Cluster and Cloud Computing Assignment 1 - HPC Data Processing

Jie Jin, 652600, The University of Melbourne April 9, 2014

Problem

Implementing a simple, parallelized search application leveraging the HPC facility that searches a collection of text files. Count the number of times a given term (word/string) appears.

Approach

For this assignment, the parallelized search application is writen by C++, using MPI for data communication. Mainly implemented by blocking communication methods (i.e. MPI_Send & MPI_Recv), which is fairly simple to avoid deadlock.

Step 1: Setting up MPI environment

Snippet 1: Data type for file offsets.

```
    //Initialize MPI environment
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &worldSize);
    MPI_Comm_rank(MPI_COMM_WORLD, &worldRank);
    //create new MPI data type for partition the scope of each ↩
       processor
    MPI_Type_contiguous(2, MPI_LONG_LONG, &rangeT);
    MPI_Type_commit(&rangeT);
```

Step 2: Processor 0 partition tasks to others

Firstly, processor 0 estimates the working scope for each processor. For example, the size of input file is 200 bytes, using 4 processors to search the file. The application would create a array {0, 49, 0, 99, 0, 149, 0, 199} (in this stage it only initializes the

ending point).
Secondly, processor 0 would adjust the offsets in order to avoiding spilts a word, which will causes a potential bug. For example, when the word "Cloud" starts at the 48 bytes. If there is no adjustment for it, it would be a mismatch for both two processors. Because for processor 0, it reads "asdjklj sdsj ... Cl" and the processor 1 will get the part "oud ... iouweonj naksd jkljl". The strategy for this is obviously shifting the offset until finding a blank. So after this, the array might look like {0, 52, 53, 100, 101, 149, 150, 199}.
Finally, processor 0 sends the start and end point to others.

Snippet 2: Processor 0 record the start time & partition the task to other processors.

```
    if (worldRank == ROOT){
        // ROOT processor calculate offsets and partition them
        wtime   = MPI_Wtime();
        offsets = EstimateOffsets(fileSize, worldSize);
        if (worldSize > 1) {
            AdjustOffsets(offsets, fileName, worldRank);
            for (int i = 1 , j = 2; i != worldSize;
                 error = MPI_Send(&offsets[j], 1, rangeT, i++, TAG,↩
                     MPI_COMM_WORLD), j+=2);
            if(error != MPI_SUCCESS)
                ErrorMessage(error, worldRank, "MPI_Send()");
        }
        range[LBOUND] = offsets[0]; range[RBOUND] = offsets[1];
    } else {
        if (worldSize > 1) {
            error = MPI_Recv(&range[0], 1, rangeT, ROOT, TAG, ↩
               MPI_COMM_WORLD, &status);
            if(error != MPI_SUCCESS)
                ErrorMessage(error, worldRank, "MPI_Recv()");
        }
    }
```

Step 3: Each processor stripes & searches the splitted file

Each processor grabs their part from the original file, remove non-alphanumeric characters, change all uppercase characters to lowercase and then create a temporary file. Count the number of times a given term (word/string) appears. After that, delete temporary files.

Snippet 3: Main searching loop

```
// Strip the origional file and create smaller temporary files
StripCreateTmpFile(fileName, worldRank, range);
// Count the number of times a given term (word/string) appears
result = ScanTmpFile(worldRank, keyword);
// Delete temp files
DeleteTmpFile(worldRank);
```

Step 4: Send back results to Processor 0 and sum them up

Snippet 4: Main searching loop

```
if (worldSize > 1){
    if (worldRank == ROOT){
        for (int i = 1; i != worldSize; result += tmp) {
            error = MPI_Recv(&tmp, 1, MPI_LONG, i++, ↩
               MPI_ANY_TAG, MPI_COMM_WORLD, &status);
            if(error != MPI_SUCCESS)
                ErrorMessage(error, worldRank, "MPI_Recv()");
        }
        OutputResult(wtime, result, keyword, worldSize);
    } else {
        error = MPI_Send(&result, 1, MPI_LONG, ROOT, worldRank,↩
            MPI_COMM_WORLD);
        if(error != MPI_SUCCESS)
            ErrorMessage(error, worldRank, "MPI_Send()");
    }
} else {
    OutputResult(wtime, result, keyword, worldSize);
}
MPI_Finalize();
```

Usage

Compilation: mpicxx -o run ass1.cpp

Execution: mpirun -n NUM_PROCESSORS ./run -f FILE_NAME -k KEYWORD

Edward: qsub run.pbs

Results

As a result, two tables clearly shows cutting the execution duration in half when doubling the number of processors. Compare the result between Edward and my own computer. The performance of Edward is slightly slower than my computer when use 1 core and 2 cores doing the computation(2.0GHz vs 2.6GHz core). However when the number of processors up to 4, edward win the competition, this is becasue my computer only have 2 cores.
Due to the application is designed for high performance computing, it does not split temporary smaller files futher when input file is extreme large. This is because it will increase the extra runtime. I would suggest employing more processors rather than do that when there are enough processors avaliable on the HPC.
FYI: Because my computer only has two core and 4G memory, the results are probably not accurate. I didn't put the results on the rows of 8 cores and the colum of CCCdara-large.txt.

Table 1: Results on Edward

Core(s)	CCCdata-small.txt	CCCdata-medium.txt	CCCdata-large.txt
1	13.1998	133.735	673.411
2	6.81108	69.3442	346.891
4	3.61653	36.5963	187.987
8(1 node)	2.33417	20.9278	117.234
8(2 nodes)	2.03049	18.3832	109.765

Table 2: Results on my own computer

Core(s)	CCCdata-small.txt	CCCdata-medium.txt	CCCdata-large.txt
1	9.82777	99.5606	•
2	6.98189	69.2085	•
4	5.01782	43.4389	•
8(1 node)	•	•	•
8(2 nodes)	•	•	•