Architecture Aware Programming on Multi-Core Systems

M.R. Pimple
Department of Computer Science & Engg.
Visvesvaraya National Institute of Technology,
Nagpur – 440010 (India)

S.R. Sathe
Department of Computer Science & Engg.
Visvesvaraya National Institute of Technology,
Nagpur – 440010 (India)

***Abstract*— In order to improve the processor performance, the response of the industry has been to increase the number of cores on the die. One salient feature of multi-core architectures is that they have a varying degree of sharing of caches at different levels. With the advent of multi-core architectures, we are facing the problem that is new to parallel computing, namely, the management of hierarchical caches. Data locality features need to be considered in order to reduce the variance in the performance for different data sizes. In this paper, we propose a programming approach for the algorithms running on shared memory multi-core systems by using blocking, which is a well-known optimization technique coupled with parallel programming paradigm, OpenMP. We have chosen the sizes of various problems based on the architectural parameters of the system like cache level, cache size, cache line size. We studied the cache optimization scheme on commonly used linear algebra applications – matrix multiplication (MM), Gauss-Elimination (GE) and LU Decomposition (LUD) algorithm.**

Keywords- multi-core architecture; parallel programming; cache miss; blocking; OpenMP; linear algebra.

I. INTRODUCTION

While microprocessor technology has delivered significant improvements in clock speed over the past decade, it has also exposed a variety of other performance bottlenecks. To alleviate these bottlenecks, microprocessor designers have explored alternate routes to cost effective performance gains. This has led to use of multiple cores on a die. The design of contemporary multi-core architecture has progressively diversified from more conventional architectures. An important feature of these new architectures is the integration of large number of simple cores with software managed cache hierarchy with local storage. Offering these new architectures as general-purpose computation platforms creates number of problems, the most obvious one being programmability. Cache based architectures have been studied thoroughly for years leading to development of well-known programming methodologies for these systems, allowing a programmer to easily optimize code for them. However, multi-core architectures are relatively new and such general directions for application development do not exist yet.

Multi-core processors have several levels of memory hierarchy. An important factor for software developers is how to achieve the best performance when the data is spread across local and global storage. Emergence of cache based multi-core systems has created a "cache aware" programming consensus. Algorithms and applications implicitly assume the existence of a cache. The typical example is linear algebra algorithms. To achieve good performance, it is essential that algorithms be designed to maximize data locality so as to best exploit the hierarchical cache structures. The algorithms must be transformed to exploit the fact that a cache miss will move a whole cache-line from main memory. It is also necessary to design algorithms that minimize I/O traffic to slower memories and maximize data locality. As the memory hierarchy gets deeper, it is critical to efficiently manage the data. A significant challenge in programming these architectures is to exploit the parallelism available in the architecture and manage the fast memories to maximize the performance. In order to avoid the high cost of accessing off-chip memory, algorithms and scheduling policies must be designed to make good use of the shared cache[12]. To improve data access performance, one of the well-known optimization technique is tiling[3][10]. If this technique is used along with parallel programming paradigm like OpenMP, considerable performance improvement is achieved. However, there is no direct support for cache aware programming using OpenMP for shared memory environment. Hence, it is suggested to couple OpenMP with tiling technique for required performance gain.

The rest of the paper is organized as follows. Section II describes the computing problem which we have considered. The work done in the related area is described in section III. Implementation of the problems is discussed in section IV. Experimental setup and results are shown in section V. The performance analysis is carried out in section VI.

II. COMPUTING PROBLEM

As multi-core systems are becoming popular and easily available choice, for not only high performance computing world but also as desktop machines, the developers are forced to tailor the algorithms to take the advantage of this new platform. As the gap between CPU and memory performance continues to grow, so does the importance of effective utilization of the memory hierarchy. This is especially evident in compute intensive algorithms that use very large data sets, such as most linear algebra problems. In the context of high performance computing world, linear algebra algorithms have to be reformulated or new algorithms have to be developed in order to take advantage of the new architectural features of

these new processors. Matrix factorization plays an important role in a large number of applications. In its most general form, matrix factorization involves expressing a given matrix as a product of two or more matrices with certain properties. A large number of matrix factorization techniques have been proposed and researched in the matrix computation literature to meet the requirements and needs arising from different application domains. Some of the factorization techniques are categorized into separate classes depending on whether the original matrix is dense or sparse. The most commonly used matrix factorization techniques are LU, Cholesky, QR and singular value decomposition (SVD).

The problem of dense matrix multiplication (MM) is a classical benchmark for demonstrating the effectiveness of techniques that aim at improving memory utilization. One approach towards the cache effective algorithm is to restructure the matrices into sequence of tiles. The copying operation is then carried out during multiplication. Also, for a system AX=B, there are several different methods to obtain a solution. If a unique solution is known to exist, and the coefficient matrix is full, a direct method such as Gaussian Elimination(GE) is usually selected.

LU decomposition (LUD) algorithm is used as the primary means to characterize the performance of high-end parallel systems and determine its rank in the Top 500 list[11]. LU Factorization or LU decomposition is perhaps the most primitive and the most popular matrix factorization techniques finding applications in direct solvers of linear systems such as Gaussian Elimination. LU factorization involves expressing a given matrix as product of a lower triangular matrix and an upper triangular matrix. Once the factorization is accomplished, simple forward and backward substitution methods can be applied to solve a linear system. LU factorization also turns out to be extremely useful when computing the inverse or determinant of a matrix because computing the inverse or the determinant of a lower or an upper triangular matrix is relatively easy.

III. Related Work

Since multi-core architectures are now becoming mainstream, to effectively tap the potential of these multiple units is the major challenge. Performance and power characteristics of scientific algorithms on multi-core architectures have been thoroughly tested by many researchers[7]. Basic linear algebra operations on matrices and vectors serve as building blocks in many algorithms and software packages. Loop tiling is an effective optimization technique to boost the memory performance of a program. The tile size selection using cache organization and data layout, mainly for single core systems is discussed by Stephanie Coleman and Kathryn S. Mckinley [10].

LU decomposition algorithm decomposes the matrix that describes a linear system into a product of a lower and an upper triangular matrix. Due to its importance into scientific computing, it is well studied algorithm and many variations to it have been proposed, both for uni and multi-processor systems. LU algorithm is implemented using recursive methods [5], pipelining and hyperplane solutions [6]. It is also implemented using blocking algorithms on Cyclops 64 architecture [8]. Dimitrios S. Nikolopoulos, in his paper [4] implemented dynamic blocking algorithm. Multi-core architectures with alternative memory subsystems are evolving and it is becoming essential to find out programming and compiling methods that are effective on these platforms. The issues like diversity of these platforms, local and shared storage, movement of data between local and global storage, how to effectively program these architectures; are discussed in length by Ioannis E. Venetis and Guang R. Gao [8]. The algorithm is implemented using block recursive matrix scheme by Alexander Heinecke and Michael Bader [1]. Jay Hoeflinger, Prasad Allavilli, Thomas Jackson and Bob Kuhn have studied scalability issues using OpenMP for CFD applications[9]. OpenMP issues in the development of parallel BLAS and LAPACK libraries have also been studied[2]. However, the issues, challenges related with programming and effective exploitation of shared memory multi-core systems with respect to cache parameters have not been considered.

Multi-core systems have hierarchical cache structure. Depending upon the architecture, there can be two or three layers, with private and shared caches. When implementing the algorithm, on shared memory systems, cache parameters must be considered. The tile size selection for any particular thread running on a core is function of size of L_1 cache, which is private to that core as well as of L_2 cache which is a shared cache. If cache parameters like, cache level, cache size, cache line size are considered, then substantial performance improvement can be obtained. In this paper, we present the parallelization of MM, GE and LUD algorithm on shared memory systems using OpenMP.

IV. Implementation

In this paper we have implemented parallelization of most widely used linear algebra algorithms, matrix multiplication, gauss elimination and LU decomposition, on multi-core systems. Parallelization of algorithms can also be implemented using message passing interface (MPI). Pure MPI model assumes that, message passing is the correct paradigm to use for all levels of parallelism available in the application and that the application “topology” can be mapped efficiently to the hardware topology. However, this may not be true in all cases. For matrix multiplication problem, the data can be decomposed into domains and these domains can be independently passed to and processed by various cores. While, in case of LU decomposition or GE problem, task dependency prevents to distribute the work load independently to all other processors. Since the distributed processors do not share a common memory subsystem, the computing to communication ratio for this problem is very low. Communication between the processors on the same node goes through the MPI software layers, which adds to overhead. Hence, pure MPI implementation approach is useful when domain decomposition can be used; such that, the total data space can be separated into fixed regions of data or domains, to be worked on separately by each processor.

For GE and LUD problems, we used the approach of 1D partitioning of the matrix among the cores and then used OpenMP paradigm for distributing the work among number of

threads to be executed on various cores. The approach of 2 D partitioning of data among cores is more suitable for array processors. For a shared memory platform, all the cores on a single die share the same memory subsystem, and there is no direct support for binding the threads to the core using OpeMP. So, we restricted our experiments with 1D partitioning technique and applied parallelization for achieving speedup using OpenMP.

A. Architecture Aware Parallelization

To cope up with memory latency, all data required during any phase of the algorithm are made available in the cache. The data sets so chosen, are accommodated into the cache. Considering the cache hierarchy, the tile size selection depends upon cache size, cache line size to eliminate self interference misses. Now depending upon the architecture of the underlying machine, the computation work is split into number of cores available. One dimensional partitioning of data is done, so that, every core receives specific number of rows (or columns), such that, the data fits in the shared cache. The blocking technique is then used which ensures that the maximum block size is equal to the size of private cache belonging to the core. Parallel computation is carried out using OpenMP pragmas by individual cores.

B. Determining Block Size

In order to exploit cache affinity, the block size is chosen such that, the data can be accommodated into the cache. The experiments were carried out on square matrix of size N. Let 's' be the size of each element of matrix and 'C_s' be the size of shared cache. Let the block size be $B \times B$.

1. For blocked matrix multiplication, C = A x B, block of matrix A & B, and one row of matrix C should be accommodated into the cache. Then the required block size can be calculated using :

$$2sB^2 + sB = C_s$$

For large cache size, we get,

$$B = \sqrt{C_s/2s} \quad (1)$$

2. For GE problem, the size of input matrix is $[N]x[N+1]$. The required block size can be calculated with the following equation:

$$B \times (B+1) \times s = C_s$$

So, the optimal block size,

$$B = \sqrt{C_s/s} \quad (2)$$

3. For LU decomposition algorithm with same matrix used for storing lower and upper triangular matrix, the optimal block size comes out to be

$$B = \sqrt{C_s} \quad (3)$$

C. Effect of Cache Line Size

Let cache line size be C_{ls}. Without the loss of generality, we assume that the first element of input array falls in the first position of cache. The number of rows that completely fit in the cache can be calculated as :

$$Rows = C_s/C_{ls} \quad (4)$$

For every memory access, the entire cache line is fetched. So block size $B < C_{ls}$ will lead to self interference misses. Also if $B > C_{ls}$, system will fetch additional cache lines, which may in turn lead to capacity misses; as less number of rows can be accommodated in the cache. So to take the advantage of spatial locality, the block sizes chosen were integral multiple of cache line size C_{ls}. We assumed that, every row in the selected tile is aligned on a cache line boundary. After finding the row size, block size can be calculated.

Block Size B = k x C_{ls}

if $(B \bmod C_{ls} = 0)$ (*k is integer*)

Or $B = \left\lfloor \frac{Rows}{C_{ls}} \right\rfloor \times C_{ls}$

Maximum Speed up is achieved when -

$(B \propto C_{ls})$ or

when B is multiple of number of Rows

The algorithm for block size selection is presented in Fig. 1.

Further improvement in the performance is achieved by using the technique of register caching for the array elements, that are outside the purview of the "*for*" loop (like value a[i][j] shown in Fig. 3). This value is cached, which is then shared by all the threads executing the "*for*" loop. The OpenMP implementation of matrix multiplication and GE problem is given in Fig. 2 and Fig. 3 respectively.

D. LU Decomposition

The main concept is to partition the matrix into smaller blocks with a fixed size. The diagonal entry in each block is processed by master thread on a single core. Then for calculating the entries in the upper triangular matrix, each row is partitioned into number of groups equal to number of cores; so that each group is processed by each core. Similarly, for calculating the entries in the lower triangular matrix, each column is partitioned into number of groups equal to number of cores; so that each group is processed by each core. The implementation divides the matrix into fixed sized blocks, that fit into the L1 data cache of the core creating first level of memory locality. On the shared memory architecture, the whole matrix is assumed to be in the globally accessible memory address space. The algorithm starts by processing the diagonal block on one processor, while all other processors wait for the barrier. When this block finishes, the blocks on the same row are processed by various cores in parallel. Then the blocks on same column are processed by various cores in parallel. In turn, each processor waits for the barrier again for the next diagonal block.

The storage space can further be reduced by storing lower and upper triangular matrices in a single matrix. The diagonal elements of lower triangular matrix are made 1, hence, they need not be stored. But this method suffers from the problem of load imbalance, if number of elements processed in each row or column by each core is not divisible by number of cores available. Also, the active portion of the matrix is reduced after each iteration and hence, load allocation after each iteration is not constant.

1) LUD computation:

Let A be an $n \times n$ matrix with rows and columns numbered from 0 to (n-1). The factorization consists of n major steps. Each step consisting of an iteration of the outer loop starting at line 3 of Fig. 5. In step k, first the partial column $A[k+1:n,\ k]$ us divided by $A[k,k]$. Then the outer product $A[k+1:n,\ k]$ x $A[k,\ k+1:n]$ is subtracted from the $(n-k)$ x $(n-k)$ sub matrix $A[k+1:n,\ k+1:n]$. For each iteration of the outer loop $k = 0\ to\ (n-1)$, the next nested loop in the algorithm goes from $k+1\ to\ (n-1)$.

```
Procedure BS(CS, CLS,N,B)
Input: CS: Cache Size
CLS: Cache Line Size
s: Size of each element in input
N: Input Matrix Rows
Output : B : Block Size(square)
Total cache lines = CS / CLS
No of rows (NR) from input problem size that can be
accommodated in cache
NR = ( )
The optimal block size B
If (NR > CLS)
       B= k xCLS   // Where k is integer constant
Else (B = CLS)
```

Figure 1. Block Size Selection

```
void mat-mult() // matrix multiplication //
{ for (i=0; i<N; i=i+B)
 { Read block of a & c;
   Read block ofbB;
   omp_set_num_threads(Omp_get_num_proc());
   #pragma omp parallel for shared(a,b,c,i)
         private(r,i1,j) schedule (static)
   for (r=i; r<(min(i+B, N)); r++)
    for (i1=i; i1<(min(i+B,N)); i1++)
      { for(j=0; j<N; j++)
                  c[r][i1] += a[r][j] * b[j][i1];
      }
   Write block of c ;
 }}
```

Figure 2. Parallel Matrix Multiplication

A typical computation of LU factorization procedure in the kth iteration of the outer loop is shown in Fig. 4. The kth iteration of the outer loop does not involve any computation on rows 1 to (k-1) or columns 1 to (k-1). Thus at this stage, only the lower right (n-k) x (n-k) sub matrix of A is computationally active. So the active part of the matrix shrinks towards the bottom right corner of the matrix as the computation proceeds.

The amount of computation increases from top left to bottom right of the matrix. Thus the amount of work done differs for different elements of matrix. The work done by the processes assigned to the beginning rows and columns would be far less than those assigned to the later rows and columns. Hence, static scheme of block partitioning can potentially lead to load imbalance. Secondly, the process working on a block may idle even when there are unfinished tasks associated with that block.

```
void forwardSubstitution() // GaussElimination loop//
// Matrix size (n x n)
{ int i, j, k, max, kk, p, q; float t;
   for (i = 0; i < n; ++i)
  { max = i;
    for (j = i + 1; j < n; ++j)
    if (a[j][i] > a[max][i]) max = j;
    for (j = 0; j < n + 1; ++j)
    {  t = a[max][j]; a[max][j] = a[i][j];
       a[i][j] = t;
    }
   for (j = n; j >= i; --j)
  {  for (k=i+1; k<n; k=k+B)
     { x=a[i][i];  // Register Caching //
      #pragma omp parallel for shared(i,j,k) private(kk)
schedule (static)
      for (kk=k; kk<(min(k+B, n)); ++kk)
           a[kk][j] -= a[kk][i]/x * a[i][j];
     }
   }
}
```

Figure 3. OpenMP parallelization of GE loop

This idling can occur if the constraints imposed by the task-dependency graph do not allow the remaining tasks on this process to proceed until one or more tasks mapped onto other processes are completed.

2) LUD OpenMP parallelization:

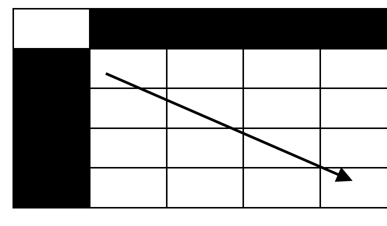
For parallelization of LU decomposition problem on shared memory, we used tiling technique with OpenMP paradigm. The block size B is selected such that, the matrix size is accommodated in a shared cache. The actual data block used by each core is less than the size of private cache so that locality of memory access for each thread is maintained.

For LUD algorithm, due to the task dependency at each iteration level, the computation cannot be started simultaneously on every core. So, algorithm starts on one core. Diagonal element is executed by master core. After the synchronization barrier, the computation part of non-diagonal elements is split over the available cores.

After computing a row and column of result matrix, again the barrier is applied to synchronize the operations for the next loop. The size of data computed by each core is determined by block size.

The size of data dealt by each core after each iteration is not the same. With static scheduling, the chunk is divided exactly into the available multiple threads and every thread works on the same amount of data.

Fig. 5 illustrates the OpenMP parallelization. The size of input matrix ‘a’ is ‘N’.



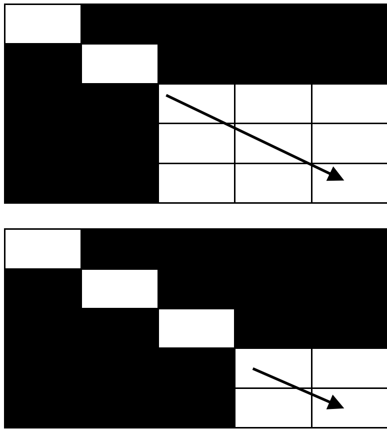


Figure 4.Processing of blocks of LU Decomposition

```
1.    Lu-Fact (a)
2.    {
3.      for (k=0; k<N; k++)
4.      {   #pragma omp single
5.          for(j=k+1; j<(N); j++)
6.              a[j][k]=a[j][k]/a[k][k];
7.          #pragma omp parallel for shared (a,k)
      private(i) schedule static
8.          for(j=k+1; j<(N); j=j+B)
9.          for (jj=j; jj<min(jj+B, N), jj++)
10.         {   v=a[k][jj];        --- caching the value
11.             #pragma omp parallel for shared
      (a,k,jj) private(i) schedule static
12.             for(i=k+1; i<(N); i++)
13.                 a[i][jj]= a[i][jj]- (a[i][k]*v);
14.         }
15.     }
16.   }
```

Figure 5. OpenMP implementation of LU Decomposition algorithm

V. Experimental Setup & Results

We conducted the experiments to test cache aware parallelization of MM, GE, LUD algorithms on Intel Dual core, 12 core and 16 core machines. Each processor had hyper threading technology such that, each processor can execute simultaneously instructions from two threads, which appear to the operating system as two different processors and can run a multi program workload. The configuration of the systems is given in Table 1.

Each processor had 32 KB data cache as L_1 cache. Intel Xeon processors (12 & 16 cores) had an eight way set associative 256 KB L_2 cache and 12 MB L_3 cache dynamically shared between the threads. The systems run Linux 2.6.x Blocked LU decomposition was parallelized at two levels using OpenMP.

We used relatively large data sets, so that the performance of the codes becomes more bound to the L_2 and L_3 cache miss latencies. The programs were compiled with C compiler (gcc 4.3.2). Fig. 6 and Fig. 7 show the speed up achieved when the block sizes are such that, the data fits in L_2 cache for matrix multiplication and Gauss elimination algorithm respectively. Fig. 8 and Fig. 9 show the results of LU decomposition algorithm for various matrix sizes on dual and 16 core system respectively

Table 1. System configuration

Processors	Intel(R) Core™2 Duo CPU E7500	Intel(R) Dual Core CPU E5300	Intel(R) Xeon(R) CPU X5650 (12 cores)	Intel(R) Xeon(R) CPU E5630 (16 cores)
Core frequency	2.93 GHz	2.60 GHz	2.67 GHz	2.53 GHz
L1 Cache size	32 KB I cache, 32 KB D cache	32 KB I cache, 32 KB D cache	32 KB I cache, 32 KB D cache	32 KB I cache, 32 KB D cache
L2 Cache size	3072 KB, shared	2048 KB, Shared	256 KB	256 KB
L3 Cache size	---	---	12 MB	12 MB

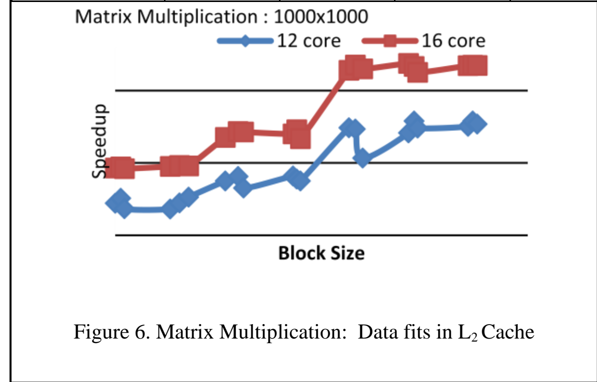


Figure 6. Matrix Multiplication: Data fits in L_2 Cache

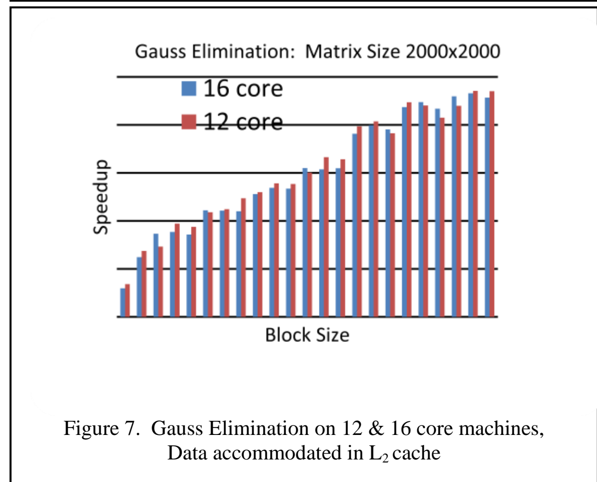


Figure 7. Gauss Elimination on 12 & 16 core machines, Data accommodated in L_2 cache

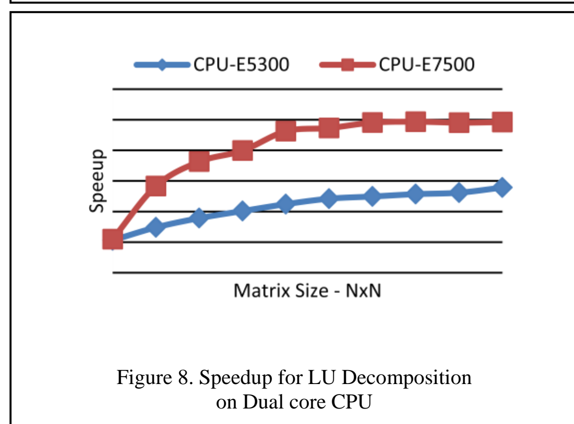


Figure 8. Speedup for LU Decomposition on Dual core CPU

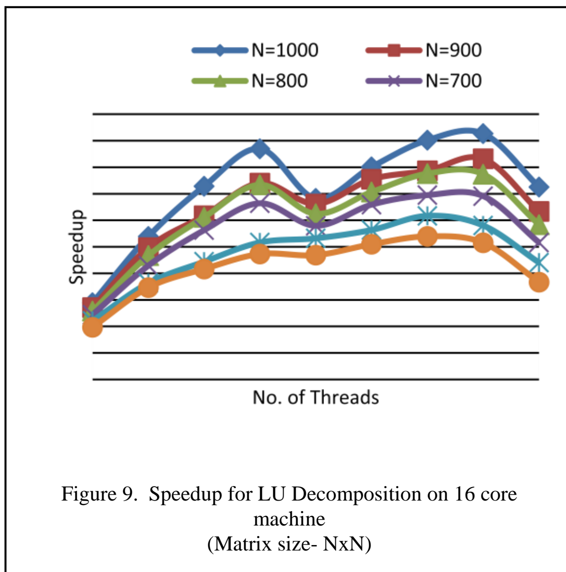


Figure 9. Speedup for LU Decomposition on 16 core machine
(Matrix size- NxN)

VI. Performance Analysis

The strategy of parallelization is based on two observations. One is that the ratio computation to communication should be very high for implementation on shared memory multi-core systems. And second is that the memory hierarchy is an important parameter to be taken into account for the algorithm design which affects load and store times of the data. Considering this, we implemented the algorithms matrix multiplication and Gauss elimination with a blocking scheme that divides the matrices into relatively small square tiles. The optimal block size is selected for each core, such that the tile is accommodated in the private cache of each core and thus avoids the conflict misses. This approach of distributing the data chunks to each core greatly improves the performance. Fig. 6 and Fig. 7 shows the performance improvement when block size is multiple of cache line size. Whenever block size is greater or less than the cache line size, performance suffers. This is due to reloading overheads of entire new cache line for the next data chunk. With this strategy, we got the speedup of 2.1 on 12 core machine and speed up of 2.4 on 16 core machine. The sub linear speedups in Fig. 6 and 7 for lower block sizes are attributed to blocking overheads.

For Gauss elimination and LU decomposition problem, the OpenMP pragma, splits the data among the available cores. The size of data dealt by every core, after every iteration is different. This leads to load imbalance problem. The chunk scheduling scheme, demands the chunk calculations at every iteration and hence affects performance. However, static scheduling ensures equal load to every thread and hence reduces the load imbalance. For LU decomposition problem with 1D partitioning of data among the cores, we observed a speedup of 1.39, & 2.46 for two dual core machines and speedup of 3.63 on 16 core machine. The maximum speedup is observed when the number of threads is equal to the number of (hardware) threads supported by the architecture. Fig. 9 shows the speed up when 16 threads are running on a 16 core machine. Speed up is directly proportional to the number of threads. The performance degrades when more software threads are in execution than the threads supported by architecture. So, for 18 threads, scheduling overhead increases and performance is degraded. However, when number of threads is more than 8, performance degrades due to communication overheads. This is because, 16 core Intel Xeon machine comprises of 2 quad cores connected via QPI link. Fig. 9 shows performance enhancement up to eight threads and degradation in the performance when number of threads is ten. When the computation is split across all the available sixteen threads, speed up is again observed, where communication overhead is amortized over all cores. Further enhancement in the performance is achieved when method of register caching is used for loop independent variables in the program. Many tiling implementations do not consider this optimal block size considerations with cache attributes. However, our implementation considers the hierarchy of caches, cache parameters and arrives at optimal block size. The block size calculations are governed by the architecture of the individual machine and the algorithm under consideration. Once the machine parameters and input problem size is available, the tailoring of the algorithm accordingly improves the performance to a greater extent. Of course, there is a significant amount of overhead in the OpenMP barriers at the end of loops; which means that load imbalance and not data locality is the problem.

VII. Conclusion & Future Work

We evaluated performance effects of exploiting architectural parameters of the underlying platform for programming on shared memory multi-core systems. We studied the effect of private cache L_1, shared cache L_2, cache line size on execution of compute intensive algorithms. The effect of exploiting L1 cache affinity does not affect the performance much, but the effects of exploiting L2 cache affinity is considerable, due its sharing among multiple threads and high reloading cost for larger volumes. If these factors are considered and coupled with parallel programming paradigm like OpenMP, performance enhancement is achieved. We conclude that, affinity awareness in compute intensive algorithms on multi-core systems is absolutely essential and will improve the performance significantly. We plan to extend the optimization techniques for the performance enhancement on multi-core systems by considering the blocking technique at register level and instruction level. We also plan to investigate and present generic guide lines for compute intensive algorithms on various multi-core architectures.

References

[1] Alexander Heinecke and Michael Bader, "Towards many-core implementation of LU decomposition using Peano Curves," *UCHPC-MAW'09*, May 18-20, 2009, Ischia, Itali.

[2] C. Addison, Y. Ren, and M. van Waveren, "OpenMP issues arising in the development of Parallel BLAS and LAPACK libraries," In *Scientific Programming Journal*, Volume 11, November 2003, pages: 95-104,IOS Press.

[3] Chung-Hsing Hsu, and Ulrich Kremer, "A quantitative analysis of tile size selection algorithms," *The Journal of Supercomputing*, 27, 279-294, 2004.

[4] Dimitrios S. Nikolopoulos, "Dynamic tiling for effective use of shared caches on multi-threaded processors," *International Journal of High*

Performance Computing and Networking, 2004, Vol. 2, No. 1, pp. 22-35.

[5] F.G. Gustavson, "Recursion leads to automatic variable blocking for dense linear algebra algorithms," *IBM Journal of Research and Development*, 41(6):737-753, Nov.,1997.

[6] H. Jin, M. Frumkin, and J. Yan, "The OpenMP implementation of NAS parallel benchmarks and its performance," Technical report nas-99-011, NASA Ames Research Centre, 1999.

[7] Ingyu Lee, "Analyzing performance and power of multi-core architecture using multithreaded iterative solver," *Journal of Computer Science*, 6 (4): 406-412, 2010.

[8] Ioannis E. Venetis, and Guang R. Gao, "Mapping the LU decomposition on a many-core architecture: Chanllenges and solutions," *CF'09*, May 18-20, 2009, Ischia, Itali.

[9] Jay Hoeflinger, Prasad Alavilli, Thomas Jackson, and Bob Kuhn, "Producing scalable performance with OpenMP: Experiments with two CFD applications," *International Journal of Parallel computing*. 27(2001), 391-413.

[10] Stephanie Coleman, and Kathryn S. McKinley, "Tile size selection using cache organization and data layout," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design & Implementation*, California, United States, 1995, pages: 279-290

[11] The Top 500 List. http://www.top500.org

[12] Vahid Kazempour, Alexandra Fedorova, and Pouya Alagheband, "Performance implications of cache affinity on multicore processors," *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*, Spain, 2008, pages:151-161

AUTHORS PROFILE

S.R. Sathe received M.Tech. in Computer Science from IIT, Bombay (India) and received Ph.D. from Nagpur University(India). He is currently working as a Professor in the Department of Computer Science & Engineering at Visvesvaraya National Institute of Technology, Nagpur(India). His research interests include paralle and distributed systems, mobile computing and algorithms.

M.R. Pimple is working as a Programmer in the Department of Computer Science & Engineering at Visvesvaraya National Institute of Technology, Nagpur(India) and persuing her M.Tech. in Computer Science. Her area of interests are computer architecture, parallel processing.