appentra
make code parallel

Worksheet:
An introduction to using Appentra Parallelware Trainer

1 Introduction

This practical will demonstrate how to use Parallelware Trainer to parallelize scientific software. Two exercises are proposed to get familiar with the different code patterns and parallelization strategies available to speedup the execution of mathematical algorithms by exploiting multi-core CPUs and GPUs through the use of OpenMP and OpenACC.

In each exercise, you will create several versions of the code for different combinations of patterns, strategies and target hardware. You will benchmark and write down the execution time of those versions and then compare the results.

1.1 Launching Parallelware Trainer from a GPU node on Cori

1. Log in to Cori enabling X11 forwarding: `ssh -X <your_login>@cori.nersc.gov`
2. Copy the sample codes to be used during the course:
   `cp -a /global/common/cori_cle7/software/pwtrainer/Workshop-Oct19-examples.zip ~`
3. Uncompress the sample codes in hour home directory:
   `unzip Workshop-Oct19-examples.zip`
4. Prepare the environment by loading the appropriate modules: `module purge && module load esslurm cuda gcc/8.1.1-openacc-gcc-8-branch-20190215 pgi/19.9 pwtrainer`
5. Open an interactive session in a GPU node: `salloc -t 60 -N 1 -c 8 -C gpu --gres=gpu:1 --mem=32GB --reservation=trainer -A nintern`
6. Run the Parallelware Trainer tool from the GPU node: `pwtrainer &`

2 Parallelizing the computation of π

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. The algorithm presents a **scalar reduction** pattern for which 3 different versions implementing the **built-in reduction**, **atomic** and **explicit privatization** parallelization strategies will be implemented using OpenMP. Another version implementing the built-in reduction using OpenACC will also be created.

1. Launch Parallelware Trainer (see section 1.1 for details).
2. 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.

3. Double-click '`pi.c`' to open it. Notice a green circle next to line 27. This identifies a loop parallelization opportunity. The Parallelware output console gives a brief of how many opportunities were found in the source file, in this case it shows: 'Analysis completed: 1 opportunity found'.

4. Click the green circle. Use the default selected options (OpenMP, CPU and Multithreading). Click 'Parallelize' and Parallelware will insert correct parallelization pragmas for you based on these requirements. Notice the text printed to the Parallelware output console. It shows that an **scalar reduction** pattern was identified for variable 'sum' and that the **built-in reduction** strategy was implemented. You may click some of these terms in the console to obtain more information. Parallelware Trainer will also save 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.

5. Save 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_omp_reduction'**. For more information on saving and restoring versions of your project code read section 3.6 of the Parallelware Trainer user manual.

6. 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'.

7. Now you are going to generate a different type of parallelization using **atomic protection**. Click the green circle and 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.

8. As in step 5 save a new version named **'pi_omp_atomic'**.

9. 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 6), then 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.

10. As in step 5, save a new version named **'pi_omp_privatization'**.

11. Finally, create one last version using OpenACC to **offload** the work to the GPU. Restore the original version (see step 6), then click the green circle. Select OpenACC, GPU and Offloading. Click 'Parallelize' and Parallelware will now insert OpenACC parallelization pragmas instead of OpenMP ones. As in step 5 save a new version named **'pi_acc_reduction'**.

12. Now you should have 5 versions of the '`pi.c`' file, namely: '`Original 1`', **'pi_omp_reduction'**, **'pi_omp_atomic'**, **'pi_omp_privatization'** and **'pi_acc_reduction'**. 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 in `Project > Configuration`:

    (a) Switch between GCC and PGI compilers depending on whether your code uses OpenMP or OpenACC. We recommend using the PGI compiler instead of GCC for OpenACC due to its increased support.

    - Build the OpenMP versions using GCC by entering the following command in the build dialog: `gcc -fopenmp -lm -o pi pi.c`
    - Build the OpenACC version using PGI by changing the build command to: `pgcc -acc -ta=tesla:cc70,cuda10.0 -Minfo=accel -lm -D_OPENMP -o pi pi.c`

    (b) In the run dialog enter: `srun env OMP_NUM_THREADS=1 ./pi 900000000`

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.
    - When switching between OpenMP and OpenACC versions, update the build command to use the appropriate the compiler.
    - 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.

15. To change the number of threads, open `Project > Configuration` and edit the run command to: `srun env OMP_NUM_THREADS=8 ./pi 900000000`. This will run the job on 8 threads. This setting only affects OpenMP code. Now go ahead and experiment with different values.

16. Take away the work you have done in this practical, either by exporting the created versions through `File > Export File Versions` or by copying the whole project data:

    - Close the project PI by clicking `File > Close project`.
    - Go to the project directory and display the files in the hidden directory `.pwt/default/versions/pi.c/`. All the versions of the code `pi.c` will be there.
    - Compress the directory of PI in a ZIP file and take it away with you.

Experimental results for Pi

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 new row 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 Parallelizing a LULESH microkernel

A wide variety of science and engineering problems require modeling hydrodynamics, describing the motion of materials relative to each other when subject to forces. The Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (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, LULESHmk, used in this example, has been developed to provide a representative piece of software that can be parallelized in the time limitations of this course. It is based on the Coral benchmark LULESH.

You should follow the steps that you followed in the walkthrough of the Pi project (above). This example is significantly more complex, presenting a **sparse reduction** parallel pattern and will require you to investigate more opportunities for parallelization. Two OpenMP versions will be created, **'luleshmk_omp_atomic'** and **'luleshmk_omp_privatization'**, applying atomic and explicit privatization strategies respectively. Then, an OpenACC version, **'luleshmk_acc_atomic'**, will be created, also implementing the atomic strategy but targeting the GPU.

1. Launch Parallelware Trainer (see section 1.1 for details).
2. Open the project: go to `File > Open Project` and select the 'LULESHmk' folder in the examples directory. This should open the project in the Parallelware Trainer project panel.
3. Double-click '`luleshmk.c`' to open it. You should notice the successful analysis message in the Parallelware output console: 'Analysis completed: 12 opportunities found'.
4. Build and run the sequential code:
   (a) Enter the build command: `gcc -fopenmp -lm -o luleshmk luleshmk.c`
   (b) Enter the run command: `srun env OMP_NUM_THREADS=1 ./luleshmk`
   (c) Run the sequential code, recording the performance of the code in the table at the end of this document.
5. 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.
6. You should notice a loop that seems a little more complex than the rest, this is the application performance hotspot. Click the green circle next to line 132 and use the default selected options (OpenMP, CPU and Multithreading) to insert OpenMP parallelization.
7. Take a look at the Parallelware output console. It shows that a **sparse reduction** pattern has been identified for the `domain_m_fx`, `domain_m_fy` and `domain_m_fz` variables and that an **atomic privatization** strategy has been implemented for each one of them. The sum operations were protected with an atomic directive which will avoid race-conditions in the concurrent updates of `domain_m_fx`, `domain_m_fy` and `domain_m_fz`. Benchmark this version with 1, 2 and 4 threads and record the results in the table.
8. You can now save this version as **'luleshmk_omp_atomic'**.
9. Restore the original sequential version.
10. Now create a new version using an **explicit privatization** of the variables `domain_m_fx`, `domain_m_fy` and `domain_m_fz`. 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, for instance `domain_m_fx_private`. Each thread is then able to operate on `domain_m_fx_private`, before merging the data into the result variable, `domain_m_fx`. Luckily, Parallelware Trainer can help us do that. Click again in the green circle on the same loop, write 'domain_m_fx, domain_m_fy, domain_m_fz' on the 'Explicit privatization' field and press 'Parallelize'.

11. As you can see, some parts of the code are missing, which will result in both the analysis and build processes failing. This is due to missing information about the size of the `domain_m_fx`, `domain_m_fy` and `domain_m_fz` variables. Undo this version (restore the original code) and try the parallelization again with more information. Just like before, open the parallelization dialog and write 'domain_m_fx, domain_m_fy, domain_m_fz' in the 'Explicit privatization' field. Now you must also write the size of each variable under the 'Array ranges' field as a comma-separated list following the syntax `<variable>[<start>:<length>]` (e.g. `domain_m_fx[0:NUM_NODES]`). This new version should now build correctly. Save it as **'luleshmk_omp_privatization'**.

12. Now, create one last version using OpenACC to **offload** the work to the GPU. Restore the original version, click the green circle and then select OpenACC, GPU and Offloading. Click 'Parallelize' and Parallelware will now insert OpenACC parallelization pragmas instead of OpenMP ones.

13. Notice that there are array ranges missing in the 'acc data' directive. Replace the 'copy' and 'copyin' clauses with the following ones: `copy(domain_m_fx[0:NUM_NODES], domain_m_fy[0:NUM_NODES], domain_m_fz[0:NUM_NODES]) copyin(domain_m_nodelist[0:MAX_NODELIST], gamma[0:4][0:8], numElem)`

14. Since the loop invokes the 'CalcElemFBHourglassForce_workload' and 'CalcElemFBHourglassForce' functions, we must instruct OpenACC to create versions of these functions in the GPU. To do this, insert `#pragma acc routine seq` right above each function declaration at line numbers 58 and 68. Save a new version named **'luleshmk_acc_atomic'**.

15. Build, run and benchmark these versions. Use the following build command depending on whether you are benchmarking an OpenMP or OpenACC version:

    - As instructed in step 4, build OpenMP versions using GCC through the following command: `gcc -fopenmp -lm -o luleshmk luleshmk.c`
    - For OpenACC, we recommend using the PGI compiler over GCC due to its increased support. Therefore, to build the `'luleshmk_acc_atomic'` version, change the build command to: `pgcc -acc -ta=tesla:cc70,cuda10.0 -Minfo=accel -lm -D_OPENMP -o luleshmk luleshmk.c`

16. Compare the performance of the two OpenMP versions over different thread counts and the OpenACC (which is not affected by the thread count setting).

17. Analyze the data structure to see how you could minimize data transfers when offloading to the GPU. For instance, can data be transferred just once for all the iterations of a loop? What about for several loops or functions?

18. Take away the work you have done in this practical, either by exporting the created versions through `File > Export File Versions` or by copying the whole project data:

    - Close the project LULESHmk by clicking `File > Close project`.
    - Go to the project directory and display the files in the hidden directory `.pwt/default/versions/luleshmk.c/`.
      All the versions of the code `luleshmk.c` will be there.
    - Compress the directory of LULESHmk in a ZIP file and take it away with you.

Experimental results for LULESHmk

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

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

4 Notes...