SPM Assignment - 4

Simone Passèra

May 31, 2024

1 Introduction

This assignment was intended to parallelize an algorithm that implements a numerical computation on a set of keywords.

I implemented two parallel versions of the `nkeys.cpp` code using MPI. The implementations and the tests performed on the cluster provided by the University of Pisa are presented in more detail below.

2 Code Implementation

```
const long SIZE = 64;

int main(int argc, char* argv[]) {
  .....

  for(int i = 0; i < length; i++) {
    key1 = random(0, nkeys-1);  // value in [0,nkeys[
    key2 = random(0, nkeys-1);  // value in [0,nkeys[

    if (key1 == key2) // only distinct values in the pair
      key1 = (key1+1) % nkeys;

    map[key1]++;  // count the number of key1 keys
    map[key2]++;  // count the number of key2 keys

    float r1;
    float r2;

    // if key1 reaches the SIZE limit, then do the computation and then reset the counter ....
    if (map[key1] == SIZE && map[key2]!=0) {
      r1= compute(map[key1], map[key2], key1, key2);
      V[key1] += r1;  // sum the partial values for key1
      resetkey1 = true;
    }

    // if key2 reaches the SIZE limit ....
    if (map[key2] == SIZE && map[key1]!=0) {
      r2= compute(map[key2], map[key1], key2, key1);
      V[key2] += r2;  // sum the partial values for key1
      resetkey2=true;
    }

    if (resetkey1) {
      // updating the map[key1] initial value before restarting the computation
      auto _r1 = static_cast<unsigned long>(r1) % SIZE;
      map[key1] = (_r1>(SIZE/2)) ? 0 : _r1;
      resetkey1 = false;
    }

    if (resetkey2) {
      // updating the map[key2] initial value before restarting the computation
      auto _r2 = static_cast<unsigned long>(r2) % SIZE;
      map[key2] = (_r2>(SIZE/2)) ? 0 : _r2;
```

```
44        resetkey2 = false;
45      }
46    }
47
48    // compute the last values
49    for(long i = 0; i < nkeys; i++) {
50      for(long j = 0; j < nkeys; j++) {
51        if (i==j) continue;
52
53        if (map[i]>0 && map[j]>0) {
54
55          auto r1= compute(map[i], map[j], i, j);
56          auto r2= compute(map[j], map[i], j, i);
57          V[i] += r1;
58          V[j] += r2;
59        }
60      }
61    }
62
63    .....
64  }
```

Listing 1: nkeys sequential version.

The code in Listing 1, shows the main parts of the sequential version (`nkeys.cpp`). The program basically generates a sequence of key pairs (of length *length*), and for each key in the pair increments the counter associated with that key by one, line 13. When the counter reaches a specific value (SIZE), it performs a matrix product that will later be summed to the final value for the key that reached the limit, and the counter will be reset to a new value that depends directly on the value just calculated. As you can see the main computation resides in the `compute` function and the parallelization of this code is to distribute the various computations among several MPI processes. The versions i made were implemented with the aim of providing the final key values with the same arithmetic precision as the sequential version. This implies that relaxing this assumption could achieve better results with different algorithms.

Furthermore, for each version, explained below, a variant was made that exploits OpenMP to try to parallelize the matrix product as well.

2.1 "Distributed" version

This first parallel version is realized in the following way:

The root process (rank 0) generates the sequence of key pairs and sends it one portion at a time via the collective broadcast to all processes. Each process holds in a sense the ownership of a set of keys that need to be computed. Thus, before a later submission of the sequence, each process consumes the sequence and performs the matrix product of the keys it has when the limit is reached, for the others simply increments the counter. Each time the matrix product is calculated, the process sends the new map value to everyone else. Whenever the matrix product is calculated for a given key (key1), it is essential to know the map value for the other key (key2) and vice versa. Therefore if the process does not possess the key (key2) and the map (map[key2]) exceeds SIZE, it waits with `MPI_Recv` for the new value calculated by another process. After the entire sequence is completed, each process proceeds to calculate the last for (line 49) with the same ownership logic. Eventually, each process sends the final values for its keys to the root process through the collective gather.

2.2 "Farm" version

This second parallel version is realized in the following way:

In this case the processes realize a farm pattern. The root process (rank 0) is the emitter and the other processes the workers. The emitter basically executes the sequential code, however when the compute function is to be executed it sends the parameters to a worker that will

eventually compute and return the value to the root process. The emitter distributes tasks to workers in a round robin fashion, and the emitter itself is also included, which can compute. In this version the emitter waits for the value computed by a worker for a given key when the key is generated again in the sequence, to get the new value of the associated map. However, for a future version one could exploit the fact that the computed value belongs to the range $[0, SIZE/2]$, to try to postpone as much as possible the time when to wait for the new value.

3 Evaluation

This section discusses the tests performed on the implemented parallel versions in order to evaluate its performance. The tests were conducted on the internal nodes of the cluster *spmcluster.unipi.it*. Each measurement was performed 10 times and the average value was considered. The standard deviation is negligible, and is not present in this document. Within the root directory is the Makefile with which the various versions can be compiled and also the Slurm scripts that were used for testing.

Absolute Speedup is the metric used to evaluate the performance of the parallel versions. Tests were performed by varying the number of physical nodes and the number of processes for a computation of 2 and 100 keys.

All versions using OpenMP to parallelize the matrix product were tested with a number of threads equal to 4, this is because it turned out to be the optimal value after running a few tests with the SIZE value equal to 64.

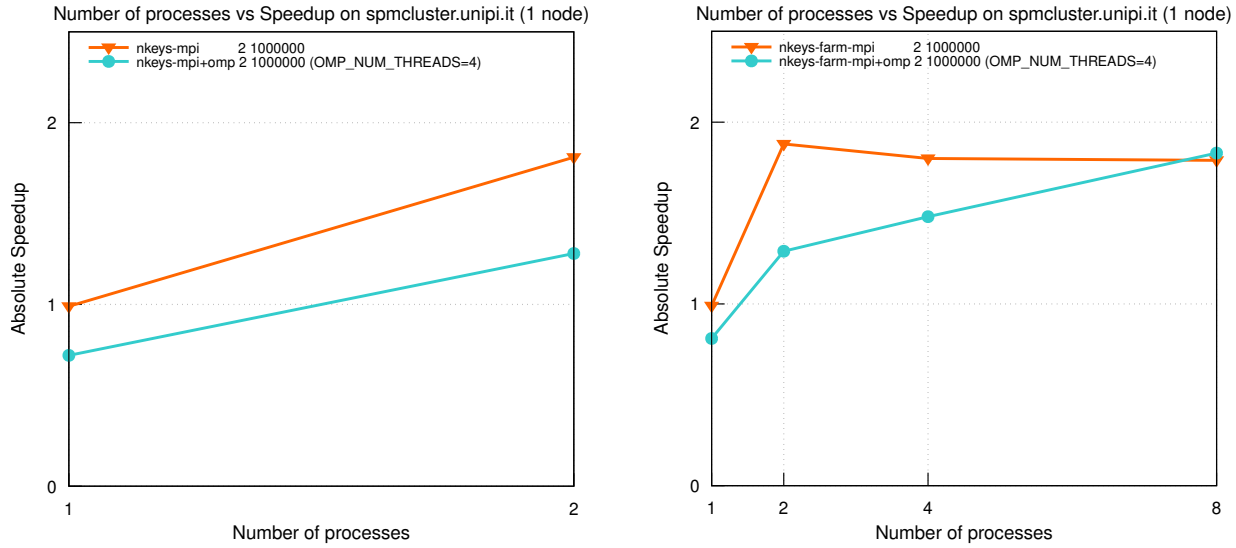

Figure 1: Absolute Speedup vs Number of processes performed on a single internal node of spmcluster.unipi.it. (number of keys = 2). `Left`: "Distributed" version. `Right`: "Farm" version.

In Figure 1, it can be seen that the "Distributed" version was not evaluated scalability for more than two processes for computing two keys. This is because with 2 processes, each of them has a key and therefore creating multiple processes is possible but totally useless since these processes will not contribute to the calculation. On the other hand for the "Farm" version increasing the number of workers can increase the speedup, but for only two keys the parallelism that can be exploited is not that high.

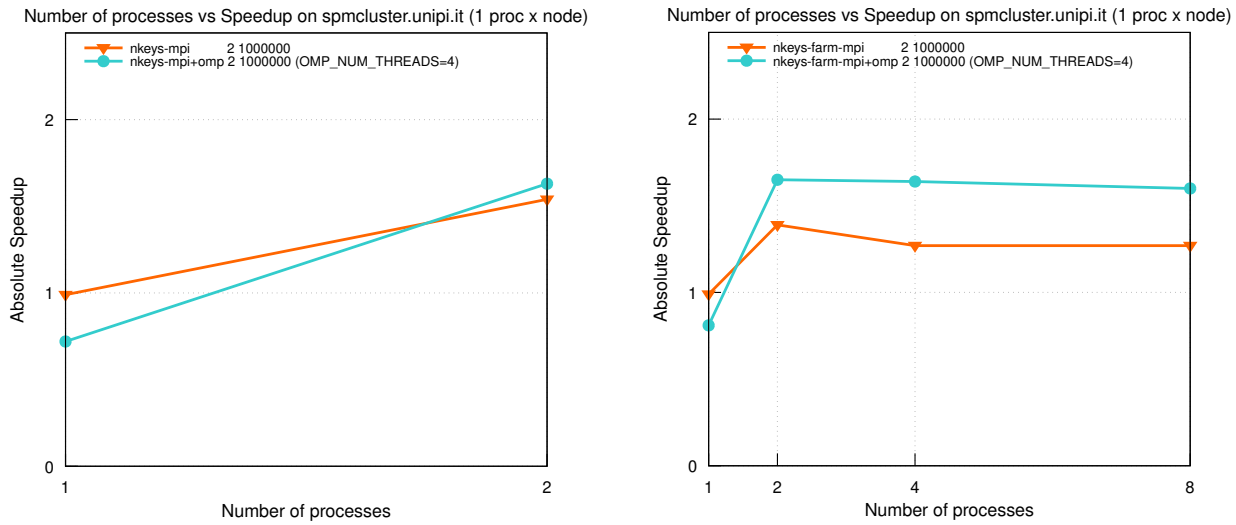

Figure 2: Absolute Speedup vs Number of processes performed on a maximum of 8 node of spmcluster.unipi.it (one process per node). (number of keys = 2). `Left`: "Distributed" version. `Right`: "Farm" version.

In Figure 2, you can see how using the same number of processes, as in Figure 1, but on different nodes, leads to similar or somewhat worse results because of the more expensive communications. For the version with OpenMP, the other way around, distributing the threads on different nodes instead of all on the same node allows a higher speedup.

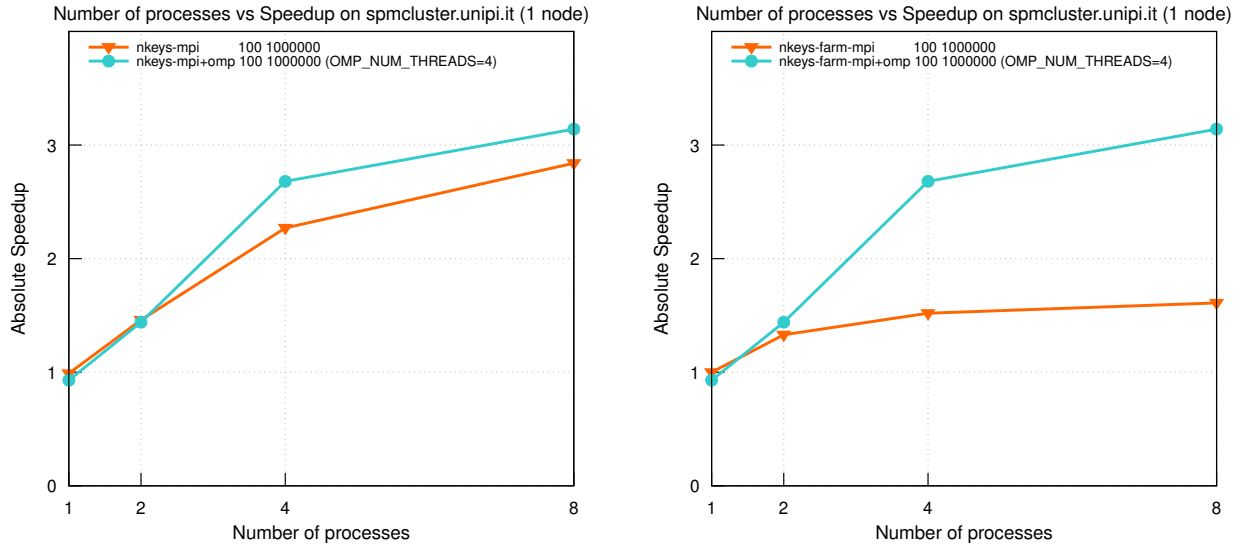

Figure 3: Absolute Speedup vs Number of processes performed on a single internal node of spmcluster.unipi.it. (number of keys = 100). `Left`: "Distributed" version. `Right`: "Farm" version.

In Figure 3, you can see how by increasing the number of keys you can obtain a better speedup compared to Figure 1. These processes are once again executed on a single cluster node. Both versions achieve more or less the same speedup with OpenMP. Instead, without it, the "Distributed" version makes better use of parallelism, in fact, as previously mentioned, the "Farm" version can be further improved.

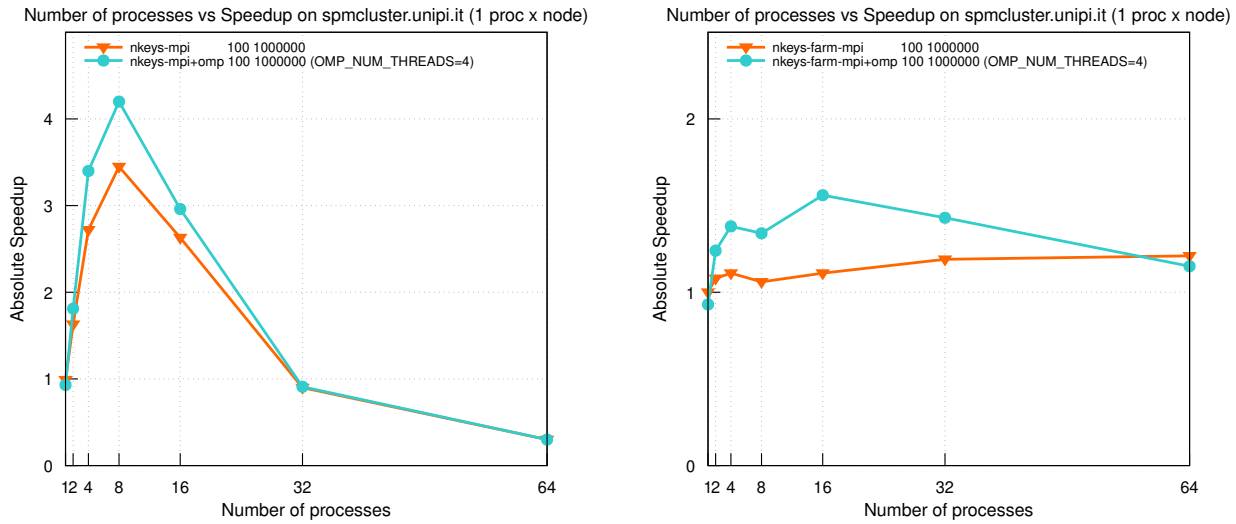

Figure 4: Absolute Speedup vs Number of processes performed on a maximum of 8 node of spmcluster.unipi.it (one process per node). (number of keys = 100). `Left`: "Distributed" version. `Right`: "Farm" version.

In Figure 4, the two versions perform a calculation for 100 keys, and in this case multiple processes were used until the eight nodes available were completely used. As you can see, the "Distributed" version achieves the highest speed up detected, using 8 nodes with one process per node. On the other hand, as is evident from the graph, further increasing the number of processes leads to a drop in performance. However, the "Farm" version does not have as noticeable a drop as the "Distributed" version, most likely due to the lower number of communications involved.

4 Conclusions

To summarize, the two implemented versions try to parallelize the problem via MPI following two different approaches. The "Distributed" version achieved better results than the "Farm" version. However, the farm version could exploit parallelism more and thus achieve better results also because it uses less communication than the other version. In addition, the use of OpenMP has proven to be useful for achieving higher performance.