Uppsala University
Department of Information Technology
Scientific Computing

Programming of Parallel Computers, 10hp, 2014-03-17

Time: $8^{00} - 13^{00}$
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) Linear algebra operations often consist of three types of loops which can be nested. What are these types, explain the three types with examples. When we know these types we can use this knowledge for efficient parallelization of the more complex algorithms, explain how.

   b) The solution algorithm for (upper) triangular systems of equations, $Ux = b$, can be written as:

Solution algorithm 1

```
for j=n to 1
   x(j)=b(j)/U(j,j)
   for i=j-1 to 1
      b(i)=b(i)-x(j)*U(i,j)
   end for
end for
```

An alternative way to solve the system can be written as:

Solution algorithm 2

```
for j=n to 1
   for i=j+1 to n
      b(j)=b(j)-x(i)*U(j,i)
   end for
   x(j)=b(j)/U(j,j)
end for
```

Analyze the loops according the types above in the respective algorithm, i.e., what types of loops do we have? Assume that we want to parallelize the solution algorithm for triangular systems of equations, $Ux = b$, with OpenMP. Which algorithm should we choose and why? Discuss what factors will limit the performance of the parallelization in respective case?

2. a) The Quick-sort algorithm can be parallelized in many ways. Here we will consider two inherently different ways. The first way is a *divide-and-conquer parallelization* where we acquire a new thread for each recursion step and sort the two lists in parallel on the two threads. After a number of recursion levels we proceed without requiring new threads using the serial Quick-sort algorithm on the respective thread. The other way is a *peer parallelization* where we create p threads as peers, divide the data equally assigning n/p elements to each thread, and sort the elements internally within each thread using the serial Quick-Sort algorithm. Then, in p phases alternating with your left and your right neighbor, merge data and keep either the right or the left part of the data. One can argue that the divide-and-conquer parallelization has the best performance when the number of threads is considerably larger than the number of cores while the peer parallelization has an optimal performance for twice as many threads as cores. Explain why we have these differences.

   b) When doing the divide-and-conquer parallelization one has to be careful of how to create and terminate the threads otherwise we can get subtle bugs that only shows occasionally. In the following code sequence we have such a bug, what is the bug, what can happen (explain the behavior of the bug), and how can we correct it?

```
void *pquick(void *arg){
   ...
   if (level<maxlevel){
      ...
      pthread_create(&new_thread,NULL,pquick,(void*)&data_left);
      pquick((void*)&data_right);
      pthread_join(new_thread);
   }
   else
      serial_quick_sort(array,left,right);
   pthread_exit(NULL);
}
```

3. Consider the Bucket-sort algorithm, here the elements are first filtered into buckets (i.e. elements within range $[a_i\ b_i]$ belong to bucket i). Then the buckets can be sorted concurrently and independently in different threads, e.g., using the Quick-sort algorithm. The algorithm is straightforward to parallelize as the sorting of the buckets is trivially parallel. The problem becomes to get a good load balance among the processors and cores. Assume that we want to use OpenMP for the parallelization, how can we then do the load balancing of the buckets to the threads? Describe at least four *conceptually different* ways in OpenMP to get a good load balance and discuss their advantages and disadvantages.

4. a) When communicating point-to-point in MPI the communication time can be modeled with a linear model with respect to the message length. Explain what the parameters in the model are and what they depend on.

   b) The parameters in the model can be estimated with a simple *ping-pong test*, i.e., sending messages back and forth between two processors. Write a MPI program, with appropriate MPI commands, that performs a ping-pong test. The program itself does not have to estimate the parameters but produce data from which it is possible to later estimate the parameters.

5. a) Could we expect always to have speed-ups when we porting our code to the GPU (assuming the code is fully parallel)? If not, explain why.

   b) Explain the concepts of *thread, block* and *grid* in CUDA. How many threads CUDA will schedule for test1, test2, test3 and test4?

```
int n = 44;
int bs = 22;
dim3 dimBlock( bs );
dim3 dimGrid( n/bs );

test1<<<dimGrid, dimBlock>>>(x, y);
```

```
int n = 44+1;
int bs = 22;
dim3 dimBlock( bs );
dim3 dimGrid( n/bs );

test2<<<dimGrid, dimBlock>>>(x, y);
```

```
int n = 44;
int bs = 22+1;
dim3 dimBlock( bs );
dim3 dimGrid( n/bs );

test3<<<dimGrid, dimBlock>>>(x, y);
```

```
int n = 44;
int bs = 22;
dim3 dimBlock( bs );
dim3 dimGrid( n/bs + 1);

test4<<<dimGrid, dimBlock>>>(x, y);
```

6. To compute the determinant of a matrix of size n one can use the cofactor method. Here, define the submatrix A_{ij} of order (n-1)x(n-1) by deleting the i^{th} row and the j^{th} column of A. The determinant of A can then be computed as $detA = \sum_{i=1}^{n}(-1)^{i+j} \cdot a_{ij} \cdot detA_{ij}$ using development by j^{th} column where a_{ij} is the element in the matrix A with index i and j. The column j can be chosen arbitrary (e.g. j=1). Note, the determinant A_{ij} can be computed with the same method making the algorithm recursive (by using that the determinant of a scalar is the scalar itself). Choose an appropriate parallelization model (MPI, Pthreads or OpenMP) and write a parallel code for this algorithm. Make the parallelization as general as possible, i.e., think about cases where the number of cores is larger than n and where n is larger than the number of cores. You can assume that you have a function that extracts the submatrix A_{ij} by deleting the i^{th} row and the j^{th} column of A, i.e., `Anew=ExtractMat(A,i,j);`

Good Luck!