Worksheet:
An introduction to using Appentra Parallelware Trainer

1 Introduction

This practical will demonstrate how to use Parallelware Trainer to parallelize scientific software.

2 Setting up Parallelware Trainer

2.1 Getting Started

You can download and execute Parallelware Trainer on your own laptop for Windows, Mac or Linux.

2.2 Installation instructions

1. To download a copy of the software, go to https://appentra.com and click on "Try Parallelware Trainer".
2. You will be requested to register on the website. Once registered, you will receive an email to activate your account.
3. Activating your account will redirect you to a page where you can download the package for your platform (standalone compressed package or installer) and your trial license.
4. Once downloaded and installed (or extracted), open the executable on the installation folder. A message with "No license was found" should appear. Click on the "Click here to update your license" message and choose the previously downloaded license.

3 Exercises

3.1 Parallel Trainer Walkthrough: Parallelizing the computation of π

This exercise is designed to get you used to using Parallelware Trainer.

The number π is a mathematical constant that was originally defined as the ratio of a circle's circumference to its diameter. With multiple equivalent definitions and uses it is commonly used in all areas of mathematics and physics. The calculation of π is common in many numerical libraries, with more specialized versions that have been developed to calculate π and other common mathematical constants to any desired precision.

The code that you will work on in this exercise is a simple algorithm for calculating the π constant and will allow you to understand how to use Parallelware Trainer and the ability to compare different parallel implementations quickly and effectively.

1. Launch Parallelware Trainer
2. Download and unzip the PI folder from the example package:
   `https://www.appentra.com/download/8497/`
3. Open the PI code example. Go to `File > Open Project` and select the 'PI' folder in the examples directory. This should open the project in the Parallelware Trainer project panel.

4. Double-click '`pi.c`' inside the 'src' folder to open it. Notice a green circle next to line 7. This identifies a loop parallelization opportunity.

5. Click the green circle. Use the default selected options (OpenMP, CPU and loop). Click 'Parallelize' and Parallelware will insert correct parallelization pragmas for you based on these requirements. Parallelware Trainer will also create a new version containing the original code in the right-hand panel, the 'Version Manager'. To show the Version Manager, click the '`<`' arrow located to the right of the source-code window.

6. Create a new version of the parallelized code by clicking the button located to the right of the tab bar in the code editor. A dialog will ask you to name the version: enter '`pi_reduction`'. For more information on storing and creating versions of your project code read section 3.7 of the Parallelware Trainer user manual.

7. Restore the original version by clicking the restore button (located to the left of the version manager tab bar). A dialog will pop up, asking you to confirm that you want to overwrite the code in the editor with that of the version. Confirm by clicking 'OK'.

8. Now generate a different type of parallelization by again clicking the green circle. Retain the same options as before (i.e. default), but in the 'Atomic protection' input box enter the name of the variable that is being protected: i.e. 'sum'. Close the dialogue by clicking 'Parallelize' to update your code with an atomic clause. Delete the auto-generated Original 2 version as it contains the same code as Original 1 by hitting the X button next to its name.

9. As in step 6 create a new version named '`pi_atomic`'.

10. Generate another version. In instances where the built-in reduction does not support the reduction operator, privatization can be used instead. To create a version using explicit privatization restore the original version (see step 7), click the green circle. Retain the same options as before (i.e. default), but in the 'Explicit Privatization' input box enter the name of the variable that is being reduced: i.e. 'sum'. Close the dialogue by clicking 'Parallelize' to update your code to perform an explicit reduction using privatization. Delete the auto-generated Original 2 version.

11. As in step 6, create a new version named '`pi_privatization`'.

12. Now you should have 4 versions of the '`pi.c`' file: '`original`', '`pi_reduction`', '`pi_atomic`' and '`pi_privatization`'. Note that if you change to another file in the code editor, these versions won't appear in the version manager as they are associated to the '`pi.c`' file.

13. To run the code you need to configure the build and run commands: `Project > Configuration`.

    (a) In the build dialog enter the build command make.

        - Make sure your default GCC compiler can compile OpenMP code. If not, please edit the Makefile to use an appropriate compiler. Add `CC := /path-to-preferred-compiler` to line 4 of the Makefile, and make sure the file is saved.
        - Make sure that the build command executes in the top-level PI folder that contains the Makefile.

    (b) In the run dialog enter `./pi 100000000`. Make sure that run command executes in the top-level PI folder that will contain the executable.

14. Now you are ready to experiment and measure runtimes to compare the performance of the different versions.

    - To measure the execution time of a particular version restore it from the version manager (right hand window) to the code editor (left hand window). Whenever you do this, be sure to have a copy of the code currently displaying in the Code Editor in the Version Manager before overwriting it.
    - Click 'Run' - the play icon located at the bottom - to build and run the code.
    - The Execution output console will show the runtime.
    - Compare the runtime for the different versions created.

- Switch between the different versions by restoring the one you want to measure. Then simply click 'Run', which is the button with the play icon located in the bottom. Try to do so with all the versions and compare the times shown in the 'Execution output'.

15. To change the number of threads open `Project > Configuration`, and edit the run command to: `env OMP_NUM_THREADS=8 ./pi 100000000`. This will run the job on 8 threads.

Comparing parallel performance

Compare the performance of the different parallelization strategies you have implemented, across different numbers of threads. Use the following table to record your results to help identify the best combination of parallelization strategies. Use a newline in the table for each version.

Code version / notes	OpenMP	**Runtime (s)**				**Speedup**
	threads	Time 1 (s)	Time 2 (s)	Time 3 (s)	Mean (s)	T(seq)/T(mean)
Sequential	1					1

3.2 Parallelizing a simplified version of LULESH: CORAL_lulesh_2.0.3_mk

The aim of this exercise is to use Parallelware Trainer to introduce OpenMP directives into a real problem. The sample code is based on the Coral benchmark LULESH.

3.2.1 The Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH)

A wide variety of science and engineering problems require modeling hydrodynamics, describing the motion of materials relative to each other when subject to forces. LULESH is a highly simplified 'proxy' application, that represents the real challenges involved in the numerical simulation of hydrodynamic problems. Developed by Lawrence Livermore National Laboratory (LLNL), it has been used extensively in understanding how to reach exascale in applications.

The version, Lulesh_2.0.3_mk, used in this example, has been developed for this course to provide a representative piece of software, that can be parallelized in the time limitations of this course.

3.2.2 Parallelizing CORAL_lulesh_2.0.3_mk

You should follow the steps that you followed in the walkthrough of the Pi project (above). This example is significantly more complex and will require you to investigate more opportunities for parallelization.

1. Launch Parallelware Trainer
2. Download and unzip the CORAL_lulesh_mk folder from the example package: `https://www.appentra.com/download/8497/`.
3. Open the project: Go to `File > Open Project` select the (CORAL_lulesh_2.0.3_mk) folder in the examples directory. This should open the project in the Parallelware Trainer project panel.

4. Double-click 'CORAL_lulesh_2.0.3_mk.c' inside the 'src' folder to open it. You should notice the successful analysis by Parallelware in the Parallelware console. If there is a problem, you may need to ensure that you are using an OpenMP enabled compiler in the Makefile.

5. Build and run the sequential code:

   (a) Enter the build command: 'make'

   (b) Enter the run command: 'env OMP_NUM_THREADS=1 bin/lulesh'.

   (c) Run the sequential code, recording the performance of the code in the table at the end of this document.

6. Now, take a look at all the opportunities that Parallelware Trainer has found. Use the knowledge you have just acquired to try and parallelize some of those loops.

7. You should notice a loop that seems a little more complex than the rest, this is the application performance hotspot. Click the green circle and use the default selected options (OpenMP, CPU and loop) to insert OpenMP parallelization.

8. You should notice that the last line of the loop was protected with an atomic directive. This will avoid race-conditions in the concurrent updates of T. Benchmark this version with 1, 2 and 4 threads and record the results in the table.

9. You can now save this version (e.g. as "OpenMP atomic").

10. Restore the original sequential version.

11. Now compile using an explicit privatization of the variable, T. This is an alternative implementation of this sparse parallel reduction, that avoids the race-conditions in the loop. Explicit privatization requires the creation of private copies of the array for each thread, T_private. Each thread is then able to operate on T_private, before merging the data into the result variable, T. Luckily, Parallelware Trainer can help us do that. Click again in the green circle on the same loop, write 'T' on the 'Explicit privatization' field and press 'Parallelize'.

12. As you can see, some parts of the code are missing, which will result in the build process failing. This is due to missing information about the size of the T variable. Let's undo this version (restore the original code) and try the parallelization again with more information. Open the parallelization dialog again, and write 'T' in the 'Explicit privatization' field. Now you must write the size of the T variable under the "Array ranges" field as of T[<start>:<length>] (Hint: you should be able to infer the array size from the loop below). This new version should now build correctly.

13. Compile, execute and benchmark this versions.

14. Compare the performance of the two versions over different thread counts.

4 Notes...

Results

Use the following table to record your results to help identify the best combination of parallelization strategies. Use a newline in the table for each version, but run each version at least 3 times to calculate a mean runtime.

Experimental results for CORAL_lulesh_2.0.3_mk

Code version / notes	OpenMP	**Runtime (s)**				**Speedup**
	threads	Time 1 (s)	Time 2 (s)	Time 3 (s)	Mean (s)	T(seq)/T(mean)
Sequential	1					1