**Cannon’s Algorithm Testing**

I built my program to multiply two matrices using Cannon’s algorithm. The size of the matrix was determined by dividing the runtime argument, n, by the square root of the number of processors. Because of this method, the number of processors was required to be a perfect square (1, 4, 9, 25, 36). The runtime argument ‘n’ had to be a multiple of the square root of the number of processors. This meant that the initialized matrices ‘a’ and ‘b’ had n/sqrt(p) columns and rows. Each process held (n/sqrt(p))^2/p numbers of the initialized matrices. Separate functions were written for initializing the arrays, printing the arrays, and multiplying the corresponding ‘a’ rows and ‘b’ columns. The time of the algorithm, starting after the arrays are initialized, was then measured.

I tested the program using squares of the numbers 1-6 with three varying sizes of ‘n’: 900, 1800, and 8100. However, the size of the matrix was determined by dividing ‘n’ by the root number of processes specified, which means the matrix size initialized by ‘a’, ‘b’, and ‘c’ varied – as can be seen in the table below.

**Communication**

Initially, communication was an issue. Minimizing the sends and receives across processes was a priority from the start. Indeed, as the following tables show my program ran fastest when running with 1 process. The communication costs are clearly problematic. Due to time constraints, I was unable to test my program using arrays larger than 8100 by 8100. Since the time of running the algorithm with 1 process was so much faster than with multiple, the speedup was measured against the runtime of the algorithm with the next highest perfect square of processes (4).

After the matrices were initialized and the Cartesian grid of the processes created, the matrices were skewed as specified by Cannon’s algorithm. In the main loop of the program, there are two Sendrecv calls that send the entire matrices ‘a’ and ‘b’ that shift the matrices after the necessary multiplication. These communications yield an n^2/p cost.

**Computation**

Given that each process held (n/sqrt(p))^2/p numbers, the computation cost of my program was (n/sqrt(p))^3. That is, an n3/p3/2 computational cost.

**Memory**

Since each process initialized matrices ‘a’ and ‘b’ as opposed to process 0 initializing and scattering the results, communication was saved at the cost of memory. Had more communication been used in my program, memory costs could have been cut – but possibly at the cost of runtime.

**Test table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Number of Proc** | **Time (seconds)** | **Speedup** | **n** | **matrix** |
| 1 | 0.000007 | 1 | 900 | 900 x 900 |
| 4 | 0.726491 | 1 | 900 | 450 x 450 |
| 9 | 0.431988 | 1.681738845 | 900 | 300 x 300 |
| 25 | 0.187618 | 3.872181774 | 900 | 180 x 180 |
| 36 | 0.136102 | 5.337842207 | 900 | 150 x 150 |
|  |  |  |  |  |
| 1 | 0.000002 | 1 | 1800 | 1800 x 1800 |
| 4 | 5.816707 | 1 | 1800 | 900 x 900 |
| 9 | 3.45333 | 1.68437624 | 1800 | 600 x 600 |
| 25 | 1.491059 | 3.90105757 | 1800 | 360 x 360 |
| 36 | 1.079449 | 5.388588993 | 1800 | 300 x 300 |
|  |  |  |  |  |
| 1 | 0.000002 | 1 | 8100 | 8100 x 8100 |
| 4 | 531.932974 | 1 | 8100 | 4050 x 4050 |
| 9 | 316.052514 | 1.683052501 | 8100 | 2700 x 2700 |
| 25 | 136.567097 | 3.89503025 | 8100 | 1620 x 1620 |
| 36 | 98.80127 | 5.383867778 | 8100 | 1350 x 1350 |

\*speedup was measured against 4 processes instead of 1

Speedup was achieved once more than one process was used. As can be seen in each case speedup nearly reached the square root of the processes used.

**Speedup vs Runtime Graphs**

\*Time not displayed as some processes took well over 5 minutes

**Code:**

\*some verification and comments for testing were removed

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <mpi.h>

#include <time.h>

void print(int \* m, int size, const char\* msg, int rank, bool whole);

void initializeMatrix(int \* m, int size, bool empty);

void multiply(int \* a, int \* b, int \* c, int size);

void swap(int \* m, int \* recvbuf, int \* tmp);

int main(int argc, char \*argv[])

{

int rank, p;

int n, local\_n; /\*local\_n used for matrices\*/

int dimensions[2]; /\*dims for cart\*/

int wrap\_arounds[2]; /\*for skewing\*/

int left,right,up,down; /\*will act as shifting the arrays\*/

int \*a,\*b,\*c;

int \*recvbuf,\*tmp; /\*tmp for swapping recvbuf with matrix array\*/

double start, stop;

MPI\_Comm comm;

MPI\_Init(&argc,&argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD,&rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD,&p);

srand(time(0));

dimensions[0] = dimensions[1] = 0;

wrap\_arounds[0] = wrap\_arounds[1] = 1;

MPI\_Dims\_create(p,2,dimensions); /\*cartesian array of proc\*/

n = atoi(argv[1]);

local\_n=n/dimensions[0]; /\* n/sqrt(p) \*/

a = new int[local\_n \* local\_n];

b = new int[local\_n \* local\_n];

recvbuf = new int[local\_n \* local\_n]; /\*for sendrecv\*/

c = new int[local\_n \* local\_n];

//set up matrices

initializeMatrix(a, local\_n, false);

initializeMatrix(b, local\_n, false);

initializeMatrix(c, local\_n, true);

//initial skew

MPI\_Cart\_create(MPI\_COMM\_WORLD,2,dimensions,wrap\_arounds,1,&comm);

MPI\_Cart\_shift(comm,0,1,&left,&right);

MPI\_Cart\_shift(comm,1,1,&up,&down);

start=MPI\_Wtime();

for(int i=0;i<dimensions[0]-1;i++) {

//if(i==dimensions[0]-1)

//break;

multiply(a, b, c, local\_n);

//shifting MPI\_Sendrecv(a,local\_n\*local\_n,MPI\_INT,left,1,recvbuf,local\_n\*local\_n,MPI\_INT,right,1,comm,MPI\_STATUS\_IGNORE);

swap(a, recvbuf, tmp); MPI\_Sendrecv(b,local\_n\*local\_n,MPI\_INT,up,2,recvbuf,local\_n\*local\_n,MPI\_INT,down,2,comm,MPI\_STATUS\_IGNORE);

swap(b, recvbuf, tmp);

}

MPI\_Barrier(comm);

stop=MPI\_Wtime();

if(rank==0)

printf("p: %d\tn: %d\ttime: %f\n",p, n, stop-start);

free(a); free(b); free(recvbuf); free(c);

MPI\_Finalize();

return 0;

}

void swap(int \* m, int \* recvbuf, int \* tmp){

tmp = recvbuf;

recvbuf = m;

m = tmp;

}

void multiply(int \* a, int \* b, int \* c, int size){

for(int i=0;i<size;i++)

for(int j=0;j<size;j++)

for(int k=0;k<size;k++)

c[i\*size+k]+=a[i\*size+j]\*b[j\*size+k];

}

void initializeMatrix(int \* m, int size, bool empty){

for(int i=0;i<size;i++)

for(int j=0;j<size;j++) {

/\*initialize empty array with 0\*/

int r = 0;

if (!empty)

r = rand() % size + 1;

m[i\*size+j]=r;

}

}

void print(int \* m, int size, const char\* msg, int rank, bool whole){

printf(msg);

printf(" rank: %d\n", rank);

if(whole){/\*print entire matrix\*/

for (int i = 0; i < size \* size; i++){

printf("%d ", m[i]);

if ( (i+1) % size == 0){

printf("\n");

}

}

}

else{

for(int i=0;i<size;i++)

for(int j=0;j<size;j++) {

printf("%d ", m[i\*size+j]);

}

}

printf("\n");

}

**Submission Result:**

workdir is: /scratch/rga001/cannons

/scratch/rga001/cannons

p: 1 n: 900 time: 0.000007

p: 4 n: 900 time: 0.726491

p: 9 n: 900 time: 0.431988

p: 25 n: 900 time: 0.187618

p: 36 n: 900 time: 0.136102

p: 1 n: 1800 time: 0.000002

p: 4 n: 1800 time: 5.816707

p: 9 n: 1800 time: 3.453330

p: 25 n: 1800 time: 1.491059

p: 36 n: 1800 time: 1.079449

p: 1 n: 8100 time: 0.000002

p: 4 n: 8100 time: 531.932974

p: 9 n: 8100 time: 316.052514

p: 25 n: 8100 time: 136.567097

p: 36 n: 8100 time: 98.801270

PBS Job Statistics:

PBS Input:

#PBS -N cannon

#PBS -q tiny12core

#PBS -j oe

#PBS -m e

#PBS -o cannon.$PBS\_JOBID

#PBS -l nodes=3:ppn=12

#PBS -l walltime=01:00:00

cd $PBS\_O\_WORKDIR

echo "workdir is: $PBS\_O\_WORKDIR"

echo $PBS\_O\_WORKDIR

mpirun -np 1 -machinefile $PBS\_NODEFILE /scratch/rga001/cannons/cannon 900

mpirun -np 4 -machinefile $PBS\_NODEFILE /scratch/rga001/cannons/cannon 900

mpirun -np 9 -machinefile $PBS\_NODEFILE /scratch/rga001/cannons/cannon 900

mpirun -np 25 -machinefile $PBS\_NODEFILE /scratch/rga001/cannons/cannon 900

mpirun -np 36 -machinefile $PBS\_NODEFILE /scratch/rga001/cannons/cannon 900

mpirun -np 1 -machinefile $PBS\_NODEFILE /scratch/rga001/cannons/cannon 1800

mpirun -np 4 -machinefile $PBS\_NODEFILE /scratch/rga001/cannons/cannon 1800

mpirun -np 9 -machinefile $PBS\_NODEFILE /scratch/rga001/cannons/cannon 1800

mpirun -np 25 -machinefile $PBS\_NODEFILE /scratch/rga001/cannons/cannon 1800

mpirun -np 36 -machinefile $PBS\_NODEFILE /scratch/rga001/cannons/cannon 1800

mpirun -np 1 -machinefile $PBS\_NODEFILE /scratch/rga001/cannons/cannon 8100

mpirun -np 4 -machinefile $PBS\_NODEFILE /scratch/rga001/cannons/cannon 8100

mpirun -np 9 -machinefile $PBS\_NODEFILE /scratch/rga001/cannons/cannon 8100

mpirun -np 25 -machinefile $PBS\_NODEFILE /scratch/rga001/cannons/cannon 8100

mpirun -np 36 -machinefile $PBS\_NODEFILE /scratch/rga001/cannons/cannon 8100

PBS Job ID: 712352.sched

Resource List: neednodes=3:ppn=12,nodes=3:ppn=12,walltime=01:00:00

Resources Used: cput=03:01:40,mem=2709488kb,vmem=8705940kb,walltime=00:18:48

Queue Name: tiny12core

Program Return Code: 0

Head Node:

compute1131

Slave Nodes:

compute1133

compute1134

nnodes= 3 ncores= 36 njobs=1