Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 431–438

ISBN 978-83-60810-22-4

Cache-Aware Matrix Multiplication on Multicore Systems for IPM-based LP Solvers

Mujahed Eleyat
Miriam AS, Halden &
IDI, NTNU, Trondheim
Norway
Email: mujahed@miriam.as

Lasse Natvig, Jørn Amundsen
Department of Computer and Information Science (IDI)
Norwegian University of Science and Technology (NTNU)
Trondheim, Norway
Email: lasse@idi.ntnu.no, jorn.amundsen@ntnu.no

***Abstract*—We profile GLPK, an open source linear programming solver, and show empirically that the form of matrix multiplication used in interior point methods takes a significant portion of the total execution time when solving some of the Netlib and other LP data sets. Then, we discuss the drawbacks of the matrix multiplication algorithm used in GLPK in terms of cache utilization and use blocking to develop two cache-aware implementations. We apply OpenMP to develop parallel implementations with load balancing. The best implementation achieved a median speedup of 21.9 when executed on a 12-core AMD Opteron.**

Keywords: sparse matrix multiplication, cache optimization, interior point methods, multicore systems.

I. Introduction

DURING recent years, processor designers have moved away from uniprocessor systems to multicore systems. This shift is mainly due to manufactures inability to continue enhancing the performance of single-core processors [1]. Increasing clock speeds requires higher voltage and causes, consequently, too much heat to dissipate. On the other hand, using deeper pipelines and other advanced architectural techniques have yielded decreasing improvements. In addition, and due to the speed gap between main memory and the processing cores, there has been more demand for an efficient cache system to allow exploiting the collective processing power [2]. As a result, multicore programmers need not only to provide a parallel implementation of the application, but they also have to take cache utilization into consideration for efficient utilization of the multicore system. Techniques to reduce cache and TLB misses depend on the application memory access pattern, for example, tiling/blocking [3] is the most popular method for applications with poor exploitation of temporal locality.

A Linear Programming (LP) solver is one of many compute-intensive applications that could benefit from the high multicore performance. It works as a decision maker that chooses values of many variables to achieve a goal (maximum profit, best resource allocation, etc.) while satisfying a set of constraints that are specified as mathematical equalities and inequalities [5]. If we have m constraints and n variables, the LP-problem in standard form can be written as:

$$\text{minimize } z = c^T x, \text{subject to } Ax = b, x \geq 0,$$

where x is an n-dimensional column vector, c^T is an n-dimensional row vector, A is a $m \times n$ matrix, and b is an m-dimensional column vector.

Solving LP problems in an efficient way is crucial for industrial and scientific fields, especially since an application might need to solve large problems and/or a long sequence of problems. For example, Miriam Regina, a network gas flow simulator developed by Miriam AS [4], solves thousands of LP problems to make a single allocation of gas flow in the network. On the other hand, it needs to solve bigger LP instances for the simulation to cover large networks that span the national boundaries.

The motivation for investigating matrix multiplication in interior point methods (IPM) [5, 6] is that it takes a large fraction of the total computation time when solving some of the data sets. In addition, it is a special form of multiplication of the form ADA^T, where A is a sparse matrix, D is a diagonal matrix, and A^T is the transpose of A. Moreover, sparse multiplication is a form of irregular computation that is much more challenging to accelerate than dense multiplication. On the other side, the structure of the multiplication result is constant through all IPM iterations, a fact that may be used to enhance its computation performance.

In this paper, we profile serial GLPK [7], an open source LP solver, and present empirical results showing that matrix multiplication takes a relatively long time to compute for some Netlib and miscellaneous problems [8, 9]. We also analyse memory access patterns of sparse multiplication and develop cache-aware algorithms that reduce the rate of cache and TLB misses. Moreover, a parallel version is also provided while trying to exploit the cache hierarchy of the multicore system.

The paper is organized as follows: Section II gives a brief overview of the AMD Opteron compute node used. Then, compute-intensive parts of the LP solver are introduced in section III. Section IV explains GLPK implementation of sparse matrix multiplication while section V describes techniques to enhance cache utilization. We present related work in section VI and conclude with experimental results and future work.

II. MULTI-CORE HARDWARE

Introduced in 2009, the 64-bit Istanbul processor is the first 6-core AMD Opteron® processor and is available for 2-, 4-, and 8-socket systems, with clock speeds ranging from 2.0 to 2.8 GHz [10].

Fig. 1 shows a simplified block diagram. The processor has six cores, three levels of cache, a crossbar connecting the cores, the System Request Interface, the Memory controller, and three HyperTransport 3.0 links. The memory controller supports DDR2 memory with a bandwidth of up to 12.8 GB/s. In addition, the HyperTransport 3.0 links provide an aggregate bandwidth of 57.6 GB/s and are used to allow communication between different Istanbul processors. Each core has two levels of cache, a 512 KB L2 cache, 64 KB data cache and 64 KB instruction cache. However, all cores share a 6 MB L3 cache.

AMD Opteron multiprocessor systems are based on the cache coherent Non-Uniform Memory Access (ccNUMA) architecture. Each processor is connected directly to its own dedicated memory banks and it uses HT links to communicate with I/O busses and the other processor(s). Fig. 2 shows a block diagram of a 2-socket system.

III. GLPK AND IPM COMPUTATIONAL KERNELS

The GLPK (GNU Linear Programming Kit) package is a set of ANSI C routines contained into a callable library and intended for solving large-scale linear programming, mixed integer programming, and other related problems [7]. GLKP has routines for solving LP problems using either simplex or one of the primal-dual interior point methods (IPMs), namely the Mehrotra's predictor-corrector method [6]. This method, as well as other primal-dual interior point methods, keeps repeating a set of matrix operations, until it converges to an optimal solution. Every iteration of the algorithm includes the following computations [11]:

1) Sparse matrix-matrix multiplication of the form $S = PAD(PA)^T$, where P is a permutation matrix stored as a single dimensional array, A is the sparse working constraint matrix stored using the CRS format, D is a diagonal matrix stored using a single dimensional array, and $(PA)^T$ is the transpose of matrix PA. The output matrix S is a symmetric positive definite matrix.
2) Cholesky factorization of a symmetric sparse matrix S, the result of step 1, into LL^T where L is the lower factor matrix and L^T is the transpose of L.
3) Backward/forward solving using Cholesky factors.

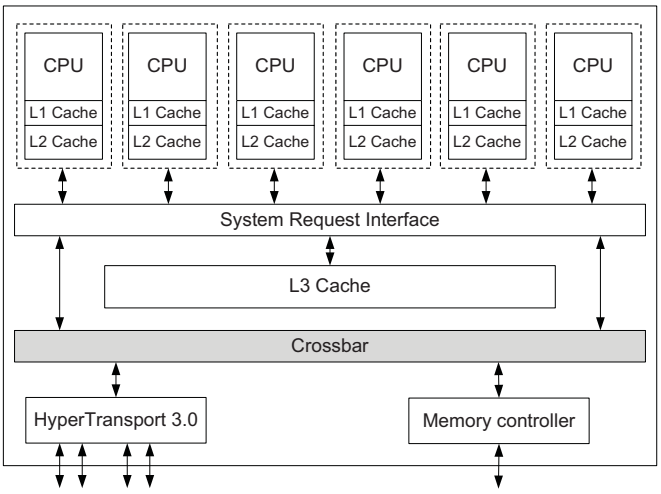


Fig. 1. Simplified block diagram of an AMD Opteron Istanbul processor.

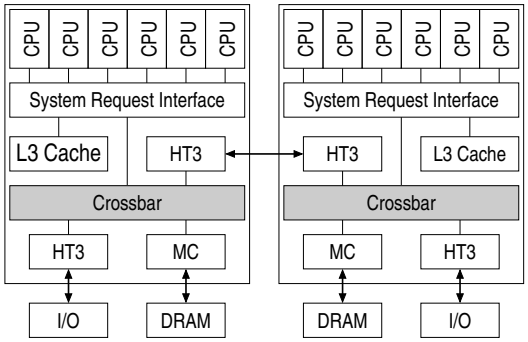


Fig. 2. Block diagram of a 2-socket system

A. Compressed row storage CRS

Since most practical problems are very sparse, GLPK uses compressed row storage (CRS) to store the constraint matrix and other matrices used in the IPM algorithm. CRS is a general storage format that makes no assumptions about the sparsity structure of the matrix [12]. As shown in Table I, CRS uses three contiguous arrays to store a sparse matrix A: A_{val} stores all nonzero elements row by row, A_{ind} holds the column indices of the nonzeros, and A_{ptr} holds the offset of each row into A_{val}. CRS is usually used to access the matrix row by row, while another format called compressed column storage (CCS) is used when a column by column access is needed. Similar to CRS, CSS uses three arrays to store the matrix, however, it stores the nonzeros column by column and A_{ptr} have pointers to the start of columns in A_{val}.

B. Time analysis of IPM computational kernels

Time analysis of serial IPM-based GLPK has been performed when solving big data sets taken from Netlib and the BPMPD website [9]. Size, number of nonzeros, and sparsity, fraction of nonzero elements of the matrix, of each data set

TABLE I
A MATRIX A AND ITS CRS STORAGE

$$A = \begin{bmatrix} 0 & 3 & 0 & 0 & 1 \\ 4 & 1 & 0 & 0 & 0 \\ 0 & 5 & 9 & 2 & 0 \\ 6 & 0 & 0 & 5 & 3 \\ 0 & 0 & 5 & 9 & 0 \end{bmatrix}$$

A_{val}	3	1	4	1	5	9	2	6	5	3	5	9
A_{ind}	2	5	1	2	2	3	4	1	4	5	3	4
A_{ptr}	1	3	5	8	11	13 (nonzero count + 1)						

are shown in Table II. Results of time analysis are presented in Table III and they show that Cholesky factorization and sparse matrix multiplication of the form ADA^T are the two most computationally expensive tasks of the solver. However, a few of the data sets, namely BPMPD, CZPROB, and NEMSEMM1, show that a considerable amount of time is spent executing other parts of the code. These results show the importance of accelerating Cholesky factorization and sparse multiplication in order to enhance the performance of IPM based LP solvers.

IV. Original implementation of GLPK matrix multiplication and cache problems

As mentioned earlier, the sparse matrix product S in IPM has the form $S = PAD(PA)^T$. This product is computed in two phases: symbolic and numeric. The symbolic phase is performed once and is used to determine the nonzero structure of S for use in the numeric phase.

The numeric phase, which is implemented based on Gustavson's algorithm [15], is executed every iteration to determine the numeric values of $s_{i,j\geq i}$ of S. Algorithm 1 shows the pseudocode of the GLPK serial implementation of $S = PAD(PA)^T$ where A is an $m \times n$ matrix and P is stored in a permutation vector π. The matrix product is computed row by row (line 1) and the permutation is applied at lines 2 and 5. If row k of S has n_{nz} nonzeros, then its computation requires multiplying row i (i_p after permutation) of A by D and by other n_{nz} rows of A. Since rows have different sparsity, row i_p is decompressed into a vector w (line 3) as illustrated in Algorithm 2 . Each nonzero in row i of S is finally obtained by performing a dot product between Dw and a row of A (line 6).

The GLPK implementation of $S = PAD(PA)^T$ suffers from the following problems with regard to cache utilization:

- Because of the sparsity of A, a small fraction of the values in D and w need to be read for the computation of each nonzero of S. However, these values are scattered irregularly over large vectors, D and w, that don't have room in L1 and L2 cache. Such irregular access pattern will cause a high cache miss rate.

TABLE II
Information about the test data sets

Problem name	Column count	Row count	Nonzeros	Sparsity $\times 10^{-4}$
FIT2D	10525	21024	150042	6.8
CZPROB	929	3333	10022	32.4
NEMSEMM1	5668	74151	1036227	24.7
WORLD	47259	79053	220891	0.6
NSCT2	23003	37563	697738	8.1
BPMPD	33841	1144020	3450992	0.9
OLIVIER	11144	22977	108562	4.2
BAS1LP	9872	14286	596697	42.3
DFL001	6084	12243	35658	4.8
QAP12	3192	8856	38304	13.5
QAP15	6330	22275	94950	6.7

TABLE III
Profiling serial GLPK

Problem name	ADA^T (%)	Cholesky (%)	Bck/fwd solver (%)	Others (%)
FIT2D	98.8	0.3	0.1	0.8
CZPROB	69.1	7.7	2.2	21.0
NEMSEMM1	66.3	13.7	1.0	19.0
WORLD	6.3	81.4	4.8	7.5
NSCT2	6.1	92.6	0.4	0.9
BPMPD	34.9	13.8	1.5	49.8
OLIVIER	33.4	57.5	3.0	6.1
BAS1LP	11.4	85.8	0.8	2.0
DFL001	0.2	98.7	0.8	0.3
QAP12	0.1	99.2	0.6	0.2
QAP15	0.0	98.7	0.3	1.0

Algorithm 1 Serial implementation of $S = PAD(PA)^T$, using A_α for row α of A.

1: **for** $i = 1 \rightarrow m$ **do**
2: $\quad i_p = \pi(i)$
3: $\quad w = (A_{i_p})^T$ {see Algorithm 2}
4: $\quad$ **for** $j = i \rightarrow m$**and**$s_{ij} \neq 0$ **do**
5: $\quad\quad j_p = \pi(j)$
6: $\quad\quad s_{ij} = A_{j_p} Dw$
7: $\quad$ **end for**
8: **end for**

- Although the rows of A have small number of values that are stored contiguously, the permutations makes it difficult to benefit from data locality and might cause much TLB misses for matrices with high number of nonzeros [13].

V. Cache-aware matrix multiplication

Trying to exploit cache and avoid the problems mentioned in the previous section, we use 1D and 2D partitioning and develop techniques to avoid the overhead of accessing zero blocks and zero block rows. Both extensions of the original algorithm avoid the negative effect of permutation by performing it during the blocking phase, i.e. the rows are permuted in memory before they are split into several blocks. This allows more uniform access to rows of partitions during multiplication. The new algorithms are explained in the following subsections.

A. 1D partitioning of the matrix A

The method is based on a vertical partitioning of PA into blocks $A^{(1)}, A^{(2)}, \ldots, A^{(v)}$, where v is the number of vertical partitions. Moreover, partitioning is made once since A is

Algorithm 2 Decompression of a row of matrix A into w.

1: **for** $k = A_{\text{ptr}}(i_p) \rightarrow A_{\text{ptr}}(i_p + 1)$ **do**
2: $\quad l = A_{\text{ind}}(k)$
3: $\quad w(l) = A_{\text{val}}(k)$
4: **end for**

constant and only D changes through the IPM iterations. In addition, each of the blocks is stored in memory as an independent matrix using CRS. Algorithm 3 shows the pseudocode of the 1D algorithm. D and w are accessed in smaller chunks $D^{(1)}, D^{(2)}, \ldots, D^{(v)}$ and w' whose size depends on the width of A-partitions. The goal is to have $D^{(\cdot)}$'s and w' that can fit into L1/L2 cache and be reused through loop iterations at line 5.

One of the drawbacks of 1D partitioning is that many of the partitioned rows have no elements (zero rows) which waste cycles on loading and comparing A_{ptr} values. Fig. 3A shows an example of a vertical partitioning where 60% of the blocked rows have no elements. In fact, the percentage of zero rows is much higher in real problems as the matrices are much more sparse than the one shown in Fig. 3A. Another drawback is the extra storage of ptr array of each partition.

To avoid wasting time on zero partition rows, a higher level of compressed storage is used to efficiently access the nonzero rows of partitions. The matrix, as shown in Fig. 3A, is treated as an $m \times v$ matrix and a new second level of CRS structure (only ptr and ind) is added and used to access nonzero partition rows. This adds more to storage requirements, but has a good effect on performance. The added $\text{Parts}_{\text{ptr}}$ and $\text{Parts}_{\text{ind}}$ are shown Fig. 3B, and the associated multiplication algorithm is shown in Algorithm 4.

B. 2D partitioning of the matrix A

Data blocking is a well known technique to utilize data spatial locality [20] and it is well suited for dense matrix multiplication. We try to apply the same technique to sparse matrix multiplication by dividing matrix A into $M \times N$ blocks and consequently matrix S into $M \times M$ blocks all stored using CRS. Computation of an S block is achieved by multiplying two rows of A blocks as shown in Algorithm 5. Blocks have different sparsity and many A and S blocks may have no elements (zero blocks). Different sparsity of different blocks is a reason why blocking is not as efficient as when dealing with dense matrices. However, trying to exploit the existence of zero blocks we add extra information to S blocks as shown in the following:

- Each block of S has an array that stores the indices of nonzero rows.

Algorithm 3 Vertically partitioned implementation of $S = PAD(PA)^T$, using $A_\alpha^{(p)}$ for row α of partition $A^{(p)}$.

Require: $A \leftarrow PA$ {performed when partitioning}

1: **for** $i = 1 \rightarrow m$ **do**
2: **for** partition $p = 1 \rightarrow v$ **do**
3: $w' = (A_i^{(p)})^T$
4: **for** $j = i \rightarrow m$**and** $s_{ij} \neq 0$ **do**
5: $s_{ij} += A_j^{(p)} D^{(p)} w'$
6: **end for**
7: **end for**
8: **end for**

Algorithm 4 Extension of Algorithm 3 with second level CRS

Require: $A \leftarrow PA$ {performed when partitioning}

1: **for** $i = 1 \rightarrow m$ **do**
2: **for** $t = \text{Parts}_{\text{ptr}}(i) \rightarrow \text{Parts}_{\text{ptr}}(i+1)$ **do**
3: partition $p = \text{Parts}_{\text{ind}}(t)$
4: $w' = (A_i^{(p)})^T$
5: **for** $j = i \rightarrow m$**and** $s_{ij} \neq 0$ **do**
6: $s_{ij} += A_j^{(p)} D^{(p)} w'$
7: **end for**
8: **end for**
9: **end for**

Algorithm 5 2D partitioning of matrix A

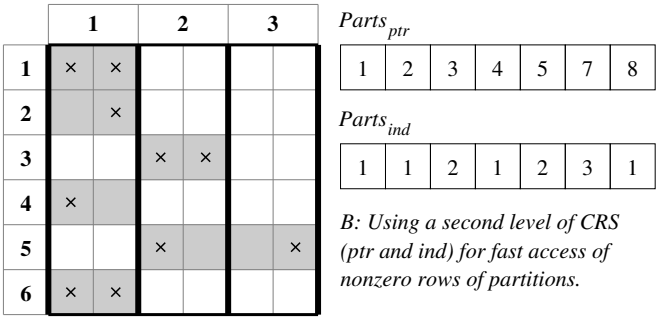
1: **for** $I = 1 \rightarrow M$ **do**
2: **for** $J = 1 \rightarrow M$ **do**
3: **for** $K = 1 \rightarrow N$ **do**
4: $S_{I,J} += A_{I,K} A_{J,K}$
5: **end for**
6: **end for**
7: **end for**

- Each block of S has an array of indices of participating pairs of A blocks.

The goal of the first point is to allow utilization of the already known S matrix structure. However, the second point aims at avoiding accessing A blocks that don't participate into computation of an S block. Two pairs of A blocks participate in the computation if their product produces one or more nonzeros, which is simply determined by checking if they have at least one common index of nonzero columns. We determine participating blocks just after the symbolic phase and use it through all IPM iterations. Suppose that matrix A has $M \times N$ blocks then S has $M \times M$ blocks as shown in Algorithm 5.

C. Parallel computation of S and load balancing

Parallelization of original GLPK implementation and our implementations of the sparse matrix multiplication have been achieved with OpenMP, mainly for loop parallelization. The original GLPK implementation is parallelized by parallelizing the for loop shown at line 1 in Algorithm 1, causing each core



A: Blocking A into 3 vertical partitions

B: Using a second level of CRS (ptr and ind) for fast access of nonzero rows of partitions.

Fig. 3. 1D partitioning of matrix A

Listing 1. Parallel block based matrix multiplication

```
#pragma omp parallel for
for row = 1 to M
  for col = 1 to M
    ...
```

A. Parallel 2D algorithm without load balancing

```
shares = assign_shares(num_cores, num_nonzeros[]);
#pragma omp parallel for
for i = 1 to num_cores
  for row = shares[i].from to shares[i].to
    for col = 1 to M
      ...
```

B. Parallel 2D algorithm with load balancing

to compute a chunk of S rows. Similarly, 1D partitioning of A also uses the same principle as the outer loop iterates over rows of S. Finally, multiplication based on 2D partitioning of A is parallelized by parallelizing the for loop shown at line 1 of Algorithm 5, causing each core to compute a chunk of rows of S blocks.

Due to different levels of sparsity within the same matrix in general, and as will be seen in the performance results section, our parallel implementation suffers from load imbalance. To address this, we divide S into a number of shares that equals the number of cores, and assign one core to each share. An ideal share would be any share whose number of nonzeros equals the number of nonzeros in S divided by the number of cores. Therefore, we try to assign shares such that they differ as less as possible from an ideal share. To force a core to compute one share, we add an outer loop over shares and apply the `omp parallel for` construct to the new added loop as shown in Listing 1.

A share is composed of a number of consecutive S rows in the original and 1D algorithms, but it is made of a number of consecutive rows of blocks in the 2D algorithm, making it harder to determine shares that are close to an ideal share. For the 2D algorithm, we try to determine shares that are bigger than an ideal share within a specified tolerance. If t denotes tolerance and d denotes number of nonzeros in an ideal share, then a share can have up to $(1+t)d$ nonzeros. In our implementation, we start trying a 5% tolerance and decide all the shares except the last one. If the size of the last share doesn't satisfy the tolerance constraint, we keep increasing the tolerance by 5% and repeat the algorithm until all shares respect the tolerance constraint.

VI. Related work

Our implementations, although not intended for general sparse matrix multiplication, are based on the classical Gustavson algorithm [15] using compressed row storage of matrices. That algorithm is also used in Csparse [19] and Matlab [21] and is proven to be optimal with respect to number of operations and storage space of general sparse matrices.

Algorithms for sparse multiplication are developed with focus on optimizing number of operations and storage requirements, however they only perform better than Gustavson's algorithm when working on certain class of matrices. For example, Park et al. [17] built an efficient algorithm that is based on a compact storage of banded and triangular matrices. On the other hand, Buluç et al. [16] introduced what they called the doubly compressed sparse column (DCSC) which uses less space than compressed column storage (CSC) for storing hypersparse matrices, matrices where number of nonzeros is less than the dimension of the matrix. Such matrices may be the result of a 2D partitioning of sparse matrices for parallel processing.

To our knowledge, Sulatycke et al. are the only researchers who presented sparse matrix algorithms that take efficiency of caches into consideration [14]. Their cache aware algorithms are based on interchanging loops of a standard multiplication algorithm. Moreover, they presented a parallel version that is based on static and dynamic splitting of matrix rows among several threads. However, their experiments were conducted on up to 1000 x 1000 10% sparse matrices, which are much smaller and less sparse than those tested in this paper.

Most recent research about sparse matrix multiplication have been performed by Buluç et al. In [16], they discussed the scalability limitations of matrix multiplication on thousands of processors. Moreover, they developed a sequential hypersparse matrix multiplication algorithm using the DCSC sparse storage to overcome the presented limitations. Parallel implementation was simulated by dividing input matrices using 2D blocking decomposition, excluding other costs like updates and parallelization overheads. Based on their work in [22], load imbalance, hiding communication costs, and additions of submatrices, are the main challenges of parallelizing sparse multiplication. In addition, they have also analysed the scalability of using 1D and 2D block decomposition to divide the work among the processors and show analytically and experimentally that the 2D based algorithms are more scalable than those based on 1D blocking.

VII. Performance results and conclusion

A. Cache aware matrix multiplication

Our experiments are performed on a 2 x 6 cores AMD Opteron (Istanbul 2431) compute node. All code including GLPK 4.43 is compiled with GCC 4.4.3, with optimization level 3 (-O3). Moreover, execution time of only the first IPM iteration has been measured when solving each of the data sets because iterations caused by solving one data set take the same amount of execution time. Table IV reports the execution time of original GLPK implementation and the new two implementations of the serial matrix multiplication, executed on a single core. Speedup is calculated taking the original implementation as a baseline. The following can be concluded:

1) The new 1D and 2D algorithms execute faster than the original one for all data sets. However, FIT2D is

accelerated much more than other data sets. This can be explained by the unique nonzero structure of its PA as shown in Fig. 4. The figure is created by placing a dot in the location of each nonzero element, i.e. the horizontal thin bar in the figure represents a group of adjacent dense rows in the matrix. The nonzero structure of this matrix is special because most rows have two values while the last few rows are dense. The original implementation is slow because it accesses most of the D values when one of the dense rows in the bottom of PA is involved, causing much L1 and L2 cache misses. However, the 1D and 2D implementations utilize the cache and improve locality of access as explained in section V.

2) Different data sets are accelerated by different values due to the difference in sparsity. Moreover, the distribution of nonzeros is different among different data sets.
3) 2D avoids accessing nonzero blocks and blocks whose multiplication doesn't result in any nonzeros, while 1D avoids accessing nonzero rows of partitions. 2D cause more performance when nonzeros are concentrated in chunks causing a lot of nonparticipating blocks to be avoided.

B. Size of partitions/blocks

Table V shows sizes of partitions/blocks that cause optimal speedup for both implementations and for different data sets. The results show that the optimal dimensioning of block-/partitions are more related to the distribution of nonzeros than to the sparsity of data sets. If we fix partition size in the 1D implementation to 100 and the block size in the 2D implementation to 100×100, the speedup of NSCT2 and BAS1LP is reduced by 16% and 14% respectively. However, the speedup of 2D partitioning for CZPROB, NEMSEMM1, NSCT2, BPMPD, OLIVIER, and BAS1LP is reduced by an average of 8%.

C. Parallel sparse matrix multiplication

The performance of the parallel matrix multiplication for original implementation and our implementations before and after load balancing is shown in Figs. 5-9. The results show the high importance of the load balancing. In addition, they show that problems that have longer serial execution time scale better than those which have relatively lower execution time.

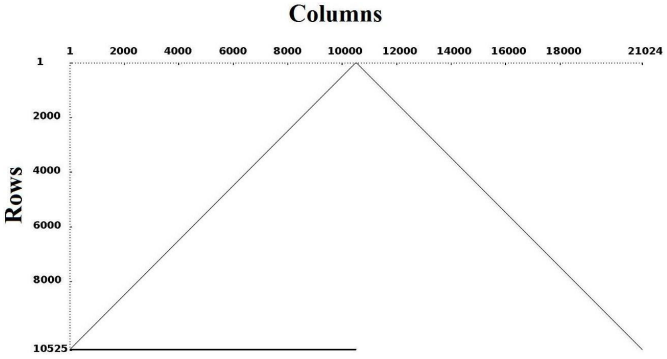


Fig. 4. Nonzero structure of FIT2D after permutation.

Although both implementations show a comparable speedup when executed serially, the later has better speedup when both are executed in parallel. The median speedup of the 1D and 2D implementations is 12.0 and 21.9. Table VI shows the speedup achieved when executing the implementations on 10 cores with the execution time of the original serial algorithm as a baseline. We chose to show the results on 10 cores since some data sets show bad performance when executed on 11 and 12 cores.

To have a more clear view, we show the parallel performance of NEMSEMM1 as an example, in Fig. 10. The figure shows that speedup doesn't increase smoothly with increasing number of cores. This behaviour is due to two main reasons. First, a strange varying OpenMP overhead is observed. It is measured as the difference between the matrix multiplication time and the execution time of the thread that takes most time to finish computing its share. Second, because nonzeros can be concentrated in a small part(s) of the matrix, using nonzeros to divide the shares among threads doesn't always guarantee that load balancing will be improved. For example, in the 2D algorithm, one thread might be responsible for computing many very sparse blocks, while a second one might be responsible for computing a much lower number of dense blocks. The overhead caused by these two reasons can have a relatively big effect on performance as shown when using 11 and 12 cores.

VIII. Conclusion and future work

An efficient LP solver is crucial for many scientific and industrial applications. However, most research has been fo-

TABLE IV
SERIAL TIMINGS OF ORIGINAL AND NEW IMPLEMENTATIONS

Problem name	Orig. [s]	1D [s] 1	2D [s] 3	Speedup 1	Speedup 3
FIT2D	2.455	0.0261	0.0227	94.1	108.0
CZPROB	0.004	0.0006	0.0007	6.6	5.7
NEMSEMM1	0.279	0.0783	0.0684	3.6	4.1
WORLD	0.049	0.0292	0.0414	1.7	1.2
NSCT2	2.015	1.6290	1.2490	1.2	1.6
BPMPD	0.433	0.1047	0.1211	4.1	3.6
OLIVIER	0.088	0.0164	0.0180	5.4	4.9
BAS1LP	0.992	0.7724	0.5933	1.3	1.7

TABLE V
SIZES OF PARTITIONS/BLOCKS

Problem name	Sparsity $\times 10^{-4}$	1D width	2D width x height
FIT2D	6.8	100	100 x 100
CZPROB	32.4	100	100 x 50
NEMSEMM1	24.7	100	100 x 50
WORLD	0.6	100	100 x 100
NSCT2	8.1	1500	100 x 50
BPMPD	0.9	100	100 x 70
OLIVIER	4.2	100	600 x 625
BAS1LP	42.3	400	200 x 50

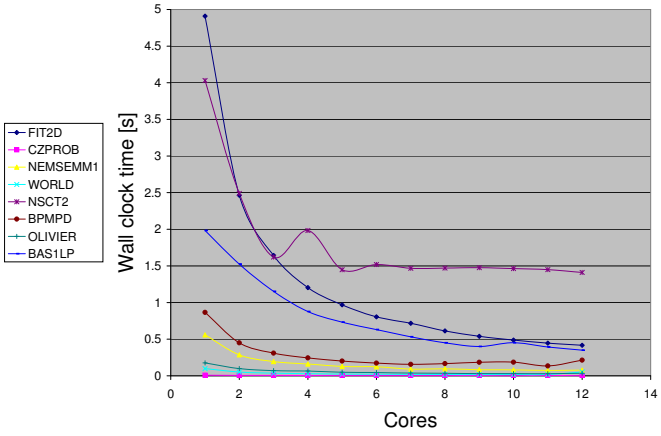


Fig. 5. Parallel original GLPK implementation without load balancing.

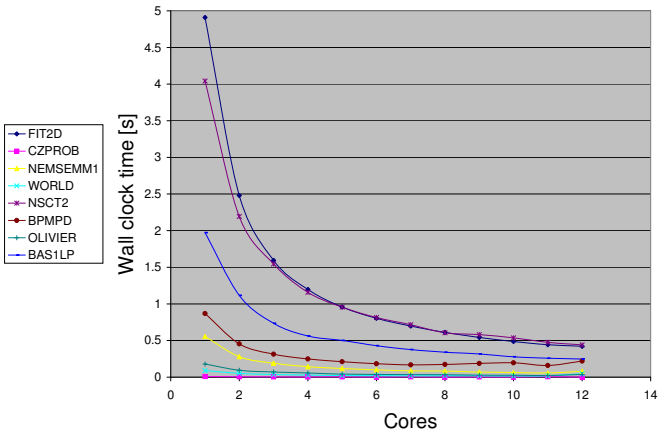


Fig. 6. Parallel original GLPK implementation with load balancing.

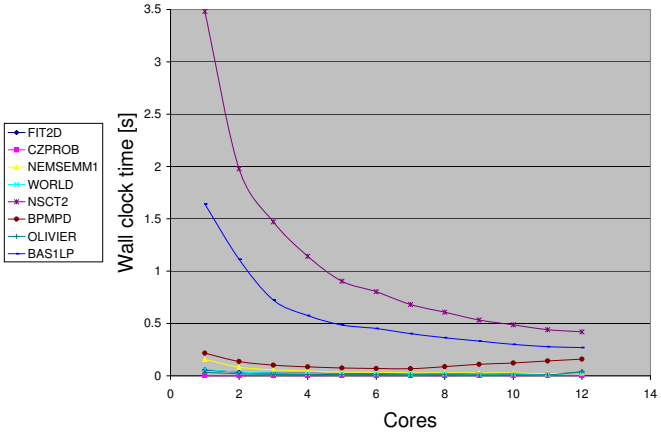


Fig. 7. Parallel 1D algorithm with load balancing.

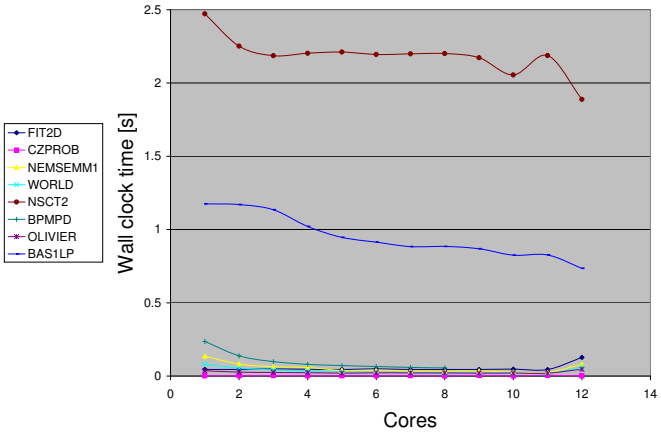


Fig. 8. Parallel 2D without load balancing.

cused on an efficient Cholesky factorization since it is the most expensive computation in interior point methods. We showed that, similar to Cholesky factorization, sparse matrix multiplication in IPM-based solvers use a relatively high percentage of the total execution time when solving some big data sets, and proposed two cache-aware implementations of the sparse multiplication algorithm used in GLPK. Moreover, we used OpenMP to parallelize the multiplication and developed a simple, but efficient technique for load balancing.

Due to many zero rows of very sparse blocks, CRS and CCS are not optimal wrt. space for storing very sparse blocks, but we had to use them for two reasons,

- Block sparsity varies a lot even in the same data set

TABLE VI
PARALLEL SPEEDUP OF THE TWO ALGORITHMS ON 10 CORES

Problem name	Speedup	
	1D	2D
FIT2D	270.3	374.4
CZPROB	15.6	23.9
NEMSEMM1	19.7	28.8
WORLD	7.4	8.9
NSCT2	8.3	9.5
BPMPD	7.0	24.1
OLIVIER	18.8	19.8
BAS1LP	6.6	10.2
Median Speedup	**12.0**	**21.9**

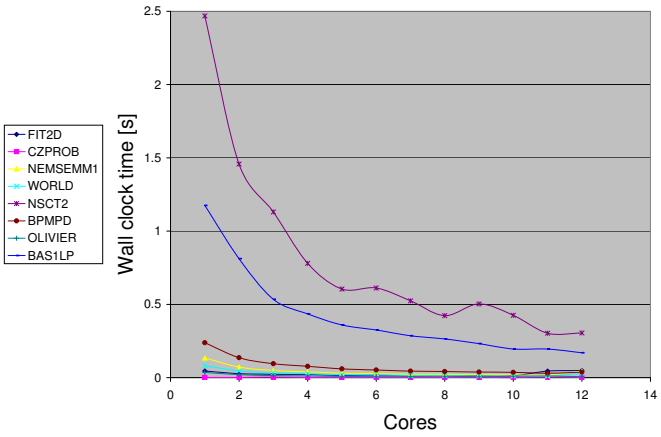


Fig. 9. Parallel 2D with load balancing.

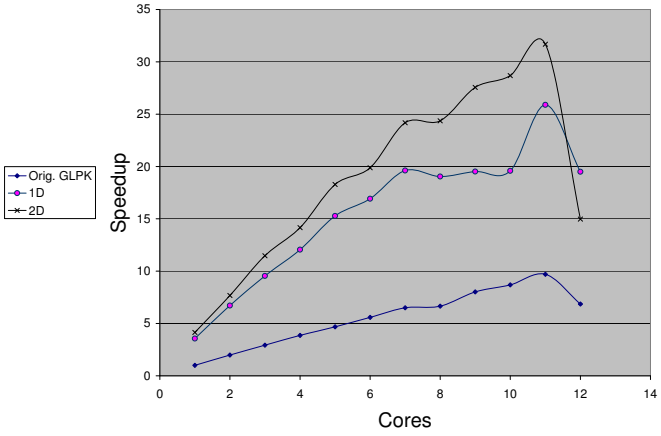


Fig. 10. NEMSEMM1 with load balancing. Original serial execution time is used as a baseline.

- Our algorithms requires very fast access to rows/columns.

It might be possible to use different storage and computation mechanisms for blocks based on their sparsity. One approach to accomplish this is to use 2D partitioning but using larger blocks, and then choose the appropriate storage mechanism and block multiplication procedure based on the sparsity level. In case of dense (or close to dense blocks), another level of blocking can be performed to better utilize the cache.

Since blocking is our main technique of exploiting cache, it is interesting to try our algorithms on other multicore systems that have different cache systems.

REFERENCES

[1] S. H. Fuller and L. I. Millett, "The Future of Computing Performance: Game Over or Next Level," *IEEE Computer,* vol. 44, pp. 31–38, January 2011.

[2] M. V. Wilkes, "The Memory Gap and the Future of High Performance Memories," *ACM Computer Architecture News,* vol. 29, pp. 2–7, March 2001.

[3] M. S. Lam, E. E. Rothberg, and M. E. Wolf, "The Cache Performance and Optimizations of Blocked Algorithms," *Proc. 4th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems,* California, pp. 63–74, April 1991.

[4] Miriam AS, available at http://www.miriam.as (accessed September 2010)

[5] D. G. Luenberger and Y. Ye, "Linear and Nonlinear Programming," *Springer Science,* 3rd ed., N.Y., 2007.

[6] S. Mehrotra, "On the Implementation of a Primal-Dual Interior Point Method", *SIAM J. on Optim.,* pp. 575–601, 1992.

[7] GLPK (GNU Linear Programming Kit), available at http://gnu.org/software/glpk/ (accessed: September 2010).

[8] The NETLIB LP Test Problem Set, available at http://www.numerical.rl.ac.uk/cute/netlib.html (accessed September 2010).

[9] BPMPD Home Page, available at http://www.sztaki.hu/~meszaros/bpmpd/ (accessed September 2010).

[10] Paul G. Howard, "Six-Core AMD Opteron processor Istanbul," white paper, Microway Inc., 2009.

[11] M. Smelyanskiy, V. W. Lee, D. Kim, A. D. Nguyen, and P. Dubey, "Scaling Performance of Interior-Point Method on Large-Scale Chip Multiprocessor System", *Proc. ACM/IEEE Conf. Supercomputing (SC07),* 2007.

[12] R. Shahnaz, A. Usman, and I. R. Chughtai, "Review of Storage Techniques for Sparse Matrices", *IEEE INMIC 2005 Conf. Proc.,* pp. 1–7, December 2005.

[13] K. Kaspersky, "Code Optimization: Effective Memory Usage," *A-List Publishing,* Wayne, Pennsylvania, 2003.

[14] P. D. Sulatycke and K. Ghose, "Caching Efficient Multithreaded Fast Multiplication of Sparse Matrices," *Proc. Merged Int'l Symp. Par. Proc. and Par. and Distr. Proc.,* pp. 117–124, 1998.

[15] F. G. Gustavson, "Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition," *ACM Trans. Math. Software,* vol. 4, pp. 250–269, 1978.

[16] A. Buluç and J. R. Gilbert, "On the Representation and Multiplication of Hypersparse Matrices," *IPDPS,* IEEE, pp. 1–11, 2008.

[17] S. C. Park, J. P. Draayer, and S. Q. Zheng, "Fast Sparse Matrix Multiplication," *Comp. Phys. Comm.,* vol. 70, pp. 557–568, 1992.

[18] R. Yuster and U. Zwick, "Fast Sparse Matrix Multiplication," *ACM Trans. on Algorithms,* vol. 1, pp. 2–13, 2005.

[19] T. A. Davis, "Direct Methods for Sparse Linear Systems," *Soc. for Ind. and Appl. Math.,* 2006.

[20] M. Kowarschik and C. Weiss, "An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms," *LNCS,* vol. 2625, pp. 213–232, 2003.

[21] J. R. Gilbert, C. Moler, and R. Schreiber, "Sparse Matrices in MATLAB: Design and Implementation," *SIAM J. Matrix Anal. and Appl.,* vol. 13, pp. 333–356, 1992.

[22] A. Buluç and J. R. Gilbert, "Challenges and Advances in Parallel Sparse Matrix-Matrix Multiplication", Int'l Conf. on Par. Proc. (ICPP'08), pp. 503-510, September 2008.