Performance analysis between explicit scheduling and implicit scheduling of parallel array-based domain decomposition using OPENMP

Article *in* Journal of Engineering Science and Technology · October 2014

CITATION
1

READS
3,422

4 authors, including:

Mohammed Faiz Aboalmaaly
Al-Zahraa University for Women
19 PUBLICATIONS **46** CITATIONS
SEE PROFILE

Ali Abdulrazzaq
University of Mosul
15 PUBLICATIONS **15** CITATIONS
SEE PROFILE

Sureswaran Ramadass
Universiti Sains Malaysia
171 PUBLICATIONS **1,208** CITATIONS
SEE PROFILE

Journal of Engineering Science and Technology
Vol. 9, No. 5 (2014) 522 - 532

PERFORMANCE ANALYSIS BETWEEN EXPLICIT SCHEDULING AND IMPLICIT SCHEDULING OF PARALLEL ARRAY-BASED DOMAIN DECOMPOSITION USING OPENMP

MOHAMMED FAIZ ABOALMAALY*, ALI ABDULRAZZAQ KHUDHER, HALA A. ALBAROODI, SURESWARAN RAMADASS

National Advanced IPv6 Centre (NAV6), Universiti Sains Malaysia,
11800 USM, Penang, Malaysia
*Corresponding Author: essa@nav6.usm.my

Abstract

With the continuous revolution of multicore architecture, several parallel programming platforms have been introduced in order to pave the way for fast and efficient development of parallel algorithms. Back into its categories, parallel computing can be done through two forms: Data-Level Parallelism (DLP) or Task-Level Parallelism (TLP). The former can be done by the distribution of data among the available processing elements while the latter is based on executing independent tasks concurrently. Most of the parallel programming platforms have built-in techniques to distribute the data among processors, these techniques are technically known as automatic distribution (scheduling). However, due to their wide range of purposes, variation of data types, amount of distributed data, possibility of extra computational overhead and other hardware-dependent factors, manual distribution could achieve better outcomes in terms of performance when compared to the automatic distribution. In this paper, this assumption is investigated by conducting a comparison between automatic and our newly proposed manual distribution of data among threads in parallel. Empirical results of matrix addition and matrix multiplication show a considerable performance gain when manual distribution is applied against automatic distribution.

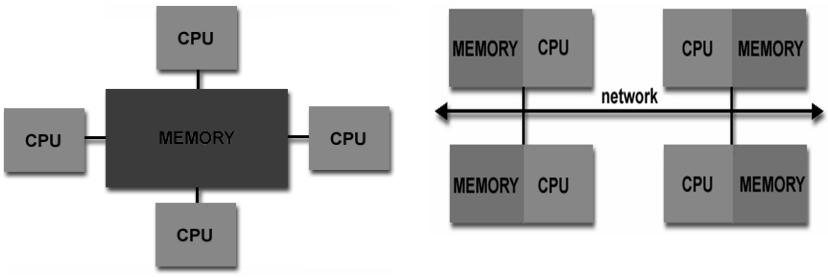
Keywords: Parallel computing, Array-based applications, Speedup, Profiling.

1. Introduction

In terms of general-purpose processors, Multicore architecture is the most, if not the only, equipped hardware architecture in the digital devices against the single

core architecture. However, graphical processors are gaining more popularity in commodity hardware these days as well. Currently, Server machines, personal computers, laptops, cameras and handheld devices, are all equipped with more than one processing element. This horizontal scaling, in terms of number of processors has already paved the way for parallel computing, more particularly parallel computing on shared memory architectures.

Back into its categorization, parallelism can be achieved in two different ways based on the hardware architecture used. These two categories are: shared memory approach and distributed memory approach [1]. The former is when processors share the same address space (memory) such as multicore in a personal computer, while the latter is when each processor in a parallel environment has its own dedicated address space and the communications between the processors in this category are done across the network, as an example, a cluster of several computing nodes. Figure 1 illustrates the memory wise architectural difference between (a) shared and (b) distributed memories respectively [2].



(a) Shared **(b) Distributed**

Fig. 1. Memory Architectures.

Several programming paradigms have been innovated to facilitate/motivate the adoption of parallel computing utilisation. OpenMP and MPI are two examples of well-known Application Programming Interfaces (APIs) to parallelize both shared memory and distributed memory applications respectively [3].

To date, most of the known programming languages such as C++, Java and Microsoft C# are created to serve a sequential execution of code, while parallelization capability has been added as an add-on feature into these languages. Generally, efficient parallelization needs to be done with a programmer's touch and in most of cases is not a straightforward compiler step. Moreover, Regardless of their forms and orientations, each parallel programming paradigm provides ways to facilitate its utilization. By emphasizing on the DLP approach, automatic distribution of data among processors can be done with few lines of code. However, this facilitation might also raise several questions about its performance efficiency.

OpenMP is one of the most common language extensions or APIs used for parallel computing for shared memory architectures. It is categorized as a cross-platform, scalable and easy to use. OpenMP is a collection of compiler directives, library functions, and environment variables that can be used to specify shared-

memory parallelism in FORTRAN, C and C++ programs [4]. OpenMP uses the fork-join model of parallel execution. Although this fork-join model is useful for solving a variety of problems, it is somewhat tailored for large array-based applications [5].

As a matter of fact, in order to achieve better performance by using parallelization, it is recommended to parallelize, when possible, the most computational intensive part of a given serial code. "FOR" loops can be relatively considered to require more computations than the single program's statement [6]. OpenMP provides a single line of code to parallelize "FOR" loops by breaking up the loop iterations across available cores in a particular architecture. The syntax of the early mentioned pragma in C++ language is illustrated in Fig. 2.

```
#pragma omp parallel  for
for loop
{
Structured Block
}
```

Fig. 2. Parallel for Loop Using OpenMP.

However, several variation of *#pragma omp parallel* can be made with additional OpenMP keywords. Back to the pseudo code in the example above, which is called implicit distribution or implicit scheduling, where the compiler handles the distribution of the loop iterations by itself. In this paper, a comparison will be conducted between the above approach with respect to its efficiency and the newly proposed approach where the work distribution is done by the programmer with miner modification to the "FOR" loop which to the best of our knowledge, this proposed approach is new and it has not been suggested by other studies before.

The rest of the paper is organized as follows, in Section 2, a background to the domain decomposition methods is provided. Section 3, is the core part of this paper, where the newly proposed method (explicit scheduling) is explained. In Section 4, the results are presented in section 4 along with a brief discussion. Section 5, is the section where the conclusion and recommendations are highlighted.

2. Domain Decomposition

Domain decomposition is an essential scheme in several algorithms. In terms of parallelism, domain decomposition is used to distribute the data among processing elements for faster processing. The main idea of domain decomposition comes from the concept of graph partitioning [7].

Generally, there are two types of domain decomposition methods that were introduced in the academic research, namely overlapping domain decomposition and non-overlapping domain decomposition. As the name implies, the former is when the subdomains are overlapped among each other while the non-overlapping is when the subdomains do not share a common space. Figures 3 and 4 illustrate the core difference between these two methods.

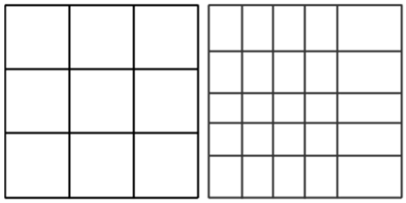


Fig. 3. Non-Overlapping Domain Decomposition.

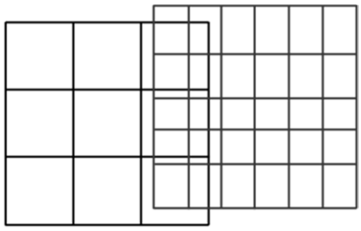


Fig. 4. Overlapping Domain Decomposition.

In addition to early domain decomposition categorization, there is another categorization for the domain decomposition, which is based on the dimensions of the domain itself. In this categorization, a domain decomposition method can be a 1D, 2D, …., ND domain decomposition. The following two figures; Figs. 5 and 6, show the difference between 1D and 2D domain decomposition respectively. In this study, we will focus on parallelizing a two dimensional domain using the 1D domain decomposition method.

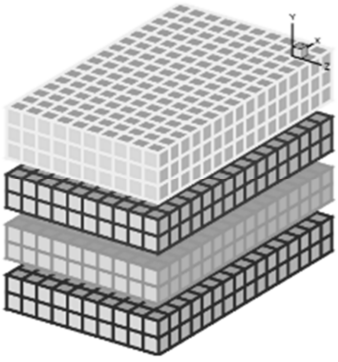


Fig. 5. 1D Domain Decomposition.

Fig. 6. 2D Domain Decomposition.

3. Explicit (Manual) Domain Decomposition using OpenMP

Manual distribution is when the data is distributed with regards to its type, hardware architecture, the procedure of distribution and most importantly the purpose of this distribution. In this paper, an explicit 1D domain decomposition is used and it is then compared to its implicit counterpart of OpenMP. However, in order to have a better understanding, the following subsections will illustrate this scenario.

Before explaining our proposed approach, let us first consider the programming problem that will be adopted in this study. Since OpenMP is used mainly for large array-based applications, it is more appropriate to examine the proposed method with an example of array-based applications such as matrix addition or matrix multiplication. The typical matrix addition used in our experiment is illustrated in Fig. 7, where A and B are two matrices with similar dimensions, m is the total number of rows and n is the total number of columns, C is used to store the result of the addition.

$$\mathbf{A}+\mathbf{B}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}+\begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{bmatrix}$$

$$\mathsf{C}=\begin{bmatrix} a_{11}+b_{11} & a_{12}+b_{12} & \cdots & a_{1n}+b_{1n} \\ a_{21}+b_{21} & a_{22}+b_{22} & \cdots & a_{2n}+b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}+b_{m1} & a_{m2}+b_{m2} & \cdots & a_{mn}+b_{mn} \end{bmatrix}$$

Fig. 7. Matrix Addition.

Following Fig. 7, the OpenMP Style used to add these two matrices in parallel is as shown in Fig. 8, where the compiler directive *#pragma omp parallel for* is used to parallelize the "FOR" loop, while the *#pragma omp barrier* is used to make sure that all parallel threads finish their work before moving to the next section of code in order to avoid the program's logical errors. As might be noticed, the per-thread work allocation is not explicitly specified and it is done based on the compiler and following the OpenMP specifications.

```
#pragma omp parallel for private (j)
        for (int i = 0; i<row; i++)
                for (int j = 0; j<col; j++)
                 {
                    c[i][j] = a[i][j]+b[i][j];
                 }
#pragma omp barrier
```

Fig. 8. Parallel Matrix Addition Using in Typical OpenMP-Style.

As aforementioned, OpenMP is a collection of compiler directives, library functions, and environment variables. One of the library functions is used to retrieve the thread ID, (*omp_get_thread_num(*)). Thread ID is a thread-dependent integer value starts from 0 to N-1, where N is typically the number of processors in a particular hardware architecture. As an example, in Fig. 9, there are 4 threads running in parallel to execute a particular programming problem, the thread IDs for these threads are: 0, 1, 2 and 3.

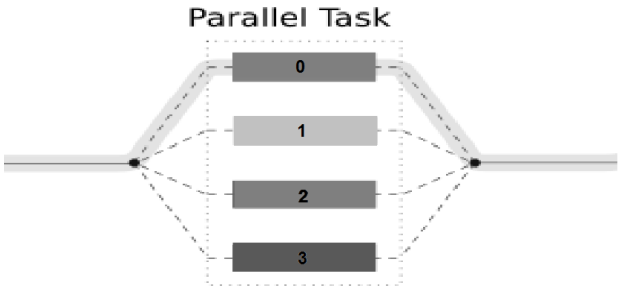


Fig. 9. Threads in Parallel.

Since this integer value is thread-dependent, it can be used mathematically inside the "FOR" loop in order to explicitly schedule the loop among threads in parallel. Let us consider the example illustrated in Fig. 7: three matrices A, B and C, each of them with m rows and *n* columns, where:

$$C[m][n] = A[m][n] + B[m][n] \quad (1)$$

The typical OpenMP style is illustrated in Fig. 8. However, our approach is shown in Fig. 10.

```
#pragma omp parallel
  {
     int threadID = omp_get_thread_num();
     for (int i = 0; i<row; i++)
       {
   for (int j = threadID*(col/numberofthread) ;
j< (threadID+1)*(col/numberofthread); j++)
          {
             c[i][j] = a[i][j]+b[i][j];
          }
       }
   }
#pragma omp barrier
```

Fig. 10. Matrix Addition Using Explicit Loop Scheduling.

As previously mentioned, the OpenMP library function *omp_get_thread_num()* will retrieve the thread's ID. Since all threads will call this function, the integer value of the variable threadID is unique for each thread. Hence, this variable will be used to split up the iterations of the "FOR" loop among threads.

If it was to be considered that *numberofthread* is 4 and the *threadID* of each thread starts from 0 to 3 (0,1,2 and 3), row = col = 1000, then following Fig. 10 above, the explicit distribution of data by the inner "FOR" loop will be as follows:

Thread 0:

J_{min} = 0*1000/4, J_{max} = (0+1)*1000/4 = 0 to 1*250 = (from 0 to 250),

Thread 1:

J_{min} = 1*1000/4, J_{max} = (1+1)*1000/4 = 250 to 2*250 = (from 250 to 500)

Thread 2:

J_{min} = 2*1000/4, J_{max} = (2+1)*1000/4 = 500 to 3*250 = (from 500 to 750)

Thread 3:

J_{min} = 3*1000/4, J_{max} = (3+1)*1000/4 = 750 to 4*250 = (from 750 to (1000)

where J_{min} indicates the beginning (initial value) of the loop iteration and J_{max} indicates the end value of the loop iteration. Additionally, as the end value of the inner "FOR" loop shown in Fig. 11 ends with less than only (<) and not less than or equals to (≤), hence the last iteration of each value is not included within the same loop. As an example, in Thread 0 below, the actual range is from 0 to 249.

Moreover, there is no need to use the compiler directive *#pragma omp parallel for* for the "FOR" loop, because if it was used, the loop will be doubled distributed; one from the *#pragma omp parallel for* and another one from the explicit distribution, and by doing this, it will lead to incorrect results. Hence, the *#pragma omp parallel* will only be used since the "FOR" loop is explicitly distributed among threads. Finally, the barrier pragma is used for the same purpose as illustrated in the OpenMP-style in Fig. 8.

Similar to the matrix addition problem illustrated earlier, and following the same idea in our proposed method, this paper has also covered the matrix multiplication problem.

In terms of complexity, matrix multiplication problem is relatively associated with higher complexity compared to the matrix addition problem as it involves more mathematical operations for each index in the matrix as illustrated in the Fig. 11.

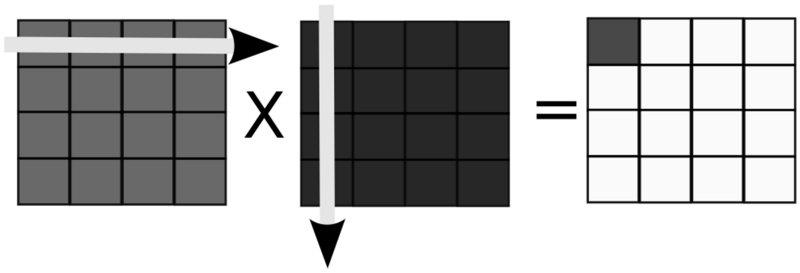


Fig. 11. Matrix Multiplication.

From a programming perspective, matrix multiplication can be done with three "FOR" loops. Serial matrix multiplication is shown in Fig. 12. While the OpenMP-style that multiplies two matrices in parallel is shown in Fig. 13.

```
for ( int i = 0 ; i < row ; i++ )
   {
     for ( int j = 0 ; j < col ; j++ )
        {
            for ( int k = 0 ; k < col ; k++ )
               {
                  sum = sum + a[i][j]*b[j][k];
               }

                  c[i][j] = sum;
                  sum = 0;
        }
   }
```

Fig. 12. Serial Matrix Multiplication.

```
#pragma omp parallel for private (j,k) reduction (+:sum)
            for ( int i = 0 ; i < row ; i++ )
               {
                 for ( int j = 0 ; j < col ; j++ )
                    {
                        for ( int k = 0 ; k < col ; k++ )
                            {
                                sum = sum + am[i][j]*bm[j][k];
                            }

                             cm[i][j] = sum;
                             sum = 0;
                     }
               }
#pragma omp barrier
```

Fig. 13. Parallel Matrix Multiplication Using Typical OpenMP-Style.

Our solution for the matrix multiplication problem is more or less the same as the one proposed in the matrix addition problem. However, the former problem needs more effort in order to ensure correct multiplication results. In Fig. 14, the manual distribution of the matrix multiplication is illustrated.

```
#pragma omp parallel private (j,k) reduction(+:sum)
   {
      int threadID = omp_get_thread_num( );
    for ( int i = threadID*(row/numberofthread) ;
    i< (threadID+1)*(row/numberofthread); i++ )
       {
            for ( int j = 0 ; j < col ; j++)
              {
                  for ( int k = 0; k < col ; k++)
                      {
                         sum += a[i][j]*b[j][k];
                      }
                       {
                          c[i][j] = sum;
                           sum = 0;
                       }

                }
       }
    }
```

Fig. 14. Matrix Multiplication Using Explicit Loop Scheduling.

As can be seen in Fig. 14, similar usage of the variable threadID is used to distribute the arrays for multiplication among processors. However, new programming statements are also introduced such as the *reduction ( )* clause. This clause performs a reduction on the variables that appear in its list. In other words,

a private copy for each list variable is created for each thread. At the end of the reduction, the reduction variable is applied to all private copies of the shared variable, and the final result is written to the global shared variable which in the matrix multiplication problem is the sum variable used to store the result of multiplication of each index.

4. Results and Discussion

Several testing scenarios are used in this study in order to show the better performance of the proposed method compared to the typical method of OpenMP. The variation of the scenarios was in terms of the size of the arrays A, B and C. the sizes used in this paper were 1000×1000, 2000×2000, 4000×4000, 6000×6000, 8000×8000 10000×10000 and 12000×12000. However, the matrices' sizes mentioned earlier are arbitrary and it can be any other values as long as the size is large enough to reflect the speedup gain and as long as the system memory allows allocating the desired size. The data type used was integer. Dynamic allocation was used to allocate memories for the matrices A, B and C since static allocation does not allow allocating more memory compared to the dynamic approach. The hardware and software architecture used in this experiment was as follows:

CPU: Intel(R) Corei5 M480 @ 2.67GHz with hyper threading (HT)

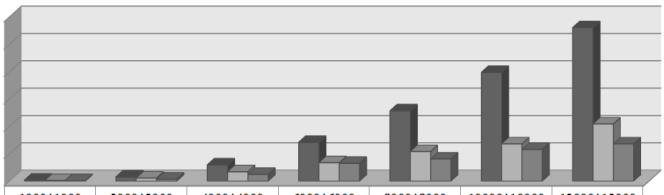
RAM : 8 GB

IDE: Microsoft Visual Studio 2010 Professional Edition

OpenMP version: 2.0 formally (OpenMP 2.0)

Moreover, *QueryPerformanceCounter( )* function [8] is used to accurately measure the time in micro-seconds (ms) which is used as a performance metric for the sake of comparing the results. In order to ensure a more accurate comparison, the sequential version was also considered in the experiment to form three cases: sequential, parallel OpenMP-style and the proposed explicit parallel style. Additionally, each empirical test was repeated three times and an arithmetic mean is taken for these three attempts to reduce the influence of any hardware or software factors that would affect the results.

Figure 15 shows the bar chart for the sequential, parallel OpenMP-style, and, the proposed parallel explicit method to parallelize the 2D matrix addition of different matrix sizes.

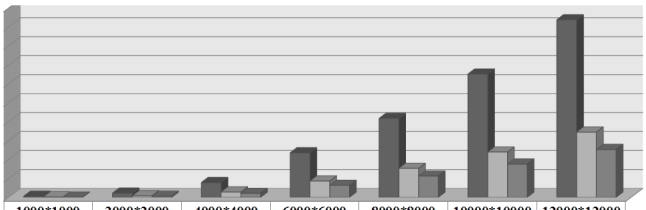


	1000*1000	2000*2000	4000*4000	6000*6000	8000*8000	10000*10000	12000*12000
Serial	82.3857	308.906	1192.47	2839.07	5144.47	7952.89	11227.6
Parallel OpenMP-Style	49.1988	229.378	690.766	1352.08	2165.45	2724.26	4181.53
Our Approach	44.2318	148.246	500.324	1311.63	1621.82	2322.93	2726.61

Fig. 15. Results of Matrix Addition in Micro Seconds (ms).

It is clear that the proposed approach has outperformed the parallel OpenMP-style as well as the serial style over all array sizes. Surprisingly, the higher performance gain, which is up to 34.8%, was at the higher matrix size used in this experiment (12000×12000) which is a good empirical proof for the scalability of the proposed method.

In terms of matrix multiplication, the same scenario has been applied. Figure 16 shows the results obtained from running the matrix multiplication problem.



	1000*1000	2000*2000	4000*4000	6000*6000	8000*8000	10000*10000	12000*12000
Serial	11207.1	44828.4	156899.4	470698.2	833135.81	1301358.14	1873955.72
Parallel OpenMP-Style	4120.04	16480.16	57680.56	173041.68	306283.77	478415.25	688917.96
Our Approach	3114.48	12457.92	42356.92	127070.78	224915.28	351317.67	505897.45

Fig. 16. Results of Matrix Multiplication in Micro Seconds (ms).

As illustrated in Fig. 16, there was a noticeable performance gain when the manual data distribution was applied versus the OpenMP-style (automatic data distribution). Among all matrix sizes, the performance gains between the OpenMP-style and our approach were about the same (27%). Relatively, the performance gain of the matrix multiplication problem does not reach the higher performance gain obtained from the matrix addition problem, this was due to the execution of the *reduction ( )* clause as it causes extra overhead.

5. Conclusions

In this paper, the influence of manual domain decomposition verses automatic domain decomposition based on OpenMP standard has been studied. With several matrix sizes and using two different array-based problems (matrix addition and matrix multiplication), there was a considerable performance gain compared to the automatic version defined by the OpenMP standard. However, although the proposed method has proved its efficiency on a matrix addition and matrix multiplication only, the simplicity of the idea can be easily applied to any other array-based applications. Moreover, the proposed idea would be a good introduction to the area of multimedia compression where the main contents are images and videos in which these two contents are basically a representation of two-dimensional arrays.

Despite the early mentioned performance gain, there is one limitation associated in our approach in which solving it has been pushed to a future research. The limitation is represented by the size of the input matrix; in other words, the dimensions of the input matrix should be divisible to the number of threads in the development environments in order to ensure correct results. To

avoid this limitation, different sizes of sub-domains should be assigned to each processing element.

The proposed approach has proved a good scalability in terms of data sizes; further research would be focused on examining the scalability with respect to the number of cores 4, 6, 8 or more.

References

1. Tinetti, F.G.; and Wolfmann, G. (2009). Parallelization analysis on clusters of multicore nodes using shared and distributed memory parallel computing models. *Proceedings of* 2009 *WRI World Congress on Computer Science and Information Engineering*, 2, 466-470.
2. Barney, B. (2009). Introduction to parallel computing. Retrieved January 14, 2013, from https://computing.llnl.gov/tutorials/parallel_comp/.
3. Diaz, J.; Munoz-Caro, C.; and Nino, A. (2012). A survey of parallel programming models and tools in the multi and many-core era. *IEEE Transactions on Parallel and Distributed Systems*, 23(8), 1369-1386.
4. OpenMP Architecture Review Board (2012). OpenMP application program interface. Retrieved January 14, 2013, from http://www.openmp.org/mp-documents/OpenMP4.0RC1_final.pdf.
5. Jin, H.; Jespersen, D.; Mehrotra, P.; Biswas, R.; Huang, L.; and Chapman, B. (2011). High performance computing using MPI and OpenMP on multi-core parallel systems. *Parallel Computing*, 37(9), 562-575.
6. Li, J.; Shu, J.; Chen, Y.; Wang, D.; and Zheng, W. (2005). Analysis of factors affecting execution performance of OpenMP programs. *Tsinghua Science and Technology*, 10(3), 304-308.
7. Eguzkitza, B.; Houzeaux, G.; Aubry, R.; Owen, H.; and Vázquez, M. (2012). A parallel coupling strategy for the Chimera and domain decomposition methods in computational mechanics. *Computers & Fluids*, 80, 128-141.
8. Microsoft. QueryPerformanceCounter function (Windows). Retrieved January 23, 2013, from http://msdn.microsoft.com/en-us/library/windows/desktop/ms644904(v=vs.85).aspx.