Parallelware Analyzer NPB Quickstart

Event agenda:

https://www.bsc.es/education/training/patc-courses/patc-heterogeneous-programming-gpus-mpi-ompss-1/agenda

What is Parallelware Analyzer?

Parallelware Analyzer is a suite of command-line tools aimed at helping software developers build better quality parallel software in less time by leveraging Parallelware static code analysis technology. Designed around their needs, Parallelware Analyzer provides the appropriate tools for the key stages of the parallel development workflow.

Current version of Parallelware Analyzer provides the following command-line tools:

- **pwreport**: provides high-level reports of the code which can be exported in JSON format.
- **pwcheck**: looks for **defects** such as race-conditions and issues **recommendations** on best-practices.
- **pwloops**: provides insight into the **parallel properties of loops** found in the code which constitute opportunities for parallelism.
- **pwdirectives**: helps to insert **OpenMP and OpenACC directives** in your code to create parallel versions.

Where to start?

Parallelware Analyzer tools have been designed to cover the different stages of the parallel development workflow from looking for opportunities for parallelism to implementing them. The **pwreport** tool normally serves as a good entry point since it provides the higher-level reporting, including code coverage metrics, opportunities for parallelization and defects found in your code that should be fixed right away. It also offers suggestions on which tools to invoke next.

Useful options common to all tools

--help: print usage information.
--brief: all analyses support outputting a more compact version.

--show-progress: reports analysis progress per file, which can be useful when analyzing folders containing many source files since it can take some time.
--show-failures: prints error messages for files that could not be analyzed. This is useful to detect missing required compiler flags (such as include paths when header file errors are reported).

Quickstart with NAS Parallel Benchmarks

To get started with Parallelware Analyzer we will use the SNU NPB Suite, which is a set of the NAS Parallel Benchmarks (NPB) in C with four implementations: serial version (NPB-SER-C), OpenMP version (NPB-OMP-C), OpenCL version (NPB-OCL) and OpenCL for multiple devices version (NPB-OCL-MD). Specifically, we will work with the serial implementation: **NPB-SER-C**.

1. Download Parallelware Analyzer and its license file, then uncompress and move inside the license file with name *pwa.lic*:

```
$ tar xvfz pwanalyzer-0.13.0-ca47ead_linux-x64.tgz
$ mv pwanalyzer-Mar20-PATC.lic
pwanalyzer-0.13.0-ca47ead_linux-x64/pwa.lic
```

2. Download and uncompress the NPB suite: *SNU_NPB-1.0.3.tar.gz*

```
$ tar xvfz SNU_NPB-1.0.3.tar.gz
```

3. Move into the serial implementation:

```
$ cd SNU_NPB-1.0.3/NPB3.3-SER-C
```

4. Choose a benchmark and build it, for instance for BT:

```
$ make bt
```

> Note that Parallelware Analyzer only analyzes source code and does not require the execution of the benchmark. Invoking make is needed to create the executables of the NPB benchmarks and because a *npbparams.h* file is generated for each benchmark upon building.

5. Execute the selected benchmark to measure the performance of the serial version, for instance for BT:

```
$ bin/bt.S.x
...
 Time in seconds =                0.06
...
 Verification    =         SUCCESSFUL
...
```

> The *S* in the binary name represents the NPB workload class, in this case the smallest one. Look for the outputted correctness and runtime information.

6. Run **pwreport** to get a first overview of the code:

```
$ pwreport BT/*.c
```

> Some files fail due to missing includes which are located into the *common* folder. You should pay attention to the suggestions outputted by the different tools. In this case you are instructed to use the `--show-failures` flags.

7. Re-run with the `--show-failures` flag:

```
$ pwreport BT/*.c --show-failures
...
In file included from BT/exact_rhs.c:34:
BT/header.h:53:10: fatal error: 'type.h' file not found
#include "type.h"
         ^~~~~~~~
...
```

8. Add *common* directory to the include path. Compiler flags are passed to all Parallelware Analyzer tools after all other arguments and separated by "--", following the GCC/Clang syntax:

```
$ pwreport BT/*.c -- -I common
...
SUMMARY
  Total defects:                0
  Total recommendations:        155
  Total opportunities:          68
...
```

> Notice the number of recommendations and opportunities reported. The goal is to reduce them by applying best practices recommendations to your code and by implementing parallel versions of the opportunities.

9. Run **pwloops** to get information about opportunities for parallelization:

```
$ pwloops BT/*.c -- -I common
...
Loop                                  Patterns Opportunities
------------------------------------   -------- -------------
BT/exact_rhs.c:exact_rhs:47:3         forall   multi
BT/exact_rhs.c:exact_rhs:48:5         n/a
...
```

10. You can also use **pwloops** to visualize the source code annotated with opportunities. You can do so filtering by function to narrow the output to the relevant functions:

```
$ pwloops BT/*.c --code --function BT/exact_rhs.c:exact_rhs -- -I
common
...
Line Opp  BT/exact_rhs.c
---- ---  ------------------------------------------------
  39      void exact_rhs()
  40      {
  41          double dtemp[5], xi, eta, zeta, dtpp;
  42          int m, i, j, k, ip1, im1, jp1, jm1, km1, kp1;
  43
  44          //---------------------------------------------
  45          // initialize
  46          //---------------------------------------------
  47  P       for (k = 0; k <= grid_points[2]-1; k++) {
  48            for (j = 0; j <= grid_points[1]-1; j++) {
  49              for (i = 0; i <= grid_points[0]-1; i++) {
  50                 for (m = 0; m < 5; m++) {
  51                   forcing[k][j][i][m] = 0.0;
  52                 }
  53              }
  54            }
  55          }
...
```

11. Once you have selected a loop, parallelize it using the *Patterns* information from **pwloops**. For instance, for a *forall*:

- for OpenMP multithreaded execution use `#pragma omp parallel for`
- for OpenMP vector/SIMD execution use `#pragma omp simd`

12. Recompile and run the selected benchmark.

- Verify the correctness of the parallel version.
- Verify the new runtime of the parallel version and if it is a speedup or a slowdown.
- Fill-in a new row in the record sheet for this iteration.

13. Run **pwcheck** to look for defects that may have been introduced during the parallelization of the code (such as race conditions):

```
$ pwcheck BT/*.c --only-defects -- -I common
...
PWD001    PWD002    PWD003
     0         0         0
...
```

No defects should be reported. If there is any, it must be fixed right away.

14. Repeat steps **8** to **14.**

Record sheet

	pwreport invocation		Benchmark binary invocation (eg. bin/bt.S.x)		
Step	# opportunities	# recommendations	Runtime	Speedup	Correctness(y/n)
1				n/a	y
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					
16					
17					
18					
19					
20					
21					
22					
23					
24					
25					
26					
27					
28					
29					
30					
31					
32					
33					
34					