Homework 2 Solution

CSE 661 - Parallel and Vector Architectures

2.5 (4 pts) In a uniprocessor, **we would use blocking instead of spinning** to implement event synchronization. When spinning is used, a process (the spinning process) continually reads a flag until another process (the releasing process) writes that flag and changes its value, releasing the spinning process. On a uniprocessor, both processes run on the same processor. Thus, the spinning process must release the processor so that the releasing process can run, before the releasing process can release the spinning process. This will happen when the spinning process's scheduling quantum times out, wasting CPU cycles. (If there were no quantum-timeout context switches imposed by the operating system, the spinning process would simply continue in an infinite loop on the processor and no one would ever be able to release it.) If the process blocks when it reaches the synchronization event, on the other hand, the releasing process will take over the processor very quickly (after the context switch overhead is incurred), so it can release the blocked process much more quickly.

In a multiprocessor, the releasing process is likely to be running on a different processor, so the above issue is not serious unless the releasing process is not running in parallel. The key tradeoff here is that a spinning process still consumes processor resources, which could be used by another process that is perhaps unrelated or perhaps part of the same application. On the other hand, the context switch required for blocking invokes the operating system and is expensive. Usually, spinning is better than blocking if either the spinning process is likely to be released soon or there is no other ready process to run. Otherwise, blocking may be better since it frees up the processor for other tasks. A commonly used solution that can be proved to be close to optimal is to spin for as long as it takes to perform a context switch, and then if the process is not yet released then block it (incurring the context switch cost).

2.6 (6 pts) The barrier in line 16a ensures that all processes have finished their initialization of the global *diff* variable before any process starts performing the next iteration. The reason is that otherwise some processes may perform their work for the next iteration, including accumulating their private *mydiff* variables into the shared *diff*, but after that another process that was slowed down somehow may set the shared *diff* to 0 at the beginning of its iteration, thus wiping out the accumulations of the *mydiff*s of other processes that had proceeded faster.

The barrier in line 25f ensures that no process initializes the shared *diff* variable to be zero for the next iteration before all have tested its value for convergence in the current iteration and reached a consistent verdict. Otherwise, some processes may think the computation has not converged and will enter the next iteration, one of these may zero out the *diff* variable, and then some processes that have not yet performed the test from the previous iteration may read the zero or low value for *diff*, think the computation has converged, and not enter the next iteration. The processes that entered the next iteration will then hang waiting for others to arrive at the first barrier in that iteration (i.e. the barrier in line 16a), but those others will never arrive.

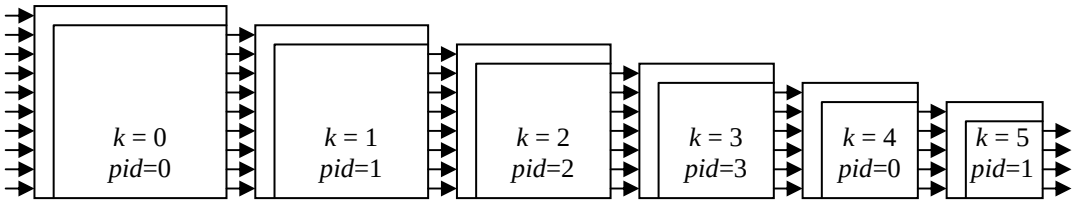
We can eliminate the first barrier in line 16a by doing some reordering of the computation. First, *done* is declared as a global variable initialized to 0. Similarly, the global variable *diff* is initialized to 0. Next, we eliminate the initialization of global variable *diff* in line 16. This will eliminate the barrier in line 16a. Next, we allow only process 0 to compute the value of *done* (line 25f) and to reset *diff* to 0 (line 25g) after the first barrier. A second barrier is needed in line 25i to ensure that all processes wait for the updated value of *done* and the reset of *diff* before starting the next iteration.

```
15.  while (!done) do
16.     mydiff = 0; /* remove initialization diff = 0 */
17.     for i = mymin to mymax do
18.        for j = 1 to n do
19.           temp = A[i,j];
20.           A[i,j]= 0.2 * (A[i,j]+A[i,j-1]+A[i-1,j]+
21.                 A[i,j+1]+A[i+1,j]);
22.           mydiff += abs(A[i,j]-temp);
23.        endfor
24.     endfor
25a.    LOCK(diff_lock);
25b.    diff += mydiff;
25c.    UNLOCK(diff_lock);
25d.    BARRIER(bar1, nprocs);
25e.    if (pid == 0) then
25f.       done = (diff/(n*n) < TOL);
25g.       diff = 0;
25h.    endif
25i.    BARRIER(bar1, nprocs);
26.  endwhile
```

2.9 **a.** (5 pts) The outer loop iterations indexed by *k* are assigned to processes in a cyclic fashion. For example if number of processes is 4, process 0 is assigned iterations *k* = 0, 4, 8, …, etc. Process 1 is assigned iterations *k* = 1, 5, 9, …, and so on. To synchronize the operation of processes on the various rows, a 1D array of integers called *row*[], indexed by the row index, is used to signal the availability of a given row to a given process. For example, if *row*[*i*] = *pid* then only process *pid* can compute row *i* and the other processes have to wait to get their turn. Once process *pid* has finished computing row *i* it assigns this row to next process (*pid*+1 **mod** *nprocs*) in a cyclic manner. Thus the computation or rows flows among the processes in a pipelined manner.

```
shared int n, nprocs;

main()
begin
  read (n); read(nprocs);
  float A[][] = G_MALLOC(n by n floats);
  read(A);
  int row[] = G_MALLOC(n) initialized to 0;
  CREATE(nprocs-1, Eliminate, A);
  Eliminate(A);
  WAIT_FOR_END(nprocs-1);
end main
```




```
procedure Eliminate(float A[][])
begin
  int i, j, k, pid=getpid();
  for k = pid to n-1 step nprocs do
  begin
     // Busy wait until row[k] is made available to process pid
     while (row[k] != pid);
     for j = k+1 to n-1 do
       A[k][j] = A[k][j] / A[k][k];
     endfor
     A[k][k]=1;
     for i = k+1 to n-1 do
        // Busy wait until row[i] is made available to pid
        while (row[i] != pid);
        for j = k+1 to n-1 do
           A[i][j] = A[i][j] – A[i][k] * A[k][j];
        endfor
        A[i][k] = 0;
        // Make row i available to next process in cycle
        row[i] = pid+1 mod nprocs;
     endfor
  endfor
end procedure
```

b. (5 pts) For message-passing, we will partition the rows of the 2D matrix in a cyclic manner among the processes. For example, if 4 processes exist then process 0 will allocate space for rows 0, 4, 8, etc. Process 1 will allocate space for rows 1, 5, 9, and so on. The last process (*pid* = *nprocs* – 1) will read matrix *A* row by row and send each row as a message to process 0. Each process receives all the sent rows from the previous process, computes and stores the pivot row in its address space, and then computes and sends the remaining rows to the next process. The rows are received and sent one by one in a pipelined manner. At some given time, all processes will be simultaneously active computing different rows of the matrix. The best combination of SEND and RECEIVE primitives is to have a non-blocking asynchronous SEND with a blocking synchronous RECEIVE, so that a process will synchronize its row computation with the arrival of the row.

```
int n, nprocs;

main()
begin
  read (n); read(nprocs);
  CREATE(nprocs-1, Eliminate);
  Eliminate();
  WAIT_FOR_END(nprocs-1);
end main
```

```
procedure Eliminate()
begin
  int i, j, k, pid=getpid();
  int prevpid = (pid == 0 ? nprocs-1 : pid-1);
  int nextpid = (pid == nprocs-1 ? 0 : pid+1);
  float myA[][] = malloc(n/nprocs by n);
  float onerow[] = malloc(n);

  // Last process will read matrix A
  // And will send its rows one by one to process 0
  if (pid == nprocs – 1) then
     for (i=0 to n-1) do
        read(&onerow[], n);
        SEND(&onerow, n*sizeof(float), 0, i);
     endfor
  endif

  for k = pid to n-1 step nprocs do
  begin
     // Receive pivot row k from previous process
     RECEIVE(&onerow[k], (n-k)*sizeof(float), prevpid, k);
     for j = k+1 to n-1 do
        // Compute and store pivot row k
        // Map pivot row k to myA at row index [k/nprocs]
        myA[k/nprocs][j] = onerow[j] / onerow[k];
     endfor
     myA[k/nprocs][k]=1;
     for i = k+1 to n-1 do
        // Receive rows k+1 to n-1 one by one
        RECEIVE(&onerow[k], (n-k)*sizeof(float), prevpid, i);
        for j = k+1 to n-1 do
           onerow[j] = onerow[j] – onerow[k] * myA[k/nprocs][j];
        endfor
        // Send row i to next process
        SEND(&onerow[k+1], (n-k-1)*sizeof(float), nextpid, i);
     endfor
  endfor
end procedure
```