Parallelizing Matrix Multiplication

Raghavendr Galeppa
1PE15CS113

Nithesh MS
1PE15CS100

March 19, 2018

Assignment Report for Multi-Core Architecture and Programming(15CS666)

1 Introduction

Nowaday's, the computers being sold commercially usually contain more than one core for a processor. In order to make use of these processor cores efficiently, Parallel programming paradigm's exist. Parallel Programming is executing a task in more than one processore core at the same time in parallel. Theoritically Parallel Programming is a very simple concept however it is operationally much more difficult to achieve. There are many things that the programmer must take care of while writing parallel programs such as Deadlocks, Thread Synchronization, Program partitioning, Load Balancing, Data Dependencies, Communication etc. In general programs write Parallel Code to achieve Performance improvement and efficiency from the programs.

There are two forms of parallelism that exists. The first one is called **Function Parallelism** where each thread or process executes different code sections that are independent of each other. *Cilk Plus* is somewhat based on function parallelism. The other form of parallelism is called **Data Parallelism** where each process or thread does the same work on unique and independent pieces of data. *OpenMP* and *MPI* provide Data Parallel programming environments. Data Parallelism is more scalable than Functional Parallellism

There are many ways one can write parallel programs. Many API's have been designed for Parallel Programming such as

- OpenMP
- POSIX threads
- MPI (Message Passing Interface)
- Java Threads

1.1 OpenMP

OpenMP is a shared-memory application programming interface(API) whose features are based on efforts to facilitate shared-memory parallel programming. OpenMP in itself is not a programming language but a notation that can be added to a sequential program in *FORTRAN, C, C++* to describe how the work is shared among the threads that will execute on different processors and to order the access to shared data. OpenMP's directives tell the compiler which instructions to execute in parallel and how to distribute them among the threads that will run the code.

A **pragma** directive is used to provide the instructions to tell the compiler which sections to parallelize. The format is,

```
#pragma omp parallel
{
   // Block of code to be run in parallel

}
```

1.2 POSIX Threads

For UNIX systems, a standardized C language threads programming interface has been specified by the *IEEE POSIX 103.c* standard. Implementations that adhere to this standard are referred to

as **POSIX Threads** or **Pthreads**. Technically a *Thread* is defined as an independent stream of instructions that can be scheduled to run as such by the Operating System. Unlike processes, all threads created by a process share the same virtual memory address and are less of an overhead to create than processes. The subroutines provided by PThreads can be grouped into four major groups: **Thread Management, Mutexes, Condition variables, Synchronization**.

The following code shows how to crate threads using Pthreads in C:

```
#include<pthread.h>  /* The header file for pthreads*/
#define NUM_THREADS 4
.
.
void *parallelFunc(void *param) {
   // Body of the function
   pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
   ....
   ....
   int threads[NUM_THREADS];
   long x = 0;
   for(x = 0; x<NUM_THREADS; x++) {
      int rc = pthread_create(&threads[i], NULL, parallelFunc, (void *)x);
      if(rc) {
         printf(``ERROR: unable to create thread!'');
         exit(1);
      }
   }
   ...
   ...
   ...
}
```

2 Matrix Multiplication

Matrix Multiplication is multiplying two given matrices of sizes M1=m*n and M2=p*q with number of rows of first matrix M1 i.e, m and the no.of columns in second matrix M2 and m is equal to q. You cannot multiply two matrices if they don't satisfy the condition that number of rows in first matrix is equal to number of columns in the second matrix.

Example: If you multiply the following two matrices

$$\begin{bmatrix} 3 & 6 & 7 \\ 5 & 3 & 5 \\ 6 & 2 & 9 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 7 \\ 0 & 9 & 3 \\ 6 & 0 & 6 \end{bmatrix}$$

you get,

$$\begin{bmatrix} 55 & 26 & 80 \\ 63 & 33 & 72 \\ 54 & 48 & 96 \end{bmatrix}$$

2.1 Sequential Code for Matrix Multiplication

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
int m;
int n;
void matrix_multiply();
struct matrix {
   double *a;
   double *b;
   double *c;
} p;
int main(int argc, char *argv[])
{
   int i, j;
   double elapsed;
   clock_t t;
   m = atoi(argv[1]);
   n = atoi(argv[2]);
   if(!(p.a = (double *)malloc(sizeof(double)*m*n)))
   printf("ERROR\n");
   p.b = (double *)malloc(sizeof(double)*m*n);
   p.c = (double *)malloc(sizeof(double)*m*n);


   for(i = 0; i<m; i++)
```

```
      for(j = 0; j<n; j++)
         p.a[i*n+j] = 0.0;

   for(i = 0; i<m; i++)
      for(j = 0; j<n; j++)
         p.c[i*n+j] = rand()%10;

   for(i = 0; i<m; i++) {
      for(j =0; j<n; j++) {
         p.b[i*n+j] = rand()%10;
      }
   }
   t = clock();
   matrix_multiply();
   t = clock() - t;
   elapsed = (double)t/CLOCKS_PER_SEC;

   printf("\n TIME TAKEN IN s: %f\n",elapsed);
   free(p.a);
   free(p.b);
   free(p.c);
}

void matrix_multiply()
{
   int i, j = 0, k = 0;
   for(i = 0; i<m; i++)
      for(j = 0; j<n; j++) {
         for(k = 0; k<n; k++)
         p.a[i*n+j] += p.b[i*n+k] * p.c[k*n+j];
      }
}
```

2.2 Approach to solve the problem

We first browsed through the OpenMP official website to learn about multi-core architecture, how parallel processes are run on a multi-core architecture. Then we took a sample C program which executes a sample matrix multiplication problem. We then took that matrix code and modified it by adding the OpenMP libraries and OpenMP functions. We further added a timer function to calculate the total time it took for the processor to compute the final matrix. We then ran into a problem, which was we were unable to give a matrix input beyond 1000*1000. We ran into an error called Segmentation fault. So we had to dynamically allocate memory for the matrix and then free the space we created to solve this problem so that the program, when it is ran, accepts matrix sizes of very large inputs. This mechanism is necessary for our purposes because with large input matrix sizes it is clearer to see the differences in speedup of our OpenMP algorithm versus matrix multiplication without openMP algorithm.

3 Parallelizing Matrix Multiplication

The calculations involved in Matrix Multiplication for each item are completely independent of each other and hence can be parallelized.

3.1 Using POSIX Threads

The following is the code for parallelizing Matrix Multiplication using pthreads.

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<time.h>
#define NUM_THREADS 10
long x;
void *matrix_multiply();
struct matrix {
   double *a;
   double *b;
   double *c;
} ;
int m;
int n;
int task[NUM_THREADS+1];
struct matrix p;
int main(int argc, char *argv[])
{
   struct timespec start, finish;
   double elapsed;
   int i, j;
   pthread_t threads[NUM_THREADS];
   m = atoi(argv[1]);
   n = atoi(argv[2]);
   if(!(p.a = (double *)malloc(sizeof(double)*m*n)))
      printf("\n ERROR\n");
   p.b = (double *)malloc(sizeof(double)*m*n);
   p.c = (double *)malloc(sizeof(double)*m*n);


   for(i = 0; i<m; i++)
      for(j = 0; j<n; j++)
         p.a[i*n+j] = 0.0;

   for(i = 0; i<m; i++)
      for(j = 0; j<n; j++)
         p.c[i*n+j] = rand()%10;

   for(i = 0; i<m; i++) {
      for(j =0; j<n; j++) {
         p.b[i*n+j] = rand()%10;
```

```
      }
   }
   long l;
   for(i = 0; i<NUM_THREADS; i++) {
      task[i] = i*m/NUM_THREADS;
   }
   task[i] = m;
   clock_gettime(CLOCK_MONOTONIC, &start);
   for(l= 0;l<NUM_THREADS; l++) {
      pthread_create(&threads[l],NULL, matrix_multiply,(void *)l);
   }
   for(l= 0;l<NUM_THREADS; l++) {
      pthread_join(threads[l], NULL);
   }
   clock_gettime(CLOCK_MONOTONIC, &finish);
   elapsed = (finish.tv_sec - start.tv_sec);
   elapsed += (finish.tv_sec - start.tv_sec)/1000000000.0;
   for(i = 0; i<m; i++) {
      for(j = 0; j<n; j++) {
         printf("%f ",i,j,p.c[i*n+j]);
      }
      printf("\n");
   }
   printf("\n");
   for(i = 0; i<m; i++) {
      for(j = 0; j<n; j++) {
         printf("%f ",i,j,p.b[i*n+j]);
      }
      printf("\n");
   }
   printf("\n");
   for(i = 0; i<m; i++) {
      for(j = 0; j<n; j++) {
         printf("%f ",i,j,p.a[i*n+j]);
      }
      printf("\n");
   }
   printf("\n-- TIME TAKEN IN sec: %f\n",elapsed);

   free(p.a);
   free(p.b);
   free(p.c);
}

void *matrix_multiply(void *x)
{
   int k = 0;
   long i = task[(long)x];
   int j;
   for(; i<task[(long)x+1]; i++) {
```

```
      for(j = 0; j<n; j++) {
         for(k = 0; k<n; k++) {
            p.a[i*n+j] += p.b[i*n+k] * p.c[k*n+j];
         }
      }
   }
}
```

3.2 Working of the code

The program takes the matrix size as command line arguments. It then creates two matrices of same size with each element containing a random number from 0-10. The total no.of rows are divided by the number of threads available and then each thread is assined a set number of rows to handle. The following code segment does exactly that:

```
for(i = 0; i<NUM_THREADS; i++) {
   task[i] = i*m/NUM_THREADS;
}
```

Each thread is then created and put into action by the following code:

```
for(l= 0;l<NUM_THREADS; l++) {
   pthread_create(&threads[l],NULL, matrix_multiply,(void *)l);
}
```

3.3 Results(Pthreads)

The following results are taken from a computer running Intel i5 4th Gen CPU.

Input size n*n	Sequential Code time in secs	Parallelized Code time in sec
500x500	1.04	1.00
800x800	5.96	3.00
1000x1000	11.53	6.00
1500x1500	44.42	23.00
2000x2000	105.90	53.00

From the results it is clear that the use of threads has increased the perfomance 2 times for inputs other than the first one.

3.4 Using OpenMP

The following section shows the code for parallelizing Matrix Multilication using OpenMP

```
#include <stdlib.h>
#include <stdio.h>
```

```
#include <omp.h>
int main(int argc, char *argv[])
{
   int **a,**b,**c;
   int a_r,a_c,b_r,b_c, nthreads, tid, chunk =10;

   double dif;

   int i,j,k;

   again:
   printf("\nenter rows and columns for matrix A:");
   scanf("%d%d",&a_r,&a_c);
   printf("\nenter rows and columns for matrix B:");
   scanf("%d%d",&b_r,&b_c);
   if(a_c!=b_r )
   {
   printf("\ncan not multiply");
   goto again;
   }

   a=(int **) malloc(10*a_r);
   for( i=0;i<a_c; i++)
   {
      a[i]=(int *) malloc(10*a_c);
   }

   b=(int **) malloc(10*b_r);
   for( i=0;i<b_c; i++)
   {
      b[i]=(int *) malloc(10*b_c);
   }

   c=(int **) malloc(10*a_r);
   for( i=0;i< b_c; i++)
   {
      c[i]=(int *) malloc(10*b_c);
   };
   double start = omp_get_wtime( );

   #pragma omp parallel shared(a,b,c,nthreads,chunk) private(tid,i,j,k)
   {
      tid = omp_get_thread_num();
      if (tid == 0)
      {
         nthreads = omp_get_num_threads();
         printf("Starting matrix multiple example with %d threads\n",nthreads);
      }

   #pragma omp for schedule (static, chunk)
```

```
      for(i=0;i<a_r; i++)
      {
         for(j=0;j<a_c; j++)
         {
            a[i][j] = i+j;
         }

      }

   #pragma omp for schedule (static, chunk)
      for(i=0;i<b_r; i++)
      {
         for(j=0;j<b_c; j++)
         {
            b[i][j] = i*j;
         }
   }

   #pragma omp for schedule (static, chunk)
      for(i=0;i<a_r; i++)
      {
         for(j=0;j< b_c; j++)
         {
            c[i][j]=0;
         }
      }

   printf("Thread %d starting matrix multiply...\n",tid);
   #pragma omp for schedule (static, chunk)
   for(i=0;i<a_r; i++)
   {

      for(j=0;j<a_c; j++)
      {
         for(k=0;k<b_c; k++)
         {
            c[i][j]=c[i][j]+a[i][k]*b[k][j];
         }
      }
   }
}

   printf ("Done.\n");
   double end = omp_get_wtime( );
   dif = end - start;
   printf("Parallelization took %f sec. time.\n", dif);

   for(i=0;i<a_r; i++)
   {
      free(a[i]);
```

```
   }
   free(a);
   for(i=0;i<a_c; i++)
   {
      free(b[i]);
   }
   free(b);
   for(i=0;i<b_c; i++)
   {
      free(c[i]);
   }
   free(c);
}
```

3.5 Results(OpenMP)

The following results for runnin Matrix Multiplication in parallel using OpenMP.

Input size n*n	**Sequential Code time in secs**	**Parallelized Code time in sec**	**Speed increase**
500x500	0.808	.561	1.44x
800x800	5.352	2.997	1.7857x
1000x1000	10.41	5.686	1.8322x
1500x1500	39.04	20.12	2.345x
2000x2000	100.47	50.41	2.000x

3.6 Interpretation

BY MANUALY CHANGING NUMBER OF THREADS You can set the number of threads to one.; the computation of the result matrix takes longer using OpenMP with one thread than it does without OpenMP. Another interesting point to note, as we have seen based on the results, as we increase the number of threads, with OpenMP, the speedup is significantly faster than in comparison to using only four threads. When more than four threads are used, the result matrix is computed sequentially rather than in parallel. With four threads, each core is assigned a thread with OpenMP. However, with more than four threads, each core is assigned multiple threads.

OPENMP RESULT As expected the speedup is up to four times with the small amount of trials we have conducted our best speedup was about 3.578 times, which is almost close to the ideal 4 times using four cores of a processor to compute the final matrix instead of one. Had we chosen to include even a larger matrix, based on the graphs and tables, the speedup would have been as close to 4 times. Therefore, as the number of input size approaches infinity the speedup of our algorithm with OpenMP in comparison to without OpenMP would approach 4 times, as expected.

4 References

1. *Portable Shared Memory Parallel Programming* by Barbara Chapman, Gabreile Jost and Ruud Van Der Pas.

2. *Multi-Core Programming* by Shameen Akhter and Jason Roberts.

3. https://openmp.org, official *OpenMP* website.

4. https://computing.llnl.gov/tutorials/pthreads/ , A tutotrial website for learning *PThreads*.

5. https://en.wikipedia.org/wiki/OpenMP , Wikipedia page for *OpenMP*.

6. https://en.wikipedia.org/wiki/POSIXThreads , Wikipedia page for *Pthreads*.

7. https://standards.ieee.org/findstds/interps/1003-1c-95int/index.html , IEEE Std 1003.1c. IEEE Standard for Portable Operating System Interface(POSIX) Threads