Search for Intelligent Puzzles
Designing Parallel Algorithms

Joshua Church | JQC10

October 3rd, 2017

Overview

The task given was to design code that utilized the bag-of-tasks model of parallel programming. The problem given was to search through puzzles read in through an input file and determine whether or not each puzzle was solvable. Each puzzle formed a 5x5 game board, where each of the slots in the matrix could hold a peg. *X* was used to represent a peg and *0* was used to represent a hole. When a peg jumped over another peg, the jumped-over peg was removed and replaced with a *0*. However, not every puzzle was solvable.

In order to implement this functionality within the code base with arrays, *0* remained as the hole representation. *1* was used to represent a peg and *2* was used to represent that it was neither a hole or peg.

The solver was already provided, and the task was to speed up the process of solving using a client-server model in MPI. The process used will be discussed in the following sections.

Theoretical Analysis

Designing in parallel allows for a wide variety of approaches to a single problem. There are two major sending and receiving functions when using MPI: *MPI_Send / MPI_Recv* (blocking) and *MPI_Isend / MPI_Irecv* (non-blocking). The task given was to develop the codebase utilizing non-blocking techniques; therefore, *MPI_Isend* and *MPI_Irecv* were used. With a non-blocking send, the processor starts the sending process and immediately returns to do the next instruction. These means that the sending process is no longer concerned with the data stored in the buffer or whether the receiving process actually obtained the data. With a non-blocking receive, the processor checks to see if there is data to get at the time it is called. If not, it immediately returns and does its next instruction. With blocking sending and receiving functions, MPI will block the next set of instructions until the data has either been successfully sent and/or received by the other process. There are handlers for non-blocking processes which must be utilized to properly pass data between processes. Although there are many functions that MPI uses to mitigate this, *MPI_Wait* (blocking) and *MPI_Test* (non-blocking) will be discussed.

MPI_Wait and *MPI_Test* both follow the *MPI_Isend* or *MPI_Irecv* functions. *MPI_Wait* will wait until the send / receive request is successfully sent / received or until there is an error. If there is an error during the communication process, this can be handled with an *MPI_Status* object. *MPI_Test* will check for the request at that given time. Unless told otherwise (using looping or some other blocking process), it will go to the next instruction. Additional usage information will be discussed the **Methods and Techniques** section below.

There are pros and cons with blocking and non-blocking functionality. This allows the developer to try different implementations to test overall performance metrics. In this implementation, the server processor (denoted by rank 0) is developed to delegate jobs to each of the client

processors; however, there is idle time during the communication process. With a blocking implementation, the server would send a job, wait until the client has successfully received the data buffer, then send another job to the next available client processor. This can be represented as t_{idle}, which is the amount of idle time during the communication process. Although this ensures proper delivery to the client processor, there is idle time that the server could be using to perform a job. If non-blocking sending was used instead, the server could send off a job to the client(s) and return immediately to perform a task in the idle time.

The implications of using the server as another compute resource can be viable in many instances. This reduces the amount of idle time and wasted available resources. However, using the server can be detrimental in some cases. Assume that the complexity of the task increases, therefore taking more time for a processor to perform its task. The server has sent off a job (or set of jobs) to the client processors. While the server is waiting for a response from any client source, it performs a complex task as well. Now, assume that at the commencement of the task, the client processors send back messages. The communication process is now in deadlock until the server completes the task, which holds up the client resources. This type of scenario should be expected, and the developer should have an implementation to mitigate such issues. One of the ways to handle issues like this is to implement chunking.

Chunking is the process of sending batches (denoted by the developer) of jobs to the available processors. In order to properly implement this process, buffer sizes (address space in memory) must be properly allocated between processors. Memory is abundant in most cases, but the developer should aim to only use what is needed. Sending multiple batches of jobs to another processor means less message passing between processors.

In parallel computing, there is latency in two main forms: *startup* (t_s) and *per-word-transfer* (t_w). Startup latency derives from the time it takes for each of the available processors to start the communication process. Word transfer latency derives from the time it takes to send it word. Depending on the system architecture, results can change drastically depending on the latency and must be accounted for by the developer.

Methods and Techniques

This section will breakdown the methods and techniques implemented in the parallel codebase.

Tags

Tags are optional integer values that allow the programmer to specify which information to accept and/or receive during the MPI communication. The following are custom MPI tags used to relay information within the program.

```
const unsigned int TAG_SOLVE = 1;
const unsigned int TAG_SOLUTION = 2;
const unsigned int TAG_NO_SOLUTION = 3;
const unsigned int TAG_FINISHED = 4;
const unsigned int TAG_READY = 5;
```

TAG_SOLVE was sent by the server processor to the client processor(s). When the client processor received this tag, the client was instructed to solve whatever job was stored in the buffer.

TAG_SOLUTION was sent by the client processor(s) to the server processor after processing a job. When the server processor received this tag, the server was instructed to increment the total number of solutions found and add the buffer values to the output stream.

TAG_NO_SOLUTION was sent by the client processor(s) to the server processor after processing a job. The server processor received this tag when the client was unable to find a solution with the received job.

TAG_FINISHED was sent by the server to the client processor(s) once the total number of jobs has been finished. When the client(s) received this tag, the MPI communication should come to a halt.

TAG_READY was sent by the client(s) to the server processor once the client(s) was/were ready to receive the initial set of jobs.

Status

Status is an MPI object that handles the status of the communication request between processors.

```
MPI_Status status;
```

The use of status was used to extract the source from the sending process, along with an associated tag.

Request

Request is an MPI object that handles non-blocking communication. Whenever an *MPI_Isend* or *MPI_Irecv* process is used, request must be used to ensure proper delivering or receiving of a message.

```
MPI_Request request;
```

Approach to Problem

In this section, the solving code will not be discussed. To simplify this, the solving code will be called *solver*. It should be assumed that standard C / C++ libraries and header files have been included. It should also be assumed that the proper development environment has been established to compile and execute the program.

Inside of the *main* file, there are three main functions: *int main*, *void server*, and *void client*.

Main

The *main* function is handles the initial MPI setup. The communication size and rank are established here. Once the process begins, processor rank 0 will execute the *server* code, while rank 1 to p-1 will execute the *client* code. A timer keeps up with the MPI process and is logged into the output stream. Once the program finishes, *MPI_Finalize* closes and cleans up the communication process.

Server

Inside the server function, there is a continuous loop that continues to execute until all the games have been handled. Starting the process, the server listens for a *TAG_READY, TAG_SOLUTION,* or *TAG_NO_SOLUTION* from the client using *MPI_Irecv*. Since this is non-blocking, if nothing is received, it solves a puzzle using *solver*. If a solution was found, the output is added to the stream, along with incrementing the total number of solutions. *MPI_Test* runs again to see if anything has been sent by a client processor. This continues until something is received from the client. If something is received, *status* is used to check the source and tag. If the tag is equal to *TAG_SOLUTION*, the server adds the contents of the buffer to the output stream and increments the total number of solutions. Otherwise, the server sends another game to the client it just received a message from. While waiting for the client to receive, it performs another job. This process continues until all games are done or an MPI error occurs. Once all

jobs are handled, the server sends of *TAG_FINISHED* to the client processors to close the MPI communication and shutdown.

Client

Inside the client function, there is a continuous loop that continues to execute until the *TAG_FINISHED* has been received. When the client is ready, it sends the *TAG_READY* to the server to receive its first job. It must now use *MPI_Wait* because it **must** wait until the request has been delivered. Upon successful delivery, the client sits in idle until a job has been received. *MPI_Wait* **must** be used because the purpose of the client is to solely perform jobs from the server. Once a job is received, the client uses *solver* to attempt to find a solution from the puzzle. If a solution is found, it sends back the output stream of the solution along with a *TAG_SOLUTION*. If no solution is found, it sends back *TAG_NO_SOLUTION*.

Performance Metrics

Hardware

The HPC system hardware used to measure the performance metrics shown in this section is labeled in Table 1.

Table 1: Engineering Research and Development Center (ERDC) Utility Server Hardware Specifications [1]

	Login Nodes	Compute Nodes
Total Nodes	2	44
Operating System	RHEL	RHEL
Cores/Node	16	16
Core Type	AMD Opteron 6134 Magny-Cours (x2)	AMD Opteron 6134 Magny-Cours (x2)
Core Speed	2.3 GHz	2.3 GHz
GPU Type	N/A	N/A
Memory/Node	64 GB	128 GB
Accessible Memory/Node	62 GB	126 GB
Memory Model	Shared on node	Shared on node. Distributed across cluster.
Interconnect Type	QDR Infiniband	QDR Infiniband

Easy Sample Data Metrics

The file *easy_sample.dat* is a data file provided to essentially help debug the program. Within the file, there are 1000 games to solve that are categorized as easy. The following tables show the performance metrics with *easy_sample.dat*.

Table 2a: Single processor (serial)

Run #	Runtime (s)
1	0.0770159
2	0.0768569
3	0.0769310
4	0.0789909
5	0.0766289
Average	0.07728472

Table 2b: 2 processors (1 server, 1 client)

Run #	Runtime (s)
1	0.108760
2	0.104412
3	0.118517
4	0.105783
5	0.105955
Average	0.1086854

Table 2c: 4 processors (1 server, 3 clients)

Run #	Runtime (s)
1	0.145572
2	0.140153
3	0.162649
4	0.141375
5	0.154744
Average	0.1488986

Table 2d: 8 processors (1 server, 7 clients)

Run #	Runtime (s)
1	0.145828
2	0.155922
3	0.139575
4	0.157892
5	0.147386
Average	0.1493206

Table 2e: 16 processors (1 server, 15 clients)

Run #	Runtime (s)
1	0.143914
2	0.140249
3	0.276754
4	0.146964
5	0.303111
Average	0.2021984

Table 2f: 32 processors (1 server, 31 clients)

Run #	Runtime (s)
1	0.385356
2	0.401548
3	0.268165
4	0.417353
5	0.413749
Average	0.3772342

Table 2g: Comparisons against serial running easy_sample.dat

# Processors	Average Runtime (s)	Actual Speedup	Speedup?
1	0.07728472	N/A	N/A
2	0.1086854	-28.89	Slower
4	0.1488986	-48.09	Slower
8	0.1493206	-48.24	Slower
16	0.2021984	-61.77	Slower
32	0.3772342	-79.51	Slower

Synthesis

Amdahl's Law Parallel Speedup

$S(n) = 1 / [(1-P) + P/n]$

S(n): Theoretical speedup
P: Part of algorithm in parallel
n: Number of processors

When designing parallel algorithms, it is expected that parallel implementation should be faster than its serial counterpart. The expected speedup of program using Amdahl's Law is as follows.

*Assuming 80% can be run in parallel (20% serial - reading / writing files, setting up, etc…)

S(2) = 1.67 times faster
S(4) = 2.50 times faster
S(8) = 3.33 times faster
S(16) = 4.00 times faster
S(32) = 4.44 times faster

However, the actual results show that there was a decrease in performance speed as the number of processors increased. This is because the latency between the output stream, transfer time, and startup time outweighed the time to solve the less-complex problems. Chunking is the better solution in this scenario because it reduces the amount of transfer time between the processors.

Hard Sample Data Metrics

The file *hard_sample.dat* is a data file provided to test the program for efficiently. Within the file, there are 1000 games to solve that are categorized as hard. The following tables show the performance metrics.

Table 3a: Single processor (serial)

Run #	Runtime (s)
1	232.365
2	231.906
3	231.947
4	231.911
5	231.881
Average	232.002

Table 3b: 2 processors (1 server, 1 client)

Run #	Runtime (s)
1	172.527
2	172.923
3	169.141
4	172.907
5	172.827
Average	172.065

Table 3c: 4 processors (1 server, 3 clients)

Run #	Runtime (s)
1	111.169
2	106.631
3	108.699
4	112.894
5	111.297
Average	110.138

Table 3d: 8 processors (1 server, 7 clients)

Run #	Runtime (s)
1	72.1904
2	60.6327
3	65.9642
4	70.0581
5	64.4033
Average	66.64974

Table 3e: 16 processors (1 server, 15 clients)

Run #	Runtime (s)
1	30.892
2	37.2464
3	44.4857
4	32.1046
5	38.1182
Average	36.56938

Table 3f: 32 processors (1 server, 31 clients)

Run #	Runtime (s)
1	36.144
2	30.534
3	30.6935
4	32.6657
5	26.3112
Average	31.26968

Table 3g: Comparisons against serial running hard_sample.dat

# Processors	Average Runtime (s)	Speedup Over Serial (%)	Speedup?
1	232.002	N/A	N/A
2	172.065	+25.83	Faster
4	110.138	+52.53	Faster
8	66.64974	+71.27	Faster
16	36.56938	+84.24	Faster
32	31.26968	+86.52	Faster

Synthesis

Amdahl's Law Parallel Speedup

$S(n) = 1 / [(1-P) + P/n]$

S(n): Theoretical speedup
P: Part of algorithm in parallel
n: Number of processors

When designing parallel algorithms, it is expected that parallel implementation should be faster than its serial counterpart. The expected speedup of program using Amdahl's Law is as follows.

*Assuming 80% can be run in parallel (20% reading / writing files, setting up, etc…)

S(2) = 1.67 times faster
S(4) = 2.50 times faster
S(8) = 3.33 times faster
S(16) = 4.00 times faster
S(32) = 4.44 times faster

The actual results on the more complex problems resulted in much faster speedup. The complexity of the problems resulted in the drastic difference over the previous metrics in the *easy_sample.dat* file. This is because the processors had to spend additional time solving, so the latency was masked behind the complexity.

Conclusion

Parallel programming has many different approaches and styles. It can be challenging to find the optimal solution to any one problem given the variety of approaches. While blocking functionality does make the process simpler to solve, it does leave many resources (such as time and computational power) wasted. Non-blocking communication proves to be useful, but many checks must be accounted for to ensure proper delivery and handling of messages. There were many variables that were not thought about until dwelling deeper into the problem. Overall, the program helped relay the importance of parallel programming in regards to program efficiency, while showing many of the complexities that come with it.

References

[1] https://erdc.hpc.mil/hardware/index.html