Parallelware Analyzer NPB Quickstart

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 consists of 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 and performs **data-race analysis.**
- **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 (direct download here), 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-0.16.0-686cf9c_linux-x64.tgz
$ mv pwanalyzer-eap.lic pwanalyzer-0.16.0-686cf9c_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.W.x
...
 Time in seconds =               2.20
...
 Verification    =         SUCCESSFUL
...
```

> The *W* in the binary name represents the NPB workload class, in this case a workstation size. Look for the outputted correctness and runtime information.

6. Rebuild with profile information. For **gcc**, this is achieved through the *-pg* flag. Given how NPB build is organized, you need to edit *config/make.def* to add *-pg* to both the *CFLAGS* and *CLINKFLAGS* (lines 113 and 119, respectively). Once you have done so, rebuild the benchmark:

```
$ vim config/make.def
$ make clean bt
```

7. Running the benchmark again, will produce a *gmon.out* file containing the profiling information. You can then use **gprof** to find hotspots:

```
$ bin/bt.W.x
$ gprof bin/bt.W.x
...
```

```
 33.80       0.73   0.73  6712596       0.00   0.00  binvcrhs
 16.20       1.08   0.35      201       1.74   3.68  y_solve
 13.89       1.38   0.30  6712596       0.00   0.00  matmul_sub
 10.65       1.61   0.23      201       1.14   3.08  x_solve
 10.19       1.83   0.22      202       1.09   1.09  compute_rhs
...
```

> Now you have information about which functions consume most of the execution time.

> Although in this example we have used **gcc** and **gprof**, feel free to experiment with other profilers: there is no limitation by Parallelware Analyzer.

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

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

> Some files fail due to missing includes which are located into NPB's *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.

9. 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"
         ^~~~~~~~
...
```

10. Add NPB's *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:        176
  Total opportunities:          15
  Total data races:             0
  Total data-race-free:         0
```

> 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.

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

```
$ pwloops BT/*.c -- -I common
...
Loop                          Analyzable Compute patterns Opportunity Auto-Parallelizable Parallelized
----------------------------- ---------- ---------------- ----------- ------------------- ------------
BT/exact_rhs.c:exact_rhs:47:3   x          forall           multi         x
BT/exact_rhs.c:exact_rhs:48:5   x          n/a
...
```

12. 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          }
...
```

> Filtering by function comes very handy to focus on the hotspots detected through profiling. Remember that **gprof** reports which functions consume most of the runtime.

13. Once you have selected a loop, parallelize it using the *Patterns* information from **pwloops**. For instance, you can typically parallelize a *forall* pattern using OpenMP multithreaded execution by annotating the loop with the directive: `#pragma omp parallel for`

In order to enforce parallel programming best practices, you can also take advantage of **pwdirectives** to insert the multithreading directives in the loop to be parallelized using the *<file>:<function>:<line>:<column>* syntax:

```
$ pwdirectives --in-place BT/exact_rhs.c:47:3 -- -I common
...
Parallel forall: variable 'forcing'
Loop parallelized with multithreading using OpenMP directive 'for'
Complete access range for variables: 'grid_points', 'forcing'
Parallel region defined by OpenMP directive 'parallel'
Successfully updated BT/exact_rhs.c

$ sed -n 47,51p BT/exact_rhs.c
  #pragma omp parallel default(none) shared(forcing, grid_points)
private(i, j, k, m)
```

```
        {
        #pragma omp for private(i, j, m) schedule(auto)
        for (k = 0; k <= grid_points[2]-1; k++) {
             for (j = 0; j <= grid_points[1]-1; j++) {
```

14. Recompile with OpenMP enabled and re-run the selected benchmark. To enable OpenMP, edit *config/make.def* just like you did in step 6, but this time adding the *-fopenmp* flag.

```
$ vim config/make.def
$ make clean bt
$ bin/bt.W.x
```

> Fill-in a new row in the record sheet for this iteration:
>
> - Write down the number of defects, opportunities, recommendations, data races and data race frees from **pwreport**.
> - Verify the correctness of the parallel version.
> - Verify the new runtime of the parallel version and if it is a speedup or a slowdown.

15. 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    PWD004    PWD005    PWD006    PWD007    PWD008
     0         0         0         0         0         0         0         0
...
```

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

16. Repeat steps **10** to **16.**

Using a configuration file

All previous invocations of Parallelware Analyzer command-line tools required to pass the same flags once and again (ie. you wrote "`-- -I common`" at the end of each invocation). You can use a configuration file to store the required compiler flags and re-use it for all your analyses:

17. Create a *config.json* file and add the following contents:

```
$ vim config.json
{
    "version": 2,
    "analyses": [
        {
            "match": "*.c",
            "command": "gcc %f -I common"
        }
    ]
}
```

> This configuration tells Parallelware Analyzer that for files matching the `*.c` pattern (ie. C source files) it should use the `gcc %f -I common` command. This command starts with `gcc` to specify that what follows are flags using the GCC/Clang syntax; then, it comes `%f` which represents the matched source file. Lastly, the command specifies the already known `-I common` flag that we wish to apply to all C source files. Parallelware Analyzer will parse the `gcc` command to extract the compiler flags.

18. Try executing any of the previous Parallelware Analyzer invocations using the configuration file instead of the compiler flags:

```
$ pwreport BT/*.c --config config.json
```

Parallelware Analyzer supports using a configuration file to address this and more complex scenarios such as integration with CMake or declaring dependencies between files.

- Integration with **CMake** can be accomplished by specifying a `compile_commands.json` compilation database file in your configuration file. Parallelware Analyzer will look for compilation commands for files to be analyzed in order to gather all the relevant flags.
- Declaring **file dependencies** instructs Parallelware Analyzer to analyze related source files together. This may be essential to discover new opportunities for parallelization in cases where loops invoke functions whose definition is located in a different source file: in most cases the function body must also be analyzed to determine whether the loop can be parallelized or not.

Refer to the `docs/ConfigurationFile.md` file for exhaustive documentation and to the `examples/config/` folder to find configuration file examples.

Record sheet

	pwreport invocation					Benchmark binary invocation (eg. bin/bt.W.x)		
Step	Defects	Recommendations	Opportunities	Data races	Data race free	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								