Dynamic Load Balancing for Networked Resources using Work Stealing and Job Splitting

Yangxing Chen
Abraham Mukwana
Tim Parker

INTRODUCTION

As computers get faster and have more memory, the complexity of the tasks performed by computers has also grown. Analyzing water flow through a turbine using fluid dynamics, analyzing geological data to find oil fields, testing structural forces for automotive crashes, and testing new proteins to cure cancer all require immense computational power. If these were performed on a single processor they would take months, if not years to complete. The emergence of parallel processor computer technology has resulted in increased interest as well as research in developing efficient dynamic load balancing techniques. In a parallel processor environment where several jobs or sub-jobs can be handled simultaneously we can imagine parallel scheduling as a two step process, the first of which determines which jobs are to be allocated to which machines and the second step is to determine the sequence of jobs allocated to each machine. Although problems with due date related performance measures have been studied in many research articles, not much progress has been made in minimizing total tardiness in parallel machine scheduling problems, except for the case of identical parallel machines [1]. This leads to lower efficiency in throughput as it takes a significant amount of time to schedule the jobs to different processors as well as to reconcile the data once the computations are complete. Also since it is not easy to obtain optimal solutions for parallel machine tardiness problems of a practical size, researchers have focused on development of heuristic algorithms [2]. For an identical processor problem in which the job can be split arbitrarily into sub-jobs of continuous units Xing and Zhang propose a heuristic algorithm for the objective of minimizing makespan and analyze the worst case performance of the algorithm [3]. There are many methods for dynamic load balancing, methods of deciding job priority, and splitting a job into smaller jobs for parallelization. We are investigating a method to combine two of these into a single method to improve parallelization and increase throughput. The job splitting method is designed for peer to peer processors in a single machine, and not for networks. The job splitting portion of the paper focuses on the problem of scheduling independent jobs on m identical parallel machines with the objective of minimizing total tardiness of the jobs considering a job splitting property. This portion of the paper, regarding job splitting, is implemented as a mixed integer programming model and solved by a Tabu search algorithm. The networked load balancing method creates a lot of jobs but does not change any aspects of them. Our method integrates the job splitting method for increased parallelization and has multiple machines connecting to each other through the network in order to move idle jobs to idle processors. This will decrease the size of each job, allowing faster processing and higher throughput, and also increase the parallelization by increasing the number of processors available in the pool. By using mutex for critical regions, process communication through networking, multiple job storage, and task scheduling, we believe we can improve the performance of the method. We need to look into efficient ways to move jobs between machines, and efficient ways of splitting a job into smaller jobs. We are concentrating on merging the two methods, but will not have a means to test a large scale version like the one the papers authors did. This project covers aspects that were learned in the class such as scheduling algorithms, process synchronization through the use of mutex locks and semaphores and job scheduling on processors which we implement using a variety of algorithms such as FIFO and LIFO. We believe that our approach is better than currently available techniques because we will be

combining job splitting along with scheduling to multiple processors whereas as current techniques have focused on only one of the two aspects. As noted above we had to minimize our sample space as we do not have access to unlimited resources. We thus limited our testing to five computers on a network each of which could join the network arbitrarily, poll for available jobs in the system and execute them.

Basis and Review

Jobs often spawn multiple sub jobs. Waiting for all of these to complete on one processor takes a long time, so a method to split the jobs into smaller pieces and send them to other machines is investigated. We reviewed several papers related to load balancing between processors and of job splitting, but we did not see any papers that combined them. The scalable work stealing paper uses three different methods to generate jobs: Multiresolution ADaptive NumErical Scientific Simulation (MADNESS) tree creation kernel, Unbalanced Tree Search Benchmark (UTS), and Bouncing Producer-Consumer (BPC) benchmark. These methods do not split jobs, but instead just create new jobs as required. This is a major disadvantage in the overall throughput and utilization of the systems resources. This is an area in which our research addresses and with which we were able to capitalize on and thus improve. With regards to the job splitting portion of our paper, we focus on the task of scheduling n independent jobs on m identical parallel machines with the objective of minimizing total tardiness of the jobs while maintaining well balanced loads on the processors as well. With regard to the scheduling aspect, jobs can be assigned to different processors or split into sub-jobs that can be processed independently on two or more processors/parallel machines at the same time. This research is motivated by a practical need of a printed circuit board (PCB) manufacturing system which is stated at [2]. According to the standard three field notation, the problem is designated as:

$$P|ST_{si,}split|\sum T_j$$

A possible drawback is that the job splitting technique is limited to identical parallel machines and in addition we will require a means of data sharing in order to ensure that once sub-jobs have been successfully computed on separate machines, they can be reconciled (put back together) correctly. To address the issue of various systems been able to access the split jobs, we ensure that the queue is accessible to all the machines on the systems. Therefore once jobs are generated, they can be placed on a queue (shared disk) which is accessible to all the processors that are able to process the jobs. Furthermore, we need to determine a policy by which jobs are to be split into sub-jobs as well as what data needs to be passed to the different processors. A possible option could involve shared memory that can be accessed by all the processors. This would require the use of synchronization to ensure that two or more jobs that access shared data do so correctly while at the same time ensuring that deadlock conditions are avoided. The use of mutex locks by the processors when they attempt to obtain a job was utilized to ensure synchronization and prevent two or more different processors from attempting to acquire the same job. Also our split focused on dividing a job into smaller jobs of relatively equal size that could then be executed by independent threads and then merged upon completion of the independent processing. In order to monitor the performance of a proposed Tabu Search, comparison against best results needs to be performed. This will serve as a bench mark by which we can compare our simulated results to against a best case scenario. One possible option to address the issue of setting a test bench would be to run the mix integer mathematical model formed for the problem in Lingo program. This would take a long period of time to reach a problem of practical size but is ideal for test problems of small size.

By merging a method to send jobs over a network to different processors, and a method to split jobs into pieces that can be run in parallel, this should result in better performance than each method individually. Once the jobs have been split and assigned to the job queue, the scheduling of job queue is also a part of this research. We decided to use First Come First Server(FCFS) scheduling to build up this job queue, which could make the method more smooth and efficient. In addition to this, jobs were placed in both private and public queues, of which the public queue jobs could be "stolen" by otherwise idle processors thus ensuring that CPU utilization had a probability very close to 1 (high CPU utilization). Even processes that were I/O bound would do little to reduce CPU utilization because the idle CPUs' would just acquire any idle jobs from the systems public queue.

Hypothesis

We believe taking the parallelization of a single job, and allowing it to be broken up to multiple machines will increase throughput for the task. This hypothesis is supported in situations in which we can limit the communication time(message passing) between the different parallel processors on the network. The job splitting aspect of the paper might also require us to make a series of assumptions in order to have data that can be readily and easily computed and measured.

- All jobs are available at time zero.
- Each job has a single operation.
- Each machine can process only one sub-job at a time.
- Each sub-job can be processed on only one machine.

We are limited by a variety of factors, one of which is how quickly data can be processed and also the reliability of the network as a whole. If it takes a significant amount of time to pass the data and prepare the jobs, there are periods of time over which the system could be idle either because jobs are not yet available to be placed on the queue or possibly because some jobs are eventually dropped from the system.

Methodology

To generate the test data, we decided to use a large text file which would be split into various sub jobs which in turn would then be assigned to different processors and sorted based on the words (strings) available in the text. The words were sorted based on the ASCII string value and merged together in the final document. The methods used for sorting the data were both a QUICKSORT as well as a MERGESORT technique. The QUICKSORT was used when the job was processed on an individual machine whereas the MERGESORT technique along with a merge routine were used when jobs were to be reconciled to a single sorted file once processing on the sub-jobs was complete. Therefore a job that was to be processed on a single machine was processed using only the QUICKSORT technique, whereas the sub-jobs were processed using both sorting algorithms.
To test our hypothesis, we are coding the new method in C++, using the following algorithm for the job splitting:
Basic Tabu Search Algorithm

k := 1.
generate initial solution
WHILE the stopping condition is not met DO
  Identify N(s). (Neighbourhood set)
  Identify T(s,k). (Tabu set)
  Identify A(s,k). (Aspirant set)
  Choose the best s' $\in$ N(s,k) = {N(s) - T(s,k)}+A(s,k).
  Memorize s' if it improves the previous best known solution
  s := s'.
  k := k+1.
END WHILE

The tabu search algorithm to split the jobs and schedule them will have the following structure.

This section covers the structure of the Tabu Search problem.

As the initial solution, the earliest due date dispatch rule (EDD) is implemented, whereby jobs are sequenced and assigned to machines one by one. In neighbourhood structure we adapt a course of action that includes interchange and insert actions. For the insert action, jobs could be defined in such a way that two jobs are placed such that the first job comes before the second job whereas in interchange they are placed in a manner such thats jobs are placed in each others' former location. Pairwise interchange is performed only on jobs on the same machine, allowing us to schedule more than one sub-job of the same job on the same machine. The candidate list procedure is used to form a sub-group of the entire neighbourhood and is performed only over insert actions since they form larger neighbourhoods when compared to pairwise interchange. Studies of the Tabu search algorithm imply that the best Tabu durations have values between 5 and 12 which may either be used statically or dynamically. This system will attempt to examine the efficiency of job splitting and scheduling on parallel systems given certain parameters as well as determine if there is a parameter set that yields the best result.

Determination of Tabu Parameters:

A total of four factors that could dramatically affect the performance of the proposed Tabu search are listed below.

- Candidate list strategy.
- Tabu duration for pairwise interchange Tabu list.
- Tabu duration for insert Tabu list.
- Halting criterion.

Factor	Low Level	Medium	High
Candidate list Strategy	Machine Job Load	-	Machine Tardiness
Tabu	12	20	25

duration (for pariwise interchange)			
Tabu duration for insert	7	12	15
Sequencing number	1500	-	5000

Blige et al was used to determine the levels of the sequencing number [4]. Since candidate list strategy and sequencing number have two levels whereas the other two factors have three, there are 36 different experiment combinations available for full factorial experiment in test problems.

Each combination was applied to each of the 3 problems in test problems through repeating 3 times in order to minimize the error resulting in 324 experiments for the test problem. It was then observed that candidate list strategy and sequencing number were critical for test problems at 5% significance level.

In order to see the performance of the proposed tabu search on the problems of larger size the authors tried to get solutions on the problem in different sizes. Their Lingo program was given a one hour time period in case the problem could not reach an optimal solution within a reasonable time period in the Lingo runs for test period. For their test problem, the proposed tabu search produced better results by an average of 59.06% in 6.33 seconds on average compared to the results obtained through Lingo program run for an hour.

Next, we need to perform dynamic load balancing between machines in order to move waiting jobs to idle machines. When a process is running, there is two parts to a queue. The head portion is private, and contains all of the jobs that only the local processor can reach. When a new job is created, it is placed into the head. When the head is full, it places the next jobs into a public queue reachable by other processors. When a new process requests a job, it can only pull from the public queue and will pull one, two, or one half of the available jobs, placing them in it's own private and public queues. When the private queue is empty, a pull from the public queue is performed, unless it is empty. If empty, it will then try to find remote public queues with jobs that can be pulled. Please refer to the load balancing algorithm for more details.

load balancing algorithm:

```
While (work_exists) {
   v=select_victim();
   m=get(v.head_information);
   if (work_available(m)) {
      lock (v);
      m=get(v.head_information);
      if (work_available(m)) {
         w=reserve_work(m);
         m=m-w;
         put(m, v.head_information);
         push(queue, get(w, v.queue));
      }
      unlock(v);
```

```
        } // end if(work_available)
    } // end while
```

The networking portion of the proposal will handle multiple connections from any machine on the network. The initial machine runs and will create it's own public and private queues, and starts running jobs. As each additional machine is added, any machine in the pool can be connected to in order to start the connections. The initial connection sends a data packet requesting all known machines in the pool, and when connecting to the rest of the machines a data packet specifying to just connect is given. Using this method, all machines will be connected to all other machines in a star topology. In order to detect if all jobs are complete, a different network packet is sent requesting if a machine has any jobs outstanding for a specific job ID. This will be generated by the originating machine, and will be generated once it has completed all of the jobs it has locally. If any jobs are still outstanding, jobs will be moved from one remote queue to the originating machine to work on. If there are no jobs available due to being located in private queues then a wait for a delay of a random back off will happen and then another check. Please refer to the network connection algorithm for more details:

Network connection algorithm:

```
bool first_connection = true;
While (running) {
    if (first_connection) {
        connect_new_host();
        receive_all_known_hosts();
        connect_to_all_hosts();
    } else {
        if (req = receive_request(remote_machine)){
            if (req == new_host()) {
                // new host joining pool, so send current list of hosts
                send(host_list, remote_machine);
            } else if (req == get_jobs) {
                // requesting jobs to steal.
                num_avail = get_num_jobs_avail();
                if (num_avail > 0) {
                    for (int i=0; i<num_avail/2; i++) {
                        // send half
                        send_job(get_job(), remote_machine);
                    }
                } else {
                    // no jobs to steal
                    send_job(no_jobs, remote_machine);
                }
            } else if (req==num_jobs_for_id) {
                // send number of jobs left for job id
                num_for_job_id = get_num_jobs_avail(job_id);
                send(remote_machine, num_for_job_id);
            }
        }
    }
}
```

We propose to test multiple jobs on a single machine while recording the time to execute.

If we run the same jobs while allowing job splitting and load balancing, we expect the times to decrease. We can potentially test this using a pool of various sizes to show scalability. For instance, if the job is run on one machine it will take x seconds to run. If run on two machines, it should take less than x for a time of y, and if run on three machines it should take less than y.
The output is generated based on the text file that is processed by the individual machine or the multiple machines that then merge the final file that is completely sorted. The output of the processing is shown below depending on the test condition.
Testing against the hypothesis was easily achieved. To determine whether our system that incorporates both job stealing and job splitting as opposed to a system that only uses one of the options, we tested the processing of the job under various conditions. Running the job on only one machine with no job stealing or job-splitting and using a QUICKSORT technique to process the job on only one machine took approximately 2400 seconds (40 minutes). The next test scenario involved running the job using job-splitting but only on one machine which therefore means that only the QUICKSORT technique and not the MERGESORT algorithm was used. This took 828 seconds (approximately 14 minutes).Finally running with job-splitting coupled with job-stealing on two machines resulted in job completion taking approximately 838 seconds (14 minutes).

Implementation
The code to implement the project was done using C++ and is as shown below:

```
# ================================================================
# README.txt
# ================================================================
To build on linux:
make
To build on Solaris:
make sol

Run work_steal on first machine.  It will display the host name.
Run work_steal <other machine host name> to join the pool.
Run work_steal <any machine in pool> for each additional host.
There is a maximum of five machines in the host pool.

Commands:
on any host, there is a user interface.
help will list commands.
Valid commands (only first four characters required):
help    : help menu
list    : list known machines
create  : create job into local queue
crand   : create random number of jobs into local queue
remove  : remove job (from private)
nlocal  : number of local jobs
nremote : number of remote jobs
nsteal  : number of jobs that can be stolen
jobs    : list number of total jobs
steal   : attempt to steal job
shalf   : attempt to steal half the jobs
quit    : quit/exit from pool
vars    : display some debug information
debug   : Enable/Disable debug information
----------------------------------------
runlocal: run local job, timing it.
asteal  : Toggle between auto job stealing and manual mode.
```

runsplit: run with job splitting/stealing.

runlocal will read in final.txt, run a quicksort, and output to
final.loc.txt. The length of time to run is displayed.
runsplit will read in final.txt, split into multiple jobs, and perform each
one.
Jobs are sort using quicksort for 10000 words at a time.
Mergesort is used for two 10000 word sorted files.
Time to run is printed at the end.

Input file is hard coded to be final.txt. This is multiple text books from
Project Gutenberg, spliced together to get a data set.

```
# ============================================================
#       Makefile
# ============================================================
all:  common.cpp common.hpp work_steal.cpp
      #g++ -o work_steal work_steal.cpp -lpthread -lnsl -lsocket -lresolv
      rm -f final.txt.*
      g++ -o work_steal work_steal.cpp -lpthread
debug:      common.cpp common.hpp work_steal.cpp
      #g++ -o work_steal work_steal.cpp -lpthread -lnsl -lsocket -lresolv
      rm -f final.txt.*
      g++ -g -o work_steal work_steal.cpp -lpthread
clean:
      rm -f *.o first_machine second_machine core
sol:  common.cpp common.hpp work_steal.cpp
      g++ -o work_steal work_steal.cpp -lpthread -lnsl -lsocket -lresolv
soldebug:   common.cpp common.hpp work_steal.cpp
      g++ -g -o work_steal work_steal.cpp -lpthread -lnsl -lsocket -lresolv


// ============================================================
//    common.hpp
// ------------------------------------------------------------
// Author : Tim Parker
// Date :   5/25/10
// ============================================================

// include files
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <signal.h>
#include <iostream>
```

```
#include <string>
#include <sstream>
#include <errno.h>
#include <vector>
#include <pthread.h>
#include <sys/unistd.h>
#include <ctype.h>
#include <fcntl.h>
#include <fstream>



// constant declaration
#define PORT "10015"
#define INTPORT 10015
#define BACKLOG 1
#define MAXDATASIZE 300
#define MAX_HOSTS 5
#define STEAL_DELAY_WAIT 1;
#define MAXWORDS 10000
// num_private not working as define
//#define NUM_PRIVATE 2;
using namespace std;

//mutex
pthread_mutex_t public_queue_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t lock;
pthread_t autojob_thread_ID;
void do_quicksort(string &, string &, bool);
void do_mergesort(string &, string &, string &, bool);

int num_jobs_to_steal;
fd_set master_fds;
int fd_max;
int num_remote_jobs;
bool get_jobs;
bool get_num_jobs_steal;
int NUM_PRIVATE;
bool job_running;
struct jobdata {
    string host_machine;   // this will probably change, need some way to
identify a unique machine.  IP address?
    string jobid;
    string jobtype;
    string filename;
    string filename1;
    string filename_out;
};
struct NetJob {
    string cmd;
    jobdata job;
};
void printqueue(int);
char connect_machine[50];
vector<string> host_list;
string local_host_ip;
vector<jobdata> queue;
```

```
int num_job_id_remaining(string);
int num_avail_steal(void);
void get_private_job(jobdata &);
void get_public_job(jobdata &);
jobdata empty_job;
bool DEBUG;
void put_job(jobdata);
int queue_private_index;
fd_set readfds;
int sock; //int listening_sockfd;
fd_set socks;
int steal_delay;
bool autosteal;
int connectlist[MAX_HOSTS];
bool running;
int num_outstanding_job_requests;
int total_jobs;
void *network_interface(void *arg);
void run_job_loc();
void run_job_rem();
ofstream logfile;

// ========================================================
//      common.cpp
// Contains the common code.
// --------------------------------------------------------
// Author:      Tim Parker
// Date:        5/25/10
// ========================================================
// get sockaddr, IPv4 or IPv6:
#include <ctype.h>
#include <stdarg.h>
#include <string.h>
void *get_in_addr(struct sockaddr *sa) {
    if (sa->sa_family == AF_INET) {
        return &(((struct sockaddr_in*)sa)->sin_addr);
    }
    return &(((struct sockaddr_in6*)sa)->sin6_addr);
}
void sigchld_handler(int s) {
    while(waitpid(-1, NULL, WNOHANG) > 0);
}
template <class T>
bool from_string(T& t,
                 const std::string& s,
                 std::ios_base& (*f)(std::ios_base&))
{
  std::istringstream iss(s);
  return !(iss >> f >> t).fail();
}



void pack_and_send (int index, NetJob c_job) {
    stringstream ss;
    unsigned char buf[MAXDATASIZE];
```

```
    ss
<<c_job.cmd<<";"<<c_job.job.host_machine<<";"<<c_job.job.jobid<<";"<<c_job.jo
b.jobtype<<";"<<c_job.job.filename<<";"<<c_job.job.filename1<<";"<<c_job.job.
filename_out<<";"<<endl;
    ss>>buf;
    if (index > host_list.size()-1) {
      cout<<"ERROR in pack_and_send--index greater than host_list.size()-
1"<<endl;
      exit(1);
    }
    if (DEBUG==1) {
        cout<<"Pack_job generated buffer of "<<buf<<" and sending to FD
"<<connectlist[index]<<" host "<<host_list.at(index)<<endl;
        logfile<<"Pack_job generated buffer of "<<buf<<" and sending to FD
"<<connectlist[index]<<" host "<<host_list.at(index)<<endl;
    }
    if (send(connectlist[index], buf,MAXDATASIZE-1, 0) == -1){
        perror("send");
    }
}
string split_str(string &str) {
    int x=str.find(";");
    string retval;
    if (x != string::npos) {
        retval = str.substr(0,x);
        str.erase(0,x+1);
    } else {
        retval = str;
        str.erase();
    }
    return retval;
}

string split_str_space(string &str) {
    int x=str.find(" ");
    string retval;
    if (x != string::npos) {
        retval = str.substr(0,x);
        str.erase(0,x+1);
    } else {
        retval = str;
        str.erase();
    }
    return retval;
}

void unpack (NetJob *c_job, unsigned char *buf) {
    // split packet into job details
    stringstream ss;
    ss << buf;
    string cstr, rstr;
    ss >> cstr;
    c_job->cmd = split_str(cstr);
    c_job->job.host_machine= split_str(cstr);
    c_job->job.jobid = split_str(cstr);
    c_job->job.jobtype = split_str(cstr);
    c_job->job.filename = split_str(cstr);
```

```
    c_job->job.filename1= split_str(cstr);
    c_job->job.filename_out = split_str(cstr);
    if (DEBUG==1) {
        cout<<"cmd='"<<c_job->cmd<<"' host_machine='"<<c_job-
>job.host_machine<<"' jobid='"<<c_job->job.jobid<<"' jobtype='"<<c_job-
>job.jobtype<<"' filename='"<<c_job->job.filename<<"' filename1='"<<c_job-
>job.filename1<<"' filename_out='"<<c_job->job.filename_out<<"'"<<endl;
    }
    logfile<<"cmd='"<<c_job->cmd<<"' host_machine='"<<c_job-
>job.host_machine<<"' jobid='"<<c_job->job.jobid<<"' jobtype='"<<c_job-
>job.jobtype<<"' filename='"<<c_job->job.filename<<"' filename1='"<<c_job-
>job.filename1<<"' filename_out='"<<c_job->job.filename_out<<"'"<<endl;
}

void display_help() {
    cout <<"Help menu for work stealing and job splitting."<<endl;
    cout <<"Valid commands (only first four characters required):"<<endl;
    cout <<"help    : help menu"<<endl;
    cout <<"list    : list known machines"<<endl;
    cout <<"create  : create job into local queue"<<endl;
    cout <<"crand   : create random number of jobs into local queue"<<endl;
    cout <<"remove  : remove job (from private)"<<endl;
    cout <<"nlocal  : number of local jobs"<<endl;
    cout <<"nremote : number of remote jobs"<<endl;
    cout <<"nsteal  : number of jobs that can be stolen"<<endl;
    cout <<"jobs    : list number of total jobs"<<endl;
    cout <<"steal   : attempt to steal job"<<endl;
    cout <<"shalf   : attempt to steal half the jobs"<<endl;
    cout <<"quit    : quit/exit from pool"<<endl;
    cout <<"vars    : display some debug information"<<endl;
    cout <<"debug   : Enable/Disable debug information"<<endl;
    cout <<"-------------------------------------------"<<endl;
    cout <<"runlocal: run local job, timing it."<<endl;
    cout <<"asteal  : Toggle between auto job stealing and manual
mode."<<endl;
    cout <<"runsplit: run with job splitting/stealing."<<endl;
    cout << endl;
}
int myrand (int maxval) {
    double rv = rand();
    int retval = int(rv) % maxval;
//    cout << "rv = "<<rv<<"  retval="<<retval<<endl;
    return retval;
}

void steal_request() {
    int steal_index = -1;
    while (num_jobs_to_steal > 0) {
        if (steal_index == -1) {
            // if -1, pick random machine to steal from
            int num_hosts = 0;
            for(int i=0; i<MAX_HOSTS; i++) {
                if (connectlist[i]> 0) {
                    num_hosts++;
                }
            }
            int rmote_sel = myrand(num_hosts);
```

```
            if (DEBUG==1) {
                cout<<"Selected host offset "<<rmote_sel<<" as victim"<<endl;
            }
          logfile<<"Selected host offset "<<rmote_sel<<" as victim"<<endl;
            // now find proper offset in case a host has disconnected
            for(int i=0; i<MAX_HOSTS; i++) {
                if ((connectlist[i] > 0) && (rmote_sel > -1)) {
                    if (rmote_sel == 0) {
                        steal_index = i;
                    }
                    rmote_sel--;
                }
            }
            if (steal_index < 0) {
                cout<<"Error!  Index to steal from is invalid!"<<endl;
                logfile<<"Error!  Index to steal from is invalid!"<<endl;
                exit(1);
            }
          if (steal_index > host_list.size()-1) {
            cout<<"ERROR in steal_request--index greater than
host_list.size()-1"<<endl;
            exit(1);
          }
            if (DEBUG==1) {
                cout<<"Attempting to steal from host slot "<<steal_index<<"
host "<<host_list.at(steal_index)<<endl;
            }
          logfile<<"Attempting to steal from host slot "<<steal_index<<" host
"<<host_list.at(steal_index)<<endl;
        }
        if (steal_delay > 0) {
            sleep(steal_delay);
            steal_delay = 0;
            // delay is to allow network packet to be sent
            // before generating next packet.
        } else {
            if (DEBUG == 1) {
                cout << "Stealing a job"<<endl;
            }
          logfile << "Stealing a job"<<endl;
            NetJob sjobs;
            sjobs.cmd = "STEAL";
            sjobs.job.host_machine = "0";
            sjobs.job.jobid = "0";
            sjobs.job.filename ="EMPTY";
            if ((steal_index > -1) and (steal_index < MAX_HOSTS)){
                    pack_and_send(steal_index, sjobs);
                    steal_delay = STEAL_DELAY_WAIT;
                    pthread_mutex_lock(&lock);
                    if (num_jobs_to_steal == 1) {
                        // when last job stolen, revert steal_index
                        steal_index = -1;
                    }
                    num_jobs_to_steal--;
                    pthread_mutex_unlock(&lock);
            }
        }
```

```
    }
}

void *autosteal_jobs(void *arg) {
    while(autosteal==true) {
        if (num_avail_steal() > 0) { // do not steal if have public jobs
            sleep(1);
        } else {
            //num_jobs_to_steal+=NUM_PRIVATE-num_job_id_remaining("0");
          if (num_jobs_to_steal < 5) {
            pthread_mutex_lock(&lock);
            num_jobs_to_steal+= 5;
            pthread_mutex_unlock(&lock);
          }
            steal_request();
          sleep(1);
        }
    }
}
void user_interface(int pid) {
    while (running) {
        string inputcmd = "";
        cout<<"Enter command (or help):";
        getline(cin, inputcmd);
        if (inputcmd.find("help") != string::npos) {
            display_help();
        } else if (inputcmd.find("list") != string::npos) {
            cout<<"List of hosts in pool:";
            cout<<local_host_ip<<" ";
            logfile<<"List of hosts in pool:";
            logfile<<local_host_ip<<" ";
            for(int i=0; i<MAX_HOSTS; i++) {
                if (host_list.at(i).compare("EMPTY") != 0) {
                    cout<<host_list.at(i)<<" ";
                    logfile<<host_list.at(i)<<" ";
                }
            }
            cout<<endl;
            logfile<<endl;
        } else if (inputcmd.find("crea") != string::npos) {
            // create job
            jobdata nj;
            nj = empty_job;
            nj.host_machine = "localhost";
            stringstream ss;
            ss << pid;
            //nj.jobid = ss.str();
            ss>>nj.jobid;
            int sval = myrand(500);
            ss << sval;
            nj.filename = "FakeFile"+ss.str()+".txt";
            put_job(nj);
        } else if (inputcmd.find("cran") != string::npos) {
            // create random job
            jobdata nj;
            nj = empty_job;
            nj.host_machine = "localhost";
```

```
            stringstream ss;
            ss << pid;
            //nj.jobid = ss.str();
            ss>>nj.jobid;
            int ranjobs = myrand(20);
            cout<<"Creating "<<ranjobs<<" local jobs."<<endl;
            logfile<<"Creating "<<ranjobs<<" local jobs."<<endl;
            for (int i=0; i<ranjobs; i++) {
                int sval = myrand(500);
                ss << sval;
                nj.filename = "FakeFile"+ss.str()+".txt";
                put_job(nj);
            }
        } else if (inputcmd.find("remo") != string::npos) {
            //remove job (from private)
            jobdata r;
            get_private_job(r);
        } else if (inputcmd.find("nloc") != string::npos) {
            //list number of local jobs
            stringstream ss;
            string spid;
            ss << pid;
            ss>>spid;
            cout<<"Local machine has "<<num_job_id_remaining(spid)<<" jobs
remaining."<<endl;
            logfile<<"Local machine has "<<num_job_id_remaining(spid)<<" jobs
remaining."<<endl;
        } else if (inputcmd.find("nrem") != string::npos) {
            //list number of remote jobs
            if (DEBUG == 1) {
                cout << "Checking remote machines for number of jobs to
steal"<<endl;
            }
          logfile << "Checking remote machines for number of jobs to
steal"<<endl;
            NetJob gjobs;
            gjobs.cmd = "NSJOBS";
            gjobs.job = empty_job;
            for (int i=0; i<MAX_HOSTS; i++) {
                if (connectlist[i] > 0) {
                    pack_and_send(i, gjobs);
                }
            }
        } else if (inputcmd.find("nste") != string::npos) {
            //list number of jobs that can be stolen
            cout<<"Local machine has "<<num_avail_steal()<<" jobs that can be
stolen."<<endl;
            logfile<<"Local machine has "<<num_avail_steal()<<" jobs that can
be stolen."<<endl;
        } else if (inputcmd.find("jobs") != string::npos) {
            //j : list number of total jobs
            pthread_mutex_lock(&lock);
            total_jobs = 0;
            pthread_mutex_unlock(&lock);
            if (DEBUG == 1) {
                cout << "Checking remote machine for number of jobs"<<endl;
            }
```

```
  logfile << "Checking remote machine for number of jobs"<<endl;
    NetJob gjobs;
    gjobs.cmd = "NTJOBS";
    gjobs.job.host_machine = "0";
    gjobs.job.jobid = "0";
    gjobs.job.filename ="EMPTY";
    num_outstanding_job_requests = 0;
    for (int i=0; i<MAX_HOSTS; i++) {
        if (connectlist[i] > 0) {
            pack_and_send(i, gjobs);
            num_outstanding_job_requests++;
        }
    }
} else if (inputcmd.find("stea") != string::npos) {
    //attempt to steal job
    pthread_mutex_lock(&lock);
    num_jobs_to_steal++;
    pthread_mutex_unlock(&lock);
    steal_request();
} else if (inputcmd.find("shal") != string::npos) {
    //attempt to steal half the jobs
    pthread_mutex_lock(&lock);
    num_jobs_to_steal+=num_remote_jobs/2;
    pthread_mutex_unlock(&lock);
    steal_request();
} else if (inputcmd.find("quit") != string::npos) {
    //quit/exit from pool
    // send packet to all connected machines to
    // let them know.
    NetJob gjobs;
    gjobs.cmd = "DISCONNECT";
    gjobs.job.host_machine = "0";
    gjobs.job.jobid = "0";
    gjobs.job.filename ="EMPTY";
    num_outstanding_job_requests = 0;
    for (int i=0; i<MAX_HOSTS; i++) {
        if (connectlist[i] > 0) {
            pack_and_send(i, gjobs);
            num_outstanding_job_requests++;
        }
    }
    running = false;
} else if (inputcmd.find("debu") != string::npos) {
    // toggle debug information
    if (DEBUG == false) {
        DEBUG = true;
        cout<<"Turning debug information ON"<<endl;
    } else {
        DEBUG = false;
        cout<<"Turning debug information OFF"<<endl;
    }
} else if (inputcmd.find("vars") != string::npos) {
    cout<<"Debug information!"<<endl;
    cout<<"Running = "<<running<<endl;
    cout<<"Num_jobs_to_steal = "<<num_jobs_to_steal<<endl;
    cout<<"Num_remote_jobs = "<<num_remote_jobs<<endl;
    cout<<"Job queue size = "<<queue.size()<<endl;
```

```
            cout<<"Remote host connection FD list:";
            logfile<<"Debug information!"<<endl;
            logfile<<"Running = "<<running<<endl;
            logfile<<"Num_jobs_to_steal = "<<num_jobs_to_steal<<endl;
            logfile<<"Num_remote_jobs = "<<num_remote_jobs<<endl;
            logfile<<"Job queue size = "<<queue.size()<<endl;
            logfile<<"Remote host connection FD list:";
            int nhosts = 0;
            for (int i=0; i<MAX_HOSTS; i++) {
                cout<<connectlist[i]<<" ";
                logfile<<connectlist[i]<<" ";
            }
            cout <<endl;
            logfile <<endl;
            cout<<"local host = "<<local_host_ip<<endl;
            logfile<<"local host = "<<local_host_ip<<endl;
            for (int i=0; i<MAX_HOSTS; i++) {
                if (host_list.at(i).compare("EMPTY") != 0) {
                    cout<<"    host_list["<<i<<"] = "<<host_list.at(i)<<endl;
                    logfile<<"    host_list["<<i<<"] =
"<<host_list.at(i)<<endl;
                }
            }
            for (int i=0; i<queue.size(); i++) {
                printqueue(i);
            }
        } else if (inputcmd.find("runl") != string::npos) {
            cout<<"Running job locally."<<endl;
            logfile<<"Running job locally."<<endl;
            run_job_loc();
        } else if (inputcmd.find("aste") != string::npos) {
            if (autosteal == true) {
                cout <<"Setting autosteal to false."<<endl;
                logfile <<"Setting autosteal to false."<<endl;
                autosteal = false;
                void *exit_status;
                pthread_join(autojob_thread_ID, &exit_status);
            } else {
                cout <<"Setting autosteal to true."<<endl;
                logfile <<"Setting autosteal to true."<<endl;
                autosteal = true;
                pthread_create(&autojob_thread_ID, NULL, autosteal_jobs,
NULL);
            }
        } else if (inputcmd.find("runs") != string::npos) {
            cout<<"Running job with splitting/stealing enabled."<<endl;
            logfile<<"Running job with splitting/stealing enabled."<<endl;
            run_job_rem();
        } else {
            cout<<"Unknown command.  Valid commands are:
help,list,crea,remo,nloc,nrem,nste,jobs,stea,shalf,quit"<<endl;
        }
    }
}

int create_connection(string address) {
    // connect_machine is address
```

```
    if (DEBUG==1) {
        cout<<"Creating new connection for "<<address<<endl;
    }
    logfile<<"Creating new connection for "<<address<<endl;
    struct addrinfo hints, *servinfo, *p;
    struct sockaddr_storage their_addr; // connector's address information
    socklen_t sin_size;
    int send_sockfd;
    struct sigaction sa;
    char yes='1';
    char s[INET6_ADDRSTRLEN];
    int rv;
    char connect_machine[50];
    strcpy(connect_machine, address.c_str());
    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE; // use my IP
    if ((rv = getaddrinfo(connect_machine, PORT, &hints, &servinfo)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
        exit(1);
    }
    // loop through all the results and bind to the first we can
    for(p = servinfo; p != NULL; p = p->ai_next) {
        if ((send_sockfd = socket(p->ai_family, p->ai_socktype,
            p->ai_protocol)) == -1) {
            perror("socket");
            continue;
        }
        if (connect(send_sockfd, p->ai_addr, p->ai_addrlen) == -1) {
            close(send_sockfd);
            perror("connect");
            continue;
        }
        break;
    }
    if (p == NULL) {
        fprintf(stderr, "failed to connect\n");
        exit(2);
    }
    inet_ntop(p->ai_family, get_in_addr((struct sockaddr *)p->ai_addr),
        s, sizeof s);
    cout<<"connecting to "<<s<<" on fd "<<send_sockfd<<endl;
    logfile<<"connecting to "<<s<<" on fd "<<send_sockfd<<endl;
    freeaddrinfo(servinfo); // all done with this structure
    for (int listnum = 0; listnum < MAX_HOSTS; listnum++) {
        if (connectlist[listnum] == 0) {
            FD_SET(send_sockfd,&socks);
            if (DEBUG==1) {
                cout<<"Setting connectlist["<<listnum<<"] to be FD
"<<send_sockfd<<endl;
            }
          logfile<<"Setting connectlist["<<listnum<<"] to be FD
"<<send_sockfd<<endl;
            connectlist[listnum] = send_sockfd;
            host_list.at(listnum) = s;
            return listnum;
```

```
        }
    }
}
void deal_with_data(int listnum) {
    unsigned char buf[MAXDATASIZE];
    if (DEBUG == 1) {
        cout<<"deal_with_data, slot "<<listnum<<endl;
    }
    logfile<<"deal_with_data, slot "<<listnum<<endl;
    if (FD_ISSET(connectlist[listnum], &socks)) {
        int numbytes = recv(connectlist[listnum], buf, MAXDATASIZE-1,0);
        if (DEBUG == 1) {
            cout << "Got a network packet!"<<endl;
        }
      logfile << "Got a network packet!"<<endl;
        if (numbytes < 0) {
            // connection closed, close this end and
            // free up entry in connectlist
            cout <<"Connection lost: FD="<<connectlist[listnum]<<"
Slot="<<listnum<<endl;
            logfile <<"Connection lost: FD="<<connectlist[listnum]<<"
Slot="<<listnum<<endl;
            close(connectlist[listnum]);
            connectlist[listnum] = 0;
        } else {
            NetJob rjobdetails;
            unpack(&rjobdetails, buf);
            if (rjobdetails.cmd.compare("") == 0) {
                close(connectlist[listnum]);
            } else if (rjobdetails.cmd.compare("NEW") == 0) {
                if (DEBUG == 1) {
                    cout << "Got a NEW network packet!"<<endl;
                }
            logfile << "Got a NEW network packet!"<<endl;
                NetJob newconnect;
                newconnect.cmd = "CON";
                newconnect.job.host_machine = local_host_ip;
                stringstream ss;
                ss << getpid();
                ss>>newconnect.job.jobid;
                if (DEBUG==1) {
                    cout<<"local host = "<<local_host_ip<<endl;
                    for (int i=0; i<MAX_HOSTS; i++) {
                        if (host_list.at(i).compare("EMPTY") != 0) {
                            cout<<"    host_list["<<i<<"] =
"<<host_list.at(i)<<endl;
                        }
                    }
                }
            logfile<<"local host = "<<local_host_ip<<endl;
            for (int i=0; i<MAX_HOSTS; i++) {
                if (host_list.at(i).compare("EMPTY") != 0) {
                  logfile<<"    host_list["<<i<<"] =
"<<host_list.at(i)<<endl;
                }
            }
                for (int i=0; i<MAX_HOSTS; i++) {
```

```
                    if (host_list.at(i).compare("EMPTY") != 0) {
                        newconnect.job.filename = host_list.at(i);
                        pack_and_send(listnum, newconnect);
                    }
                }
            } else if (rjobdetails.cmd.compare("CON") == 0) {
                if (DEBUG == 1) {
                    cout << "Got CONNECTION information for new host."<<endl;
                }
            logfile << "Got CONNECTION information for new host."<<endl;
                // check if known machine, otherwise connect to new host
                bool found = false;
                for (int i=0; i<MAX_HOSTS; i++) {
                    if (host_list.at(i).compare(rjobdetails.job.filename)==0)
{
                        // matches, so do not add again
                        if (DEBUG==1) {
                            cout<<"host_list["<<i<<"]="<<host_list.at(i)<<"
and incoming host machine is "<<rjobdetails.job.filename<<" so not
adding"<<endl;
                        }
                  logfile<<"host_list["<<i<<"]="<<host_list.at(i)<<" and
incoming host machine is "<<rjobdetails.job.filename<<" so not adding"<<endl;
                        found = true;
                    }
                }
                if (local_host_ip.compare(rjobdetails.job.filename)==0) {
                    found = true;
                }
                if (found==false) {
                    cout<<"Adding existing host
"<<rjobdetails.job.filename<<" to known pool"<<endl;
                    logfile<<"Adding existing host
"<<rjobdetails.job.filename<<" to known pool"<<endl;
                    int connection =
create_connection(rjobdetails.job.filename);
                    // find proper connectlist location
                    for (int j=0; j<MAX_HOSTS; j++) {
                        if (connectlist[j] == connection) {
                            host_list.at(j) = rjobdetails.job.filename;
                        }
                    }
                }

            } else if (rjobdetails.cmd.compare("NTJOBS") == 0) {
                if (DEBUG == 1) {
                    cout << "Got a NTJOBS network packet!"<<endl;
                }
            logfile << "Got a NTJOBS network packet!"<<endl;
                // get number of jobs
                int num = num_job_id_remaining(rjobdetails.job.jobid);
                stringstream ss;
                ss << num;
                NetJob newconnect;
                newconnect.cmd = "NTANS";
                newconnect.job.host_machine = "0";
                newconnect.job.jobid = "0";
```

```
                ss>>newconnect.job.filename;
                pack_and_send(listnum, newconnect);
            } else if (rjobdetails.cmd.compare("NTANS") == 0) {
                if (DEBUG == 1) {
                    cout << "Got an NTANS network packet!"<<endl;
                }
            logfile << "Got an NTANS network packet!"<<endl;
                // number of jobs
                stringstream ss;
                ss << rjobdetails.job.filename;
                int x;
                ss>>x;
                total_jobs += x;
                num_outstanding_job_requests--;
                if (num_outstanding_job_requests==0) {
                    total_jobs += num_job_id_remaining("0");
                    cout<<"There are a total of "<<total_jobs<<" jobs
remaining."<<endl;
                    logfile<<"There are a total of "<<total_jobs<<" jobs
remaining."<<endl;
                }
            } else if (rjobdetails.cmd.compare("NSJOBS") == 0) {
                if (DEBUG == 1) {
                    cout << "Got an NSJOBS network packet!"<<endl;
                }
            logfile << "Got an NSJOBS network packet!"<<endl;
                // get number of stealable jobs
                int num = num_avail_steal();
                stringstream ss;
                ss << num;
                NetJob newconnect;
                newconnect.cmd = "NSANS";
                newconnect.job.host_machine = "0";
                newconnect.job.jobid = "0";
                ss>>newconnect.job.filename;
                pack_and_send(listnum, newconnect);
            } else if (rjobdetails.cmd.compare("NSANS") == 0) {
                if (DEBUG == 1) {
                    cout << "Got an NSANS network packet!"<<endl;
                }
            logfile << "Got an NSANS network packet!"<<endl;
                // number of jobs
                cout<<"Remote machine "<<host_list.at(listnum)<<" has
"<<rjobdetails.job.filename<<" jobs that can be stolen."<<endl;
                logfile<<"Remote machine "<<host_list.at(listnum)<<" has
"<<rjobdetails.job.filename<<" jobs that can be stolen."<<endl;
                stringstream ss;
                ss << rjobdetails.job.filename;
                int x;
                ss>>x;
                num_remote_jobs = x;
                //pthread_mutex_lock(&lock);
                //get_num_jobs_steal = false;
                //pthread_mutex_unlock(&lock);
            } else if (rjobdetails.cmd.compare("STEAL") == 0) {
                if (DEBUG == 1) {
                    cout << "Got a STEAL network packet!"<<endl;
```

```
            }
            logfile << "Got a STEAL network packet!"<<endl;
                // steal a job if available
                int x= num_avail_steal();
                NetJob newconnect;
                jobdata sj;
                if (x > 0) {
                if (DEBUG == 1) {
                  cout<<"Stealing a job..."<<endl;
                }
                logfile<<"Stealing a job..."<<endl;
                    newconnect.cmd = "NJOB";
                    get_public_job(sj);
                    newconnect.job = sj;
                    pack_and_send(listnum, newconnect);
                }
            } else if (rjobdetails.cmd.compare("NJOB") == 0) {
                if (DEBUG == 1) {
                    cout << "Got an NJOB network packet!"<<endl;
                }
            logfile << "Got an NJOB network packet!"<<endl;
                // job from the remote machine
                put_job(rjobdetails.job);
            } else if (rjobdetails.cmd.compare("DISCONNECT") == 0) {
                cout<<"Host "<<host_list.at(listnum)<<" is disconnecting from
pool."<<endl;
                logfile<<"Host "<<host_list.at(listnum)<<" is disconnecting
from pool."<<endl;
                close(connectlist[listnum]);
                host_list.at(listnum) = "EMPTY";
                connectlist[listnum] = 0;
            }
        }
    }
}

void printjob (jobdata job ) {
    cout<<" host_machine="<<job.host_machine;
    cout<<" jobid="<<job.jobid;
    cout<<" jobtype="<<job.jobtype;
    cout<<" filename="<<job.filename;
    cout<<" filename1="<<job.filename1;
    cout<<" filename_out="<<job.filename_out;
    cout<<endl;
    logfile<<" host_machine="<<job.host_machine;
    logfile<<" jobid="<<job.jobid;
    logfile<<" jobtype="<<job.jobtype;
    logfile<<" filename="<<job.filename;
    logfile<<" filename1="<<job.filename1;
    logfile<<" filename_out="<<job.filename_out;
    logfile<<endl;
}


bool FileExists(string &strFilename) {
  struct stat stFileInfo;
  bool blnReturn;
```

```
  int intStat;

  // Attempt to get the file attributes
  intStat = stat(strFilename.c_str(),&stFileInfo);
  if(intStat == 0) {
    // We were able to get the file attributes
    // so the file obviously exists.
    blnReturn = true;
  } else {
    // We were not able to get the file attributes.
    // This may mean that we don't have permission to
    // access the folder which contains this file. If you
    // need to do that level of checking, lookup the
    // return values of stat which will give you
    // more details on why stat failed.
    blnReturn = false;
  }

  return(blnReturn);
}
void *do_job_queue(void *arg) {
    if (DEBUG==1) {
        cout<<"do_job_queue started"<<endl;
    }
    logfile<<"do_job_queue started"<<endl;
    while(running==true) {
        if (num_job_id_remaining("0") == 0) {
            sleep(10);
        } else {
            // handle first job
            jobdata loc_job;
          get_private_job(loc_job);
          if (DEBUG==1) {
            cout<<"Got a private job!";
            logfile<<"Got a private job!";
            printjob(loc_job);
          }
            if (loc_job.jobtype.compare("SORT") == 0) {
                // check if file exists
                if (FileExists(loc_job.filename) == true) {
                    // call quick sort, (source, dest)
                    do_quicksort(loc_job.filename, loc_job.filename_out,
true);
                } else {
                    // file has not been created, move to back of queue
                    put_job(loc_job);
                sleep(1);
                }
            } else if (loc_job.jobtype.compare("MERGE") == 0) {
                // check if file exists
                if (FileExists(loc_job.filename) &&
FileExists(loc_job.filename1)) {
                    // call merge sort, (source1, source2, dest)
                    do_mergesort(loc_job.filename, loc_job.filename1,
loc_job.filename_out, true);
                } else {
                    // file has not been created, move to back of queue
```

```
                    put_job(loc_job);
                }
            }
        }
    }
}


int num_job_id_remaining(string jobid) {
    // search entire queue for jobid = jobid
    // stop this--for now, just return number of jobs.
    return queue.size() - queue_private_index;
//    int num_remaining = 0;
//    for (int i=queue_private_index; i<queue.size(); i++) {
//        if (queue.at(i).jobid.compare(jobid) == 0) {
//            num_remaining++;
//        }
//    }
//    return num_remaining;
}

bool job_is_empty(jobdata job) {
    if ((job.host_machine.compare(empty_job.host_machine) == 0) &&
        (job.jobid.compare(empty_job.jobid) == 0) &&
        (job.jobtype.compare(empty_job.jobtype) == 0) &&
        (job.filename.compare(empty_job.filename) == 0)) {
            return true;
    } else {
        return false;
    }
}


void put_job(jobdata job) {
    // check if job matches an empty job
    if (job_is_empty(job)) {
        // do not put on empty jobs
    } else {
        pthread_mutex_lock(&public_queue_mutex);
        if (DEBUG==1) {
            cout<<"Adding job to queue.  ";
            logfile<<"Adding job to queue.  ";
            printjob(job);
        }
        queue.push_back(job);
        pthread_mutex_unlock(&public_queue_mutex);
    }
}

void get_private_job(jobdata &ret) {
    pthread_mutex_lock(&public_queue_mutex);
    if (queue_private_index >= NUM_PRIVATE) {
        // need to "steal" jobs from public queue.
        // due to vectors, just remove jobs at beginning
        if (queue.size() < NUM_PRIVATE) {
            queue.erase(queue.begin(),queue.begin()+queue.size());
        } else {
```

```
            queue.erase(queue.begin(),queue.begin()+NUM_PRIVATE);
        }
        queue_private_index = 0;
    }
    if (queue_private_index < queue.size()) {
        ret = queue.at(queue_private_index);
    } else {
        ret = empty_job;
    }
    queue_private_index++;
    pthread_mutex_unlock(&public_queue_mutex);
}
void get_public_job(jobdata &ret) {
    if (queue.size() <= NUM_PRIVATE) {
        ret= empty_job;
    }
    pthread_mutex_lock(&public_queue_mutex);

    // steal from middle
    ret = queue.at(NUM_PRIVATE);
    queue.erase(queue.begin() + NUM_PRIVATE);
    // steal from end
//    ret = queue.back();
//    queue.pop_back();

    pthread_mutex_unlock(&public_queue_mutex);
}


int num_avail_steal(void) {
    int num_steal = queue.size();
    if (queue.size() > NUM_PRIVATE) {
        return (num_steal - NUM_PRIVATE);
    } else {
        return 0;
    }
}

void printqueue (int i) {
    cout<<" Queue["<<i<<"] = ";
    logfile<<" Queue["<<i<<"] = ";
    printjob(queue.at(i));
}

void split_txt_file (vector<string> &file_split, string &filename) {
    ifstream datafile(filename.c_str(), ifstream::in);
    string splline, retval;
    getline(datafile, splline);
    while (!datafile.eof()) {
            // strip off leading and trailing whitespace
            size_t startpos = splline.find_first_not_of(" ");
            size_t endpos = splline.find_last_not_of(" ");
            if (endpos > startpos) {
                  splline = splline.substr(startpos, endpos-startpos+1);
            }
            // convert tabs to spaces
            int x = splline.find("\t");
```

```
            while (x < string::npos) {
                  splline.replace(x, 1, " ");
                  x = splline.find("\t");
            }
            // convert runs of spaces to single space
            x = splline.find("  ");
            while (x < string::npos) {
                  splline.replace(x, 2, " ");
                  x = splline.find("  ");
            }
        if (splline.length() > 1) {
            while (splline.length() > 0) {
                retval = split_str_space(splline);
//                cout<<"Debug--adding entry number
"<<file_split.size()+1<<endl;
                file_split.push_back(retval);
            }
        }
        getline(datafile, splline);
    }
}


// ============================================================
//    work_steal.cpp
// Work steal client.  If run without a host, just runs
// locally.  If run with a host, will attempt to connect.
// ------------------------------------------------------------
// Author:    Tim Parker
// Date:    5/25/10
// ============================================================

#include "common.hpp"
#include <pthread.h>
#include "common.cpp"
using namespace std;

void setnonblocking(int sock)
{
    int opts;

    opts = fcntl(sock,F_GETFL);
    if (opts < 0) {
        perror("fcntl(F_GETFL)");
        exit(EXIT_FAILURE);
    }
    opts = (opts | O_NONBLOCK);
    if (fcntl(sock,F_SETFL,opts) < 0) {
        perror("fcntl(F_SETFL)");
        exit(EXIT_FAILURE);
    }
    return;
}
int build_select_list(int highsock) {
    int listnum;         /* Current item in connectlist for for loops */

    /* First put together fd_set for select(), which will
```

```
       consist of the sock veriable in case a new connection
       is coming in, plus all the sockets we have already
       accepted. */


    /* FD_ZERO() clears out the fd_set called socks, so that
        it doesn't contain any file descriptors. */

    FD_ZERO(&socks);

    /* FD_SET() adds the file descriptor "sock" to the fd_set,
        so that select() will return if a connection comes in
        on that socket (which means you have to do accept(), etc. */

    FD_SET(sock,&socks);

    /* Loops through all the possible connections and adds
        those sockets to the fd_set */
    int temp_highsock = highsock;
    for (listnum = 0; listnum < MAX_HOSTS; listnum++) {
        if (connectlist[listnum] != 0) {
            FD_SET(connectlist[listnum],&socks);
            if (connectlist[listnum] > temp_highsock)
                temp_highsock = connectlist[listnum];
        }
    }
    return temp_highsock;
}

void handle_new_connection() {
    int listnum;         /* Current item in connectlist for for loops */
    int connection; /* Socket file descriptor for incoming connections */
    struct sockaddr_storage their_addr; // connector's address information
    socklen_t sin_size;
    char s[INET6_ADDRSTRLEN];
    /* We have a new connection coming in!  We'll
    try to find a spot for it in connectlist. */
    sin_size = sizeof their_addr;
    connection = accept(sock, (struct sockaddr *)&their_addr, &sin_size);
    if (connection < 0) {
        perror("accept");
        exit(EXIT_FAILURE);
    }
    inet_ntop(their_addr.ss_family,
    get_in_addr((struct sockaddr *)&their_addr),
        s, sizeof s);
    if (DEBUG==1) {
        printf("server: got connection from %s\n", s);
    }
    for (listnum = 0; (listnum < MAX_HOSTS) && (connection != -1); listnum
++)
        if (connectlist[listnum] == 0) {
            printf("\nConnection accepted:   FD=%d; Slot=%d\n",
                connection,listnum);
            connectlist[listnum] = connection;
            host_list.at(listnum) = s;
            connection = -1;
```

```
        }
    if (connection != -1) {
        /* No room left in the queue! */
        printf("\nToo many connections, so connect attempt refused.\n");
        NetJob gjobs;
        gjobs.cmd = "MAXHOSTS";
        gjobs.job.host_machine = "0";
        gjobs.job.jobid = "0";
        gjobs.job.filename ="EMPTY";
        pack_and_send(connection, gjobs);
        close(connection);
    }
}
void read_socks() {
    int listnum;         /* Current item in connectlist for for loops */

    /* OK, now socks will be set with whatever socket(s)
       are ready for reading.  Lets first check our
       "listening" socket, and then check the sockets
       in connectlist. */

    /* If a client is trying to connect() to our listening
        socket, select() will consider that as the socket
        being 'readable'. Thus, if the listening socket is
        part of the fd_set, we need to accept a new connection. */

    if (FD_ISSET(sock,&socks))
        handle_new_connection();
    /* Now check connectlist for available data */

    /* Run through our sockets and check to see if anything
        happened with them, if so 'service' them. */

    for (listnum = 0; listnum < MAX_HOSTS; listnum++) {
        if (FD_ISSET(connectlist[listnum],&socks))
            deal_with_data(listnum);
    } /* for (all entries in queue) */
}

void *listen_interface(void *arg) {
    int highsock = sock;
    struct timeval timeout;  /* Timeout for select */
    int readsocks;       /* Number of sockets ready for reading */
    cout<<"Listen interface is running..."<<endl;
    logfile<<"Listen interface is running..."<<endl;
    while (1) { /* Main server loop - forever */
        highsock = build_select_list(highsock);
        timeout.tv_sec = 1;
        timeout.tv_usec = 0;

        /* The first argument to select is the highest file
            descriptor value plus 1. In most cases, you can
            just pass FD_SETSIZE and you'll be fine. */

        /* The second argument to select() is the address of
            the fd_set that contains sockets we're waiting
            to be readable (including the listening socket). */
```

```
        /* The third parameter is an fd_set that you want to
            know if you can write on -- this example doesn't
            use it, so it passes 0, or NULL. The fourth parameter
            is sockets you're waiting for out-of-band data for,
            which usually, you're not. */

        /* The last parameter to select() is a time-out of how
            long select() should block. If you want to wait forever
            until something happens on a socket, you'll probably
            want to pass NULL. */

        readsocks = select(highsock+1, &socks, (fd_set *) 0,
          (fd_set *) 0, &timeout);

        /* select() returns the number of sockets that had
            things going on with them -- i.e. they're readable. */

        /* Once select() returns, the original fd_set has been
            modified so it now reflects the state of why select()
            woke up. i.e. If file descriptor 4 was originally in
            the fd_set, and then it became readable, the fd_set
            contains file descriptor 4 in it. */

        if (readsocks < 0) {
            perror("select");
            exit(EXIT_FAILURE);
        }
        if (readsocks == 0) {
            /* Nothing ready to read, just show that
               we're alive */
//          printf(".");
//          fflush(stdout);
        } else
            read_socks();
    } /* while(1) */
}


int main(int argc, char *argv[]) {
//    vector<string> file_split_results;
//    split_txt_file(file_split_results, "final.txt");
//    cout <<"file_split_results has "<<file_split_results.size()<<"
entries."<<endl;
//    int maxval = 20;
//    if (file_split_results.size() < 20) {
//        maxval = file_split_results.size();
//    }
//    for (int i=0; i<maxval; i++){
//        cout<<file_split_results.at(i)<<" ";
//    }
//    cout<<endl;
//    return 0;
    struct sockaddr_in server_address; /* bind info structure */
    void *exit_status;
    int reuse_addr = 1;  /* Used so we can re-bind to our port
                while a previous connection is still
```

```
            in TIME_WAIT state. */
char hostname[1024];
size_t size;
int rv;
NUM_PRIVATE = 2;
num_outstanding_job_requests = 0;
total_jobs = 0;
DEBUG=1;
queue_private_index = 0;
job_running = true;
autosteal = false;
running = true;
// empty job for queue
empty_job.host_machine = "0";
empty_job.jobid = "0";
empty_job.filename ="EMPTY";
empty_job.jobtype ="EMPTY";
empty_job.filename1 ="EMPTY";
empty_job.filename_out ="EMPTY";
srand(time(NULL));
pthread_t do_job_thread_ID;
pthread_create(&do_job_thread_ID, NULL,do_job_queue , NULL);
for(int i = 0; i<= MAX_HOSTS; i++) {
    host_list.push_back("EMPTY");
}
pthread_mutex_init(&lock, NULL);
int ghn_err = gethostname(hostname, 1023);
hostname[1023] = '\0';
if (ghn_err != 0) {
    cout<<"Error running gethostname!  Return code of "<<ghn_err<<endl;
}
struct hostent *localmachine;
localmachine = gethostbyname(hostname);
char * LocalIP;
LocalIP = inet_ntoa(*(struct in_addr *)*localmachine->h_addr_list);
local_host_ip = LocalIP;
cout<<"work_steal running on host "<<hostname<<" IP "<<LocalIP<<endl;
struct addrinfo hints, *servinfo, *p;
string outfilename = "work_steal.log."+local_host_ip;
logfile.open(outfilename.c_str());
/* Obtain a file descriptor for our "listening" socket */
memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;
if ((rv = getaddrinfo(NULL, PORT, &hints, &servinfo)) != 0) {
    cout<<"getaddrinfo: "<<gai_strerror(rv)<<endl;
    return 1;
}
// loop through all the results and bind to the first we can
for(p = servinfo; p != NULL; p = p->ai_next) {
    if ((sock = socket(p->ai_family, p->ai_socktype,
            p->ai_protocol)) == -1) {
        perror("server: socket");
        continue;
    }
    if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &reuse_addr,
```

```
                sizeof(reuse_addr)) == -1) {
            perror("setsockopt");
            exit(1);
        }
        if (bind(sock, p->ai_addr, p->ai_addrlen) == -1) {
            close(sock);
            perror("server: bind");
            continue;
        }
        break;
    }
    if (p == NULL) {
        fprintf(stderr, "server: failed to bind\n");
        return 2;
    }
    freeaddrinfo(servinfo); // all done with this structure
    setnonblocking(sock);
//    cout<<"Starting to listen"<<endl;

    /* Set up queue for incoming connections. */
    listen(sock,MAX_HOSTS);

    /* Since we start with only one socket, the listening socket,
       it is the highest socket so far. */
    memset((char *) &connectlist, 0, sizeof(connectlist));
    if (argc > 1) {
//        cout<<"Calling create_connection with "<<argv[1]<<endl;
        int firstcon = create_connection(argv[1]);
        cout<<"Connection created, FD="<<connectlist[firstcon]<<endl;
      logfile<<"Connection created, FD="<<connectlist[firstcon]<<endl;
        NetJob newconnect;
        int numbytes;
        newconnect.cmd = "NEW";
        newconnect.job.host_machine = "000";
        newconnect.job.jobid = "00";
        newconnect.job.filename = "REQUEST_HOSTS";
        unsigned char buf[MAXDATASIZE];
        pack_and_send(firstcon, newconnect);
    }
    pthread_t listen_thread_ID;
    pthread_create(&listen_thread_ID, NULL, listen_interface, NULL);
    int pid = getpid();
    user_interface(pid);
//    pthread_join(listen_thread_ID, &exit_status);
    pthread_join(do_job_thread_ID, &exit_status);
}
// Handle the swap
template <typename DataType>
void my_swap(DataType & first, DataType & second)

{
      DataType temp = first;
      first = second;
      second = temp;
}

// Handles the split operation
```

```
int split (vector<string> &x, int first, int last)
{
      string pivot = x[first]; // pivot element
      int left = first, // index for left search
          right = last; // index for right search
      while (left < right)
      {
            while (pivot.compare(x[right]) < 0) // search from right for
            right--; // element <= pivot
                   // search from left for
            while (left < right && // element > pivot
                  ((x[left].compare(pivot) < 0) || (x[left].compare(pivot) ==
0)))
                  left++;
            if (left < right) // if searches haven't met
            my_swap(x[left], x[right]); // interchange elements
      }
      // End of searches; place pivot in correct position
      int pos = right;
      x[first] = x[pos];
      x[pos] = pivot;
      return pos;
}
// Handles the quicksort operation
void quicksort(vector<string> &x, int first, int last)
{
      int pos; // pivot final position
      if (first < last) // list of size is > 1
      {
            pos = split(x, first, last); // split into 2 sublists
            quicksort(x, first, pos - 1); // sort left sublist
            quicksort(x, pos + 1, last); // sort right sublist
      }
      // else list has 0 or 1 element and requires no sorting
}

void do_quicksort(string &infilename, string &outfilename, bool delfiles) {
    vector<string> filecontents;
    split_txt_file(filecontents, infilename);
    // contents of filename now in filecontents
    quicksort(filecontents, 0, filecontents.size()-1);
    string otemp = outfilename+".tmp";
    ofstream outdatafile(otemp.c_str());
    for(int i=0; i<filecontents.size(); i++) {
        outdatafile<<filecontents[i]<<endl;;
    }
    outdatafile.close();
    if (delfiles) {
        remove(infilename.c_str());
    }
    rename(otemp.c_str(), outfilename.c_str());

}

void do_mergesort(string &file1, string &file2, string &ofile, bool delfiles)
{
    string ofile_temp = ofile+".tmp";
```

```
    ofstream outfile(ofile_temp.c_str());
    ifstream infile1(file1.c_str(), ifstream::in);
    ifstream infile2(file2.c_str(), ifstream::in);
    string line1, line2;
    getline(infile1, line1);
    getline(infile2, line2);
    while ((!infile1.eof()) &&(!infile2.eof())) {
        if (line1.compare(line2) > 0) {
            outfile<<line2<<endl;
            getline(infile2, line2);
        } else {
            outfile<<line1<<endl;
            getline(infile1, line1);
        }
    }
    while (!infile1.eof()) {
        outfile<<line1<<endl;
        getline(infile1, line1);
    }
    while (!infile2.eof()) {
        outfile<<line2<<endl;
        getline(infile2, line2);
    }
    infile1.close();
    infile2.close();
    outfile.close();
    if (delfiles) {
        remove(file1.c_str());
        remove(file2.c_str());
    }
    rename(ofile_temp.c_str(), ofile.c_str());
}

void run_job_loc() {
    // get start time
    time_t start, end;
    double diff;
    time (&start);
    // call quick sort with file
    string file1 = "final.txt";
    string file2 = "final.loc.done";
    do_quicksort(file1, file2, false);
    // get finish time
    time (&end);
    // print time difference
    diff = difftime(end, start);
    cout<<"Running locally took "<<diff<<" seconds to execute."<<endl;
    logfile<<"Running locally took "<<diff<<" seconds to execute."<<endl;
}

int split_file_into_parts(string filename) {
    if (FileExists(filename)==false) {
      cout<<"Error!  File "<<filename<<" does not exist!"<<endl;
      exit(1);
    }
    ifstream infile1(filename.c_str(), ifstream::in);
    int filecounter=0;
```

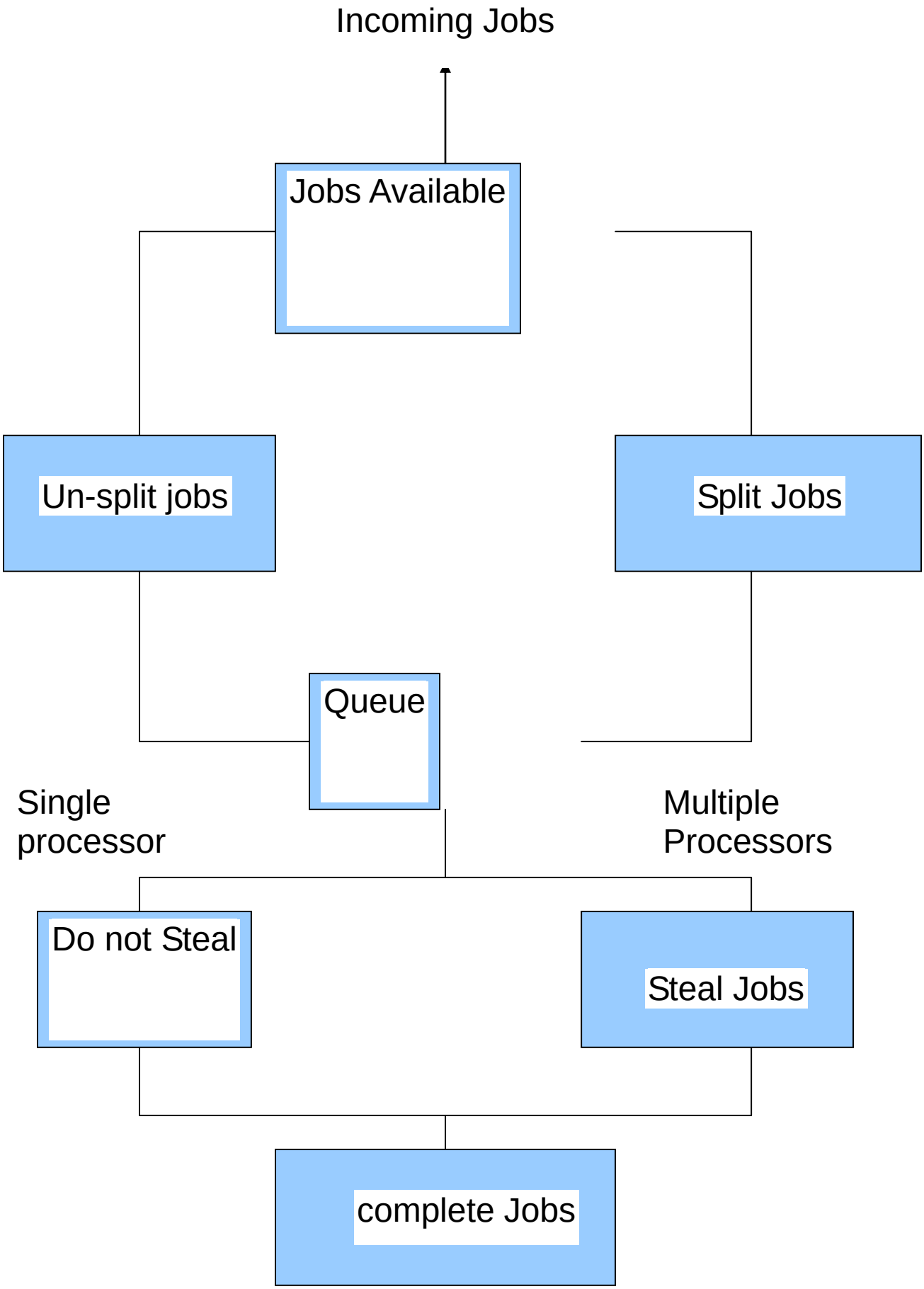
```
    vector<string> filecontents;
    split_txt_file(filecontents, filename);
    string ofilename = filename;
    stringstream ss;
    string tstr;
    ss<<filecounter;
    ss>>tstr;
    FILE *p = NULL;
    ofilename=filename+"."+tstr;
//    fstream outfile(ofilename.c_str(), ios::out|ios::ate);
    p=fopen(ofilename.c_str(), "w");
    int word_counter = 0;
    for(int index=0; index<filecontents.size(); index++) {
        int len = strlen(filecontents[index].c_str());
        fwrite(filecontents[index].c_str(), len, 1, p);
        fwrite("\n", 1, 1, p);
//        outfile<<filecontents[index]<<endl;
        word_counter++;
        if (word_counter>=MAXWORDS) {
//            outfile.close();
            fclose(p);
            filecounter++;
            stringstream ss2;
            ss2<<filecounter;
            ss2>>tstr;
            ofilename=filename+"."+tstr;
            p=fopen(ofilename.c_str(), "w");
//            ofstream outfile(ofilename.c_str(), ios::out|ios::ate);
            word_counter=0;
        }
    }
//    outfile.close();
    fclose(p);
    return filecounter+1;
}


void run_job_rem() {
    double diff;
    // get start time
    time_t start, end;
    time (&start);
    cout<<"splitting file into jobs"<<endl;
    logfile<<"splitting file into jobs"<<endl;
    int num_files = split_file_into_parts("final.txt");
    jobdata nj;
    string tnum;
    vector <string> sorted_files;
    // create sort jobs
    cout<<"creating sort jobs"<<endl;
    logfile<<"creating sort jobs"<<endl;
    for (int i = 0; i<num_files; i++) {
        nj = empty_job;
        nj.jobtype = "SORT";
        stringstream ss;
        ss<<i;
        ss>>tnum;
```

```
    nj.filename = "final.txt."+tnum;
    nj.filename_out = "final.txt."+tnum+".sorted";
    put_job(nj);
    sorted_files.push_back(nj.filename_out);
}
// create merge jobs
int merge_file_num = 0;
vector<string> merged_files;
cout<<"creating merge jobs"<<endl;
logfile<<"creating merge jobs"<<endl;
for (int i=0; i<sorted_files.size(); i=i+2) {
    if (i+1 < sorted_files.size()) {
        nj = empty_job;
        nj.jobtype = "MERGE";
        nj.filename = sorted_files.at(i);
        nj.filename1 = sorted_files.at(i+1);
        stringstream ss;
        ss<<merge_file_num;
        ss>>tnum;
        nj.filename_out = "final.txt.m."+tnum;
        merge_file_num++;
        put_job(nj);
        merged_files.push_back(nj.filename_out);
    }
}
// check for odd number of sort jobs
if (sorted_files.size()%2 > 0) {
    merged_files.push_back(sorted_files.back());
}
// create merge jobs based on previous merge jobs
while (merged_files.size() > 2) {
    nj = empty_job;
    nj.jobtype = "MERGE";
    nj.filename = merged_files.at(0);
    nj.filename1 = merged_files.at(1);
    stringstream ss;
    ss<<merge_file_num;
    ss>>tnum;
    nj.filename_out = "final.txt.m."+tnum;
    merge_file_num++;
    put_job(nj);
    merged_files.push_back(nj.filename_out);
    merged_files.erase(merged_files.begin(), merged_files.begin()+2);
}
// last job should result in a done file
nj = empty_job;
nj.jobtype = "MERGE";
nj.filename = merged_files.at(0);
nj.filename1 = merged_files.at(1);
stringstream ss;
ss<<merge_file_num;
ss>>tnum;
nj.filename_out = "final.merged.done";
put_job(nj);
// now wait for final file to appear
string finalfile = "final.merged.done";
cout<<"Job creation done."<<endl;
```

```
    logfile<<"Job creation done."<<endl;
    while (not FileExists(finalfile)) {
        sleep(5);
    }
    // get finish time
    time (&end);
    // print time difference
    diff = difftime(end, start);
    cout<<"Running with splitting took "<<diff<<" seconds to execute."<<endl;
    logfile<<"Running with splitting took "<<diff<<" seconds to
execute."<<endl;
}
```

Design and Flowchart



Data Analysis and Discussion

The output for the various test scenarios is as follows:

Running with quicksort only on file--2400+ (stopped after 40 minutes):

This is the case in which only one machine processes the job and there is no job-splitting or job-stealing incorporated into the processing of the job. As can be see, this takes a long period of time given a large test sample because computations can only be performed sequentially.

Running with job splitting on one machine--828 seconds. (13m48s):

This scenario is for the case in which we allowed job-splitting on a single processor. As can be seen, the performance is improved from the situation above in which there is no job-splitting. The files are smaller and thus less complex which allows for sorting to be done quicker given the QUICKSORT algorithm. In addition MERGESORT is required which takes additional overhead but it is still faster and more efficient than the earlier scenario.

Running with job splitting using two machines--838 seconds. (13m58s):

The final test sample involved running the jobs on two different machines. This allows the jobs to be not only be split between the two processors, but also to be computed in parallel thus allowing for faster processing. In comparison to the previous situation of processing with one CPU that allows for job splitting we see that processing with two CPU's takes longer. This is due to the fact that there is additional overhead involved in polling the queue to detect if any jobs can be stolen along with the time that is involved in merging the complete job from the sub-jobs. Files that need to be merged may yet to be completed and hence the system may have to wait for the jobs to become available (complete). There would be a marked improvement if we computed the tasks on several machines in parallel, particularly with large data samples. This would result in less jobs waiting to be completed, thus facilitating the merging of the final file a lot quicker and improving throughput.

Some of the problems we ran across that affected the performance of our system included situations in which we had to reduce our network performance speed as packets were likely to be dropped in cases in which the network was fast. A slower network ensures that all the jobs are placed on the queue and limits packet loss. The down side to this is the fact that message passing and job availability is diminished and hence overall throughput suffers. In addition, we are limited to situations in which a bottleneck is caused when one system is slower than the other one. Thus overall performance and throughput is only as fast as the slowest system as the final merging may be impossible due to one CPU still processing the jobs in its queue. This would be similar to a situation in which one processor is overloaded with jobs to execute and hence merging cannot be completed until it is done.

Conclusion and Recommendations

In conclusion we believe that we have demonstrated empirical evidence that proves that we can greatly improve overall system performance by incorporating two different techniques that individually improve the efficiency of processing individual jobs. By using job-splitting coupled with job-stealing we are able to show that jobs can be processed quicker and hence increase the overall throughput of the system. In addition we have also concluded that with a good scheduling and merging algorithm, the overhead incurred can be minimized and thus fully support the need or desire to implement this scheme. This is particularly evident when we have abstractly large amounts of data to be processed which would otherwise bog down a single processor system. We therefore conclude that job-stealing coupled with job-splitting can be used as a means of increasing efficiency not only by increasing the throughput of the system, but also by maximizing CPU utilization. An area that needs to be improved upon is testing with files of various sizes and therefore provide a larger data sample which would serve to clearly define our saturation point (performance threshold). Also we discovered that we need to improve on both our message passing technique as well

as improve our sorting algorithm, which will ensure that jobs are both split as well as queued with minimal overhead.

Bibliography

James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, Jarek Nieplocha. Conference on High Performance Networking and Computing. Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. Article No.: 53, 2009 ISBN:978-1-60558-744-8

Cenk Çelik, İnci Sarıçiçek. Tabu Search for Parallel Machine Scheduling with Job Splitting

Wenzheng Li, Hongyan Shi. Dynamic Load Balancing Algorithm Based on FCFS. 2009 Fourth International Conference on Innovative Computing, Information and Control

References

[1] S.O. Shim and Y.D. Kim, "A branch and bound algorithm for an identical parallel machine scheduling problem with a job splitting property", Computers and Operations Research, 35, 2008, pp.863-875.

[2] Y.D. Kim, S.O. Shim, S.B. Kim, Y.C.Choi and H.M. Yoon, "Parallel machine scheduling considering a job splitting property", International Journal of Production Research, 42, 2004, pp 4531-4546.

[3] W. Xing and J Zhang, parallel machine scheduling with splitting jobs, Discrete Applied Mathematics, 103, 2000, pp. 259-269.

[4] U. Bilge, F. Kirac, M. Kurtulan and P.A. Pekgun, "Tabu search algorithm for parallel machine total tardiness problem ", Computers and Operations Research, 31, 2004, pp. 397-414.