Lab 2: Master-worker

James Kizer & Sean Herman

Overview

We implemented a basic Master-Worker design pattern for computing the number of divisors for numbers using C++, OpenMPI library for message passing functionality, and GMP for large number support.

In our implementation, the Master process sets up the working set and distributes individual work tasks to Worker processes. Worker processes receive work, compute results, and send these results back to the Master process. The Master process aggregates these results, and reports the combined results after all work has been distributed and all workers have finished sending results.

Prerequisites and Setup

- C++11 / C++0x
- OpenMPI
- GMP version 5 or greater

The divisors processing application may compiled using the following command.

```
make divisors_app
```

The divisors_app binary make then be executed directly with mpirun:

```
mpirun -n procs [-hostfile <file>] divisors_app target [work_size]
```

API & Implementation

Master-Worker API Specification

Work and Results

Files

- Work.hpp
- Result.hpp

Generic Work.hpp and Results.hpp define abstract classes for work and result items in the Master-Worker pattern.

```
class Work {
public:
      virtual Result *compute() = 0;
      virtual std::string *serialize() = 0;
…
}

class Result {
public:
  virtual std::string *serialize() = 0;
…
}
```

End users of this API spec must provide concrete implementations of these Work and Result classes. The API also provides an abstract class MW_API, which includes functions to fulfill the following tasks:

Master Worker App

Files

- MW_API.hpp
- MW_Master.hpp MW_Master.cpp
- MW_Worker.hpp MW_Worker.cpp

MW_API declares a virtual function which returns the full universe of work.

```
virtual std::list<Work *> *work() = 0;
```

The results method outputs the results list to the user (e.g., on stdout).

```
virtual int results(std::list<Result *> *) = 0;
```

Two deserializer methods translate message buffers (represented as strings) into Work and Result objects.

```
virtual Work *workDeserializer(const std::string &) = 0;
virtual Result *resultDeserializer(const std::string &) = 0;
```

API Implementation

MW_API

Files

- MW_API.hpp MW_API.cpp
- gmp_factors_API.cpp

The main function for our program is included in gmp_factors_API.cpp. This portion of the program creates a DivisorApplication object (a subclass of MW_API). A concrete implementation of the MW_API class across DivisorApplication.hpp and DivisorApplication.cpp. The DivisorApplication object is instantiated 2 strings, the divisorString and workSizeString (optional). These parameters control the target value for the divisors calculations, and the amount of work that will be transferred to a worker at one time, respectively.

For example, the constructor below translates the first string as the divisor target, and the second string as the work size.

```
DivisorApplication(std::string &, std::string &);
```

Work and Results

Files

- DivisorWork.hpp DivisorWork.cpp
- DivisorResult.hpp DivisorResult.cpp

The work() method creates a list, consisting of a series of DivisorWork objects. The contents of DivisorWork objects will vary depending on the divisor and work size parameters passed to the App object.

Generally speaking, the DivisorWork defines work as a collection of 3 values: the divisor, the firstValueToTest, and count.

```
class DivisorWork : public Work {
        ...
        mpz_class dividend;
        mpz_class firstValueToTest;
        mpz_class count;
        …
    };
```

When compute() is called on a DivisorWork object, all the values that divide the divisor (confusing word choice, we realize) equally between firstValueToTest and firstValueToTest+count are first added to a list, then returned as a DivisorResult object.

```
    for(mpz_class i=0; i<count; i++)
        {
           mpz_class ithValue = firstValueToTest + i;
           // std::cout<<"Testing " << ithValue << std::endl;
           if(dividend % ithValue == 0)
          …
    }
    ...
    return new DivisorResult(divisors);
```

MW_Run

Files

- MW_API.hpp MW_API.cpp
- MW_Process.hpp
- MW_Master.hpp MW_Master.cpp
- MW_Worker.hpp MW_Worker.cpp

MW_Run depends on 3 classes created to encapsulate the Master process and Worker processes. MW_Process defines some static variables relevant for both processes, and could serve to consolidate the similar methods (send/receive) on the more generic base class in a future version, to allow for some clearer program layout in MW_Run (e.g., use of the base class as the variable type).

MW_Worker objects hold the process id, world size, and lists for the remaining work, the results, and the available Worker processes

```
    std::list<Work *> *workToDo;
    std::list<Result *> *results;
    std::list<int> *workers;
```

Master objects also currently hold an instance of the MW_API class as well, in order to access the Work and Result deserializer methods. We would like to move these deserializers elsewhere in a future revision of the API. Masters also implement methods for delivering work to Workers, receiving results from Workers, and sending the done signal to workers.

```
    virtual void send_done();
    virtual void send_one(int worker_id);
    virtual void receive_result();
```

MW_Worker objects have corresponding methods to receive Work messages from Master and send Result messages back to Master.

```
    virtual int receiveWork();
    virtual void sendResults();
```

Internally, Worker objects maintain a list of pending work (workToDo) and a list of results.

```
    std::list<Work *> *workToDo;
    std::list<Result *> *results;
```

MW_Run initializes and finalizes the MPI library, and creates a message loop between the Master and Worker processes. When the Master has work and there are workers in its list of available workers (`workers`), then it gets a new Work object, serializes it, and sends this message to a single worker. When the Master either hsa work, but no workers are available (noWorkersHasWork), or there are no workers and no Work (noWorkersNoWork), then Master simply calls MPI Recv, and waits for a Result message from any Worker.

When no Work remains on Master and all the Workers are available, then Master sends a special DONE message to every Worker. Finally, the Master process exits the loop.

Meanwhile, workers continuously call MPI Recv, waiting for messages from the Master process. When these messages have the WORK tag, then the Worker deserializes the message into a Work object, computes the Result object, serializes the result, and sends this message back to the Master process.

```
    while (1) {
        message_tag = proc->receiveWork();  // MPI_Recv
        if (message_tag == MW_Process::WORK_TAG) {
            // Deserialize to Work, compute, and MPI_Send Result
        }
    }
```

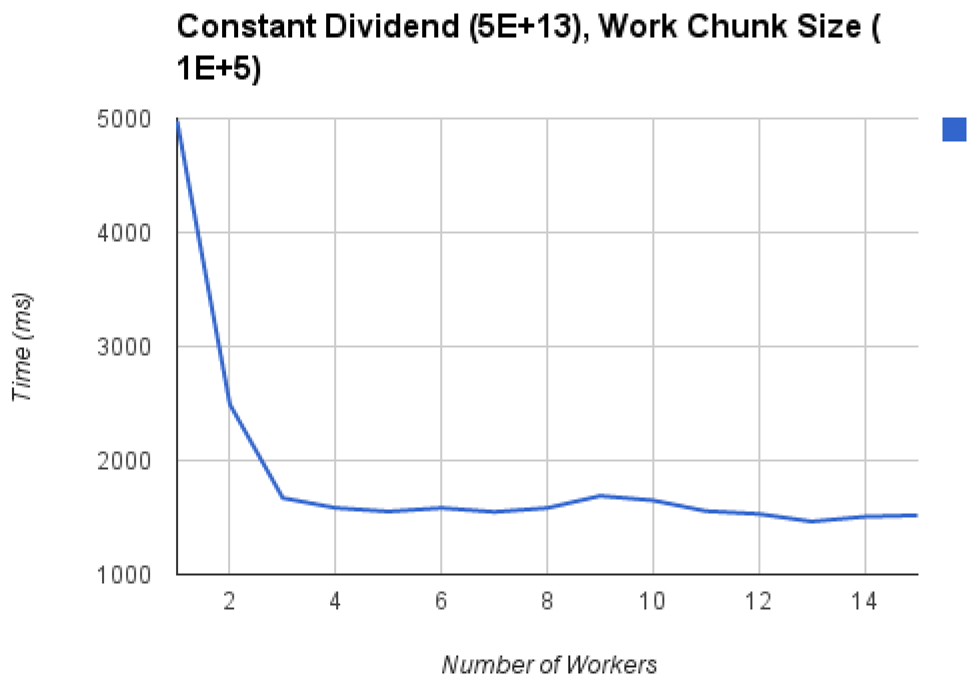
When a Worker process receives a message with the DONE tag, it simply breaks out of its infinite loop, and exits the program.

Performance Report

We wanted to see how the system performed under a variety of conditions. Ultimately we chose 3 variables to test:

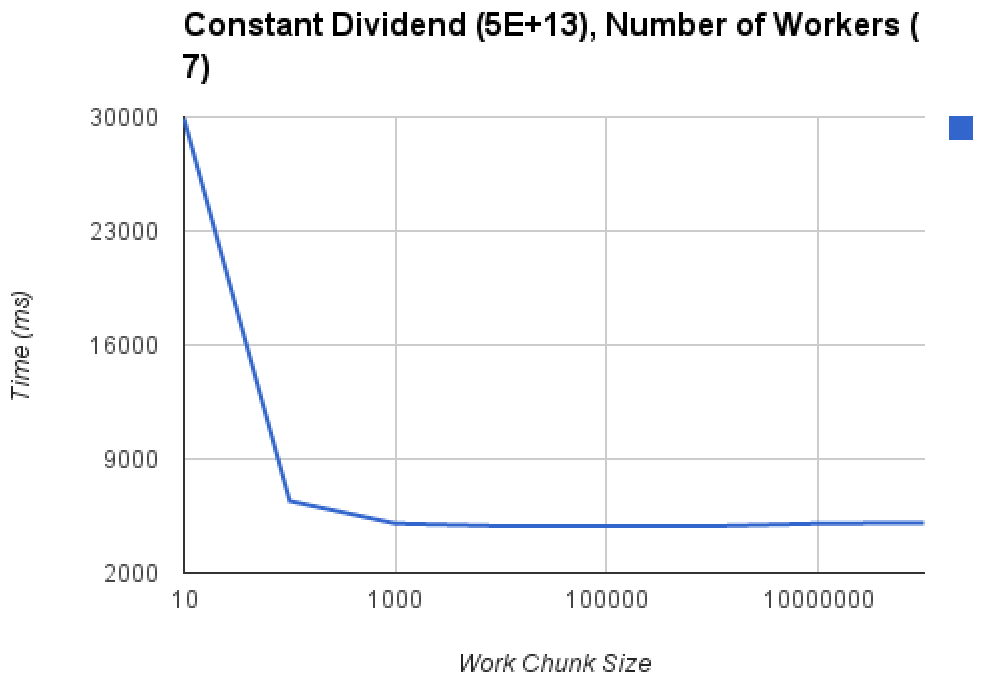
1. Number of workers
2. Work chunk size
3. Number of divisions, by varying the dividend.

For the first test, we tested 1 master, n workers, where n ranged from 1..15, under constant conditions where dividend = 5E+13 and work chunk size = 1E+5. Knowing results from lab 1, we figured that performance should peak around 7 or 8 workers. See chart below:



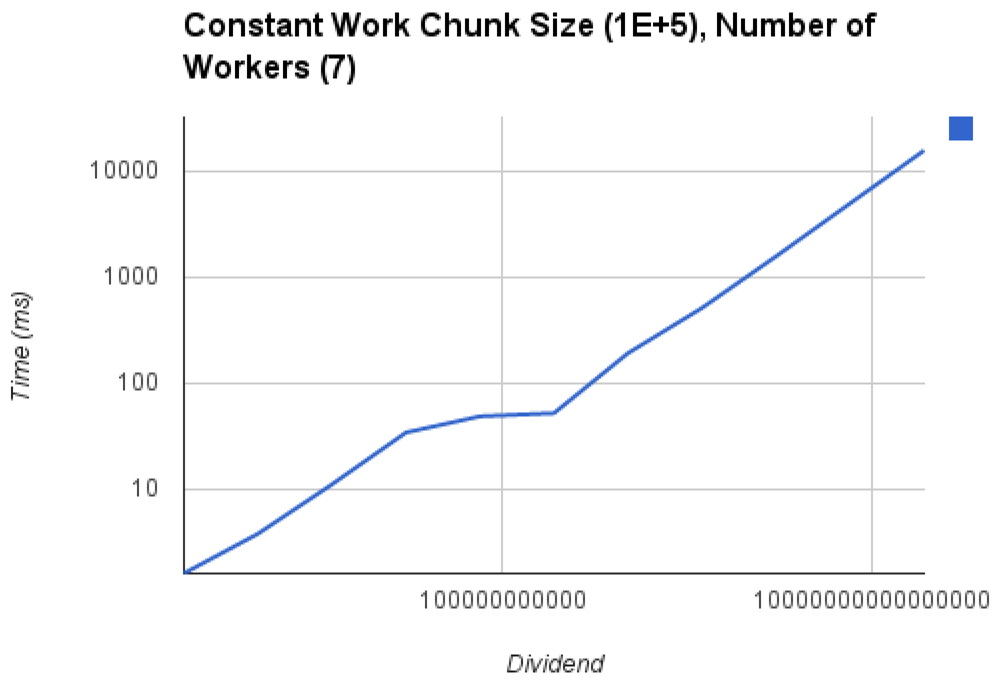
We were somewhat surprised that our performance topped out at 3 workers. For follow up, we would probably increase the Dividend and Work Chunk Size.

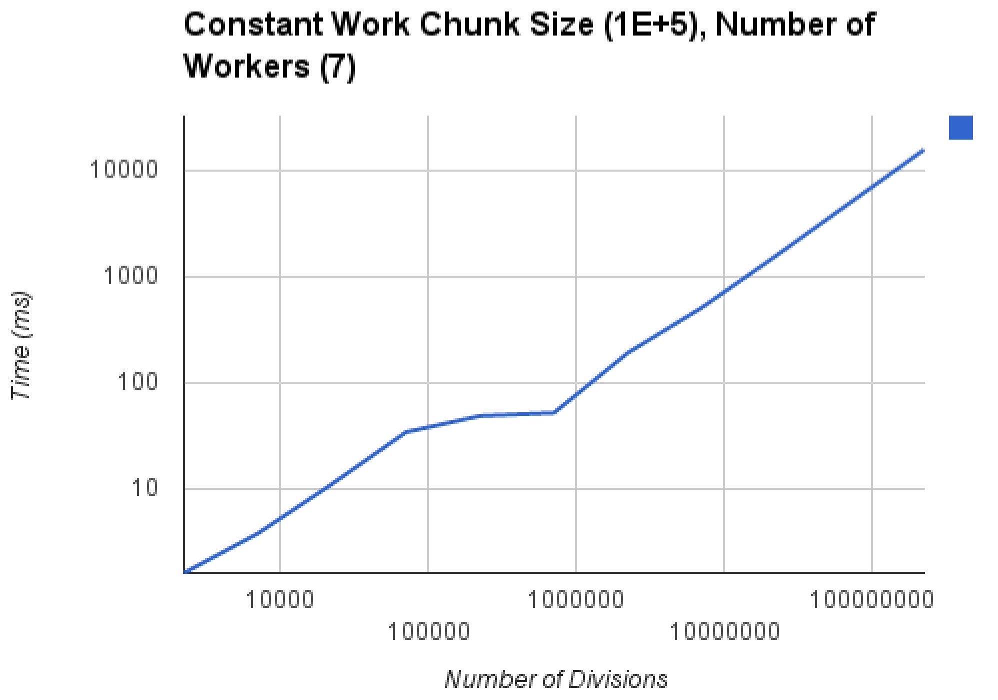
For the second test, we tested the work chunk size, where the size ranged from 10 to 10E+8, under constant conditions where dividend = 5E+13 and number of workers = 7. We expected smaller chunks of work to take longer than larger chunks of work due to the decreased overhead of communicating larger chunks of work.



The results of this test generally met our expectations in terms of shape. However, we didn't expect it to level off as soon as it did.

For the third test, we tested the various dividends (and thus, total work, where the dividend ranged from 5E+6 to 5E+16, under constant conditions where work chunk size = 1E+5 and number of workers = 7. We expect to see a linear relationship between time and amount of work.





As you can see from the above charts, the data generally shows a linear relationship between the amount of work and time. There are a few data points that seem to point to a nonlinear relationship. Looking back at the raw data, this is the point where the number of chunks of work goes from 1 to 3 to 8. Thus, there is a strong linear relationship before this point because all the work is computed on a single worker. During this transition range, we see more workers being activated and we see nonlinear behavior. After this point, all the workers are busy, so we continue to see a linear relationship.