Introduction to High Performance Computing Systems
CS 1645

University of Pittsburgh
Department of Computer Science
Spring, 2015
Homework 4

Instructions

- The solutions to all the problems in this homework should be the result of the student's individual effort. Presenting the words or ideas of somebody else under your name hinders your skills in academic and professional work and violates the university's policy on academic integrity:
  `http://www.provost.pitt.edu/info/ai1.html`

- The submission process for this homework will use the SVN repository:
  `https://collab.sam.pitt.edu/svn/CS1645/users/<pittID>`
  Create a `hw4` directory into your repository.

- You have to check-in the following files into the `hw4` directory:
  - File `stencil.cpp` containing the parallel code for the stencil computation.
  - File `implicit.cpp` containing the parallel code for the implicit moving squares computation.
  - File `report.pdf` with a report summarizing your results (it needs to be a PDF file).

  by 11:59pm **Monday March 2, 2015**.

- Late submissions will not be accepted.

- The SVN repository will be the only submission mechanism accepted.

- Your code ***MUST*** run on the Stampede system with the software infrastructure provided (Makefile, main files, and other utility files). You should use the files provided in the following repository:
  `https://collab.sam.pitt.edu/svn/CS1645/assignments/hw4`

- Consider using interactive job submissions for developing and debugging your code:
  `idev -A TG-CIE140012`

1 Stencil

In mathematics, especially the areas of numerical analysis concentrating on the numerical solution of partial differential equations, a stencil is a geometric arrangement of a nodal group that relate to the point of interest by using a numerical approximation routine. Stencils are the basis for many algorithms to numerically solve partial differential equations (PDE)[1].

A 7-point stencil operates on each internal entry (excluding the borders, which remain constant during the computation) of a three-dimensional array. The stencil computes the value of $A_{t+1}[i][j][k]$ as:

$$\frac{A_t[i][j][k] + A_t[i-1][j][k] + A_t[i+1][j][k] + A_t[i][j-1][k] + A_t[i][j+1][k] + A_t[i][j][k-1] + A_t[i][j][k+1]}{7}$$

where A_t represents the grid in the current iteration, and A_{t+1} represents the grid in the next iteration. The stencil is applied iteratively until the values converge, i.e., the error is lower than a tolerance value.

You should write a parallel program using OpenMP 4.0 directives that implements a 7-point stencil on a three-dimensional grid. Your program must offload as much computation as possible onto the accelerator (Intel Xeon Phi, or MIC, on Stampede). You must explore the different alternatives in order to find the *best* way to parallelize the code. You will need to identify appropriate places for inserting directives after considering the dependencies in the code. Think carefully where to insert the directives to minimize the data transmission cost.

Make sure you follow these instructions:

a) You should implement the `stencil` function in file `stencil.cpp`. This function must return the number of iterations executed.

b) You should add OpenMP 4.0 directives into the code in order to find the *best* way to parallelize it.

c) Generate a sequential version of your code by compiling without flag `-openmp`.

d) Generate a parallel version of your code by compiling without the `offload` pragmas.

e) Your program must run with the following parameters:
`./stencil <X> <Y> <Z>`
or
`./stencil <X> <Y> <Z> <file>`
where X, Y, and Z are the dimensions of the grid. The optional parameter `<file>` specifies a file with input data for the grid.

f) Use the following command to analyze the performance of your code using different numbers of threads:
`MIC_OMP_NUM_THREADS=240 ./stencil <X> <Y> <Z>`

[1] Extracted from Wikipedia

2 Implicit Moving Squares

Moving least squares is a method of reconstructing continuous functions from a set of unorganized point samples via the calculation of a weighted least squares measure biased towards the region around the point at which the reconstructed value is requested [2].

The implicit moving squares method takes a set P of particles in a three dimensional space and computes the *effect* of those particles on a three dimensional grid with dimensions $X \times Y \times Z$. The following algorithm describes the implicit moving squares method.

Algorithm 1 Implicit Moving Squares

$\sigma \leftarrow 0.008$
$min_X \leftarrow -0.5$
$max_X \leftarrow 0.6$
$min_Y \leftarrow -1.4$
$max_Y \leftarrow 1.1$
$min_Z \leftarrow 2.4$
$max_Z \leftarrow 3.4$
$dx \leftarrow \frac{max_X - min_X}{X-1}$
$dy \leftarrow \frac{max_Y - min_Y}{Y-1}$
$dz \leftarrow \frac{max_Z - min_Z}{Z-1}$
for every particle p **do**
  for every cell $grid[i][j][k]$ **do**
    $x \leftarrow min_X + i \times dx$
    $y \leftarrow min_Y + j \times dy$
    $z \leftarrow min_Z + k \times dz$
    $diff_x \leftarrow x - p.x$
    $diff_y \leftarrow y - p.y$
    $diff_z \leftarrow z - p.z$
    $\phi \leftarrow e^{-\frac{diff_x^2 + diff_y^2 + diff_z^2}{2\sigma^2}}$
    $sum \leftarrow (diff_x \times p.nx + diff_y \times p.ny + diff_z \times p.nz) \times \phi$
    $grid[i][j][k] \leftarrow grid[i][j][k] + sum$
    $phi[i][j][k] \leftarrow grid[i][j][k] + \phi$
  end for
end for
for every cell $grid[i][j][k]$ **do**
  $grid[i][j][k] \leftarrow \frac{grid[i][j][k]}{phi[i][j][k]}$
end for

You should write a parallel program using OpenMP 4.0 directives that implements the implicit moving squares method. Your program must offload as much computation as possible onto the accelerator (Intel Xeon Phi, or MIC, on Stampede). You will need to identify appropriate places for inserting directives after considering the dependencies in the code. Think carefully where to insert directives to minimize the data transmission cost. Start by applying a scatter-to-gather transformation to the code. Note that each point p affects all grid cells (i, j, k). This transformation should allow you remove the data structure $phi[i][j][k]$.

Make sure you follow these instructions:

a) You should add OpenMP 4.0 directives into the code in order to find the *best* way to parallelize it.

[2] Extracted from Wikipedia

b) Write your code in the provided file `implicit.cpp`.

c) Generate a sequential version of your code by compiling without flag `-openmp`.

d) Generate a parallel version of your code by compiling without the `offload` pragmas.

e) Your program must run with the following parameters:
`./implicit <X> <Y> <Z> <N>`
or
`./implicit <X> <Y> <Z> <N> <file>`
where X, Y, and Z are the dimensions of the grid, and N stands for the number of particles. The optional parameter `file` specifies the positions of input particles.

f) Use the following command to analyze the performance of your code using different numbers of threads:
`MIC_OMP_NUM_THREADS=240 ./implicit <X> <Y> <Z> <N>`

Report

You should create a report with the following sections for each of the two programs above:

1. A general strategy of the parallelization effort. Why did you choose those directives to parallelize the program?

2. A speedup analysis. Using an *interesting* problem sizes for both problems and these number of threads $\{30, 60, 120, 240\}$, make a speedup plot. Compare your execution time to the sequential version. Also report the execution time of the OpenMP processor (no offload) version with 16 threads.

3. An efficiency analysis. Using an *interesting* problem sizes for both problems and these number of threads $\{30, 60, 120, 240\}$, make an efficiency plot. Compare your execution time to the sequential version. Also report the execution time of the OpenMP processor (no offload) version with 16 threads.

4. A description of the performance bottlenecks. What is preventing the program from getting linear speedup?