In this assignment we will work on the Linux-Cluster operated by LRZ. To take advantage of the compute power we parallelize our code using OpenMP. Before you start working on the cluster, consider the following rules:

- The cluster is divided into login and compute nodes. Do not use the login nodes to run your application! You are only allowed to run short serial programs on the login nodes. To use the compute nodes, you have to submit batch jobs or use interactive jobs.
- You can submit multiple batch jobs but it is not allowed to start more than one interactive job!
- The space available in your home directory is limited and **shared** with all other users. Do not put large input and output files in your home directory! The environment variable `$SCRATCH` contains the path to the scratch folder. You can use this folder for large input and output files. Access to files in this folder is also faster than using your home directory.

Literature

- *Linux-Cluster*: `https://www.lrz.de/services/compute/linux-cluster/`
- *Intel Compiler User and Reference Guides*: `http://software.intel.com/sites/products/documentation/doclib/iss/2013/compiler/cpp-lin/index.htm`
- *Intel VTune Amplifier Help*: `http://software.intel.com/sites/products/documentation/doclib/iss/2013/amplifier/lin/ug_docs/index.htm`
- *OpenMP Specifications*: `http://www.openmp.org/mp-documents/OpenMP3.1.pdf`
- *OpenMP Tutorial*: `https://computing.llnl.gov/tutorials/openMP/`

1 Linux-Cluster

The Linux-Cluster has several segments[1]. Each segment has different hardware and/or different configurations. We recommend that you use the MPP segment for the exercises in this assignment.

With your cluster account and password you can login to one of the login nodes listed on the LRZ homepage[2] using `ssh`. When compiling your code on the login nodes, make sure they have the same configuration as the compute nodes you are going to use, i.e. do not compile your code for Intel CPUs and run it on AMD.

Since you are not allowed to run simulations on the login node, you have to allocate a compute node, to run SWE. Use the command `salloc` to get an interactive shell on a

[1] `https://www.lrz.de/services/compute/linux-cluster/overview/`
[2] `https://www.lrz.de/services/compute/linux-cluster/intro/`

compute node. This command gives exclusive access to the compute node. No applications from other users will be running on this node once you have successfully allocated the node. However, there are only an limited number of (interactive) compute nodes available. Hence, you may have to wait some time for a free node.

As soon as you have allocated an interactive node, you can start your application with the command `srun_ps ./<application> <parameters>`. It is important that you use `srun_ps` for interactive jobs. Otherwise the application will still run on the login node. You can test this using the commands `hostname` and `srun_ps hostname` in an interactive shell[3]. When your application is finished, use the command `exit` to leave the interactive shell.

Interactive shells are a fast way to test your application. However, they are not meant for productive simulations and have a short time limit (15 min by default). If you want to run your simulation for a longer time period, you have to submit a batch job. A batch job consists of a script that describes the configuration of your job (e.g. number of task, required time) and the commands that should be used to start the application. LRZ provides example batch scripts that you can use as a starting point.[4] Since we do not use MPI, set the `ntasks` parameter to 1 and omit `srun_ps`. The maximum run time should be set large enough that your application can finish within this time but as small as possible. Jobs with a shorter maximum run time may be scheduled faster. You can submit the job with the command `sbatch <script>`. Do not forget to specify the stdout/stderr file (good practice is to put these files in the scratch folder as well).

Tasks

1. Upload and compile your code on the cluster. You can use the command line tools `scp` or `sftp` to copy files or directories to the cluster. Instead of coping the code using `scp`/`sftp` you can also clone your git repository on the cluster.

2. Run different scenarios on the interactive and the batch nodes. Compare the results from the cluster with those from the last assignments . Are you getting the same results?

3. Is the cluster faster than your own computer? You can use the functions `resetCpuClockToCurrentTime` and `updateCpuTime` from the `Logger` to measure the time. To get a good value for the *time per cell per iteration*, do not include file I/O and setup time in your measurement.

2 Intel Compiler

Not all compilers generate the same code. In this chapter we will compare the GCC with the Intel Compiler (icpc). The Intel Compiler is already installed on the Linux Cluster and the `SConstruct` file has a switch to enable the compiler. If the Intel Compiler does not find the license file, copy the file `/lrz/sys/intel/licenses/license.lic` to `~/intel/licenses/license.lic`.

[3] `http://unixhelp.ed.ac.uk/CGI/man-cgi?hostname`
[4] `http://www.lrz.de/services/compute/linux-cluster/batch_parallel/`

Tasks

1. Recompile your code using the Intel Compiler. Which compiler generates faster code? Remember not to run the code on the login nodes!

2. Compile the code with both compilers with different optimization switches (e.g. `-O2`, `-fast`) and compare the runtime of the resulting code.

3. The Intel Compiler has two additional options `-vec-report3` and `-opt-report`.[5] Both options generate additional information about vectorization and optimizations (e.g. inline functions) the compiler is able to do (or not to do). Add those two options in the `SConstruct` file. Is the compiler able to vectorize your main loops? Is your solver inlined?

4. (Optional) Compile and test the code with the Intel Compiler on your own computer. You can get a student license from Intel:
`http://software.intel.com/en-us/intel-education-offerings`

3 VTune

The compiler reports are a fast and easy way to get some first optimization hints for your code. To get a better inside on the bottlenecks we will use the Intel VTune Amplifier for a runtime analysis in this chapter. VTune is available on the Linux Cluster. You can enable it using the command `module load amplifier_xe`. VTune consists of a GUI (`amplxe-gui`) and a command line interface (`amplxe-cl`).

1. Login to the cluster with enabled X11-forwarding (see hints) and start the VTune GUI. Create a new project for your application. Add an analysis to the project but do not start the analysis from the GUI because your application would run on the login node. You can not use all analysis types since you do not have access to the kernel driver but `Hotspots` and `Concurrency` are a good starting point.

2. Start the analysis using the command line tool in a batch job. You can copy & paste the whole command from the GUI.

3. Once you batch job has finished, use the GUI to visualize the results. Try adding debug symbols to the code or disable function inlining (`-fno-inline`) to get finer results.

4. Which parts are compute intensive? Did you expect this?

5. Think about how you could improve the performance of your code, especially the compute intensive parts. Consider features of C++ (e.g. inlining, templates) as well as algorithmic features (e.g. reuse calculated values, avoid unnecessary square-roots or divisions, ...)

[5] `http://software.intel.com/sites/default/files/m/e/f/4/5/2/6747-vtune_compiler_advice.pdf`

4 OpenMP

In this chapter we will add threads to SWE to parallelize the important loops. However, instead of using a low level threading API, we will use OpenMP. OpenMP allows us to use simple pragmas to parallelize loops, e.g. to parallelize a for-loop only `#pragma omp parallel for` in front of the loop is required. A compiler with OpenMP support will generate code that executes the loop with multiple threads. You can enable OpenMP support in GCC (`-fopenmp`) and the Intel Compiler (`-openmp`).

With OpenMP the order in which loop iterations are executed is no longer guaranteed. To get the correct results, you have to make sure that you do not have any dependencies between loop iterations, i.e. two loop iterations should not read or write to the same variable. Section 2.9.3 (Data-Sharing Attribute Clauses) in the OpenMP documentation explains constructs that can be used to share data between threads. For example, add a `private` clause to the pragma to declare a variable thread local, i.e. each thread gets its own copy of the variable:

```
float a[N];
float b[N];
float result[N];

float sum;
#pragma omp for private(sum)
for (int i = 0; i < n; i++) {
  sum = a[i] + b[i];
  result[i] = 2 * sum;
}
```

For parallel applications we are interested in the speedup. The speedup is defined as follows:

$$S_p = \frac{T_1}{T_p} \tag{1}$$

S_p = speedup for p threads
T_p = run time with p threads

The ideal case is $S_p = p$, which means that the application runs on p cores p-times faster than on one core.[6] However, due to serial parts and communication/synchronization between threads the achieved speedup is usually smaller: $S_p < p$.

Tasks

1. Parallelize your application using OpenMP.
2. Compare the run time of your parallelized application with the serial version on the cluster. What speedup do you get for 2, 4, 8 and 16 threads. You can specify the number of OpenMP threads with the environment variable `OMP_NUM_THREADS`. Is it useful to start more than 16 threads?

[6] In very rare cases it is possible to get a speedup $S_p > p$ because of cache effects.

3. On a two-dimensional domain, is it better to parallelize the outer or inner loop?

4. Try different scheduling strategies (see section 2.5.1 in the OpenMP documentation). Which one performs best?

5 Coarse Output

With our parallel version of SWE we are now able to run larger simulations on the Linux-Cluster. However, we still load the results on a single core with limited main memory for visualization. Thus, we are not able to visualize the output of large simulations. To avoid this issue we will modify the output writer such that is averages several neighboring cells of the computational grid into one cell in the output file. A domain of size $n \times m$ cells should result in an output file with $\lceil n/x \rceil \times \lceil m/x \rceil$ cells per time step. The user should be able to specify x via a command line argument.

Tasks

1. Modify the output writers as described above.

2. Repeat the simulations of the 2011 Tohoku Tsunami with at least 5600×3200 cells and the 2010 Chile Tsunami with at least 3108×2331 cells. Visualize your coarse output.

6 Project Plan

In the final part of the lab course your are going to work on a project of your own. The project can for example include further parallelization and optimization of SWE or verification, but is not limited to. As part of the projects, each group prepares a project proposal for their final project which includes goals and important milestones. At the next meeting (10th June, 2013) each group presents a detailed project plan containing all goals, milestones and work packages. Keep in mind that the final presentation for your project is scheduled for 1st July, 2013.

Tasks

1. Hand in your proposal of at least half a page not later than 08:00 AM, Monday, 3rd July, 2013.

2. Set up a “small” meeting with us during the regular time slot (4 PM - 6 PM) on Monday, 3rd June, 2013.

3. Create a detailed project plan and include it into your presentation during the next “big” meeting.

Deliverables

The following deliverables have to be handed in no later than 08:00 AM, Monday, 10th June, 2013. Small files (<1 MB in total) can be send as an attachment directly to *breuera AT in.tum.de* and *rettenbs AT in.tum.de*, larger files have to be uploaded at a place of your choice, i.e. `https://github.com/`, `http://home.in.tum.de/`, `https://www.dropbox.com`. In either case inform us about the final state of your solution via e-mail.

- Code in a git-repository. Doxygen documentation and unit tests are mandatory.
- Slides for the presentation during the next meeting.
- Pictures and animations of all runs.
- Documentation how to build and use your code, especially if you extend the SCons-script.
- The batch scripts you used to submit the jobs.
- A stdout/stderr file from a successful batch job.
- The vector and optimization reports from the Intel Compiler.
- Speedup graphs of your measurements. The graphs should also show your improvements.
- The project plan for your final project.
- 08:00 AM, Monday, 3rd June, 2013: The project proposal for your project.

Hints

- You can print the content of an environment variable with the command `echo $VARIABLENAME`.
- Files in the scratch folder older than 30 days will be automatically deleted. If you want to keep a file longer than 30 days, use the command `touch <filename>` to update the time stamp of the file.
- There are also graphical sftp clients available.
- When you have enabled X11-forwarding (`ssh -X`) you can use `sview` to visualize the compute nodes and jobs that are currently running or waiting for resources.
- You may have to wait some days till your batch jobs start. Make sure they finish before the deadline.
- VTune does not work with `srun_ps`, thus you have to use batch jobs.
- Do not forget to load the VTune module within your batch job.