ECE453: Advanced Computer Architecture II
Homework 1

Assigned: January 18, 2005

From *Parallel Computer Architecture: A Hardware/Software Approach*, Culler, Singh, Gupta:

1.15) The cost of a copy of n bytes with a transfer rate of B_{copy} is given by:

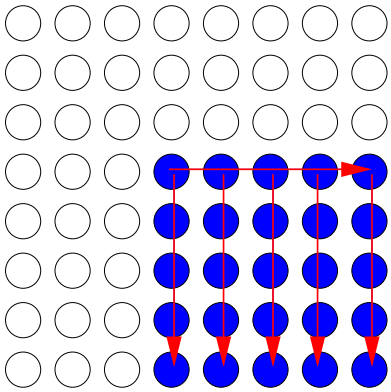
$$Copy\ Time(n) = \frac{n}{B_{copy}} = n \times \frac{5\ cycles}{4\ bytes} \times \frac{1\ second}{100\ MHz} = n \times 12.5\ ns$$

The message time is $T_0 + \frac{n}{B_{msg}}$, which is $100,000\ ns + n \times 12.5\ ns$. So the total cost for the transfer as a sum of the message and copy times is:

$$Transfer\ Time(n) = 100,000\ ns + n \times 25\ ns$$

The cost of the copy effectively halves the bandwidth of the transfer. Assuming that a fast call to the OS takes $10us$, this would be the time to copy 400 bytes.

2.7) (a)



(b) Shared address space code for main loop:

```
void gauss_sas(float **A, n)
{

    int mymin = (pid * n/nprocs);            /* assumes that n is divisble */
    int mymax = mymin + n/nprocs - 1;        /* by nproc */

    for(k = 0; k < n; k++) {                 /* loop over all rows */
BARRIER(bar1, nprocs);               /* wait for everybody */

if (k >= mymin && k <= mymax) {      /* only one thread */
    for(j = k+1; j < n; j++)         /* (which owns row k) */
A[k][j] = A[k][j] / A[k][k]; /* should execute this block */
    A[k][k] = 1.0;
}

BARRIER(bar1, nprocs);               /* wait for everybody */

/* loop over rows within assignment */
for(i = max(myminx,k+1); i <= mymax; i++) {
    for(j = k+1; j < n; j++)
A[i][j] = A[i][j] - A[i][k] * A[k][j];
    A[i][k] = 0.0;
}
    }
}
```

(c) Message passing code for main loop:

```
void gauss_msg(float **myA, n)
{
    int mymin = (pid * n/nprocs);            /* assumes that n is divisble */
    int mymax = mymin + n/nprocs - 1;        /* by nproc */

    for(k = 0; k < n; k++) {                 /* loop over all rows */
if (k >= mymin && k <= mymax) {
    /* the thread responsible for this row will perform computations */
    for(j = k+1; j < n; j++)
myA[k][j] = myA[k][j] / myA[k][k];
    myA[k][k] = 1.0;

    /* broadcast results to the remaining threads */
    for(i = pid; i < nprocs; i++)
SEND(&myA[k][0], sizeof(float) * n, i, ROW);
}
else if (k < mymin) {
    /* wait for the results if below k in matrix */
    RECEIVE(&myA[k][0], sizeof(float) * n, k / nprocs, ROW);
}

/* loop over rows within assignment */
for(i = max(myminx,k+1); i <= mymax; i++) {
    for(j = k+1; j < n; j++)
myA[i][j] = myA[i][j] - myA[i][k] * myA[k][j];
    myA[i][k] = 0.0;
}
    }
}
```

(d) The first problem exists even for a sequential program. Consider the time and storage complexity of Gaussian elimination. For an n-by-n matrix, the time complexity is $O(n^3)$ and the memory requirement is $O(n^2)$. This means that on average every data element in the matrix is accessed n times. If the processor cache were big enough to fit the entire matrix, then we would have n cache hits for every element and one cache miss the first time we bring the element in.

However, for large n, the matrix does not fit in the cache; in this case, by the time the processor accesses an element A, then accesses the rest of the row or rows of elements before coming back to access A again, A has likely been replaced from the cache due to capacity or conflicts. So we may incur cache misses on A as many as n times. This would hurt performance substantially.

In a shared address space, an element being accessed n times may be allocated in a remote memory, so the misses can be very expensive and can also cause a lot of network traffic and contention. In the message passing model, another problem with the current structure of the program is when a pivot element (or a pivot row element) has to be communicated to other processors, a message has to be sent containing only a single element. The overhead of sending a message is high, as we

shall learn soon, and it is not amortized over data in this case. We would like to find a method in which we can do a lot computation on a chunk of data that fits in the cache without accessing too much other data, thus obtaining many cache hits, and then transfer that chunk of data in a large message to other processors that need it.

3.2) The major advantages are a possible increase in locality if a process requests tasks from its own local queue most of the time, and a reduction in contention since all the processes are not fighting to use a single global task queue. Disadvantages are rooted in complexity issues, including choosing which queues to place tasks in, when and where to steal tasks from, and determining when all tasks are done (termination detection).

Small tasks generally increase overhead in task management because the task queues are accessed more often. With distributed task queues, the communication and contention will be increased if locality is not preserved. However, if a process can find tasks in its own queue the majority of time, the impact will be small.

3.4) Since busy-useful time is defined as the time doing useful work that would also have to be done in an equivalent sequential program, if every instruction takes the same amount of time in the sequential and parallel cases then busy-useful time should not be different when summed across processes compared to the sequential program, unless a different basic algorithm is used in the parallel case than the sequential algorithm. All additional busy time should be included in the busy-overhead component. Differences in interactions with the processor pipeline and instruction scheduling and the effects of the memory system on it are one reason that the busy-useful time in the parallel case may be different from the sequential case.

3.5) The first question to address is whether to exploit only topological parallelism, only innode parallelism, or both. The next, and related, question is whether to exploit the parallelism statically or dynamically.

Topological parallelism, in which each node is processed sequentially, has the advantage of higher data locality in the computation within a node, since it is all done by one processor. However, there many not be enough tasks to balance the workload effectively, and the individual tasks may be too large and imbalanced. Static load balancing is especially difficult, since even if we can predict the relative workloads of different nodes for a given problem size, and use the dependences among nodes to schedule the tasks for load balance, the relative workloads of nodes will change with problem size. So this approach is usually managed dynamically. As a result, there can be significant losses of locality when a child node is processed by a different processor than its parent, since data written by the processor computing the child node will be read by a different processor when computing the parent.

In-node parallelism, in which all processors collaborate on each node and synchronize before moving between nodes, has the advantage that load balance is easier to provide statically, since the relative work for different entries within a node is usually quite uniform or at least quite predictable and similarly varying even when problem size changes. The other important advantage is that if the computation can be scheduled properly, then locality can be preserved across nodes; i.e. when processors move from a node to its parent, we may be able to partition the nodes so that the entries that a processor writes when working on the child node are the entries it reads when working on the

parent. The disadvantage is greatest when nodes are small; in this case, there may not be enough work within a node to partition in a load balanced way among all processors, and the cost of synchronizing across nodes may be too high relative to useful work. The other general disadvantage of in-node-only parallelism in some computatons is providing load balance, especially when nodes are small or the computation within them is not predictably distributed.

Exploiting both forms of parallelism combines the advantages and disadvantages. It can be orchestrated by breaking up each node into tasks that are chunks of matrix computations, and then assigning the tasks either fully statically, fully dynamically, or with a good static guess followed by dynamic task stealing. One strategy that is sometimes used is to have all processors collaborate on every node, but use topological parallelism to avoid synchronizing when moving from one node to another. When nodes are large, or when most of the computation is in a few nodes, and when the dependences across matrix elements in parent and child nodes are such that locality can be managed statically across nodes, in-node only paritioning is found to work well.