This assignment has three objectives: first, to become familiar with the Exemplar program development and execution environment; second, to develop an understanding of the decomposition and assignment steps of the parallelization process; and finally, to understand the performance impact of cache coherence in small-scale shared memory multiprocessors.

Particulars

For this assignment you will write a parallel program for blocked LU decomposition (without pivoting). This program will be written for the HP/Convex Exemplar using parallelism extensions to the C language (Exemplar C). You will need to use only the loop parallelism directive (loop parallel). To help you get started with the assignment, I have provided the following files in my home directory (~vijayk/homework2/) on billie.ncsa.uiuc.edu:

- lu.c: Sequential version of the LU decomposition code. The subroutines are written to handle blocking, even though the main function invokes these routines with a block size of 1.
- ttimers.c: Per-thread CPU and wall-clock timing routines. As lu.c demonstrates, you should use the wall-clock times for performance measurements.
- Makefile.{seq,convex}: Makefiles for a generic sequential machine (assumes availability of gcc) and the Exemplar. The latter makefile contains the options needed to turn off automatic parallelization by the compiler.
- job.script: A script for submitting jobs to the lsbatch system on the Exemplar. You will need to use the batch queues to get valid performance measurements.

The LU kernel factors a dense matrix into the product of a lower triangular and an upper triangular matrix, and is used to solve linear systems of the form Ax = b. The program for this assignment divides the dense n x n matrix A into an N x N array of B x B blocks (n = NB) to exploit temporal locality on submatrix elements. The pseudo code for the sequential algorithm is shown in the code fragment below (refer to any graduate-level numerical analysis text for details):

```plaintext
for k = 0 to N - 1 do
    factor diagonal block A_kk
    update all perimeter blocks in column k
    and row k using A_kk
    for j = k + 1 to N - 1 do
        for i = k + 1 to N - 1 do
            /* update interior blocks */
            A_ij = A_ij - A_ik * A_kj
```

Given this background, the assignment consists of four parts: the first three of which add up to the 10 points, and an optional fourth part (for extra credit):

1. (1 point) Modify the driver loop in lu.c to use the block size parameter provided on the command line. This should require only minor modifications to the loop bounds for each loop in the
main function. You should first test that the modified program works on a sequential machine before moving it back to the Exemplar.

2. (6 points) Given the sequential blocked LU decomposition program,

(a) First, identify the concurrency structure of the program. Restrict your attention to the driver loop in the main function. Assuming that the work done in a loop is entirely due to the floating point operations, what does the concurrency profile of LU look like? Note that the profile will need to be parameterized in terms of $n$ (the matrix size), and $B$ (the block size). Can you derive an analytical expression of speedup achievable with $P$ processors assuming that there is no parallelization overhead? You can assume that $n/B$ is sufficiently larger than $P$.

(b) Then, specify this concurrency structure using loop parallel directives. This should require no more than 5 or 6 pragma statements. For $n = 512$ and block sizes of 2, 4, 8, and 16, measure and plot the parallel program speedup with 1, 2, 4, 8, 12, and 16 threads. Use the single processor time with a block size of 16 as the sequential execution time. Note that to get valid measurements, you will have to use the batch queues. **However, please ensure first that your program behaves as expected with smaller numbers of threads (e.g. 2 or 4).** Are your speedup plots consistent with the analytical expressions of speedup? Can you list the various sources of parallelism overhead? Why does the program performance vary with block size?

3. (3 points) Modify the loop parallelism directives in the above program with the following attribute: `chunk_size = 1` (see Handout #4 for an example of how to do this). For $n = 512$, and block sizes of 2 and 16, measure and plot the speedup with 1, 2, 4, 8, 12, and 16 threads. Compare this plot against the speedup obtained for the same block size and number of processors but without the `chunk_size` annotation.

How does the `chunk_size` attribute influence parallelism overheads? Is the behavior different for the two block sizes? If yes, why?

4. (extra credit: 3 points) Parts 1-3 above have focused on exploiting loop-level parallelism. However, examining the fundamental dependencies in the application (ignoring loop structure) exposes additional concurrency. What are the fundamental dependencies? Can the resulting concurrency structure be easily specified? You will need to become familiar with how region-based parallelism is expressed, and fine-grained mechanisms for inter-thread synchronization (e.g., `gates`). **You do not need to actually write and measure this program: just describe what the program structure should look like.**

Guidelines

You are expected to hand in a write-up that should contain the speedup plots, and answers to individual questions asked above. Please also include a code listing of the modified portions of the program.