

Parallel Architectures and Systems

Homework 3

Dr. Edwin Sha

Due Date: **1PM, March 30, 2010**

Remember that you are a GRADUATE student. The homework is to help you to learn. You should have no problem to finish the homework in time. In this course, many homework questions do not have standard solutions. And many questions are intended to be vague. So try your best and make your reasonable assumptions. But put down your assumptions in the report. Enjoy the homework! If you have questions, ask TA first.

PUT DOWN YOUR EMAIL ADDRESS IN YOUR REPORT. By the way, I don't like handwritten report. Please use some computer formatting tool such as LaTeX. To save your time, you can draw figures by hand if you do not know how to draw figures using tools such as xfig.

1. (5 points) This is not a real question. I want you to begin your term project. Term project is a very important part of the course. The term project can be ANYTHING related to "parallel" stuffs or anything you like to study further. Write a title and a short description to describe the idea of your term project. It can be changed later. The term project can be done by 3, 4 or 5 students. You should not work alone.

List **the names of your partners** clearly. Check the course web page to see some suggested ideas.

2. Given the following code segments on a shared address machine with each processor P_i , say what results are not possible under sequential consistency (SC). Assume that all variables are initialized to 0 before this code is reached.

(A) (8 points) What values of (u, v, w) are not possible. Explain.

P1: $A = 1;$

P2: $u = A; B = 1;$

P3: $v = B; w = A;$

(B) (8 points) What values of (u, v, w, x) are not possible. Explain.

P1: $A = 1;$

P2: $u = A; v = B;$

P3: $B = 1;$

P4: $w = B; x = A;$

3. (15 points) Try reordering the loops in the matmul routine (Matrix Multiplication) for multiplying N-by-N A, B arrays. Do the program as listed in the following pseudo code.

```
float *a, *b, *c;
a = malloc(sizeof(float)*N*N); //N is the size
// then malloc for b and c
int init= 1325;
for (i=0; i<N; i++){
    for (j=0; j<N; j++){
        init = 3125*init % 65536;
        a[i,j] = (init - 32768.0) / 16384.0;
        init = 3125*init % 65536;
        b[i,j] = (init - 32768.0) / 16384.0;
        c[i,j] = 0.0;
    }
}
c1 = clock();
matmul(a,b,c,N);
c2 = clock();
```

- (A) Try all the six orders of **ijk**, tell me if the results are the same? You can print out some elements of C to check.
- (B) Run your program in babbage.utdallas.edu or apache only. Use **gcc -O3** to compile. And try the following different orders: **ijk**, **ikj** and **jki**. Try N= 400, 700 and 1000 for several times. What is the running time of running matmul routine for each case? Which one is the fastest? List the running times.
4. (10 points) The well-known Amdahl's Law shows the speedup is upper bounded by a constant $1/s$ where s is the sequential part of the program. It is quite discouraging for parallel processing. But in the Example in Slide 10 of Chapter 2, it shows that the speedup limit can be almost linear to the number of processors. What is going on? Is Amdahl's law wrong? Explain it. You also can derive another speedup limit?
5. (15 points) Question 2.6 in the textbook. See the enclosed hard copy if you dont have the book. And see the Slide 28 for the program.
6. (15 points) Question 2.7 in the textbook. See the enclosed hard copy if you dont have the book. Answer those a, b, c and d. Hint for question d: consider cache for shared address machine and consider message numbers and sizes for message passing machine.

7. (10 points) You may want to consult with MPI documents on the web to answer this question.
(A) Explain clearly the meaning of the "completion of a synchronous send (SSEND)" in MPI?

(B) Will the following MPI code be deadlock or always safe? Explain your reason. Note that the count might be large.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_SSEND(a, count, MPI_REAL, 1, tag0, comm, ierr)
    CALL MPI_SEND(b, count, MPI_REAL, 1, tag1, comm, ierr)
ELSE    ! rank.EQ.1
    CALL MPI_IRECV(a, count, MPI_REAL, 0, tag0, comm, r, ierr)
    CALL MPI_RECV(b, count, MPI_REAL, 0, tag1, comm, ierr)
    CALL MPI_WAIT(r, status, ierr)
END IF
```

8. (20 points) For this question, you do not need to really RUN the program using MPI. Each one writes an MPI program BY HAND to compute the expected value and the variance for a random variable X . Let an array V from $V(1)$ to $V(N)$ store all the possible values of X and let an array P store the probability values such that $P(i)$ is the probability that $X = V(i)$. The expected value of X , $E(X)$, is computed by $\sum_{i=1}^N V(i) \times P(i)$. And the variance is computed as $E((X - E(X))^2)$.

You should use ONLY collective communication operations such as scatter, gather, reduce, etc, for this program. Assume that Process 0 initially has array V and P , and it will distribute the loads evenly among all the processes. Let M be the number of processes in MPI_COMM_WORLD. To make things simple, assume M divides N and all the values are declared as "double". The final result will be printed out by Process 0.

9. Programming Assignment (80 points)

EACH TEAM JUST TURNS IN ONE HARD COPY. Please use **MPI** to write a program for matrix multiplications. Go to the course web page to see how to run MPI program in our machines. The program should multiply two n by n matrices A and B in p computers and give results of matrix C . For simplicity and easier to test, each entry of A and B is a value 1.1 defined by Type *float*. You may write your program in a SPMD with master/slave model in which a master task (Process 0) sends data to slaves and slaves execute and return the results back. Please finish the following programs. In the first one, broadcast functions among the slaves are allowed, but in the second one, this function is not allowed.

- The master task distributes n/p **rows** of A and B to p slaves. Each slave task should compute the corresponding rows of matrix C . Each slave task can broadcast its local B rows to other slave tasks. This should be an easy program to write.
- Similar to the previous one, The master task distributes n/p **rows** of A and B to p slaves. Each slave task should compute the corresponding rows of matrix C . For this program, however, slave tasks can not use any broadcast functions. You should write the program based on the idea of the following sequential code:

```
for i = 1 to n do
  for j = 1 to n do
    for k = 1 to n do
      kk=((k-1+i-1) mod n) + 1;
      C[i,j] = C[i, j] + A[i, kk] *B[kk,j];
    endfor
```

Try to trace this sequential code and see if it produces the same results as the common one. Try to parallelize it and write a MPI program.

- Run your program on 1 computer, 2 computers, and 4 computers for the cases when n is 256. You may try to run more cases if you want, for example $n = 512$, etc. Because I don't want you to jam the network, I only put a small case when $n = 256$. If you feel the effect of the speed-up is not obvious, you may try larger one such as $n= 512$. Since all the entries of A and B are 1.1, the master does not need to allocate space for the entire A and B initially.

Before you try to run big case, you should have your program tested in a very small case such as $n= 4$ in one computer.

- Give me the performance data for each case such as total running time, speedup, or whatever you can think of. And also give me a discussion about the performance data you collected (this is probably the most important part in your report). Any discussion

about these two approaches? Under what condition, the second approach is better than the first one? Does the communication cost play a significant overhead here?

- This program can be done by a team. Each team just turn in ONE hard-copy of the program. Be honest. If your program does not work, SAY SO. TA will arbitrarily select some students to run their programs. If your program does not work but you "claim" yours works, you will have severe penalty! By the way, for program part, everyone in one team gets the same grade. The clarity of your programs and programming style will affect your grade.

- 2.5 Would you use spinning on a flag or blocking of processes for interprocess synchronization in uniprocessor operating systems? What do you think the trade-offs are between blocking and spinning on a multiprocessor? *slide 28*
- 2.6 In the shared address space parallel equation solver (Figure 2.13), why do we need the first and third barriers in a while loop iteration (lines 16a and 25f)? Can you eliminate them without inserting any other synchronization, perhaps altering when certain operations are performed? Think about all possible scenarios.
- 2.7 Gaussian elimination is a well-known technique for solving simultaneous linear systems of equations. Variables are eliminated one by one until there is only one left, and then the discovered values of variables are back-substituted to obtain the values of other variables. In practice, the coefficients of the unknowns in the equation system are represented as a matrix A , and the matrix is first converted to an upper-triangular matrix (a matrix in which all elements below the main diagonal are 0). Then back-substitution is used. Let us focus on the conversion to an upper-triangular matrix by successive variable elimination. Pseudocode for sequential Gaussian elimination is shown in Figure 2.18. The diagonal element for a particular iteration of the k loop is called the *pivot element*, and its row is called the *pivot row*.
- Draw a simple figure illustrating the dependences among matrix elements.
 - Assuming a decomposition into rows and an assignment into blocks of contiguous rows, write a shared address space parallel version using the primitives used for the equation solver in this chapter.
 - Write a message-passing version for the same decomposition and assignment, first using synchronous message passing and then any form of asynchronous message passing.
 - Can you see obvious performance problems with this partitioning? (We will discuss this further in the next chapter.)
 - Modify both the shared address space and message-passing versions to use an interleaved assignment of rows to processes.
 - Discuss the trade-offs (programming difficulty and any likely major performance differences) in programming the shared address space and message-passing versions.
- 2.8 Suppose that a system supporting a shared address space did not support barriers but only semaphores. Even global event synchronization would have to be constructed through semaphores or ordinary flags. The use of semaphores can be illustrated as follows. Suppose process P_2 has to indicate to process P_1 (using semaphores) that P_2 has reached a point b in the program so that P_1 can proceed past a point a (where it was waiting). P_1 performs a wait (also called P or down) operation on a semaphore when it reaches point a , and P_2 performs a signal (or V or up) operation on the same semaphore when it reaches point b . If P_1 gets to a before P_2 gets to b , P_1 suspends or blocks itself and is awakened by P_2 's signal operation.
- How might you orchestrate the synchronization in the shared address space parallel Gaussian elimination with (i) flags and (ii) semaphores replacing the

```

procedure Eliminate (A)           /*triangularize the matrix A*/
begin
  for k ← 0 to n-1 do           /*loop over all diagonal (pivot) elements*/
    begin
      for j ← k+1 to n-1 do     /*for all elements in the row of, and to the right of,
                                the pivot element*/
        Ak,j = Ak,j / Ak,k;    /*divide by pivot element*/
      Ak,k = 1;
      for i ← k+1 to n-1 do     /*for all rows below the pivot row*/
        for j ← k+1 to n-1 do  /*for all elements in the row*/
          Ai,j = Ai,j - Ai,k * Ak,j;
        endfor
        Ai,k = 0;
      endfor
    endfor
  endfor
end procedure

```

FIGURE 2.18 Pseudocode describing sequential Gaussian elimination

barriers? Could you use point-to-point or group event synchronization instead of global event synchronization?

b. Answer the same for the equation solver example.

2.9 In the straightforward, loop-based approach to parallelizing Gaussian elimination discussed so far, parallelism is exploited only within an iteration of the outermost, k , loop. Since the pivot element and its row (called the *pivot row*) are effectively broadcast directly to all processes that need it, this is called the *broadcast version*. Gaussian elimination can also be parallelized in a form that is more aggressive in exploiting the available concurrency, even across outer loop iterations. During the k th iteration, the process assigned the pivot row can simply pass the pivot row on to the next process instead of broadcasting it. This process can use the pivot row to update its assigned rows immediately, as well as pass it on to the next process, and so on. As soon as this process has done its computation for the k th iteration of that loop in the sequential program, it can immediately perform its pivot row computation for the $(k + 1)$ th iteration without waiting for all other processes to receive the k th row and perform their work for the k th iteration. It can then pass this $(k + 1)$ th row on to the next process as well, which can use it right away instead of waiting for the entire previous k loop iteration to complete. Multiple k loop iterations are in progress at once; rows are passed down the processor pipeline as soon as they are computed and are computed as soon as the rows needed have arrived through the pipeline. We call this the *pipelined* form of parallelization.

a. Write a shared address space pseudocode, at a similar level of detail as Figure 2.13, for a version that implements pipelined parallelism at the granularity of individual elements. Show all synchronization necessary. Do you need barriers?