Phys240Project

Justin Dion

April 2015

Abstract

In this Project the speed of different programming languages and parralel method's were analyzed and compared using the compute time of a 2-D Ising model. The speed test was ran using programming languages : Python, Fortran, C and compared with parralization using Fortran Open MPI and Nvidia CUDA C.

1 Introduction

With the emergent of technology it has become possible to not only run computer programs in serial form, but also parralize it to have multiple operations done at the same time. There are many methods to do this, some taking advantage to the multiple cores on CPU's and others taking advantage of computers GPU's. For this project the speed of both of these methods were testing using an Ising Model to establish the time each method takes. The Ising model is used to study the phase change from ferromagnetic to antiferromagnetic, by modeling it as a 2-d array of with values of -1 or 1 corresponding to the magnetic moment. From this model the temperature of at which the phase transition can be calculated for a given lattice of magnetic moments. The Ising model was programmed in python, fortran, and C and compared against the speed of parralized code using fortran OpenMPI and CUDA-C.

2 Ising Model

Using the Ising model the energy is expressed as:

$$E = -J \sum_{jk}^{N} S_j S_k - B \sum_{k}^{N} s_k \tag{1}$$

which can be broken down into two components a coupling component between the spins and a magnetic terms. The coupling terms depends on each elements nearest neighbor. To get expection values for the mean energy and magnetization we must use the probability distribution for a given temperation:

$$P_i(\beta) = \frac{e^{-\beta E_i}}{Z} \tag{2}$$

where $\beta = 1/kT$ k is the boltzman constant T is the temperature and E_i is the energy of state i. Using this the Energy of a speific configuration can be written as with B = 0:

$$E_i = -J \sum_{j=1}^{N-1} S_j S_{j+1} \tag{3}$$

and the magnetization of the state:

$$M_i = \sum_{j=1}^{N} S_j \tag{4}$$

Example of the algorithim:
1.Declare the size of the array N, and the amount of iterations I.
2. Set up a 2-d array with N by N dimensions with values of -1 or 1.
3. For each iterations sweep through each elements of the array switching the values from 1 to -1 or vice versa, calculate the coupling term of each elements nearest neighbors if $E_n ew E_o ld$ accept swith otherwise accept with a probablity $e^{-(E_{new}-E_0)/(K_b T)}$ after accepting a switch update the new energy.
4. Do this for multiple temperatures T and plot the E vs T, and M vs T.
For lower temperatures the probability of accepting a position will be small and the system will tend towards the lower Energy states, but as temperature increase so does the probability leading to a transition where most of the switches are accepting indicating a phase change which can be seen when energy is plotted against temperature with a large slope at the critical temperature, T $\approx$ 2.27.

3 System Information

OS:Ubuntu 14.04.2
Cpu:Intel(R) Core(TM) i3 CPU
Clock speed :2.13 Ghz
Cores/Threads : 2/4

Gpu: NvidiaGeforce 310M
Gpu Engine Specs
Cude Cores: 16
Processor Clock (Mhz) : 1530 Mhz
Gigaflops : 73
Memory Specs
Memory Clock: Up to 800MHZ (DDR3)
Memory: Up to 1GB
Memory Inteface Width: 64-bit
Compute Capability: CUDA 1.2

4 Python,Fotran, and C

The first step was to code the Ising model into normal serial code in Python, Fortran, and C. This was done by input the size of the NxN array into the variable N and by determining the amount of steps into the variable I(In this model it was decided to have a fixed number of iterations so each language was doing the same amount of calculations and less variation in compute time, when actually doing the model want a while loop until desired uncertainty in values is reached). The N by N array was then constructed choosing a value of -1 or 1 for each index, and then fed into a flip function. The flip function sweeps through the matrix starting from first index to the last index (some variations randomly pick index to flip) flipping each and calculating the energy change of the matrix due to the flip. The energy change calculation was done using a relative energy function, instead of calculating the energy of the whole matrix which would take time just the energy of the nearest neighbor of the index is calculated, since the rest stays constant. After the flip function the total energy of the matrix is calculated and the process is repeated for I iterations, averaging the total energy over each iteration. This process was repeated for $0 < k_B T < 5$ where k_B is the boltzman constant and T is the temperature.

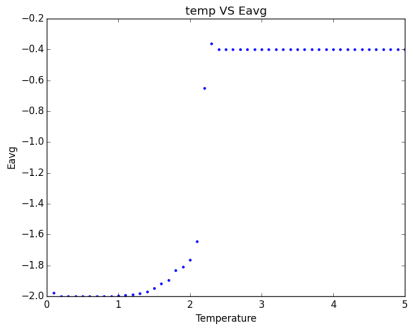

Figure 1: Results of Ising model with 10x10 matrix with 100 iterations, plot of T vs Eavg

5 Parrallization using OpenMPI in fortran

Code is normally run in serial, which means one step after another. For awhile processor speeds kept increasing, so this meant that programmers got faster, but eventually the increased speed caused an increase in temperature and the processors weren't in safe conditions so the manufactures decide instead of making the processors faster put more on each chip. This allows for programs to be parrallelized, or have multiple tasks running at the same time on each core. Parallelizing the Ising model was straight forward since each core and have its own task and there isn't any communication between the cores. To parallelize

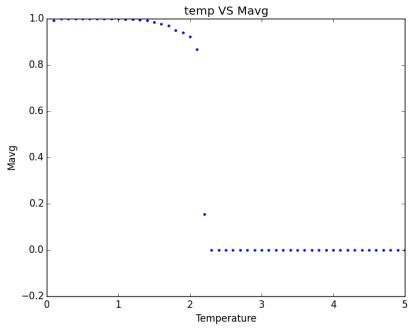

Figure 2: Results of Ising model with 10x10 matrix with 100 iterations, plot of T vs Mavg

the code fortran open MPI was used which stands for open source Message Passing interface.
When Parralelizing code it is important to think about how you are going to break the task up and into how many tasks, you also must know the hardware limitations and how many threads it can have before breaking down the problem. The CPU used had 4 threads so the Ising model was broken down into 4 different tasks. Since the Ising model was programmed to do a set number of iterattions on each temperature, it was parralized by dividing the different temperatures evenly among each thread (Important to note that, if there wasn't a set number of iterations and wanted to gain the uncertaintly in measurements instead of compare speeds, this would be ineffecient since lower temperatures would take a longer time).
For the actual coding using Open MPI, you must put "include 'mpif.h'" at the top to make sure the library is included and then the library is initialized using:

```
call MPI_INIT(ierr)
if (ierr .ne. MPI_SUCCESS) then
   print *,'Error starting MPI program. Terminating.'
   call MPI_ABORT(MPI_COMM_WORLD, rc, ierr)
end if
```

where ierr is a variable declared before hand, the if statement is there if it declares to initialize. To get the number of processors and the rank use:

```
call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
```

MPU_COMM_WORLD is the comminicator for the process's, MPI_COMM_SIZE gets the number of tasks available and MPI_COMM_RANK gives an integer number to the task from 0 to tasks -1. Each projecessor runs through the code on its own, and each processor can be specified using its rank, for example:

```
if(rank==0)then
```

```
    do j = 1, N
                do k = 1,N
                        CALL RANDOM_NUMBER(num)
                        if(num<0.5) Then
                                arr(k,j) = 1
                        Else
                                arr(k,j) = -1
                        END IF
                end do
        end do
end if
```

This code sets the first processor(rank==0) to set up the Matrix for the Ising model. The Ising model was then parralized using

```
!Initial temperature
temp = rank/10.
```

At the end of the program add :

```
call MPI_FINALIZE(ierr)
```

to release the resources used. The program is then compiled using:

```
mpif90 Isingmpi.f90 -o Isingmpi
```

and ran using :

```
mpirun -np 4 ./Isingmpi
```

where the 4 in the run command is how many threads you want to use. The data from each thread was then output into its own file by using the rank value within the write function and recombined when plotting.

```
write(11+rank,*), Eavg
write(15+rank,*), Mavg
```

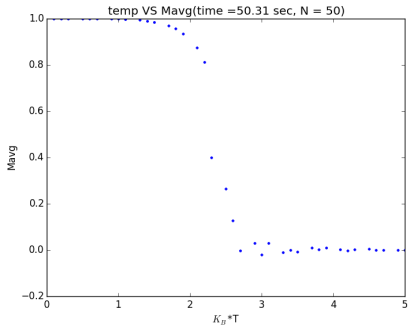

Figure 3: Plot of $k_N T$ vs Mavg using fortran openmpi

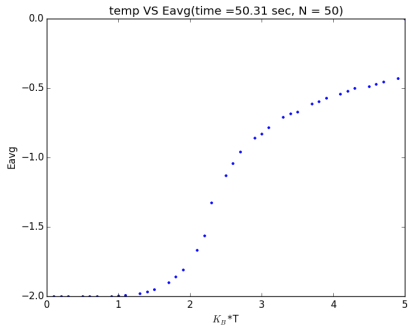

Figure 4: Plot of $k_N T$ vs Eavg using fortran openmpi

6 Parrallization using GPGPU

While CPU's are fast they have a small amount of threads, where as GPU's have been designed to deal with a bunch of calculations at the same time for use in video games. Recently there has been an emergent of also using GPU's for running computer programs that can be highly parallized, due to the high amount of threads available which has been given the name GPGPU(general puprose computing on graphic processing units). With this trend NVIDIA, a company that makes GPU's released tools for using their GPU's as GPGPU in the form of their CUDA extension to languages.
For this project CUDA-C was used to test the capabilities of GPGPU's. Parallizing the code using CUDA is a bit different then using multiple CPU cores, since the GPU is it's own device and it doesn't communicate to other hardware while it is running the program, so after including the cuda library with #include ¡cuda.h¿ you must use the built in functions to copy the variables from the cpu to the gpu:

```
cudaMalloc(&d_Arr, arrsize);
cudaMemcpy(d_Arr, h_Arr, arrsize, cudaMemcpyHostToDevice);
```

cudaMalloc allocates the memory in the variable d_Arr with memory size arrsize on to the device, then the function cudaMemcpy transfers the data from h_arr into the variable d_Arr and the cudaMemcpyHostToDevice declares to copy it from the cpu to the gpu. Once all variables you want are copied onto the device you have to determine how many threads you want. The device has threads put into blocks and then blocks put into a grid: Each thead is aware of its position in the grid and can be specificied using blockIdx,blockIdy,blockIdz, same for threadIdx,threadIdy,threadIdz. Depending on hardware the max threads each blocks can have is between 512 or 1024. If more threads are needs the problem must be broken down using more blocks. The dimensions of your block and grid can be assigned using:

```
dim3 dimBlock( 51 );
```

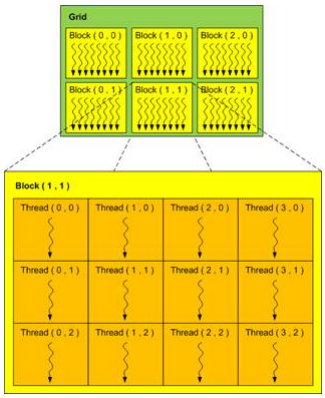

Figure 5: Show's how CUDA breaks down threads using grids and blocks

```
dim3 dimGrid( 1 );
```

Where the variable dimBlock is accepting an (x,y,z) value, if a coordinate is left out it is implied one. In the example the block has 51 threads in the x and 1 in both y,z. The same reasoning goes for the dimGrid variable with an implied dimenions of x,y,z=1. These are then used when you call the cuda function by placing them in <<<>>> brackets as seen below:

```
Ising<<<dimGrid, dimBlock>>>(d_Arr, d_E, d_N, states);
```

This calls the functions Ising that is on the cuda device taking the dimenions of the variable dimGrid and dimBlock and with the variable d_arr,d_E,d_N, and states. The device functions are called kernels and they have an indicators in front of them ___global___ or ___device___, global functions can be run from your main program where as device functions can only be ran within the device.

```
__global__ void Ising(int *arr, float *Earr,int *N,curandState_t* states)
{
  int itt = 100;
  int k = 0;
  int x = threadIdx.x;
  int y =(int ) *N;
  for (k = 0; k<itt; k++){
        Earr[x] += flip(y,arr,states);
  }
}
```

This is an example of a cuda kernel, each thread runs this code and can addressed using the blockId and threadId variables. In this case there was only one block with all threads in the x they are addressed using threadIdx. Each thread runs the device function flip and puts the value returned into the corresponding spot. The Ising model is a monte carlo and requires the use of random numbers, but the gpu kernels can't run normal cpu functions such as rand() to over come

this cuRAND must be included and used. To use CuRAND a global function is made to take in input for the seed and save the states for each thread.

```
__global__ void init(unsigned int seed, curandState_t* states) {

  /* we have to initialize the state */
  curand_init(seed, threadIdx.x, 51,&states[threadIdx.x]);
}
```

This function is then called from the main code with a allocatable variable for the states and using the time for the seed.

```
/* allocate space on the GPU for the random states */
cudaMalloc((void**) &states, N * sizeof(curandState_t));

/* invoke the GPU to initialize all of the random states */
init<<<1, 51>>>(time(0), states);
```

There are many other functions other than rand() that can't be called in the kernel and some are hardware dependent. Such as the printf() function, if you have a device with cuda compute capability 2.0 and greater then you can use it in the gpu kernel if not cuprintf can be download and used instead
Another problem that can be ran into has to do with operating systems "watchdog" timer which will return a malfunction and crash your computer if it doesn't constantly get feedback from a display device. So if your GPU is also used for display, you can only run short programs or break your program up into small parts to avoid this situation, or alternatively have a second gpu for programming.

7 Results

Below is a plot and table of some data points of the time taken by each method for an N by N array with 100 iterations. From these results it can be seen that the fortran Open MPI is the fastest, and is very close to 4 times faster then the regular fortran which means the processors and about equal in sharing the work. Even more interesting is the results for the CUDA-C where it is actually running slower then regular serial code fortran. This is probably due to using a very low end GPU and only using 51 threads for the problem.

NxN	Python	C	fortran	OpenMPI	CUDA C
10	29.83	0.065	0.068	0.023 s	0.0353 s
12	66.9 s	0.091 s	0.088 s	0.025 s	0.050 s
14	98.8 s	0.125 s	0.120 s	0.035 s	0.068 s
16	164.3 s	0.212 s	0.157 s	0.042 s	0.089s
18	252.9 s	0.272 s	0.199 s	0.053 s	0.114 s
20	382.7 s	0.327 s	0.245 s	0.065 s	0.139 s
22	N/A	0.395 s	0.296 s	0.079 s	0.172 s
24	N/A	0.470 s	0.353 s	0.094 s	0.208 s
26	N/A	0.556 s	0.411 s	0.110 s	0.243
28	N/A	0.646s	0.480 s	0.128 s	0.285

Using C as a bases it can be seen that fortran is about the same speed to 1.3 times faster, cuda is about 1.8 to 2.0 faster and fortran OpenMPI is about 2.8 to five times faster. The difference of speed greatly increases with calculations as seen with the bigger matrix's favoring the parralel programming the most.

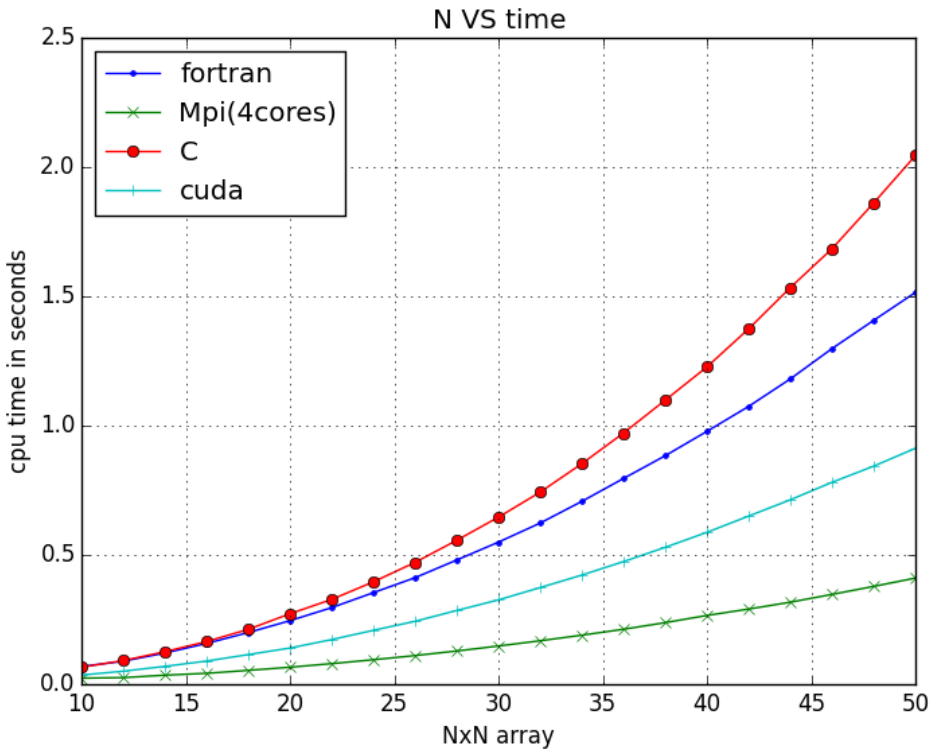

Figure 6: Comparison of time vs size of matrix for different methods(python wasn't include due to huge difference in time)