Sparse Matrix Decomposition with Optimal Load Balancing*

Ali Pınar and Cevdet Aykanat

Computer Engineering Department, Bilkent University
TR06533 Bilkent, Ankara, Turkey
{apinar/aykanat}@cs.bilkent.edu.tr

Abstract

Load balancing in the decomposition of sparse matrices without disturbing the row/column ordering is investigated. Both asymptotically and run-time efficient algorithms are proposed and implemented for one-dimensional (1D) striping and two-dimensional (2D) jagged partitioning. Bisection method is successfully adopted to 1D partitioning by deriving and exploiting tight bounds on the value of an optimal solution. A bisection algorithm is proposed for 2D jagged partitioning by introducing a new 2D probing scheme. A novel bidding algorithm is proposed for both 1D and 2D partitioning. Proposed algorithms are also space efficient since they only need the conventional compressed storage scheme for the given matrix, avoiding the need for a dense workload matrix in 2D decomposition. Experimental results on a large set of test matrices show that considerably better decompositions can be obtained by using optimal load balancing algorithms instead of heuristics. Proposed algorithms are 100 times faster than a single sparse matrix vector multiplication (SpMxV), in the 64-way 1D decompositions, on the overall average. Our jagged partitioning algorithms are only 60% slower than a single SpMxV computation in the 8×8-way 2D decompositions, on the overall average.

1 Introduction

Sparse matrix vector multiplication (SpMxV) constitutes the most time consuming operation in iterative solvers. Parallelization of SpMxV operation requires the decomposition and distribution of the coefficient matrix. Two objectives in the decomposition are the minimization of the communication requirement and the load imbalance. Graph theoretical approach is the most commonly used decomposition technique in the literature. Graph–partitioning based decomposition corresponds to one-dimensional decomposition (i.e., either rowwise or columnwise) through row/column permutations of the given matrix. We have recently proposed hypergraph–partitioning based decomposition schemes with better models for the communication requirement [4]. Both graph and hypergraph partitioning problems are NP-hard problems, and hence efficient heuristics are used for finding good decompositions. In graph/hypergraph approaches, both communication and load imbalance metrics are explicitly handled for minimization during the partitioning.

Graph/hypergraph partitioning based decomposition may not be appropriate for some applications. First, reordering the matrix may not be feasible for some reason, e.g., the matrix might be already ordered for factorization. Second, graph/hypergraph partitioning might be too expensive as a preprocessing step for the sake of parallelization. Finally, graph/hypergraph models may suffer from scalability in the decomposition of small matrices for large number of processors because of their one-dimensional decomposition restriction.

In this work, we investigate the decomposition of sparse matrices without disturbing the given row/column ordering. In this approach, communication volume metric is handled implicitly by the selection of proper matrix partitioning and parallel SpMxV computation schemes at the beginning. Here, partitioning scheme refers to the scheme to be used for partitioning the given matrix to K submatrices, where K denotes the number of processors. Communication cost is determined by the partitioning scheme and the associated SpMxV algorithm. That is, the communication cost is assumed to be independent of the matrix sparsity pattern. Hence, load balance is the only metric explicitly considered in the decomposition. *Cyclic (scattered)* partitioning schemes automatically resolve the load balancing problem. However, these schemes suffer from high communication cost. Block partitioning schemes considerably reduce the communication cost in two-dimensional decomposition. Uniform block partitioning easily achieves perfect load balance in dense matrices. However, load-balanced block partitioning becomes an important issue in the decomposition of irregularly sparse matrices.

We consider the load balancing problem in both one-dimensional (1D) and two-dimensional (2D) block partitioning schemes. 1D partitioning corresponds to *rowwise or columnwise block striping* [15]. Fig. 1(a) illustrates 4-

*This work is partially supported by the Commission of the European Communities, Directorate General for Industry under contract ITDC 204–82166 and The Scientific and Technical Research Council of Turkey under grant EEEAG-160

Figure 1: (a) 4-way rowwise striping and (b) 4×4-way rowwise jagged partitioning

way rowwise striping. 2D partitioning increases the scalability of the decomposition while reducing the volume of communication. *Block-checkboard partitioning* [15] leads to an efficient SpMxV algorithm with low communication requirement [10]. This partitioning scheme is also referred to as *rectilinear partitioning* [19] and *generalized block distribution (GBD)* [17]. This scheme is very well suited to dense matrices and matrices with uniform sparsity pattern. However, it is hard to achieve good load balance on sparse matrices with non-uniform sparsity pattern because of the restriction of rectilinear splits on both rows and columns. *Jagged rectilinear* partitioning is commonly used to alleviate this problem. In this scheme, rectilinear splits are restricted to either rows or columns of the matrix thus increasing the search space for load balancing. In rowwise (columnwise) jagged partitioning, matrix is partitioned into P horizontal (vertical) strips, and every horizontal (vertical) strip is independently partitioned into Q submatrices, where $K = P \times Q$. That is, splits span the entire matrix in one dimension, while they are jagged in the other dimension. This scheme is also referred to as *semi-generalized block distribution (SBD)* [17], *basic partitioning configuration* [5], and *multiple recursive decomposition (MRD)* [21]. Fig. 1(b) illustrates 4×4 rowwise jagged partitioning. Without loss of generality, we restrict our discussions to rowwise striped and jagged partitioning schemes. All results of this paper can easily be extended to columnwise schemes.

Despite the recent theoretical results on optimal block partitioning of workload arrays, heuristics are still commonly used in the sparse matrix community. This may be due to the ease of implementation, efficiency, and expectation of "good" quality decompositions. These heuristics are based on *recursive decomposition (RD)* of 1D workload arrays. For example, in rowwise striping, K-way decomposition is achieved through $\lg K$ bisection levels, where K is a power of 2. At each bisection step in a level, the current row stripe is divided evenly into two row stripes. Here, even division corresponds to two row stripes with equal numbers of nonzeros as much as possible. In jagged partitioning, this even bisection strategy is adopted both in the P-way row-striping and in the Q-way columnwise striping of every row stripe. Prime factorization of K and P, Q values is used to avoid the power-of-two restriction on these integer values for 1D and 2D decompositions, respectively [21].

Although optimal division can easily be achieved at every bisection step, the sequence of bisections may lead to poor load balancing. In Section 4, we demonstrate that qualities of the decompositions obtained through RD heuristic substantially deviate from those of the optimal ones through experimental results. In Sections 3.1 and 3.2, we propose efficient algorithms for optimal load-balancing in 1D striped and 2D jagged partitioning of sparse matrices. Experimental results presented in Section 4 demonstrate the feasibility of using optimal load balancing algorithms in sparse matrix domain. Proposed algorithms are 100 times faster than a single SpMxV computation, in the 64-way 1D decompositions, on the overall average. Initial implementations of our jagged partitioning algorithms are only 60% slower than a single SpMxV computation in the 64-way (8×8) 2D decompositions, on the overall average. Proposed algorithms are also feasible in terms of memory requirement since they only need the conventional compressed storage scheme for the given matrix contrary to the existing optimal partitioning algorithms which depend on the existence of a dense workload matrix for 2D decomposition.

2 Previous Work on Partitioning of Workload Arrays

1D partitioning of sparse matrices is equivalent to the *chains-on-chains* partitioning problem with unweighted edges. The objective of chains-on-chains partitioning problem is to divide a 1D task array T of length M into K consecutive parts such that the load of the maximally loaded part is minimized. In rowwise striping, $T[i]$ is equal to the number of nonzeros in row i of the given $M \times N$ sparse matrix. The algorithm paradigms used for the solution of the chains-on-chains problem can be classified as *probe* and *dynamic programming (DP)* approaches. The probe approach relies on repeated investigations for the existence of a partition with a *bottleneck* value no greater than a given value. The probe approach goes back to Iqbal's [11] and Bokhari's [2] works describing ϵ-approximate and optimal algorithms running in $O(MK\log(W_{tot}/\epsilon))$ and $O(MK^3)$ times, respectively. Here, W_{tot} denotes the sum of the weights in the workload array T, i.e., $W_{tot} = \sum_{i=1}^{M} T[i]$. Iqbal and Bokhari [12], and Nicol and O'Hallaron [18] later proposed an $O(MK\log M)$ algorithm, and finally Nicol [19] proposed an $O(M + (K\log M)^2)$ algorithm. Anily and Federgruen [1] initiated the DP approach with an $O(M^2K)$ algorithm. Hansen and Lih [9] independently proposed an $O(M^2K)$ algorithm. Choi and Narahari [6], Manne and Sørevik [16], and Olstad and Manne [20] introduced asymptotically faster $O(MK)$, $O((M-K)K\log K)$ and $O((M-K)K)$ algorithms, respectively.

Theoretical work on optimal 2D partitioning is relatively rare. Nicol [19] conjectured the NP-completeness of the block checkboard (2D rectilinear) partitioning problem by considering the closely related NP-complete multi-stage linear assignment problem [14]. The NP-completeness of this problem (GBD) has later been proven by Grigni and Manne [8]. Manne and Sørevik [17] extended the DP approach to optimal jagged partitioning of 2D workload arrays. Their algorithm runs in $O((M-P)(N-Q)PQ)$-time for jagged partitioning (SBD) of an $M \times N$ workload array to a $P \times Q$ processor array. In sparse matrix domain, the workload array T represents the sparsity pattern of the given matrix A, such that $T[i,j]=0$ and $T[i,j]=1$ if $a_{ij}=0$ and $a_{ij}=1$, respectively.

3 Proposed Load Balancing Algorithms

The objective of this paper is to formulate both asymptotically and run-time efficient optimal load-balancing algorithms for 1D striped and 2D jagged partitioning schemes. An optimal decomposition corresponds to a partitioning which minimizes the number of nonzeros in the most heavily loaded processor (bottleneck processor). The load of the bottleneck processor is called the bottleneck value of the partition. Efficiency in terms of memory requirement is also considered in these formulations since maintaining an $M \times N$ workload array for an $M \times N$ sparse matrix is not acceptable. So, our algorithms use either the *row compressed storage (RCS)* or *column compressed storage (CCS)* schemes for the given sparse matrix. RCS is used for rowwise striped and rowwise jagged partitioning schemes.

We have developed and experimented several optimal load balancing algorithms. In this section, we present and discuss only two algorithms for 1D striped and 2D jagged partitioning schemes due to the lack of space. These algorithms seem to be the most promising algorithms according to the current implementations. We restrict our discussion to probe-based approaches because of extremely high execution times of DP-based approaches on sparse test matrices. This finding is experimentally verified in Section 4.

3.1 One-Dimensional Striped Partitioning

In this section, we consider optimal K-way row striping of an $M \times N$ sparse matrix. Bisection method is a very promising approach for sparse matrix decomposition for the following two reasons. First, tight bounds can be set for the *bottleneck* value of an optimal solution. The bottleneck value B_{opt} of an optimal partition ranges between $LB=B^*$ and UB $=B^*+w_{max}$, where w_{max} is the maximum element in the workload array, and $B^*=W_{tot}/K$ is the bottleneck value of perfectly load balanced partition. In rowwise striping, $W_{tot}=Z$ corresponds to the total number of nonzeros in the sparse matrix, and w_{max} is the number of nonzeros in the most dense row. Note that $w_{max} \ll N$ in most sparse matrices arising in various fields. Second, the ϵ-approximation restriction does not apply since the workload array is composed of integers.

The generic bisection algorithm is illustrated in Fig. 2. The workload array T is such that $T[i]$ is equal to the number of nonzeros in the ith row r_i of the matrix, i.e., $T[i]=w_i$. Prefix-sum on the task array T enables the constant-time computation of the weight $W_{i,j}=\sum_{k=i}^{j} w_k$ of the

BISECT1D(T,K)
$w_{max} \leftarrow \max_{1\le i\le M}\{T[i]\}$;
Prefix sum on $T[0\ldots M]$;
$W_t \leftarrow T[M]$;
$LB \leftarrow W_t/K$;
$UB \leftarrow LB + w_{max}$;
repeat
  $midB \leftarrow (UB+LB)/2$;
  if *PROBE*$(K, midB)$ **then**
    $UB \leftarrow midB$;
  else
    $LB \leftarrow midB+1$;
until $UB \le LB$;
return $B_{opt} \leftarrow UB$;

PROBE(K,B)
  $B_s \leftarrow B$; $p \leftarrow 1$;
  while $p \le K$ **and** $B_s < W_t$ **do**
    $s_p \leftarrow BINSRCH(T, B_s)$;
    $B_s \leftarrow T[s_p]+B$;
    $p \leftarrow p+1$;
  if $B_s < W_t$ **then**
    return FALSE ;
  else
    return TRUE;

Figure 2: Bisection algorithm for 1D K-way rowwise striping.

row-stripe $r_i, r_{i+1}, \ldots, r_j$ through $W_{i,j}=T[j]-T[i-1]$. Integer weights in the task array restrict the optimal bottleneck value B_{opt} to *UB*–*LB* $=w_{max}$ distinct integer values within the range $LB \le B_{opt} <$ *UB*. Hence, binary search can be efficiently used to find B_{opt} through probes in the interval $[UB, LB]$.

Given a bottleneck value B, *PROBE*(T,K,B) tries to find a K-way partition of T with a bottleneck value no greater than B. *PROBE* finds the largest index s_1 so that $W_{1,s_1} \le B$, and assigns the row-stripe $r_1, r_2, \ldots r_{s_1}$ to processor 1. Hence, the first row in the second processor is r_{s_1+1}. Probe then similarly finds the largest index s_2 so that $W_{s_1+1,s_2} \le B$, and assigns the row-stripe $r_{s_1+1}, r_{s_1+2}, \ldots r_{s_2}$ to processor 2. This process continues until either all rows are assigned or the processors are exhausted. The former case denotes the existence of a partition with bottleneck value no greater than B, whereas the latter shows the inexistence of a partition with bottleneck value smaller than or equal to B [19]. As seen in Fig 2, the indices $s_1, s_2, \ldots s_{K-1}$ are efficiently found through binary search (*BINSRCH*) on the prefix-summed array T. Note that an optimal solution can easily be constructed by making a last *PROBE* call with B_{opt}.

The complexity of one *PROBE* call is $O(K \lg M)$. The bisection algorithm makes $\lg w_{max}$ *PROBE* calls. Thus, the overall complexity of the algorithm is $O(M + K \lg M \lg w_{max})$ together with the initial $O(M)$-time prefix-sum operation. The algorithm is surprisingly fast, so that the initial prefix-sum operation dominates the overall execution time. Fortunately, the data structure for the RCS scheme is efficiently exploited to avoid the initial prefix-sum operation without any additional operations, thus reducing the complexity to $O(K \lg M \lg w_{max})$.

In this work, we further exploit the nice bounds on optimal bottleneck value to restrict the search space for s_p separator values during *BINSRCH* in *PROBE* calls. That is, for each processor $p=1,2,\ldots K-1$, $SL_p \le s_p \le SH_p$, where SL_p and SH_p correspond to the smallest and largest indices such that $W_{1,SL_p} \ge p(B^*-w_{max}(K-p)/K)$ and

```
BIDDING(T, K)
    s_p ← 0 for p ← 0, 1, ..., K − 1;  and  s_K ← M;
    Perform prefix sum on T[0 ... M] with T[0] = 0;
    BIDS[0].B ← W_tot ← T[M];
    B ← W_tot/K;  p ← 0;
    while W_tot − T[s_{K−1}] > B do
        repeat p ← p + 1
          if s_p = 0 then
             s_p ← BINSRCH(T, T[s_{p−1}] + B) ;
          else
             while T[s_p + 1] − T[s_{p−1}] ≤ B do
                s_p ← s_p + 1;
          mybid ← T[s_p + 1] − T[s_{p−1}];
          if mybid ≤ BIDS[p − 1].B then
             BIDS[p].⟨B, q⟩ ← ⟨mybid, p⟩;
          else
             BIDS[p].⟨B, q⟩ ← BIDS[p − 1].⟨B, q⟩;
          rbid ← (W_tot − T[s_p])/(K − p)
        until rbid > B or p = K − 1;
        if rbid < BIDS[p].B then
           B ← rbid;
        else
           ⟨B, p⟩ ←BIDS[p].⟨B, q⟩;
        p ← p − 1;
    return B_opt ← B;
```

Figure 3: Bidding algorithm for 1D K-way rowwise striping.

$W_{1,SH_p} \le p(B^* + w_{max}(K - p/K))$, respectively. This scheme reduces the complexity of an individual probe call to $O(K \lg K + K \lg(w_{max}/w_{avg}))$, where $w_{avg} = W_{tot}/M$ denotes the average number of nonzeros per row. This reduces the overall complexity to $O(K \lg w_{max} \lg K + K \lg w_{max} \lg(w_{max}/w_{avg}) + K \lg M)$, together with the initial cost of $O(K \lg M)$ for setting the SL_p and SH_p values.

In this work, we propose a novel algorithm which also works for real-valued workload arrays. This new algorithm, namely the *BIDDING* algorithm, is presented in Fig. 3. In this algorithm, we dynamically increase the bottleneck value B, starting from the perfect bottleneck value B^*, until a feasible partition is obtained. This leads to an incremental probing scheme, where the decision is given by modifying the separators from the previous probe call. The separator indices $s_1, s_2, \ldots, s_{K-1}$ are set as the largest indices such that $W_{s_{p-1}+1,s_p} \le B$ with $s_0 = 0$ for $p = 1, 2, \ldots, K-1$. As in the conventional probing scheme, $W_{s_{K-1},M} > B$ denotes the infeasibility of the current B value. After detecting an infeasible B value, the important issue is to determine the next larger B value to be investigated. Undoubtfully, at least one of the separators should move to the right for a feasible partition. So, the next larger B value is computed by selecting the minimum of the set of $\{W_{s_{p-1}+1,s_p+1}\}_{p=1}^{K-1} \cup \{W_{s_{K-1}+1,M}\}$ values. We call the $W_{s_{p-1}+1,s_p+1}$ value the *bid* of processor p, which refers to the load of processor p if the first row r_{s_p+1} of the next processor is added to processor. Note that the bid of the last processor K is equal to the load of the remaining rows. If the best bid B comes from part p^*, probing with new B is performed only for the remaining processors $(p^*, p^* + 1, \ldots)$. In this scheme, we prefer to determine the new positions of the separators by moving them to the right one by one, since their new positions are likely to be in a close neighborhood of their previous values. Note that binary search is used only for setting the separator indices for the first time. As seen in Fig. 3, we maintain prefix-minimum array *BIDS* for computing the next larger B value in constant time. Here, *BIDS* is an array of records of length K, where *BIDS*$[p].B$ and *BIDS*$[p].q$ store the best bid value of the first p processors and the corresponding processor, respectively. *BIDS*$[0]$ is used to enable the running prefix-minimum operation.

After the separator index s_p is set for processor p, the *repeat-until-loop* terminates if it is not possible to partition the remaining segment $T_{s_p+1,M}$ into $K-p$ processors without exceeding the current B value, i.e., $rbid = (W_{tot} - T[s_p])/(K-p) > B$. In this case, the next larger B value is determined by considering the best bid among the first p processors and *rbid*. Here, *rbid* represents the bottleneck value of the perfect $(K-p)$-way partitioning of the remaining segment $T_{s_p+1,M}$. Note that this variable also stores the bid of the last processor, when all separator indices $s_1, s_2, \ldots, s_{K-1}$ are assigned nonzero values.

3.2 Two-Dimensional Jagged Partitioning

In this section, we consider optimal $(P \times Q)$-way rowwise jagged partitioning of an $M \times N$ sparse matrix. Bisection method can be extended to 2D jagged partitioning by setting tight bounds on the value of the optimal bottleneck value and defining an appropriate 2D probe function. We compute the bounds by constructing a conditionally optimal jagged partition as follows. We first construct an optimal 1D P-way rowwise striping of the given matrix. Then, we construct a jagged partition by constructing optimal 1D columnwise striping of every row stripe found in the first phase. The upper bound *UB* is set to the bottleneck value of this conditionally optimal jagged partition. The lower bound LB is computed by dividing the bottleneck value of the optimal rowwise striping by Q. The bounds on LB and *UB* can be derived as $LB \ge Z/PQ$ and *UB* $\le Z/PQ + w_{rmax}/P + w_{cmax}$, respectively. Hence, the search range for binary search will always be less than $w_{rmax}/P + w_{cmax}$ distinct integers. Here, w_{rmax} and w_{cmax} denote the number of nonzeros in the most dense row and column, respectively.

In this work, we propose a 2D probe algorithm. Given a bottleneck value B, *PROBE2D* tries to find a $P \times Q$-way jagged partition of matrix A with a bottleneck value no greater than B. *PROBE2D* finds the largest row index R_1 so that there exists a Q-way columnwise striping of the the row-stripe (submatrix) A_{1,R_1} with a bottleneck value no greater than B. *PROBE2D* then similarly finds the largest row index R_2 so that there exists a Q-way columnwise striping of the next row-stripe A_{R_1+1,R_2} with a bottleneck value no greater than B. This process continues until either all rows are consumed, or P row-stripes are obtained. The for-

BISECT2D(IA, LB, UB, P, Q)
repeat
$midB \leftarrow (LB + UB)/2;$
if *PROBE2D*$(IA, P, Q, midB)$ **then**
$UB \leftarrow midB;$
else
$LB \leftarrow midB + 1;$
until $UB = LB;$
return $B_{opt} \leftarrow UB;$

PROBE1D(T, B, Q)
$j \leftarrow 1;$
for $p \leftarrow 1$ **to** P **do**
$sum \leftarrow T[j];$
while $sum \leq B$ **and** $j \leq M$ **do**
$sum \leftarrow sum + T[j];$
if $j > M$ **and** $sum \leq B$ **then**
return TRUE;
else $j \leftarrow j - 1;$
return FALSE;

PROBE2D(IA, B, P, Q)
for $p \leftarrow 1$ **to** $P - 1$ **do**
$rs \leftarrow R_{p-1} + 1;\ \ R_p \leftarrow RL_p;$
$rl \leftarrow RL_p + 1;\ \ \ rh \leftarrow RH_p;$
while $rl < rh$ **do**
$rm \leftarrow (rl + rh)/2;$
construct $TC_{rs,rm};$
if *PROBE1D*$(TC_{rs,rm}, B, Q)$ **then**
$R_p \leftarrow rm;\ \ rl \leftarrow rm + 1;$
else
$rh \leftarrow rm - 1\ ;$
construct $TC_{R_{P-1}+1,N};$
if *PROBE1D*$(TC_{rs,N}, B, Q)$ **then**
for $p \leftarrow 1$ **to** $P - 1$ **do**
$RH_p \leftarrow R_p$
return TRUE;
else
for $p \leftarrow 1$ **to** $P - 1$ **do**
$RL_p \leftarrow R_p$
return FALSE;

Figure 4: Bisection algorithm for 2D $P \times Q$-way jagged partitioning

mer case denotes the existence of a jagged partition with bottleneck value no greater than B, whereas the latter shows the inexistence of a partition with bottleneck value smaller than or equal to B. In a similar way to our *BISECT1D* algorithm, we further exploit the nice upper bound on optimal bottleneck value to restrict the search space for R_p separator values during the binary search in *PROBE2D* calls. That is, for each processor $p = 1, 2, \ldots K-1$, $RL_p \leq R_p \leq RH_p$, where RL_p and RH_p correspond to the smallest and largest row indices such that $W_{1,RL_p} \geq UB \times Q \times p$ and $W_{1,RH_p} \leq W_{tot} - UB \times Q \times p$, respectively.

As seen in Fig. 4, *rl* and *rh* variables define the space to be explored for finding R_p values. The *while-loop* inside the *for-loop* implements the binary search for finding R_p in this space for the first $P-1$ row-stripes. Here, variable rs denotes the starting row index of the current row stripe p. At each binary search step rm, 1D workload array $TC_{rs,rm}$ is constructed for investigating the feasibility of a Q-way columnwise striping of the submatrix $A_{rs,rm}$ with a bottleneck value no greater than B. We favor a different 1D probing scheme (*PROBE1D*) which does not adopt prefix-sum, since the same row-stripe is not likely to be explored multiple times for Q-way 1D probing. The last *if-then-else* statement in the outer *for-loop*, determines the feasibility of the current B value by probing on the workload array of the remaining submatrix $A_{R_{P-1}+1,M}$. This *if-then-else* statement also involves an efficient scheme proposed for dynamically reducing the size of the search space for the R_p values in 2D probes. That is, whenever a bottleneck value B succeeds, current R_p values provide new tighter upper bounds for the following binary search steps. Similarly, for an unsuccessful B value, current R_p values provide new tighter lower bounds for the following binary search steps.

1D bidding algorithm presented in Section 3.1 is also extended to 2D jagged partitioning. The critical point in this algorithm is how to compute the next larger B value. In 2D bidding algorithm, P row-stripes bid for the next B value. The bid of each row-stripe is determined by the optimal bottleneck value of columnwise striping of the submatrix composed of the current rows of the stripe and the first row of the next stripe. Due to lack of space, we cannot present the details here.

Table 1: Properties of sparse test matrices.

name	number of row/col	number of non-zeros total	per row/column avg	min	max	ex. time SpMxV msecs
pilot87	2030	238624	117.55	1	738	43.90
cre-b	9648	398806	41.34	1	904	77.80
cre-d	8926	372266	41.71	1	845	72.20
ken-11	14694	82454	5.61	2	243	19.65
ken-18	105127	609271	5.80	2	649	167.40
CO9	10789	249205	23.10	1	707	51.45
CQ9	9278	221590	23.88	1	702	45.90
NL	7039	105089	14.93	1	361	22.55
mod2	34774	604910	17.40	1	941	124.05
world	34506	582064	16.87	1	972	119.45

4 Experimental Results

We have experimented the performance of the proposed load balancing algorithms for the rowwise striped and jagged partitioning of various test matrices arising in linear programming domain. Table 1 illustrates the properties of the test matrices. These test matrices are obtained from *Netlib* suite [7], and *IOWA Optimization Center* (ftp://col.biz.uiowa.edu:pub/testprob/lp/gondzio/). The sparsity pattern of these matrices are obtained by multiplying the respective rectangular constraint matrices with their tranposes. Table 1 also displays the execution time of a single SpMxV operation for each test matrix.

All algorithms are implemented in C language. All experiments are carried out on a workstation equipped with a 133MHz *PowerPC* with 512-KB external cache, and 64 MB of memory. We have experimented 16, 32, 64, 128, 256 way row-striping and 4×4, 4×8, 8×8, 8×16, 16×16 way jagged partitioning of every test matrix.

Table 2 illustrates relative performance results of various load balancing algorithms. In this table, RD refers to the recursive decomposition heuristic mentioned in Section 1. Recall that RD is equalivalent to MRD scheme mentioned earlier in Section 1. RD scheme is implemented as efficiently as possible by adopting binary search on 1D prefix-summed workload arrays. DP and Nic. refer to the dynamic programming and probe-based 1D decomposition schemes, implemented with respect to guidelines provided in [6, 20] and [19], respectively. Bsct and Bid stand for the proposed bisection and bidding algorithms described in Section 3.

In Table 2, percent load imbalance values are computed as $(W_{max} - B^*)/B^*$, where W_{max} denotes the number of nonzeros in the most heavily loaded processor (part, submatrix), and $B^* = Z/K$ denotes the number of nonzeros in every processor under perfectly balanced partitioning. OPT denotes the percent load imbalance obtained by the optimal partitioning algorithms. The table clearly shows that considerably better decompositions can be obtained by using optimal load balancing algorithms instead of heuristics. The quality gap between the solutions of optimal algorithms and heuristics increases with decreasing granularity, as expected. As also expected, 2D jagged partitioning always produces better decompositions than 1D striping. This quality gap becomes considerably large for larger number of processors.

Table 2 displays the execution times of various decomposition algorithms normalized with respect to a single SpMxV time. Note that normalized execution times of 1D decomposition algorithms are multiplied by 100 because of the difficulty of displaying extremely low execution times of the proposed bisection (Bsct), and bidding (Bid) algorithms. DP approaches are not recommended for sparse matrix decomposition because of their considerably large execution times relative to those of the proposed algorithms. In 1D decompostion of sparse matrices, both of our algorithms are definitely faster than Nicol's algorithm, the best known chains-on-chains partitioning algorithm. Although our algorithms are slower than RD heuristic, their additional processing time should be justified because of their considerably better decompositon quality and extremely low execution times compared to a single SpMxV computation time.

The execution times for 2D partitioning algorithms are relatively high compared to 1D partitioning, however the quality of the partitions and the execution times of the initial implementations encourage further research for faster algorithms and implementations.

5 Conclusion and Future Research

Efficient optimal load balancing algorithms were proposed for 1D striped and 2D jagged partitioning of sparse matrices. Experimental results on a large set of test matrices verified that considerably better decompositions can be obtained by using optimal load balancing algorithms instead of heuristics. The proposed algorithms were found to be orders of magnitudes faster than a single matrix-vector multiplication in 1D decomposition. The proposed algorithms for 2D partitioning are slightly slower than a matrix-vector multiplication, while producing significiantly better decompositions than the heuristics. We are currently working on improving the speed performance of our 2D load balancing algorithms.

References

[1] Anily, S., and Federgruen, A. Structured partitioning problems. *Operations Research*, **13** (1991), 130–149.

[2] Bokhari, S. H. Partitioning problems in parallel, pipelined, and distributed computing. *IEEE Trans. Computers*, **37**, 1 (1988), 48–57.

[3] Bultan, T., and Aykanat, C. A new mapping heuristic based on mean field annealing. *J. Parallel and Distributed Computing*, **16** (1992), 292–305.

[4] Catalyurek, U. V., and Aykanat, C. Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication. submitted to special issue of *J. Parallel and Distributed Computing* on Irregular Problems.

[5] Charny, B. Matrix partitioning on a virtually shared memory parallel machine. *IEEE Trans. Parallel and Distributed Systems*, **7**, 4 (1996), 343–355.

[6] Choi, H., andn Narahari, B. Algorithms for mapping and partitioning chain structured parallel computations. *Proc. 1991 Int. Conf. on Parallel Processing*, 1991, pp. I-625–I-628.

[7] Gay, D. M. Electronic mail distribution of linear programming test problems. *Mathematical Programming Society COAL Newsletter*, 1985.

[8] Grigni, M., and Manne, F. On the complexity of the generalized block distribution. *Proc. 3rd Int. Workshop on Parallel Algorithms for Irregularly Structured Problems (IRREGULAR'96)*, 1996, pp. 319–326.

[9] Hansen, P., and Lih, K. W. Improved algorithms for partitioning problems in parallel, pipelined and distributed computing. *IEEE Trans. Computers*, **41**, 6 (June 1992), 769–771.

[10] Hendrickson, B., Leland, R., and Plimpton, S. An efficient parallel algorithm for matrix-vector multiplication. *Int. J. High Speed Computing* **7**, 1 (1995), 73–88.

[11] Iqbal, M. A. Approximate algorithms for partitioning and assignment problems. Tech. Rep. 86-40, ICASE, 1986.

[12] Iqbal, M. A., and Bokhari, S. H. Efficient algorithms for a class of partitioning problems. Tech. Rep. 90-49, ICASE, 1990.

[13] Karypis, G., and Kumar, V. A fast and high quality multilevel scheme for partitioning irregular graphs. Tech. Rep., Dept. of Computer Science, University of Minnesota, 1995.

[14] Kincaid D. M., Nicol, D M., Shier, D., and Richards, D. A multistage linear array assignment problem. *Operations Research*, **38**, 6 (1990), 993–1005.

[15] Kumar. V., Grama, A., Gupta, A., and Karypis, G. *Introduction to Parallel Computing*. Benjamin/Cummings, 1994.

[16] Manne, F., and Sørevik, T. Optimal partitioning of sequences. *J. Algorithms*, **19** (1995), 235–249.

[17] Manne, F., and Sørevik, T. Partitioning an array onto a mesh of processors. *Proc. 3rd Int. Workshop on Applied Parallel Computing (PARA'96)*, 1996, pp. 467–476.

[18] Nicol, D. M., and O'Hallaron, D. R. Improved algorithms for mapping pipelined and parallel computations. *IEEE Trans. Computers*, **40**, 3 (1991) 295–306.

[19] Nicol, D. M. Rectilinear partitioning of irregular data parallel computations. *J. of Parallel and Disributed Computing*, **23** (1994), 119–134.

[20] Olstad, B., and Manne, F. Efficient partitioning of sequences. *IEEE Trans. Computers*, **44**, 11 (1995), 1322–1326.

[21] Romero, L. F., and Zapata, E. L. Data distributions for sparse matrix vector multiplication. *Parallel Computing*, **21** (1995), 583–605.

Table 2: Relative performance results of various load balancing algorithms.

		1D Decompostion (rowwise striping)							2D Decomposition (rowwise jagged)					
name	K	percent load imbalance		100 $\times$ execution time normalized wrt SpMxV					K	percent load imbalance		execution time norm. wrt SpMxV		
		OPT	RD	RD	DP	Nic.	Bsct	Bid	PxQ	OPT	RD	RD	Bid	Bsct
pilot87	16	1.15	1.15	0.05	12	4	0.34	0.06	4x4	0.46	0.52	0.46	0.54	0.50
	32	1.85	4.39	0.07	25	13	0.46	0.23	4x8	0.66	1.02	0.46	0.51	0.48
	64	2.99	5.89	0.11	50	44	1.03	0.46	8x8	0.98	1.27	0.46	0.64	0.56
	128	6.05	12.22	0.25	96	148	1.94	1.20	8x16	1.76	2.78	0.47	0.76	0.67
	256	14.26	36.14	0.43	174	495	3.64	4.50	16x16	2.13	4.38	0.48	0.97	0.77
cre-b	16	0.25	0.84	0.03	33	4	0.64	0.06	4x4	0.10	0.21	0.49	0.67	0.60
	32	0.81	3.92	0.05	66	16	0.84	0.13	4x8	0.30	1.23	0.50	1.38	0.69
	64	1.20	6.12	0.09	135	61	1.61	0.22	8x8	0.49	1.92	0.53	1.58	0.86
	128	2.51	17.05	0.18	275	214	2.44	0.80	8x16	0.88	3.22	0.54	2.08	1.18
	256	10.02	20.74	0.37	533	671	3.53	1.99	16x16	1.42	5.92	0.61	3.58	1.84
cre-d	16	0.45	0.53	0.03	32	4	0.55	0.03	4x4	0.14	0.38	0.50	0.62	0.60
	32	0.63	3.74	0.06	70	18	0.83	0.10	4x8	0.25	0.61	0.51	0.82	0.62
	64	1.73	4.34	0.10	137	62	1.52	0.28	8x8	0.64	1.98	0.53	1.16	0.99
	128	2.88	16.70	0.19	274	218	2.63	0.80	8x16	1.09	7.00	0.53	1.86	1.16
	256	10.85	35.20	0.37	532	677	3.88	1.63	16x16	1.57	6.25	0.60	3.32	2.05
ken-11	16	0.21	0.98	0.10	227	19	3.56	0.25	4x4	0.07	0.11	0.66	1.83	1.40
	32	1.18	3.74	0.15	460	77	4.83	1.91	4x8	0.13	0.13	0.68	2.60	1.50
	64	1.29	13.17	0.31	930	292	6.62	2.67	8x8	0.36	1.14	0.88	6.11	3.28
	128	6.80	13.17	0.76	1859	1011	10.43	20.36	8x16	0.59	1.21	0.96	10.28	2.75
	256	7.11	50.89	1.53	3618	3360	16.03	36.01	16x16	1.22	2.46	1.24	27.30	4.99
ken-18	16	0.17	0.56	0.01	243	5	3.55	0.18	4x4	0.00	0.01	0.80	1.78	1.25
	32	0.23	2.26	0.02	499	19	3.76	0.27	4x8	0.08	0.38	0.81	9.55	3.06
	64	0.89	2.26	0.06	996	73	4.24	1.82	8x8	0.15	0.59	0.99	20.74	4.51
	128	0.95	9.08	0.13	1950	269	5.23	2.82	8x16	0.21	1.37	0.99	23.67	3.99
	256	4.58	9.08	0.26	3822	980	7.35	27.63	16x16	0.42	1.72	1.36	60.82	8.91
CO9	16	0.34	1.37	0.02	63	6	0.97	0.10	4x4	0.36	0.40	0.52	1.31	0.84
	32	1.07	4.60	0.06	126	24	1.55	0.39	4x8	0.45	1.48	0.53	1.51	0.86
	64	1.93	4.96	0.12	254	94	2.53	1.21	8x8	1.08	1.62	0.58	3.40	1.40
	128	4.73	18.75	0.25	533	328	4.47	3.98	8x16	1.39	6.12	0.59	5.15	1.75
	256	13.62	42.58	0.52	1013	1062	5.54	19.44	16x16	1.90	13.20	0.62	9.40	3.96
CQ9	16	0.58	0.58	0.04	55	7	0.98	0.22	4x4	0.28	0.66	0.50	1.07	0.73
	32	0.80	2.24	0.07	118	26	1.42	0.22	4x8	0.48	1.84	0.51	2.00	0.77
	64	1.43	7.64	0.13	233	95	2.40	0.87	8x8	0.86	1.84	0.56	2.39	1.13
	128	3.51	22.34	0.33	485	336	3.70	3.05	8x16	1.38	4.44	0.57	4.11	1.81
	256	14.72	58.62	0.59	908	1104	6.54	17.65	16x16	2.36	13.00	0.66	8.68	3.44
NL	16	0.35	1.20	0.04	89	12	2.00	0.22	4x4	0.26	0.49	0.55	1.51	0.93
	32	0.85	3.44	0.13	173	44	2.44	0.55	4x8	0.46	1.89	0.55	1.88	0.89
	64	2.37	5.60	0.31	393	167	4.21	2.11	8x8	0.79	2.86	0.61	3.48	1.71
	128	4.99	22.78	0.58	717	597	5.99	5.21	8x16	1.34	5.84	0.63	4.41	1.86
	256	14.25	60.78	1.15	1419	1714	11.31	23.61	16x16	1.83	5.48	0.75	10.13	4.52
mod2	16	0.03	0.06	0.02	92	5	1.41	0.02	4x4	0.05	0.11	0.63	1.80	1.21
	32	0.07	0.19	0.04	185	20	1.65	0.08	4x8	0.09	0.13	0.64	2.12	1.06
	64	0.18	2.18	0.08	374	73	2.14	0.24	8x8	0.11	0.20	0.64	3.57	1.59
	128	0.41	2.46	0.18	746	275	3.35	0.58	8x16	0.28	1.84	0.65	5.20	2.14
	256	1.23	18.92	0.34	1500	959	5.24	3.69	16x16	0.43	2.92	0.80	14.05	4.09
world	16	0.04	0.09	0.02	92	5	1.34	0.04	4x4	0.05	0.06	0.56	1.90	1.00
	32	0.08	0.27	0.03	192	20	1.72	0.08	4x8	0.10	0.38	0.57	2.20	1.09
	64	0.28	4.73	0.09	376	78	2.26	0.36	8x8	0.17	0.52	0.66	3.65	1.64
	128	0.76	6.37	0.18	762	282	3.43	1.78	8x16	0.26	1.29	0.66	4.02	2.06
	256	1.11	27.41	0.36	1468	1023	5.40	2.89	16x16	0.50	3.36	0.80	12.39	4.86