ELEC6230 Homework 6

Answers to Parallel Processing Homework 6

Michael J. Carroll

October 15, 2010

Question 1 Results

I created the sequential C++ program included below in Appendix A.

The execution time was significantly less than the matrix multiplication, so I choose to run the program several times to come up with an average execution time. The matrix multiplication would have been on the order of n^3, while the convolution problem is on the order of n^2. Even sequentially, convolution is much faster than matrix multiplication. The times shown below in Table 1 are in milliseconds.

Question 2 Results

I created the parallel C++ program included below in Appendix B.

Looking at the average execution time for each number of processing elements in Table 2 yields interesting results. The speedup seems relatively constant until the 6th processor is added. While the mode of the execution time is 16, the mean ends up being 32 milliseconds. This is because of some large outliers in the data. The results based on the test are in the table, but if I assume that the execution time of the 6 processing element condition is 16 milliseconds, this brings speedup to 5.58, and efficiency to 93%.

I'm not sure what in the algorithm would be causing these sudden jumps in the time. My guess is that using all 6 processors available in interactive mode means that there will be resource conflicts with other users on the system. This will cause the anomalous spikes in the timing. To test this theory, I also submitted my algorithm to the batch job queue (less chance of interference from other active users). I found that the batch job had equally random timings when adding the 6th PE.

This leads me to believe that there may actually be something wrong with my algorithm, but I have not been able to find it.

Run	1	2	3	4	5	6	7	8	9	10	**Average**
Time (ms)	75	26	50	79	29	50	75	26	78	28	**51.2**

Table 1: Execution Time in Seconds for 1 PE, Sequential

Run	1 PE (ms)	2 PE (ms)	3 PE (ms)	4 PE (ms)	5 PE (ms)	6 PE (ms)
1	89	46	31	23	19	16
2	89	46	31	23	19	36
3	90	46	31	24	19	16
4	89	46	31	23	19	102
5	89	46	31	24	19	16
6	90	46	31	24	19	16
7	89	46	31	23	28	16
8	91	46	31	23	19	16
9	90	46	31	23	19	16
10	89	46	31	24	19	68
Average	89	46	31	23	20	32
Speedup	1.00	1.95	2.90	3.82	4.49	2.8
Efficiency	100%	97.5%	96.8%	95.4%	89.8%	46.5%
Average(outliers removed)	89	46	31	23	20	16
Speedup	1.00	1.95	2.90	3.82	4.49	5.58
Efficiency	100%	97.5%	96.8%	95.4%	89.8%	93%

Table 2: Speedup and Efficiency for 1-6 PEs, Chunk Size: 10

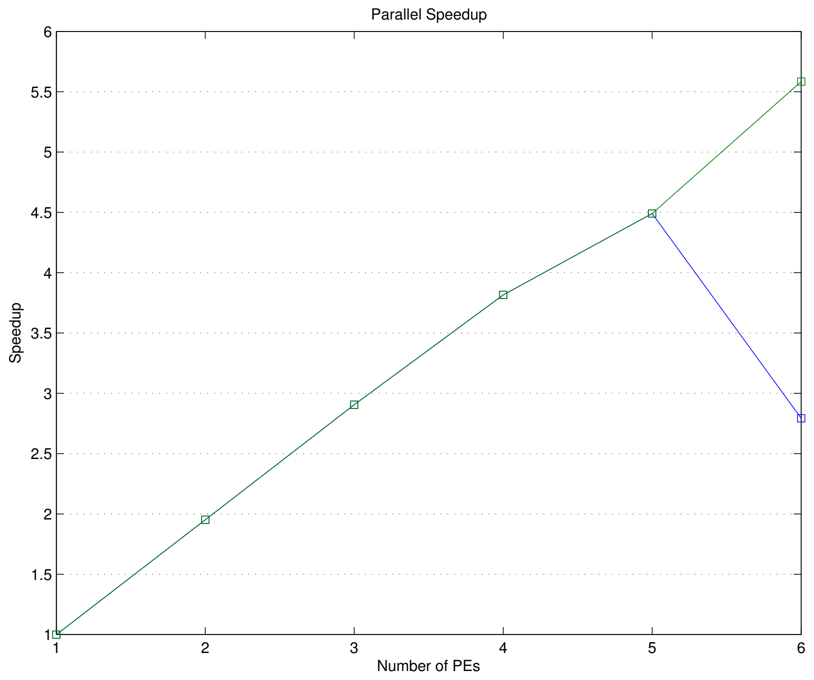

Figure 1: Speedup Plot

Appendix A - Sequential Operation

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

#define matrixSize 2160

main()
{
float a[matrixSize][matrixSize],
  a_new[matrixSize][matrixSize];
int i,j,m,n;
float error = 0;
struct timeval t0,t1,t2,t3;

/* Made a kernel matrix.  For this assignment it won't really matter*/
float kernel[3][3] = {{1,1,1},{1,1,1},{1,1,1}};
float kernelweight = 9;
float accum;

/* Create a matrix with entry 9, if j is even, 0 if odd */
gettimeofday(&t0,0);

for (i=0;i<matrixSize;i++)
    for (j=0;j<matrixSize;j++)
       a[i][j] =j%2?9:0;

gettimeofday(&t1,0);
for(i=0; i<matrixSize;i++){
    for(j=0;j<matrixSize;j++){
        accum = 0;
        for(m=-1;m<=1;m++){
            for(n=-1;n<=1;n++){
                /* Handle boundary conditions:
                 * An alternative approach would be to pad the entire outside
                 * of the matrix with zeros.  I chose to use more computational
                 * time than memory for this program */
                if((i+m)<0 || (i+m)>=matrixSize ||(j+n)<0 || (j+n)>=matrixSize)
                    accum += 0;
                else
                    accum += a[i+m][j+n];
            }
        }
        a_new[i][j]=accum/kernelweight;
    }
}
/* Set the old matrix equal to the new matrix, as per the project
 * requirementsi
 */
**a = **a_new;
gettimeofday(&t2,0);

float totalInt = t2.tv_sec - t0.tv_sec + (t2.tv_usec - t0.tv_usec)*1.0E-06;
float setupInt = t1.tv_sec - t0.tv_sec + (t1.tv_usec - t0.tv_usec)*1.0E-06;
float convInt = t2.tv_sec - t1.tv_sec + (t2.tv_usec - t1.tv_usec)*1.0E-06;

/*
printf("Total Execution Time:\t%e\n",totalInt);
printf("Setup Time:\t\t%e\n",setupInt);
printf("Convolution Time:\t%e\n",convInt);
*/
printf("%e\n%e\n%e\n",totalInt,setupInt,convInt);
}
```

Appendix B - Parallel Operation

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

#define matrixSize 2160

main(int argc, char *argv[])
{
float a[matrixSize][matrixSize],a_new[matrixSize][matrixSize];
int i,j,m,n;
int numOfThreads = 1;
double w0,w1,w2,setupInt,totalInt,convInt;

/* Made a kernel matrix.  For this assignment it won't really matter*/
float kernel[3][3] = {{1,1,1},{1,1,1},{1,1,1}};
float kernelweight = 9;
float accum;

/* Create a matrix with entry 9, if j is even, 0 if odd */
if (argc==2)
    numOfThreads = atoi(argv[1]);

omp_set_num_threads(numOfThreads);

w0 = omp_get_wtime();

#pragma omp parallel for
for (i=0;i<matrixSize;i++){
    for (j=0;j<matrixSize;j++){
       a[i][j] =j%2?9:0;
    }
}

w1 = omp_get_wtime();

#pragma omp parallel for
for(i=0; i<matrixSize;i++){
    for(j=0;j<matrixSize;j++){
        accum = 0;
        for(m=-1;m<=1;m++){
            for(n=-1;n<=1;n++){
                /* Handle boundary conditions:
                 * An alternative approach would be to pad the entire outside
                 * of the matrix with zeros.  I chose to use more computational
                 * time than memory for this program */
                if((i+m)<0 || (i+m)>=matrixSize ||(j+n)<0 || (j+n)>=matrixSize)
                    accum += 0;
                else
                    accum += a[i+m][j+n];
            }
        }
        a_new[i][j]=accum/kernelweight;
    }
}
/* Set the old matrix equal to the new matrix, as per the project
 * requirementsi
 */
**a = **a_new;

w2 = omp_get_wtime();

totalInt = w2 - w0;
setupInt = w1 - w0;
convInt = w2 - w1;

//printf("Total Execution Time:\t%e\n",totalInt);
//printf("Setup Time:\t\t%e\n",setupInt);
printf("%e\n",convInt);
}
```

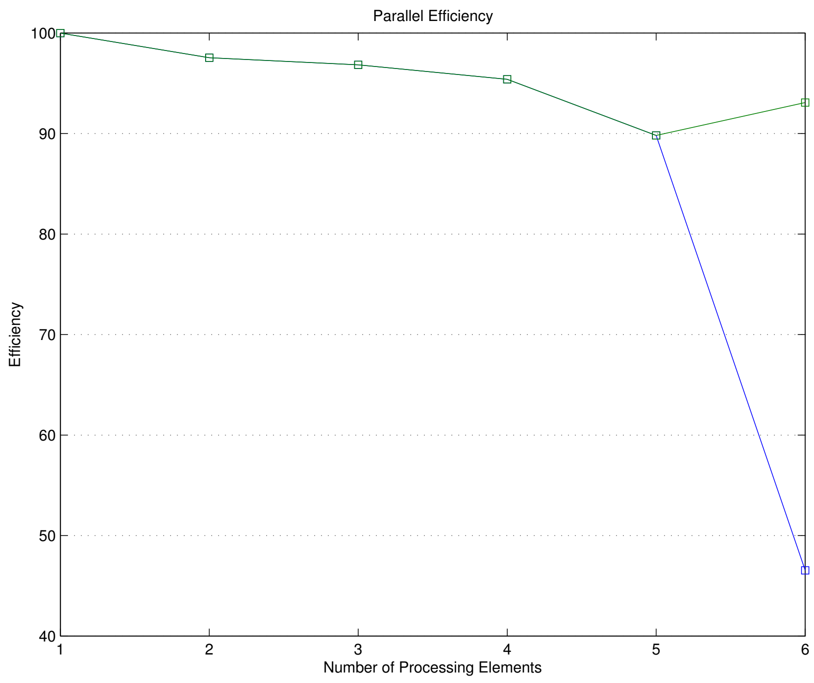

Figure 2: Efficiency Plot