Study Guide

Week 3 January 25th 2016 - January 31st 2016

Author B. Wilkinson Modification date January 22, 2016

Study Materials on Moodle

- *PowerPoint Slides*
  - Programming with Shared Memory-II
  - Programming with Shared Memory-III
- *Video*
  - Lecture 4 video: 75-minute video of Lecture 4 in Fall 2014 continuing programming with shared memory and OpenMP synchronization routines.
  - Lecture 5-2 video: video of the second part of Lecture 5 in Fall 2014 on shared memory performance issues. (The higher quality video contains the whole Lecture 5. The first part on the stencil pattern and assignment 2 will be covered in Week 4)
  - Lecture 6 video: 75-minute video of Lecture 6 in Fall 2014 continuing on shared memory performance issues.
- *Sample Quiz Questions*
  - Shared memory performance issues

Tasks

- **Mini-Quiz:** Answer the short posted quiz before *11:55 pm Sunday January 31st, 2016.*
- **Complete Assignment 1** OpenMP Tutorial
  - Assignment 1 Instructions (Week 2)
  - Assignment 1 Due: *Sunday January 31st, 2016* (Week 3)

Moodle Saba meeting –*7 pm Friday January 29th, 2016*

Programming with Shared Memory-II continues the basic concepts on shared memory programming - issues with accessing shared data, introducing critical sections, locks, condition variables, how critical sections serialize code, how deadlock might occur, semaphores, and monitors. There is a Pthreads program example. It is not expected that you become a Pthreads programmer. Pthreads program examples are for illustration purposes. In this course, we do not write low-level Pthreads programming in assignments. Rather we focus on higher level tools.

OpenMP was introduced in Week 2 and is continued in Week 3, dealing with accessing shared data and synchronization, OpenMP critical, barrier, atomic, and flush.

Programming with Shared Memory-III covers how to recognize parallelism. It begins with generic notations to specify parallelism, including the **forall** notation, which will appear later. A basic way to recognize parallelism is to use Bernstein's conditions, which is described. You are expected to be able to apply Bernstein's conditions to a series of statements to determine whether they can be executed together or in any order.

Processor caches are provided so that code and data can be retrieved by the processor at a higher speed than using the main memory. This topic is covered fully in an architecture class. Here we are interested in a negative effect on the performance of parallel programs called *false sharing*. When a processor makes request for a memory location and it is not in its cache, a set of consecutive locations from memory (called a block or line) containing the requested location are brought in the cache. This is done because it is expected that the other locations will be needed in the future and it is more efficient to transfer the complete block or line. A problem arises in a multiprocessor system is that consecutive locations might be used store data items used by different processors but not shared. In a multiprocessor system, there will be multiple caches. If a data item is altered in the cache of one processor, any copies of the complete line holding the data item in other caches must be updated or invalidated (depending upon the cache coherence policy). Either way is not good for performance. If the line is invalidated, when the cache is accessed by their respective processors for the same line, they must retrieve the line from the main memory even though they had not altered their data items in the line. The final part of the slides describes the concept of *sequential consistency*. You would be expected to be able to define this term.