This assignment has two objectives: first, to develop an understanding of the decomposition and assignment steps of the parallelization process; and second, to understand the performance impact of load balance and cache locality 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 small-scale shared multiprocessors (SMPs) using the pthreads library.

To help you get started with the assignment, I have provided a sequential version of the LU decomposition code in my home directory (~vijayk/Homework2/) on bm.scs.cs.nyu.edu. The subroutines are written to handle blocking as described below, even though the main function invokes these routines with a block size of 1.

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 sequential algorithm for this assignment divides the dense $n \times n$ matrix $A$ into an $N \times N$ array of $B \times B$ blocks ($n = NB$) to exploit temporal locality on submatrix elements. The pseudo code for this algorithm is shown in the code fragment below (refer to any graduate-level numerical analysis text for details):

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 */
      $A_{ij} = A_{ij} - A_{ik} \times A_{kj}$

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.

2. **(3 points)** Given this blocked LU decomposition program, identify the concurrency structure of the program assuming that you will only be exploiting loop-level parallelism in the main function. In other words, parallelism in the program arises from the concurrent execution of independent loop iterations.

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 $P$. 

3. **(4 points)** Write a `pthreads` program, which exploits the concurrency structure identified above, while respecting required synchronization constraints. Your program should take the number of worker threads as a command-line argument, and partition the iterations of each parallel loop *cyclically* among these threads (iteration 0 run by thread 0, iteration 1 run by thread 1, ...).

Note that you should create the worker threads upfront, as opposed to creating/destroying them for each parallel loop. You may also want to define higher-level synchronization abstractions such as barriers, instead of managing all synchronization using low-level `pthreads` primitives. Finally, to simplify the work involved in the third part of the assignment, you may want to separate out (into a macro or a function), the decision of which loop iteration is executed by which thread.

Your program should achieve the correct result irrespective of the number of threads specified on the command line.

4. **(3 points)** For \( n = 512 \) and block sizes of 2, 8, and 32, measure and plot the parallel program speedup with 1, 2, and 4 threads (the last only on a 4-processor SMP) for the following two situations:

(a) *Default OS scheduling*: Scheduling of the program threads is left completely to the operating system. Recall that the OS attempts to *affinity schedule* threads, implying that a thread is likely to continue executing on the processor where it was originally scheduled.

(b) *Binding threads to processors*: Using platform-specific primitives, deterministically bind thread \( i \) to execute on processor \( i \).

On Solaris SMPs, this binding can be achieved using a two-step process: (1) ensuring that a `pthread` is realized as an LWP (a kernel thread) using the `pthread_attr_setscope` call with the `PTHREAD_SCOPE_SYSTEM` contention scope attribute; and (2) binding each LWP to a distinct processor using the `processor_bind` system call. Note that you will need to use the `sysconf` and `p_online` system calls to obtain a list of the legal processor numbers.

Directions for binding threads to processors on Linux SMPs will be posted on the web site in a day or two.

Explain your measurements for the two situations. Are your speedup numbers 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? Does the OS really *affinity schedule* the program threads?

5. **(4 points)** Extend your parallel program to implement the following two additional partitioning strategies:

(a) *Blocked Row Partitioning*: Thread \( i \) is assigned the matrix blocks in rows \([i*\frac{N}{P}, (i+1)*\frac{N}{P})\). Loop iterations are partitioned among the threads using the *owner-computes* rule: an iteration involving a matrix block is executed by the thread that is assigned that block.

(b) *Cyclic Row Partitioning*: Thread \( i \) is assigned the following rows of matrix blocks: \( i, i + \frac{N}{P}, \ldots \). As before, loop iterations are partitioned to threads using the *owner-computes* rule.

Compare the speedup performance of these two partitioning schemes with that achieved by the loop-based scheme above. In all cases, you should explicitly bind threads to processors to ensure that your measurements are deterministic.

To explain your measurements, instrument your program to breakdown the time spent by the threads for computation and for synchronization operations. Which of the schemes achieves the best load balance? Which of the schemes achieves the best locality? What are the cache traffic implications in each case?
6. (extra credit: 3 points) Parts 1-5 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 do not need to actually write and measure this program: just describe what the program structure should look like. Of course, if you really want to become proficient at pthreads programming, you may want to take a shot at writing and debugging this program. Does the excess concurrency translate into a noticeable performance improvement?

Guidelines

You are expected to hand in a write-up that should contain the speedup plots, and answers to the individual questions asked above. Your answers should be detailed enough to convince me that you understand the reasons for the observed performance. Please also include a code listing of the modified portions of the program.