2016 Hawaii University International Conferences
Science, Technology, Engineering, Art, Math & Education June 10 - 12, 2016
Hawaii Prince Hotel Waikiki, Honolulu

Parallel Programming Multi- core Computers

Alaghband, Gita
University of Colorado Denver
Computer Science & Engineering Department

Fardi, Hamid
University of Colorado Denver
Electrical Engineering Department

Prof. Gita Alaghband
Computer Science and Engineering Department
University of Colorado Denver.

Prof. Hamid Fardi
Electrical Engineering Department
University of Colorado Denver.

Parallel Programming Multi-core Computers

Synopsis:

With advances in hardware technology, we as educators find ourselves with multi-core computers as servers, desktops, and personal computers in our laboratories while teaching students how to design for sequential environments. We propose to develop pedagogy for teaching undergraduate students how to develop software and design algorithms for multi-core architectures in a laboratory setting. The key success and challenge of this project will be helping students to think in parallel again.

Parallel Programming Multi-core Computers

Gita Alaghband[1], Lan Vu[1], H. Z. Fardi[2]

[1]University Of Colorado Denver
Department of Computer Science & Engineering
Campus Box 109, P.O. Box 173364, Denver, CO 80217-3364

[2]University Of Colorado Denver
Department of Electrical Engineering
Campus Box 110, P.O. Box 173364, Denver, CO 80217-3364

Abstract With advances in hardware technology, we as educators find ourselves with multi-core computers as servers, desktops, and personal computers in our laboratories while teaching students how to design for sequential environments. We propose to develop pedagogy for teaching undergraduate students how to develop scientific computing programs for multi-core architectures in a laboratory setting. The key success and challenge of this project will be helping students to think in parallel.

Introduction and Motivation

It is time to develop methodologies for infusion of parallel computing and programming at the undergraduate computer science and engineering curriculum level. The two decades of the eights and nineties were the height of supercomputing for machine design, parallel compilers, languages and algorithm design involving select group of highly specialized scientists with interest in high performance computing and number crunching applications. The surprising thing today is not that parallel computers and computation are attracting increased attention, but that so much of computing has revolved around a sequential, one thing at a time, style of software design. Today, with all the advances in hardware technology, we as educators find ourselves with multi-core computers as servers, desktops, personal computers, and even handheld devices in our laboratories while still teaching students how to design system software, algorithms and programming languages for sequential environments.

We propose to develop methodologies for teaching undergraduate students how to develop software and design algorithms for multi-core architectures for a scientific computing course in a laboratory setting. The key success and challenge of this project will be helping students to "think in parallel" [1]. We will develop material for junior-level computer science and engineering student who has already had some programming language courses.

The major intellectual challenge of this project is in developing strategies and material that will help students overcome the conditioning of thinking sequentially with which they are familiar. To accomplish this, concepts of global parallelism will be integral in our development.

Parallel Design Concepts through an Example

We introduce the idea of "global parallelism" through two versions of parallel Gaussian Elimination implementation implemented in OpenMP C++. The choice of Gaussian elimination is due to its vast applicability to most areas of science. It is used to solve systems of linear equations that exist in engineering, chemistry, physics, economics and others with applications such as circuit simulation, fluid dynamics simulation, weather modeling, molecule dynamics, etc. The choice for OpenMP is due to being the most common parallel language platform used within current shared memory multicore computer architectures. OpenMP is not a full language by itself; it is designed as an extension to a sequential programming language to support shared memory MIMD programming. OpenMP extenstions exist for C, C++, and Fortran [2].
OpenMP supports a form of fork/join semantics. Execution starts with a single sequential process that forks a fixed number of *threads* upon encountering a *parallel region*. The *team* of threads will then execute the body of the parallel region and are joined back into the original process (single stream). Every time a new parallel region is encountered a different number of threads may be forked, but the number remains fixed within each parallel region. Three user specified environment variables manage parallel execution:

- num_threads (integer): specifies the number of threads,
- dynamic (Boolean): specifies whether the number of threads can change from one parallel region to the next, and a second Boolean,
- nested (Boolean): specifies whether nested parallelism is allowed.

The programs presented here use several OpenMP constructs that won't be described in order to focus on the parallel programming style. The language constructs are described in details in [2].
For the sake of simplicity and complete presentation, the code sections for the main program (Figure 1), matrix generation of a test matrix (Figure 2), and user input (Figure 3) are presented separate from the two different parallel implementations of the Gaussian elimination. This is done to ensure that the comparison is only on the impact of parallel style and not sequential method. Both programs are sequentially identical (and optimized). The main program calls the "*ComputeGaussianElimination*" function; two versions of this function are discussed in details as focus of this paper. The LU decomposition is performed by Gaussian elimination, leaving L and U factors stored in place of the original matrix, and prints the time for performing the elimination and, optionally, the L and U factors. The execution time is calculated from calls to omp_get_wtime(), assumed to return time in seconds for "*ComputeGaussianElimination*" function. Forward and back substitution for solving equation sets with specific right hand sides using the L and U factors are not included in this program.
For ease of readability, the OpenMP constructs are highlighted in bold green and the related parallel comments are presented in red. Function calls are highlighted in in purple.
The matrix is initialized in parallel in Figure 2. The created threads within the "**pragma omp parallel"** region work on different rows, I, in parallel generating the matrix elements; the inner loop over columns are done sequentially within each parallel thread.

```
// Main Program
int main(int argc, char *argv[])
	{int n, numThreads, isPrintMatrix;
	float **a;
	double runtime;
	bool isOK;
	if (GetUserInput(argc,argv,n,numThreads,isPrintMatrix)==false) return 1;
	//specify number of threads created in parallel region
	omp_set_num_threads(numThreads);
	InitializeMatrix(a,n);			//Initialize the value of matrix A[n x n]
	if (isPrintMatrix)
		{cout<< "The input matrix" << endl;
		PrintMatrix(a,n); }
	runtime = omp_get_wtime();
   //Compute the Gaussian Elimination for matrix a[n x n]
	isOK = ComputeGaussianElimination(a,n);
	runtime = omp_get_wtime() - runtime;
	if (isOK == true)		//The eliminated matrix is as below:
		{if (isPrintMatrix)
			{cout<< "Output matrix:" << endl;
			PrintMatrix(a,n); }
		//print computing time
		cout<< "Gaussian Elimination runs in "<< setiosflags(ios::fixed)
			<< setprecision(2)
				<< runtime << " seconds\n";}
	else	{cout<< "The matrix is singular" << endl;}
	DeleteMatrix(a,n);
	return 0;}
```

Figure 1. Main program

```
//Initialize the value of matrix a[n x n]
void InitializeMatrix(float** &a,int n)
	{a = new float*[n];
	a[0] = new float[n*n];
	for (int i = 1; i < n; i++) a[i] = a[i-1] + n;
	#pragma omp parallel for schedule(static)
	for (int i = 0 ; i < n ; i++)
		{for (int j = 0 ; j < n ; j++)
		{if (i == j)		a[i][j] = (((float)i+1)*((float)i+1))/(float)2;
		 Else			a[i][j] = (((float)i+1)+((float)j+1))/(float)2;}}}
void DeleteMatrix(float **a,int n)		//Delete matrix matrix a[n x n]
	{delete[] a[0];
	 delete[] a; }
void PrintMatrix(float **a, int n)		//Print matrix
	{for (int i = 0 ; i < n ; i++)
		{cout<< "Row " << (i+1) << ":\t" ;
		for (int j = 0 ; j < n ; j++)		{printf("%.2f\t", a[i][j]);}
		cout<<endl ;}}
```

Figure 2. Matrix is initialized by the program rather than input by the user

```
// Input User Data for Gaussian elimination program : C++ OpenMP
// Row-wise Data layout & Row-wise Elimination
#include <iostream>
#include <iomanip>
#include <cmath>
#include <omp.h>
#include <time.h>
using namespace std;
//   Get user input of matrix dimension and printing option
bool GetUserInput(int argc, char *argv[],int& n,int& numThreads,int& isPrint)
        {bool isOK = true;
        if(argc < 3)
                {cout << "Arguments:<X> <Y> [<Z>]" << endl;
                cout << "X : Matrix size [X x X]" << endl;
                cout << "Y : Number of threads" << endl;
                cout << "Z = 1: print the input/output matrix if X < 10" << endl;
                cout << "Z <> 1 or missing: do not print input/output matrix" << endl;
                isOK = false;}
        else    {//get matrix size
                n = atoi(argv[1]);
                if (n <=0)          {cout << "Matrix size must be larger than 0" <<endl;
                                    isOK = false;}
                numThreads = atoi(argv[2]);      //get number of threads
                if (numThreads <= 0)
                        {cout << "Number of threads must be larger than 0" <<endl;
                        isOK = false;}
//print the input/output matrix
                if (argc >=4)     isPrint = (atoi(argv[3])==1 && n <=9)?1:0;
                else      isPrint = 0;}
        return isOK;}
```

Figure 2. User input values and desired printing options

Two different implementations of "*ComputeGaussianElimination*" are presented (Figures 4 and 6) to illustrate the principle ideas between designing the parallel versions using global parallelism as opposed to applying parallelism when the code segments are observed to be candidates for parallel expression.

Figure 4 shows a parallelized version of the Gaussian elimination code with a style that is familiar and often used by sequential programmers and software engineers new to parallel computation. Figure 5 is the corresponding high-level flowchart depicting the parallelization and parallel operations taking place in the implementation style of Figure 4. A single stream of control characterizes the overall program structure, with parallel operations applied to large data structures with independent elements.

```
//Compute the Gaussian Elimination for matrix a[n x n], Version 1
bool ComputeGaussianElimination(float **a,int n)
{float pivot,gmax,pmax,temp;
int  pindmax,gindmax,i,j,k;
omp_lock_t lock;
omp_init_lock(&lock);
for (k = 0 ; k < n-1 ; k++)          //Perform rowwise elimination
        {gmax = 0.0;                 //Find the pivot row among rows k, k+1,...n
//The following OMP parallel statement will create parallel processes. Each thread works on a number of rows to find the //local max value pmax
        #pragma omp parallel shared(a,gmax,gindmax) firstprivate(n,k)
        private(pivot,i,j,temp,pmax,pindmax)
                {pmax = 0.0;
                #pragma omp for schedule(static)
                for (i = k ; i < n ; i++)
                        {temp = abs(a[i][k]);
                        if (temp > pmax)
                        {pmax = temp;
                        pindmax = i;}}
// Update private max value to global gmax one process at a time:
                #pragma omp critical
                        {if (gmax <= pmax)
                                {gmax = pmax;
                                gindmax = pindmax;      }}}
At this point all processes join into one stream.
        if (gmax == 0) return false;         //If matrix is singular set the flag & quit.
        if (gindmax != k)            //Swap rows if necessary
//The following OMP parallel statement will create parallel processes. Swap pivot row
        {#pragma omp parallel for shared(a) firstprivate(n,k,gindmax) private(j,temp)
        schedule(static)
        for (j = k; j < n; j++)
                {temp = a[gindmax][j];
                a[gindmax][j] = a[k][j];
                a[k][j] = temp;}}
// At this point all processes join into one stream.
                pivot = -1.0/a[k][k];        // Compute the pivot
//The following OMP parallel statement will create parallel processes to do row reductions
                #pragma omp parallel for shared(a) firstprivate(pivot,n,k) private(i,j,temp)
                schedule(dynamic)
                for (i = k+1 ; i < n; i++)
                        {temp = pivot*a[i][k];
                        for (j = k ; j < n ; j++)
                                {a[i][j] = a[i][j] + temp*a[k][j];}}}
// At this point all processes join into one stream.
        omp_destroy_lock (&lock);
        return true;}
```

Figure 4. Version 1 of OpenMP C++ ComputeGaussianElimination

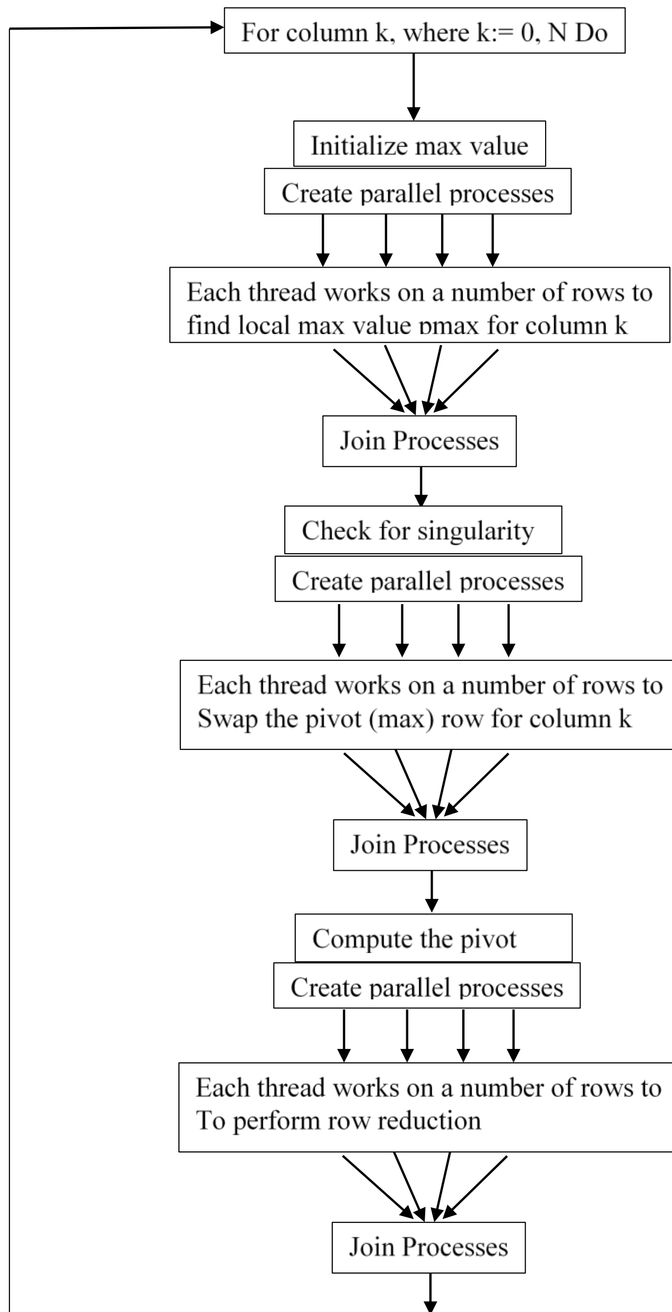

Figure 5. Parallelism corresponding to Version 1 (Figure 4)

In contrast in the implementation of Figure 6, the entire team of processes work over the entire flow of control rather than at specific points in the computation [3]. In this version, parallel processes are created once at the entry to the "**#pragma omp parallel**" region at the beginning of the function and are joined at the end of the region which is the last statement of the function before return. If sequential operation is required by the program logic, it is limited to specific program points where it can coordinate the activities of parallel threads (for example in

"**#pragma omp single**" construct). Figure 7 shows the high-level parallel flow diagram of this version.

```
//Compute the Gaussian Elimination for matrix a[n x n], Version 2
bool ComputeGaussianElimination(float **a,int n)
{float pivot,gmax,pmax,temp;
int  pindmax,gindmax,i,j,k;
bool isOK = true;
//Find the pivot row among rows k, k+1,...n
// The following OMP parallel statement will create parallel processes.
#pragma omp parallel shared(a,gmax,gindmax) firstprivate(n,k)
private(pivot,i,j,temp,pmax,pindmax)
        {//Perform row-wise elimination. Each thread works on a number of rows to find the local
max value pmax. Then update this local max value to the global variable gmax
        for (k = 0 ; k < n-1 ; k++)
                {#pragma omp single
                        {gmax = 0.0;     }
                pmax = 0.0;
                #pragma omp for schedule(static)
                for (i = k ; i < n ; i++)
                        {temp = abs(a[i][k]);
                        if (temp > pmax)
                                {pmax = temp;
                                pindmax = i;     }}
//gmax is updated one by one
                #pragma omp critical
                        {if (gmax <= pmax)
                                {gmax = pmax;
                                gindmax = pindmax;      }}
                #pragma omp barrier  //All threads have to reach this point before continue
                if (gmax == 0.0)          //If matrix is singular set the flag & quit
                        {isOK = false;
                        break;}
                if (gindmax != k)          //Swap rows if necessary
                        {#pragma omp for schedule(static)
                        for (j = k; j < n; j++)
                                {temp = a[gindmax][j];
                                a[gindmax][j] = a[k][j];
                                a[k][j] = temp;}}
                pivot = -1.0/a[k][k];      //Compute the pivot
                #pragma omp for schedule(dynamic)  //Perform row reductions
                for (i = k+1 ; i < n; i++)
                        {temp = pivot*a[i][k];
                        for (j = k ; j < n ; j++)
                                {a[i][j] = a[i][j] + temp*a[k][j];}}}}
// At this point all processes join into one stream.
        return isOK;}
```

Figure 6. Version 2 of OpenMP C++ ComputeGaussianElimination

The outer for-loop (K) over diagonal elements in Figure 6 is a sequential loop executed independently by every parallel thread. The barriers embedded within this loop ensure that threads will complete iterations of the loop in close synchrony (i.e., they reach the barrier point before they can continue). For each diagonal element, a parallel search for the element with maximum absolute value in the pivot column below the diagonal is done within **#pragma omp for schedule(static)**. The small amount of work for each iteration makes static scheduling appropriate to avoid the synchronization overhead of dynamic scheduling. Similarly rows are swapped to bring the pivot into the diagonal position within another statically scheduled for loop. In the row reduction loop, each iteration involves a full row to the right of the diagonal, so overhead may be small enough to make the better load balance resulting from dynamic scheduling beneficial. This is not likely to help much here with processors of the same speed since the work per parallel iteration is the same for all threads.

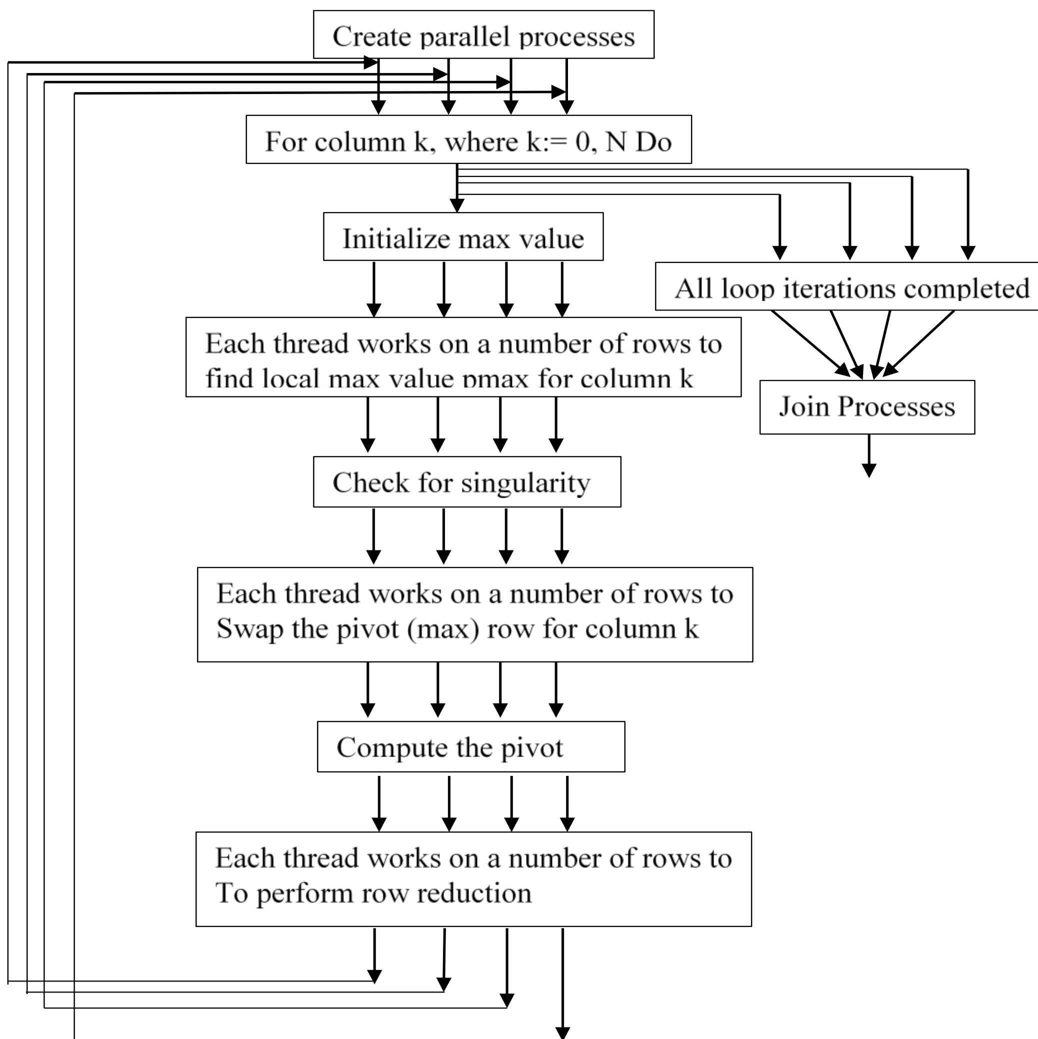

Figure 7. Parallelism corresponding to Version 2 (Figure 6)

Results and Conclusion

The two implementations result in drastically different performance. Figure 8 shows the result of running the two versions of our Gaussian elimination programs on a matrix of dimension 6000 on a 64-core AMD-Bulldozer multicore architecture. Note that the difference in execution times are only due to the number of times processes are created and joined, otherwise the two programs (even parallel techniques employed) are identical.

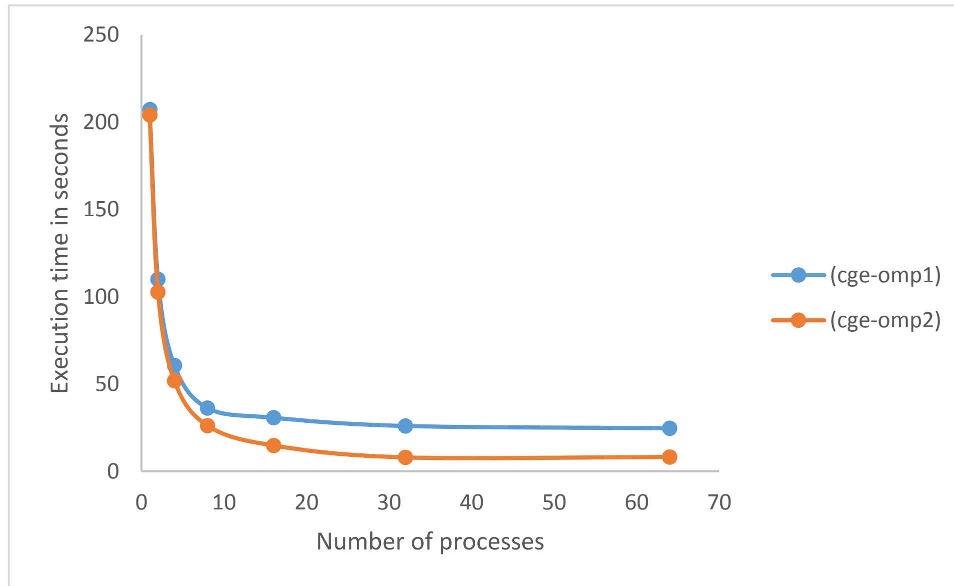

Figure 8. Performance comparison of the two implementations of Gaussian

This method of applying parallelism within the code as observed or perceived applicable, not only won't exploit all the parallelism possible in a given sequential code, it also prevents the design of a parallel solution technique from the start. Starting the design from a sequential perspective leads to designing good sequential programs which often is not a suitable base for parallel processing. In so many cases, the best parallel solution will perform poorly on a sequential machine. It is only when it is executed in parallel on a parallel computer with enough number of processors that we observe the best performance. Learning about the trade-offs between parallelism and memory usage, inherently sequential access data structures versus data structures that allow for parallel access and operations, and allowing more operations to be performed in a parallel version compared to the sequential version solving the same problem can be done most effectively when students observe these factors in a hands-on laboratory environment and exercises that re-enforce lectures.

Reference

[1] Gita Alaghband, Harry F. Jordan, "Overview of the force scientific parallel language", *Journal of Scientific Programming*, Volume 3 Issue 1, Spring 1994.

[2] OpenMP Applications Program Interface is available from http://www.openmp.org.

[3] Harry F. Jordan, Gita Alaghband, "*Fundamentals of Parallel Processing*", Prentice Hall, 2003