

Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.846: Parallel Computing

Spring 2008, Agarwal

Handout #18

Homework # 2

Due: Mar 13

More Computational Models

The first homework exposed you to the messaging computational model. These exercises expose you to two more: the stream computational model and the shared-memory model. As you do these exercises, try to develop some insights into the benefits and drawbacks of each model. For example, did you find one model easier to program and to debug? Was performance easier to obtain in some model?

Ex 1: The stream computational model As we saw in homework 1, the messaging computational model using iLib messages provides a simple interface to get a parallel Jacobi relaxation implemented. However, for the problem sizes we looked at, there was not enough work per Jacobi iteration to amortize the overhead of the message.

For this exercise, re-implement the Jacobi relaxation using iLib's raw channels. The stream computational model is particularly relevant in multicores because on-chip communication between cores can be implemented with very low latency compared to communication in discrete multiprocessors. Attach your new code, the output of your program, and the speedup of the channel based computation.

Since Jacobi communication patterns in each iteration are identical, it is possible to establish communication channels between neighboring cores once at the beginning of the program. Each iteration will then use the established channels for data communication. You may want to use buffered channels initially to get the program working. Then use raw channels to improve performance. The final code you turn in should use raw channels.

Be sure to read the channels section of the iLib API manual before starting this homework. The iLib manual has a detailed discussion of each type of channel and several examples.

Ex 2: The shared-memory work-queue computational model

The *work-queue parallelization model* (also called the *self-scheduled model*, *work-stealing model*, or *work-pile model*) is easy to implement in shared memory and is a key computational model for multicore programming. In this model, each core runs a “worker” process that takes work off a shared queue of “work to be done”.

The worker processes running on each core take work off a shared queue when they are idle. Worker processes may possibly generate a few more tasks to put on the queue, then go back to look for more work. Access to the queue must be mediated via locks of some kind (*e.g.*, via `ilib_mutex_lock()`).

Workers who went looking for work when none was available could do some form of backoff while testing the queue. In other words, when you attempt to get some work and do not find any, you then wait for some time and then check again. If you do not find work the second time around, you wait for a longer duration before trying again, and so on. Alternatively, workers could poll, or continuously check the queue for work. One interesting issue for this kind of strategy is determining the termination condition, *i.e.*, when can all the nodes assume that no more work is forthcoming?

For this exercise, implement the “naive” branch and bound algorithm for solving the traveling salesman problem using iLib shared memory to implement a work queue with one worker process per core. Recall that the naive algorithm explores the search tree of possible routes, at each city keeping track of the accumulated distance (or cost) to reach that city.

The bound variable and the work queue should be allocated in shared memory using `malloc_shared()`. You may find it is helpful to allocate other data in shared memory as well. You will want to use an `ilibMutex` object to control access to the shared data. The iLib API manual describes the use of the `ilibMutex` and contains several examples of iLib shared memory programs.

(1) Hand in the source code for your implementation of parallel TSP, as well as the relevant parts of the output for a one- and four-core implementation. Use the following inter-city costs in a TSP problem involving the 5 cities, A, B, C, D, and E:

	A	B	C	D	E
A	∞	2	3	1	∞
B	∞	∞	5	7	4
C	3	1	∞	6	1
D	8	2	4	∞	3
E	7	∞	6	1	∞

(2) In the above implementation, the bound variable can easily become a serial bottleneck. Suggest how you might modify your implementation to alleviate this bottleneck, possibly allowing some amount of wasteful searching in the process. (Note: although you are free to do so, you are not required to implement this change).

Suggestions: The key to this assignment is the implementation of the work queue data structure. We strongly suggest that you get the work queue implemented and tested in a single-core program before parallelizing the code. Once the queue is implemented, the additional work of this problem is implementing the termination condition and synchronizing the worker processes.

Each entry in the work queue should represent one work unit. Feel free to define your own work units, but we recommend an implementation where the work unit is the task of taking an existing incomplete tour and updating it by one city. For example, one work unit might be taking the tour A-C with two cities, creating all possible three-city tours

that start with A-C, and adding the three-city tours that are under the current bound to the work queue.

Some items that might be useful to include in a single work queue entry: the length of the partial tour, the cities (and order) of the partial tour, the current cost of the partial tour. Note that the current bound should not be included in the work unit, as it should be accessed and updated separately from the queue.