Practice problems

2023

Question 1:

State if the following statements are True or False. No explanations are necessary.

1. MPI_Comm_split needs to be called by only those processes which desire to be a part of some new sub-communicator created by the call.

2. If an MPI program is executed on a single processor machine using multiple processes, the processes can communicate with each other using global variables.

3. It is possible to broadcast a message to a subset of all processes using MPI_Bcast by using a subcommunicator.

4. When MPI_Isend is used, corresponding MPI_Wait/MPI_Test must be used to determine whether the operation has completed and the communication buffer is safe for reuse.

5. Compulsory misses can be avoided with prefetching.

6. For a fully associative cache, when a cache line is brought in from the memory, it can go to any cache frame that is empty. As a result, all cache misses that could happen for a fully associative cache are capacity misses.

7. False sharing can cause incorrect results

8. Register renaming eliminates all dependencies

9. An MPI program where all communications are collective cannot deadlock

10. In shared memory of a GPU, consecutive words are stored in adjacent banks

11. If threads of a GPU warp access words from the same block of 32 words, their memory requests are clubbed into one.

Question 2:

For the following questions, circle correct answer(s). No explanations are necessary.

1. Process 0 has a set of p unique IDs which it needs to distribute to all the p processes in its communicator including itself. Which call is best suited for this task?

   (a) MPI_Bcast
   (b) MPI_Send
   (c) MPI_Scatter
   (d) MPI_Allgather

2. A matrix of size NXN is multiplied by a vector of size $NX1$. The total sequential work in this computation is

   (a) $O(N)$
   (b) $O(N^2)$
   (c) $O(N^3)$
   (d) $O(N \log N)$

3. Which function is better used in the situation when a process has something else to do besides waiting for a non-blocking MPI communication operation to complete? (i.e. overlapping communication and computation)

   (a) MPI_Wait
   (b) MPI_Test (It returns immediately instead of blocking if the message hasn't arrived yet)

4. Which of the following is not an MPI collective communication operation (i.e. a group of processes using the same communicator to communicate and synchronize)?

   (a) MPI_Reduce
   (b) MPI_Bcast
   (c) MPI_Waitall
   (d) MPI_Barrier

5. Which MPI collective operation would you use to compute the transpose of a matrix that is distributed in a row-wise manner across P processes?

   (a) MPI_Allreduce
   (b) MPI_Alltoall
   (c) MPI_Scan
   (d) MPI_Reduce

Question 3:

Figure 1 illustrates the Compressed Sparse Row (CSR) format for sparse matrices.

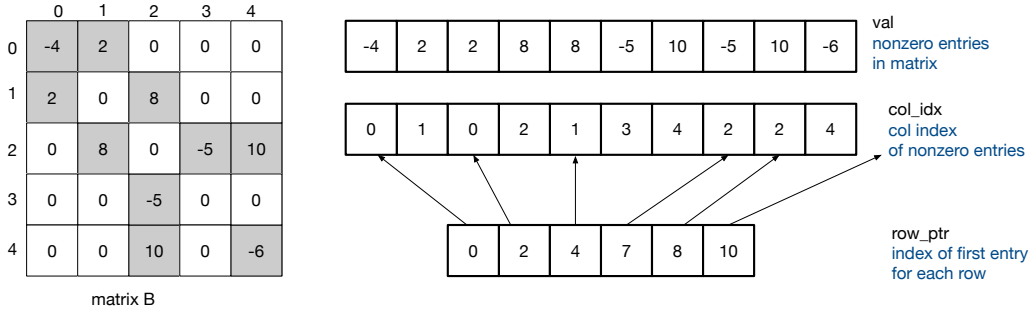

Figure 1: CSR Matrix representation

The C code below computes the product of an `M×N` sparse matrix `B` with `NNZ` non-zero entries stored in CSR format, with a dense vector `c` of length `N`. The result is stored in vector `a` of length `M`.

```
float val[NNZ], a[M], c[N];
int row_ptr[M+1], col_idx[NNZ];

for(int i=0;i<M; i++)
 for(int j=row_ptr[i]; j<row_ptr[i+1]; j++)
   a[i] += val[j]*c[col_idx[j]];
```

Q3a:

You have below 3 OpenMP versions of the SpMV computation

Version 1:

```
#pragma omp parallel for
for(int i=0;i<M; i++)
  for(int j=row_ptr[i]; j<row_ptr[i+1]; j++)
    a[i] += val[j]*c[col_idx[j]];
```

Version 2:

```
for(int i=0;i<M; i++)
  #pragma omp parallel for reduction(+:a[i])
  for(int j=row_ptr[i]; j<row_ptr[i+1]; j++)
    a[i] += val[j]*c[col_idx[j]];
```

Version 3:

```
#pragma omp parallel for collapse(2) reduction(+:a)
for(int i=0;i<M; i++)
  for(int j=row_ptr[i]; j<row_ptr[i+1]; j++)
    a[i] += val[j]*c[col_idx[j]];
```

Is each of these 3 versions correct? If not, why?

Which of the correct versions you expect will perform better? Why?

Q3b:

For those versions above that you determined to be correct, add the `schedule` clause that you believe will give you best performance in the general case (nothing is known on the structure of matrix `B`, except its dimensions and the number of non-zeroes.)

Explain your choice

Q3c:

Modify the code so that the outer loop is executed sequentially, but each instance of the inner loop is dispatched for execution on a GPU.

Question 4:

The program below finds the index of the largest entry in a distributed array containing positive integers. In case of ties, it returns the index of the first largest entry. The result is returned on process 0. Please complete the missing MPI calls.

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

#define N 1000

int a[N]; /* local portion of distributed array */
int rank, size;

int maxloc(void) {
    int i, index, gmax, maxindex;
    int max = -1;
    /* find local maximum */
    for(i=0;i<N;i++)
    if(a[i]>max) {
     index=i;
     max=a[i];
     }

    /* find global maximum */
    MPI___;



    if (gmax==max)
     index= N*rank+index;
    else
     index=N*size;

    /* find index of first occurence of global maximum */
    MPI___;



    if (rank==0)
     return maxindex;
    else
    return -1;
```

```
}

int main() {
    int result;
    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    ... /* initialize a */
    result = maxloc();
    if (rank == 0)
     printf("%d\n", result);
    MPI_Finalize();
}
```

Question 5:

A *band matrix* is a square matrix where all nonzero elements are in a band around the diagonal. The matrix has *bandwidth* B if the band has width $2B+1$, i.e., $a_{ij} = 0$ if $|i-j| > B$. We illustrate below a 7×7 (pentadiagonal) matrix with bandwidth 2.

$$\begin{pmatrix} 2 & 3 & 1 & 0 & 0 & 0 & 0 \\ 1 & 3 & 5 & 2 & 0 & 0 & 0 \\ 2 & 1 & 4 & 6 & 3 & 0 & 0 \\ 0 & 3 & 7 & 1 & 3 & 2 & 0 \\ 0 & 0 & 4 & 5 & 1 & 2 & 6 \\ 0 & 0 & 0 & 2 & 3 & 7 & 8 \\ 0 & 0 & 0 & 0 & 1 & 2 & 1 \end{pmatrix}$$

The code below is an MPI version of the banded matrix times vector computation. The inputs `a` and `b` are initially at process 0, and the output `c` is returned at process 0. We assume that the number N of rows evenly divides by the number of processes. Please complete the missing arguments for the three collective calls.

```
...
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
int rows = N/size;
if (N%size != 0) exit(0);
double aa[rows][N];
double cc[rows];
/* COMPLETE CALL BELOW TO SCATTER a*/
MPI_Scatter(...);



/* COMPLETE CALL BELOW TO BROADCAST  b */
MPI_Bcast(...);


for(i=0; i<rows; i++) {
  int ii = rows*myrank+i;
  jfirst = (ii>B) ? ii-B : 0;
  jlast = (ii+B < N-1) ? ii+B : N-1;
  cc[i] = 0;
  for (j=jfirst; j<=jlast; j++)
    cc[i] += aa[i][j]*b[j];
}
/* COMPLETE CALL BELOW TO GATHER c */
```

```
MPI_Gather(...);



}
```

Question 6:

Which of the following programs has a data race?

```
int sum = 0;
#pragma omp parallel
{
    sum = omp_get_thread_num();
    ++sum;
}
```

```
#pragma omp parallel
{
    int sum = 0;
    ++sum;
}
```

```
int sum = 5;
++sum;
#pragma omp parallel
{
    int a = sum;
}
```

```
#pragma omp parallel
{
    int sum = 0;
    sum = omp_get_thread_num();
    ++sum;
}
```

Question 7

What is the maximum length vector operation which can be applied to the following loop? Assume the largest vector register in 512 bits, an int is 32 bits, and a and b have N elements. Give your answer in terms of the number of ints that a vector operation can be applied to.

```
int *a, *b;
int N = 10000;
...
for (int i = 16; i < N; ++i) {
        a[i] = a[i-4] + b[i-10];
}
```

MPI

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)

int MPI_Comm_size(MPI_Comm comm, int *size)

int MPI_Send(const void *buf, int count, MPI_Datatype
   datatype,
   int dest, int tag, MPI_Comm comm)

int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
   int source, int tag, MPI_Comm comm, MPI_Status *status)

int MPI_Isend(const void *buf, int count, MPI_Datatype
   datatype,
   int dest, int tag, MPI_Comm comm, MPI_Request *request)

int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
   int source, int tag, MPI_Comm comm, MPI_Request *
      request)

int MPI_Wait(MPI_Request *request, MPI_Status *status)

int MPI_Waitall(int count, MPI_Request array_of_requests
   [],
   MPI_Status *array_of_statuses)

int MPI_Barrier(MPI_Comm comm);

int MPI_Scatter(const void *sendbuf, int sendcount, MPI_
   Datatype sendtype void *recvbuf, int recvcount, MPI_
   Datatype recvtype, int root, MPI_Comm comm);

int MPI_Scatterv(const void *sendbuf, const int sendcounts
   [], const int displs[], MPI_Datatype sendtype, void *
   recvbuf, int recvcount, MPI_Datatype recvtype, int root
   , MPI_Comm comm);

int MPI_Gather(const void *sendbuf, int sendcount, MPI_
   Datatype sendtype, void *recvbuf, int recvcount, MPI_
   Datatype recvtype, int root, MPI_Comm comm);
```

```
int MPI_Gatherv(const void *sendbuf, int sendcount, MPI_
    Datatype sendtype, void *recvbuf, const int recvcounts
    [], const int displs[], MPI_Datatype recvtype, int root
    , MPI_Comm comm);

int MPI_Bcast(void *buffer, int count, MPI_Datatype
    datatype, int root, MPI_Comm comm);

int MPI_Cart_create(MPI_Comm old_comm, int ndims, const
    int dims[], const int periods[], int reorder, MPI_Comm
    *comm_cart);

int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims,
    int coords[]);

int MPI_Cart_shift(MPI_Comm comm, int direction, int disp,
     int *rank_source, int *rank_dest);

int MPI_Win_create(void *base, MPI_Aint size, int disp_
    unit, MPI_Info info, MPI_Comm comm, MPI_Win *win);

int MPI_Get(void *origin_addr, int origin_count, MPI_
    Datatype origin_datatype, int target_rank, MPI_Aint
    target_disp, int target_count, MPI_Datatype target_
    datatype, MPI_Win win);
```

OpenMP

#pragma omp parallel for *[clause[[,]clause] ...]*
for-loop

clause:
private(*list*)
firstprivate(*list*)
lastprivate(*list*)
reduction(*reduction-identifier : list*)
schedule(***kind****[,* ***chunk_size****]*)
collapse(*n*)

kind:
static
dynamic
guided

#pragma omp target data *clause[[[,]clause] ...]*
structured-block

clause:
map(*[map-type:] list)*

map-type
alloc
to
from
tofrom

#pragma omp target enter data *clause[[[,]clause] ...]*

clause:
map(*[map-type:] list)*

#pragma omp target exit data *clause[[[,]clause] ...]*

clause:
map(*[map-type:] list)*

#pragma omp target update *clause[[[,]clause] ...]*

clause:
to(*list*)
from(*list*)

#pragma omp target *[clause[[,]clause] ...]*
structured-block

clause:
private(*list*)
firstprivate(*list*)
map(*map-type:] list*)