Parallel and distributed computing
Lab 3 Report
Yukti Abrol(ya246), Rohit Jain(rj288)

We started by taking three files from Lab 2: mw_api.c, mw_api.h, and sample_q2.c. mw_api.h was the header file for mw_api.c. mw_api.c and mw_api.h are built on top of api.c and api.h. For this we made changes to the API. Additionally, we created queue.h and queue.c, which contains the structures and methods necessary for a queue data structure containing work_units.

Brief Approach

Part 1

We decided to follow the heartbeat approach for detecting master failure. The worker needs to send heartbeat at regular intervals. The way it does that is by returning a partial result to the worker processor whenever it has been executing for more than the timeout period. The worker then holds on to the partial result and resumes the computation. The master is maintaining a last seen timeout for each processor that is updated every time a heartbeat is received.

The master maintains a work queue and whenever a worker dies, the work assigned to that worker is again added to the work queue. The master then assigns it to another worker whenever it asks for work.

There are some cases where master detects a false timeout on the worker. To handle those cases it sends a terminate to every processor after it's status is changed to dead.

Part 2

Failure Detection & Handling

For this case we use the result that the master sends to worker as the heartbeat. Each worker has a timeout set for the master. Once the worker detects the master is dead, it sends out a message to all the other workers. Every worker then uses the id of the sources it receives messages from to gain the information about what all processes are alive. If any of the message was received from a process that had a lower rank than the process itself, the new master id is set to that. This way the new master id is the process with the lowest rank. Once all the processes have been informed and new master id is set, the new master is initialized using function named master.

State Resuming

We want the master to resume from the place it left off. To enable this we are doing checkpointing. We mainly checkpoint 3 things.

1. All the work
2. Result Computed so far
3. Work id's of the chunks of work that have been already processed.

Master writes all the work to a file when it is originally initialized, so that if it is the resumed master (function called from the worker) then it can read from the file. The results, along with the work id are written every time they are received from the workers. Whenever the master function starts, it reads in all the results and the processed work id's from the file. So that when it assigns work it can skip the work id's that have already been calculated.

Issues and drawbacks

Parallel and distributed computing
Lab 3 Report
Yukti Abrol(ya246), Rohit Jain(rj288)

#1:

We faced some issues with the master timeouts. When the workers detect master failure, we make an assumption that work should finish before the timeout period in the worst case. eg. when there is a single piece of work and it takes way too long to compute (more the the timeout), other workers will assume master dead.

Possible Solution

Use a fixed heartbeat like we have used for worker.

#2

We are able to switch single master and do the computation if the second master is guaranteed not to fail. But, if master uses the same F_send function as the worker then the probability of backup master going down is also really high. So in most cases, all the masters end up failing as well.

Possible Solution

We try to handle this by allowing the initialization of master with a failure probability. This works sometimes but most of the times all the workers end up failing.

#3

There is a timeout syncing issue to let all the workers communicate when the master dies. If the message from the lowest rank processor doesn't reach the other processors in time some other process will take over as the master. We tried to handle it by keeping the communication timeout sufficiently large but it then slows down the progress and is really slow.

Code details

Part 1:

mw_api.h : The header file contains the general struct declaration for work and result. It also contains the headers for the create function, compile function, and compute function. The create function creates work. The compile function compiles the results together to form a final result. The compute function finds the result of the inputted work. The work size and result size are also declared. Additionally, the MW_Run and testing functions are declared.

The serialization and deserialization functions for work and results are declared. Prior to sending each chunk of work, master serializes the work, and then uses MPI_Send to each worker. Since we also need to have workers send heartbeats, we need to interrupt their work, so we declared a few helper methods: get_result_state, work_first, combine_partial_results, get_result_object, and reinit.

Parallel and distributed computing
Lab 3 Report
Yukti Abrol(ya246), Rohit Jain(rj288)

mw_api.c : We declare the random_fail function. The F_Send function is declared. We define the structure of a processor, which contains the processor ID, the status, the work ID, whether it is master, when it was last seen, and the number of times it missed sending a heartbeat.

The **processor struct** helps the master keep track of status of each processor, and the last seen time for each processor.

The init_processor function takes the processor ID, status, work ID, and whether it is master, and initializes the processor with these values. The time last seen is initialized to the current time and the missed_count is initialized at 0. The initialized processor is returned.

The get_idle_processor function takes an array of all the processors and checks if any worker processors have a status of idle. It then returns the idle processor's ID.

The is_any_processor_busy function takes an array of all the processors and checks if the status of the worker processors is not idle or dead. It then returns true if there is any processor that is busy.

The is_any_processor_alive function takes an array of all the processors and checks if the status of all workers is dead. It returns true if there is any processor that is not dead.

The terminate_workers function takes an array of all the processors and the size of the work. If a worker is still alive, empty work is sent to it with a TERMINATE tag. The status of the worker is changed to dead.

The get_lost_work function looks for all the workers who are alive and not idle. It adds a missed_count to the workers who were last seen longer than the timeout, and if the missed_count is greater than one, we assume that the worker is dead. We send work with a TERMINATE tag and change the worker's status to dead. The dead worker's work is then reinitialized using the reinit function and this is put into a node. The node is enqueued on the work queue so that it can be assigned to the next idle processor.

The send_heartbeat function takes the worker id and sends a character to master to let it know that it is still alive.

The send_work function takes the work id, work queue, mw_api_spec, array of all the processors, and the processed work queue. It uses the inputs as the arguments in MPI_Send and writes the serialized work to file. The processor with the inputted work id is initialized and the work queue is dequeued.

The master function puts the entire work pool in nodes on the **work queue** after creating the work. Chunks of work is then sent to all the workers in round robin fashion. The master then waits for workers to send results back. After that, whenever a process finishes its work, new work is assigned from the work queue. This is a change from the last lab where we used to send out all the work initially. Master also checks if there are dead workers and puts their work back on the work queue. It uses MPI_Iprobe to state that it is still alive. If the master is assumed to be dead, its status is changed and the process is terminated. The master is still responsible for receiving results and it places them on an array upon deserialization. When all the results have been received, the master terminates all the workers and compiles the final result before destroying both queues. The master also updates the processors' last seen time when it receives a heartbeat.

The send_all_processes uses MPI_ISend to send a tag to all the processes.

Parallel and distributed computing
Lab 3 Report
Yukti Abrol(ya246), Rohit Jain(rj288)

We define the MW_Run function. If it is the master process, it calls the master function. Otherwise, it checks if the master is alive or not. If it is not, it informs all the other workers that the master is dead, so that the worker with the smallest ID will become the new master. If the worker has received work, it processes the work after deserializing it. Since it has to send a heartbeat and may not have finished computing the work, it keeps temporary results and combines the partial results. When all the work has been computed, it serializes the result and sends it to master.

The F_Send function uses the random_fail function to determine if the process has failed. If it has, the process ends. Otherwise, a normal MPI_Isend is done.

The random_fail function takes the processor ID and threshold value. It calculates a random number. If it is above the threshold value, the process will fail.

New Files

queue.h: This is the header file for queue.c. It contains the structure for each node in the queue and the queue itself. It also includes the function declarations for all the needed methods. This file and it's associated c file are primarily for the work_unit and organizing the work.

queue.c: The queue_create function creates a queue and initializes the nodes to be null as there are no nodes added to the queue at this point.

The queue_empty function takes the work queue and checks if it is empty.

The dequeue function takes the work queue and, after checking if there are nodes that can be dequeued, removes the head and updates the pointers for the other nodes in the queue that would be affected by the removal.

The queue_destroy function takes the queue and dequeues the first node until the queue is empty. Then it frees the memory that was allocated for the queue.

The get_work_node function takes the work_unit and work ID to create a work_node. The information obtained from work_unit and work ID is put into the work_node. The pointer to the next node is left as NULL.

The dequeue_by_id function takes the queue and the work ID of the node that should be dequeued. After checking if the queue exists and has at least one node in it, the method goes through each node and checks its work ID. If the work ID of the node is the same as the argument, it is removed

The enqueue function takes the queue and a new node. If there are no nodes in the queue, it makes the head and rear the new node. Otherwise, it points the previous rear to the new node and makes the new node the rear.

The print_queue function takes the queue and prints the work IDs of each node starting from the front.

sample_q2.c: This was our user defined file. It contained the methods required to obtain the list of factors of a large number. We first defined the work and result structures.

Parallel and distributed computing
Lab 3 Report
Yukti Abrol(ya246), Rohit Jain(rj288)

The do_work function is the major change since it is responsible for sending the heartbeat. It maintains a timeout and whenever the computation has been running for more than the timeout period it returns with the partial result to the worker. It checks if each value given in work is a factor of the input large number. It retains the numbers that are factors.

The work structure included the large number and the starting and ending number for the chunk to check. We added the temp number. The result structure included the pointer to an array of numbers that were factors. We added the result state and length of factors.

The serialize_result function converted the result array into a byte stream of unsigned characters. The serialize_work function converted the work array into a byte stream of unsigned characters. The deserialize_result function converted the result byte stream of unsigned characters back into the result array. The deserialize_work result converted the work byte stream of unsigned characters back into the work array.

The create_work function converted the large number input from a long to a mpz_t so that the square root function could be applied to the number. The result was then converted back into a long. An array of integers from 1 to the square root of the large input was created. This returns an array of work_units.

The process_results function compiled the list of result factors into one array.

The main function instantiates the API specification structure and passes it to the MW_Run function to run it.

We added a few new functions in the sample_q2.c file.

The get_result_state function takes a result unit and returns its state.

The get_work_first function takes a work_unit and returns the first number of the work_unit.

The reinit_work function takes a work_unit and changes the first number to the temp number of that work_unit.

The get_result_object function creates a result_unit to its initialized state and returns it. The state of the result unit is considered to be PARTIAL until it is complete.

The combine_partial_results function takes two result_units and creates a new result_unit which is a combination of the inputs. The new result_unit is returned.

run.sh is a script file that we used to run these files.

Part 2:

Code details

In addition to the previous files, we created created fileio.h and fileio.c, which contain the methods to read and write to file.

Parallel and distributed computing
Lab 3 Report
Yukti Abrol(ya246), Rohit Jain(rj288)

fileio.h: This is the header file for fileio.c. It contains the declarations for the functions: write_workList, write_result_list, write_completed_workID, read_workList, create_blank_files, read_result_list, and write_workList_length.

fileio.c: The create_blank_files function creates or overwrites a textfile named workList and a textfile named result. This is used so that write_workList and write_result_list will not append to a file created in the previous run of the entire program.

The write_workList function takes the serialized work and prints the character length and serialized work at the end of the workList text file.

The write_result_list function takes the serialized result, character length, and work id. It prints these values in the result text file.

The write_workList_length function takes the length and adds it to the workList textfile.

The read_work_list function scans the workList text file. It deserializes the work and places it in an array of work_units, which it then returns.

The read_result_list function scans the result text file. It then filters out the results that have been processed and returns an array of result_units that have been deserialized.

The write_completed_workID function prints the work that has been completed in the completedWordID text file.