高性能 lab2

实验目的

Implement parallel BFS (Breadth First Search) to help familiarize yourself with OpenMP and OpenMP+MPI mixed programming.

NOTICE:

1. 我需要补交的算法是bfs_omp_mpi.cpp（提交时还有错误），bfs_omp.cpp按时交的

实验任务

- Fill in the `bfs_omp` function in the `bfs_omp.cpp` file and the `bfs_omp_mpi` function in the `bfs_omp_mpi.cpp` file. Use OpenMP and OpenMP+MPI to implement the parallelized BFS algorithm in these two functions, and finally get the shortest from node 0 to all nodes Distance (length of each side is 1)

实验实现

After reading and understanding the given top-down algorithm in `bfs_common.cpp` and how it utilized CSR to store graph data, I tried many methods to improve the algorithm.

bfs_omp

I first implemented a BFS Bottom-Up approach. In some cases, this method is sometimes more efficient because instead of iterating over all vertices in the frontier and marking all vertices as a neighbor to the frontier. We can implement a BFS algorithm that has each node check whether it should be added to the frontier. However, without parallelism, it would not make a significant difference in execution time. Thus, I tried a parallelized BFS top down algorithm.

In order to do so, I first had to identify where I could add parallelism, as well as inserting the appropriate synchronization to ensure correctness.

I inserted this

```
 #pragma omp parallel num_threads(NUM_THREADS) private(local_count)
```

which would parallelize the standard sequential BFS top down method

I utilized

```
__sync_bool_compare_and_swap((T* __p, U __compVal, V __exchVal)
```

which compares the value of `_compVal` with the value of the variable that `_p` points to. If they are equal , the value of `_exchVal` is stored in the address specified by `_p` , otherwise, no operation is performed. A full memory barrier is created when this function is invoked. It ensures that local frontier is to be synchronized with the global new_frontier

bfs_omp_mpi

实验结果

bfs_omp

对 OpenMP 版本，报告使用 1, 7, 14, 28 线程 在 `68m.graph` 、 `200m.graph` 或 `500m.graph` 图下 `bfs_omp` 函数的运行时间（三个图选择一个报告即可），及相对单线程的加速比 。

I chose to test the 68m.graph. In order to change the number of threads running, there is global variable `NUM_THREADS`

68m.graph with parallel top_down_method

change `NUM_THREADS` in order to change number of threads being used

```
$ srun -N -1 -n ./bfs_omp ./graph/68m.graph
Graph stats:
  Edges: 68993773
  Nodes: 4847571


Thread Count = 1 : Average execution time of function bfs_omp is 1274.688800 ms.
    加速比例: 1.0
Thread Count = 7 : Average execution time of function bfs_omp is 397.577700 ms.
    加速比例: 3.207
Thread Count = 14 : Average execution time of function bfs_omp is 223.818100 ms.
    加速比例: 5.452
Thread Count = 28 : Average execution time of function bfs_omp is 123.047500 ms.
    加速比例: 10.359
```

bfs_omp_mpi

对 OpenMP+MPI 版本，报告 1×11×1, 1×21×2, 1×41×4, 1×141×14, 1×281×28, 2×12×1, 2×22×2, 2×42×4, 2×142×14, 2×282×28, 4×14×1, 4×24×2, 4×44×4, 4×144×14, 4×284×28 进程（N×PN×P 表示 NN 台机器，每台机器 PP 个进程，线程数 TT 自定，但 P×TP×T 不建议超过 28）在 `68m.graph` 、 `200m.graph` 或 `500m.graph` 图下 `bfs_omp_mpi` 函数的运行时间（三个图选择一个报告即可），及相对单进程的加速比 。此部分测试建议使用脚本提交运行、记录输出，以减少工作量。

In this experiment, `NUM_THREADS` always equals 28. I used a hybrid method that used dynammically chose between your my top down and buttom algorithms based on the size of the frontier or other properties of the graph. This is represented by the `THRESHOLD` variable and can be manipulated. In this article, https://crd.lbl.gov/assets/pubs_presos/mtaapbottomup2D.pdf, I learned that a high-performance BFS will use the top-down approach for the beginning and end of the search (when the frontier is relatively small) and the bottom-up approach for the middle steps (when the frontier is relatively large). I used this principle when constructing my BFS algorithm.

```
$ srun -N 1 -n 1 ./bfs_omp_mpi ./graph/68m.graph
Graph stats:
  Edges: 68993773
  Nodes: 4847571
```

```
1x1: Average execution time of function bfs_omp_mpi is 66.612400 ms.
1x2: Average execution time of function bfs_omp_mpi is 146.273500 ms.
1x4: Average execution time of function bfs_omp_mpi is 296.347800 ms.
1x14: Average execution time of function bfs_omp_mpi is 1051.268500 ms.
1x28: libgomp: Thread creation failed: Resource temporarily unavailable

2x1: srun: Warning: can't run 1 processes on 2 nodes, setting nnodes to 1
2x2: Average execution time of function bfs_omp_mpi is 49.374700 ms.
2x4: Average execution time of function bfs_omp_mpi is 146.368900 ms.
2x14: Average execution time of function bfs_omp_mpi is 514.499900 ms.
2x28: Average execution time of function bfs_omp_mpi is 1044.177400 ms.
4x1: srun: Warning: can't run 1 processes on 4 nodes, setting nnodes to 1
4x2: srun: Warning: can't run 2 processes on 4 nodes, setting nnodes to 2
4x4: Average execution time of function bfs_omp_mpi is 51.953600 ms.
4x14: Average execution time of function bfs_omp_mpi is 294.713200 ms.
4x28: Average execution time of function bfs_omp_mpi is 522.284600 ms.
```

In the 1x28 testcase, I always got the error `libgomp: Thread creation failed: Resource temporarily unavailable` but was not sure why.

Throughout these test cases, I forgot that it is recommended that $P \times T <= 28$. Thus I ran some of the first few test cases again, but I would change `NUMBER_THREADS` so that $P \times T = 28$.

```
srun -N 1 -n 2 ./bfs_omp_mpi ./graph/68m.graph  (NUM_THREAD = 14)
1x2: Average execution time of function bfs_omp_mpi is 147.205400 ms.

srun -N 1 -n 4 ./bfs_omp_mpi ./graph/68m.graph (NUM_THREAD = 7)
Average execution time of function bfs_omp_mpi is 172.518400 ms.

srun -N 1 -n 14 ./bfs_omp_mpi ./graph/68m.graph (NUM_THREAD = 2)
Average execution time of function bfs_omp_mpi is 529.669100 ms.

srun -N 2 -n 2 ./bfs_omp_mpi ./graph/68m.graph
Average execution time of function bfs_omp_mpi is 69.595800 ms.

srun -N 2 -n 2 ./bfs_omp_mpi ./graph/68.graph
Average execution time of function bfs_omp_mpi is 108.558600 ms.

...
```

This improved test cases where the `nprocs` were greater than or equal to 4.

实验分析

From the experiment, we can see that increasing the number of procs per node increasingly slowed the mixed top bottom and bottum up bfs algorithm the `bfs_omp_mpi`. The most efficent BFS searchss occured when each node/computer only had one process. Each time it was still faster than the `bfs_omp` algorithm which only implemented a parallel BFS top down search.

Resources

https://stackoverflow.com/questions/42970700/openmp-dynamic-vs-guided-scheduling

https://crd.lbl.gov/assets/pubs_presos/mtaapbottomup2D.pdf