High Performance Computing Achieved in Personal Computers

Muhammad Saeed*[1], Syed Asim Ali*[2] , Maryam Feroze*[3] and Dr Nasir Touheed#[4]

*** Department of Computer Science/UBIT, University of Karachi, Pakistan**

 # Faculty of Computer Science, Institute of Business Administration, Karachi , Pakistan

Abstract

The purpose of this paper is to provide an up to date survey of the technologies that enables high performance computing on general purpose personal computers. Multiprocessor or multicore computers are widely available these days along with graphical processing units installed for gaming and other high speed common computing. Researchers can exploit data, instruction or function level parallelism in their research tasks and can write high speed multithreaded programs for Multiprocessor or Multicores in OpenMP, or can develop parallel programs similar of super computer applications for GPU in CUDA-C or Open-CV. This paper covers a brief over view of each environment along with programming examples.

Keywords: *ILP - Instruction Level Parallelism, SMP - Symmetric Multi-Processing, GPU - Graphical Processing Units.*

1. Introduction

Researchers who don't have state of the art super computers or parallel or distributed computing environments to work can still done research and development of recent personal computers available all over the world. All research tasks are usually done through special applications written in some programming language and every program can be divided into Data and Instructions for processing. On the basis of instructions and data Flynn's categorized four types of computer architectures known as Flynn's Taxonomy [1]. Fig-1 shows the Flynn's computer classifications. Each type of architectures contains processing and storage devices and different improvements can be introduced at each level to gain processing speed. Normal computers comes under two categories one is uniprocessor SISD (Single Instruction Stream Single Data Stream) and the other comes under Shared Memory Multiprocessor MIMD (Multiple Instruction Stream Multiple Data Stream). Graphical Processing Units are available these days for general purpose computing in personal computer and are similar as super computers of SIMD (Single Instruction Stream Multiple Data Stream) vector and array processor.

High-performance computing can be achieved in all three types of general purpose computing devices in different ways. In uniprocessor environment programs can be developed in such a way that can exploit the existence of cache, pipeline, multiple functional units and multi-issue processors. In multiprocessing environments Programs can be divided into multiple independent threads and can be scheduled on all available cores or processors. Highly parallel programs can be written in CUDA-C or in OpenCL to exploit GPU for research purpose.

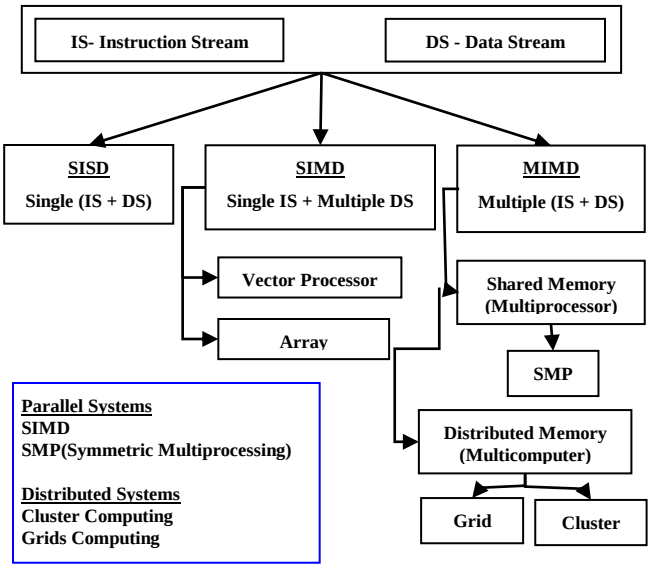

Fig 1. Flynn's Taxonomy

The table 1 provides the list of tools and techniques available on different types of computers. Each technique will be explained in their relevant subsections.

Table 1: High Performance Computing Tools for Personal Computers

Hardware or Operating System Feature	Language or Library
Multithreading on Single Processor	Any Language
Multiprocessing	OpenMP , P-Threads
GPU	CUDA-C , OpenCL

In the subsequent sections each type of environments is explained. Section 2 will cover the techniques that can be used to gain high speed in uniprocessor environment. Section 3 will cover multiprocessing environment and tools that can be used for speedups. Section 4 will cover tools and techniques to exploit GPU parallel computing power.

2. High Performance in Uniprocessor

In SISD uniprocessor environment programs can be developed in such a way that can exploit the existence of cache, pipeline, multiple functional units and multi-issue processors. Section 2.1 explains how the better code can be written to gain speedup with cache memory. Section 2.2 is about loop unrolling and code rescheduling for better pipeline performance.

2.1 Speedup gain using careful coding for Cache

Memory Cache is used to speedup data transfer rates between memory and processor. Most of the research applications uses data stored in one or two dimensional arrays and most of the time especially in the computer vision or graphics applications same operations are applied on all data [2][3][4]. Usually arrays are stored in row or column major in different languages and have very high impact while accessing through loops. Here in the following sections we have briefly described cache memory, row and column major array arrangements in memory and then coding habits to exploits cache for best speedup gain.

2.1.1 Cache Memory

CPU cache is a fast speed memory installed within or outside of processor to speed up the memory access. During reading from main memory if the required data is found within the cache it is called **"Cache Hit"** and data can be provided by cache with reduced access time as compare to the main memory. If data is not found in cache then it is called **"Cache Miss"** and then data will be provided by the main memory with slow speed. When new data come from main memory it may get new space if available otherwise some older cached data is replaced depending on **"Replacement Algorithm"**. While writing back to the memory there is a choice to updating only cache and mark the entry as **"Dirty Cache"** or writing the memory contents on the same time with slow speed. The unit or amount of data transfers between cache and processors is always in few words (1, 2, 4 or 8 bytes) on the other hand between cache and memory unit of transfer is in blocks of multiple words.

2.1.2 Array organization in Memory

Primary memory is basically a linear array of storage elements. Any data stored in memory can be accessed by providing memory address. Most of the applications hold data in arrays (collection or set of similar data elements) either in one or two dimensional forms. One dimensional array is used to represent vector of similar data elements and can be accessed randomly with one to one address mapping through single array index. On the other hand two dimensional arrays are used to represent matrices and any element can be accessed randomly through two indexes.

Two dimensional arrays are most important in most of the scientific computing applications [3]. Row Major and Column Major are two ways to store two dimensional arrays in main memory. The impact of programming style on execution speed is discussed in following sections for both row and column major orders. In the Table 2 two dimensional array is given that can be stored only linearly in memory.

Table 2: Two Dimensional Array = Data [row , column]

[0,0]=0	[0,1]=1	[0,2]=2	[0,3]=3
[1,0]=4	[1,1]=5	[1,2]=6	[1,3]=7
[2,0]=8	[2,1]=9	[2,2]=10	[2,3]=11
[3,0]=12	[3,1]=13	[3,2]=14	[3,3]=15

2.1.2.1 Row Major

In row major memory order multidimensional arrays are stored row by row in linear memory. And in each row elements are stored column wise from lower index to higher index. C/C++, Python and Mathematica are famous languages and tools for researchers that stored multidimensional data in row major order. The row major order of matrix given in Table 2 is given in Table 3.a.

Table 3: Data[Row][Column] , a. Row Major, b. Column Major

a. Row Major		b. Column Major	
Data [0,0]→	0	Data [0,0]→	0
Data [0,1]→	1	Data [1,0]→	4
Data [0,2]→	2	Data [2,0]→	8
Data [0,3]→	3	Data [3,0]→	12
Data [1,0]→	4	Data [0,1]→	1

Data [1,1]→	5	Data [1,1]→	5
Data [1,2]→	6	Data [2,1]→	9
Data [1,3]→	7	Data [3,1]→	13
Data [2,0]→	8	Data [0,2]→	2
Data [2,1]→	9	Data [1,2]→	6
Data [2,2]→	10	Data [2,2]→	10
Data [2,3]→	11	Data [3,2]→	14
Data [3,0]→	12	Data [0,3]→	3
Data [3,1]→	13	Data [1,3]→	7
Data [3,2]→	14	Data [2,3]→	11
Data [3,3]→	15	Data [3,3]→	15

2.1.2.2 Column Major

In column major memory order multidimensional arrays are stored column by column in linear memory. And in each column elements are stored row wise from lower index to higher index. FORTRAN, OpenGL and MATLAB are famous languages and tools for researchers that stored multidimensional data in column major order.

The column major order of matrix given in Table 2 is given in Table 3.b.

2.1.3 Code Optimization for Column Major Arrays.

Program use to access two dimensional arrays stored in row major memory mapping in least possible time is given below in Table 4. This is the style that programmers are used to process multidimensional arrays.

Table 4: Accessing Row Major Arrays

```
for(int row=0; row <Maximum_Rows; row++)
 for(int column=0; row<Maximum_Columns; column ++)
   reading from ← Data[row][column] ← writing to
```

If the cache block size is enough to store an entire row then cache miss only occurs when the entire row is processed and the total miss count will be equals to the total number of row in array.

The major problem that researchers faced while access data stored in multidimensional arrays is that programmers are used to access arrays in row major style and most of them used row index in outer loop and column index in inner loop. If the row major program is used to access column major data then each element access will cause a cache miss. And cache will give no speedup in that case. To get benefit of cache speedup programmers and researchers should understand how the compiler is arranging multidimensional arrays in memory. If writing programs in FORTRAN or MATLAB following code given in Table 5 will be more effective.

Table 5: Accessing Column Major Arrays

```
for(int column=0;row<Maximum_Columns;column ++)
 for(int row=0;row <Maximum_Rows;row++)
    reading from← Data[row][column] ←writing to
```

2.1.4 Cache blocking Algorithm and performance

Lots of research has been done to improve execution performs by getting advantage of cache memory [2][3][4].Loop blocking or cache blocking algorithms are used to reduce chances of cache miss while accessing data stored in multidimensional arrays.

To show the working of cache blocking, matrix multiplication example is considered in most of the research papers as it is the most frequently used mathematical operations in research and development.

To gain the performance by reducing cache miss the matrix must be accessed in blocks. The following code in Table 6 is much better [2][4] and gives improved access of data with low latency.

Table 6: Block Code for Matrix Multiplication

```
for (kk=0;kk< N /block_size; kk++)
 for (jj=0; jj<N/ block_size; jj++)
  for (row= 0 ; row < N; row++)
   for (column=kk; column < min(kk+block_size-1, N) , column++)
       {    r = A[row, column] /* register allocated */
          for (j= jj; j < min(jj+block_size-1, N); j++ )
            C [row,j] += r*B[column,j]
       }
```

The above code is proven and tested by many researchers [2][4] and they are agreed that blocking is very effective to reduce cache miss rates and memory access latency.

2.2 Speedup gain using careful coding for Pipelining

Since long time microprocessors are coming with instruction pipelines and only compilers optimize code for its better performance gain. Researchers and Programmers can also write better code to gain pipeline performance.

2.2.1 Instruction Pipeline

In MIPS architecture normal instruction executes in five phases known as Fetch (F), Decode (D), Execute (E),

Memory Access (M) and Write back (W) and circuit is designed in five separate modules. In a simple system new instruction always fetch after the completion of previous instruction means next instruction fetched after each 5 clock ticks. While instruction is in one phase the circuit of remaining phases remains unused and idle and makes 80% of the processor unused. To overcome this problem and for the 100% utilization of full five phases circuit pipeline is the best solution.

2.2.2 Instruction Level Parallelism

Pipeline allows simultaneous executions of multiple instructions each in different phase and this is referred as ILP (Instruction Level Parallelism) [18, 19].

One major requirement to use pipeline to achieve Instruction Level Parallelism is that instruction comes in pipeline must be independent of each other or result must be ready to read before next instruction requires it to process. One of the major challenges is the data dependency that cannot be handled in hardware. Compilers or operating systems are required to change instruction sequence to resolved dependency.

Problems	Solution
Control Hazard	Hardware Improvement Branch Prediction Code Rescheduling
Data Hazard	Forwarding Code Rescheduling

2.2.3 Loop Optimization to gain speedups

Most of the big data processing applications require looping to perform same operation on a given dataset. Each loop statement carries looping overhead in the form counter initialization, condition checking and counter increments that also causes control and data hazards. Loop unrolling and then code scheduling can be done to reduce loop overhead and gives 100% utilization of instruction pipelining.

Table 7: Simple Loop

```
for(int I = 0; I < 500 ; I++)
 V1[I] = V2[I]+V3[I]
```

In high level coding the above code try to execute V1[I] = V2[I] + V3[I] 500 times but loop also contains 3 additional statements I=0, I <500, and I++. In total 3 statements run total 500 times that will make 1500 and I=0 run once so in total 1501 instructions cycle are required to complete this loop. Loop unrolling will reduce 1501 count and rescheduling will help in reducing stalls. 2 times unrolled loop can be written as.

Table 8: Unrolled Optimized Code

```
for(int I = 0; I < 500 ; I=I+2)
 {
   V1[I] = V2[I]+V3[I]
   V1[I+1] = V2[I+1]+V3[I+1]
 }
```

The above code in Table 8 reduces loop overhead statements 50%. Condition checking and counter increment statements will required 250 instructions cycles now. The above technique can be used to reduce the program execution time specially when we process huge data and perform same operations on every element.

3. High Performance using GPU

GPU are just like SIMD Vector and Array processor where all processors can perform same operations on multiple data sets. GPUs are now available on general purpose computers for gaming applications. CUDA (Compute Unified Device Architecture) and OpenCL (Open Computing Language) are two famous programming platforms that can use GPU and executes instructions on it. In the following subsections GPU, CUDA Model and one code example is given to show how researchers can get performance and speedup gain by using GPU in personal computers.

3.1 Graphics Process Unit (GPU)

Graphics Process Unit (GPU) is a special display processing cards that are capable to process images and videos on real time. Normally GPUs are used in gaming consoles, robotics, smart phones, workstations and personal computers. Modern GPUs are dinged in such a way that it can efficiently manipulate computer graphics in real time. GPU contains highly parallel structure and two dimensional arrays of floating point ALU that's makes GPU more effective than general-purpose CPUs.

GPU can be configured both with uniprocessor and symmetric multiprocessing environment [5]. It has its own memory and for processing data must be transferred in GPU memory.

Each GPU (CUDA) consists of Streaming Multiprocessors (SM) and each SM consists of following units.

a. Stream Processor (SP): Single (32bits) and double (64bits) precisions floating point functional units.
b. SFU : Special Functional Units for frequently used approximation functions like log/exp, sin/cos,rcp/rsqrt
c. Wrap Scheduler: for instruction scheduling on execution units.
d. Constant Cache: use to buffer data to provide SM for execution.
e. Shared Memory: for data sharing among threads.
f. Other graphical units for texturing etc.

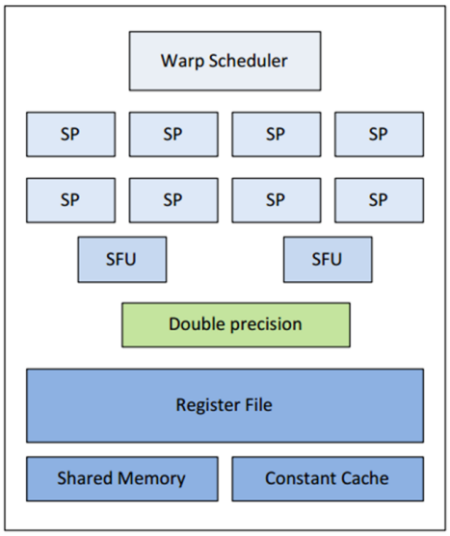

Fig 2. Basic architecture of Stream Multiprocessor

Fig 2. is a very basic SM (Stream Multiprocessor) and in advanced GPU there are hundreds of SMs connected in a grid form and shares GPU memory.

3.2 CUDA High Performance Programming Model

CUDA (Compute Unified Device Architecture) is a well-known application programing platform and framework that can use parallel architecture of GPUs to speedup application executing. CUDA model is shown in Fig. 3.

This framework allows creating multiple threads that can be executed in parallel on separate execution units. Threads can be single or multidimensional in CUDA and can be addressed through indexes. Group of consecutive threads that define minimum work unit is called “thread wrap” and multiple threads groups are called “Block”. The part of host program that is used to dispatch work to GPU is call kernel.

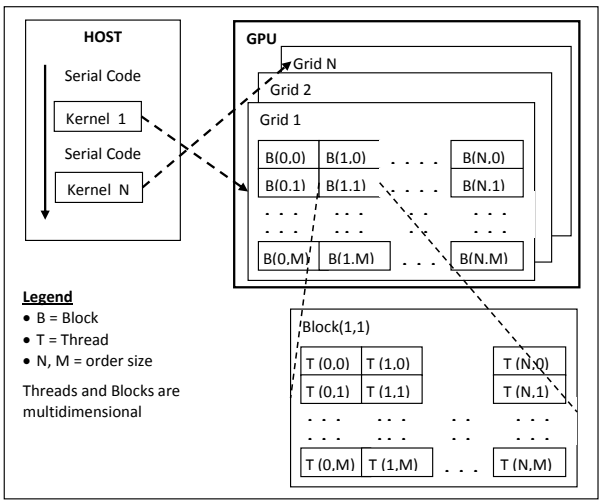

Fig 3. CUDA Programming Model

Lots of researches are now moved to use GPU for their research as CUDA is not only provides high speed parallel floating point operations but can also be used for general purpose computing and they are getting incredible speed. Though CUDA was introduced for highly data parallel programs in computer graphics but there are lots of APIs available in CUDA that provide synchronizations and atomic operations whenever sequential operations are required

3.3 Matrix Addition examples on CUDA

In CUDA programming Blocks can be arrange in 1 or 2 dimensional space and Threads can be 1,2 or 3 dimensional space. For working in vectors 1 dimensional allocation is best for mapping.

Matrix Addition (using 1 Block/SM)

```
__global__ void matrixAddition(float M1[N][N], float M2[N][N], float M3[N][N])
{
  int r = threadIdx.x;
  int c = threadIdx.y;
  M3[r][c] = M1[r][c] + M2[r][c];
}
int main()
{
 dim3 blocksPerGrid(1);          /* 1 block per grid (1D) */
 dim3 threadsPerBlock(N, N);  /* NxN threads per block (2D) */
 matrixAddition<<<blocksPerGrid, threadsPerBlock>>>(M1, M2, M3);
}
```

Matrix Addition (using Multiple Blocks/SM)

```
__global__ void matrixAddition(float M1[N][N], float M2[N][N], float M3[N][N])
{
  int r = blockIdx.x * blockDim.x + threadIdx.x;
  int c = blockIdx.y * blockDim.y + threadIdx.y;
  M3[r][c] = M1[r][c] + M2[r][c];
}
int main()
{
dim3 blocksPerGrid(N/16,N/16);    / / (N/16)x(N/16) blocks/grid (2D)
dim3 threadsPerBlock(16, 16);   // 16x16 threads/block (2D)
matrixAddition<<<blocksPerGrid, threadsPerBlock>>>(M1, M2, M3);
}
```

Fig 4. Matrix addition in CUDA-C

Matrix addition is the perfect example of two dimensional operations where all individual additions are independent

of each other. In the following code matrix additions is done first through 1 block and then through multiple blocks.

The complexity of CUDA programs went reduced since it tries to execute multiple instructions in same clock cycle in SIDM style. The complexity of vector and matrix addition is reduced to O(1) if the number of operations done is equal or less than the number of processing units available otherwise become linear.

4. High Performance using Multiprocessor

Shared memory multiprocessing systems are general purpose computers and can be used as desktop/personal computer, workstation or server computers. Researcher programmers can develop their multithreaded application and can schedule utilized all processors or cores available on the motherboard to speed up the execution of their task.

4.1 SMP Symmetric Multiprocessing

In shared memory multiprocessing systems multiple processors are installed on a single motherboard and uses a single shared memory. All processors must be of same architecture and can execute same instruction set that's why also known as SMP- Symmetric Multiprocessing systems. All processor can execute simultaneously and can run different or same instruction on different data elements in same instruction cycle.

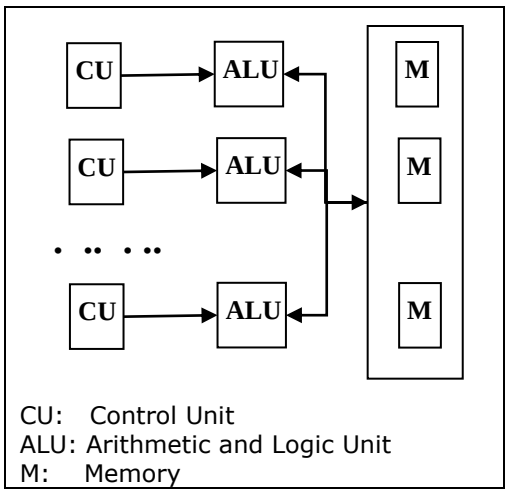

Fig 5. Flynn's MIMD Multiprocessor

Normally operating systems are multitasking and supports multithreading at kernel level. These operating systems can use one processor and can interleave processes or threads while scheduling and majority of the application developers can work on such systems. To support multiple processors on shared memory architecture operating systems must supports SMP (Symmetric Multiprocessing). These days almost all operating systems supports SMP and allows applications to be designed and built in such a ways that their threads can run parallel on different processor concurrently.

Process	Threads	Time Slot at (**Processor-0** and **Processor-1**)						
		1	2	3	4	5	6	7
A	8	A_{T1}		A_{T2}	A_{T3}	A_{T4}	A_{T6}	A_{T8}
						A_{T5}	A_{T7}	
B	4	B_{T1}	B_{T2}	B_{T3}	B_{T4}			
C	1		C_{T1}					

Fig 6. SMP : Overlap in Thread Execution

Fig 6 shows how multiple threads are overlap on different processor and interleave at the same processor. Light gray box shows that the thread is running on processor-0 and dark gray is used for processor-1.

4.2 Multiprocessing using OpenMP

OpenMP is an advanced multithreading technique that allows research developers to use all processors available on motherboard. The performance gained on these systems can be N times speedup when using OpenMP to parallelized sequential programs. It comes as a set of APIs that allow thread overlapping and parallel execution of threads of same process over different and multiple processors.

4.2.1 Vector and Matrix Addition using OpenMP

Vector and matrix addition are examples of coarse grained operations in which each operation is independent of each other. As now there are multiple processors available so the whole data can be divided into N equal parts where N represent number of processors and each processor can done job sequentially. Or we can create more threads and divide data equally and allow OpenMP to schedule threads in parallel. Table 9 contains the code for vector addition in OpenMP.

Table 9: Vector Addition using OpenMP

```
void v_add(double* V1, double* V2, double* V3)
{  #pragma omp parallel
   { #pragma omp for
     for(int i=0; i<Vector_Array_SIZE; i++)
       V3[i] = V1[i] + V2[i];
   }
}
```

In OpenMP "#pragma omp parallel" is used to create threads and any code enclosed in curly braces executed in parallel. The above code will run vector addition loop in

parallel and will also use enhanced parallel for loop to split the work automatically. "#pragma omp for" is used to invoke enhanced for loop. This code will give N times speedup for N processors.

Table 10: Matrix Addition using OpenMP

```
Void m_add(double M1[][],double M2[][] ,double *M3[][])
{
  #pragma omp parallel for private(c)
  for (r = 0; i < TotalRows;  r++)
   for (c = 0; j < TotalCols;  c++)
     M3[r][c] = M1[r][c] + M2[r][c];
}
```

The above code in table 10 will perform matrix addition just like the vector addition. Here the matrices are divided through their row number while column number is a private variable in each thread. This code will also give N times speedup for N processors.

5. Conclusions

We have provided a brief survey of different technologies that can provide high performance capabilities on personal computers and how we can gain speed-ups in programs execution while using the different hardware features of general purpose computers. Programming habits for accessing multidimensional arrays was discussed and a better algorithm is proposed to access arrays stored in column major order. Cache blocking, loop unrolling examples were also described. The structure of GPU and development framework CUDA was explained with examples. At the end OpenMP was described to show multiprocessing over multiprocessor and multicore architectures.

References

[1] Flynn, M. J. (September 1972). "Some Computer Organizations and Their Effectiveness". IEEE Trans. Compute. C–21 (9): 948–960.
[2] Lam, Monica D., Edward E. Rothberg, and Michael E. Wolf. "The cache performance and optimizations of blocked algorithms." ACM SIGOPS Operating Systems Review 25.Special Issue (1991): 63-74.
[3] Donald E. Knuth, The Art of Computer Programming Volume 1: Fundamental Algorithms, third edition, section 2.2.6 (Addison-Wesley: New York, 1997).
[4]Intel Developers Zone, "How to Use Loop Blocking to Optimize Memory Use on 32-Bit Intel® Architecture", Dec 10, 2008.
[5] Nicholas Wilt , "The CUDA Handbook , A Comprehensive Guide to GPU Programming", Addison-Wesley.
[6] Chao Wang et al, "Parallel algorithms for mining frequent structural motifs in scientific data", ICS '04 Proceedings of the 18th annual international conference on Supercomputing, ACM 2004.
[7] Jin-Soo Kim et al, "Memory Characterization of a Parallel Data Mining Workload", WWC '98 Proceedings of the Workload Characterization: Methodology and Case Studies, IEEE 1998.