IN4200 – High Performance Computing and Numerical Projects

Final Exam

Candidate 15244

June 11 - June 18, 2020

Problem 1

1a

The following table shows one efficient way of executing the 25 tasks by 3 workers:

During hour 1: $T_{1,1}$

During hour 2: $T_{1,2}$ and $T_{2,1}$

During hour 3: $T_{1,3}$, $T_{3,1}$ and $T_{2,2}$

During hour 4: $T_{2,3}$, $T_{4,1}$ and $T_{3,2}$

During hour 5: $T_{3,3}$, $T_{5,1}$ and $T_{4,2}$

During hour 6: $T_{4,3}$, $T_{5,2}$ and $T_{1,4}$

During hour 7: $T_{2,4}$, $T_{5,3}$ and $T_{1,5}$

During hour 8: $T_{3,4}$ and $T_{2,5}$

During hour 9: $T_{4,4}$ and $T_{3,5}$

During hour 10: $T_{5,4}$ and $T_{4,5}$

During hour 11: $T_{5,5}$

Thus, 3 workers need 11 hours to finish all the 25 tasks.

1b

Since all tasks are equally time-consuming, the minimum time usage needed by 1 worker will simply be

$$T_{N,P=1} = N^2$$

Similarly, with $P = N$ workers the dependency relationship between the tasks dictate that

$$T_{N,P=N} = N + N - 1 = 2N - 1$$

For an arbitrary amount of workers this generalizes to

$$T_{N,P} = \left\lceil \frac{N^2 - P}{P} \right\rceil + P,$$

or alternatively

$$T_{N,P} = \left\lfloor \frac{N^2 - 1}{P} \right\rfloor + P$$

Here, $\lceil \cdot \rceil$ is the ceiling function and $\lfloor \cdot \rfloor$ the floor function, which both ensures that no tasks are left undone.

Problem 2

The first loop, assigning values "1" to the array (with the exception of the first two elements), can be parallelized with the combined directive **parallel for**. The **parallel** directive specifies the region which should be executed in parallel, and the **for** directive specifies that the following for loop will be executed in parallel. The **for** directive uses a static schedule as default, i.e. it divides the iterations into contiguous chunks of (roughly) equal size.

In the subsequent nested loop, the most speedup is achieved by parallelizing the inner loop since $\sqrt{N} << N$. That is, the inner loop has many more iterations than the outer and hence benefits the most from parallelization. Again, the (inner) loop can be parallelized with the combined directive **parallel for**.

The inner loop is only executed if the if-test is passed, which happens if the ith array element has value 1. The (inner) loop construct specifies an iteration where the loop variables i will be prime numbers less than or equal to $\sqrt{N}$. In other words, the inner loop is executed *number of primes* $\leq \sqrt{N}$ times. Regarding potential speedup achievable by increasing the number of workers, or threads, P; having $P < \sqrt{N}$ workers should then in theory result in a speedup. However, potential slowdown due to load imbalance and/or overhead should be assessed when choosing an optimal P.

The parallelized version of the code segment is shown below:

```
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);
```

Problem 3

3a

Since $n << N$ the two outermost loops will have many more iterations than the two innermost, so the outermost should be parallelized. This can be achieved by the directive **parallel for collapse(2)**. A regular **for** directive will only parallelize one level, but adding the **collapse** clause makes it possible to parallelize more than one level. It is only possible to collapse nested loops where the loop body of the outer loop only consist of the inner loop. In general, the efficiency of the parallelization increases the more work there is to divide over a number of threads. By parallelizing more levels of loops instead of just the outer one, the amount of work can be increased. The next clause needed is **reduction(+: temp)**, since temp is a shared variable which is modified in every iteration. The **reduction** clause prevents concurrent writes to the variable. Not preventing concurrent writes might introduce a data race, since multiple threads could try to update the shared variable simultaneously. Finally, the loop variables of the two innermost loops are declared private by the clause **private(ii, jj)**. When a variable is declared private, each thread is assigned a local copy.

The parallelized version of the code segment is shown below:

```
void sweep (int N, double **table1, int n, double **mask, double**table2)
{
  int i,j,ii,jj;
  double temp;
  #pragma omp parallel for collapse(2) reduction(+: temp) private(ii, jj)
  {
    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;
      }
    }
}
```

3b

The machine balance, B_m, of a processor is the ratio between the maximum memory bandwidth and the peak FP performance

$$B_m = \frac{\text{memory bandwidth [GWords/sec]}}{\text{peak FP performance [GFlops/sec]}} = \frac{b_{\max}}{P_{\max}}$$

The peak FP performance is given by

$$P_{\max} = \text{number of cores} \times \text{clock rate} \times \text{DP Flops/cycle}$$

For a Intel Xeon 24-core processor of model 8168, the peak FP performance is thus

$$P_{\max} = 24 \text{ cores} \times 2.70 \text{ GHz clock rate} \times 32 \text{ DP Flops/cycle} = 2074 \text{ GFlops/sec}$$

The maximum memory bandwidth is given by

$$b_{\max} = \text{clock rate} \times \text{no. channels} \times \text{Words/Transfer}$$

For a Intel Xeon 24-core processor of model 8168, the maximum memory bandwidth is thus

$$b_{\max} = 2.666 \text{ GT/sec} \times 6 \text{ channels} \times 1 \text{ Words/T} = 16 \text{ GWords/sec}$$

The machine balance for two Intel Xeon 24-core processors is then

$$B_m = \frac{2 \cdot b_{\max}}{2 \cdot P_{\max}} = \frac{16 \text{ GWords/sec}}{2074 \text{ GFlops/sec}} = 0.0077 \text{ W/F}$$

Next, to characterize a loop, we can calculate the code balance B_c:

$$B_c = \frac{\text{data traffic [Words]}}{\text{floating-point operations [Flops]}}$$

In our loop, each iteration loads/stores $3n^2 + 1$ values from/to memory and executes $4n^2$ floating-point operations. Hence

$$B_c = \frac{(2n^2+1) \text{ Words}}{4n^2 \text{ Flops}} = \left(\frac{1}{2} + \frac{1}{4n^2}\right) \text{Words/Flops}$$

which gives a max $B_c = 0.75$, for $n = 1$ and min $B_c = 0.5$, for large n.

Since $B_c >> B_m$, the speed of the computation is determined by the memory traffic. A naive formula for the minimum computing time is then

$$T = \frac{\text{total flops [Bytes]}}{\text{no. processors} \times \text{memory bandwidth [Bytes/sec]}} \tag{1}$$

With $N = 10\,000$ and $n = 10$, our naive estimate of the minimum computing time thus is

$$T = \frac{(2n^2+1)(N-n)^2 \cdot 8\text{Bytes}}{2 \text{ processors} \cdot 128 \text{ GB/sec}} \approx 0.62 \text{ sec}$$

Reasons for non-perfect speedup:

- **Load imbalance:** The problem either was not or could not be partitioned into equal sets, so the workers might no execute their tasks equally fast. OS jitter may also contribute to load imbalance.
- **Overhead:** Parallelism adds overhead, such as communication between workers. Also, one should use as few threads as possible. The scalability of the main memory bandwidth determines the optimal number.
- **Environment:** Cooling of the hardware is essential for optimal performance. Too hot hardware will result in a dip in performance.

Problem 4

4a

Let N denote the number of persons (nodes) and E the number of person-person interactions (edges) in the the graph. An edge between a pair of nodes corresponds to two people having a daily interaction with a risk of transmitting the virus if one of them is ill. Assuming the graph is stored in a format similar to the web graphs in Home Exam 1, we can easily store the interaction information in two 1D arrays, each of length E. In more detail, we assume a graph given in the format:

```
# Nodes: N Edges: E
# Person_i  Person_j
  integer1   integer2
  integer1   integer2
  .
  .
  .
```

The interaction between person i and person j can then be read and stored in two 1D arrays by the following procedure:

- Read and store N and E
- Allocate two 1D arrays, *person_i* and *person_j*, with length E
- for each line k containing interaction information assign:
  - *person_i*[k] = integer1
  - *person_j*[k] = integer2

In addition, we need an 1D array of length N, named *states*, for storing the states (healthy, infected, infected but in isolation, immune) of the persons, and another 1D array of length N, named *timer*, for counting the number of days with infection.

This way of structuring the data, with only 1D arrays, is chosen because 1D arrays are efficient due to their contiguous nature. Moreover, this approach allows for using only a single loop when simulating daily interactions in the next task.

4b

The states healthy, infected, infected but in isolation and immune can be represented by integers, for instance 2, 1, 0, -1, respectively. Initially, every person is assigned a healthy state in the *states* array. We then start the simulation by infecting a given number of people by randomly changing the state from healthy to infected, and increasing the days-infected counter for the unlucky by 1 in the *timer* array (since an infected person can start transmitting the disease the day following infection).

Below is an implementation of a possible function that simulates a daily interaction and updates the states of all people, with detailed comments.

```c
void update(int N, int E, double f, int T, int X, int *states, int *timer, int *person_i, int *
    person_j)
{
  /*
  Simulate all daily interactions and update the states (healthy, infected,
  infected but in isolation, immune) of all people. The states are assigned
  integer values (2, 1, 0, -1) respectively.

  INPUTS
  ------
  N - number of nodes (people)
  E - number of edges (interactions)
  f - the probability of being infected
  T - number of days infected interacts with others
  X - number of days to become immune after isolation
  states - array of length N storing every persons state
  timer - array of length N storing the number of days infected
  person_i - array of length E
  person_j - array of length E

  The person-person interactions are stored by the correspondence between
  person_i and person_j
  */

  // Once infected, a person cannot infect others before the following day.
  // In order to ensure that no newly-infected transmits the disease,
  // possible changes in the state are stored in a temporary states array
  int *states_tmp;
  states_tmp = (int*)calloc(N, sizeof(*states_tmp));

  // Declare variable for assigning random number
  double r;

  // Simulate daily interactions and check if someone becomes infected
  for (int k=0; k<E; k++){
    // Check whether:
    // 1. The states of person_i and person_j are different
    // 2. The state of person_i is not 'infected but in isolation' or 'immune'
    // 3. The state of person_j is not 'infected but in isolation' or 'immune'
    if ((states[person_i[k]] !=states[person_j[k]]) && !(states[person_i[k]] <= 0) && !(states[
    person_j[k]] <= 0)) {
      // If True, either person_i or person_j is infected
      // Draw a random number unif(0, 1)
      double r = (double)rand() / (double)RAND_MAX;
      // Check if the interaction results in a transmission of the disease
      // according to the prescribed probability f
      if (r<f){
        // Infection! Update the temporary state
        states_tmp[person_i[k]] = states_tmp[person_j[k]] = 1;
      }
    }
  }

 // Update states and timer
 for (int l=0; l<N; l++){
   // If state is infected, add 1 to timer
   if (states[l] == 1){
     timer[l] += 1;
   }

   // If state is infected but in isolation, add 1 to timer
   if (states[l] == 0){
     timer[l] += 1;
   }

   // If the state is changed to infected due to the daily interaction
   if (states_tmp[l] != 0){
     states[l] = 1;
   }

   // If T days have passed with infection, change state to infected
   // but in isolation
   if (timer[l] == T){
     states[l] = 0;
     }

     // If X days in isolation have passed, change state to immune
     if (timer[l] == T+X){
       states[l] = -1;
     }
   }

   free(states_tmp);
 }
```

Since there only are single loops, this approach should be reasonably efficient. However, the many if-tests may make this approach suffer some slowdown, especially in the loop that simulates the interactions. So finding an approach with less tests but still only single loops will likely be more efficient.

4c

MPI is designed for data parallel applications. That is, each MPI process runs the exact same program, but on different data. The epidemic simulator is hence perfect to parallelize with MPI, as it allows us to run multiple realizations, i.e. simulations with different parameters, concurrently.

Problem 5

NUMA is a memory design where the memory is physically distributed into multiple memory nodes and logically shared, i.e shared data is accessible to all processors. However, this hierarchical memory means that a processors memory access performance depends on the processors proximity to the memory node. For memory-bound threaded applications, the data locality is particularly important since having processors that frequently access non-local memory will be detrimental to the performance. In order to control the memory placement, a parallel first touch policy memory initialization can be used. This makes sure a memory page gets mapped into the locality domain of the processor that first accesses it.

OpenMP is a standard for shared memory parallel computing, and should in theory be the perfect candidate for a shared memory machine like this. However, there might arise problems with memory locality. The first touch policy might map the memory such that it becomes non-local for a thread that needs to access it. By adopting the philosophy of “the data is where it is”, OpenMP however allows for moving a thread to the data it needs most.

MPI is a standard for distributed memory parallel computing, and might be advantageous on NUMA architectures since it encourages memory locality. However, the performance might suffer from communication overhead and synchronization.

For large applications, a hybrid of MPI + OpenMP might scale better as long as the performance do not suffer from the additional overhead. With this approach, the MPI processes spawn OpenMP threads in the local NUMA domains with shared memory, and MPI communicates between the domains.