C MPI Torque Tutorial - TSP

Introduction

The example shown here demonstrates the use of the Torque Scheduler for the purpose of running a C/MPI program. Knowledge of C is assumed. Code is also given for the C program shown here as well as the accompanying makefile and shellscript. Having read the Basic Torque Tutorial and the C MPI Torque Helloworld Tutorial prior to this one is also highly recommended.

TSP Problem and Tutorial's Purpose

The C program example shown here solves the classic Traveling Salesman Problem. The premise: a salesman must travel to several cities. The number of cities is known as is the distance between any given pair of cities. From a given city, the salesman must visit all of the cities by way of the shortest distance possible while also returning to his starting point at the end of his journey. This problem is not difficult to solve when the number of cities is very small, but the problem becomes much more demanding as the number of cities increases.

It is worth noting that this document's purpose is not to explain the solution to the tsp problem in any detail, but rather to show how to run this C program with the use of MPI under Torque.

Part One: The Files

tsp.tgz

All the files needed to try out this example are available as a .tgz file on the Star Lab website under the help heading. Summaries of the files inside are provided below. Under Linux, these files can be extracted with the following command:

tar xvf tsp.tgz

The C files, summary

tsp.c - The C program itself (includes main) which includes a function for producing an output file and including the MPI library.
tsp.h - A description of the structures used and the functions employed in tsp.c
list.c - A C module which contains all the tsp list functions.
list.h - The corresponding .h which contains definitions for the List struct, the Node struct, and also descriptions of the various functions in list.c.
hrtimer_x86.c - A timer which is used by tsp.c to test the running time.

hrtimer_x86.h - The corresponding .h file.
makefile - The build script for this program including clean command.

Data files, summary
* These files are used by run.sh as input for tsp.c

city100.tsp - Data for 100 cities.
city500.tsp - Data for 500 cities.
city1000.tsp - Data for 1000 cities.
city5000.tsp - Data for 5000 cities.
city10000.tsp - Data for 10000 cities.

Shell Script files, summary

run.sh - Example script which can be submitted to Torque. This file runs everything else.

Output files, summary
These files are not included in tsp.gz or tsp.zip. They are created as a result of the compilation/running of the program.

tsp - The executable. This is generated by the makefile.
cluster_nodes - This file is generated (and used) by run.sh. Lists available nodes on Aries.
result - The solution to the problem (generated by tsp)
mpi.out - The solution to the problem (generated by run.sh)
run.sh.oXX - This file is generated by Torque and shows the output information, if any. XX represents the torque job number. May be used for troubleshooting.
run.sh.eXX - This file is generated by Torque and shows the error information, if any. XX represents the torque job number. May be used for troubleshooting.

Part Two: Discussion of MPI in C

Before proceeding, a description of MPI's basic operation is in order. A C MPI program progresses through several stages before completion. In short, the stages can be summarized as follows:

Include mpi.h
Initialize the mpi environment (Parallel Code starts here)
Do the work and pass information between nodes by using MPI calls
Terminate the mpi environment (Parallel Code ends here)

MPI accomplishes this by sorting the available nodes into an object called a group. The group is referenced by another object which is tied to it - called a communicator. Although it is possible to use more than one communicator and set up multiple groups, for most purposes it is acceptable to do what we have done here - used just one communicator and one group. In particular, the communicator we have used is called MPI_COMM_WORLD (the predefined communicator) and it is associated with a group which contains all the MPI processes.

Any given process can be further identified by its rank. Every process within a particular group/under a particular communicator has a unique integer rank. These ranks are numbered starting at 0 and are contiguous. For example, if we are dealing with eight processes, they will be ranked from 0 to 7. In the event that a process is included in more than one group (this situation does not occur in the example program described here), it may have more than one rank associated with it. In a case such as this, the context is important - and it will be up to the programmer to specify exactly which group is being referred to at any given time (done by specifying the communicator).

An overview of some of the common MPI commands as they appear in this example program follows. Shown below for reference is an excerpt from the tsp.c main program. (The complete code is available on the Star Lab website as mentioned in Part I).

```
#include "tsp.h"
#include "hrtimer_x86.h"
#include <sys/time.h>
#include <mpi.h>
#include <math.h>

.
.
.

int main( int argc, char* argv[])
{
        int ncities;
        int nthreads;
        int rank;
        int i;
        List result;
        Arg* arg = NULL;
        double before, after;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &nthreads);
```

```
    if (rank == 0)
    {
        fscanf(stdin, "%d\n", &ncities);
        printf("Use 1 master process and %d slave processes for %d cities.\n",
                    nthreads -1, ncities);
        MPI_Bcast(&ncities, 1, MPI_INT, 0, MPI_COMM_WORLD);
        before = gethrtime_x86();
    }


    initialize_list(&result);

    if (rank != 0)
    {
        arg = (Arg *)malloc(sizeof(Arg));
        arg->nsubcities = 0;
        initialize_list(&arg->city_list);
    }

    // assign sub cities for each of the threads.
    subcities(arg, ncities, nthreads);

    if (rank != 0)
    {
        TSP_thread(arg);
    }

    mergeCities(&result, arg, nthreads);

    if (rank == 0)
    {
        //sleep(3000);
        after = gethrtime_x86();
        printf("running time of MPI solution is %f sec\n",after-before);
        output_file(&result);
    }

    MPI_Finalize();
}

.
.
.

void subcities(Arg* arg, int ncities, int nthreads)
{
```

```
int regions = sqrt_my(nthreads - 1);
unsigned long  block_range = RANGE / regions;
int block_X, block_Y;
int flag,tag;
int i;
int rank;
int thread;
City city;
MPI_Status status;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == 0)
{

        for( i = 0; i<ncities ; i++ )
        {
                fscanf(stdin, "%lu %lu\n", &city.X, &city.Y);
                city.X = city.X % RANGE, city.Y = city.Y % RANGE;
                block_X = (int) city.X / block_range, block_Y = (int) city.Y /
                                block_range;

                if (block_Y % 2 == 1)
                {
                        block_X = (regions-1) - block_X;
                }

                thread = regions * block_Y + block_X;
                thread++;

                flag = sizeof(City);
                MPI_Send(&flag, 1, MPI_INT, thread, 100, MPI_COMM_WORLD);
                MPI_Send((char*)&city, sizeof(City), MPI_CHAR, thread, 100,
                                MPI_COMM_WORLD);
        }

        for (i = 1; i < nthreads ; i++)
        {
                flag = 0;
                MPI_Send(&flag, 1, MPI_INT, i, 100, MPI_COMM_WORLD);
        }
}
else
{
        while(1)
        {
```

```
            MPI_Recv(&flag, 1, MPI_INT, 0, 100, MPI_COMM_WORLD, &status);
            if (flag != 0)
            {
                MPI_Recv(&city, sizeof(City), MPI_CHAR, 0, 100,
                                    MPI_COMM_WORLD, &status);
                add_to_tail( &(arg->city_list), &city, sizeof(City) );
                arg->nsubcities += 1;
            }
            else
                break;
        }
    }
}

.
.
.

void mergeCities(List* result, Arg * arg, int nthreads)
{
    int i;
    int rank;
    int flag;

    Node* index;
    City city;
    int subcities;
    MPI_Status status;

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0)
    {
        for(i = 1; i < nthreads; i++)
        {
            while(1)
            {
                MPI_Recv(&flag, 1, MPI_INT, i, 100, MPI_COMM_WORLD,
                        &status);
                if (flag == 0)
                    break;
                MPI_Recv((char*)&city, sizeof(City), MPI_CHAR, i, 100,
                        MPI_COMM_WORLD, &status);
                        add_to_tail( result, &city, sizeof(City) );
            }
        }
    }
```

```
    else
    {
        if (arg->nsubcities != 0)
        {
            index = arg->city_list.head;
            subcities = arg->nsubcities;
            for ( i = 0; i < subcities; i++)
            {
                flag = sizeof(City);
                MPI_Send(&flag, 1, MPI_INT, 0, 100, MPI_COMM_WORLD);
                MPI_Send((char*)index->data, sizeof(City), MPI_CHAR, 0, 100,
                        MPI_COMM_WORLD);
                index = index->next;
            }
            flag = 0;
            MPI_Send(&flag, 1, MPI_INT, 0, 100, MPI_COMM_WORLD);
        }
        else
        {
            flag = 0;
            MPI_Send(&flag, 1, MPI_INT, 0, 100, MPI_COMM_WORLD);
        }
    }
}
```

Let us take a look at how the MPI mechanism operates in our example program as we move through main. After the variables are defined, the MPI environment is initialized with MPI_Init. From this point onward, we now have multiple versions of the code running - this is where the parallelism has begun. And so each of those versions (processes) needs to know what its rank is and also how many other processes are out there. This is assigned by use of MPI_Comm_rank and MPI_Comm_size.

From this point on, the rank 0 process serves as a sort of master process. For example, the rank 0 process sends information to the other processes by use of MPI_Bcast and also sends information to other processes in the subcities function by way of the MPI_Send command. When those processes have completed work on that information, they send the information back to the rank 0 process which combines it into the final result (in the mergeCities function). The Rank 0 process is given the task of sending the contents of the resulting list to a file before the MPI resources are again released (and the parallelism ends).

The lines of code marked off in red in the above example are MPI specific. Further commentary on their functions follow.

#include <mpi.h> - This include statement needs to go into every module that makes an MPI call.

int MPI_Init(int *argc, char ***argv) - Initializes the MPI execution environment. MPI_Init needs to be called prior to any other MPI functions, and therefore is required in every C program that uses MPI. MPI_Init must be called only once. (Note - MPI_Init obtains its arguments from the mpirun command. In our example, the mpirun command is included in the run.sh shell script - see Part Four. The mpirun command provides the number of nodes and a list of hosts to the MPI_Init command - see usage example below).

```
int main(int argc, char *argv[])
{
   ... ...

   MPI_Init(&argc,&argv);
   ... ...
}
```

int MPI_Comm_rank (MPI_Comm comm, int *rank) - Determines the rank of the calling process (and places it at the address *rank). (A rank can be anywhere from 0 up to the number of processors). The rank will also vary based upon the communicator used (in our program, the only communicator used is MPI_COMM_WORLD). A process can have one rank under one communicator, but a different rank under another communicator.

int MPI_Comm_size (MPI_Comm comm, int *size) - Determines the number of processes in the current group of comm and places that integer at the address *size. In our example, the only communicator used is called MPI_COMM_WORLD.

int MPI_Bcast (void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm) - In this example, a message (ncities) is transmitted from the process with rank 0 (as associated with the MPI_COMM_WORLD communicator) to all the other processes of that group. MPI_Bcast defines its variables in this way: *buffer is the starting address of the buffer, count represents the number of items in the buffer, datatype represents the kind of datatype in the buffer, root is the rank of the broadcasting root, and comm is the communicator.

subcities and mergeCities - This is where the majority of the MPI message passing is done. To describe the MPI component of this work we need to notice two new functions - MPI_Send and MPI_Recv. These are the most common functions used for sharing information under MPI.

In short, the subcities function works by first identifying the rank of the current process (by using the MPI_Comm_rank command). Next, the rank is checked - if it is 0, it reads cities from the standard input (in this case, a file) and sends the information out to the

other processes. On the other hand, if we are not working with the rank 0 process, the function will instead go into a loop where it will continue receiving city information until there is no more available to receive.

The mergeCities function works in almost the opposite way - in this case, it is the non-rank 0 process that do the sending and the rank 0 process that goes into receiving mode.

int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm) - A common command for sending data via MPI. buf is the initial address of the send buffer, count provides the number of elements (nonzero integer), the datatype specifies exactly what datatype is in use in the buffer, dest is the rank of the destination process, tag is the message tag, and the final parameter is the communicator.

int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status) - A common command for receiving data via MPI. buf is the initial address of the receive buffer, count provides the number of elements (nonzero integer), the datatype specifies exactly what datatype is in use in the buffer, source is the rank of the destination process, tag is the message tag, and the final parameter is the communicator.

MPI_Finalize(); - This command terminates the MPI execution environment. Therefore it needs to be the last MPI command called. This command needs to be included in every MPI program to ensure that all resources are properly released.

A great deal of additional information is available on the net on the specifics of MPI. Two excellent resources for further reading are available here:

https://computing.llnl.gov/tutorials/mpi/#Getting_Started

http://www.mhpcc.edu/training/workshop/mpi/MAIN.html#What

Part Three: An MPI Makefile

The makefile used to compile our C MPI program(with notes added) is as follows:

```
CC =  mpicc                                    -- specify mpi compiler
CFLAGS = -O3                                   -- specify complier option
BASIC_FILE = list.c hrtimer_x86.c              -- specify files included in main program
All: tsp
tsp: tsp.c
	$(CC) -o tsp tsp.c $(BASIC_FILE) $(CFLAGS) -lpthread
clean:
	rm tsp
```

The line highlighted in red is a typical MPI compiling command. Additionally, our makefile provides the following functionality:

make -- compile
make clean -- clear executables - you may clear the files before you recompile the program.

If you program is not very complex, you may choose to compile by using the command line instead of writing a makefile. In that case, the command would be specified in the following format:

```
mpicc –o <executable> -O3 <main program> <programs included in main program> -lpthread
```

Part Four: The Torque Shell Script

Torque accepts a job in the form of a shell script (a .sh file). The text that follows shows the complete run.sh script for the tsp program along with commentary.

```
#!/bin/bash

cd $PBS_O_WORKDIR

#set your machinefile
cat $PBS_NODEFILE > ./cluster_nodes      -- get available nodes and generate node file

#run mpi on these nodes
echo "--------- THE BEGINNNING ---------" >> ./mpi.out
cat city10000.tsp | mpirun -np  10 -machinefile ./cluster_nodes ./tsp >> ./mpi.out
rm run.sh.*                 --  delete all previous standard output and standard error files
```

The mpi command, shown in red, demonstrates a standard command once again. The -np switch requests ten processes for the program. The -machinefile switch identifies the file which has a list of nodes in the Ariespool, in this case, the cluster_nodes file. The C executable is specified (tsp) and output is redirected to mpi.out.

The number of processes specified with the -np switch may exceed the number of nodes available on the cluster. In our testing, we found that using roughly twice as many processes as nodes provided the best results. In this example, we are using ten processes, but will only use four physical nodes (see Part Five below).

Part Five: Submission of the Torque Script

***Important Note**: The user must be logged into the ariessrv in order to submit a job successfully to Torque.*

Run your MPI job with the following command (in this example, four nodes are requested):

qsub –l nodes=4 ./run.sh

Once the command has been issued, you will be given the job number. The qstat command can be issued at any time after submission in order to see where your job is in the queue. (See the Basic Torque Tutorial for more details on the use of this command). The mon -d command can also be used in order to see activity on the cluster (use q to quit from mon).

Once the job has finished, several new files will have been created. The run.sh.oXX file and run.sh.eXX will provide troubleshooting information in the event that there was a problem with the execution. In our example, the program solution is shown in the file mpi.out.

Part Six: Final Considerations

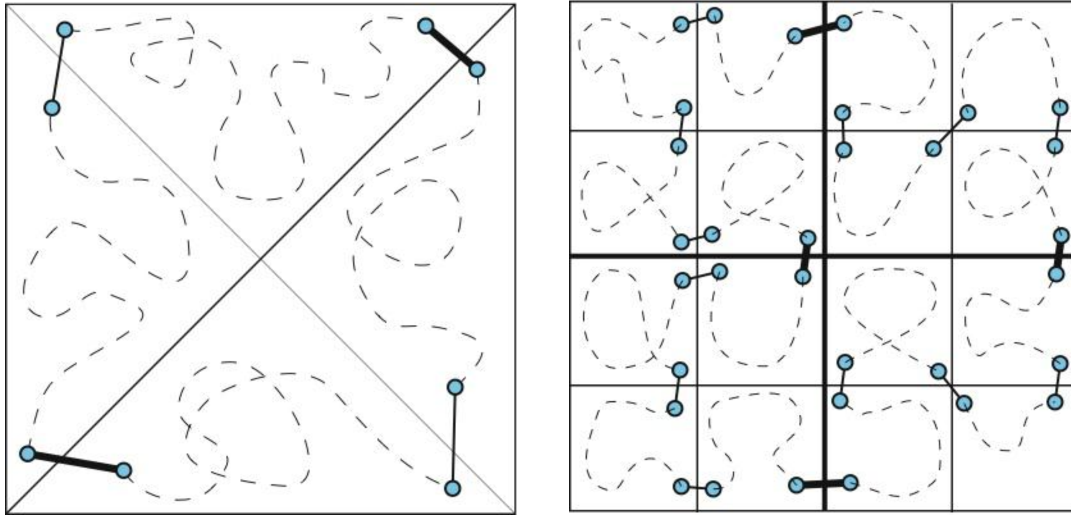
For all long jobs (several hours or more) users are asked to use the Torque Scheduler as described here out of fairness to other users and also as a way to ensure the most efficient possible use of Aries cluster resources. When a user occupies more nodes, the program will, in theory, run faster. However, in practice, because Torque will not start your job until the requested number of nodes are available, the same user may actually end up having to wait longer for the program to begin. Because of this trade-off, it is strongly suggested that users request only a moderate number of nodes when submitting a script. Not only will your program have the best chance to start running sooner, but some nodes will remain free for other users as well. Please be a good Star Lab citizen and do your part!

Appendix - The TSP MPI Algorithm, Summary

This program generates 1 master process and n-1 slave processes. The master process divides the whole region into a square number of blocks and sends the city information in the block to the corresponding slave process. Each slave process takes care of one block and, in this way, the TSP algorithm is executed in parallel. After all the work is done, every slave process sends the result back to the master process.

TSP algorithm:
The main idea of this algorithm is to exploit the fact that nodes are randomly distributed in Euclidean space: allocate geometric regions to different processes, and then stitch the regions together. You might, for example, do this recursively, solving the upper-left

triangle and the lower-right triangle, and then connecting them at the upper-right and lower-left corners, or similarly with squares:



For each process, due to the fact that the inversion takes place where the tour happens to have a loop rather than an intersection, then the loop is opened and the salesman is guaranteed a shorter tour. Therefore, if the process find there is an intersection (cross) in the tour (left figure), it will invert the tour to make an open loop (right figure).