IN4200 exam

15215

June 2020

1

(a) The time of doing a task requires one hour, and in the beginning we have no other option than to start in $T_{1,1}$. The following time steps is also without option, leaving us in with three workers in the following tasks, $T_{3,1}, T_{2,2}$ and $T_{3,3}$. Then, by trying to follow a diagonal working load which means to fill the diagonal first, we end up with a total of 11 hours, given that we also need an hour to finish $T_{5,5}$.

(b) One formula that finds the time t is the following formula,

$$t = \frac{N^2}{p} + (p-1).$$

However, this only works under a few conditions. When ever N^2/p returns a non-integer, one has to round off to the closest highest integer value. Thus, this means that if $N = 5$ and $p = 3$, it will return $5^2/3 = 8.33 \approx 9$, yielding a total time of $t = 11$, as found in task 1a).

2

I would parallize it the following way.

```
int i,j, sqrt_N;
char *array = malloc(N); // N is a predefined very large integer
array[0] = array[1] = 0;

#pragma omp parallel for
for (i=2; i<N; i++)
  array[i] = 1;

sqrt_N = (int)(sqrt(N)); // square root of N

for (i=2; i<=sqrt_N; i++) {
  if (array[i]) {

    #pragma omp parallel for
```

```
    for (j=i*i; j<N; j+=i)
      array[j] = 0;
  }
}
free (array);
```

For the first loop it is clear that there are no dependencies, and we can parallize it as and divide for loop into chuncks with appropiate sizes for each thread.

For the second part, we can see that the outer-loop is dependent on the as the first iteration can make an influence in the if-test for the later iterations for the purpose of not iterating through values that are already zero. Thus, we only parallize the inner loop. Drawbacks of this is that we have to intitialize an arbitrary number of threads several times, however, the inner loop is the more computationally heavy loop since it is a lot bigger.

I would expect a very small gain, if any at all, by parallizing this but with increasing gain as we increase the threads.

3

(a)
```
void sweep (int N, double **table1, int n, double **mask,
            double**table2){
  int i,j,ii,jj;
  double temp;

  #pragma omp parallel private(i,j,ii,jj,temp)
  for (i=0; i<=N-n; i++)
    for (j=0; j<=N-n; j++) {
      temp = 0.0;
      for (ii=0; ii<n; ii++)
        for (jj=0; jj<n; jj++)
          temp += table1[i+ii][j+jj]*mask[ii][jj];
      table2[i][j] = temp;
    }
  }
```

In this function, there is no dependencies in the loops, but there exists a temporary variable temp that gets written to many times. To avoid this being overwritten many times and provide the wrong answer, we need to make it private. Since we are not doing a sum over the many threads, we do not need to 'reduce' it.

In addition, we will also need to make all the iteration variables private, since we do not want them to give each threads their own chunks of it.

(b) I would start off with calculating the machine balance b_m, which is defined as

$$b_m = \frac{\text{memory bandwidth}}{\text{peak PF performance}}$$

From the exam text, we find that the processor has 6 memory channels and runs 2.666 GT/sec. Then we can find the memory bandwidth as

$$b_m = \cdot 6 \cdot 2.666 \text{ GT/sec} \cdot 1 \text{ word/T} = 16 \text{ GWord/sec}$$

Then, we can find the calculate the peak PF performance by using values from the exam text, where we find that the number of cores is two times 24, 2.70 GHz is the processors clock rate, and 32 bits per floating operation.

$$2 \cdot 24 \cdot 2.70 \cdot 32 = 4147.2 \text{ GFlops/sec}$$

This would make an estimate of the machine balance as $b_m = 0.0039$, which is not a big number.

In addition, we need to find the code balance, which is defined as

$$B_c = \frac{\text{data traffic}}{\text{floating-point operations}}.$$

Finding the data traffic requires a few assumptions. First off, we find that the total number of floating point operations are estimated as $2(N-n+1)^2n^2 \approx 2(N-n)^2n^2$ when N gets really big.

Second off, we find that the cache size is given as 33 MB, and since we have two processors we find a total L3 cache size of 66 MB. Given a N of atleast 10^4, this will make both table1 and table2 unable to be stored in the cache, but mask will be able to be stored in the cache. As a consequence, having mask in cache will mean we do not need to calculate the load in our calculations. Thus, the data traffic results in

$$B_c = \frac{(N-n)^2n^2 + (N-n)^2}{2(N-n)^2n^2} = \frac{n^2+1}{2n^2} = \frac{1}{2} + \frac{1}{2n^2}$$

Since the machine code is far less than the machine balance, we can tell that the memory is in this case the bottle neck. Thus, we find that the time used for the loops is very dependent on the loading and saving,

$$t = \frac{2(N-n)^2n^2 + (N-n)^2 \text{ Word}}{16 \text{ GWord/sec}}$$

To give an example, we can use 10^4, as given in the exam text, and find the time spent is 0.63 seconds.

Some reasons for discrepency can be hit and miss for the memory in the cache, which means every store has to be count as load plus store.

4

(a) In project 1 we discussed the compressed row storage (CRS) method which fits perfectly to this simple simulator. The idea is to store all the information in a matrix of $N \times N$

dimensions where N is the number of people in the system. Here we get the information of all the interactions between people, which means a row's non-zeros values will corresponds to an interaction with another person. We can update the values on the diagonals (namely the self-linkages in project 1) to keep up to date with the state of a person, being the chance of getting infected is in the range $(0, 1)$, being sick (1) or being immune (-1).

Since we want efficiency, the storage needed for a matrix will explode for a high number of people which is why we will utilize CRS. The idea here is that we will represent the matrix in three vectors instead. The first vector gets often called col-ind, and will explain the non-zero column indices for every row. The second vector is often named row-ptr, and gives the element of a new row in col-ind. This is because number of non-zero are arbitrary and varies from one row to the next. The last vector val is to keep track of the state of a person.

One way to initialize such a system is to read from a datagraph as we did in project 1.

(b) If we initialize the system such that the one person is infected, we need a temporary vector of dimension N to store temporary updated information. We also need to assume that there is no self-linkage in the system, which means that there are no values in the diagonal (since this is already stored in val).

In addition, we need to keep track how many days someone has been sick, and needs an array which is initialised with zeros of length N. And finally, one needs to take input parameter of how many days one can infect and be sick.

```
int evolve(int *row_ptr, int *col_idx, int **val,
           int** days_sick, int day_T, double f,
           int N){

  // This loop has to be independent of where we find who's sick
  // and who's immune this day.

  for (int i = 0; i < N; i++ ){
    if (val[i] != -1 && val[i] != 1){
      for (int j = row_ptr[i]; j < row_ptr[i+1]; j++ ){
        if (val[col_idx[j]] == 1 )
          (*val[i]) -= (1 - f);
      }
    }
  }


  for (int i = 0; i < N; i++){

    // Keeping track how long someone's been sick
    if (val[i] == 1)
      (*days_sick[i]) += 1;

    // An ill person gets sent home, and does not infect other
    // people. This will count like 'being immune' in this system
```

```
        // since it does not effectively have anything to say.
        // However, one can sample this parameter for data, but
        // considering hpc, we will not do this in this task.
        if (days_sick[i] == day_T)
          (*val[i]) = -1;

        // Correct formula for calculating sickness
        if (val[i] < 0) && (val[i] != -1)
          (*val[i]) += 1;

        // Finding who is infected
        if ((double)rand() / (double)RAND_MAX) < val[i])
          (*val[i]) = 1;

        return 0;
    }
```

(c) To start off, there are several ways to parallize this. One way is to use openMP and divide the loops into small tasks for threads.

Another opportunity is to parallize outside of the function by using MPI, where every core is doing their own experiment with either the same intialisation or not. One can run the same initialisation several times to see how the simulation changes from one run to another with varying parameters, and then combine the results together. From the law of big numbers, one can combine huge simulations together and find the average means using statistical methods.

In addition, one can always combine the two of them. That is dividing the simulations into cores, and then from cores to threads when running through loops.

5

A computer with multicore CPUs can be run efficiently using both openMP and MPI, however, there are some pros and cons using each one of them.

OpenMP is most of the times very easy to implement. It can be implementetd as one line in front of a loop, and most of the times the library already knows what to do. In addition, it can share memory between threads easily.

MPI can be quite difficult to implement, and needs several lines of code to do correctly. However, this is a good way to run bigger simulations on different nodes when the memory required is large. And when using a shared memory system, one can in addition share the memory while running MPI.