

Show Your Work! Point values are in square brackets. Point values are in square brackets. There are 25 points altogether.

1. Suppose that $A = (a_{ij})$ is an $n \times n$ matrix:

$$A = \begin{pmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{pmatrix}.$$

Recollect that the *transpose* of A , denoted A^T , is the matrix obtained from A by interchanging its rows and columns. So

$$A^T = \begin{pmatrix} a_{00} & a_{10} & \cdots & a_{n-1,0} \\ a_{01} & a_{11} & \cdots & a_{n-1,1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{0,n-1} & a_{1,n-1} & \cdots & a_{n-1,n-1} \end{pmatrix}.$$

If the one-dimensional array A stores the matrix A in row-major order (first row in first n elements, second row in next n elements, etc.) then the following serial code can be used to store A^T in the one-dimensional array AT .

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        AT[j*n + i] = A[i*n + j];
```

Write a parallel Pthreads function that takes as input a square matrix stored as a 1-dimensional array and returns the transpose also stored as a 1-dimensional array. The original matrix A , its transpose AT , and the order of the matrices n are stored as global variables. You can assume that n is evenly divisible by p , the number of threads. [4 points]

```
void* Transpose(void* rank) {
    long my_rank = (long) rank;
    int i, j;
    int my_first_row = my_rank*n/thread_count;
    int my_last_row = my_first_row + n/thread_count - 1;

    for (i = my_first_row; i <= my_last_row; i++)
        for (j = 0; j < n; j++)
            AT[j*n + i] = A[i*n + j];

    return NULL;
} /* Transpose */
```

2. Sally has written a Pthreads program that has the following efficiencies:

p	Input Size				
	1000	2000	4000	8000	16,000
1	1.0	1.00	1.0	1.0	1.0
2	0.9	0.9	0.9	0.9	1.0
4	0.8	0.9	0.9	0.9	0.9
8	0.8	0.8	0.8	0.8	0.9

On the basis of these data, is Sally's program

- (a) Strongly scalable?
- (b) Weakly scalable?
- (c) Scalable?

[3 points]

- (a) For a program to be strongly scalable it be the case t that the program has constant or improving efficiency as the number of processes/threads is increased *and* the problem size remains constant. Clearly, this is not the case. For all input sizes shown, the efficiency decreases as the number of threads is increased.
- (b) For a program to be weakly scalable it must be the case that it has constant (or improving) efficiency as the number of processes/threads is increased at a given rate *and* the problem size increases at the same rate. This is also not the case. For example, when the problem size is 2000 and there are 4 threads, the efficiency is 0.9, but when we double the problem size to 4000 and double the number of threads to 8, the efficiency decreases to 0.8
- (c) For a program to be scalable it must be the case that as we increase the number of processes/threads at a given rate, we can find a corresponding rate of increase in the problem size so that the efficiency is either constant or increasing. On the basis of the available data this program is scalable: if we double the number of threads and we multiply the problem size by 8, the efficiency remains constant.

Note that we don't include $p = 1$ in any of the scalability definitions: if we did, the only scalable program would have efficiency 1 for all values of p and n . Also note that we can only make statements about Sally's program on the basis of the available data. So the scalability of the program is fairly suspect, since there's so little data.

3. (a) What is one advantage of transactional memory over traditional, lock-based shared-memory programming? [1]
(b) What is one limitation of transactional memory? [1]
(a) There are several possible answers here. One possibility is that TM shifts the burden of insuring correct thread access to shared data structures from the programmer to the TM system.
(b) There are also several possible answers here. One possibility is that TM by itself doesn't provide a good solution to producer-consumer synchronization.

4. On page 29 of “The Landscape of Parallel Computing Research”, the authors write

...if the L1 data cache uses write through to an L2 cache with write back, then the L1 data cache needs only parity [a cheap form of error correction] while the L2 cache needs SEC/DED [a more expensive form of error correction].

Explain why write through cache needs only weak error correction while write back requires more expensive error correction. [2]

With write through cache, there should always be a backup copy of the data: the copy in main memory (or in this case L2 cache). So if an error is detected in a cached copy, the backup copy can be retrieved. With write back cache, the cached copy may be the only copy of the data available to the program. So in addition to determining that there's been an error, the system should be able to reconstruct a correct copy of the data.

5. On page 37 of “The Landscape of Parallel Computing Research”, the authors write

Indeed, most scalable parallel codes have all data layout, data movement, and processor synchronization manually orchestrated by the programmer. Such low-level coding is labor intensive, and usually not portable to different hardware platforms or even to later implementations of the same instruction set architecture.

Most scalable parallel codes are written in C or Fortran with MPI. Are such codes portable — i.e., can they be taken from one system and run on another? Explain your answer. [2]

There can be little doubt that a C-MPI code is portable to virtually any architecture — including from distributed memory to shared memory and vice-versa. However, there *may* be problems with the performance of a C-MPI code taken from one architecture to another. For example, a code that is scalable on an architecture with a high-performance interconnect, may no longer be scalable on an architecture with a much slower interconnect. So this question could be answered either “yes” or “no.” Correctness will depend on the explanation.

6. Section 2.2.1 (p. 5) of the Jostle paper discusses using a maximal independent subset of graph vertices to create a coarser graph.

- (a) For the graph on the board, find a maximal independent subset of vertices. (This should be *edges*, not vertices.) [1]
- (b) If each vertex and each edge in the graph has weight one, sketch the coarsened graph that results from contracting the edges in the maximal independent subset. Be sure to indicate vertex and edge weights in the coarsened graph. [1]

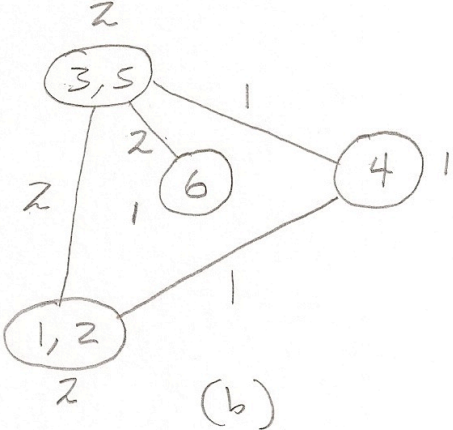
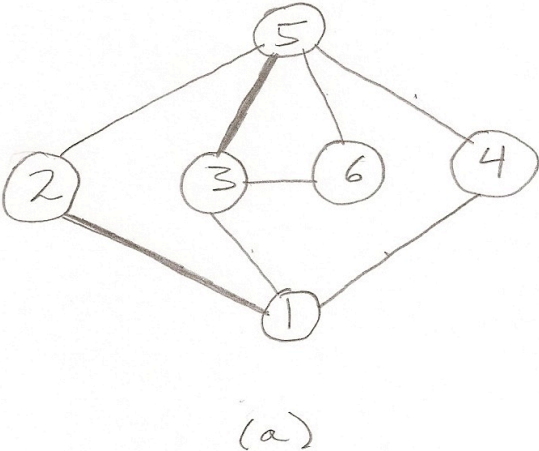


Figure 1: (a) A maximal independent subset of edges (darker edges). (b) Coarsened graph.

7. Determine the output of the following MPI program if it is run with four processes. [4]

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char* argv[]) {
    int      my_rank, p, source, dest, i, q, n = 3;
    int      list[3], tlist[3];
    MPI_Comm comm;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    comm = MPI_COMM_WORLD;
    MPI_Comm_size(comm, &p);
    MPI_Comm_rank(comm, &my_rank);

    source = (p + my_rank - 1) % p;
    dest = (my_rank + 1) % p;
    for (i = 0; i < n; i++)
        list[i] = tlist[i] = my_rank + i;

    for (q = 1; q < p; q++) {
        /* Assume adequate buffering available */
        MPI_Send(tlist, n, MPI_INT, dest, 0, comm);
        MPI_Recv(tlist, n, MPI_INT, source, 0, comm, &status);
        for (i = 0; i < n; i++)
            list[i] += tlist[i];
    }

    if (my_rank == 0) {
        for (i = 0; i < n; i++)
            printf("%d ", list[i]);
        printf("\n");
    }
    MPI_Finalize();
    return 0;
} /* main */
```

(Use the following page if you need more space.)

(Solution to number 7.)

Here's a trace of the values of the variables:

my_rank	p	src	dest	q	list	tlist
0	4	3	1		0 1 2	0 1 2
				1	3 5 7	3 4 5
				2	5 8 11	2 3 4
				3	6 10 14	1 2 3
1	4	0	2		1 2 3	1 2 3
				1	1 3 5	0 1 2
				2	4 7 10	3 4 5
				3	6 10 14	2 3 4
2	4	1	3		2 3 4	2 3 4
				1	3 5 7	1 2 3
				2	3 6 9	0 1 2
				3	6 10 14	3 4 5
3	4	2	0		3 4 5	3 4 5
				1	5 7 9	2 3 4
				2	6 9 12	1 2 3
				3	6 10 14	0 1 2

So the output is

6 10 14

8. Lemuel has written a parallel MPI function `Lems_func`. He wants to study its performance. So he adds the following code to his MPI program:

```
double start, finish, elapsed;
. . .
start = MPI_Wtime();
/* All processes call Lems_func */
Lems_func(. . .);
finish = MPI_Wtime();
elapsed = finish-start;
if (my_rank == 0)
    printf("Lems_func took %e seconds\n", elapsed);
```

- (a) The time reported by this code can be misleading. Explain why. [2]
- (b) Modify the above code so that it reports a more reliable time for `Lems_func`. [2]
- (a) When we report parallel run time, we want to know the amount of time that has elapsed from when the first process begins execution to when the last process finishes execution. This only reports the elapsed time on process 0. For example, if there are two processes, it's entirely possible that process 1 is still executing `Lems_func` when process 0 prints the elapsed time.
- (b) Lem should use something like this:

```
double my_start, my_finish, my_elapsed, elapsed;
. . .
MPI_Barrier(comm);
my_start = MPI_Wtime();
/* All processes call Lems_func */
Lems_func(. . .);
my_finish = MPI_Wtime();
my_elapsed = my_finish-my_start;
MPI_Reduce(&my_elapsed, &elapsed, 1, MPI_DOUBLE, MPI_MAX, 0, comm);
if (my_rank == 0)
    printf("Lems_func took %e seconds\n", elapsed);
```

9. The matrix A has p rows and p columns ($p > 1$). Its elements are `doubles`, and it is stored as a one-dimensional array in row-major order on process 0.

- (a) Write MPI code that builds a derived datatype that can be used to represent one *column* of A . (Syntax for some derived datatype functions is on the following page.)[1]
- (b) Use your derived datatype in MPI code that sends the last column of A from process 0 to process $p - 1$. Process $p - 1$ should receive the column into p contiguous memory locations. [1]

```
(a)      MPI_Datatype vect_mpi_t;
        . . .
        MPI_Type_vector(p, 1, p, MPI_DOUBLE, &vect_mpi_t);
        MPI_Type_commit(&vect_mpi_t);

(b)      if (my_rank == 0)
            MPI_Send(A+p-1, 1, vect_mpi_t, p-1, 0, comm);
        else if (my_rank == p-1) {
            col = malloc(p*sizeof(double));
            MPI_Recv(col, p, MPI_DOUBLE, 0, 0, comm, &status);
        }
```

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
    MPI_Datatype *newtype)
    Creates a contiguous datatype

int MPI_Type_vector(int count, int blocklength, int stride,
    MPI_Datatype oldtype, MPI_Datatype *newtype)
    Creates a vector (strided) datatype

int MPI_Type_indexed(int count, int *array_of_blocklengths,
    int *array_of_displacements, MPI_Datatype oldtype,
    MPI_Datatype *newtype)
    Creates an indexed datatype

int MPI_Type_struct(int count, int *array_of_blocklengths,
    MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types,
    MPI_Datatype *newtype)
    Creates a struct data type

int MPI_Type_commit(MPI_Datatype *datatype)
    Commits a data type for use in communication functions

int MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint *lb,
    MPI_Aint *extent)
    Returns the lower bound and extent of a data
    type

int MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb,
    MPI_Aint extent, MPI_Datatype *newtype)
    Returns a new data type with new extent and
    upper and lower bounds
```