Parallel Programming, Algorithms
and Architecture
Midterm Exam
November 2, 2000
Take Home, Open Notes and Reference Texts
Due Tuesday November 7, 2000
beginning of class

Name:

Total:
Problem 1 (20 points)

(a) 10 pts. Suppose you have a 10 stage pipelined adder, how many cycles would it take to add of 1000 pairs of numbers? What is the computational rate (operations per cycle) achieved? How close is it to the limiting performance of the pipeline?

Solution: The first result is available after 10 cycles. After that the remaining 999 results appear at a rate of one per cycle. Therefore we have a total of 1009 cycles to compute the 1000 results. This yields a computational rate of \( r_n^v = \frac{1000}{1009} = .991 \) operations per cycle. Since the limiting performance of any pipeline is one result per cycle we see that a vector length of 1000 for this 10 stage pipe yields a very large fraction of the value of \( r_{\text{inf}}^v = 1 \).

(b) 10 pts. Suppose the workload of your code spends 90% of the time in vectorizable operations. A vendor proposes that you upgrade to a machine that includes a vector unit that is 100 times faster than its scalar unit (which performs the same as your current scalar unit). Relative to the price of your current system, what is the most you would agree to pay for the new machine? Justify your answer.

Solution: This solution requires an application of Amdahl’s law. Assuming the time on the original machine is scaled to \( t_{\text{old}} = 1 \) we have the time on the new machine as:

\[
t_{\text{new}} = 0.1 + \frac{0.9}{100} = 0.109
\]

So by adding a vector unit that is 100 times faster than scalar processing we have reduced the time from \( t_{\text{old}} = 1 \) to \( t_{\text{new}} = 0.109 \). This is a factor of 9.17. Therefore, it certainly does not make sense to pay a factor of 100 more for the machine. In fact, paying a factor of 9 or 10 more than the current machine price would be the most that this analysis would support.
Problem 2 (20 points)

Consider the following sequential code segments and determine the data dependences that would hinder parallel execution. The parallel code is to be consistent with the sequential code, i.e., it is to produce the same results in all of the variables. Remember we discussed true dependences where a statement producing data must be executed before a statement consuming that data, antidependences where a statement reading data must execute before another statement updates the data. Identify all such dependences and any other types that would prevent the parallelization of the code and its producing sequentially consistent results.

(i)
S1: \[ A = B + D \]
S2: \[ C = A \times 3 \]
S3: \[ A = A + C \]

(ii)
\[
\begin{align*}
\text{do } i1 & = 1,10 \\
\text{do } i2 & = 1,10 \\
\text{do } i3 & = 1,10 \\
S1: & \quad a(i1,i2,i3)=b(i1-1,i2,i3)*c(i1,i2)+d*e \\
S2: & \quad b(i1,i2,i3)=a(i1,i2-1,i3)*f(i1,i2) \\
\end{align*}
\]

(iii)
\[
\begin{align*}
\text{do } i=1,n \\
S1: & \quad a(i)=b(i)*c(i) \\
\text{do } j=1,n \\
S2: & \quad d(i,j)=a(i-1)+e(i,j-1) \\
S3: & \quad e(i,j)=d(i,j-1)+f \\
\end{align*}
\]

\[
\begin{align*}
\text{do } k=1,n \\
S4: & \quad g(i,k)=h(i-5)+1 \\
\end{align*}
\]

\[
\begin{align*}
S5: & \quad h(i)=\sqrt{a(i-2)} \\
\end{align*}
\]
For part (i) we have true dependences from S1 to S2 and S1 to S3 because of A. We have a true dependence from S2 to S3 because of C. We have an antidependence from S2 to S3 because of A. We also have what is known as an output dependence from S1 to S3 that is defined as two statements that have the same output variable and therefore must update in the proper order in order to guarantee that the output variable value is consistent with sequential execution.

For part (ii) we have a true dependence between S1 and S2 due to the read of \( a(i_1,i_2-1,i_3) \) in S2 which would have been produced by S1 on the previous iteration of the i2 loop.

We also have a true dependence from S2 to S1 due to the read of \( b(i_1-1,i_2,i_3) \) which would have been produced by S2 on the previous iteration of the i1 loop.

For part (iii) we have a true dependence between S1 and S5 due to the access of \( a(i-2) \) in S5. We have a true dependence from S2 to S3 due to the access of \( d(i,j-1) \) in S3. We have a true dependence from S3 to S2 due to the access of \( e(i,j-1) \) in S2. We also have a true dependence from S5 to S4 due to the access of \( h(i-5) \) in S4.
Problem 3 (20 points)

Consider the matrix vector product \( y \leftarrow Ax \) where \( A \) is a square matrix of order \( n \). Recall there are two main approaches. The first uses dotproducts, i.e.,

\[
\eta_i = \sum_{j=1}^{n} a_{i,j} \xi_j
\]

where \( y \) consists of \( \eta_i, i = 1, \ldots, n \), \( x \) consists of \( \xi_i, i = 1, \ldots, n \) and \( A \) consists of \( a_{i,j} i = 1, \ldots, n \) and \( j = 1, \ldots, n \). The second uses a series of vector triads

\[
y = \xi_1 a_1 + \xi_2 a_2 + \cdots + \xi_n a_n
\]

where \( a_i \) is the \( i \)th column of \( A \).

Now suppose \( A \) is also a sparse matrix, i.e., most of the elements are zero, and it is stored using the compressed row format data structure described below.

Let \( nz \) be the total number of nonzero elements in \( A \). Let \( A(1:nz) \) be an array in which the nonzeros of \( A \) are grouped together so that elements of the same row are contiguous. No assumption can be made about the order of the elements within each row, i.e., they cannot be assumed to be sorted by column index. No assumption can be made about the order of the rows themselves, i.e., the group of contiguous elements for row \( j \) might appear before or after the group of contiguous elements for row \( i \) regardless of the relationship of \( i \) and \( j \). Let \( \text{CPTRS}(1:nz) \) be an integer array where \( \text{CPTRS}(i) \) contains the column index for the nonzero element stored in \( A(i) \). Finally we assume that we have two arrays \( \text{RS}(1:n) \) and \( \text{RE}(1:n) \) such that \( A(\text{RS}(i)) \) is the first element in row \( i \) and \( A(\text{RE}(i)) \) is the last element in row \( i \), i.e., \( \text{RS}(i) \) points to the beginning of row \( i \) in \( A(1:nz) \) and \( \text{RE}(i) \) points to the end of row \( i \) in \( A(1:nz) \).

Example: For the matrix

\[
\begin{bmatrix}
1 & 0 & 0 & 0 & 2 \\
5 & 2 & 3 & 0 & 0 \\
0 & 0 & 1 & 0 & 3 \\
7 & 0 & 0 & 3 & 0 \\
0 & 0 & 0 & 0 & 8
\end{bmatrix}
\]

one possible layout in the data structure is
• $A(1:10) = (1, 3, 5, 2, 3, 2, 1, 7, 3, 8)$
• $\text{CPTRS}(1:10) = (3, 5, 1, 2, 3, 5, 1, 1, 4, 5)$
• $\text{RS}(1:5) = (6, 3, 1, 8, 10)$
• $\text{RE}(1:5) = (7, 5, 2, 9, 10)$.

(It is one possible version due to the arbitrary ordering of elements within a row and the rows within $A$.)

1. Describe how you would implement $y \leftarrow Ax$ on a parallel shared memory architecture. This data structure should allow a simple synchronization strategy, e.g., doalls only. Describe your implementation using the Cedar Fortran constructs of the notes.

2. On the same architecture, describe how you would implement $y \leftarrow A^T x$ where $A^T$ denotes the transpose of $A$. If $B = A^T$ then the $i, j$ element of $B$ is the same as the $j, i$ element of $A$. **Assume that you must use the data structures given above.** Justify your choice of synchronization strategies. Hint: consider carefully the use of one or more critical sections.
Solution:

The data structure given is clearly a row-oriented structure, i.e., it is easy to get at the rows of $A$. Since the inner product or dot product from consists of an independent dot product for each element of $y$ we have a simple implementation consisting of a doall loop where each iteration computes one element of $y$. Note the indirect addressing required to get the appropriate elements of $x$ involved in the sparse dot product.

```plaintext
doall i = 1,n  
  integer j  
  real s  

loop  
  s = 0.0  
  do j= rs(i),re(i)  
    s = s + a(j)*x(cptrs(j))  
  end do  
  y(i) = s  
end doall
```

The second part of the question essentially consists of a matrix multiplication where the rows of the matrix $A^T$ are not readily available and therefore the above approach is not necessarily the best choice. In this case the columns of $A^T$ are readily available so the column-oriented algorithm is the natural choice. However, the triads are not independent since the scaled columns of $A^T$ must be added together some sort of synchronization is required. Therefore we must choose the form of synchronization. The simplest is to have a single lock critical section protecting all of $y$ but this will lead to excessive conflicts and delays. The other extreme is to have a lock protecting each element of $y$. A spectrum of methods can be created by using $m$ locks say in array $lck(0 : m - 1)$ and using $lck(i)$ to protect all elements of $y$ with a particular subscript range, e.g., $lck(i)$ protects $\eta_j$ where $i = j \mod m$. The code would consist of a doall where each iteration scales a column of $A$ with the appropriate element of $x$ and then locks the appropriate lock for each of the elements as they are added to the appropriate element of $y$. 

7
doall  i = 1,n
    real  s
    integer j, k

loop
    do  j=rs(i),re(i)
        s = x(i)*a(j)
        k = mod(cptrs(i),m)
        call lock(lck(k))
        y(cptrs(i)) = y(cptrs(i)) + s
        call unlock(lck(k))
    end do
end doall
**Problem 4**

Consider the matrix vector product \( y \leftarrow Ax \) where \( A \) is a square matrix of order \( n \). Suppose your are to implement this on a distributed memory processor. The partitioning of the data is given below. You do not have to worry about how the data got that way. Your solution should start from the assumed data positioning.

A block column partitioning of \( A \) is assumed. Each processor has \( k = n/p \) contiguous columns in its memory. The vector \( x \) is not distributed, all \( n \) of its elements are stored in processor 0. The solution \( y \) is to end up distributed with each element stored in the processor that contains the column of \( A \) with the same index, i.e., each processor has \( n/p \) contiguous elements of \( y \). So processor 0 has columns 0 through \( n/p \), and the elements of \( y \) with these same indices, and all of \( x \). Processors 1 to \( p - 1 \) have similar portions of \( A \) and \( y \) and none of \( x \).

Describe how you would implement this primitive using MPI subroutines for message passing. You do not have to write the complete code (unless you want to) but you should be very specific about what subroutines you use, when you use them, and what they accomplish.

Use as many global communication primitives as possible as these tend to reveal patterns in the communication that can be exploited for efficiency’s sake. Recall that we discussed: broadcast, scatter, gather, allgather, and alltoall. A subset of these will be very useful for this problem.
Solution:

Let $A_i$ refer to the $n \times n/p$ matrix consisting of the columns stored in processor $i$ for $i = 0, \ldots, p - 1$. Let $x_i$ and $y_i$ refer to subvectors of size $n/p$ elements that contain the elements of $x$ and $y$ with the indices corresponding to the column indices in $A_i$.

Clearly we must first get $x_i$ to processor $i$. This is simply a scatter from processor 0 to all of the processors. We can then compute in each processor $A_i x_i = t^{(i)}$. Here $t^{(i)}$ is a vector with $n$ elements and we know that $y = \sum_{i=0}^{p-1} t^{(i)}$. Recall however that we want $y_i$ in processor $i$ so the $t^{(i)}$ must be reshuffled. Specifically, let

$$t^{(i)} = \begin{pmatrix} t^{(i)}_0 \\ \vdots \\ t^{(i)}_{p-1} \end{pmatrix}$$

where $t^{(i)}_j$ is the contribution of $t^{(i)}$ to $y_j$. This means we need all of the $t^{(i)}_j$ to be sent to processor $j$ for $j = 0, \ldots, p - 1$. So if we view the $t^{(i)}$ as forming a matrix $T$ that is $n \times p$ we must transpose it so the columns of $T^T$ are stored in each processor. This is easily accomplished using the ALLTOALL global communication routine described in the notes and in the MPI documentation. Specifically,

$$t^{(i)} = \begin{pmatrix} t^{(i)}_0 \\ \vdots \\ t^{(i)}_{p-1} \end{pmatrix}$$

is viewed as the input buffer on each processor where $t^{(i)}_j$ is the message of length $n/p$ sent by processor $i$ to processor $j$. After the global ALLTOALL processor $j$ has in its memory $t^{(i)}_j$ for $i = 0, \ldots, p - 1$. This is exactly the data needed to compute $y_j$ locally and complete the matrix vector product with the result vector distributed in the desired fashion.
Problem 5

Suppose you are responsible for designing the protocol to control access to a single I/O resource on a distributed memory machine. We assume that any process can perform I/O but only one can perform it at any particular time. If the machine used a shared memory paradigm this access could be controlled by implementing a critical section using, say, locks. However, the distributed memory paradigm has no such support. The I/O resource itself is not a process involved in the MPI message passing activity, i.e., you should only worry about the MPI processes involved in the protocol. Communication between the I/O resource and the process that is allowed to use it will be done via special I/O routines that are not relevant here.

- Describe a strategy that would allow the processes to control their access to the I/O resource while respecting the above critical section restriction.

- How would you expect the performance to change as the number of processes involved increases?
**Solution:** There are several ways to do this but there are three main approaches. The first is a **centralized decision** in which all processes attempting to seize control of the I/O resource send a message to one process that acts as the I/O arbitration master, say it is process 0. Process 0 takes all of the requests that come in within a certain window of time decides based on some sort of priority scheme that guarantees that no process is locked out of I/O forever and that is fair. It then notifies each process with a pending I/O request whether or not they have been granted exclusive access to the I/O resource. The process that has proceeds with the I/O the others resend their request for the next round of arbitration.

A similar approach is a **distributed decision** where all of the requests are broadcast to everyone. Therefore everyone knows which processes are requesting control and they can all perform the arbitration scheme. Each process then knows the result and the one that is granted access can proceed.

The final approach that is often used in a distributed memory environment is a **token-based approach**. In this strategy an I/O token is passed continuously among the processes. A process must wait until the token arrives in order to proceed with its pending I/O. When it is finished with the I/O it puts the token back in circulation. The order in which the token moves between the processes determines the priority scheme.