Uppsala University
Department of Information Technology
Scientific Computing

Programming of Parallel Computers, 10hp, 2013-03-18

Time: $08^{00} - 13^{00}$
Help: None
Each of the six problems below can give up to five points. For maximum points, you must give detailed answers and motivate your assumptions. Grade 3: 12p, Grade 4: 18p, Grade 5: 24p

1. a) Explain how the standard *MPI_SEND* and *MPI_RECV* works in MPI, i.e., sketch the data transfer model.
   b) When you use *MPI_ISEND* do you need to wait (*MPI_WAIT*), why?
   c) If you need to implement the function *MPI_ALLREDUCE* only with *MPI_SEND* and *MPI_RECV*, how many send and receive calls will you need for N processors? Draw a sketch of the communication pattern.

2. a) What are the main restrictions in GPU/CUDA programming model for many algorithms?
   b) What characteristics should the algorithms have to be efficiently implemented on a GPU device?

3. When programming global address space computers with a thread model there are no explicit communication calls as in MPI. Instead the communication is handled implicitly due to cache misses. We can then have four types of cache misses; cold/compulsory, capacity, true sharing and false sharing misses. Explain what these types of misses are and when they happen.

4. Consider the following three pseudo code functions:

```
fct1(struct Data) {

Data.level = Data.level + 1 ;

if (Data.level < MAX)
 NewThread( fct1(Data.A) );
else
 fct1(Data.A);

fct1(Data.B);

}
```

```
fct2(struct Data) {

Data.level = Data.level + 1 ;

fct2(Data.B);

if (Data.level < MAX)
 NewThread( fct2(Data.A) );
else
 fct2(Data.A);

}


fct3(struct Data) {

Data.level = Data.level + 1 ;

if (Data.level < MAX) {
 NewThread( fct3(Data.A) );
 NewThread( fct3(Data.B) );
} else {
 fct3(Data.A)
 fct3(Data.B)
}

}
```

Answer the questions:

a) What is the formula giving the relation between the number of threads and recursion levels in each of the functions above?

b) If you need to implement QuickSort algorithm based on divide and conquer - which approach would you select? Why? Which approach is the fastest? Why?

5. Matrix-Matrix multiplication can be formulated as (Strassen, 1969), rewriting $C = AB$ on block form

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

and then computing (7 multiplies, 18 adds)

$$
\begin{aligned}
P_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) & C_{11} &= P_1 + P_4 - P_5 + P_7 \\
P_2 &= (A_{21} + A_{22})B_{11} & C_{12} &= P_3 + P_5 \\
P_3 &= A_{11}(B_{12} - B_{22}) & C_{21} &= P_2 + P_4 \\
P_4 &= A_{22}(B_{21} - B_{11}) & C_{22} &= P_1 + P_3 - P_2 + P_6 \\
P_5 &= (A_{11} + A_{12})B_{22} & & \\
P_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) & & \\
P_7 &= (A_{12} - A_{22})(B_{21} + B_{22}) & &
\end{aligned}
$$

Moreover, the 7 block-matrix-matrix multiplies can be done recursively with the same algorithm until the block size is small enough, e.g. 32 by 32 elements. The final matrix blocks are multiplied with standard matrix-matrix multiplication. Sketch a recursive version of Strassen's algorithm and parallelize it with OpenMP using the *task-directive*. You can assume that you have the functions:

```
void mult(double *C, double *A, double *B, int n);
void add(double *C, double *A, double *B, int n);
void sub(double *C, double *A, double *B, int n);
```

for standard matrix multiplication ($C = A \times B$), addition ($C = A + B$) and subtraction ($C = A - B$) of nxn matrices available. Take special care on how you do the mallocs and explain why this is important.

6. The bucketsort algorithm can be described as:

   1. Define k number of buckets in the interval [min,max] and filter the elements into the k buckets, i.e., all elements in the interval [bstart(j),bstop(j)] are placed in bucket j, for j=1 to k.
   2. Sort the buckets independently with, e.g., the quicksort algorithm.

   Assume that the data we want to sort comes from a normal distribution. Now we want to implement the algorithm in C and parallelize it with MPI. For your disposal you have the serial function `init(double *data,int start,int stop)` which initializes the data sequence that we want to sort in the interval from start to stop. The generation of one data element is independent of other data points, i.e, calling `init(data,1,len)` and `init(&data[0],1,len/2); init(&data[len/2],len/2+1,len)` is equivalent. You also have the function `quicksort(double *data,int left,int right)` which sorts the data in place.

   Sketch a memory efficient parallel implementation of the bucketsort algorithm in C using MPI. Explain what the parallel overheads are in your implementation and how you could minimize these overheads.

Good Luck!