Parallel Programming 2nd Report
MPI

Course: CPD3 (1st Semester - 2012)

Professors
José Monteiro
José Carlos Campos Costa
Luís Miguel d'Ávila Silveira

Date of submission: 30 November 2012

Group #7

Name	Student Number
Pushparaj Motamari	77160
Anh Phuong Tran	77182
Dipesh Dugar Mitthalal	77184

1. Programming approach

1.1 Problem profiling

Profiling on sequential implementation shows that calculating distance between documents and cabinets accounts up to 86.5% of the workload (Table 1)

Task	Running time (s)	Percentage of total time (%)
Reading data into memory	15.4	4.6
Calculating cabinets' average value	29.2	8.7
Calculating distances	289.6	86.5
Writing data to file	0.6	0.2
Total	334.8	100

Table 1: Profiling of sequential version with input file "ex1M-100d.in"

Even though this number will vary between input files, it is safe to say that paralleling on calculating distances will benefits us the most.

1.2 Distribution of data

In order to minimize the communication between tasks, we decide to split the documents' data into smaller chunks in row-wise fashion. Each task will be in charge of roughly (Number of Documents/ Number of Tasks).

1.2.1 Local memory of each task

A 2D array of double, a so-called **Document** array, is used to store all related information of the documents that each task has. The Document array has the size of [Number of Documents in each task] * [Number of Subjects].

Another 2D array of double called **Cabinet** array to store all related information of the documents available that belongs to a Cabinet. Note that this Cabinet only has the partial value, as task doesn't know every document. Cabinet has the size of [Number of Cabinets]*[Number of Subjects + 1]. The last element of each row store the number of documents in that cabinet.

A 1D array named **Result** with the size equals to chunk size. **Result** stores information of which cabinet a document falls into.

1.3. Communication

The implementation requires the following communication between tasks:

+ Reading data from file and sending to different tasks
+ Aggregate each task's Cabinet's value to a global value
+ Agree on the value of repeat_flag. (True repeat_flag signals the program to continue to the next iteration)
+ Sending the Result array back to Master task to print out into file.

1.3.1 Reading and sending data

To optimize the file reading and data sending process, we use 2 different buffers to interleavingly read the file and send the data. Master task checks whether buffer 1 is being sent or not, and if not, reads in the first piece of data into buffer 1. When it finishes, Master task will send it right away using MPI_Isend() call. While buffer 1 is being sent, Master continues to read the file into buffer 2, and will send as soon as it finishes.

On the receiving size, to make sure tasks fully receive the data, we use blocking MPI_Recv() routine to receive the data from Master task.

Generally, reading data from file will be faster than sending data over the network, therefore using more than 2 buffers will not benefit us in term of speed.

1.3.2 Aggregate Cabinet's value

Since we want every task to have the same global value of the Cabinet, we use MPI_Allreduce() routine to sum up the Cabinet value and distribute it back to each tasks.

1.3.3 Repeat_flag

MPI_Allreduce with the operation MPI_MAX will return either 1 (continue to next iteration) or 0 (stop).

1.3.4 Gathering results

Since each task will have the Result array of different size, we employ MPI_Gatherv() routine to make sure all data is contiguously arrange in the final Result array in the Master task.

1.4 Synchronization and load balancing

In MPI, synchronization is implicitly handled by the use of blocking functions such as MPI_Bcast(), MPI_Send(),MPI_Reduce()...

We use distributed approach in block decomposition as stated in the theory class. Tasks might have different chunk sizes, but the difference is negligible.

Also, to facilitate a correct behavior in reading and sending data task, we change the owner of each block, so that Master task will be in charge of the last (also the largest) chunk. This makes sure Master task can read every other task's data into its memory before sending it.

2. Performance evaluation

2.1 Complexity analysis

We denote the number of documents as D, number of cabinets as C, and the number of Subjects as S. The number of tasks is p.

We analyze the complexity of calculating distances between documents and cabinets. For simplicity, we assume C*S = 1 (fixed Cabinet size)

Sequential algorithm complexity: $\Theta(D*C*S) = \Theta(D)$

Parallel algorithm complexity: $\Theta(D/p)$

Communication complexity: $\Theta(\log p)$ (p Cabinets to be gathered)

Overall complexity: $\Theta(D/p + \log p)$

2.2 Algorithm scalability

Isoefficiency analysis: $T(n,1) \geq CT_0(n,p)$.

Sequential time complexity: $T(n,1) = \Theta(D)$

The parallel overhead is Cabinet's gathering:

$$T_0(n,p) = \Theta(p(\log p)) \xrightarrow{large\ n} D \geq C*p(\log p)$$

Scalability function: $M(f(p))/p$

$$M(D) = D \rightarrow \frac{M(C*p\ logp)}{p} = \frac{C*p\ log\ p}{p} = C*\log p$$

2.3 Performance result

Experiment is carried on with input file ex1M-100d.in and ex100k-200d.in

Sequential time (s)	File	Number of hosts	Number of Tasks	Execution Time (s)	While loop time (s)	Speed up
169.7/ 129.6	ex1M	1	4	93.9	38.38	1.81
		4	4	125.6	31.6	1.35
		8	8	126	18.7	1.35
		16	16	119	9.2	1.43
		4	16	103.7	9.9	1.64
126.3 / 121.7	ex100k	1	4	37.7	33.2	3.35
		4	4	48.4	29.8	2.6
		8	8	31.76	15.4	3.98
		16	16	26.26	9.1	4.81
		4	16	28.67	10.07	4.4

Table 2: Experiment result with file ex1M-100d.in& ex100k-200d.in

*Results might not be accurate as environment variables affect the computational power of machines at any given time.

While loop time decreases when the number of tasks increases. This is expected, as the number of Documents to calculate decreases proportionally.

Reading file and sending data to nodes task accounts for a large portion of time. In some cases, it can take up to 90% of the execution time. (ex1M-100d.in - 4 hosts 16 tasks)

The higher the number of machines running, the smaller the execution time. We can conclude that the time to actually send the data is much higher than the overhead of openning new transfer between machines.

In a more computation-heavy instance, better speed up can be achieved, since we can make use of multiple processors at different machines.

2.3.1 MPI & OpenMP

A hybrid version is also included in the package. Instead of parallelize the whole while loop, we only parallelizes the computation-heavy part, which is the distance calculation.

3. Conclusion

MPI is a programming method based on Distributed Memory System. The same application is executed different machines. Most of the communication method is blocking, which mean synchronization is handled implicitly. Communication between tasks should be optimized to achieve good speed up.

Other than speed up, MPI also provides a solution for problem ofvery large scale, when all data cannot be kept in 1 machine.