

Parallelware Tools: Software Performance Optimization

NUG Monthly Meeting at NERSC

Manuel Arenaz | March 2021

©Appentra Solutions S.L.



Impact of Parallelware tools on performance

The runtime was reduced up to 89% (9x faster) on multicores and up to 99% (128x faster) on GPUs

Application areas	Parallel code created by Parallelware tools	Performance gain on multicore CPU (including multithreading)		Performance gain on GPU	
		[% time faster]	[No. times faster]	[% time faster]	[No. times faster]
High Performance Computing (HPC)	PI	+73%	3.83x	+97%	41.94x
	COULOMB	+89%	9.68x	-	-
	HEAT	+77%	4.41x	+89%	9.63x
	MATMUL	+79%	4.92x	+99%	128.33x
	ATMUX	+58%	2.40x	-79%	0.56x
	LULESHmk	+73%	3.75x	+95%	24.40x
	NPB CG	-	20.52x	-	-
	ZPIC	+17%	1.22x	+33%	1.49x
Embedded systems	Canny	+23%	1.30x	-	-

Speedups measured at NERSC CORI system (Cray XC40), GPU nodes:

CPU: 2 x Intel Xeon Gold 6148 ('Skylake') @ 2.40 GHz (16 threads allocated)

MEMORY: 384 GB DDR4 memory (32 GB allocated)

GPU: 8 x NVIDIA V100, each with 16 GB HBM2 memory, connected with NVLink interconnect (2 allocated)

Parallelware Analyzer for SIMD: Intel AVX-512

The runtime was reduced up to 66% (~3x faster) on multicores using the Clang and GCC compilers

Use case: Canny (Image processing) Code versions using SIMD, multithreading and offloading parallelism	Clang 10.0 (seconds)	GCC 8.2 (seconds)
Canny SERIAL (serial version, maximum optimization without auto-vectorization)	11,08	12,03
Canny AUTO (auto-vectorized serial version, maximum optimization)	10,64 [3.9%] [1.04x]	11,63 [3.4%] [1.04x]
Canny PWA SIMD (*) (PWA SIMD + auto-vectorized serial version, maximum optimization)	4,97 [55.1%] [2.23x]	5,61 [53.4%] [2.14x]
Canny PWA MULTI+SIMD (PWA Multithreading+SIMD + auto-vectorized serial version, maximum optimization)	3,72 [66.4%] [2.98x]	4,08 [66.1%] [2.95x]
Canny PWA GPU+SIMD (PWA GPU+SIMD + auto-vectorized serial version, maximum optimization)	-	-

Higher %
is faster

CPU: Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz (EPEEC cluster)
Problem size: Image size 15360 x 8640.

(*) Performance optimization achieved by a senior performance software engineer and that is the expected maximum performance improvement to be provided by Parallelware Analyzer SIMD.

Parallelware Analyzer for SIMD: Arm NEON

The runtime was reduced up to 62% (~3x faster) on multicores using the Clang and GCC compilers

Use case: Canny (Image processing) Code versions using SIMD, multithreading and offloading parallelism	Clang 7.0 (seconds)	GCC 8.3 (seconds)
Canny SERIAL (serial version, maximum optimization without auto-vectorization)	66,87	66,75
Canny AUTO (auto-vectorized serial version, maximum optimization)	65,20 [2.5%] [1.03x]	66,11 [0.9%] [1.01x]
Canny PWA SIMD (*) (PWA SIMD + auto-vectorized serial version, maximum optimization)	39,18 [41.4%] [1.71x]	40,36 [39.5%] [1.65x]
Canny PWA MULTI+SIMD (PWA Multithreading+SIMD + auto-vectorized serial version, maximum optimization)	24,79 [62.9%] [2.70x]	25,63 [61.6%] [2.60x]
Canny PWA GPU+SIMD (PWA GPU+SIMD + auto-vectorized serial version, maximum optimization)	-	-

Higher %
is faster

CPU: ARM Cortex-A72

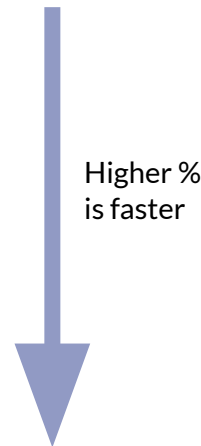
Problem size: Image size 15360 x 8640.

(*) Performance optimization achieved by a senior performance software engineer and that is the expected maximum performance improvement to be provided by Parallelware Analyzer SIMD.

Parallelware Analyzer for SIMD: Intel SSE4

The runtime was reduced up to 64% (~3x faster) on multicores using the Clang and GCC compilers

Use case: Canny (Image processing) Code versions using SIMD, multithreading and offloading parallelism	Clang 10.0 (seconds)	GCC 8.2 (seconds)
Canny SERIAL (serial version, maximum optimization without auto-vectorization)	11,86	12,26
Canny AUTO (auto-vectorized serial version, maximum optimization)	11,51 [2.9%] [1.03x]	11,85 [3.3%] [1.03x]
Canny PWA SIMD (*) (PWA SIMD + auto-vectorized serial version, maximum optimization)	5,60 [52.8%] [2.12x]	5,72 [53.4%] [2.14x]
Canny PWA MULTI+SIMD (PWA Multithreading+SIMD + auto-vectorized serial version, maximum optimization)	4,25 [64.1%] [2.79x]	4,24 [65.4%] [2.89x]
Canny PWA GPU+SIMD (PWA GPU+SIMD + auto-vectorized serial version, maximum optimization)	-	-



Higher %
is faster

CPU: Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz (EPEEC cluster)
Problem size: Image size 15360 x 8640.

(*) Performance optimization achieved by a senior performance software engineer and that is the expected maximum performance improvement to be provided by Parallelware Analyzer SIMD.

Parallelware Analyzer for SIMD: AMD AVX-2

The runtime was reduced up to 65% (~3x faster) on multicores using the Clang and GCC compilers

Use case: Canny (Image processing) Code versions using SIMD, multithreading and offloading parallelism	Clang 10.0 (seconds)	GCC 9.3 (seconds)
Canny SERIAL (serial version, maximum optimization without auto-vectorization)	10,00	10,92
Canny AUTO (auto-vectorized serial version, maximum optimization)	8,88 [11.2%] [1.13x]	10,33 [5.4%] [1.06x]
Canny PWA SIMD (*) (PWA SIMD + auto-vectorized serial version, maximum optimization)	4,51 [54.9%] [2.22x]	5,09 [53.4%] [2.15x]
Canny PWA MULTI+SIMD (PWA Multithreading+SIMD + auto-vectorized serial version, maximum optimization)	3,48 [65.2%] [2.88x]	3,81 [65.2%] [2.87x]
Canny PWA GPU+SIMD (PWA GPU+SIMD + auto-vectorized serial version, maximum optimization)	-	-

Higher %
is faster

CPU: AMD Ryzen 7 4800H with Radeon Graphics
Problem size: Image size 15360 x 8640.

(*) Performance optimization achieved by a senior performance software engineer and that is the expected maximum performance improvement to be provided by Parallelware Analyzer SIMD.

Major challenges in CPU+GPU programming

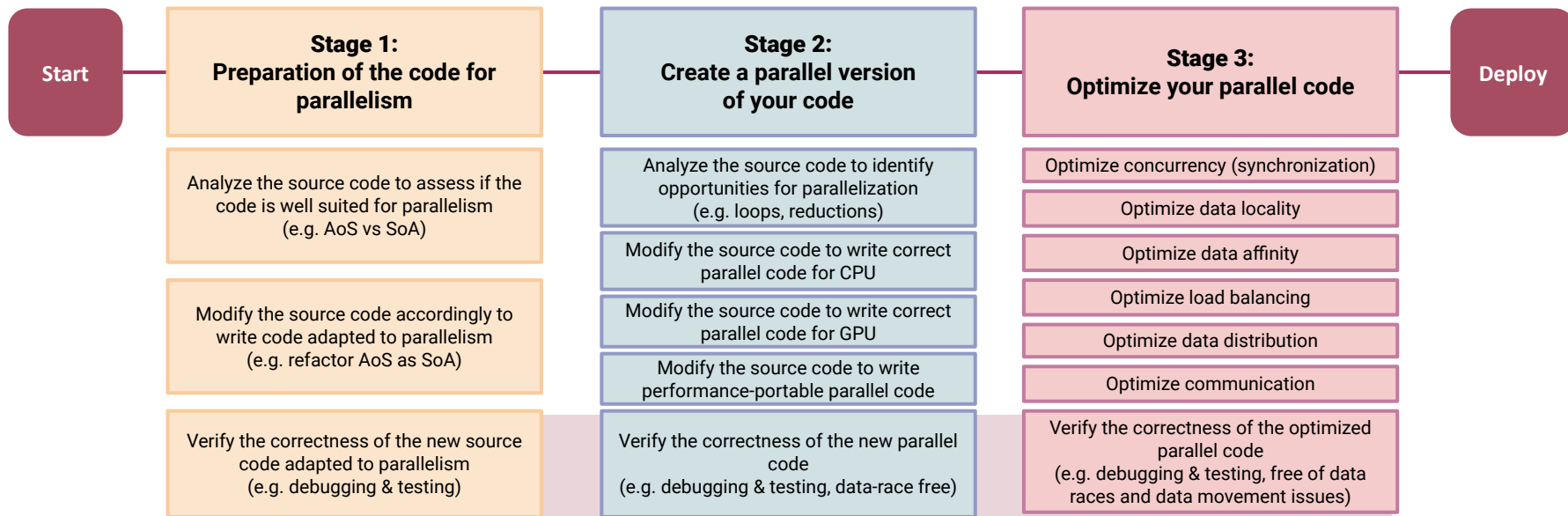
- The development and maintenance of **bug-free C/C++/Fortran parallel code** is far more complex than that of sequential software.
- **Parallel bugs are difficult to find and fix** because a buggy parallel code might run correctly 99% of the time and fail just the remaining 1%.
- This is **even more difficult for Graphical Processing Units (GPUs)**.
- In order to take advantage of the performance promised by CPU+GPUs, developers must address two main challenges:
 - **Challenge #1: Data movement** (i.e. ensure the proper data synchronization between the CPU memory and the GPU memory)
 - **Challenge #2: Data races** (i.e. running the computations on the GPU correctly without race conditions between the GPU threads)

How can we help CPU+GPU programmers?

- **CPU+GPU programming is very hard and very intrusive** because it usually requires major changes in the code
- Major efforts by the CPU+GPU programmer focus on:
 - **#1: Detect** and fix data races and data movement issues
 - **#2: Verify** that parallel code is free of data races and data movement issues
 - **#3: Discover** opportunities in the code to be executed in parallel in CPU+GPU systems
 - **#4: Implement** versions of the code for CPU+GPU systems
- **New Dev tools to improve programmer's productivity on CPU+GPUs are needed**
 - **Helping to find and fix parallel bugs**
 - **Helping to prevent parallel bugs**

Parallel Programming Best Practices

“Develop parallel code using C/C++/Fortran targeting multicore CPUs and GPUs”



Ensuring Parallel Programming Best Practices



Open catalog of defects & recommendations

Open catalog of defects and recommendations for parallel programming built in collaboration with experts in multicore and GPU programming to establish parallel programming best practices. Open set of curated example codes that clearly describe errors commonly seen in C/C++/Fortran parallel codes.

[Discover open catalog of checks ›](https://www.appentra.com/knowledge/checks/)

<https://www.appentra.com/knowledge/checks/>



Tools to automate time-consuming development tasks

Products based on the **Parallelware static code** analysis technology are the first tools supporting this innovative catalog by reporting race conditions, data movement issues and best-practice recommendations to create efficient and bug-free parallel code.

[Discover Parallelware tools ›](#)

PARALLELWARE
TRAINER

PARALLELWARE
ANALYZER



[Download the PDF](#)

Defects

PWD001: Invalid OpenMP multithreading datascoping

PWD002: Unprotected multithreading reduction operation

PWD003: Missing array range in data copy to accelerator device

PWD004: Out-of-memory-bounds array access

PWD005: Array range copied to the GPU does not cover the used range

PWD006: Missing deep copy of non-contiguous data to the GPU

PWD007: Unprotected multithreading recurrence

PWD008: Unprotected multithreading recurrence due to out-of-dimension-bounds array access

Recommendations

PWR001: Declare global variables as function parameters

PWR002: Declare scalar variables in the smallest possible scope

PWR003: Explicitly declare pure functions

PWR004: Declare OpenMP scoping for all variables

PWR005: Disable default OpenMP scoping

PWR006: Avoid privatization of read-only variables

PWR007: Disable implicit declaration of variables

PWR008: Declare the intent for each procedure parameter

PWR009: Use OpenMP teams to offload work to GPU

PWR010: Avoid column-major array access in C/C++

PWR011: Outline loop to increase compiler and tooling code coverage

PWR012: Pass only required fields from derived data types as parameters

PWR013: Avoid copying unused variables to the GPU

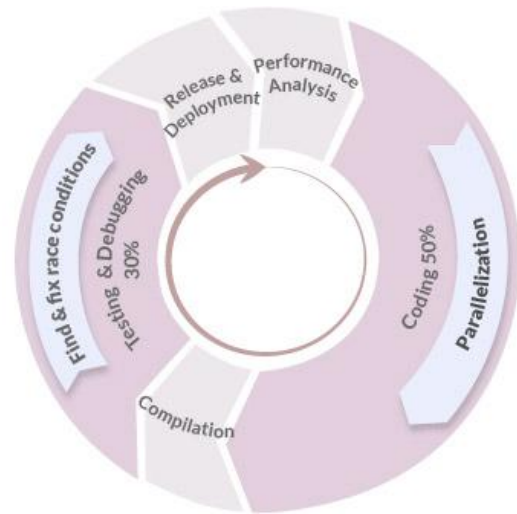
PWR014: Out-of-dimension-bounds array access

PWR015: Avoid copying unnecessary array elements to the GPU

PWR016: Use separate arrays instead of an Array-of-Structs

Parallelware Tools

1. **Enforce parallel programming best practice recommendations** in order to prepare the code for parallelization.
2. **Detect and fix defects in parallel code** (i.e. race conditions).
3. **Verify data-race free parallel code.**
4. **Discover opportunities for parallelization.**
5. **Quickly design and implement parallel code** for CPU/GPU using OpenMP/OpenACC.



Learn parallel programming faster and at your own pace

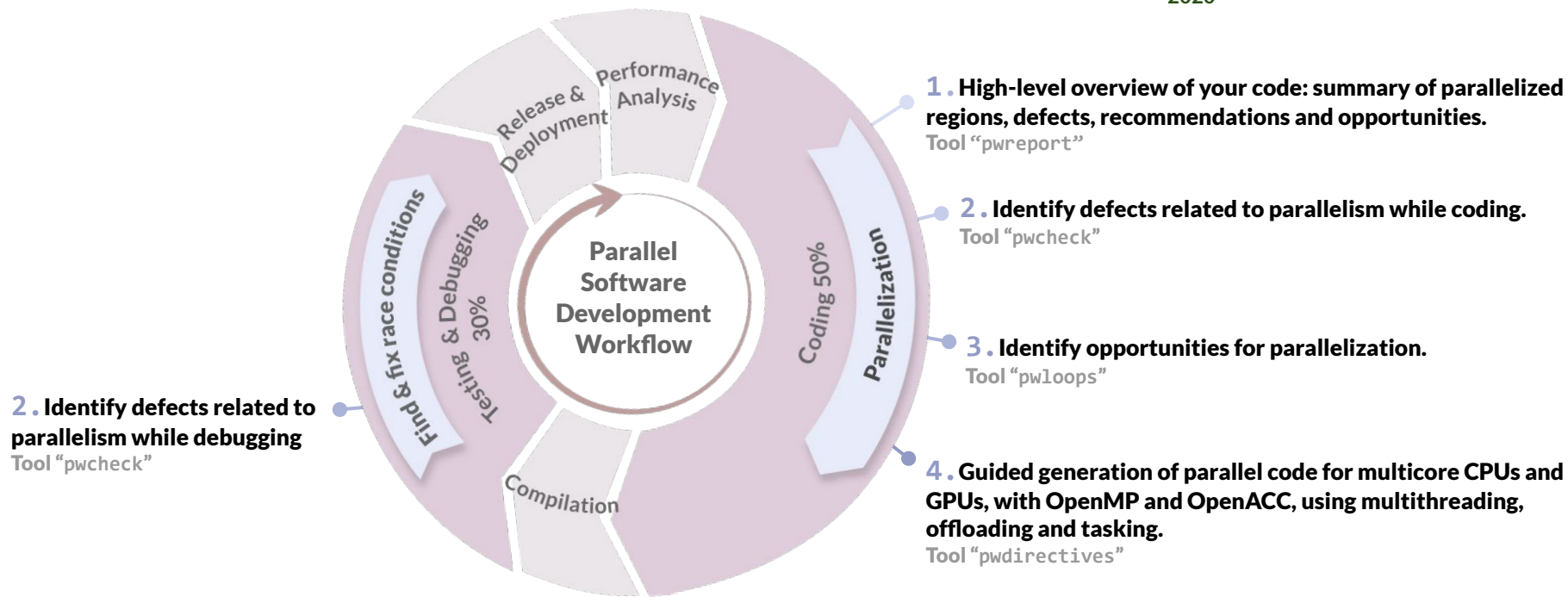
Graphical User Interface (GUI)



A static code analyzer specializing in parallelism

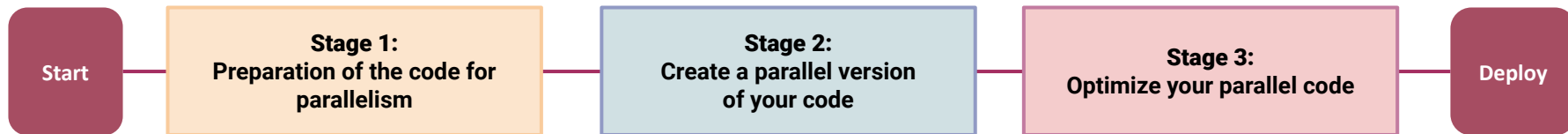
Command Line Interface (CLI)

Parallelware Analyzer



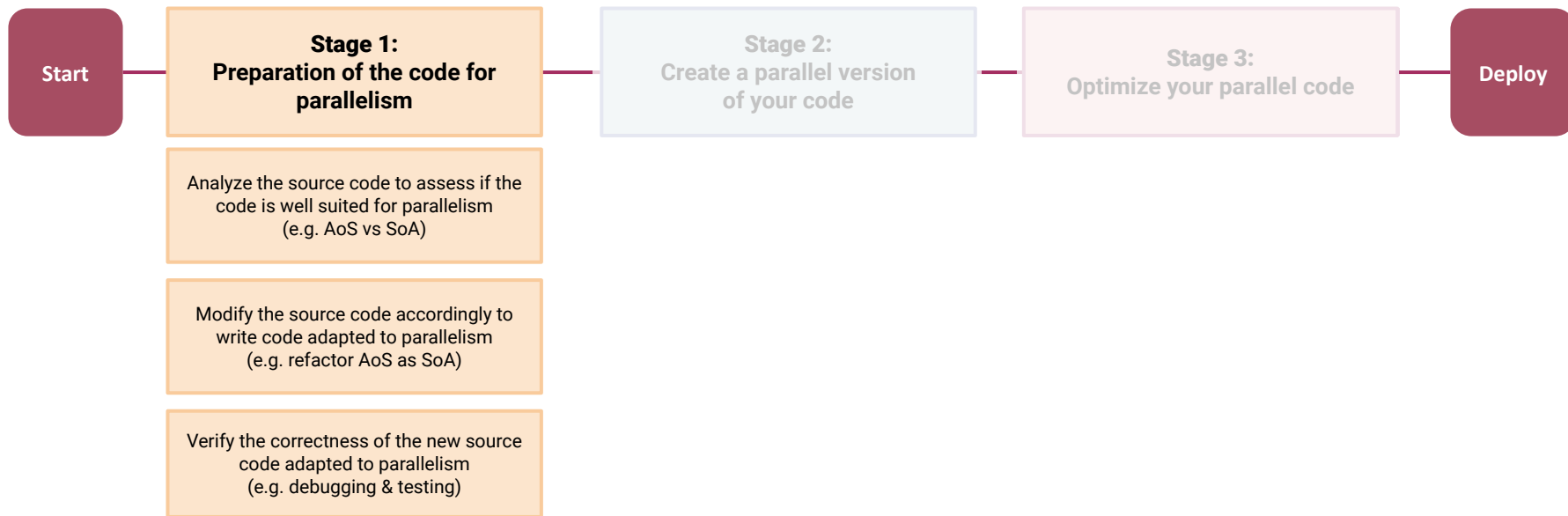
Parallel Programming Best Practices

“Develop parallel code using C/C++/Fortran targeting multicore CPUs and GPUs”



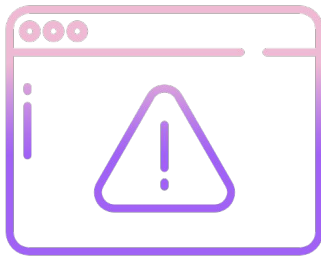
Stage 1: Prepare the code for parallelism

“Develop parallel code using C/C++/Fortran targeting multicore CPUs and GPUs”



PWR002: Declare scalar variables in the smallest possible scope

www.appentra.com/knowledge/checks/pwr002/



Prevent parallel bugs

Code example

In the following code, the function *foo* declares a variable *t* used in each iteration of the loop to hold a value that is then assigned to the array *result*. The variable *t* is not used outside of the loop.

```
1 void foo() {  
2     int t;  
3     int result[10];  
4  
5     for (int i = 0; i < 10; i++) {  
6         t = i + 1;  
7         result[i] = t;  
8     }  
9 }
```

In this code, the smallest possible scope for the variable *t* is within the loop body. The resulting code would be as follows:

```
1 void foo() {  
2     int result[10];  
3  
4     for (int i = 0; i < 10; i++) {  
5         int t = i;  
6         result[i] = t + 1;  
7     }  
8 }
```

From the perspective of parallel programming, moving the declaration of variable *t* to the smallest possible scope helps to prevent potential race conditions. For example, in the OpenMP parallel implementation shown below there is no need to use the clause *private(t)*, as the declaration scope of *t* inherently dictates that it is private to each thread. This avoids potential race conditions because each thread modifies its own copy of the variable *t*.

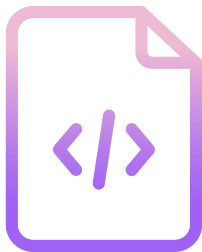
```
1 void foo() {  
2     int result[10];  
3  
4     #pragma omp parallel for default(none) shared(result)  
5     for (int i = 0; i < 10; i++) {  
6         int t = i;  
7         result[i] = t + 1;  
8     }  
9 }
```

Resources related to coding guidelines

- G.J. Holzmann (2006-06-19). "The Power of 10: Rules for Developing Safety-Critical Code". *IEEE Computer*. **39** (6): 95–99. doi:10.1109/MC.2006.212. See Rule 6: "Declare all data objects at the smallest possible level of scope". [last checked May 2019]

PWR016: Use separate arrays instead of an Array-of-Structs

www.appentra.com/knowledge/checks/pwr016/



Improve code coverage
of tools and compilers

Relevance

Using an Array-of-Structs (AoS) can increase cache misses unless all the fields are always accessed at the same time when iterating over the AoS. One example case where using an AoS is justifiable would be iterating over an array of points, each point being a struct containing the coordinates that are consumed on each iteration. However, most structs contain fields that will not be accessed together: data locality can be enhanced by breaking the struct and creating an array for each individual field.

Actions

Convert the Array-of-Structs (AoS) into separate plain arrays.

Code example

The following example shows a loop processing the x and y coordinates for an array of points:

```
1  typedef struct {
2      int x;
3      int y;
4      int z;
5  } point;
6
7  void foo() {
8      point points[1000];
9      for (int i = 0; i < 1000; i++) {
10         points[i].x = 1;
11         points[i].y = 1;
12     }
13 }
```

This could seem like an example where using an Array-of-Structs is justifiable. However, since the z coordinate is never accessed along the other two, there may be cache misses that could be avoided by creating one array for each coordinate:

```
1  void foo() {
2      int points_x[1000];
3      int points_y[1000];
4      int points_z[1000];
5      for (int i = 0; i < 1000; i++) {
6          points_x[i] = 1;
7          points_y[i] = 1;
8      }
9  }
```

The tool “pwreport”: Entry point and status

```
$ pwreport NPB3.3-OMP-C -- -I NPB3.3-OMP-C/common
```

```
Compiler flags: -I NPB3.3-OMP-C/common
```

CODE COVERAGE

```
Analyzable files:      78 / 79 ( 98.73 %)
Analyzable functions:  66 / 726 (  9.09 %)
Analyzable loops:     785 / 1588 ( 49.43 %)
Parallelized SLOCs:    0 / 0
```

METRICS SUMMARY

```
Total defects:      0
Total recommendations: 3597
Total opportunities: 205
Total data races:    0
Total data-race-free: 17
```

SUGGESTIONS

1 file could not be analyzed, get more information by enabling error reporting:

```
pwreport --show-failures NPB3.3-OMP-C -- -I NPB3.3-OMP-C/common
```

803 loops could not be analyzed, get more information with pwloops:

```
pwloops --non-analyzable NPB3.3-OMP-C -- -I NPB3.3-OMP-C/common
```

3597 recommendations were found in your code, get more information with pwcheck:

```
pwcheck --only-recommendations NPB3.3-OMP-C -- -I NPB3.3-OMP-C/common
```

205 opportunities for parallelization were found in your code, get more information with pwloops:

```
pwloops NPB3.3-OMP-C -- -I NPB3.3-OMP-C/common
```

```
78 files successfully analyzed and 1 failure in 12356 ms
```

The tool “**pwcheck**”: Defects and Recommendations

```
$ pwcheck NPB3.3-OMP-C -- -I NPB3.3-OMP-C/common
...
FUNCTION BEGIN at NPB3.3-OMP-C/common/randdp.c:vranlc:71:1
  71: void vranlc( int n, double *x, double a, double y[] )

      LOOP BEGIN at NPB3.3-OMP-C/common/randdp.c:vranlc:122:3
      122:   for ( i = 0; i < n; i++ ) {

          [PWR002] NPB3.3-OMP-C/common/randdp.c:108:34 'x1' not declared in the innermost scope possible
          108:   double t1, t2, t3, t4, a1, a2, x1, x2, z;

          [PWR002] NPB3.3-OMP-C/common/randdp.c:108:10 't1' not declared in the innermost scope possible
          108:   double t1, t2, t3, t4, a1, a2, x1, x2, z;

          [PWR002] NPB3.3-OMP-C/common/randdp.c:108:14 't2' not declared in the innermost scope possible
          108:   double t1, t2, t3, t4, a1, a2, x1, x2, z;

          [PWR002] NPB3.3-OMP-C/common/randdp.c:108:22 't4' not declared in the innermost scope possible
          108:   double t1, t2, t3, t4, a1, a2, x1, x2, z;

          [PWR002] NPB3.3-OMP-C/common/randdp.c:108:38 'x2' not declared in the innermost scope possible
          108:   double t1, t2, t3, t4, a1, a2, x1, x2, z;

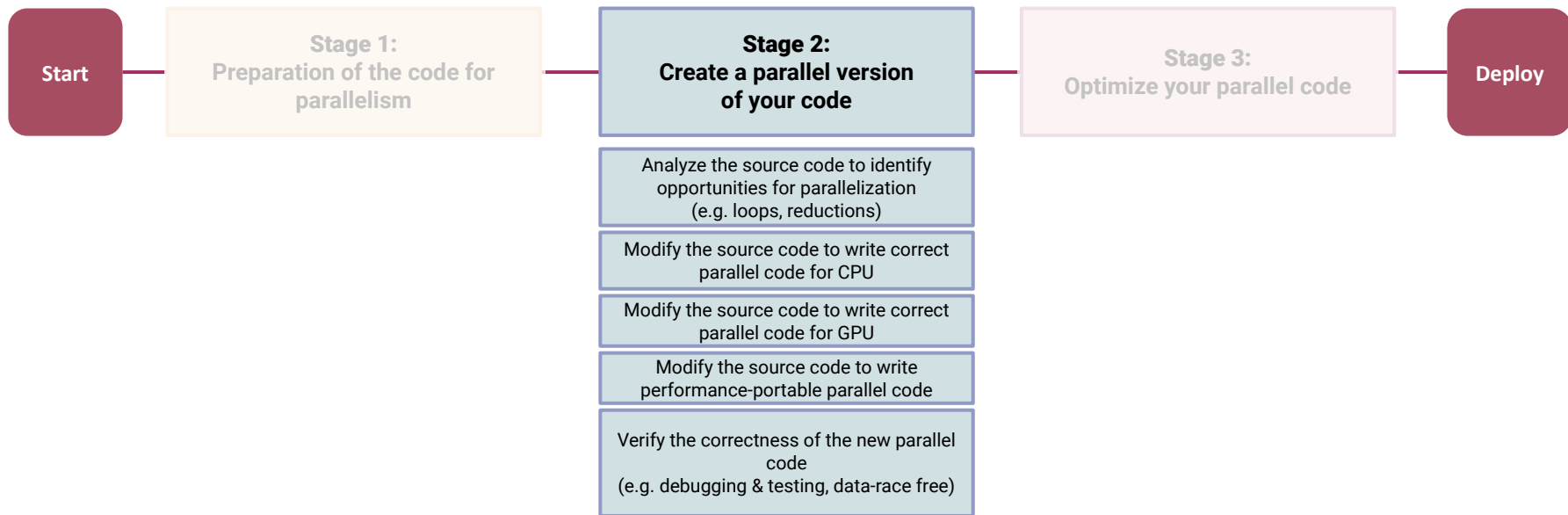
          [PWR002] NPB3.3-OMP-C/common/randdp.c:108:42 'z' not declared in the innermost scope possible
          108:   double t1, t2, t3, t4, a1, a2, x1, x2, z;

          [PWR002] NPB3.3-OMP-C/common/randdp.c:108:18 't3' not declared in the innermost scope possible
          108:   double t1, t2, t3, t4, a1, a2, x1, x2, z;
      LOOP END
FUNCTION END

...
PWR001  PWR002  PWR003  PWR004  PWR005  PWR009  PWR010  PWR011  PWR012  PWR013  PWR014  PWR015  PWR016  PWD001  PWD002  PWD003  PWD004  PWD005  PWD006  PWD007  PWD008
1433    1179      4      123      125      0      0      450      0      0      0      0      0      0      283      0      0      0      0      0      0
      0      0
Found a total of 3597 checks in 78 files successfully analyzed and 1 failure in 11489 ms
```

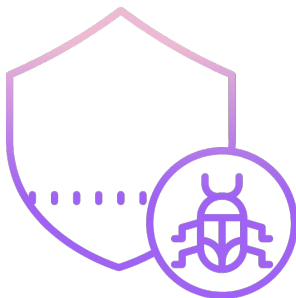
Stage 2: Create a parallel version of the code

“Develop parallel code using C/C++/Fortran targeting multicore CPUs and GPUs”



PWD001: Invalid OpenMP multithreading datascoping

www.appentra.com/knowledge/checks/pwd001/



Find & fix bugs

Definition

A variable is not being correctly handled in the OpenMP multithreading datascoping clauses.

Relevance

Specifying an invalid scope for a variable will most likely introduce a race condition, making the result of the code unpredictable. For instance, when a variable is written from parallel threads and the specified scoping is shared instead of private.

Actions

Set the proper scope for the variable.

Code example

The following code inadvertently shares the inner loop index variable *j* for all threads, which creates a race condition. This happens because a scoping has not been specified and it will be shared by default.

```
1 void foo() {  
2   int result[10][10];  
3   int i, j;  
4  
5   #pragma omp parallel for shared(result)  
6   for (i = 0; i < 10; i++) {  
7     for (j = 0; j < 10; j++) {  
8       result[i][j] = 0;  
9     }  
10  }  
11 }
```

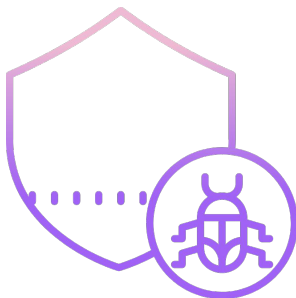
To fix this, the *j* variable must be declared private. The following code also specifies a private scope for *i* although in this case it is redundant since OpenMP automatically handles the parallelized loop index variable.

```
1 void foo() {  
2   int result[10][10];  
3   int i, j;  
4  
5   #pragma omp parallel for shared(result) private(i, j)  
6   for (i = 0; i < 10; i++) {  
7     for (j = 0; j < 10; j++) {  
8       result[i][j] = 0;  
9     }  
10  }  
11 }
```

PWD003:

Missing array range in data copy to the GPU

www.appentra.com/knowledge/checks/pwd003/



Find & fix bugs

Actions

Specify the array range to be copied to device memory.

Code example

In the following OpenMP code, a pointer is being copied to the offloading target device instead of the dynamic array data pointed by it.

```
1 void foo(int* a, int* b, int* sum, int size) {  
2     #pragma omp target map(to: a, b) map(from: sum)  
3     #pragma omp parallel for  
4     for (int i = 0; i < size; i++) {  
5         sum[i] = a[i] + b[i];  
6     }  
7 }
```

In this case, it suffices to specify the array bounds in the OpenMP map clauses:

```
1 void foo(int* a, int* b, int* sum, int size) {  
2     #pragma omp target map(to: a[0:size], b[0:size]) map(from: sum[0:size])  
3     #pragma omp parallel for  
4     for (int i = 0; i < size; i++) {  
5         sum[i] = a[i] + b[i];  
6     }  
7 }
```

The same applies to the analogous OpenACC example.

```
1 void foo(int* a, int* b, int* sum, int size) {  
2     #pragma acc data copyin(a, b) copyout(sum)  
3     #pragma acc parallel loop  
4     for (int i = 0; i < size; i++) {  
5         sum[i] = a[i] + b[i];  
6     }  
7 }
```

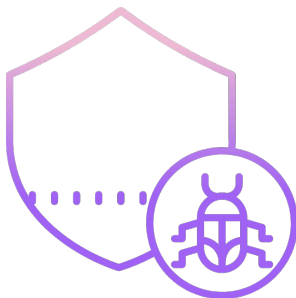
And again, specifying the array bounds fixes the problem:

```
1 void foo(int* a, int* b, int* sum, int size) {  
2     #pragma acc data copyin(a[0:size], b[0:size]) copyout(sum[0:size])  
3     #pragma acc parallel loop  
4     for (int i = 0; i < size; i++) {  
5         sum[i] = a[i] + b[i];  
6     }  
7 }
```

PWD004:

Out-of-memory-bounds array access

www.appentra.com/knowledge/checks/pwd004/



Find & fix bugs

Issue

A position outside the bounds of the array memory is being accessed which results in undefined behavior most likely causing invalid memory accesses and crashes.

Relevance

An array is essentially a collection of items that can be randomly accessed through an integer index. Obviously, only a subset of the possible integer values will correspond to array element positions; accessing an array using index values outside that subset will access a memory position not associated with any array element. This is called an out-of-memory-bounds access and has undefined behavior in C/C++, most likely causing invalid memory accesses and crashes.

Actions

Fix the array access so that only positions within the array memory bounds are accessed.

Code example

The following code uses an integer index ranging from 1 to 100 to access the array A:

```
1 void foo() {  
2     int A[100];  
3     for (int i = 0; i < 100; i++) {  
4         A[i + 1] = 1;  
5     }  
6 }
```

This is incorrect since the array positions range from 0 to 99. Thus, the array access must be fixed, for instance by changing the array reference from $A[i+1]$ to $A[i]$:

```
1 void foo() {  
2     int A[100];  
3     for (int i = 0; i < 100; i++) {  
4         A[i] = 1;  
5     }  
6 }
```

The tool “pwloops”: Parallelization Opportunities

```
$ pwloops NPB3.3-OMP-C --function rhs_norm -- -I NPB3.3-OMP-C/common
```

```
Compiler flags: -I NPB3.3-OMP-C/common
```

Loop	Analyzable	Compute patterns	Opportunity	Auto-Parallelizable	Parallelized
NPB3.3-OMP-C/SP/error.c					
- rhs_norm:95:3	x	forall	simd, multi	x	
- rhs_norm:102:5	x	forall			x
- rhs_norm:106:5	x	sparse			x
- rhs_norm:107:7	x	sparse			x
- rhs_norm:108:9	x	sparse			x
- rhs_norm:109:11	x	forall	simd	x	x
- rhs_norm:116:5	x	forall			x
- rhs_norm:122:3	x	n/a			
- rhs_norm:123:5	x	n/a			
NPB3.3-OMP-C/BT/error.c					
- rhs_norm:95:3	x	forall	simd, multi	x	
- rhs_norm:102:3	x	forall			x
- rhs_norm:106:3	x	sparse			x
- rhs_norm:107:5	x	sparse			x
- rhs_norm:108:7	x	sparse			x
- rhs_norm:109:9	x	forall	simd	x	x
- rhs_norm:116:3	x	forall			x
- rhs_norm:122:3	x	n/a			
- rhs_norm:123:5	x	n/a			

Loop : loop name following the syntax <file>:<function>:<line>:<column>

Analyzable : all C/C++/Fortran language features present in the loop are supported by Parallelware

Compute patterns : compute patterns found in the loop ('forall', 'scalar' or 'sparse' reduction, 'recurrence', 'dependency')

Opportunity : whether the loop is a parallelization opportunity and for which paradigms ('multi' for multi-threading or 'simd' for vectorization)

Auto-Parallelizable : loop can be parallelized by Parallelware

Parallelized : loop is already parallelized, for instance with OpenMP or OpenACC directives

SUGGESTIONS

Get more details about the data scoping of each variable within a loop, e.g.:

```
pwloops --datascopeing --loop NPB3.3-OMP-C/SP/error.c:rhs_norm:95:3 NPB3.3-OMP-C --function rhs_norm -- -I NPB3.3-OMP-C/common
```

Print the code annotated with opportunities, e.g.:

```
pwloops --code --function NPB3.3-OMP-C/SP/error.c:rhs_norm NPB3.3-OMP-C --function rhs_norm -- -I NPB3.3-OMP-C/common
```

Parallelize an auto-parallelizable loop, e.g.:

```
pwdirectives NPB3.3-OMP-C/SP/error.c:rhs_norm:95:3 -o <output_file> -- -I NPB3.3-OMP-C/common
```

Some file could not be analyzed, to get more details:

```
pwloops --show-failures NPB3.3-OMP-C --function rhs_norm -- -I NPB3.3-OMP-C/common
```

78 files successfully analyzed and 1 failure in 2465 ms

The tool “**pwdirectives**”: Parallel Code Generation

```
$ pwdirectives atmux.c:31 --out-file atmux_offload.c --omp offload -- -I lib
```

Compiler flags: -I lib

Results for file 'atmux.c':

Successfully parallelized loop at 'atmux.c:atmux:31:5' [using offloading]:

31:5: [INFO] Parallel sparse reduction pattern identified for variable 'y' with associative, commutative operator '+'

31:5: [INFO] Available parallelization strategies for variable 'y'

31:5: [INFO] #1 OpenMP atomic access (* implemented)

31:5: [INFO] #2 OpenMP explicit privatization

31:5: [INFO] Complete access range for variables: 'col_ind', 'val', 'y'

31:5: [INFO] Loop parallelized with teams using OpenMP directive 'target teams distribute parallel for'







Successfully created atmux_offload.c

```
$ sed -n 31,37p atmux_offload.c
```

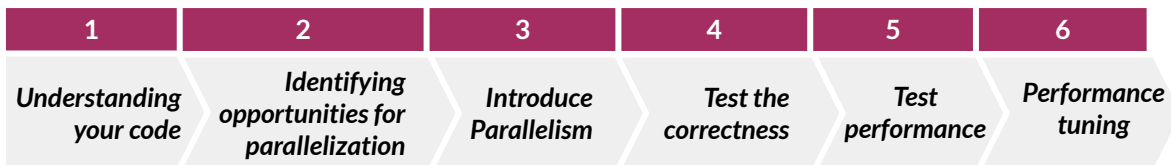
```
#pragma omp target teams distribute parallel for shared(col_ind, n, row_ptr, val, x) map(to: col_ind[:], n, row_ptr[0:n+1], val[:], x[0:n]) private(k)
map(tofrom: y[:]) schedule(auto)
for (i = 0; i < n; i++) {
    for (k = row_ptr[i]; k < row_ptr[i + 1]; k++) {
        #pragma omp atomic update
        y[col_ind[k]] = y[col_ind[k]] + x[i] * val[k];
    }
}
```

Parallelware Trainer



-  Parallelization opportunity
-  Non auto-parallelizable opportunity
-  Incomplete opportunity analysis
-  Failed opportunity analysis
-  Recommendation
-  Defect

Output Consoles



Parallel code generation with “pwdirectives”

Step 1:

- Find opportunities for parallelization in your code.
- Click on the “green circles”...



```
1 #include <math.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 double getClock();
6
7
8 int main(int argc, char *argv[]) {
9     if (argc != 2) {
10         printf("Usage: %s <steps>\n", argv[0]);
11         printf("  <steps> controls the precision of the approximation.\n");
12         return 0;
13     }
14
15     // Reads the test parameters from the command line
16     unsigned long N = atol(argv[1]);
17     printf("- Input parameters\n");
18     printf("steps\t= %lu\n", N);
19
20     printf("- Executing test...\n");
21     double time_start = getClock();
22     // =====
23
24     double out_result;
25
26
27     double sum = 0.0;
28     for (int i = 0; i < N; i++) {
29         double x = (1 + 0.5) / N;
30         sum += sqrt(1 - x * x);
31     }
32
33     out_result = 4.0 / N * sum;
34
35     // =====
36     double time_finish = getClock();
37
38     // Prints an execution report
39     printf("time (s)= %.6f\n", time_finish - time_start);
40     printf("result\t= %.8f\n", out_result);
41     const double realPiValue = 3.141592653589793238;
42     printf("error\t= %.1e\n", fabs(out_result - realPiValue));
43 }
```

Generate directives with Parallelware Trainer GUI

Step 1:

- Find opportunities for parallelization in your code.
- Click on the “green circles”...

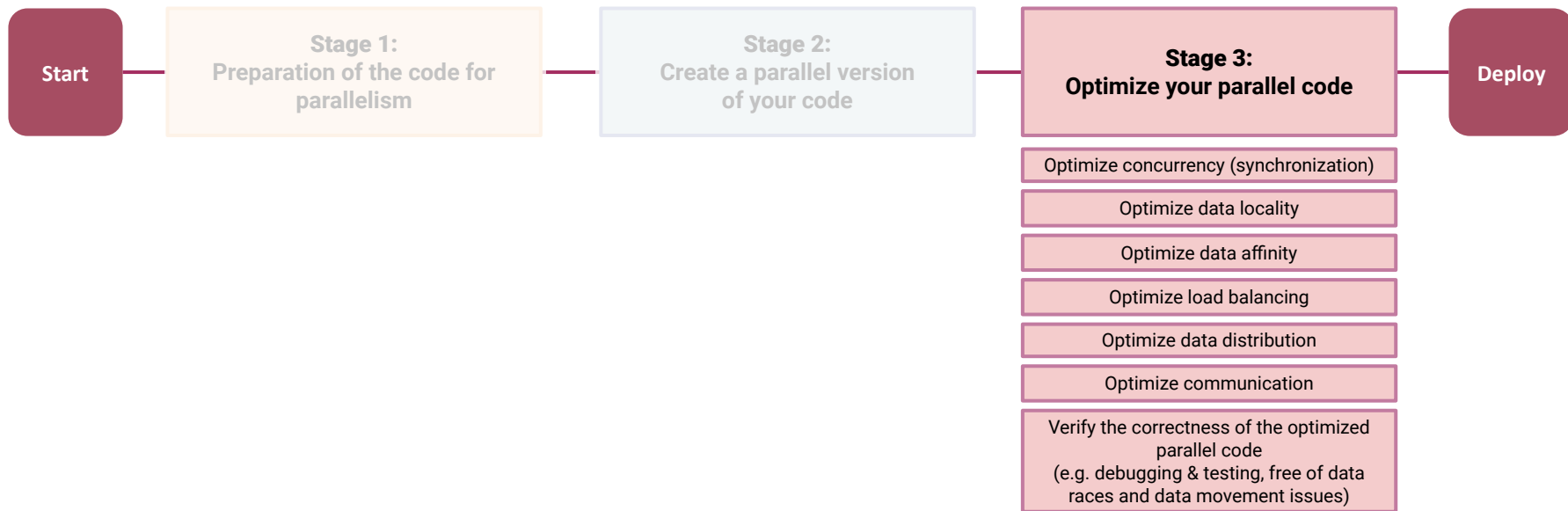
Step 2:

- Select a setup: Standard, Device & Paradigm
- Click on the button “Parallelize”...

The screenshot shows the 'Parallelization options' dialog box. It has a title bar with a close button. The dialog is divided into several sections: 'Standard' with radio buttons for 'OpenMP' (selected) and 'OpenACC'; 'Device' with radio buttons for 'CPU' (selected) and 'GPU'; 'Paradigm' with radio buttons for 'Multithreading' (selected), 'Offloading', and 'Tasking' (with a dropdown menu set to 'with taskloop'); 'Parallel reduction variables' with three text input fields for 'Atomic protection', 'Built-in reduction', and 'Explicit privatization'; and 'Ranges for array variables' with a text input field for 'Array ranges'. At the bottom, there are 'Cancel' and 'Parallelize' buttons. Four grey arrows point to the 'Standard', 'Device', 'Paradigm', and 'Parallelize' button sections, corresponding to the steps in the text.

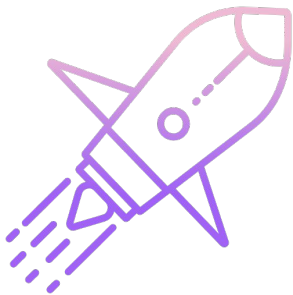
Stage 3: Optimize parallel code

“Develop parallel code using C/C++/Fortran targeting multicore CPUs and GPUs”



PWR010: Avoid column-major array access in C/C++

www.appentra.com/knowledge/checks/pwr010/



Optimize performance

Relevance

The most efficient way to process arrays is to iterate over its elements in the same order in which they are laid out in memory, so that the program performs a sequential access to consecutive data in memory. The C and C++ language specifications state that arrays are laid out in memory in a row-major order: the elements of the first row are laid out consecutively in memory, followed by the elements of the second row, and so on. As a result, in order to maximize performance C and C++ code should access multi-dimensional arrays using a row-major order.

Actions

Change the code to access the multi-dimensional array in a row-major order.

Code example

In the following code, an outer loop iterates over the columns of a bidimensional array and then an inner loop iterates over the rows for each one of those columns.

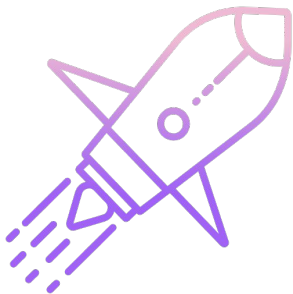
```
1  #define ROWS 100
2  #define COLS 100
3
4  void foo() {
5      int A[ROWS][COLS];
6
7      for (int j = 0; j < COLS; ++j) {
8          for (int i = 0; i < ROWS; ++i) {
9              A[i][j] = i + j;
10         }
11     }
12 }
```

This way of iterating and accessing the elements of the array doesn't match its layout in memory. The optimal way is to iterate over the rows sequentially, then do the same for each column within the row:

```
1  #define ROWS 100
2  #define COLS 100
3
4  void foo() {
5      int A[ROWS][COLS];
6
7      for (int i = 0; i < ROWS; ++i) {
8          for (int j = 0; j < COLS; ++j) {
9              A[i][j] = i + j;
10         }
11     }
12 }
```

PWR009: Use OpenMP teams to offload work to GPU

www.appentra.com/knowledge/checks/pwr009/



Optimize performance

Code example

The following code offloads a matrix multiplication computation through the *target* construct and then creates a parallel region and distributes the work through *for* construct (note that the matrices are statically sized arrays):

```
1  #pragma omp target map(to: A[0:m][0:p], B[0:p][0:n], m, n, p) map(tofrom: C[0:m][0:n])
2  {
3  #pragma omp parallel default(none) shared(A, B, C, m, n, p)
4  {
5  #pragma omp for schedule(auto)
6  for (size_t i = 0; i < m; i++) {
7      for (size_t j = 0; j < n; j++) {
8          for (size_t k = 0; k < p; k++) {
9              C[i][j] += A[i][k] * B[k][j];
10         }
11     }
12 }
13 } // end parallel
14 } // end target
```

When offloading to the GPU it is recommended to use an additional level of parallelism. This can be achieved by using the *teams* and *distribute* constructs, in this case in combination with *parallel for*:

```
1  #pragma omp target teams distribute parallel for map(to: A[0:m][0:p], B[0:p]
2  [0:n], m, n, p) shared(A, B, m, n, p) map(tofrom: C[0:m][0:n]) schedule(auto)
3  for (size_t i = 0; i < m; i++) {
4      for (size_t j = 0; j < n; j++) {
5          for (size_t k = 0; k < p; k++) {
6              C[i][j] += A[i][k] * B[k][j];
7          }
8      }
9  }
```

Related resources

- PWR009 examples at GitHub
- OpenMP 4.5 Complete Specifications, November 2015 [last checked June 2020]
- Portability of OpenMP Offload Directives – Jeff Larkin, OpenMP Booth Talk SC17, November 2017 [last checked June 2020]
- OpenMP and NVIDIA – Jeff Larkin, NVIDIA Developer Technologies [last checked June 2020]

Parallelware Tools: Software Performance Optimization



www.appentra.com



Sign up for our newsletter: appentra.com/newsletter/



Email us at: info@appentra.com