Universitetet i Bergen
Det matematisk-naturvitenskaplige fakultet

Exam in INF236 - Parallel programming, Spring 2023

Problem 1a
Explain briefly what is meant by *temporal* and *spatial* locality and explain how one can exploit both of these concepts when writing sequential code for matrix multiplication.

Answer
Temporal locality refers to the tendency of programs to reuse data elements that have recently been used. Thus recently used data elements are stored high up in the cache hierarchy.
Spatial locality refers to the tendency of programs to use data elements that are stored close in memory to recently used elements. Thus the system loads not only an element that is to be used, but also elements that are stored close by (i.e. in the same cache line) and stores these high up in the cache hierarchy.
In standard matrix multiplication, AxB, spatial locality is exploited by loading cache lines of A and B into cache. Unless B has been transposed this only helps the access to A. Temporal locality only has an effect if A (or the transposed B) is sufficiently small that an entire row can be stored in memory.
To exploit both types of locality multiplication should be done by dividing A and B into smaller blocks. If the elements of each block is stored consecutively then two blocks of sufficiently small size n^2 can be stored in the cache hierarchy and be used to perform n^3 operations.

Problem 1b
Explain briefly what is meant by *false sharing* and how one can try to avoid this when writing parallel code for updating the elements of a one dimensional array.

Answer
False sharing refers to when different threads are accessing elements on the same cache line. As the smallest unit of memory is a cache line this means that the same cache line will be moved between the caches of the different threads even though they are not writing to the same elements. This can be prevented by partitioning the data so that the cache lines containing data that a thread is writing to does not contain elements that other threads are writing to. A static partitioning will typically achieve this. It is also possible to pad an array with dummy elements to ensure that there is no overlap (measured in cache lines) between the working area of two threads.

Problem 1c
Explain briefly what is meant by *speedup* and how it is calculated, both when working in a theoretical setting and also in a practical setting.

Answer
Speedup is a measure of how much faster a parallel program is compared to a sequential one for solving the same problem. Speedup is a function of the number of threads used and also the size of the problem. It is calculated as $S(n,p) = T(n)/T_p(n)$ where n is the size of the problem and p the number of threads. $T(n)$ is here the time of the fastest sequential program for this problem on an instance of size n and $T_p(n)$ the time of the parallel program while using p threads on the same instance.

In a theoretical setting one would use the running time of the programs measured with O-notation (or a more exact expression). In a practical setting one would use the wall-clock time to calculate the speedup.

Problem 1d
Explain briefly what is meant by *latency* and *bandwidth* when writing code for parallel computers with distributed memory.

Answer
When sending data from between two processes latency is the time from when the first data item is sent until it starts arriving at its destination. Bandwidth is the capacity of the communication channel used. This measures how much data (in bytes) can be sent per time unit once the communication channel is filled.

Problem 2a
Consider an array $a = (a_0, \dots, a_{n-1})$ to be sorted. Enumeration sort counts the number of elements smaller than a_i in order to find the position of a_i in a sorted version b of a. Assuming that the array elements are distinct, a sequential code for computing b can be written as follows:

```
int i,j,pos;
for(i=0; i<n; i++) {
  pos=0;
  for(j=0; j<n; j++)
    if (a[j]<a[i]) pos++;
  b[pos] = a[i];
}
```

What OpenMP directives would you insert to parallelise the outermost for-loop (the i-loop)?

Answer
#pragma omp parallel for private(j,pos)

Note that the "parallel" is optional, one could also include i in the private() clause, but this does not make any difference as it is private by default. Scheduling should (probably) be static which is also the default.

One could also solve this by first including the i-loop in a parallel region, this way one would not have to restart the parallel section every time one starts the j-loop.

Problem 2a
What OpenMP directives would you insert to parallelise the innermost for-loop (the j-loop)?

Answer
#pragma omp parallel for reduction(+ : pos)

Again, "parallel" is optional and so is setting j as private. Scheduling should remain as static.

Problem 2c
Derive the expressions for both the parallel running times and the speedups for your solutions to problems 2a and 2b when using p threads, $1 \le p \le n$.

Answer
The sequential running time is $O(n^2)$. For 2a the parallel running time is $O(n^2/p)$ and for 2b the running time is $O(n^2/p + n\log p)$ or $O(n^2/p + np)$ (depending on how the reduction is done) when

using *p* threads. Speedup then becomes for 2a: $O(n^2/(n^2/p)) = O(p)$ and for 2b: $O(n^2/(n^2/p + n\log p)) = O(p/(1+p/n \log p)$. The expression for 2b could also be $O(p/(1+p^2/n)$ if a linear reduction is used.

Problem 2d
In the following we assume that it is possible to use nested parallelism in OpenMP, that is, to simultaneously parallelise two loops nested within each other.

Consider again the code given in problem 1a. Assume you are using *s* threads to parallelise the i-loop, at the same time as you are using *t* threads to parallelise the j-loop, where $1 \le s, t \le n$. Derive the parallel running time of the program as a function of *n*, *s* and *t*.

Answer
Each thread will run *n/s* iterations of the outer loop. Each of these iterations takes $n/t + \log t$ time (*or* $n/t + t$) for executing the inner loop. The total running time is then $n/s * (n/t + \log t) = n^2/(s*t) + n/s*\log t$ (*or* $n^2/(s*t) + n*t/s$).

Problem 3
Write an efficient parallel program using OpenMP and C or C++ for the following problem.

Given a 2-dimensional array A containing *n x n* elements of type double. Write a parallel program using OpenMP to set each element A[i][j], where 0 < i, j < *n-1,* equal to the average value of its four neighbours: A[i+1][j], A[i-1][j], A[i][j+1] and A[i][j-1].
The computation on the entire array should be repeated until the difference between the old value of each A[i][j] and the new one is less than some predefined tolerance *eps*. Note that the computation does not affect the boundary elements of A and these stay unchanged throughout the execution of the program.

You can assume that A, *n* and *eps* have all been declared and initialised appropriately.

Hint: Pay attention to avoid any race-conditions in your program.

Answer
```
// Assume that B points to a copy of A
    int i, again = 1;

#pragma omp parallel.        // It is more efficient to only enter parallel region once
{
    int j;
    double **tt,**tA,**tB;
    tA = A; tB = B;          // Create local pointers to A and B

    while (again) {          // While something has changed repeat
#pragma omp single
      again = 0;             // Set the global value to false
#pragma omp for reduction(||:again)
      for (i=1;i<n-1;i++) {
        for (j=1;j<n-1;j++) {
          t[i][j] = (A[i][j+1]+A[i][j-1]+A[i-1][j]+A[i+1][j])/4.0;
          if (fabs(A[i][j]-t[i][j]) >= eps)
            again = 1;    // Note that this is a local variable that will be reduced
        }
      }
      tt=t; t=A; A=tt;    // Switch which array is read from and which is written to
    }
}
// If the number of iterations of the while loop is odd then one could move the data
// from B to A at the end
```

Problem 4 (updated)
Write an efficient parallel program in C or C++ for computing C = A×B where A, B, and C are *n x n* matrices containing float values using *p* processes on a computer with distributed memory.

When the program starts both A and C are partitioned so that process *i* stores rows *i*n/p* up to *(i+1)n/p* in local arrays `myA` and `myC` respectively. Also, B is partitioned so that process *i* stores columns *i*n/p* to *(i+1)n/p* in a local array `myB`. Thus `myA`, `myB` and `myC` are all predefined, with both `myA` and `myC` of dimension *n/p x n,* while `myB` is of dimension *n x n/p*. When the program terminates the local matrices `myC` should together contain the final result.

You can make the following assumptions:

- The number of processes divides *n*.
- The internal send-buffer is sufficiently large so that `MPI_Send()` never blocks.
- You have access to a sequential matrix multiplication routine `MM(M1,M2,M3,d1,d2,d3)` that multiplies matrices `M1` and `M2` of dimensions *d1 x d2* and *d2 x d3* respectively and stores the result element-wise to matrix `M3` of dimension *d1 x d3*. I.e. the routine computes `M3 = M1×M2`.
- You do not have sufficient storage to allocate an array of dimension *n x n* on any process.

You can make use of the following MPI routines:

```
int MPI_Init(int *argc, char ***argv)
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
int MPI_Finalize( void )
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)
```

Answer

```
// Assuming that myA, myB and myC are already set up as described
// Also assuming an array myCT of dimension n/p x n/p

int main(argc,argv)
int argc;
char **argv;
{
  int myRank,p;                              // id of thread and number of threads
  int n,i,j;                                 // Assuming that n will be set
  MPI_Status status;

  MPI_Init(&argc,&argv);                     // Start MPI
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank);    // Get my id
  MPI_Comm_size(MPI_COMM_WORLD, &p);         // Get number of processes

  MM(myA,myB,myCT,n/p,n,n/p);                // First multiplication of initial values
  moveMatrix(myCT,myC,myRank,n/p);           // Move the result from myCT into myC

  for(i=1;i<n/p;i++) {
    MPI_Send(&(myB[0][0]),n*n/p,MPI_FLOAT,(myRank+1)%p,1,MPI_COMM_WORLD);
    MPI_Recv(&(myB[0][0]),n*n/p,MPI_FLOAT,(myRank-1+p)%p,1,MPI_COMM_WORLD,&status);
    MM(myA,myB,myCT,n/p,n,n/p);
    moveMatrix(myCT,myC,(myRank+i)%p,n/p);
  }

  MPI_Finalize();
}

moveMatrix(float **myCT,float **myC,int block,int dim) {
  int i,j;
  for(i=0;i<dim;i++)                          // Move elements from myCT to myC
    for(j=0;j<dim;j++)
      myC[i][j+dim*block] = myC[i][j];
}
```