Optimization and Scalability Lab

Carlos Rosales
carlos@tacc.utexas.edu

February 7th, 2012
Introduction to Parallel Computing
Labs Description

• **Automatic Optimization**
  – Compile a code with different optimization levels and study the run times.
  – This example shows how automatic optimization can greatly improve code performance without effort.

• **Vectorization**
  – Try to find why a kernel is not being vectorized by the compiler and change the code to fix the problem.
  – This example illustrates the common problem of data dependencies.

• **Scalability**
  – Run a given code on multiple processors and study the run times.
  – This example shows that codes should not be expected to have perfect scalability, and that scalability depends on problem size.
Setup

• Login to Ranger:
  – `ssh username@ranger.tacc.utexas.edu`

• Make sure you are using the intel compiler
  – `module swap pgi intel`
  – `module list`

• Untar the lab files:
  – `cd`
  – `tar xvf ~train00/opt_lab.tar`

• Change directories and ls to see the files:
  – `cd opt_lab`
  – `ls`
Lab 1: Automatic Optimization

• Compile auto.c or auto.f90 with increasing levels of automatic optimization:
  – icc -O0 -vec-report3 auto.c -o auto_O0
  – icc -O1 -vec-report3 auto.c -o auto_O1
  – icc -O2 -vec-report3 auto.c -o auto_O2
  – icc -O3 -vec-report3 auto.c -o auto_O3
  – ifort -O0 -vec-report3 auto.f90 -o auto_O0
  – ifort -O1 -vec-report3 auto.f90 -o auto_O1
  – ifort -O2 -vec-report3 auto.f90 -o auto_O2
  – ifort -O3 -vec-report3 auto.f90 -o auto_O3

• Notice the optimization messages printed by the Intel compiler.
  – At the O2 level three loops are vectorized
  – At the O3 level three loops are fused into one, and then the larger loop is vectorized

• Compile using -opt-report 2 to get additional information on the optimizations performed

• Submit the job through the SGE queue system:
  – qsub ./auto_job.sge
Lab 1: Automatic Optimization

• You can look at the execution status using `qstat` or `showq`.

• Once the job completes an output file called `auto.$JOBID.out` is generated, where `$JOBID` is the job number.

• Fill the table below with the execution times for the different optimization levels (get the timings from the output file `auto.$JOBID.out`)

• You should see a progressive improvements in the timings as more aggressive optimizations are used.

<table>
<thead>
<tr>
<th>OPT Level</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setup Time</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel Time</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Lab 1: Vectorization

• Compile the vec.c or vec.f90 source code:
  – icc –vec-report3 ./vec.c –o vec
  – ifort –vec-report3 ./vec.f90 –o vec

• The compiler should notify you that it is only possible to vectorize one of the loops in the code due to data dependencies.

• Submit the job to the batch system:
  – qsub ./vec_job.sge

• Once the job completes an output file called vec_job.$JOBID.out is generated, where $JOBID is the job number. Take note of the time spent in the kernel.
Lab 2: Vectorization

• Open the source code with your favorite text editor and look at the loop named \texttt{KERNEL}.

• Identify the source of the data dependence and correct it, so that the compiler is able to vectorize both the setup and the kernel loops.

• Recompile and verify both loops are vectorized.

• Submit the job again and compare the timings with the original.

• The vectorized version should be significantly faster.
Lab 3: Parallel Scalability

- In this example you will evaluate the scalability of a smoothing kernel code.

- Compile the matmult.c or matmult.f90 source code:
  - mpicc -O3 -xW ./parallel.c
  - mpif90 -O3 -xW ./parallel.f90

- Submit the job through the SGE queue system:
  - qsub ./parallel_job.sge

- The initial submission uses 2 processing cores only (-pe 2way 16). Check execution and MPI times in the parallel_job.$JOBID.out file created.
Lab 3: Parallel Scalability

- Change the submission script to use 4 cores (-pe 4way 16), 8 (-pe 8way 16) and 16 (-pe 16way 16), and build a table with the execution times.
- Does the execution time decrease linearly with the number of cores? Why do you think that is?
- Open the parallel.c or parallel.f90 file and change the parameters $x_{\max}$ and $y_{\max}$ from the default 160 to 800. Recompile the code.
- Repeat runs for 2, 4, 8, and 16 cores using this size and fill in the table below.
- Is there a difference in the scaling when you compare the results for the two different size problems? Why do you think that is?

<table>
<thead>
<tr>
<th>SIZE</th>
<th>2 cores</th>
<th>4 cores</th>
<th>8 cores</th>
<th>16 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>160 x 160</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>800 x 800</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Lab 3: Parallel Scalability

Simple smoothing kernel with a regular 1D task partition

\[ A(i, j) = \frac{A(i-1, j)}{4} + \frac{A(i, j)}{2} + \frac{A(i+1, j)}{4} \]

Problem requires regular data exchange on task boundaries

Light colored nodes are ghost nodes, used for data exchange
Problem requires regular data exchange on task boundaries

2D ghost cells

UP (1)
LEFT (3)
DOWN (7)
RIGHT (5)