appentra
make code parallel

OpenMP Best Practices with Parallelware Analyzer

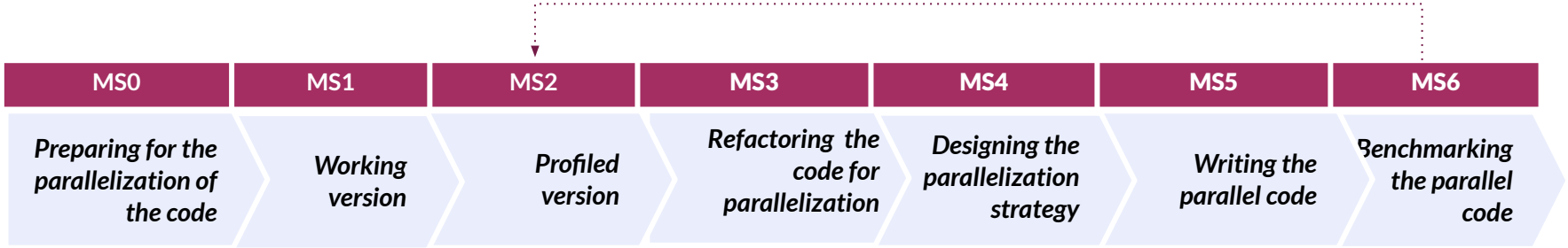
Manuel Arenaz | November 20, 2019

appentra
make code parallel

Value proposition of Parallelware Analyzer?

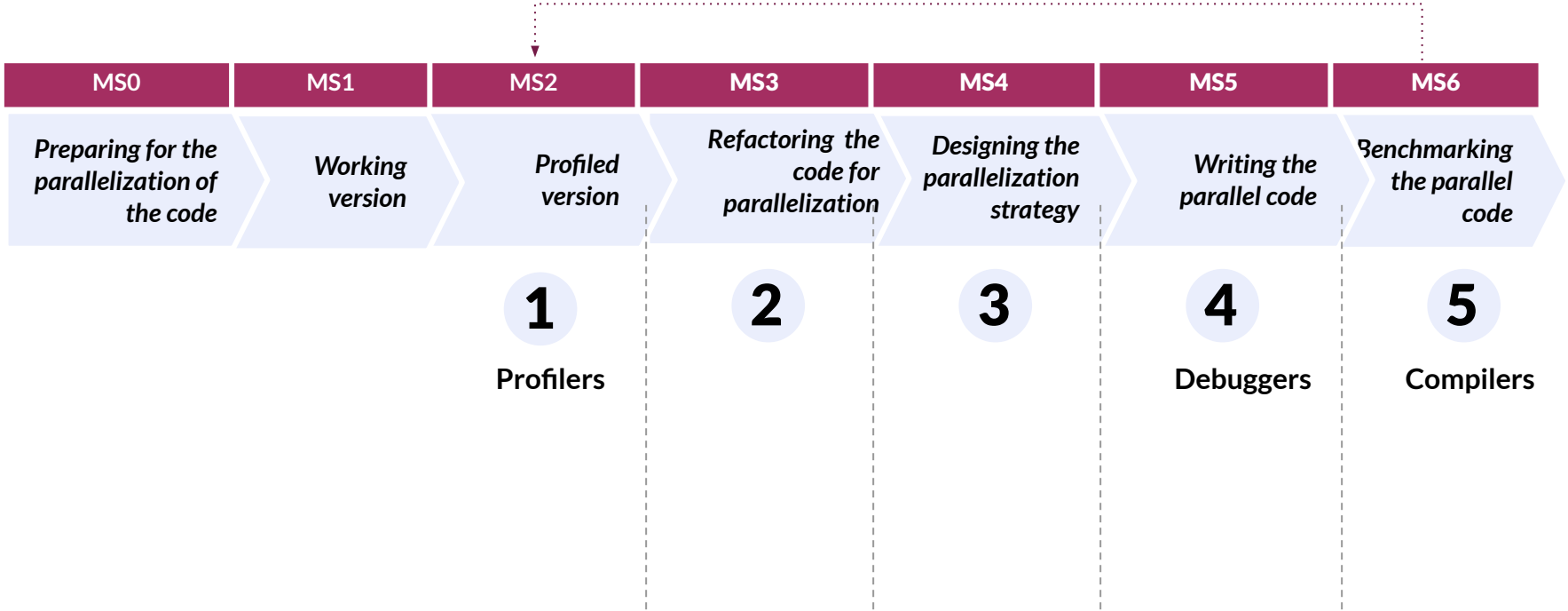
No.	Value	Metric
1	**Accelerate the runtime of the software** - Legacy software that needs to be modernised to run on modern hardware - sequential software to be parallelized - parallel software to exploit more parallelism - parallel software ported to a different hardware (eg. CPU to GPU)	**Speedup**
2	**Train customer's workforce** - developer/maintainer learns and enforces best practices for development of parallel software - development/maintainer applies code changes to clearly understand their value from the point of view of parallelization	**Defects** (eg. race conditions) **Recommendations** (eg. best practices) **Parallelization opportunities** **Parallelization strategies** **Parallelization tools**

Development methodology: Step-by-step



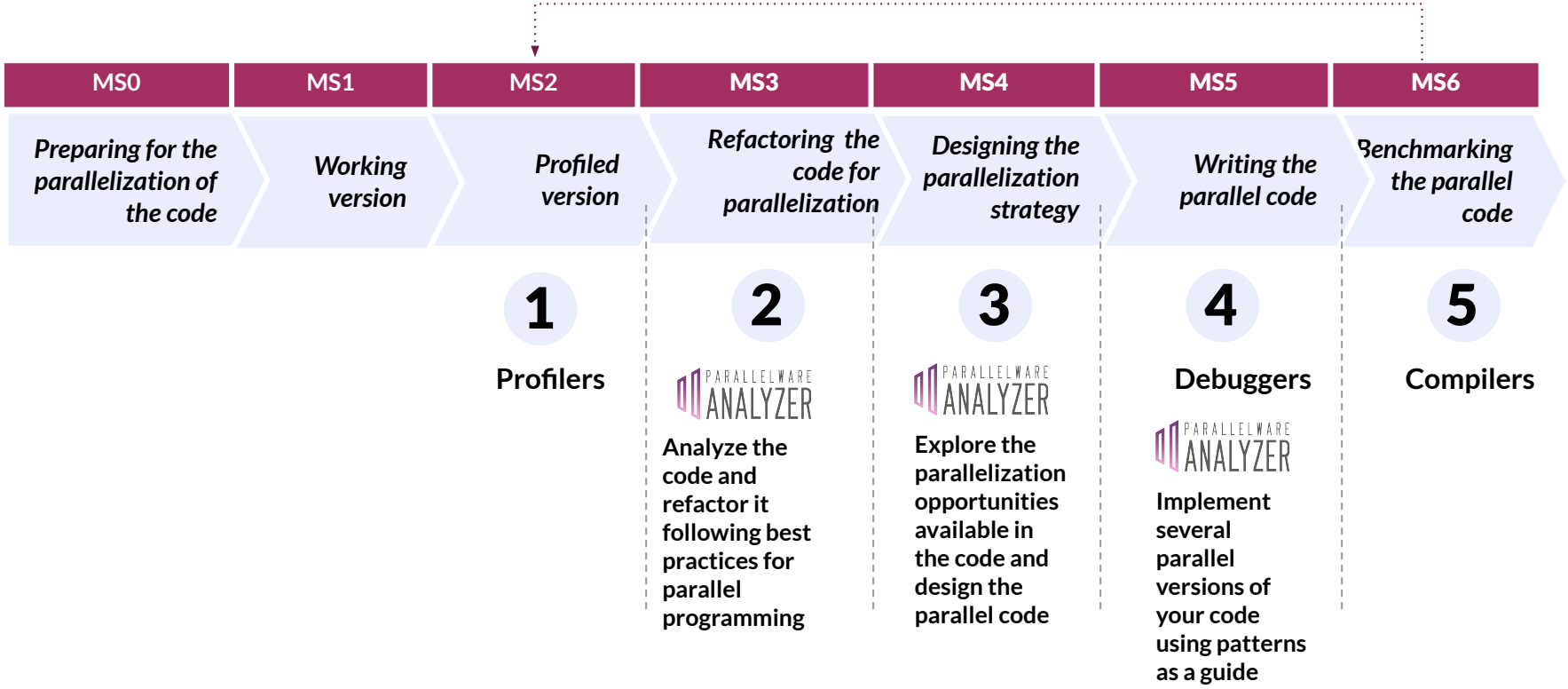
appentra
make code parallel

Development methodology: Step-by-step



appentra
make code parallel

Development methodology: Step-by-step



appentra
make code parallel

Capabilities of Parallelware Analyzer

- Detection of defects in parallel code, i.e. race conditions not detected yet
  - Static data race detection for multicore CPUs and GPUs
  - See https://www.appentra.com/knowledge/checks/
- Enforce best practices for parallel programming through suggestions for code refactorizations
  - See https://www.appentra.com/knowledge/checks/
- Discover parallelization opportunities through in-depth static code analysis
  - Source code analysis guided through code patterns, not line by line
  - See https://www.appentra.com/knowledge/patterns/
- Quickly design and implement parallel code for CPU/GPU using OpenMP/OpenACC
  - Parallel code implementation guided by code patterns, not “happy idea”
  - See https://www.appentra.com/knowledge/patterns/

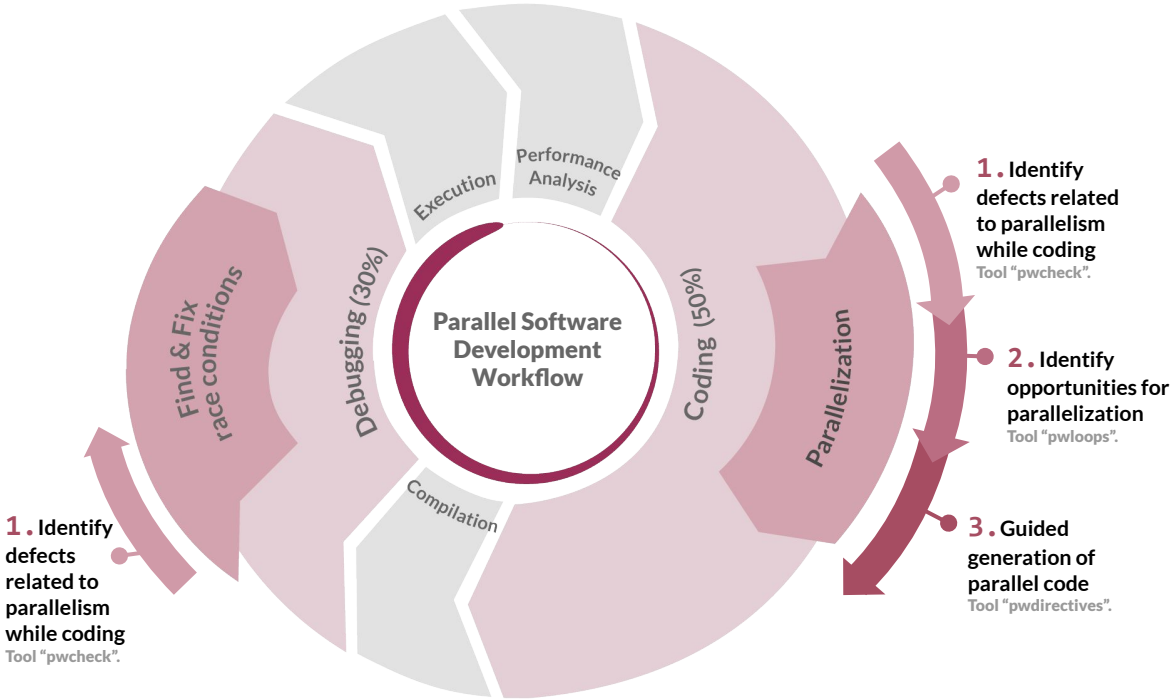
Appentra approach is to break down best practices into items that are objective, measurable and actionable

Parallelware Analyzer *(Beta)*

Emerging Technologies

SC18 Dallas, TX hpc inspires.

European Commission
Horizon 2020
European Union funding for Research & Innovation
SEAL OF EXCELLENCE



appentra
make code parallel

NAS Parallel Benchmarks: "pwcheck" summary

```
(base) javi@javi-u18:~$ pwcheck --summary NPB3.3-SER-C/ -- -I NPB3.3-SER-C/common/
Compiler flags: -I NPB3.3-SER-C/common/

Found a total of 1160 checks in 80 files successfully analyzed and 1 failures in 4472 ms:

CODE COVERAGE
  Analyzable files:              80 / 81 (98.77 %)
  Analyzable functions:          62 / 391 (15.86 %)
  Analyzable loops:             634 / 1343 (47.21 %)

SUMMARY
  Total defects:                  0
  Total recommendations:       1160
  Total opportunities:         465

SUGGESTIONS

  709 loops could not be analyzed, run pwloops to get more information about them:
        pwloops --non-analyzable NPB3.3-SER-C/ -- -I NPB3.3-SER-C/common/

  1160 recommendations were found in your code, re-run pwcheck without --summary to get details:
        pwcheck --only-recommendations NPB3.3-SER-C/ -- -I NPB3.3-SER-C/common/

  465 opportunities for parallelization were found in your code, run pwloops to get more information about them:
        pwloops NPB3.3-SER-C/ -- -I NPB3.3-SER-C/common/
```

Knowledge Base of Defects & Recommendations

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

Static code analysis tools are designed to aid software developers to build better quality software in less time, by promoting best practices and detecting defects early in the software development life cycle. A defect can lead to a minor malfunction or cause serious security and safety issues. Thus, identifying, mitigating and resolving defects is an essential part of the software development process.

Parallelware Analyzer reports code defects that constitute errors and issues recommendations to adopt best practices that prevent errors related to concurrency and parallelism.

Defects

- **PWD001**: Invalid OpenMP multithreading datascoping
- **PWD002**: Unprotected multithreading reduction operation

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

PWD001: Invalid OpenMP multithreading datascoping

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 }
```

PWR002: Declare scalar variables in the smallest possible scope

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]

Proof Points of Parallelware Analyzer

- **Parallelware Analyzer needed only 7 minutes to complete the analysis of 2M SLOCs** spread across more than 15K sources files.
- **The average run of Parallelware Analyser needed only 3 seconds** to complete the analysis of 15K SLOCs.

	Average	Total
Source files	111	15.439
SLOCs	15.410	2.141.958
Execution time (seconds)	3	418
Checks found	9,6	10.630

[Source Appentra; Statistics corresponding to 139 benchmarks from well-known public benchmark suites, including DSPstone, Patmos, MiBench, NPB, CORAL, TSVC.]

Parallelware Analyzer needed only 4.6 seconds for 86 of the 116 DataRaceBench micro-benchmarks (6x-1841x faster than dynamic data race detection tools)

Parallelware Analyzer is successfully detecting the first data races statically, including first GPU codes. Next step is to catch up with codes covered by dynamic data race detection tools.

PARALLELWARE
make code parallel

appentra
make code parallel

appentra company/appentra/ Appentra

www.appentra.com

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

Email us at: info@appentra.com

PARALLELWARE TRAINER

Try Parallelware Trainer:
appentra.com/products/parallelware-trainer

PARALLELWARE ANALYZER

Apply for Early Access Program:
appentra.com/products/parallelware-analyzer