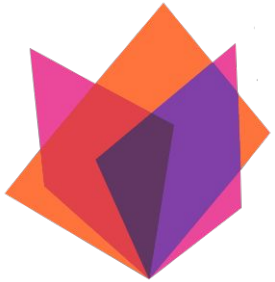
OpenMP

SC'20 Booth Talk Series



Parallelware Analyzer: Data race detection for GPUs using OpenMP

Manuel Arenaz, Appentra

Major challenges in GPU programming

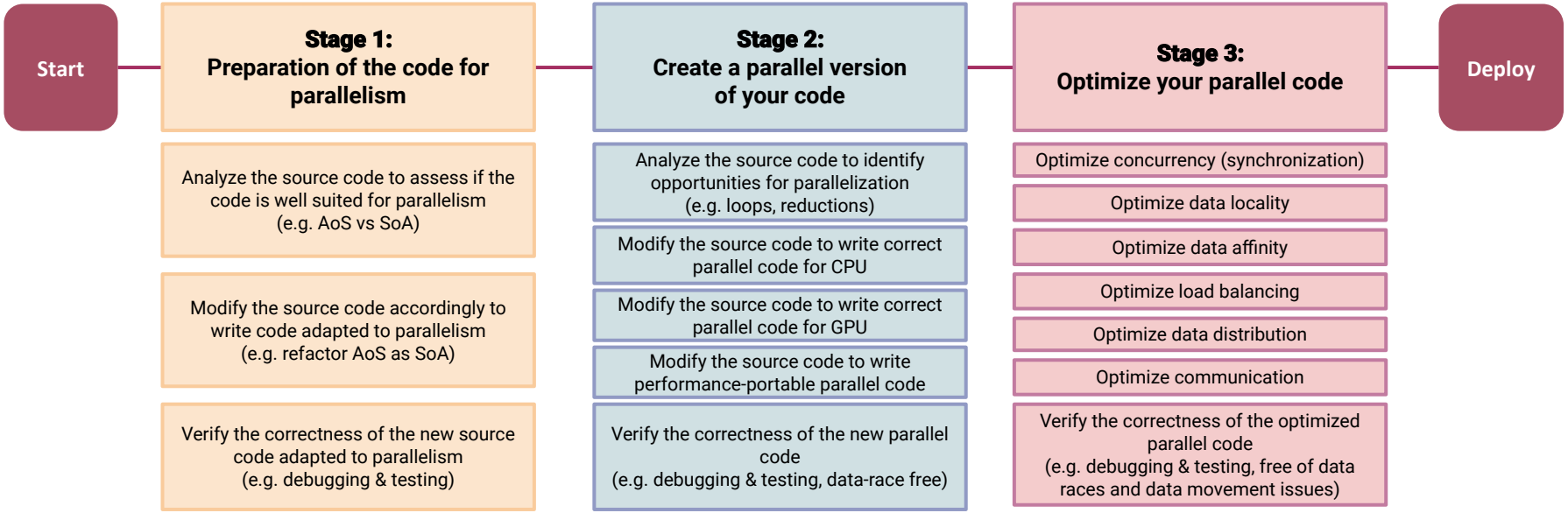
- The development and maintenance of **bug-free C/C++/Fortran parallel code** is far more complex than that of sequential software.
- **Parallel bugs are difficult to find and fix** because a buggy parallel code might run correctly 99% of the time and fail just the remaining 1%.
- This is **even more difficult for Graphical Processing Units (GPUs)**.
- In order to take advantage of the performance promised by 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 GPU programmers?

- **GPU programming is very hard and very intrusive** because it usually requires major changes in the code
- Major efforts by the 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 offloaded to the GPU
  - **#4: Implement** versions of the code for the GPU
- **New Dev tools to improve programmer's productivity on 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

'INNOVATIVE SCIENCE' CATEGORY 2020

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 Analyzer tool ›

PARALLELWARE ANALYZER

+

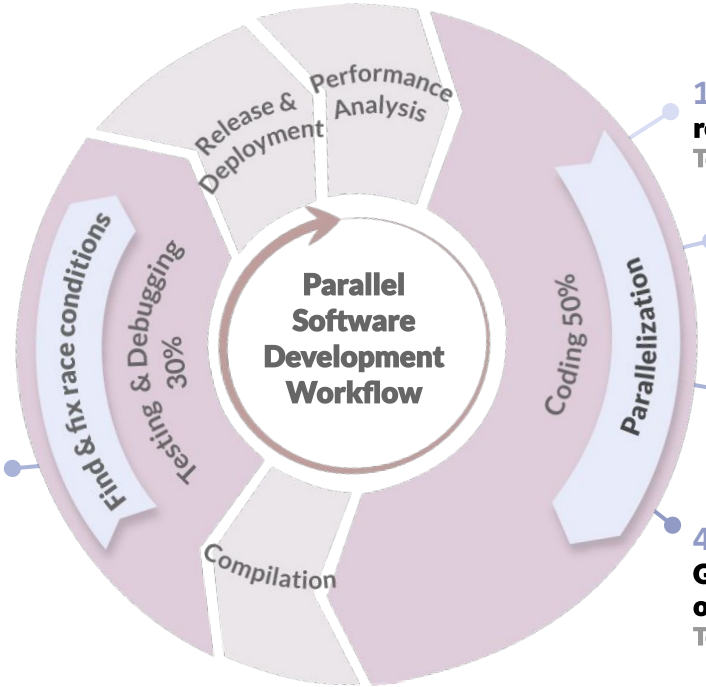
Open catalog of defects & recommendations

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 ›

appentra make code parallel

Parallelware Analyzer



1. **High-level overview of your code: summary of parallelized regions, defects, recommendations and opportunities.**
Tool "pwreport"

2. **Identify defects related to parallelism while coding.**
Tool "pwcheck"

3. **Identify opportunities for parallelization.**
Tool "pwloops"

4. **Guided generation of parallel code for multicore CPUs and GPUs, with OpenMP and OpenACC, using multithreading, offloading and tasking.**
Tool "pwdirectives"

2. **Identify defects related to parallelism while debugging**
Tool "pwcheck"

INNOVATION RADAR Prize "INNOVATIVE SCIENCE" CATEGORY 2020

SC18 Dallas, TX | hpc inspires.
Emerging Technologies

EPEEC WWW.EPEEC-PROJECT.EU
MAESTRO WWW.MAESTRO-DATA.EU
OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY
appentra make code parallel

The tool "pwreport": Entry point and status

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

34 files successfully analyzed and 1 failure in 21394 ms (pass --show-failures for error details)

CODE COVERAGE
  Analyzable files:              34 / 35 (97.14 %)
  Analyzable functions:          56 / 332 (16.87 %)
  Analyzable loops:              753 / 1010 (74.55 %)
  Parallelized SLOCs:            5597 / 16822 (33.27 %)

SUMMARY
  Total defects:                 0
  Total recommendations:         702
  Total opportunities:           121
  Total data races:              0
  Total data-race-free:          27

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

  257 loops could not be analyzed, get more information with pwloops:
    pwloops --non-analyzable NPB3.3-OMP-C -- -I NPB3.3-OMP-C/common

  702 recommendations were found in your code, get more information with pwcheck:
    pwcheck --only-recommendations NPB3.3-OMP-C -- -I NPB3.3-OMP-C/common

  121 opportunities for parallelization were found in your code, get more information with pwloops:
    pwloops NPB3.3-OMP-C -- -I NPB3.3-OMP-C/common

  5597 lines of code in parallel regions were found in your code, get more information with pwreport:
```

The tool “pwcheck”: Defects and Recommendations

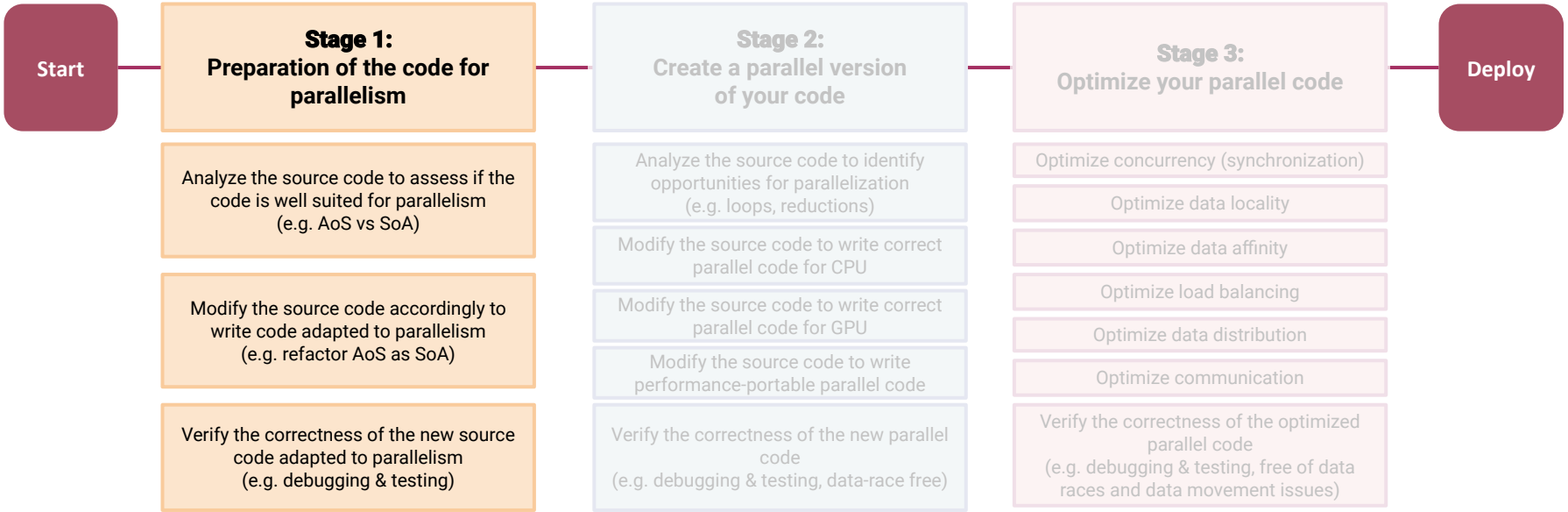
```
$ pwcheck --help
Syntax: pwcheck [options] <source files/directories> [-- <compiler flags>]

General options:
  ...
  -d [ --only-defects ]            Report only defects (ignore recommendations)
  -r [ --only-recommendations ]  Report only recommendations (ignore defects)
  -c [ --races ]                   Data race analysis reporting with format:
                                   <file>,{D|N|F|U|E}
                                   - D: data race detected
                                   - N: no data race detected
                                   - F: free of data races
                                   - U: could not determine whether there is a
                                   data race or not- E: there was an error
                                   analyzing the file

Available checks (more info at https://www.appentra.com/knowledge/checks/):
  PWR001: Declare global variables as function parameter                    C, Fortran
  PWR002: Declare scalar variables in the smallest possible scope           C
  PWR003: Explicitly declare pure functions                                 C
  PWR004: Declare OpenMP scoping for all variables                          C, Fortran
  PWR005: Disable default OpenMP scoping                                    C, Fortran
  PWR006: Avoid privatization of read-only variables                        (disabled)
  PWR007: Disable implicit declaration of variables                         Fortran
  PWR008: Declare the intent for each procedure parameter                   Fortran
  PWR009: Prefer OpenMP 'teams distribute' over 'parallel' offload          C
  PWR010: Avoid column-major array access in C/C++                          C
  PWR011: Outline loop to increase compiler and tooling code coverage       C
  PWD001: Invalid OpenMP multithreading datascoping                         C
  PWD002: Unprotected multithreading reduction operation                    C
  PWD003: Missing array range in data copy to accelerator device            C, Fortran
```

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*.

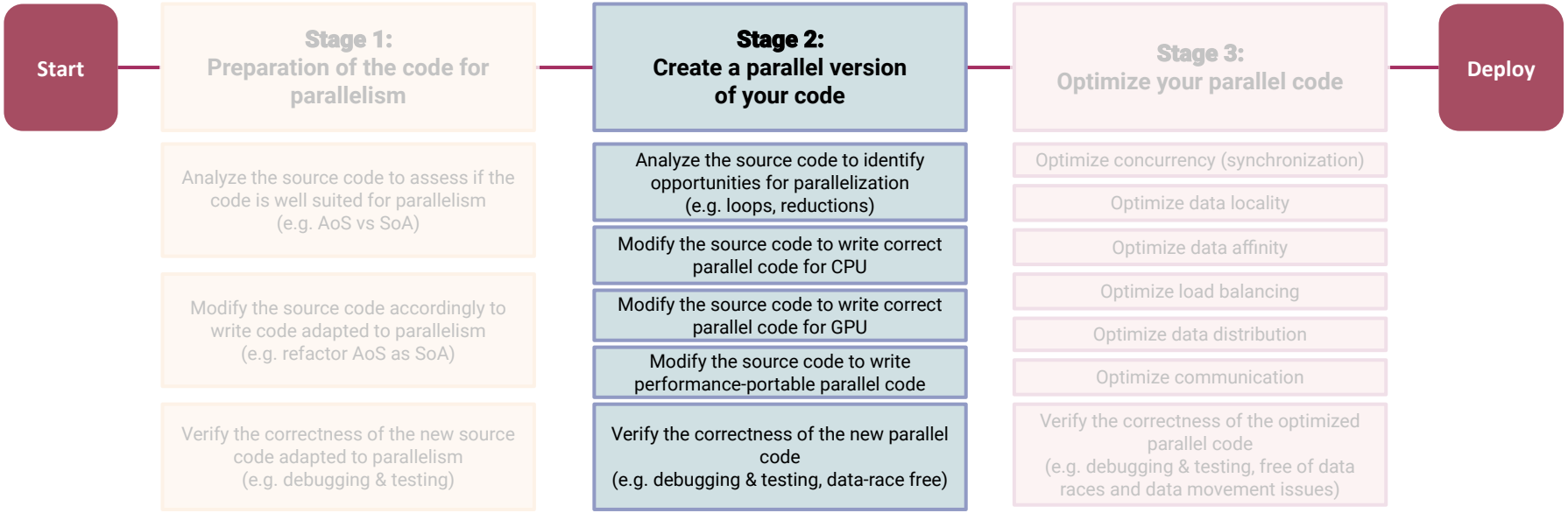
```
1  void foo() {
2    int result[10];
3
4    #pragma omp parallel for default(none) shared(result)
5    for (int i = 0; i < 10; i++) {
6      int t = i;
7      result[i] = t + 1;
8    }
9  }
```

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]

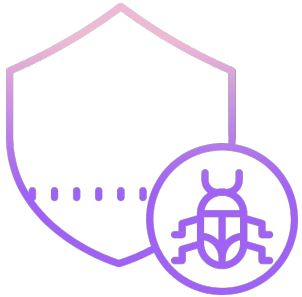
Stage 2: Create a parallel version of the code

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



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

www.appentra.com/knowledge/checks/pwd006/



Find & fix bugs

Code example

The following OpenMP code declares that the bi-dimensional array *A* should be copied to the accelerator device (see the clause *map(tofrom:A)* and the data type *int*** of the array *A*). However, this is incorrect and will not copy the data to be accessed in the loop body because OpenMP treats the pointer *A* as a zero-length array. Thus, the actual data of the variable will not be copied. As a result, dereferencing A in the GPU will cause invalid memory accesses, since its data has not been copied.

```
1  void foo(int **A) {
2    #pragma omp target teams distribute parallel for map(tofrom:A)
3    for (size_t i = 0; i < 10; i++) {
4     A[i][i] += i;
5    }
6  }
```

Adding the array ranges (see *map(tofrom:A[0:10][0:10])*) could be seen as a solution:

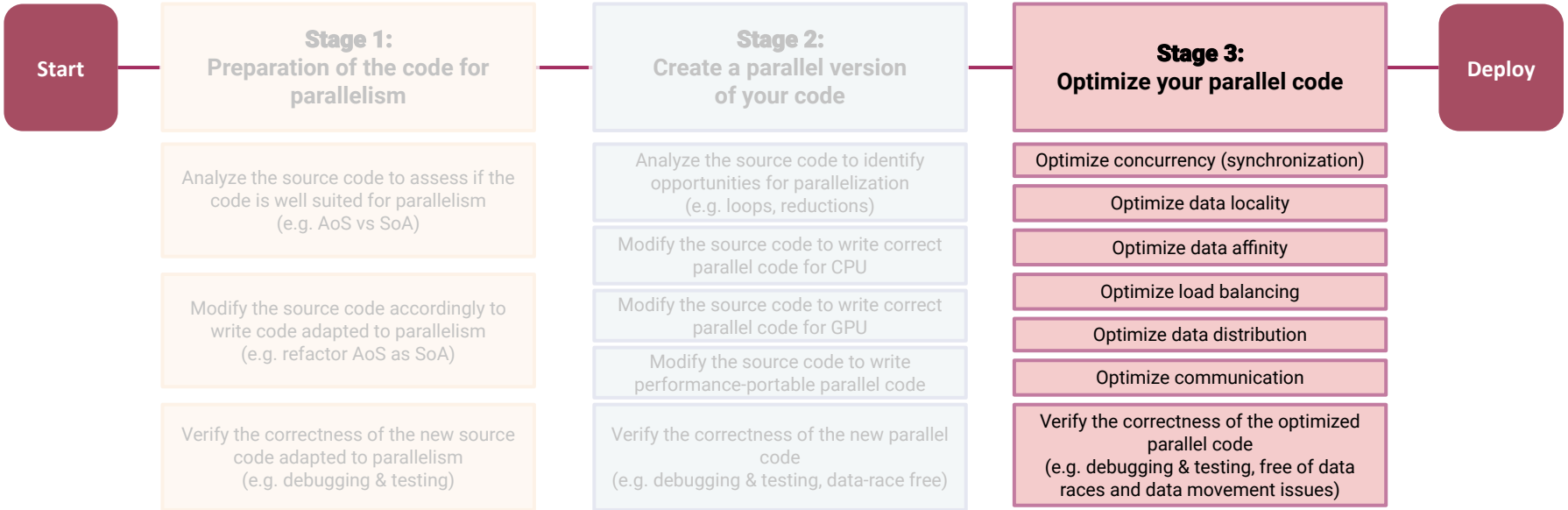
```
1  void foo(int **A) {
2    #pragma omp target teams distribute parallel for map(tofrom:A[0:10][0:10])
3    for (size_t i = 0; i < 10; i++) {
4     A[i][i] += i;
5    }
6  }
```

However, this OpenMP code does not handle non-contiguous memory properly because deep copy is not automatically supported. Therefore, each contiguous memory segment must be individually mapped to the accelerator device. This can be done through OpenMP 4.5 *enter/exit data* execution statements as follows:

```
1  void foo(int **A) {
2    #pragma omp target enter data map(to:A[0:10])
3    for (size_t i = 0; i < 10; i++) {
4     #pragma omp target enter data map(to:A[i][0:10])
5    }
6
7    #pragma omp target teams distribute parallel for
8    for (int i = 0; i < 10; i++) {
9     A[i][i] += i;
10   }
11
12   for (size_t i = 0; i < 10; i++) {
13    #pragma omp target exit data map(from:A[i][0:10])
14   }
15   #pragma omp target exit data map(from:A[0:10])
16 }
```

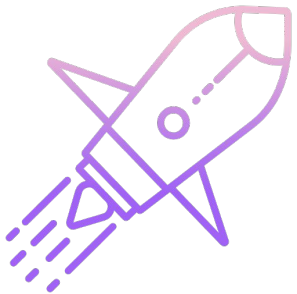
Stage 3: Optimize parallel code

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



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:

```
#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
{
#pragma omp parallel default(none) shared(A, B, C, m, n, p)
{
#pragma omp for schedule(auto)
for (size_t i = 0; i < m; i++) {
   for (size_t j = 0; j < n; j++) {
      for (size_t k = 0; k < p; k++) {
         C[i][j] += A[i][k] * B[k][j];
      }
   }
}
} // end parallel
} // 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*:

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

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]

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

Download the PDF

Defects

PWD001: Invalid OpenMP multithreading datascoping

PWD002: Unprotected multithreading reduction operation

PWD003: Missing array range in data copy to accelerator device

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

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

PWR013: Avoid copying unused variables to the GPU

appentra
make code parallel

OpenMP

SC'20 Booth Talk Series

openmp.org OpenMP API specs, forum, reference guides, and more

link.openmp.org/sc20 Videos and PDFs of OpenMP SC'20 presentations