parallelware Trainer

OpenMP Enabling HPC since 1997	`srun env OMP_NUM_THREADS=4 ./luleshmk`
OpenACC More Science, Less Programming	`srun env ACC_DEVICE_TYPE=nvidia ./luleshmk`

N.B. Instead of using the **env** command, you can add this variable by clicking the Advanced button.



Profiling of LULESH microkernel

```
$ gcc -pg -o luleshmk luleshmk.c -lm
$ ./luleshmk
$ gprof ./luleshmk
```

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 56.23     24.62    24.62 223680000     0.00     0.00  CalcElemFBHourglassForce_workload
 22.76     34.59     9.97      932     0.01     0.01  ApplyMaterialPropertiesForElems_workload
 15.26     41.27     6.68 27960000     0.00     0.00  CalcElemVelocityGradient_workload
  2.26     42.26     0.99      932     0.00     0.03  CalcFBHourglassForceForElems
  2.24     43.24     0.98 27960000     0.00     0.00  CalcElemFBHourglassForce
  0.75     43.57     0.33 27960000     0.00     0.00  CalcElemVelocityGradient
  0.34     43.72     0.15        1     0.15    43.76  luleshmk
  0.09     43.76     0.04      932     0.00     0.01  CalcKinematicsForElems
  0.09     43.80     0.04                             frame_dummy
  0.00     43.80     0.00      932     0.00     0.01  ApplyMaterialPropertiesForElems
  0.00     43.80     0.00        2     0.00     0.00  calculate_checksum
  0.00     43.80     0.00        2     0.00     0.00  getClock
  0.00     43.80     0.00        1     0.00     0.00  Parameters_create
  0.00     43.80     0.00        1     0.00     0.00  Parameters_free
  0.00     43.80     0.00        1     0.00     0.00  VerifyAndWriteFinalOutput
```

How to verify correctness of LULESH

```
$ ./luleshmk
- Configuring the test...
- Executing the test...
gprof ./luleshmk
- Verifying the test...
Run completed:
   Problem size        =  30
   MPI tasks           =  1
   Iteration count     =  932
   Final Origin Energy = 1.000000e+00
   Number of nodes    = 27000
   Number of elements = 30000
   Number of regions  = 1
      Region 1 of size 30000
   Testing Plane 0 of Energy Array on rank 0:
        MaxAbsDiff   = 8.410000e+02
        TotalAbsDiff = 1.303550e+05
        MaxRelDiff   = 9.655568e-01

Elapsed time         =      71.00 (s)
Grind time (us/z/c)  =   2.821491 (per dom)  (  2.821491
overall)
FOM                  =  354.42254 (z/s)
```

Decomposition of LULESH into patterns

Code file "luleshmk.c"		**Pattern**				
Function	**Line**	**Forall**		**Scalar reduction**	**Sparse reduction**	**Convergence loop**
CalcElemFBHourglassForce_workload()	60			sum		
CalcElemFBHourglassForce()	73	hgfx				
CalcFBHourglassForceForElems()	**131**				**domain_m_fx**	
ApplyMaterialPropertiesForElems_workload()	187					
ApplyMaterialPropertiesForElems()	210 217					
CalcElemVelocityGradient_workload()	231					
CalcKinematicsForElems()	258					
luleshmk()	292					iter (loop index)
main()	349					
	358					
	365					
VerifyAndWriteFinalOutput()	485					