COMP90025 Parallel And Multicore Computing: Assignment 2

Martin Raunkjær Andersen - login: marand
mrandersen@student.unimelb.edu.au
1011164
Matthijs van Delft - login: matthijsvandelft
mvandelft@student.unimelb.edu.au
1010922

October 20, 2018

1 Introduction

During the previous two projects, different implementations of algorithm have been implemented, utilizing *OpenMP* and *MPI*. OpenMP based algorithm only functions with shared memory while OpenMPI based algorithms functions on distributed memory (EREW) architectures. Since we have not yet used OpenMP and OpenMPI in implementations of the same algorithm, we have not observed the performance differences between these variations. Our motivation for this project is to observe and compare these performance differences (if they exist).

This report describes three different parallel implementations (OpenMP, MPI and OpenMP + MPI (hybrid)) of an 4-clique algorithm and compares and discusses the runtimes of each.

2 Research Hypothesis

Our research hypothesis is as follows: We hypothesize that an OpenMP implementation will generally have faster execution times when running on shared memory systems compared to OpenMPI variants with equal amount of cores distributed over different nodes. This is because the OpenMPI implementation has communication costs between the nodes, that OpenMP does not. We furthermore hypothesize that a hybrid implementation (utilizing both OpenMP and OpenMPI) will lie somewhere between the runtimes of the OpenMP and OpenMPI only variants.

The problem chosen is the 4-clique problem, which looks for the number of 4-cliques in a graph and is derived from the maximum clique problem[1]. The brute-force implementation for this problem involves four nested loops with a conditional statement in the innermost loop. The complexity of this algorithm is $O(n^4)$. This problem was chosen because it is easily parallelisable for all versions that we wanted to compare and the runtime complexity is sufficient to our needs.

3 Parallel

Based on the sequential algorithm for calculating the number of 4-cliques, see code snippet 1, three different parallel implementations are realized.

```
nrCliques ← 0
for (i ← 0 to numberNodes) then
    for (j ← 0 to numberNodes) then
        for (k ← 0 to numberNodes) then
            for (l ← 0 to numberNodes)
    then
                if (g[i][j] ≥ 1 AND g[i][k]
     ≥ 1 AND g[i][l] ≥ 1 AND g[j][k] ≥ 1
    AND g[k][l] ≥ 1) then
                    nrCliques ← nrCliques
    + 1
```

Listing 1: Sequential 4-clique

During the implementation of the parallel variants, the algorithm it self is not change to make sure that the difference in results is a product of the OpenMP and MPI additions.

3.1 OpenMP

The OpenMP implementation is realized by using *#pragma omp parallel for* on the most outer for-loop, see code snippet 1. By default this statement using static scheduling with a chunk size of equal to loop iterations divided by number of threads.

```
nrCliques ← 0
#pragma omp parallel for
for (i ← 0 to numberNodes) then
    ...
```

Listing 2: OpenMP 4-clique

3.2 MPI

The MPI implementation is realized in a way were there is no shared memory. This means message passing has the be executed to get distributed results in one place. A task is executed for every core in the system, which each calculates an individual sub-problem, see code snippet 3.

```
nrCliques ← 0
for (i ← 0 to (i + idProcess) < nrNodes)
    then
    m ← i + idProcess
```

```
4       for (j ← 0 to nrNodes) then
5           for (k ← 0 to nrNodes) then
6               for (l ← 0 to nrNodes) then
7                   if (g[m][j] ≥ 1 AND g[m][k]
        ≥ 1 AND g[m][l] ≥ 1 AND g[j][k] ≥ 1
        AND g[k][l] ≥ 1) then
8                       nrCliques ← nrCliques
        + 1
9
10      i ← i + nrProcesses
11
12  globalCount ← 0
13  MPI_Reduce(nrCliques, globalCount, 1,
        MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD)
```

Listing 3: MPI 4-clique

The function *MPI_Reduce* is used to combine all the partial distributed results in one total result. Since not all the sub-problems necessarily take the same time to compute, the tasks need to wait for each other the actual execute the reduce. This so called barrier makes for most of the communication overhead, especially when it is executed often and when the execution time for calculating the sub-problems varies a lot.

3.3 Hybrid

The hybrid implementation is a result of combining OpenMP and MPI. Each node runs a single task which executes a individual sub-problem. Which means that multiple sub-problems are executed at the same time on different nodes. The number of tasks per node is specified in the SLURM file for this implementation. If this arbitrary nodes contains multiple cores, OpenMP is used to divide each individual sub-problem based on the amount of cores per node by using *#pragma omp parallel for* in the second most outer for-loop of code snippet. The second most outer for-loop is used since the most outer one is used to divide the sub problems over different nodes. The partial result calculated by each node are combined in on total result using the *MPI_Reduce()* function, described in section 3.2.

```
1   nrCliques ← 0
2   for (i ← 0 to (i + idProcess) < nrNodes)
        then
3       m ← i + idProcess
4
5       #pragma omp parallel for
6       for (j ← 0 to nrNodes) then
7           ...
```

Listing 4: Hybrid 4-clique

4 Results

It is important to note that the test cases of the different implementations cannot fully overlap. The OpenMP implementation is bound the a maximum number of cores per processor, since it is not able to utilize distributed memory. Hence, the OpenMPI and hybrid implementation can be tested a higher number of cores. It is assumed that the trend for the OpenMP implementation will stay the same, even if it would be able to test for more cores. With the assumption in place, the results trend can be used to compare to the results of the OpenMPI and hybrid implementation.

Unfortunately, during the testing stage no resource have been available on Spartan for multiple days in a row. This resulted in executing the test cases on personal machines, which if far from ideal, since these are single node machines with a maximum of 4 cores. Hence, the concept of dividing the sub problems over multiple nodes is tested on a single node machine, this lead to some interesting unexpected results.

The following figure visualize the executions of the different implementations for two scenarios for an average of 10 runs. Figure 1 gives the result for a graph of 300 nodes and figure 2 for a graph with 250 nodes. The three parallel implementations are tested with an 2, 3, and 4 core configuration noted as $implementation - xt$, where x is the number of cores.

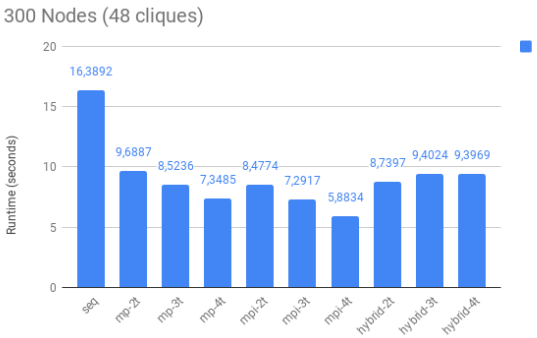

Figure 1: Resulting runtimes on a 300 node graph (averaged over 10 runs each)

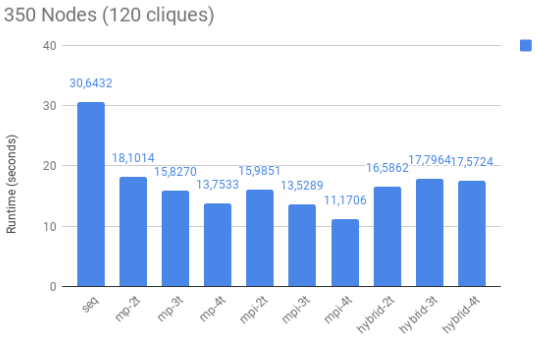

Figure 2: Resulting runtimes on a 350 node graph (averaged over 10 runs each)

5 Discussion

Looking back at the results, the hypothesis made in section 2 can be proven wrong for the test configuration used to measure the execution times. The hypothesis described expected results as OpenMP having the lowest execution times for a maximum amount of cores on a node, followed by the hybrid, OpenMPI and sequential implementations. Furthermore, the hybrid implementation was expected to have lower executions times compared to the OpenMPI implementation on multiple nodes. Surprisingly, the low to high executions times are: OpenMP, OpenMPI and hybrid. However, the hypothesis can not really be answered, as the results would have been very different when testing on Spartan.

The biggest victim of only testing on a single node (our pc) was the hybrid implementation. We used the $-np$ flag of *mpirun* to test the hybrid implementation with different number of cores. However, it turns out that this "hides" the other cores from OpenMP, only giving it access to a single core, thereby completely negating it. If something interesting can be gleaned from the hybrid results, it is that the results of 2 allocated cores are better than the results from 3 and 4 allocated cores. It seems like OpenMP might be interfering here, which increases the time overhead.

Although the execution times of the OpenMP and OpenMPI implementations seems similar, OpenMPi is constantly a little faster compared to OpenMP, even though there is *MPI_Reduce* call, which was expected to make it slower. Unfortunately, no clear reason is visible to us to explain this behaviour. At first it was thought it was the result different distributions over the number of cores. OpenMP static scheduling uses by default a chunk size of *loop count / number of threads*, where the OpenMPI implementation is similar to static scheduling with a chunk size of 1. Which could be the case highly varying executions times of loop iterations since a smaller chunk size could distributed the computational heavy parts more evenly over the cores. This is not the case though since the there are no varying execution times per loop iteration in our implementation. Furthermore, it might be that the barrier of the *MPI_Reduce* call does not appear enough to make a significant difference.

Another reason why an OpenMPI implementation might be faster than an OpenMP implementation on a single node could be a matter of increased overhead of the OpenMP implementation, because it has to manage shared memory between the threads, as well as the scheduling and management of the overall job. The *#pragma omp parallel for* construct, which is used by the OpenMP implementation, has not been optimized in any way, such as with explicit scheduling and collapse, and as such probably has to do more work at runtime to make sure that the job is running well. The OpenMPI implementation, on the other hand, assumes that it has all its memory to itself, so it does not have to safeguard against simultaneous access, and its scheduling is explicitly described, so it does not have to do any runtime work to schedule the iterations. As mentioned above, we tried to the same static scheduling for OpenMP, but it did not increase the running times noticeably, so the scheduling itself is not the entire answer, but it might be part of it. It is just an example of the work that the OpenMP has to do, which OpenMP does not. Of course, the communication overhead between nodes that would normally penalize OpenMPI is also gone.

Another noticeable thing is that efficiency for a number of cores more than 2 is falling very quickly for all versions. For instance, the efficiency of OpenMPI with 2 cores on the 350 nodes input is quite good at $30.6432/15.9851 * 2 = 0.9585$, which decreases to $30.6432/13.5289 * 3 = 0.7550$ at 3 cores and to $30.6432/11.1706 * 4 = 0.6858$ at 4 cores. And the other two versions are worse. It is curios why this fall in efficiency occurs, as the problem requires very little overhead in communication between the processors and therefore should enjoy a better degree of speedup when more processors are added than we are seeing.

References

[1] S. University, *Maximum clique problem*, `https://cs.stanford.edu/people/eroberts/courses/soco/projects/dna-computing/clique.htm`, (Accessed on 10/16/2018),