Available online at www.sciencedirect.com
SCIENCE DIRECT®
J. Parallel Distrib. Comput. 64 (2004) 974–996
Journal of Parallel and Distributed Computing
http://www.elsevier.com/locate/jpdc

Fast optimal load balancing algorithms for 1D partitioning ☆

Ali Pınar[a,1] and Cevdet Aykanat[b,*]

[a] Computational Research Division, Lawrence Berkeley National Laboratory, USA
[b] Department of Computer Engineering, Bilkent University, Ankara 06533, Turkey

Received 30 March 2000; revised 5 May 2004

Abstract

The one-dimensional decomposition of nonuniform workload arrays with optimal load balancing is investigated. The problem has been studied in the literature as the "chains-on-chains partitioning" problem. Despite the rich literature on exact algorithms, heuristics are still used in parallel computing community with the "hope" of good decompositions and the "myth" of exact algorithms being hard to implement and not runtime efficient. We show that exact algorithms yield significant improvements in load balance over heuristics with negligible overhead. Detailed pseudocodes of the proposed algorithms are provided for reproducibility. We start with a literature review and propose improvements and efficient implementation tips for these algorithms. We also introduce novel algorithms that are asymptotically and runtime efficient. Our experiments on sparse matrix and direct volume rendering datasets verify that balance can be significantly improved by using exact algorithms. The proposed exact algorithms are 100 times faster than a single sparse-matrix vector multiplication for 64-way decompositions on the average. We conclude that exact algorithms with proposed efficient implementations can effectively replace heuristics.

Keywords: One-dimensional partitioning; Optimal load balancing; Chains-on-chains partitioning; Dynamic programming; Iterative refinement; Parametric search; Parallel sparse matrix vector multiplication; Image-space parallel volume rendering

1. Introduction

This article investigates block partitioning of possibly multi-dimensional nonuniform domains over one-dimensional (1D) workload arrays. Communication and synchronization overhead is assumed to be handled implicitly by proper selection of ordering and parallel computation schemes at the beginning so that load balance is the only metric explicitly considered for decomposition. The load balancing problem in the partitioning can be modeled as the *chains-on-chains partitioning* (CCP) problem with nonnegative task weights and unweighted edges between successive tasks. The objective of the CCP problem is to find a sequence of $P-1$ separators to divide a chain of N tasks with associated computational weights into P consecutive parts so that the *bottleneck value*, i.e., maximum load among all processors, is minimized.

The first polynomial time algorithm for the CCP problem was proposed by Bokhari [4]. Bokhari's $O(N^3P)$-time algorithm is based on finding a minimum path on a layered graph. Nicol and O'Hallaron [28] reduced the complexity to $O(N^2P)$ by decreasing the number of edges in the layered graph. Algorithm paradigms used in following studies can be classified as *dynamic programming* (DP), *iterative refinement*, and *parametric search*. Anily and Federgruen [1] initiated the DP approach with an $O(N^2P)$-time algorithm. Hansen and Lih [13] independently proposed an $O(N^2P)$-time algorithm. Choi and Narahari [6], and Olstad and Manne [30] introduced asymptotically faster $O(NP)$-time, and $O((N-P)P)$-time DP-based algorithms, respectively. The iterative refinement approach starts with a partition and iteratively tries to improve the solution. The $O((N-P)P\log P)$-time

☆ This work is partially supported by The Scientific and Technical Research Council of Turkey under grant EEEAG-103E028.
*Corresponding author. Fax: +90-312-2664047.
E-mail addresses: apinar@lbl.gov (A. Pınar), aykanat@cs.bilkent.edu.tr (C. Aykanat).
[1] Supported by the Director, Office of Science, Division of Mathematical, Information, and Computational Sciences of the US Department of Energy under contract DE-AC03-76SF00098. One Cyclotron Road MS 50F, Berkeley, CA 94720.

0743-7315/$ - see front matter © 2004 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2004.05.003

algorithm proposed by Manne and Sørevik [23] falls into this class. The parametric-search approach relies on repeatedly probing for a partition with a *bottleneck* value no greater than a given value. Complexity of probing is $\theta(N)$, since each task has to be examined, but can be reduced to $O(P \log N)$ through binary search, after an initial $\theta(N)$-time prefix-sum operation on the task chain [18]. Later the complexity was reduced to $O(P \log(N/P))$ by Han et al. [12].

The parametric-search approach goes back to Iqbal's work [16] describing an ε-approximate algorithm which performs $O(\log(W_{\text{tot}}/\varepsilon))$ probe calls. Here, W_{tot} denotes the total task weight and $\varepsilon > 0$ denotes the desired accuracy. Iqbal's algorithm exploits the observation that the bottleneck value is in the range $[W_{\text{tot}}/P, W_{\text{tot}}]$ and performs binary search in this range by making $O(\log(W_{\text{tot}}/\varepsilon))$ probes. This work was followed by several exact algorithms involving efficient schemes for searching over bottleneck values by considering only subchain weights. Nicol and O'Hallaron [28,29] proposed a search scheme that requires at most $4N$ probes. Iqbal and Bokhari [17] relaxed the restriction of this algorithm on bounded task weight and communication cost by proposing a condensation algorithm. Iqbal [15] and Nicol [27,29] concurrently proposed an efficient search scheme that finds an optimal partition after only $O(P \log N)$ probes. Asymptotically more efficient algorithms were proposed by Frederickson [7,8] and Han et al. [12]. Frederickson proposed an $O(N)$-time optimal algorithm. Han et al. proposed a recursive algorithm with complexity $O(N + P^{1+\epsilon})$ for any small $\epsilon > 0$. These two studies have focused on asymptotic complexity, disregarding practice.

Despite these efforts, heuristics are still commonly used in the parallel computing community, and design of efficient heuristics is still an active area of research [24]. The reasons for preferring heuristics are ease of implementation, efficiency, the expectation that heuristics yield good partitions, and the misconception that exact algorithms are not affordable as a preprocessing step for efficient parallelization. By contrast, our work proposes efficient exact CCP algorithms. Implementation details and pseudocodes for proposed algorithms are presented for clarity and reproducibility. We also demonstrate that qualities of the decompositions obtained through heuristics differ substantially from those of optimal ones through experiments on a wide range of real-world problems.

For the runtime efficiency of our algorithms, we use an effective heuristic as a preprocessing step to find a good upper bound on the optimal bottleneck value. Then we exploit lower and upper bounds on the optimal bottleneck value to restrict the search space for separator-index values. This separator-index bounding scheme is exploited in a *static* manner in the DP algorithm, drastically reducing the number of table entries computed and referenced. A *dynamic* separator-index bounding scheme is proposed for parametric search algorithms, narrowing separator-index ranges after each probe. The upper bound on the optimal bottleneck value is also exploited to find a much better initial partition for the iterative-refinement algorithm proposed by Manne and Sørevik [23]. We also propose a different iterative-refinement technique, which is very fast for small-to-medium number of processors. Observations behind this algorithm are further used to incorporate the subchain-weight concept into Iqbal's [16] approximate bisection algorithm to make it an exact algorithm.

Two applications are investigated in our experiments: sparse matrix–vector multiplication (SpMxV) which is one of the most important kernels in scientific computing and image-space parallel volume rendering which is widely used for scientific visualization. Integer and real valued workload arrays arising in these two applications are their distinctive features. Furthermore, SpMxV, a fine-grain application, demonstrates the feasibility of using optimal load balancing algorithms even in sparse-matrix decomposition. Experiments with proposed CCP algorithms on a wide range of sparse test matrices show that 64-way decompositions can be computed 100 times faster than a single SpMxV time, while reducing the load imbalance by a factor of four over the most effective heuristic. Experimental results on volume rendering dataset show that exact algorithms can produce 3.8 times better 64-way decompositions than the most effective heuristic, while being only 11 percent slower on average.

The remainder of this article is organized as follows. Table 1 displays the notation used in the paper. Section 2 defines the CCP problem. A survey of existing CCP algorithms is presented in Section 3. Proposed CCP algorithms are discussed in Section 4. Load-balancing applications used in our experiments are described in Section 5 and performance results are discussed in Section 6.

2. Chains-on-chains partitioning (CCP) problem

In the CCP problem, a computational problem, decomposed into a chain $\mathcal{T} = \langle t_1, t_2, \ldots, t_N \rangle$ of N task/modules with associated *positive* computational weights $\mathcal{W} = \langle w_1, w_2, \ldots, w_N \rangle$, is to be mapped onto a chain $\mathcal{P} = \langle \mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_P \rangle$ of P *homogeneous* processors. It is worth noting that there are no precedence constraints among the tasks in the chain. A *subchain* of $\mathcal{T}$ is defined as a subset of contiguous tasks, and the subchain consisting of tasks $\langle t_i, t_{i+1}, \ldots, t_j \rangle$ is denoted as $\mathcal{T}_{i,j}$. The computational load $W_{i,j}$ of subchain $\mathcal{T}_{i,j}$ is $W_{i,j} = \sum_{h=i}^{j} w_h$. From the contiguity constraint, a partition Π should map contiguous

Table 1
The summary of important abbreviations and symbols

Notation	Explanation
N	number of tasks
P	number of processors
$\mathcal{P}$	processor chain
$\mathcal{P}_i$	ith processor in the processor chain
$\mathcal{T}$	task chain, i.e., $\mathcal{T} = \langle t_1, t_2, \ldots, t_N \rangle$
t_i	ith task in the task chain
$\mathcal{T}_{ij}$	subchain of tasks starting from t_i upto t_j, i.e., $\mathcal{T}_{ij} = \langle t_i, t_{i+1}, \ldots, t_j \rangle$
$\mathcal{T}_i^p$	subproblem of p-way partitioning of the first i tasks in the task chain $\mathcal{T}$.
w_i	computational load of task t_i
$w_{\max}$	maximum computational load among all tasks
w_{avg}	average computational load of all tasks
W_{ij}	total computational load of task subchain T_{ij}
W_{tot}	total computational load
$\mathcal{W}[i]$	total weight of the first i tasks
Π_i^p	partition of first i tasks in the task chain onto the first p processors in the processor chain
L_p	load of the pth processor in a partition
UB	upperbound on the value of an optimal solution
LB	lower bound on the value of an optimal solution
B^*	ideal bottleneck value, achieved when all processors have equal load.
B_i^p	optimal solution value for p-way partitioning of the first i tasks
s_p	index of the last task assigned to the pth processor.
SL_p	lowest position for the pth separator index in an optimal solution
SH_p	highest position for the pth separator index in an optimal solution

subchains to contiguous processors. Hence, a P-way chain-partition Π_N^P of a task chain $\mathcal{T}$ with N tasks onto a processor chain $\mathcal{P}$ with P processors is described by a sequence $\Pi_N^P = \langle s_0, s_1, s_2, \ldots, s_P \rangle$ of $P+1$ separator indices, where $s_0 = 0 \leqslant s_1 \leqslant \cdots \leqslant s_P = N$. Here, s_p denotes the index of the last task of the pth part so that $\mathcal{P}_p$ gets the subchain $\mathcal{T}_{s_{p-1}+1,s_p}$ with load $L_p = W_{s_{p-1}+1,s_p}$. The cost $C(\Pi)$ of a partition Π is determined by the maximum processor execution time among all processors, i.e., $C(\Pi) = B = \max_{1 \leqslant p \leqslant P} \{L_p\}$. This B value of a partition is called its *bottleneck* value, and the processor/part defining it is called the *bottleneck processor/part*. The CCP problem can be defined as finding a mapping Π_{opt} that minimizes the bottleneck value $B_{\text{opt}} = C(\Pi_{\text{opt}})$.

3. Previous work

Each CCP algorithm discussed in this section and Section 4 involves an initial prefix-sum operation on the task-weight array $\mathcal{W}$ to enhance the efficiency of subsequent subchain-weight computations. The prefix-sum operation replaces the ith entry $\mathcal{W}[i]$ with the sum of the first i entries $(\sum_{h=1}^{i} w_h)$ so that computational load W_{ij} of a subchain $\mathcal{T}_{i,j}$ can be efficiently determined as $\mathcal{W}[j] - \mathcal{W}[i-1]$ in $O(1)$-time. In our discussions, $\mathcal{W}$ is used to refer to the prefix-summed $\mathcal{W}$-array, and the $\theta(N)$ cost of this initial prefix-sum operation is considered in the complexity analysis. The presentations focus only on finding the bottleneck value B_{opt}, because a corresponding optimal mapping can be constructed easily by making a $PROBE(B_{\text{opt}})$ call as discussed in Section 3.4.

3.1. Heuristics

The most commonly used partitioning heuristic is based on *recursive bisection* (RB). RB achieves P-way partitioning through $\log P$ bisection levels, where P is a power of 2. At each bisection step in a level, the current chain is divided evenly into two subchains. Although optimal division can be easily achieved at every bisection step, the sequence of optimal bisections may lead to poor load balancing. RB can be efficiently implemented in $O(N + P \log N)$ time by first performing a prefix-sum operation on the workload array $\mathcal{W}$, with complexity $O(N)$, and then making $P-1$ binary searches in the prefix-summed $\mathcal{W}$-array, each with complexity $O(\log N)$.

Miguet and Pierson [24] proposed two other heuristics. The first heuristic ($H1$) computes the separator values such that s_p is the largest index such that $W_{1,s_p} \leqslant pB^*$, where $B^* = W_{\text{tot}}/P$ is the ideal bottleneck value, and $W_{\text{tot}} = \sum_{i=1}^{N} w_i$ denotes sum of all task weights. The second heuristic ($H2$) further refines the separator indices by incrementing each s_p value found in $H1$ if $(W_{1,s_{p+1}} - pB^*) < (pB^* - W_{1,s_p})$. These two heuristics can also be implemented in $O(N + P \log N)$ time by performing $P-1$ binary searches in the prefix-summed $\mathcal{W}$-array. Miguet and Pierson [24] have already proved the upper bounds on the bottleneck values of the partitions found by $H1$ and $H2$ as $B_{H1}, B_{H2} < B^* + w_{\max}$, where $w_{\max} = \max_{1 \leqslant p \leqslant N} \{w_i\}$ denotes the maximum task weight. The following lemma establishes a similar bound for the RB heuristic.

Lemma 3.1. *Let $\Pi_{RB} = \langle s_0, s_1, \ldots, s_P \rangle$ be an RB solution to a CCP problem $(\mathcal{W}, N, P)$. Then $B_{RB} = C(\Pi_{RB})$ satisfies $B_{RB} \leqslant B^* + w_{\max}(P-1)/P$.*

Proof. Consider the first bisection step. There exists a pivot index $1 \leqslant i_1 \leqslant N$ such that both sides weigh less than $W_{\text{tot}}/2$ without the i_1th task, and more than $W_{\text{tot}}/2$ with it. That is,

$$W_{1,i_1-1}, W_{i_1+1,N} \leqslant W_{\text{tot}}/2 \leqslant W_{1,i_1}, W_{i_1,N}.$$

The worst case for RB occurs when $w_{i_1} = w_{\max}$ and $W_{1,i_1-1} = W_{i_1+1,N} = (W_{\text{tot}} - w_{\max})/2$. Without loss of generality, assume that t_{i_1} is assigned to the left part so that $s_{P/2} = i_1$ and $W_{1,s_{P/2}} = W_{\text{tot}}/2 + w_{\max}/2$. In a similar worst-case bisection of $\mathcal{T}_{1,s_{P/2}}$, there exists an

index i_2 such that $w_{i_2} = w_{\max}$ and $W_{1,i_2-1} = W_{i_2+1,s_{P/2}} = (W_{\text{tot}} - w_{\max})/4$, and t_{i_2} is assigned to the left part so that $s_{P/4} = i_2$ and $W_{1,s_{P/4}} = (W_{\text{tot}} - w_{\max})/4 + w_{\max} = W_{\text{tot}}/4 + (3/4)w_{\max}$. For a sequence of $\log P$ such worst-case bisection steps on the left parts, processor $\mathcal{P}_1$ will be the bottleneck processor with load $B_{RB} = W_{1,s_1} = W_{\text{tot}}/P + w_{\max}(P-1)/P$. □

3.2. Dynamic programming

The *overlapping* subproblem space can be defined as $\mathcal{T}_i^p$, for $p = 1, 2, \ldots, P$ and $i = p, p+1, \ldots, N-P+p$, where $\mathcal{T}_i^p$ denotes a p-way CCP of prefix task-subchain $\mathcal{T}_{1,i} = \langle t_1, t_2, \ldots, t_i \rangle$ onto prefix processor-subchain $\mathcal{P}_{1,p} = \langle \mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_p \rangle$. Notice that index i is restricted to $p \leqslant i \leqslant N-P+p$ range because there is no merit in leaving a processor empty. From this subproblem space definition, the *optimal substructure* property of the CCP problem can be shown by considering an optimal mapping $\Pi_i^p = \langle s_0, s_1, \ldots, s_p = i \rangle$ with a bottleneck value B_i^p for the CCP subproblem $\mathcal{T}_i^p$. If the last processor is not the bottleneck processor in Π_i^p, then $\Pi_{s_{p-1}}^{p-1} = \langle s_0, s_1, \ldots, s_{p-1} \rangle$ should be an optimal mapping for the subproblem $\mathcal{T}_{s_{p-1}}^{p-1}$. Hence, recursive definition for the bottleneck value of an optimal mapping is

$$B_i^p = \min_{p-1 \leqslant j < i} \{\max\{B_j^{p-1}, W_{j+1,i}\}\}. \tag{1}$$

In (1), searching for index j corresponds to searching for separator s_{p-1} so that the remaining subchain $\mathcal{T}_{j+1,i}$ is assigned to the last processor $\mathcal{P}_p$ in an optimal mapping Π_i^p of $\mathcal{T}_i^p$. The bottleneck value B_N^p of an optimal mapping can be computed using (1) in a bottom-up fashion starting from $B_i^1 = W_{1,i}$ for $i = 1, 2, \ldots, N$. An initial prefix-sum on the workload array $\mathcal{W}$ enables constant-time computation of subchain weight of the form $W_{j+1,i}$ through $W_{j+1,i} = \mathcal{W}[i] - \mathcal{W}[j]$. Computing B_i^p using (1) takes $O(N-p)$ time for each i and p, and thus the algorithm takes $O((N-P)^2P)$ time since the number of distinct subproblems is equal to $(N-P+1)P$.

Choi and Narahari [6], and Olstad and Manne [30] reduced the complexity of this scheme to $O(NP)$ and $O((N-P)P)$, respectively, by exploiting the following observations that hold for positive task weights. For a fixed p in (1), the minimum index value j_i^p defining B_i^p cannot occur at a value less than the minimum index value j_{i-1}^p defining B_{i-1}^p, i.e., $j_{i-1}^p \leqslant j_i^p \leqslant (i-1)$. Hence, the search for the optimal j_i^p can start from j_{i-1}^p. In (1), B_j^{p-1} for a fixed p is a nondecreasing function of j, and $W_{j+1,i}$ for a fixed i is a decreasing function of j, reducing to 0 at $j = i$. Thus, two cases occur in a semi-closed interval $[j_{i-1}^p, i)$ for j. If $W_{j+1,i} > B_j^{p-1}$ initially, then these two functions intersect in $[j_{i-1}^p, i)$. In this case, the search for j_i^p continues until $W_{j^*+1,i} \leqslant B_j^{p-1}$ and then only j^* and j^*-1 are considered for setting j_i^p with $j_i^p = j^*$ if $B_{j^*}^{p-1} \leqslant W_{j,i}$ and $j_i^p = j^* - 1$ otherwise. Note that this scheme automatically detects $j_i^p = i-1$ if $W_{j+1,i}$ and B_j^{p-1} intersect in the open interval $(i-1, i)$. However if, $W_{j+1,i} \leqslant B_j^{p-1}$ initially, then B_j^{p-1} lies above $W_{j+1,i}$ in the closed interval $[j_{i-1}^p, i]$. In this case, the minimum value occurs at the first value of j, i.e., $j_i^p = j_{i-1}^p$. These improvements lead to an $O((N-P)P)$-time algorithm since computation of all B_i^p values for a fixed p makes $O(N-P)$ references to already computed B_j^{p-1} values. Fig. 1 displays a run-time efficient implementation of this $O((N-P)P)$-time DP algorithm which avoids the explicit min–max operation required in (1). In Fig. 1, B_i^p values are stored in a table whose entries are computed in row-major order.

3.3. Iterative refinement

The algorithm proposed by Manne and Sørevik [23], referred to here as the MS algorithm, finds a sequence of nonoptimal partitions such that there is only one way each partition can be improved. For this purpose, they introduce the *leftist partition* (*LP*). Consider a partition Π such that $\mathcal{P}_p$ is the leftmost processor containing at least two tasks. Π is defined as an LP if increasing the load of any processor $\mathcal{P}_\ell$ that lies to the right of $\mathcal{P}_p$ by augmenting the last task of $\mathcal{P}_{\ell-1}$ to $\mathcal{P}_\ell$ makes $\mathcal{P}_\ell$ a bottleneck processor with a load $\geqslant C(\Pi)$. Let Π be an LP with bottleneck processor $\mathcal{P}_b$ and bottleneck value B. If $\mathcal{P}_b$ contains only one task, then Π is optimal. On the other hand, assume that $\mathcal{P}_b$ contains at least two tasks. The refinement step, which is shown by the inner while–loop in Fig. 2, tries to find a new LP of lower cost by successively removing the first task of $\mathcal{P}_p$ and augmenting it to $\mathcal{P}_{p-1}$ for $p = b, b-1, \ldots,$ until $L_p < B$. Refinement fails when the while–loop proceeds until $p = 1$ with $L_p \geqslant B$. Manne and Sørevik proved that a successful refinement of an LP gives a new LP and the LP is optimal if the refinement fails. They proposed using an initial LP in which the $P-1$ leftmost processors each

```
DP (W, N, P)
  B[1,i] ← W[i]  for  i = 1,2,...,N;
  for p ← 2 to P do
    j ← p − 1;
    for i ← p to N − P + p do
      if W[i] − W[j] > B[p−1, j] then
        repeat j ← j + 1 until W[i] − W[j] ≤ B[p−1, j];
        if W[i] − W[j−1] < B[p−1, j] then
          j ← j − 1;
          B[p, i] ← W[i] − W[j];
        else
          B[p, i] ← B[p−1, j];
      else
        B[p, i] ← B[p−1, j];
  return B_opt ← B[P, N];
```

Fig. 1. $O((N-P)P)$-time dynamic-programming algorithm proposed by Choi and Narahari [6], and Olstad and Manne [30].

has only one task and the last processor contains the remaining tasks. The MS algorithm moves each separator index at most $N-P$ times so that the total number of separator-index moves is $O(P(N-P))$. A max-heap is maintained for the processor loads to find a bottleneck processor at the beginning of each repeat-loop iteration. The cost of each separator-index move is $O(\log P)$ since it necessitates one decrease-key and one increase-key operations. Thus the complexity of the MS algorithm is $O(P(N-P)\log P)$.

3.4. Parametric search

The parametric-search approach relies on repeated probing for a partition Π with a bottleneck value no greater than a given B value. Probe algorithms exploit the *greedy-choice* property for existence and construction of Π. The greedy choice here is to minimize remaining work after loading processor $\mathcal{P}_p$ subject to $L_p \leqslant B$ for $p = 1, \ldots, P-1$ in order. *PROBE*(B) functions given in Fig. 3 exploit this greedy property as follows. *PROBE* finds the largest index s_1 so that $W_{1,s_1} \leqslant B$, and assigns subchain $\mathcal{T}_{1,s_1}$ to processor $\mathcal{P}_1$ with load $L_1 = W_{1,s_1}$. Hence, the first task in the second processor is t_{s_1+1}. *PROBE* then similarly finds the largest index s_2 so that $W_{s_1+1,s_2} \leqslant B$, and assigns the subchain $\mathcal{T}_{s_1+1,s_2}$ to processor $\mathcal{P}_2$. This process continues until either all tasks are assigned or all processors are exhausted. The former and latter cases indicate feasibility and infeasibility of B, respectively.

Fig. 3(a) illustrates the standard probe algorithm. The indices $s_1, s_2, \ldots, s_{P-1}$ are efficiently found through binary search (*BINSRCH*) on the prefix-summed $\mathcal{W}$-array. In this figure, *BINSRCH*$(\mathcal{W}, i, N, Bsum)$ searches $\mathcal{W}$ in the index range $[i, N]$ to compute the index $i \leqslant j \leqslant N$ such that $\mathcal{W}[j] \leqslant Bsum$ and $\mathcal{W}[j+1] > Bsum$. The complexity of the standard probe algorithm is $O(P \log N)$. Han et al. [12] proposed an $O(P \log N/P)$-time probe algorithm (see Fig. 3(b)) exploiting P repeated binary searches on the same $\mathcal{W}$ array with increasing search values. Their algorithm divides the chain into P subchains of equal length. At each probe, a linear search is performed on the weights of the last tasks of these P subchains to find out in which subchain the search value could be, and then binary search is performed on the respective subchain of length N/P. Note that since the probe search values always increase, linear search can be performed incrementally, that is, search continues from the last subchain that was searched to the right with $O(P)$ total cost. This gives a total cost of $O(P \log(N/P))$ for P binary searches thus for the probe function.

```
MS (W, N, P)
  s_p ← p  for  p ← 0, 1, ..., P−1;   s_P ← N;
  L_p ← w_p  for  p ← 1, ..., P−1;   L_P ← Σ_{p=P}^{N} w_p;
  repeat
    b ← { p | L_p is maximum over 1 ≤ p ≤ P};
    B ← L_b;
    if s_{b−1} + 1 = s_b then
      exit the repeat-loop;
    p ← b;
    while L_p ≥ B and p > 1 do
      s_{p−1} ← s_{p−1} + 1;
      L_p ← L_p − w_{s_{p−1}};
      L_{p−1} ← L_{p−1} + w_{s_{p−1}};
      if L_p ≤ B then
        p ← p − 1;
  until L_1 ≥ B;
  return B_opt ← B;
```

Fig. 2. Iterative refinement algorithm proposed by Manne and Sørevik [23].

```
PROBE (B)
  s_0 ← 0;   p ← 1;
  Bsum ← B;
  while p ≤ P and Bsum < W_tot do
    s_p ← BINSRCH (W, s_{p−1}+1, N, Bsum);
    Bsum ← W[s_p] + B;
    p ← p + 1;
  if Bsum ≥ W_tot then
    return TRUE;
  else
    return FALSE;
```

(a)

```
PROBE (B)
  s_0 ← 0;   p ← 1;   step ← N/P;
  Bsum ← B;
  while p ≤ P and Bsum < W_tot do
    while W[step] < Bsum do
      step ← step + N/P;
    s_p ← BINSRCH (W, step−N/P, step, Bsum);
    Bsum ← W[s_p] + B;
    p ← p + 1;
  if Bsum ≥ W_tot then
    return TRUE;
  else
    return FALSE;
```

(b)

Fig. 3. (a) Standard probe algorithm with $O(P \log N)$ complexity, (b) $O(P \log(N/P))$-time probe algorithm proposed by Han et al. [12].

3.4.1. Bisection as an approximation algorithm

Let $f(B)$ be the binary-valued function where $f(B) = 1$ if $PROBE(B)$ is true and $f(B) = 0$ if $PROBE(B)$ is false. Clearly, $f(B)$ is nondecreasing in B, and B_{opt} lies between $LB = B^* = W_{tot}/P$ and $UB = W_{tot}$. These observations are exploited in the *bisection* algorithm leading to an efficient ε-approximate algorithm, where ε is the desired precision. The interval $[W_{tot}/P, W_{tot}]$ is conceptually discretized into $(W_{tot} - W_{tot}/P)/\varepsilon$ bottleneck values, and binary search is used in this range to find the minimum feasible bottleneck value B_{opt}. The bisection algorithm, as illustrated in Fig. 4, performs $O(\log(W_{tot}/\varepsilon))$ *PROBE* calls, and each *PROBE* call costs $O(P\log(N/P))$. Hence, the bisection algorithm runs in $O(N + P\log(N/P)\log(W_{tot}/\varepsilon))$ time, where $O(N)$ cost comes from the initial prefix-sum operation on $\mathcal{W}$. The performance of this algorithm deteriorates when $\log(W_{tot}/\varepsilon)$ is comparable with N.

3.4.2. Nicol's algorithm

Nicol's algorithm [27] exploits the fact that any candidate B value is equal to weight $W_{i,j}$ of a subchain. A naive solution is to generate all subchain weights of the form $W_{i,j}$, sort them, and then use binary search to find the minimum $W_{a,b}$ value for which $PROBE(W_{a,b}) =$ TRUE. Nicol's algorithm efficiently searches for the earliest range $W_{a,b}$ for which $B_{opt} = W_{a,b}$ by considering each processor in order as a candidate bottleneck processor in an optimal mapping. Let Π_{opt} be the optimal mapping constructed by greedy $PROBE(B_{opt})$, and let processor $\mathcal{P}_b$ be the first bottleneck processor with load B_{opt} in $\Pi_{opt} = \langle s_0, s_1, \ldots, s_b, \ldots, s_P \rangle$. Under these assumptions, this greedy construction of Π_{opt} ensures that each processor $\mathcal{P}_p$ preceding $\mathcal{P}_b$ is loaded as much as possible with $L_p < B_{opt}$, for $p = 1, 2, \ldots, b-1$ in Π_{opt}. Here, $PROBE(L_p) =$ FALSE since $L_p < B_{opt}$, and $PROBE(L_p + W_{s_p+1}) =$ TRUE since adding one more task to processor $\mathcal{P}_p$ increases its load to $L_p + w_{s_p+1} > B_{opt}$. Hence, if $b = 1$ then s_1 is equal to the smallest index i_1 such that $PROBE(W_{1,i_1}) =$ TRUE, and $B_{opt} = B_1 = W_{1,s_1}$. However, if $b > 1$, then because of the greedy choice property $\mathcal{P}_1$ should be loaded as much as possible without exceeding $B_{opt} = B_b < B_1$, which implies that $s_1 = i_1 - 1$, and hence $L_1 = W_{1,i_1-1}$. If $b = 2$, then s_2 is equal to the smallest index i_2 such that $PROBE(W_{i_1,i_2}) =$ TRUE, and $B_{opt} = B_2 = W_{i_1,i_2}$. If $b > 2$, then $s_2 = i_2 - 1$. We iterate for $b = 1, 2, \ldots, P-1$, computing i_b as the smallest index for which $PROBE(W_{i_{b-1},i_b}) =$ TRUE and save $B_b = W_{i_{b-1},i_b}$ with $i_P = N$. Finally, the optimal bottleneck value is selected as $B_{opt} = \min_{1 \leqslant b \leqslant P} B_b$.

Fig. 5 illustrates Nicol's algorithm. As seen in this figure, given i_{b-1}, i_b is found by performing a binary search over all subchain weights of the form $W_{i_{b-1},j}$, for $i_{b-1} \leqslant j \leqslant N$, in the bth iteration of the *for–loop*. Hence, Nicol's algorithm performs $O(\log N)$ *PROBE* calls to find i_b at iteration b, and each PROBE call costs $O(P\log(N/P))$. Thus, the cost of computing an individual B_b value is $O(P\log N\log(N/P))$. Since $P-1$ such B_b values are computed, the overall complexity of Nicol's algorithm is $O(N + P^2\log N\log(N/P))$, where $O(N)$ cost comes from the initial prefix-sum operation on $\mathcal{W}$.

Two possible implementations of Nicol's algorithm are presented in Fig. 5. Fig. 5(a) illustrates a straightforward implementation, whereas Fig. 5(b) illustrates a careful implementation, which maintains and uses the information from previous probes to answer without calling the *PROBE* function. As seen in Fig. 5(b), this information is efficiently maintained as an undetermined bottleneck-value range (LB, UB), which is dynamically refined after each probe. Any bottleneck value encountered outside the current range is immediately accepted or rejected. Although this simple scheme does not improve the asymptotic complexity of the algorithm, it drastically reduces the number of probes, as discussed in Section 6.

```
ε-BISECT (W, N, P, ε)
  LB ← B* ← W_tot / P;
  UB ← W_tot;
  repeat
    B ← (UB + LB) / 2;
    if PROBE (B) then
      UB ← B;
    else
      LB ← B;
  until UB ≤ LB + ε;
  return B_opt ← UB;
```

Fig. 4. Bisection as an ε-approximation algorithm.

4. Proposed CCP algorithms

In this section, we present proposed methods. First, we describe how to bound the separator indices for an optimal solution to reduce the search space. Then, we show how this technique can be used to improve the performance of the dynamic programming algorithm. We continue with our discussion on improving the MS algorithm, and propose a novel refinement algorithm, which we call the bidding algorithm. Finally, we discuss parametric search methods, proposing improvements for the bisection and Nicol's algorithms.

4.1. Restricting the search space

Our proposed CCP algorithms exploit lower and upper bounds on the optimal bottleneck value to restrict

```
NICOL- (W, N, P)
  i_0 ← 1;
  for b ← 1 to P − 1 do
    ilow ← i_{b−1};   ihigh ← N;
    while ilow < ihigh do
      imid ← (ilow + ihigh) / 2;
      B ← W[imid] − W[i_{b−1} − 1];
      if PROBE (B) then
        ihigh ← imid;
      else
        ilow ← imid + 1;
    i_b ← ihigh;
    B_b ← W[i_b] − W[i_{b−1} − 1];
  B_P ← W[N] − W[i_{P−1} − 1];
  return B_opt ← min_{1≤b≤P}{B_p};
```

(a)

```
NICOL (W, N, P)
  i_0 ← 1;   LB ← B* ← W_tot / P ;   UB ← W_tot;
  for b ← 1 to P − 1 do
    ilow ← i_{b−1};   ihigh ← N;
    while ilow < ihigh do
      imid ← (ilow + ihigh) / 2;
      B ← W[imid] − W[i_{b−1} − 1];
      if LB ≤ B < UB then
        if PROBE (B) then
          ihigh ← imid;
          UB ← B;
        else
          ilow ← imid + 1;
          LB ← B;
      elseif B ≥ UB
        ihigh ← imid;
      else
        ilow ← imid + 1;
    i_b ← ihigh;
    B_b ← W[i_b] − W[i_{b−1} − 1];
  B_P ← W[N] − W[i_{P−1} − 1];
  return B_opt ← min_{1≤b≤P}{B_p};
```

(b)

Fig. 5. Nicol's [27] algorithm: (a) straightforward implementation, (b) careful implementation with dynamic bottleneck-value bounding.

the search space for separator values as a preprocessing step. Natural lower and upper bounds for the optimal bottleneck value B_{opt} of a given CCP problem instance $(\mathcal{W}, N, P)$ are $LB = \max\{B^*, w_{\max}\}$ and $UB = B^* + w_{\max}$, respectively, where $B^* = W_{\text{tot}}/P$. Since $w_{\max} < B^*$ in coarse grain parallelization of most real-world applications, our presentation will be for $w_{\max} < B^* = LB$ even though all results are valid when B^* is replaced with $\max\{B^*, w_{\max}\}$. The following lemma describes how to use these natural bounds on B_{opt} to restrict the search space for the separator values.

Lemma 4.1. *For a given CCP problem instance $(\mathcal{W}, N, P)$, if B_{f} is a feasible bottleneck value in the range $[B^*, B^* + w_{\max}]$, then there exists a partition $\Pi = \langle s_0, s_1, \ldots, s_P \rangle$ of cost $C(\Pi) \leqslant B_{\text{f}}$ with $SL_p \leqslant s_p \leqslant SH_p$, for $1 \leqslant p < P$, where SL_p and SH_p are, respectively, the smallest and largest indices such that*

$$W_{1,SL_p} \geqslant p(B^* - w_{\max}(P-p)/P) \quad \text{and}$$
$$W_{1,SH_p} \leqslant p(B^* + w_{\max}(P-p)/P).$$

Proof. Let $B_{\text{f}} = B^* + w$, where $0 \leqslant w < w_{\max}$. Partition Π can be constructed by $PROBE(B)$, which loads the first p processors as much as possible subject to $L_q \leqslant B_{\text{f}}$, for $q = 1, 2, \ldots, p$. In the worst case, $w_{s_p+1} = w_{\max}$ for each of the first p processors. Thus, we have $W_{1,s_p} \geqslant f(w) = p(B^* + w - w_{\max})$ for $p = 1, 2, \ldots, P-1$. However, it should be possible to divide the remaining subchain $\mathcal{T}_{s_p+1,N}$ into $P-p$ parts without exceeding B_{f}, i.e., $W_{s_p+1,N} \leqslant (P-p)(B^* + w)$. Thus, we also have $W_{1,s_p} \geqslant g(w) = W_{\text{tot}} - (P-p)(B^* + w)$. Note that $f(w)$ is an increasing function of w, whereas $g(w)$ is a decreasing function of w. The minimum of $\max\{f(w), g(w)\}$ is at the intersection of $f(w)$ and $g(w)$ so that $W_{1,s_p} \geqslant p(B^* - w_{\max}(P-p)/P)$.

To prove the upper bounds, we can start with $W_{1,s_p} \leqslant f(w) = p(B^* + w)$, which holds when $L_q = B^* + w$ for $q = 1, \ldots, p$. The condition $W_{s_p+1,N} \geqslant (P-p)(B^* + w - w_{\max})$, however, ensures feasibility of $B_{\text{f}} = B^* + w$, since $PROBE(B)$ can always load each of the remaining $(P-p)$ processors with $B^* + w - w_{\max}$. Thus, we also have $W_{1,s_p} \leqslant g(w) = W_{\text{tot}} - (P-p)(B^* + w - w_{\max})$. Here, $f(w)$ is an increasing function of w, whereas $g(w)$ is a decreasing function of w, which yields $W_{1,s_p} \leqslant p(B^* + w_{\max}(P-p)/P)$. □

Corollary 4.2. *The separator range weights are $\Delta W_p = \sum_{i=SL_p}^{SH_p} w_i = W_{1,SH_p} - W_{1,SL_p} = 2w_{\max}p(P-p)/P$ with a maximum value $Pw_{\max}/2$ at $p = P/2$.*

Applying this corollary requires finding $w_{\max}$, which entails an overhead equivalent to that of the prefix-sum operation, and hence should be avoided. In this work, we adopt a practical scheme to construct the bounds on separator indices. We run the *RB* heuristic to find a

hopefully good bottleneck value B_{RB}, and use B_{RB} as an upper bound for bottleneck values, i.e., $UB = B_{RB}$. Then we run *LR-PROBE*(B_{RB}) and *RL-PROBE*(B_{RB}) to construct two mappings $\Pi^1 = \langle h_0^1, h_1^1, \ldots, h_P^1 \rangle$ and $\Pi^2 = \langle \ell_0^2, \ell_1^2, \ldots, \ell_P^2 \rangle$ with $C(\Pi^1), C(\Pi^2) \leqslant B_{RB}$. Here, *LR-PROBE* denotes the left-to-right probe given in Fig. 3, whereas *RL-PROBE* denotes a right-to-left probe function, which can be considered as the dual of the *LR-PROBE*. *RL-PROBE* exploits the greedy-choice property from right to left. That is, *RL-PROBE* assigns subchains from the right end towards the left end of the task chain to processors in the order $\mathcal{P}_P, \mathcal{P}_{P-1}, \ldots, \mathcal{P}_1$. From these two mappings, lower and upper bound values for s_p separator indices are constructed as $SL_p = \ell_p^2$ and $SH_p = h_p^1$, respectively. These bounds are further refined by running *LR-PROBE*(B^*) and *RL-PROBE*(B^*) to construct two mappings $\Pi^3 = \langle \ell_0^3, \ell_1^3, \ldots, \ell_P^3 \rangle$ and $\Pi^4 = \langle h_0^4, h_1^4, \ldots, h_P^4 \rangle$, and then defining $SL_p = \max\{SL_p, \ell_p^3\}$ and $SH_p = \min\{SH_p, h_p^4\}$ for $1 \leqslant p < P$. Lemmas 4.3 and 4.4 prove correctness of these bounds.

Lemma 4.3. *For a given CCP problem instance* $(\mathcal{W}, N, P)$ *and a feasible bottleneck value* B_f, *let* $\Pi^1 = \langle h_0^1, h_1^1, \ldots, h_P^1 \rangle$ *and* $\Pi^2 = \langle \ell_0^2, \ell_1^2, \ldots, \ell_P^2 \rangle$ *be the partitions constructed by LR-PROBE*(B_f) *and RL-PROBE*(B_f), *respectively. Then any partition* $\Pi = \langle s_0, s_1, \ldots, s_P \rangle$ *of cost* $C(\Pi) = B \leqslant B_f$ *satisfies* $\ell_p^2 \leqslant s_p \leqslant h_p^1$.

Proof. By the property of *LR-PROBE*(B_f), h_p^1 is the largest index such that $\mathcal{T}_{1,h_p^1}$ can be partitioned into p parts without exceeding B_f. If $s_p > h_p^1$, then the bottleneck value will exceed B_f and thus B. By the property of *RL-PROBE*(B_f), ℓ_p^2 is the smallest index where $\mathcal{T}_{\ell_p^2,N}$ can be partitioned into $P-p$ parts without exceeding B_f. If $s_p < \ell_p^2$, then the bottleneck value will exceed B_f and thus B. □

Lemma 4.4. *For a given CCP problem instance* $(\mathcal{W}, N, P)$, *let* $\Pi^3 = \langle \ell_0^3, \ell_1^3, \ldots, \ell_P^3 \rangle$ *and* $\Pi^4 = \langle h_0^4, h_1^4, \ldots, h_P^4 \rangle$ *be the partitions constructed by LR-PROBE*(B^*) *and RL-PROBE*(B^*), *respectively. Then for any feasible bottleneck value* B_f, *there exists a partition* $\Pi = \langle s_0, s_1, \ldots, s_P \rangle$ *of cost* $C(\Pi) \leqslant B_f$ *that satisfies* $\ell_p^3 \leqslant s_p \leqslant h_p^4$.

Proof. Consider the partition $\Pi = \langle s_0, s_1, \ldots, s_P \rangle$ constructed by *LR-PROBE*(B_f). It is clear that this partition already satisfies the lower bounds, i.e., $s_p \geqslant \ell_p^3$. Assume $s_p > h_p^4$, then partition Π' obtained by moving s_p back to h_p^4 also yields a partition with cost $C(\Pi') \leqslant B_f$, since $\mathcal{T}_{h_p^4+1,N}$ can be partitioned into $P-p$ parts without exceeding B^*. □

The difference between Lemmas 4.3 and 4.4 is that the former ensures the existence of all partitions with cost $\leqslant B_f$ within the given separator-index ranges, whereas the latter ensures only the existence of at least one such partition within the given ranges. The following corollary combines the results of these two lemmas.

Corollary 4.5. *For a given CCP problem instance* $(\mathcal{W}, N, P)$ *and a feasible bottleneck value* B_f, *let* $\Pi^1 = \langle h_0^1, h_1^1, \ldots, h_P^1 \rangle$, $\Pi^2 = \langle \ell_0^2, \ell_1^2, \ldots, \ell_P^2 \rangle$, $\Pi^3 = \langle \ell_0^3, \ell_1^3, \ldots, \ell_P^3 \rangle$, *and* $\Pi^4 = \langle h_0^4, h_1^4, \ldots, h_P^4 \rangle$ *be the partitions constructed by LR-PROBE*(B_f), *RL-PROBE*(B_f), *LR-PROBE*(B^*), *and RL-PROBE*(B^*), *respectively. Then for any feasible bottleneck value* B *in the range* $[B^*, B_f]$, *there exists a partition* $\Pi = \langle s_0, s_1, \ldots, s_P \rangle$ *of cost* $C(\Pi) \leqslant B$ *with* $SL_p \leqslant s_p \leqslant SH_p$, *for* $1 \leqslant p < P$, *where* $SL_p = \max\{\ell_p^2, \ell_p^3\}$ *and* $SH_p = \min\{h_p^1, h_p^4\}$.

Corollary 4.6. *The separator range weights become* $\Delta W_p = 2\min\{p, (P-p)\} w_{\max}$ *in the worst case, with a maximum value* $P w_{\max}$ *at* $p = P/2$.

Lemma 4.1 and Corollary 4.5 infer the following theorem since $B^* \leqslant B_{\text{opt}} \leqslant B^* + w_{\max}$.

Theorem 4.7. *For a given CCP problem instance* $(\mathcal{W}, N, P)$, *and* SL_p *and* SH_p *index bounds constructed according to Lemma* 4.1 *or Corollary* 4.5, *there exists an optimal partition* $\Pi_{\text{opt}} = \langle s_0, s_1, \ldots, s_P \rangle$ *with* $SL_p \leqslant s_p \leqslant SH_p$, *for* $1 \leqslant p < P$.

Comparison of separator range weights in Lemma 4.1 and Corollary 4.5 shows that separator range weights produced by the practical scheme described in Corollary 4.5 may be worse than those of Lemma 4.1 by a factor of two. This is only the worst-case behavior however, and the practical scheme normally finds much better bounds, since order in the chain usually prevents the worst-case behavior and $B_{RB} < B^* + w_{\max}$. Experimental results in Section 6 justify this expectation.

4.1.1. Complexity analysis models

Corollaries 4.2 and 4.6 give bounds on the weights of the separator-index ranges. However, we need bounds on the sizes of these separator-index ranges for computational complexity analysis of the proposed CCP algorithms. Here, the size $\Delta S_p = SH_p - SL_p + 1$ denotes the number of tasks within the pth range $[SL_p, SH_p]$. Miguet and Pierson [24] propose the model $w_i = \theta(w_{\text{avg}})$ for $i = 1, 2, \ldots, N$ to prove that their $H1$ and $H2$ heuristics allocate $\theta(N/P)$ tasks to each processor, where $w_{\text{avg}} = W_{\text{tot}}/N$ is the average task weight. This assumption means that the weight of each task is not too far away from the average task weight. Using Corollaries 4.2 and 4.5, this model induces $\Delta S_p = O(P\, w_{\max}/w_{\text{avg}})$. Moreover, this model can be exploited to induce the optimistic bound $\Delta S_p = O(P)$. However,

we find their model too restrictive, since the minimum and maximum task weights can deviate substantially from w_{avg}. Here, we establish a looser and more realistic model on task weights so that for any subchain $\mathcal{T}_{i,j}$ with weight $W_{i,j}$ sufficiently larger than w_{max}, the average task weight within subchain $\mathcal{T}_{i,j}$ satisfies $\Omega(w_{avg})$. That is, $\Delta_{i,j} = j - i + 1 = O(W_{i,j}/w_{avg})$. This model, referred to here as model $\mathcal{M}$, directly induces $\Delta S_p = O(P w_{max}/w_{avg})$, since $\Delta W_p \leqslant \Delta W_{P/2} = P w_{max}/2$ for $p = 1, 2, \ldots, P-1$.

4.2. Dynamic-programming algorithm with static separator-index bounding

The proposed DP algorithm, referred to here as the DP+ algorithm, exploits bounds on the separator indices for an efficient solution. Fig. 6 illustrates the proposed DP+ algorithm, where input parameters SL and SH denote the index bound arrays, each of size P, computed according to Corollary 4.5 with $B_f = B_{RB}$. Note that $SL_P = SH_P = N$, since only $B[P, N]$ needs to be computed in the last row. As seen in Fig. 6, only B_j^p values for $j = SL_p, SL_p + 1, \ldots, SH_p$ are computed at each row p by exploiting Corollary 4.5, which ensures existence of an optimal partition $\Pi_{opt} = \langle s_0, s_1, \ldots, s_P \rangle$ with $SL_p \leqslant s_p \leqslant SH_p$. Thus, these B_j^p values will suffice for correct computation of B_i^{p+1} values for $i = SL_{p+1}, SL_{p+1} + 1, \ldots, SH_{p+1}$ at the next row $p + 1$.

As seen in Fig. 6, explicit range checking is avoided in this algorithm for utmost efficiency. However, the j-index may proceed beyond SH_p to $SH_p + 1$ within the *repeat–until-loop* while computing B_i^{p+1} with $SL_{p+1} \leqslant i \leqslant SH_{p+1}$ in two cases. In both cases, functions $W_{j+1,i}$ and B_j^p intersect in the open interval $(SH_p, SH_p + 1)$ so that $B^p_{SH_p} < W_{SH_p+1,i}$ and $B^p_{SH_p+1} \geqslant W_{SH_p+2,i}$. In the first case, $i = SH_p + 1$ so that $W_{j+1,i}$ and B_j^p intersect in $(i-1, i)$, which implies that $B_i^{p+1} = W_{i-1,i}$, with $j_i^{p+1} = SL_p$, since $W_{i-1,i} < B_i^p$, as mentioned in Section 3.2. In the second case, $i > SH_p + 1$, for which Corollary 4.5 guarantees that $B_i^{p+1} = W_{SL_p+1,i} \leqslant B^p_{SH_p+1}$, and thus we can safely select $j_i^{p+1} = SL_p$. Note that $W_{SL_p+1,i} = B^p_{SH_p+1}$ may correspond to a case leading to another optimal partition with $j_i^{p+1} = s_{p+1} = SH_p + 1$. As seen in Fig. 6, both cases are efficiently resolved simply by storing ∞ to $B^p_{SH_p+1}$ as a *sentinel*. Hence, in such cases, the condition $W_{SH_p+1,i} < B^p_{SH_p+1} = \infty$ in the *if–then* statement following the *repeat–until-loop* is always true so that the j-index automatically moves back to SH_p. The scheme of computing $B^p_{SH_p+1}$ for each row p, which seems to be a natural solution, does not work since correct computation of $B^{p+1}_{SH_{p+1}+1}$ may necessitate more than one B_j^p value beyond the SH_p index bound.

A nice feature of the DP approach is that it can be used to generate all optimal partitions by maintaining a $P \times N$ matrix to store the minimum j_i^p index values defining the B_i^p values at the expense of increased execution time and increased asymptotic space requirement. Recall that index bounds SL and SH computed according to Corollary 4.5 restrict the search space for at least one optimal solution. The index bounds can be computed according to Lemma 4.4 for this purpose, since the search space restricted by Lemma 4.4 includes all optimal solutions.

The running time of the proposed DP+ algorithm is $O(N + P \log N) + \sum_{p=1}^{P} \theta(\Delta S_p)$. Here, $O(N)$ cost comes from the initial prefix-sum operation on the $\mathcal{W}$ array, and $O(P \log N)$ cost comes from the running time of the *RB* heuristic and computing the separator-index bounds SL and SH according to Corollary 4.5. Under model $\mathcal{M}$, $\Delta S_p = O(P w_{max}/w_{avg})$, and hence the complexity is $O(N + P \log N + P^2 w_{max}/w_{avg})$. The algorithm becomes linear in N when the separator-index ranges do not overlap, which is guaranteed by the condition $w_{max} = O(2W_{tot}/P^2)$.

```
DP+ (W, N, P, SL, SH)
  B[1, i] ← W[i]  for  i = SL_1, SL_1 + 1, ..., SH_1;
  B[1, SH_1+1] ← ∞;
  for p ← 2 to P do
    j ← SL_{p-1};
    for i ← SL_p to SH_p do
      if W[i] − W[j] > B[p−1, j] then
        repeat j ← j + 1 until W[i] − W[j] ≤ B[p−1, j];
        if W[i] − W[j−1] < B[p−1, j] then
          j ← j − 1;
          B[p, i] ← W[i] − W[j];
        else
          B[p, i] ← B[p−1, j];
      else
        B[p, i] ← B[p − 1, j];
    B[1, SH_p+1] ← ∞;
  return B_opt ← B[P, N];
```

Fig. 6. Dynamic-programming algorithm with static separator-index bounding.

4.3. Iterative refinement algorithms

In this work, we improve the MS algorithm and propose a novel CCP algorithm, namely the *bidding* algorithm, which is run-time efficient for small-to-medium number of processors. The main difference between the MS and the bidding algorithms is as follows: the MS algorithm moves along a series of feasible bottleneck values, whereas the bidding algorithm moves along a sequence of infeasible bottleneck values so that the first feasible bottleneck value becomes the optimal value.

4.3.1. Improving the MS algorithm

The performance of the MS algorithm strongly depends on the initial partition. The initial partition proposed by Manne and Sørevik [23] satisfies the leftist

partition constraint, but it leads to very poor run-time performance. Here, we propose using the partition generated by $PROBE(B^*)$ as an initial partition. This partition is also a leftist partition, since moving any separator to the left will not decrease the load of the bottleneck processor. This simple observation leads to significant improvement in run-time performance of the algorithm. Also, using a heap as a priority queue does not give better run-time performance than using a running maximum despite its superior asymptotic complexity. In our implementation, we use a running maximum.

4.3.2. Bidding algorithm

This algorithm increases the bottleneck value gradually, starting from the ideal bottleneck value B^*, until it finds a feasible partition, which is also optimal. Consider a partition $\Pi_t = \langle s_0, s_1, \ldots, s_P \rangle$ constructed by $PROBE(B_t)$ for an infeasible B_t. After detecting the infeasibility of this B_t value, the point is to determine the next larger bottleneck value B to be investigated. Clearly, the separator indices of the partitions to be constructed by future $PROBE(B)$ calls with $B > B_t$ will never be to the left of the respective separator indices of Π_t. Moreover, at least one of the separators should move right for feasibility, since the load of the last processor determines infeasibility of the current B_t value (i.e., $L_P > B_t$). To avoid missing the smallest feasible bottleneck value, the next larger B value is selected as the minimum of processor loads that will be obtained by moving the end-index of every processor to the right by one position. That is, the next larger B value is equal to $\min\{\min_{1 \leq p < P} \{L_p + w_{s_p+1}\}, L_P\}$. Here, we call the $L_p + w_{s_p+1}$ value the *bid* of processor $\mathcal{P}_p$, which refers to the load of $\mathcal{P}_p$ if the first task t_{s_p+1} of the next processor is augmented to $\mathcal{P}_p$. The bid of the last processor $\mathcal{P}_P$ is equal to the load of the remaining tasks. If the smallest bid B comes from processor $\mathcal{P}_b$, probing with new B is performed only for the remaining processors $\langle \mathcal{P}_b, \mathcal{P}_{b+1}, \ldots, \mathcal{P}_P \rangle$ in the suffix $\mathcal{W}_{s_{b-1}+1:N}$ of the $\mathcal{W}$ array.

The bidding algorithm is presented in Fig. 7. The innermost *while–loop* implements a linear probing scheme, such that the new positions of the separators are determined by moving them to the right, one by one. This linear probing scheme is selected because new positions of separators are likely to be in a close neighborhood of previous ones. Note that binary search is used only for setting the separator indices for the first time. After the separator index s_p is set for processor $\mathcal{P}_p$ during linear probing, the *repeat–until-loop* terminates if it is not possible to partition the remaining subchain $\mathcal{T}_{s_p+1,N}$ into $P-p$ processors without exceeding the current B value, i.e., $rbid = L_r/(P-p) > B$, where L_r denotes the weight of the remaining subchain. In this case, the next larger B value is determined by considering the best bid among the first p processors and $rbid$.

```
BIDDING (W, N, P)
  s_p ← 0  for  p ← 0, 1, ..., P−1;   s_P ← N;
  BIDS[0].B ← L_r ← W_tot;
  B ← B*;   k ← 0;
  while L_r > B do
    repeat p ← p + 1;
      if s_p = 0 then
        s_p ← BINSRCH (W, s_{p−1} + 1, N, W[s_{p−1}] + B);
        L_p ← W[s_p] − W[s_{p−1}];
      else
        while L_p + w_{s_p+1} ≤ B do
          s_p ← s_p + 1;
          L_p ← L_p + w_{s_p};
      mybid ← L_p + w_{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⟩;
      L_r ← W_tot − W[s_p];   rbid ← L_r / (P − p);
    until rbid > B or p = P − 1;
    if rbid < BIDS[p].B then
      B ← rbid;
    else
      ⟨B, k⟩ ←BIDS[p].⟨B, q⟩;
    p ← p − 1;
  return  B_opt ← B;
```

Fig. 7. Bidding algorithm.

As seen in Fig. 7, we maintain a prefix-minimum array *BIDS* for computing the next larger B value. Here, *BIDS* is an array of records of length P, where $BIDS[p].B$ and $BIDS[p].b$ store the best bid value of the first p processors and the index of the defining processor, respectively. $BIDS[0]$ helps the correctness of the running prefix-minimum operation.

The complexity of the bidding algorithm for integer task weights under model $\mathcal{M}$ is $O(N + P \log N + P\, w_{\max} + P^2(w_{\max}/w_{\text{avg}}))$. Here, $O(N)$ cost comes from the initial prefix-sum operation on the $\mathcal{W}$ array, and $O(P \log N)$ cost comes from initial settings of separators through binary search. The B value is increased at most $B_{\text{opt}} - B^* < w_{\max}$ times, and each time the next B value can be computed in $O(P)$ time, which induces the cost $O(P\, w_{\max})$. The total area scanned by the separators is at most $O(P^2(w_{\max}/w_{\text{avg}}))$. For noninteger task weights, complexity can reach $O(P \log N + P^3(w_{\max}/w_{\text{avg}}))$ in the worst case, which occurs when only one separator index moves to the right by one position at each B value. We should note here that using a min-heap for finding the next B value enables terminating a repeat-loop iteration as soon as a separator-index does not move. The trade-off in this scheme is the $O(\log P)$ cost incurred at each separator-index move due to respective key-update operations on the heap. We implemented this scheme as well, but observed increased execution times.

4.4. Parametric search algorithms

In this work, we apply theoretical findings given in Section 4.1 for an improved probe algorithm. The improved algorithm, which we call the restricted probe (*RPROBE*), exploits bounds computed according to Corollary 4.5 (with $B_f = B_{RB}$) to restrict the search space for s_p separator values during binary searches on the $\mathcal{W}$ array. That is, $BINSRCH(\mathcal{W}, SL_p, SH_p, Bsum)$ in *RPROBE* searches $\mathcal{W}$ in the index range $[SL_p, SH_p]$ to find the index $SL_p \leqslant s_p \leqslant SH_p$ such that $\mathcal{W}[s_p] \leqslant Bsum$ and $\mathcal{W}[s_p+1] > Bsum$ via binary search. This scheme and Corollaries 4.2 and 4.6 reduce the complexity of an individual probe to $\sum_{p=1}^{P} \theta(\log \Delta_p) = O(P \log P + P \log(w_{\max}/w_{\text{avg}}))$. Note that this complexity reduces to $O(P \log P)$ for sufficiently large P where $P = \Omega(w_{\max}/w_{\text{avg}})$. Figs. 8–10 illustrate *RPROBE* algorithms tailored for the respective parametric-search algorithms.

4.4.1. Approximate bisection algorithm with dynamic separator-index bounding

The proposed bisection algorithm, illustrated in Fig. 8, searches the space of bottleneck values in range $[B^*, B_{RB}]$ as opposed to $[B^*, W_{\text{tot}}]$. In this algorithm, if $PROBE(B_t) = \text{TRUE}$, then the search space is restricted to $B \leqslant B_t$ values, and if $PROBE(B_t) = \text{FALSE}$, then the search space is restricted to $B > B_t$ values. In this work, we exploit this simple observation to propose and develop a dynamic probing scheme that increases the efficiency of successive *PROBE* calls by modifying separator index-bounds depending on success and failure of the probes. Let $\Pi_t = \langle t_0, t_1, \ldots, t_P \rangle$ be the partition constructed by $PROBE(B_t)$. Any future $PROBE(B)$ call with $B \leqslant B_t$ will set the s_p indices with $s_p \leqslant t_p$. Thus, the search space for s_p can be restricted to those indices $\leqslant t_p$. Similarly, any future $PROBE(B)$ call with $B \geqslant B_t$ will set s_p indices $\geqslant t_p$. Thus, the search space for s_p can be restricted to those indices $\geqslant t_p$.

As illustrated in Fig. 8, dynamic update of separator-index bounds can be performed in $\theta(P)$ time by a *for–loop* over *SL* or *SH* arrays, depending on failure or success, respectively, of *RPROBE* (B_t). In our implementation, however, this update is efficiently achieved in $O(1)$ time through the pointer assignment $SL \leftarrow \Pi$ or $SH \leftarrow \Pi$ depending on failure or success of *RPROBE* (B_t).

```
ε-BISECT+ (W, N, SL, SH, P)
  LB ← B*;
  UB ← B_RB;
  repeat
    B_t ← (UB + LB) / 2;
    if RPROBE (B_t) then
      for p ← 1 to P − 1 do SH_p ← s_p;
      UB ← B_t;
    else
      for p ← 1 to P − 1 do SL_p ← s_p;
      LB ← B_t;
  until UB ≤ LB + ε;
  return B_opt ← UB;
```

```
RPROBE (B)
  Bsum ← B;
  for p ← 1 to P−1 do
    s_p ← BINSRCH (W, SL_p, SH_p, Bsum);
    Bsum ← W[s_p] + B;
  if Bsum ≥ W_tot then
    return TRUE;
  else
    return FALSE;
```

Fig. 8. Bisection as an ε-approximation algorithm with dynamic separator-index bounding.

```
EXACT-BISECT (W, N, SL, SH, P)
  LB ← B*;   UB ← B_RD;
  repeat
    B_t ← (UB + LB) / 2;
    if RPROBE (B_t) then
      for p ← 1 to P − 1 do SH_p ← s_p;
      UB ← max_{1≤p≤P}{L_p};
    else
      for p ← 1 to P − 1 do SH_p ← s_p;
      LB ← min{min_{1≤p<P}{L_p + w_{s_p+1}}, L_P};
  until UB = LB;
  return B_opt ← UB;
```

```
RPROBE (B)
  Bsum ← B;
  for p ← 1 to P−1 do
    s_p ← BINSRCH (W, SL_p, SH_p, Bsum);
    Bsum ← W[s_p] + B;
    L_p ← W[s_p] − W[s_{p−1}];
  L_P ← W[N] − W[s_{P−1}];
  if L_P ≤ B then
    return TRUE;
  else
    return FALSE;
```

Fig. 9. Exact bisection algorithm with dynamic separator-index bounding.

```
NICOL+ (W, N, SL, SH, P)
  i_0 ← 1;   LB ← W_tot/P;   UB ← B_RB;
  for b ← 1 to P − 1 do
    ilow ← SL_b;   ihigh ← SH_b;
    while ilow < ihigh do
      imid ← (ilow + ihigh) / 2;
      B ← W[imid] − W[i_{b−1} − 1];
      if LB ≤ B < UB then
        if RPROBE (b, imid, B) then
          for p ← b + 1 to P − 1 do  SH_p ← s_p;
          UB ← B;
          ihigh ← imid;
        else
          for p ← b + 1 to P − 1 do  SL_p ← s_p;
          LB ← B;
          ilow ← imid + 1;
      elseif B ≥ UB
        ihigh ← imid;
      else
        ilow ← imid + 1;
    i_b ← ihigh;
    B_b ← W[i_b] − W[i_{b−1} − 1];
  B_P ← W[N] − W[i_{P−1} − 1];
  return min_{1≤b≤P}{B_b};
```

```
RPROBE (b, imid, B)
  Bsum ← W[imid] + B;
  for p ← b + 1 to P − 1 do
    s_p ← BINSRCH (W, SL_p, SH_p, Bsum);
    Bsum ← W[s_p] + B;
  if Bsum ≥ W_tot then
    return TRUE;
  else
    return FALSE;
```

Fig. 10. Nicol's algorithm with dynamic separator-index bounding.

Similar to the ε-BISECT algorithm, the proposed ε-BISECT+ algorithm is also an ε-approximation algorithm for general workload arrays. However, both the ε-BISECT and ε-BISECT+ algorithms become exact algorithms for integer-valued workload arrays by setting $\varepsilon = 1$. As shown in Lemma 3.1, $B_{RB} < B^* + w_{\max}$. Hence, for integer-valued workload arrays the maximum number of probe calls in the ε-BISECT+ algorithm is $\log w_{\max}$, and thus the overall complexity is $O(N + P\log N + \log(w_{\max})(P\log P + P\log(w_{\max}/w_{\text{avg}})))$ under model $\mathcal{M}$. Here, $O(N)$ cost comes from the initial prefix-sum operation on $\mathcal{W}$ and $O(P\log N)$ cost comes from the running time of the *RB* heuristic and computing the separator-index bounds *SL* and *SH* according to Corollary 4.5.

4.4.2. Bisection as an exact algorithm

In this section, we will enhance the bisection algorithm to be an exact algorithm for general workload arrays by clever updating of lower and upper bounds after each probe. The idea is, after each probe moving upper and lower bounds on the value of an optimal solution to a realizable bottleneck value (total weight of a subchain of $\mathcal{W}$). This reduces the search space to a finite set of realizable bottleneck values, as opposed to an infinite space of bottleneck values defined by a range $[LB, UB]$. Each bisection step is designed to eliminate at least one candidate value, and thus the algorithm terminates in finite number of steps to find the optimal bottleneck value.

After a probe *RPROBE* (B_t), the current upper bound value *UB* is modified if *RPROBE* (B_t) succeeds. Note that *RPROBE* (B_t) not only determines the feasibility of B_t, but also constructs a partition Π with $cost(\Pi_t) \leqslant B_t$. Instead of reducing the upper bound *UB* to B_t, we can further reduce *UB* to the bottleneck value $B = cost(\Pi_t) \leqslant B_t$ of the partition Π_t constructed by *RPROBE* (B_t). Similarly, the current lower bound *LB* is modified when *RPROBE* (B_t) fails. In this case, instead of increasing the bound *LB* to B_t, we can exploit the partition Π_t constructed by *RPROBE* (B_t) to increase *LB* further to the smallest realizable bottleneck value B greater than B_t. Our bidding algorithm already describes how to compute

$$B = \min\left\{\min_{1 \leqslant p < P}\{L_p + w_{s_p+1}\}, L_P\right\},$$

where L_p denotes the load of processor $\mathcal{P}_p$ in Π_t. Fig. 9 presents the pseudocode of our algorithm.

Each bisection step divides the set of candidate realizable bottleneck values into two sets, and eliminates one of them. The initial set can have a size between 1 and N^2. Assuming the size of the eliminated set can be anything between 1 and N^2, the expected complexity of

the algorithm is

$$T(N) = \frac{1}{N^2}\sum_{i=1}^{N^2} T(i) + O(P\log P + P\log(w_{\max}/w_{\text{avg}})),$$

which has the solution $O(P\log P\log N + P\log N\log(w_{\max}/w_{\text{avg}}))$. Here, $O(P\log P + P\log(w_{\max}/w_{\text{avg}}))$ is the cost of a probe operation, and $\log N$ is the expected number of probes. Thus, the overall complexity becomes $O(N + P\log P\log N + P\log N\log(w_{\max}/w_{\text{avg}}))$, where the $O(N)$ cost comes from the initial prefix-sum operation on the $\mathcal{W}$ array.

4.4.3. Improving Nicol's algorithm as a divide-and-conquer algorithm

Theoretical findings of previous sections can be exploited at different levels to improve performance of Nicol's algorithm. A trivial improvement is to use the proposed restricted probe function instead of the conventional one. The careful implementation scheme given in Fig. 5(b) enables use of dynamic separator-index bounding. Here, we exploit the bisection idea to design an efficient divide-and-conquer approach based on Nicol's algorithm.

Consider the sequence of probes of the form $PROBE(W_{1,j})$ performed by Nicol's algorithm for processor $\mathcal{P}_1$ to find the smallest index $j = i_1$ such that $PROBE(W_{1,j}) = \text{TRUE}$. Starting from a naive bottleneck-value range $(LB_0 = 0, UB_0 = W_{\text{tot}})$, success and failure of these probes can narrow this range to (LB_1, UB_1). That is, each $PROBE(W_{1,j}) = \text{TRUE}$ decreases the upper bound to $W_{1,j}$ and each $PROBE(W_{1,j}) = \text{FALSE}$ increases the lower bound to $W_{1,j}$. Clearly, we will have $(LB_1 = W_{1,i_1-1}, UB_1 = W_{1,i_1})$ at the end of this search for processor $\mathcal{P}_1$. Now consider the sequence of probes of the form $PROBE(W_{i_1,j})$ performed for processor $\mathcal{P}_2$ to find the smallest index $j = i_2$ such that $PROBE(W_{i_1,j}) = \text{TRUE}$. Our key observation is that the partition $\Pi_t = \langle 0, t_1, t_2, \ldots, t_{P-1}, N\rangle$ to be constructed by any $PROBE(B_t)$ with $LB_1 < B_t = W_{i_1,j} < UB_1$ will satisfy $t_1 = i_1 - 1$, since $W_{1,i_1-1} < B_t < W_{1,i_1}$. Hence, probes with $LB_1 < B_t < UB_1$ for processor $\mathcal{P}_2$ can be restricted to be performed in $\mathcal{W}_{i_1:N}$, where $\mathcal{W}_{i_1:N}$ denotes the $(N - i_1 + 1)$th suffix of the prefix-summed $\mathcal{W}$ array. This simple yet effective scheme leads to an efficient divide-and-conquer algorithm as follows. Let $\mathcal{T}_i^{P-p}$ denote the CCP subproblem of $(P-p)$-way partitioning of the $(N-i+1)$th suffix $\mathcal{T}_{i,N} = \langle t_i, t_{i+1}, \ldots, t_N\rangle$ of the task chain $\mathcal{T}$ onto the $(P-p)$th suffix $\mathcal{P}_{p,P} = \langle \mathcal{P}_{p+1}, \mathcal{P}_{p+2}, \ldots, \mathcal{P}_P\rangle$ of the processor chain $\mathcal{P}$. Once the index i_1 for processor $\mathcal{P}_1$ is computed, the optimal bottleneck value B_{opt} can be defined by either W_{1,i_1} or the bottleneck value of an optimal $(P-1)$-way partitioning of the suffix subchain $\mathcal{T}_{i_1,N}$. That is, $B_{\text{opt}} = \min\{B_1 = W_{1,i_1}, C(\Pi_{i_1}^{P-1})\}$. Proceeding in this way, once the indices $\langle i_1, i_2, \ldots, i_p\rangle$ for the first p processors $\langle \mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_p\rangle$ are determined, then $B_{\text{opt}} = \min\{\min_{1\leqslant b\leqslant p}\{B_b = W_{i_{b-1},i_b}\},\ C(\Pi_{i_p}^{P-p})\}$.

This approach is presented in Fig. 10. At the bth iteration of the outer *for–loop*, given i_{b-1}, i_b is found in the inner *while–loop* by conducting probes on $\mathcal{W}_{i_{b-1}:N}$ to compute $B_b = W_{i_{b-1},\ i_b}$. The dynamic bounds on the separator indices are exploited in two distinct ways according to Theorem 4.7. First, the restricted probe function RPROBE is used for probing. Second, the search spaces for the bottleneck values of the processors are restricted. That is, given i_{b-1}, the binary search for i_b over all subchain weights of the form $W_{i_{b-1}+1,j}$ for $i_{b-1} < j \leqslant N$ is restricted to $W_{i_{b-1}+1,j}$ values for $SL_b \leqslant j \leqslant SH_b$.

Under model $\mathcal{M}$, the complexity of this algorithm is $O(N + P\log N + w_{\max}(P\log P + P\log(w_{\max}/w_{\text{avg}})))$ for integer task weights, because the number of probes cannot exceed $w_{\max}$, since there are at most $w_{\max}$ distinct bound values in the range $[B^*, B_{RB}]$. For noninteger task weights, the complexity can be given as $O(N + P\log N + w_{\max}(P\log P)^2 + w_{\max}P^2\log P\log(w_{\max}/w_{\text{avg}}))$, since the algorithm makes $O(w_{\max}P\log P)$ probes. Here, $O(N)$ cost comes from the initial prefix-sum operation on $\mathcal{W}$, and $O(P\log N)$ cost comes from the running time of the *RB* heuristic and computing the separator-index bounds *SL* and *SH* according to Corollary 4.5.

5. Load balancing applications

5.1. Parallel sparse matrix–vector multiplication

Sparse matrix–vector multiplication (SpMxV) is one of the most important kernels in scientific computing. Parallelization of repeated SpMxV computations requires partitioning and distribution of the matrix. Two possible 1D sparse-matrix partitioning schemes are *rowwise striping* (RS) and *columnwise striping* (CS). Consider parallelization of SpMxV operations of the form $y = Ax$ in an iterative solver, where A is an $N \times N$ sparse matrix and y and x are N-vectors. In RS, processor $\mathcal{P}_p$ owns the pth row stripe A_p^r of A and is responsible for computing $y_p = A_p^r x$, where y_p is the pth stripe of vector y. In CS, processor $\mathcal{P}_p$ owns the pth column stripe A_p^c of A and is responsible for computing $y^p = A_p^c x$, where $y = \sum_{p=1}^{P} y^p$. All vectors used in the solver are divided conformably with row (column) partitioning in the RS (CS) scheme, to avoid unnecessary communication during linear vector operations. RS and CS schemes require communication before or after local SpMxV computations, thus they can also be considered as *pre-* and *post*-communication schemes, respectively. In RS, each task $t_i \in \mathcal{T}$ corresponds to the

atomic task of computing the inner-product of row i of matrix A with column vector x. In CS, each task $t_i \in \mathcal{T}$ corresponds to the atomic task of computing the sparse $DAXPY$ operation $y = y + x_i a_{*i}$, where a_{*i} is the ith column of A. Each nonzero entry in a row and column of A incurs a multiply-and-add operation, thus the computational load w_i of task t_i is the number of nonzero entries in row i (column i) in the RS (CS) scheme. This defines how load balancing problem for rowwise and columnwise partitioning of a sparse matrix with a given ordering can be modeled as a CCP problem.

In RS (CS), by allowing only row (column) reordering, load balancing problem can be described as the number partitioning problem, which is NP-Hard [9]. By allowing both row and column reordering, the problem of minimizing communication overhead while maintaining load balance can be described as graph and hypergraph partitioning problems [5,14], which are NP-Hard [10,21] as well. However, possibly high preprocessing overhead involved in these models may not be justified in some applications. If the partitioner is to be used as part of a run-time library for a parallelizing compiler for a data-parallel programming language [33,34], row and column reordering lead to high memory requirement due to the irregular mapping table and extra level of indirection in locating distributed data during each multiply-and-add operation [35]. Furthermore, in some applications, the natural row and column ordering of the sparse matrix may already be likely to induce small communication overhead (e.g., banded matrices).

The proposed CCP algorithms are surprisingly fast so that the initial prefix-sum operation dominates their execution times in sparse-matrix partitioning. In this work, we exploit the standard *compressed row storage* (CRS) and *compressed column storage* (CCS) data structures for sparse matrices to avoid the prefix-sum operation. In CRS, an array $DATA$ of length NZ stores nonzeros of matrix A, in row-major order, where $NZ = W_{\text{tot}}$ denotes the total number of nonzeros in A. An index array COL of length NZ stores the column indices of respective nonzeros in array $DATA$. Another index array ROW of length $N+1$ stores the starting indices of respective rows in the other two arrays. Hence, any subchain weight $W_{i,j}$ can be efficiently computed using $W_{i,j} = ROW[j+1] - ROW[i]$ in $O(1)$ time without any preprocessing overhead. CCS is similar to CRS with rows and columns interchanged, thus $W_{i,j}$ is computed using $W_{i,j} = COL[j+1] - COL[i]$.

5.1.1. A better load balancing model for iterative solvers

The load balancing problem for parallel iterative solvers has usually been stated considering only the SpMxV computations. However, linear vector operations (i.e., DAXPY and inner-product computations) involved in iterative solvers may have a considerable effect on parallel performance with increasing sparsity of the coefficient matrix. Here, we consider incorporating vector operations into the load-balancing model as much as possible.

For the sake of discussion, we will investigate this problem for a coarse-grain formulation [2,3,32] of the *conjugate-gradient* (CG) algorithm. Each iteration of CG involves, in order of computational dependency, one SpMxV, two inner-products, and three DAXPY computations. DAXPY computations do not involve communication, whereas inner-product computations necessitate a post global *reduction* operation [19] on results of the two local inner-products. In rowwise striping, pre-communication operations needed for SpMxV and the global reduction operation constitute pre- and post-synchronization points, respectively, for the aggregate of one local SpMxV and two local inner-product computations. In columnwise striping, the global reduction operation in the current iteration and post-communication operations needed for SpMxV in the next iteration constitute the pre- and post-synchronization points, respectively, for an aggregate of three local DAXPY and one local SpMxV computations. Thus, columnwise striping may be favored for a wider coverage of load balancing. Each vector entry incurs a multiply-and-add operation during each linear vector operation. Hence, DAXPY computations can easily be incorporated into the load-balancing model for the CS scheme by adding a cost of three to the computational weight w_i of atomic task t_i representing column i of matrix A. The initial prefix-sum operation can still be avoided by computing a subchain weight $W_{i,j}$ as $W_{i,j} = COL[j+1] - COL[i] + 3(j-i+1)$ in constant time. Note that two local inner-product computations still remain uncovered in this balancing model.

5.2. Sort-first parallel direct volume rendering

Direct volume rendering (DVR) methods are widely used in rendering unstructured volumetric grids for visualization and interpretation of computer simulations performed for investigating physical phenomena in various fields of science and engineering. A DVR application contains two interacting domains: object space and image space. Object space is a 3D domain containing the volume data to be visualized. Image space (screen) is a 2D domain containing pixels from which rays are shot into the 3D object domain to determine the color values of the respective pixels. Based on these domains, there are basically two approaches for data parallel DVR: image- and object-space parallelism, which are also called as sort-first and sort-last parallelism according to the taxonomy based on the point of data redistribution in the rendering pipeline [25]. Pixels or pixel blocks constitute the atomic tasks in sort-first

parallelism, whereas volume elements (primitives) constitute the atomic tasks in sort-last parallelism.

In sort-first parallel DVR, screen space is decomposed into regions and each region is assigned to a separate processor for local rendering. The primitives, whose projection areas intersect more than one region, are replicated. Sort-first parallelism has an advantage of processors generating complete images for their local screen subregion, but it faces load-balancing problems in the DVR of unstructured grids due to uneven on-screen primitive distribution.

Image-space decomposition schemes for sort-first parallel DVR can be classified as *static* and *adaptive* [20]. Static decomposition is a view-independent scheme, and the load-balancing problem is solved implicitly by scattered assignment of pixels or pixel blocks. Load-balancing performance of this scheme depends on the assumption that neighbor pixels are likely to have equal workload since they are likely to have similar views of the volume. As the scattered assignment scheme assigns adjacent pixels or pixel blocks to different processors, it disturbs image-space coherency and increases the amount of primitive replication. Adaptive decomposition is a view-dependent scheme, and the load-balancing problem is solved explicitly by using the primitive distribution on the screen.

In adaptive image-space decomposition, the number of primitives with bounding-box approximation is taken to be the workload of a screen region. Primitives constituting the volume are tallied to a 2D coarse mesh superimposed on the screen. Some primitives may intersect multiple cells. The inverse-area heuristic [26] is used to decrease the amount of error due to counting such primitives multiple times. Each primitive increments the weight of each cell it intersects by a value inversely proportional to the number of cells the primitive intersects. In this heuristic, if we assume that there are no shared primitives among screen regions, then the sum of the weights of individual mesh cells forming a region gives the number of primitives in that region. Shared primitives may still cause some errors, but such errors are much less than counting such primitives multiple times while adding mesh-cell weights.

Minimizing the perimeter of the resulting regions in the decomposition is expected to minimize the communication overhead due to the shared primitives. 1D decomposition, i.e., horizontal or vertical striping of the screen, suffers from unscalability. A Hilbert space-filling curve [31] is widely used for 2D decomposition of 2D nonuniform workloads. In this scheme [20], the 2D coarse mesh superimposed on the screen is traversed according to the Hilbert curve to map the 2D coarse mesh to a 1D chain of mesh cells. The load-balancing problem in this decomposition scheme then reduces to the CCP problem. Using a Hilbert curve as the space-filling curve is an implicit effort towards reducing the total perimeter, since the Hilbert curve avoids jumps during the traversal of the 2D coarse mesh. Note that the 1D workload array used for partitioning is a real-valued array because of the inverse-area heuristic used for computing weights of the coarse-mesh cells.

6. Experimental results

All algorithms were implemented in the *C* programming language. All experiments were carried out on a workstation equipped with a 133 MHz *PowerPC* and 64 MB of memory. We have experimented with $P = 16$-, 32-, 64-, 128-, and 256-way partitioning of each test data.

Table 2 presents the test problems we have used in our experiments. In this table, w_{avg}, w_{min}, and w_{max} columns display the average, minimum, and maximum task weights. The dataset for the sparse-matrix decomposition comes from matrices of linear programming test problems from the *Netlib* suite [11] and the *IOWA Optimization Center* [22]. The sparsity patterns of these matrices are obtained by multiplying the respective rectangular constraint matrices with their transposes. Note that the number of tasks in the sparse-matrix (SpM) dataset also refers to the number of rows and columns of the respective matrix. The dataset for image-space decomposition comes from sort-first parallel DVR of curvilinear grids *blunt-fin* and *post* representing the results of computational fluid dynamic simulations, which are commonly used in volume rendering. The raw grids consist of hexahedral elements and are converted into an unstructured tetrahedral data format by dividing each hexahedron into five tetrahedrons. Triangular faces of tetrahedrons constitute the primitives mentioned in Section 5.2. Three distinct 1D workload arrays are constructed both for *blunt-fin* and *post* as described in Section 5.2 for coarse meshes of resolutions 256×256, 512×512, and 1024×1024 superimposed on a screen of resolution 1024×1024. The properties of these six workload arrays are displayed in Table 2. The number of tasks is much less than coarse-mesh resolution because of the zero-weight tasks, which can be compressed to retain only nonzero-weight tasks.

The following abbreviations are used for the CCP algorithms: $H1$ and $H2$ refer to Miguet and Pierson's [24] heuristics, and *RB* refers to the recursive-bisection heuristic, all described in Section 3.1. DP refers to the $O((N-P)P)$-time dynamic-programming algorithm in Fig. 1. MS refers to Manne and Sørevik's iterative-refinement algorithm in Fig. 2. εBS refers to the ε-approximate bisection algorithm in Fig. 4. NC- and NC refer to the straightforward and careful implementations of Nicol's parametric-search algorithm in

Table 2
Properties of the test set

Sparse-matrix dataset							Direct volume rendering (DVR) dataset					
Name	No. of tasks N	Workload: No. of nonzeros				ex. time	Name	No. of tasks N	Workload			
		Total W_{tot}	Per row/col (task)			SpMxV			Total	Per task		
			w_{avg}	w_{min}	w_{max}	(ms)			W_{tot}	w_{avg}	w_{min}	w_{max}
NL	7039	105 089	14.93	1	361	22.55	blunt256	17 303	303 K	17.54	0.020	1590.97
cre-d	8926	372 266	41.71	1	845	72.20	blunt512	93 231	314 K	3.36	0.004	661.50
CQ9	9278	221 590	23.88	1	702	45.90	blunt1024	372 824	352 K	0.94	0.001	411.04
ken-11	14 694	82 454	5.61	2	243	19.65	post256	19 653	495 K	25.19	0.077	3245.50
mod2	34 774	604 910	17.40	1	941	124.05	post512	134 950	569 K	4.22	0.015	1092.00
world	34 506	582 064	16.87	1	972	119.45	post1024	539 994	802 K	1.49	0.004	1546.78

Figs. 5(a) and (b), respectively. Abbreviations ending with "+" are used to represent our improved versions of these algorithms. That is, DP+, εBS+, NC+ refer to our algorithms given in Figs. 6, 8, and 10, respectively, and MS+ refers to the algorithm described in Section 4.3.1. BID refers to our bidding algorithm given in Fig. 7. EBS refers to our exact bisection algorithm given in Fig. 9. Both εBS and εBS+ algorithms are effectively used as exact algorithms for the SpM dataset with $\varepsilon = 1$ for integer-valued workload arrays in the SpM dataset. However, these two algorithms were not tested on the DVR dataset, since they remain approximation algorithms due to the real-valued task weights in DVR.

Table 3 compares load-balancing quality of heuristics and exact algorithms. In this table, percent load-imbalance values are computed as $100 \times (B - B^*)/B^*$, where B denotes the bottleneck value of the respective partition and $B^* = W_{tot}/P$ denotes ideal bottleneck value. OPT values refer to load-imbalance values of optimal partitions produced by exact algorithms. Table 3 clearly shows that considerably better partitions are obtained in both datasets by using exact algorithms instead of heuristics. The quality gap between exact algorithms and heuristics increases with increasing P. The only exceptions to these observations are 256-way partitioning of *blunt256* and *post256*, for which the *RB* heuristic finds optimal solutions.

Table 4 compares the performances of the static separator-index bounding schemes discussed in Section 4.1. The values displayed in this table are the sums of the sizes of the separator-index ranges normalized with respect to N, i.e., $\sum_{p=1}^{P-1} \Delta S_p/N$, where $\Delta S_p = SH_p - SL_p + 1$. For each CCP instance, $(N-P)P$ represents the size of the search space for the separator indices and the total number of table entries referenced and computed by the DP algorithm. The columns labeled $L1$, $L3$, and $C5$ display total range sizes obtained according to Lemma 4.1, Lemma 4.3 and Corollary 4.5, respectively. As seen in Table 4, proposed practical scheme $C5$ achieves substantially better separator-index bounds than $L1$, despite its inferior worst-case behavior (see Corollaries 4.2 and 4.6). Comparison of columns $L3$ and $C5$ shows the substantial benefit of performing left-to-right and right-to-left probes with B^* according to Lemma 4.4. Comparison of $(N-P)P$ and the $C5$ column reveals the effectiveness of the proposed separator-index bounding scheme in restricting the search space for separator indices in both SpM and DVR datasets. As expected, the performance of index bounding decreases with increasing P because of decreasing N/P values. The numbers for the DVR dataset are better than those for the SpM dataset because of the larger N/P values. In Table 4, values less than 1 indicate that the index bounding scheme achieves nonoverlapping index ranges. As seen in the table, scheme $C5$ reduces the total separator-index range sizes below N for each CCP instance with $P \leqslant 64$ in both the SpM and DVR datasets. These results show that the proposed DP+ algorithm becomes a linear-time algorithm in practice.

The efficiency of the parametric-search algorithms depends on two factors: the number of probes and the cost of each probe. The dynamic index bounding schemes proposed for parametric-search algorithms reduce the cost of an individual probe. Table 5 illustrates how the proposed parametric-search algorithms reduce the number of probes. To compare performances of EBS and εBS+ on the DVR dataset, we forced the εBS+ algorithm to find optimal partitions by running it with $\varepsilon = w_{min}$ and then improving the resulting partition using the BID algorithm. Column εBS + &BID refers to this scheme, and values after "+" denote the number of additional bottleneck values tested by the BID algorithm. As seen in Table 5, exactly one final bottleneck-value test was needed by BID to reach an optimal partition in each CCP instance.

In Table 5, comparison of the NC- and NC columns shows that dynamic bottleneck-value bounding drastically decreases the number of probes in Nicol's algorithm. Comparison of the εBS and εBS+ columns in the SpM dataset shows that using B_{RB} instead of W_{tot}

Table 3
Percent load imbalance values

Sparse-matrix dataset						DVR dataset					
CCP instance		Heuristics			OPT	CCP instance		Heuristics			OPT
Name	P	$H1$	$H2$	RB		Name	P	$H1$	$H2$	RB	
NL	16	2.60	2.44	1.20	0.35	blunt256	16	4.60	1.20	0.49	0.34
	32	5.02	5.75	3.44	0.95		32	6.93	2.61	1.94	1.12
	64	8.95	9.01	5.60	2.37		64	14.52	9.44	9.44	2.31
	128	33.13	27.16	22.78	4.99		128	38.25	24.39	16.67	4.82
	256	69.55	69.55	60.78	14.25		256	96.72	37.03	34.21	34.21
cre-d	16	2.27	0.98	0.53	0.45	blunt512	16	0.95	0.98	0.98	0.16
	32	4.19	4.42	3.74	1.03		32	1.38	1.38	1.18	0.33
	64	7.12	4.92	4.34	1.73		64	2.87	2.69	1.66	0.53
	128	25.57	18.73	16.70	2.88		128	5.62	8.45	4.62	0.97
	256	37.54	26.81	35.20	10.95		256	14.18	14.33	9.34	2.28
CQ9	16	1.85	1.85	0.58	0.58	blunt1024	16	0.94	0.57	0.95	0.10
	32	5.65	2.88	2.24	0.90		32	1.89	1.21	0.97	0.14
	64	13.25	11.49	7.64	1.43		64	4.99	2.16	1.44	0.26
	128	33.96	32.22	22.34	3.51		128	10.06	4.25	2.47	0.57
	256	58.62	58.62	58.62	14.72		256	19.65	9.68	9.68	0.98
ken-11	16	3.74	2.01	0.98	0.21	post256	16	1.10	1.43	0.76	0.56
	32	3.74	4.67	3.74	1.18		32	3.23	3.98	3.23	1.11
	64	13.17	13.17	13.17	1.29		64	17.28	11.04	10.90	3.10
	128	13.17	16.89	13.17	6.80		128	45.35	29.09	29.09	8.29
	256	50.99	50.99	50.99	7.11		256	67.86	67.86	67.86	67.86
mod2	16	0.06	0.06	0.06	0.03	post512	16	0.49	1.25	0.33	0.33
	32	0.19	0.19	0.19	0.07		32	0.94	1.61	0.90	0.58
	64	7.42	2.72	2.18	0.18		64	4.85	5.33	4.85	0.94
	128	16.15	6.29	2.46	0.41		128	18.03	14.55	10.15	1.72
	256	19.47	19.47	18.92	1.23		256	30.03	25.29	25.29	3.73
world	16	0.27	0.18	0.09	0.04	post1024	16	0.70	0.70	0.53	0.20
	32	1.03	0.37	0.27	0.08		32	1.49	1.49	1.41	0.54
	64	4.73	4.73	4.73	0.28		64	2.85	2.85	1.49	0.91
	128	6.37	6.37	6.37	0.76		128	13.15	11.79	9.10	1.11
	256	27.99	27.41	27.41	1.11		256	40.50	13.37	14.19	2.54
Averages over P											
	16	1.76	1.15	0.74	0.36		16	1.44	1.01	1.04	0.28
	32	3.99	3.39	2.88	0.76		32	2.64	2.05	1.61	0.64
	64	9.01	6.78	5.69	1.43		64	7.89	5.58	4.96	1.31
	128	19.97	17.66	14.09	3.36		128	21.74	15.42	12.02	2.91
	256	43.89	38.91	36.04	9.18		256	44.82	27.93	26.76	18.60

for the upper bound on the bottleneck values considerably reduces the number of probes. Comparison of the εBS+ and EBS columns reveals two different behaviors on SpM and DVR datasets. The discretized dynamic bottleneck-value bounding used in EBS produces only minor improvement on the SpM dataset because of the already discrete nature of integer-valued workload arrays. However, the effect of discretized dynamic bottleneck-value bounding is significant on the real-valued workload arrays of the DVR dataset.

Tables 6 and 7 display the execution times of CCP algorithms on the SpM and DVR datasets, respectively. In Table 6, the execution times are given as percents of single SpMxV times. For the DVR dataset, actual execution times (in msecs) are split into prefix-sum times and partitioning times. In Tables 6 and 7, execution times of existing algorithms and their improved versions are listed in the same order under each respective classification so that improvements can be seen easily. In both tables, BID is listed separately since it is a new algorithm. Results of both εBS and εBS+ are listed in Table 6, since they are used as exact algorithms on SpM dataset with $\varepsilon = 1$. Since neither εBS nor εBS+ can be used as an exact algorithm on the DVR dataset, EBS, as

Table 4
Sizes of separator-index ranges normalized with respect to N

Sparse-matrix dataset						DVR dataset					
CCP instance		$(N-P)P$	$L1$	$L3$	$C5$	CCP instance		$(N-P)P$	$L1$	$L3$	$C5$
Name	P					Name	P				
NL	16	15.97	0.32	0.13	0.036	blunt256	16	15.99	0.38	0.07	0.003
	32	31.86	1.23	0.58	0.198		32	31.94	1.22	0.30	0.144
	64	63.43	4.97	2.24	0.964		64	63.77	5.10	3.81	0.791
	128	125.69	19.81	19.60	4.305		128	127.06	21.66	12.88	2.784
	256	246.73	77.96	82.83	22.942		256	252.23	101.68	50.48	10.098
cre-d	16	15.97	0.16	0.04	0.019	blunt512	16	16.00	0.09	0.17	0.006
	32	31.89	0.76	0.90	0.137		32	31.99	0.35	0.26	0.043
	64	63.55	2.89	1.85	0.610		64	63.96	1.64	0.69	0.144
	128	126.18	11.59	15.12	2.833		128	127.83	6.65	4.59	0.571
	256	248.69	46.66	57.54	16.510		256	255.30	28.05	16.71	2.262
CQ9	16	15.97	0.30	0.03	0.023	blunt1024	16	16.00	0.05	0.09	0.010
	32	31.89	1.21	0.34	0.187		32	32.00	0.18	0.18	0.018
	64	63.57	4.84	2.96	0.737		64	63.99	0.77	0.74	0.068
	128	126.25	19.20	18.66	3.121		128	127.96	3.43	2.53	0.300
	256	248.96	74.84	86.80	23.432		256	255.82	13.92	20.36	1.648
ken-11	16	15.98	0.28	0.11	0.024	post256	16	15.99	0.35	0.04	0.021
	32	31.93	1.10	1.13	0.262		32	31.95	1.50	0.52	0.162
	64	63.73	4.42	7.34	0.655		64	63.79	6.12	4.26	0.932
	128	126.89	17.60	14.56	4.865		128	127.17	26.48	23.68	4.279
	256	251.56	69.18	85.44	13.076		256	252.68	124.76	90.48	16.485
mod2	16	15.99	0.16	0.00	0.002	post512	16	16.00	0.13	0.02	0.003
	32	31.97	0.55	0.04	0.013		32	31.99	0.56	0.14	0.063
	64	63.88	2.14	1.22	0.109		64	63.97	2.33	2.41	0.413
	128	127.53	8.62	2.59	0.411		128	127.88	9.33	10.15	1.714
	256	254.12	34.49	39.02	2.938		256	255.52	37.08	46.28	6.515
world	16	15.99	0.15	0.01	0.004	post1024	16	16.00	0.17	0.07	0.020
	32	31.97	0.59	0.06	0.020		32	32.00	0.54	0.27	0.087
	64	63.88	2.30	2.81	0.158		64	63.99	2.19	0.61	0.210
	128	127.53	9.23	7.19	0.850		128	127.97	8.77	9.48	0.918
	256	254.11	36.97	53.15	2.635		256	255.88	34.97	27.28	4.479
Averages over K											
	16	15.97	0.21	0.06	0.018		16	16.00	0.20	0.08	0.011
	32	31.88	0.86	0.68	0.136		32	31.98	0.73	0.28	0.096
	64	63.52	3.43	2.63	0.516		64	63.91	3.03	2.09	0.426
	128	126.07	13.68	12.74	2.731		128	127.65	12.72	10.55	1.428
	256	248.26	54.28	57.55	14.089		256	254.57	56.74	41.93	6.915

$L1$, $L3$ and $C5$ denote separator-index ranges obtained according to results of Lemmas 4.1, 4.3 and Corollary 4.5.

an exact algorithm for general workload arrays, is listed separately in Table 7.

As seen in Tables 6 and 7, the *RB* heuristic is faster than both $H1$ and $H2$ for both SpM and DVR datasets. As also seen in Table 3, *RB* finds better partitions than both $H1$ and $H2$ for both datasets. These results reveal *RB*'s superiority to $H1$ and $H2$.

In Tables 6 and 7, relative performance comparison of existing exact CCP algorithms shows that NC is two orders of magnitude faster than both DP and MS for both SpM and DVR datasets, and εBS is considerably faster than NC for the SpM dataset. These results show that among existing algorithms the parametric-search approach leads to faster algorithms than both the dynamic-programming and the iterative-improvement approaches.

Tables 6 and 7 show that our improved algorithms are significantly faster than the respective existing algorithms. In the dynamic-programming approach, DP+ is two-to-three orders of magnitude faster than DP so that

Table 5
Number of probes performed by parametric search algorithms

Sparse-matrix dataset								DVR dataset						
CCP instance		Parametric search algorithms						CCP instance		Parametric search algorithms				
Name	P	NC−	NC	NC+	εBS	εBS+	EBS	Name	P	NC−	*NC*	NC+	εBS+&*BID*	EBS
NL	16	177	21	7	17	6	7	blunt256	16	202	18	6	16 + 1	9
	32	352	19	7	17	7	7		32	407	19	7	19 + 1	10
	64	720	27	5	16	7	6		64	835	18	10	21 + 1	12
	128	1450	40	14	17	8	8		128	1686	25	15	22 + 1	13
	256	2824	51	14	16	8	8		256	3556	203	62	23 + 1	11
cre-d	16	190	20	6	19	7	5	blunt512	16	245	20	8	17 + 1	9
	32	393	25	11	19	9	8		32	502	24	13	18 + 1	10
	64	790	24	10	19	8	6		64	1003	23	11	18 + 1	12
	128	1579	27	10	19	9	9		128	2023	26	12	20 + 1	13
	256	3183	37	13	18	9	9		256	4051	23	12	21 + 1	13
CQ9	16	182	19	3	18	6	5	blunt1024	16	271	21	10	16 + 1	13
	32	373	22	8	18	8	7		32	557	24	12	18 + 1	12
	64	740	29	12	18	8	8		64	1127	21	11	18 + 1	12
	128	1492	40	12	17	8	8		128	2276	28	13	19 + 1	13
	256	2971	50	14	18	9	9		256	4564	27	16	21 + 1	15
ken-11	16	185	21	6	17	6	6	post256	16	194	22	9	18 + 1	7
	32	364	20	6	17	6	6		32	409	19	8	20 + 1	12
	64	721	48	9	17	7	7		64	823	17	11	22 + 1	12
	128	1402	61	8	16	7	7		128	1653	19	13	22 + 1	14
	256	2783	96	7	17	8	8		256	3345	191	102	23 + 1	13
mod2	16	210	20	6	20	5	4	post512	16	235	24	9	16 + 1	10
	32	432	26	8	19	5	5		32	485	23	12	18 + 1	11
	64	867	24	6	19	8	8		64	975	22	13	20 + 1	14
	128	1727	38	7	20	7	7		128	1975	25	17	21 + 1	14
	256	3444	47	9	20	9	9		256	3947	29	20	22 + 1	15
world	16	211	22	6	19	5	4	post1024	16	261	27	10	17 + 1	13
	32	424	26	5	19	6	6		32	538	23	13	19 + 1	14
	64	865	30	12	19	9	9		64	1090	22	12	17 + 1	14
	128	1730	44	10	19	8	8		128	2201	25	15	21 + 1	16
	256	3441	44	11	19	10	10		256	4428	28	16	22 + 1	16
Averages over P														
	16	193	20.5	5.7	18.5	5.8	5.2		16	235	22.0	8.7	16.7+1	10.2
	32	390	23.0	7.5	18.3	6.8	6.5		32	483	22.0	10.8	18.7+1	11.5
	64	784	30.3	9.0	18.0	7.8	7.3		64	976	20.5	11.3	19.3+1	12.7
	128	1564	41.7	10.2	18.0	7.8	7.8		128	1969	24.7	14.2	20.8+1	13.8
	256	3108	54.2	11.3	18.0	8.8	8.8		256	3982	83.5	38.0	22.0+1	13.8

DP+ is competitive with the parametric-search algorithms. For the SpM dataset, DP+ is 630 times faster than DP on average in 16-way partitioning, and this ratio decreases to 378, 189, 106, and 56 with increasing number of processors. For the DVR dataset, if the initial prefix-sum is not included, DP+ is 1277, 578, 332, 159, and 71 times faster than DP for $P = 16, 32, 64, 128,$ and 256, respectively, on average. This decrease is expected because the effectiveness of separator-index bounding decreases with increasing P. These experimental findings agree with the variation in the effectiveness of separator-index bounding values seen in Table 4. In the iterative refinement approach, MS+ is also one-to-three orders of magnitude faster than MS, where this drastic improvement simply depends on the scheme used for finding an initial leftist partition.

As Tables 6 and 7 reveal, significant improvement ratios are also obtained for the parametric search algorithms. On average, NC+ is 4.2, 3.5, 3.1, 3.7, and 3.7 times faster than NC for $P = 16$, 32, 64, 128, and 256, respectively, for the SpM dataset. For the DVR dataset, if the initial prefix-sum time is not included, NC+ is 4.2, 3.2, 2.6, 2.5, and 2.7 times faster than NC for $P = 16, 32, 64, 128,$ and 256, respectively. For the

Table 6
Partitioning times for sparse-matrix dataset as percents of SpMxV times

CCP instance		Heuristics			Exact algorithms								
Name	P	$H1$	$H2$	RB	Existing				Proposed				
					DP	MS	εBS	NC	DP+	MS+	εBS+	NC+	BID
NL	16	0.09	0.09	0.09	93	119	0.93	1.20	0.40	0.44	0.40	0.40	0.22
	32	0.18	0.18	0.13	177	194	1.77	2.17	1.73	1.73	0.80	0.80	0.44
	64	0.35	0.35	0.31	367	307	3.15	5.85	5.81	4.12	1.86	1.77	1.82
	128	0.89	0.84	0.71	748	485	6.30	17.12	19.87	26.92	4.26	7.32	4.21
	256	1.51	1.55	1.37	1461	757	11.09	40.31	91.80	96.41	9.80	15.70	21.06
cre-d	16	0.03	0.01	0.01	33	36	0.33	0.37	0.12	0.06	0.10	0.11	0.03
	32	0.04	0.06	0.06	71	61	0.65	0.93	0.47	1.20	0.29	0.33	0.10
	64	0.12	0.12	0.08	147	98	1.22	1.69	1.66	1.83	0.61	0.71	0.21
	128	0.28	0.29	0.14	283	150	2.41	3.59	5.47	10.94	1.68	1.68	0.61
	256	0.53	0.54	0.25	571	221	4.20	10.22	27.05	26.02	3.30	4.46	1.20
CQ9	16	0.04	0.04	0.04	59	73	0.48	0.59	0.20	0.13	0.15	0.13	0.17
	32	0.09	0.09	0.07	127	120	0.94	1.31	0.94	0.72	0.41	0.41	0.28
	64	0.20	0.17	0.15	248	195	1.85	3.18	3.01	4.47	1.00	1.44	0.68
	128	0.44	0.44	0.31	485	303	3.18	8.52	9.65	18.82	2.44	3.05	2.51
	256	0.92	0.92	0.63	982	469	6.51	20.33	69.56	72.2	5.32	7.45	14.92
ken-11	16	0.10	0.10	0.10	238	344	1.27	1.88	0.56	0.61	0.46	0.41	0.25
	32	0.20	0.20	0.15	501	522	2.29	3.10	3.82	4.78	1.22	1.17	1.63
	64	0.46	0.46	0.36	993	778	4.48	13.08	9.41	24.78	3.10	3.46	2.29
	128	1.17	1.17	0.71	1876	1139	9.21	39.59	50.13	50.03	6.41	6.62	17.4
	256	2.14	2.14	1.42	3827	1706	17.76	119.13	129.01	241.48	14.91	14.50	29.11
mod2	16	0.02	0.02	0.01	92	119	0.27	0.34	0.07	0.05	0.06	0.06	0.03
	32	0.04	0.04	0.02	188	192	0.51	0.78	0.19	0.18	0.16	0.15	0.07
	64	0.11	0.10	0.06	378	307	1.04	1.91	0.86	2.18	0.51	0.55	0.23
	128	0.24	0.23	0.11	751	482	2.28	5.92	2.61	3.72	1.06	1.23	0.53
	256	0.48	0.48	0.23	1486	756	4.26	11.14	12.11	50.04	2.91	3.43	3.66
world	16	0.02	0.02	0.02	99	122	0.29	0.33	0.08	0.05	0.07	0.08	0.03
	32	0.04	0.04	0.04	198	196	0.59	0.88	0.26	0.21	0.18	0.19	0.08
	64	0.12	0.11	0.09	385	306	1.16	1.70	1.09	4.84	0.64	0.54	0.35
	128	0.24	0.23	0.19	769	496	2.19	5.32	4.48	10.15	1.31	1.23	1.77
	256	0.47	0.47	0.36	1513	764	4.06	11.83	11.54	69.99	3.46	3.50	2.48
Averages over P													
	16	0.05	0.05	0.05	102	136	0.60	0.78	0.24	0.22	0.21	0.20	0.12
	32	0.10	0.10	0.08	210	214	1.12	1.53	1.23	1.47	0.51	0.51	0.43
	64	0.23	0.22	0.18	420	332	2.15	4.57	3.64	7.04	1.29	1.41	0.93
	128	0.54	0.53	0.36	819	509	4.26	13.34	15.37	20.1	2.86	3.52	4.51
	256	1.01	1.01	0.71	1640	779	7.98	35.49	56.84	92.69	6.62	8.17	12.07

SpM dataset, εBS+ is 3.4, 2.5, 1.8, 1.6, and 1.3 times faster than εBS for $P = 16$, 32, 64, 128, and 256, respectively. These improvement ratios in the execution times of the parametric search algorithms are below the improvement ratios in the numbers of probes displayed in Table 5. Overhead due to the *RB* call and initial settings of the separator indices contributes to this difference in both NC+ and εBS+. Furthermore, costs of initial probes with very large bottleneck values are very cheap in εBS.

In Table 6, relative performance comparison of the proposed exact CCP algorithms shows that BID is the clear winner for small to moderate P values (i.e., $P \leqslant 64$) in the SpM dataset. Relative performance of BID degrades with increasing P so that both εBS+ and NC+ begin to run faster than BID for $P \geqslant 128$ in general, where εBS+ becomes the winner. For the DVR dataset, NC+ and EBS are clear winners as seen in Table 7. NC+ runs slightly faster than EBS for $P \leqslant 128$, but EBS runs considerably faster than NC+ for $P = 256$. BID can compete with these two algorithms only for 16- and 32-way partitioning of grid *blunt-fin* (for all mesh resolutions). As seen in Table 6, BID takes less than 1% of a single SpMxV time

Table 7
Partitioning times (in msecs) for DVR dataset

CCP instance		Prefix	Heuristics			Exact algorithms							
Name	P	sum	$H1$	$H2$	RB	Existing			Proposed				
						DP	MS	NC	DP+	MS+	NC+	EBS	BID
blunt256	16		0.02	0.02	0.03	68	49	0.36	0.21	0.53	0.10	0.14	0.03
	32		0.05	0.05	0.05	141	77	0.77	0.76	0.76	0.21	0.27	0.24
	64	1.95	0.11	0.12	0.09	286	134	1.39	3.86	8.72	0.64	0.71	0.78
	128		0.24	0.25	0.20	581	206	3.66	12.37	29.95	1.78	1.84	2.33
	256		0.47	0.50	0.37	1139	296	52.53	42.89	0.78	13.96	3.11	56.69
blunt512	16		0.03	0.03	0.03	356	200	0.55	0.25	0.91	0.14	0.16	0.09
	32		0.07	0.07	0.07	792	353	1.31	1.23	2.13	0.44	0.43	0.22
	64	13.45	0.18	0.19	0.14	1688	593	2.65	3.68	9.28	0.94	1.10	0.45
	128		0.39	0.40	0.28	3469	979	5.84	15.01	46.72	2.29	2.63	1.65
	256		0.74	0.75	0.54	7040	1637	9.70	57.50	164.49	5.59	5.95	10.72
blunt1024	16		0.03	0.03	0.03	1455	780	0.75	0.93	4.77	0.25	0.28	0.19
	32		0.12	0.11	0.08	3251	1432	1.81	2.02	8.92	0.60	0.62	0.31
	64	59.12	0.27	0.27	0.19	6976	2353	3.50	7.35	22.28	1.27	1.37	1.39
	128		0.50	0.51	0.37	14 337	3911	9.14	33.01	69.38	3.36	3.56	16.21
	256		0.97	1.00	0.68	29 417	6567	16.59	180.09	538.10	8.48	8.98	16.29
post256	16		0.02	0.03	0.03	79	61	0.46	0.18	0.13	0.10	0.11	0.24
	32		0.05	0.05	0.05	157	100	0.77	1.05	1.74	0.26	0.31	0.76
	64	2.16	0.10	0.11	0.10	336	167	1.37	5.26	9.72	0.61	0.70	4.67
	128		0.25	0.26	0.18	672	278	2.90	22.94	55.10	1.65	1.77	23.96
	256		0.47	0.48	0.37	1371	338	45.16	84.78	0.77	13.04	3.62	299.32
post512	16		0.02	0.02	0.03	612	454	0.63	0.20	0.44	0.14	0.16	1.21
	32		0.07	0.07	0.07	1254	699	1.30	2.49	1.95	0.38	0.42	3.36
	64	20.55	0.18	0.18	0.13	2559	1139	2.58	16.93	38.73	0.97	1.07	8.53
	128		0.38	0.39	0.28	5221	1936	5.84	68.25	156.15	2.27	2.44	31.54
	256		0.70	0.73	0.53	10 683	3354	12.67	256.48	726.54	5.44	5.39	140.88
post1024	16		0.03	0.04	0.03	2446	1863	0.91	2.88	5.73	0.17	0.21	2.63
	32		0.09	0.08	0.08	5056	2917	1.61	13.57	17.37	0.51	0.58	12.73
	64	69.09	0.25	0.26	0.17	10 340	4626	3.56	35.05	33.25	1.35	1.47	32.53
	128		0.51	0.53	0.36	21 102	7838	7.90	156.34	546.92	2.95	3.28	76.96
	256		0.95	0.98	0.67	43 652	13770	16.30	764.59	1451.87	7.55	7.02	300.70
Averages normalized w.r.t. RB times (prefix-sum time not included)													
	16		0.83	0.94	1.00	27 863	18919	20.33	25.83	69.50	5.00	5.89	24.39
	32		1.10	1.06	1.00	23 169	12156	18.47	47.37	72.82	5.83	6.46	39.02
	64		1.30	1.35	1.00	22 636	9291	17.88	82.81	145.19	7.00	7.81	53.81
	128		1.35	1.39	1.00	22 506	7553	20.46	168.36	481.19	8.60	9.31	86.81
	256		1.35	1.39	1.00	24 731	6880	59.10	390.25	772.99	19.55	10.51	286.77
Averages normalized w.r.t. RB times (prefix-sum time included)													
	16		1.00	1.00	1.00	32	23	1.08	1.04	1.09	1.01	1.02	1.03
	32		1.00	1.00	1.00	66	36	1.15	1.21	1.29	1.04	1.05	1.13
	64		1.00	1.00	1.00	135	58	1.27	1.97	2.90	1.11	1.12	1.55
	128		1.01	1.01	1.00	275	94	1.62	4.75	10.53	1.28	1.30	3.25
	256		1.02	1.02	1.00	528	142	7.98	14.64	13.71	2.95	1.55	26.73

for $P \leqslant 64$ on average. For the DVR dataset (Table 7), the initial prefix-sum time is considerably larger than the actual partitioning time of the EBS algorithm in all CCP instances except 256-way partitioning of *post256*. As seen in Table 7, EBS is only 12% slower than the *RB* heuristic in 64-way partitionings on average.

These experimental findings show that the proposed exact CCP algorithms should replace heuristics.

7. Conclusions

We proposed runtime efficient chains-on-chains partitioning algorithms for optimal load balancing in 1-D decompositions of nonuniform computational domains. Our main idea was to run an effective heuristic, as a preprocessing step, to find a ''good'' upper bound on the optimal bottleneck value, and then exploit lower and upper bounds on the optimal bottleneck value to restrict the search space for separator-index values. This separator-index bounding scheme was exploited in a static manner in the dynamic-programming algorithm, drastically reducing the number of table entries computed and referenced. A dynamic separator-index bounding scheme was proposed for parametric search algorithms to narrow separator-index ranges after each probe. We enhanced the approximate bisection algorithm to be exact by updating lower and upper bounds into realizable values after each probe. We proposed a new iterative-refinement scheme, that is very fast for small-to-medium numbers of processors.

We investigated two distinct applications for experimental performance evaluation: 1D decomposition of irregularly sparse matrices for parallel matrix–vector multiplication, and decomposition for image-space parallel volume rendering. Experiments on the sparse matrix dataset showed that 64-way decompositions can be computed 100 times faster than a single sparse matrix vector multiplication, while reducing the load imbalance by a factor of four over the most effective heuristic. Experimental results on the volume rendering dataset showed that exact algorithms can produce 3.8 times better 64-way decompositions than the most effective heuristic, while being only 11% slower, on average.

8. Availability

The methods proposed in this work are implemented in C programming language and are made publicly available at http://www.cse.uiuc.edu/~alipinar/ccp/.

References

[1] S. Anily, A. Federgruen, Structured partitioning problems, Oper. Res. 13 (1991) 130–149.

[2] C. Aykanat, F. Ozguner, Large grain parallel conjugate gradient algorithms on a hypercube multiprocessor in: Proceedings of the 1987 International Conference on Parallel Processing, Penn State University, University Park, PA, 1987, pp. 641–645.

[3] C. Aykanat, F. Ozguner, F. Ercal, P. Sadayappan, Iterative algorithms for solution of large sparse systems of linear equations on hypercubes, IEEE Trans. Comput. 37 (12) (1988) 1554–1567.

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

[5] U.V. Çatalyurek, C. Aykanat, Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication, IEEE Trans. Parallel Distributed Systems 10 (7) (1999) 673–693.

[6] H. Choi, B. Narahari, Algorithms for mapping and partitioning chain structured parallel computations, in: Proceedings of the 1991 International Conference on Parallel Processing, Austin, TX, 1991 pp. I-625–I-628.

[7] G.N. Frederickson, Optimal algorithms for partitioning trees and locating *p*-centers in trees, Technical Report CSD-TR-1029, Purdue University (1990).

[8] G.N. Frederickson, Optimal algorithms for partitioning trees in: Proceedings of the Second ACM-SIAM Symposium Discrete Algorithms, Orlando, FL, 1991.

[9] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman, San Fransisco, CA, 1979.

[10] M.R. Garey, D.S. Johnson, L. Stockmeyer, Some simplified np-complete graph problems, Theoret. Comput. Sci. 1 (1976) 237–267.

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

[12] Y. Han, B. Narahari, H.-A. Choi, Mapping a chain task to chained processors, Inform. Process. Lett. 44 (1992) 141–148.

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

[14] B. Hendrickson, T. Kolda, Rectangular and structurally nonsymmetric sparse matrices for parallel processing, SIAM J. Sci. Comput. 21 (6) (2000) 2048–2072.

[15] M.A. Iqbal, Efficient algorithms for partitioning problems, in: Proceedings of the 1990 International Conference on Parallel Processing, Urbana, Champaign, IL, 1990, pp. III-123–127.

[16] M.A. Iqbal, Approximate algorithms for partitioning and assignment problems, Int. J. Parallel Programming 20 (5) (1991) 341–361.

[17] M.A. Iqbal, S.H. Bokhari, Efficient algorithms for a class of partitioning problems, IEEE Trans. Parallel Distributed Systems 6 (2) (1995) 170–175.

[18] M.A. Iqbal, J.H. Saltz, S.H. Bokhari, A comparative analysis of static and dynamic load balancing strategies in: Proceedings of the 1986 International Conference on Parallel Processing, Penn State Univ University Park, PA, 1986, pp. 1040–1047.

[19] V. Kumar, A. Grama, A. Gupta, G. Karypis, Introduction to Parallel Computing, Benjamin, Cummings, Redwood City, CA, 1994.

[20] H. Kutluca, T.M. Kurç, C. Aykanat, Image-space decomposition algorithms for sort-first parallel volume rendering of unstructured grids, J. Supercomput. 15 (1) (2000) 51–93.

[21] T. Lengauer, Combinatorial Algorithms for Integrated Circuit Layout, Wiley, Chichester, UK, 1990.

[22] Linear programming problems, Technical Report, IOWA Optimization Center, ftp://col.biz.uiowa.edu:pub/testprob/lp/gondzio.

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

[24] S. Miguet, J.M. Pierson, Heuristics for 1d rectilinear partitioning as a low cost and high quality answer to dynamic load balancing, Lecture Notes in Computer Science, Vol. 1225, Springer, Berlin, 1997, pp. 550–564.

[25] S. Molnar, M. Cox, D. Ellsworth, H. Fuchs, A sorting classification of parallel rendering, IEEE Comput. Grap. Appl. 14 (4) (1994) 23–32.

[26] C. Mueller, The sort-first rendering architecture for high-performance graphics, in: Proceedings of the 1995 Symposium on Interactive 3D Graphics, Monterey, CA, 1995, pp. 75–84.

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

[28] D.M. Nicol, D.R. O'Hallaron, Improved algorithms for mapping pipelined and parallel computations, Technical Report 88-2, ICASE, 1988.

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

[30] B. Olstad, F. Manne, Efficient partitioning of sequences, IEEE Trans. Comput. 44 (11) (1995) 1322–1326.
[31] J.R. Pilkington, S.B. Baden, Dynamic partitioning of non-uniform structured workloads with spacefilling curves, IEEE Trans. Parallel Distributed Systems 7 (3) (1996) 288–299.
[32] Y. Saad, Practical use of polynomial preconditionings for the conjugate gradient method, SIAM J. Sci. Statist. Comput. 6 (5) (1985) 865–881.
[33] M.U. Ujaldon, E.L. Zapata, B.M. Chapman, H.P. Zima, Vienna-Fortran/hpf extensions for sparse and irregular problems and their compilation, IEEE Trans. Parallel Distributed Systems 8 (10) (1997) 1068–1083.
[34] M.U. Ujaldon, E.L. Zapata, S.D. Sharma, J. Saltz, Parallelization techniques for sparse matrix applications, J. Parallel Distributed Computing 38 (1996) 256–266.
[35] M.U. Ujaldon, E.L. Zapata, S.D. Sharma, J. Saltz, Experimental evaluation of efficient sparse matrix distributions, in: Proceedings of the ACM International Conference of Supercomputing Philadelphia, PA, (1996) pp. 78–86.

Ali Pinar's research interests are combinatorial scientific computing—the boundary between scientific computing and combinatorial algorithms—and high performance computing. He received his B.S. and M.S. degrees in Computer Science from Bilkent University, Turkey, and his Ph.D. degree in Computer Science with the option of Computational Science and Engineering from University of Illinois at Urbana-Champaign. During his Ph.D. studies he frequently visited Sandia National Laboratories in New Mexico, which had a significant influence on his education and research. He joined the Computational Research Division of Lawrence Berkeley National Laboratory in October, 2001, where he currently works as a research scientist.

Cevdet Aykanat received the B.S. and M.S. degrees from Middle East Technical University, Ankara, Turkey, both in electrical engineering, and the Ph.D. degree from Ohio State University, Columbus, in electrical and computer engineering. He was a Fulbright scholar during his Ph.D. studies. He worked at the Intel Supercomputer Systems Division, Beaverton, Oregon, as a research associate. Since 1989, he has been affiliated with the Department of Computer Engineering, Bilkent University, Ankara, Turkey, where he is currently a professor. His research interests mainly include parallel computing, parallel scientific computing and its combinatorial aspects, parallel computer graphics applications, parallel data mining, graph and hypergraph-partitioning, load balancing, neural network algorithms, high performance information retrieval and GIS systems, parallel and distributed web crawling, distributed databases, and grid computing. He is a member of the ACM and the IEEE Computer Society. He has been recently appointed as a member of IFIP Working Group 10.3 (Concurrent Systems).