One-Dimensional Partitioning for Heterogeneous Systems: Theory and Practice ⋆

Ali Pınar [a,1], E. Kartal Tabak [b] and Cevdet Aykanat [b,2]

[a] *Computational Research Division. Lawrence Berkeley National Laboratory*

[b] *Department of Computer Engineering, Bilkent University*

Abstract

We study the problem of one-dimensional partitioning of nonuniform workload arrays with optimal load balancing for heterogeneous systems. We look at two cases: chain-on-chain partitioning, where the order of the processors is specified, and chain partitioning, where processor permutation is allowed. We present polynomial time algorithms to solve the chain-on-chain partitioning problem optimally, while we prove that the chain partitioning problem is NP-complete. Our empirical studies show that our proposed exact algorithms produce substantially better results than heuristics while the solution times remain comparable.

Key words: parallel computing, one-dimensional partitioning, load balancing, chain-on-chain partitioning, dynamic programming, parametric search

Preprint submitted to Journal of Parallel and Distributed Computing14 October 2007

1 Introduction

In many applications of parallel computing, load balancing is achieved by mapping a possibly multi-dimensional computational domain down to a one-dimensional (1D) array, and then partitioning this array into parts with equal weights. Space filling curves are commonly used to map the higher dimensional domain to a 1D workload array to preserve locality and minimize communication overhead after partitioning [6,7,10,15]. Similarly, processors can be mapped to a 1D array so that communication is relatively faster between close processors in this processor chain [11]. This eases mapping for computational domains and improves efficiency of applications. The load balancing problem for these applications can be modeled as the chain-on-chain partitioning (CCP) problem, where we map a chain of tasks onto a chain of processors. Formally, 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 to minimize maximum load among processors.

In our earlier work [17], we studied the CCP problem for homogenous systems, where all processors have identical computational powers. We have surveyed the rich literature on this problem, proposed novel methods as well as im-

⋆ This work is partially supported by The Scientific and Technological Research Council of Turkey (TÜBİTAK) under project EEEAG-106E069.

Email addresses: `apinar@lbl.gov` (Ali Pınar), `tabak@cs.bilkent.edu.tr` (E. Kartal Tabak), `aykanat@cs.bilkent.edu.tr` (Cevdet Aykanat).

[1] Supported by the Director, Office of Science, Division of Mathematical, Information, and Computational Sciences of U.S. Department of Energy under contract DE-AC03-76SF00098.

[2] Corresponding author.

provements on existing methods, and studied how these algorithms can be implemented efficiently to be effective in practice. In this work, we investigate how these techniques can be generalized for heterogeneous systems, where processors have varying computational powers. Two distinct problems arise in partitioning chains for heterogeneous systems. The first problem is the CCP problem, where a chain of tasks is to be mapped onto a chain of processors, i.e., the pth task subchain in a partition is assigned to the pth processor. The second problem is the chain partitioning (CP) problem, where a chain of tasks is to be mapped onto a *set*, as opposed to a chain, of processors, i.e., processors can be permuted for subchain assignments. For brevity, the CCP problem for homogenous systems and heterogeneous systems will be referred to as the homogenous CCP problem and heterogeneous CCP problem, respectively. The CP problem refers to the chain partitioning problem for heterogeneous systems, since it has no counterpart for homogenous systems.

In this article, we show that the heterogeneous CCP problem can be solved in polynomial time by enhancing the exact algorithms proposed for the solution of the homogenous CCP problem [17]. We present how these exact algorithms for homogenous systems can be enhanced for heterogeneous systems and implemented efficiently for runtime performance. We also present how the heuristics widely used for the solution of homogenous CCP problem can be adapted for heterogeneous systems. We present the implementation details and pseudocodes for the exact algorithms and heuristics for clarity and reproducibility. Our experiments with workload arrays coming from image-space-parallel volume rendering and row-parallel sparse matrix vector multiplication applications show that our proposed exact algorithms produce substantially better results than the heuristics while the solution times remain compara-

ble. On average, optimal solutions provide 4.1 and 5.5 **??**times better load imbalance than heuristics for 128-way partitionings of volume rendering and sparse matrix datasets, respectively. On average, the time it takes to compute an optimal solution is less than 2.20 **??** times the time it takes to compute an approximation using heuristics for 128 processors, and thus the preprocessing times can be easily compensated by the improved efficiency of the subsequent computation even for a few iterations.

The CP problem on the other hand, is NP-complete as we prove in this paper. Our proof uses a pseudo-polynomial reduction from the 3-Partition problem, which is known to be NP-complete in the strong sense [8]. Our empirical studies showed that processor ordering has a very limited effect on the solution quality, and an optimal CCP solution on a random processing ordering serves as an effective CP heuristic.

The remainder of this paper is organized as follows. Table 1 summarizes important symbols used throughout the paper. Section 2 introduces the heterogeneous CCP problem. In Section 3, we summarize the solution methods for homogenous CCP. In Section 4, we discuss how solution methods for homogenous systems can be enhanced to solve the heterogeneous CCP problem. In Section 5, we discuss the CP problem, prove that it is NP-Complete. We present the results of our empirical studies with the proposed methods in Section 6, and finally, we conclude with Section 7.

2 Chain-on-chain (CCP) Problem for Heterogeneous Systems

In the heterogeneous CCP problem, a computational problem, which is decomposed into a chain $\mathcal{T} = \langle t_1, t_2, \ldots, t_N \rangle$ of N tasks with associated *positive* computational weights $\mathcal{W} = \langle w_1, w_2, \ldots, w_N \rangle$ is to be mapped onto a pro-

Table 1
The summary of important abbreviations and symbols

Notation	Explanation
N	number of tasks
$\mathcal{T}$	task chain, i.e., $\mathcal{T} = \langle t_1, t_2, \ldots, t_N \rangle$
t_i	ith task in the task chain
$\mathcal{T}_{i,j}$	task subchain of tasks from t_i upto t_j, i.e., $\mathcal{T}_{i,j} = \langle t_i, t_{i+1}, \ldots, t_j \rangle$
w_i	computational load of task t_i
$w_{\max}$	maximum computational load among all tasks
w_{avg}	average computational load of all tasks
$w_{\min}$	minimum computational load of all tasks
$W_{i,j}$	total computational load of task subchain $\mathcal{T}_{i,j}$
W_{tot}	total computational load, i.e., $W_{\text{tot}} = W_{1,N}$
P	number of processors
$\mathcal{P}$	processor chain, i.e., $\mathcal{P} = \langle \mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_P \rangle$ in the CCP problem
	processor set, i.e., $\mathcal{P} = \{\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_P\}$ in the CP problem
$\mathcal{P}_p$	pth processor in the processor chain
$\mathcal{P}_{q,r}$	processor subchain from $\mathcal{P}_q$ upto $\mathcal{P}_r$, i.e., $\mathcal{P}_{q,r} = \langle \mathcal{P}_q, \mathcal{P}_{q+1}, \ldots, \mathcal{P}_r \rangle$
e_p	execution speed of processor $\mathcal{P}_p$
$E_{q,r}$	total execution speed of processor subchain $\mathcal{P}_{q,r}$
E_{tot}	total execution speed of all processors, i.e., $E_{\text{tot}} = E_{1,P}$
B^*	ideal bottleneck value, achieved when all processors have load in proportion to their speed
UB	upper bound on the value of an optimal solution
LB	lower bound on the value of an optimal solution
s_p	index of the last task assigned to the pth processor.
$\lg x$	base-2 logarithm of x, i.e., $\lg x = \log_2 x$.

cessor chain $\mathcal{P} = \langle \mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_P \rangle$ of P processors with associated execution speeds $\mathcal{E} = \langle e_1, e_2, \ldots, e_P \rangle$. The execution time of task t_i on processor $\mathcal{P}_p$ is w_i/e_p. For clarity, we note that there are no precedence constraints among the tasks in the chain.

A *task subchain* $\mathcal{T}_{i,j} = \langle t_i, t_{i+1}, \ldots, t_j \rangle$ is defined as a subset of contiguous tasks. Note that $\mathcal{T}_{i,j}$ defines an empty task subchain when $i > j$. The computational weight of $\mathcal{T}_{i,j}$ is $W_{i,j} = \sum_{i \le h \le j} w_h$. A partition Π should map contiguous task subchains to contiguous processors. Hence, a P-way partition of a task chain with N tasks onto a processor chain with P processors is described by a sequence $\Pi = \langle s_0, s_1, \ldots, s_P \rangle$ of $P+1$ separator indices, where $s_0 = 0 \le s_1 \le \cdots \le s_P = N$. Here, s_p denotes the index of the last task of the pth part so that processor $\mathcal{P}_p$ receives the task subchain $\mathcal{T}_{s_{p-1}+1,s_p}$ with load $W_{s_{p-1}+1,s_p}/e_p$.

The cost $C(\Pi)$ of a partition Π is determined by the maximum processor load among all processors, i.e.,

$$C(\Pi) = \max_{1 \le p \le P} \left\{ \frac{W_{s_{p-1}+1,s_p}}{e_p} \right\} \tag{1}$$

This $C(\Pi)$ value of a partition is called its *bottleneck value*, and the processor defining it is called the *bottleneck processor*. The CCP problem is to find a partition Π_{opt} that minimizes the bottleneck value $C(\Pi_{\text{opt}})$.

Similar to the task subchain, a processor subchain $\mathcal{P}_{q,r} = \langle \mathcal{P}_q, \mathcal{P}_{q+1}, \ldots, \mathcal{P}_r \rangle$ is defined as a subset of contiguous processors. Note that $\mathcal{P}_{q,r}$ defines an empty processor subchain when $q > r$. The computational speed of $\mathcal{P}_{q,r}$ is $E_{q,r} = \sum_{q \le p \le r} e_p$.

The ideal bottleneck value B^* is defined as

$$B^* = \frac{W_{\text{tot}}}{E_{\text{tot}}}, \tag{2}$$

where E_{tot} is the sum of all processor speeds and W_{tot} is the total task weight; i.e., $E_{\text{tot}} = E_{1,P}$ and $W_{\text{tot}} = W_{1,N}$. Note that B^* can only be achieved when all processors are equally loaded, so it constitutes a lower bound on the achievable bottleneck values, i.e., $B^* \le C(\Pi_{\text{opt}})$.

3 CCP Algorithms for Homogenous Systems

The homogenous CCP problem can be considered as a special case of the heterogeneous CCP problem, where the processors are assumed to have equal speed, i.e., $e_p = 1$ for all p. Here, we review the CCP algorithms for homogenous systems. A comprehensive review and presentation of homogenous CCP algorithms are available in [17].

3.1 Heuristics

Possibly the most commonly used CCP heuristic is *recursive bisection* (RB), a greedy algorithm. RB achieves P-way partitioning through $\lg P$ levels of bisection steps. At each level, the workload array is divided evenly into two. RB finds the optimal bisection at each level, but the sequence of optimal bisections at each level may lead to a multi-way partition which is far away from an optimal. Pınar and Aykanat [17] proved that RB produces partitions with bottleneck values no greater than $B^* + w_{\max}(P-1)/P$.

Miguet and Pierson [13] proposed another heuristic that determines s_p by bipartitioning the task chain in proportion to the length of the respective processor subchains. That is, s_p is selected in such a way that $W_{1,s_p}/W_{1,N}$ is as close to the ratio p/P as possible. Miguet and Pierson [13] prove that the bottleneck value found by this heuristic has an upper bound of $B^* + w_{\max}$.

These heuristics can be implemented in $O(N + P \lg N)$ time. The $O(N)$ time is due to prefix-sum operation on the tasks array, after which each separator index can be found by a binary search on the prefix-summed array.

3.2 Dynamic Programming

The overlapping subproblems and the optimal substructure properties of the CCP problem enable dynamic programming solutions. The overlapping subproblems are partitioning the first i tasks onto the first p processors, for all possible i and p values. For the *optimal substructure* property, observe that if the last processor is not the bottleneck processor in an optimal partition, then the partitioning of the remaining tasks onto the first $P-1$ processors must be optimal. Hence, the recursive definition for the bottleneck value of an

optimal partition is

$$B_i^p = \min_{0 \le j \le i} \left\{ \max \left\{ B_j^{p-1}, W_{j+1,i} \right\} \right\} \tag{3}$$

Here, B_i^p denotes the optimal solution value for partitioning the first i tasks onto the first p processors. In Eq. (3), 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 in an optimal partition. This definition defines a dynamic programming table of size PN, and computing each entry takes $O(N)$ time, resulting in an $O(N^2P)$-time algorithm. Choi and Narahari [3], and Manne and Olstad [12] reduced the complexity of this scheme to $O(NP)$ and $O((N-P)P)$, respectively. Pınar and Aykanat [17] presented enhancements to limit the search space of each separator by exploiting upper and lower bounds on the optimal solution value for better practical performance.

3.3 Parametric Search

Parametric search algorithms rely on two components: a probing operation to determine if a solution exists whose bottleneck value is no greater than a specified value, and a method to search the space of candidate values. The probe algorithm can be computed only in $O(P \lg N)$ time by using binary search on the prefix-summed workload array. Below, we summarize algorithms to search the space of bottleneck values.

3.3.1 Nicol's Algorithm

Nicol's algorithm [14] exploits the fact that any candidate B value is equal to the weight of a task subchain. A naive solution is to generate all subchain weights, sort them, and then use binary search to find the minimum value for which a probe succeeds. Nicol's algorithm efficiently searches for this subchain by considering each processor in order as a candidate bottleneck processor. For

each processor $\mathcal{P}_p$, the algorithm does a binary search for the smallest index that will make $\mathcal{P}_p$ the bottleneck processor. With the $O(P \lg N)$ cost of each probing, Nicol's algorithm runs in $O(N+(P \lg N)^2)$ time.

Pınar and Aykanat [17] improved Nicol's algorithm by utilizing the following simple facts. If the probe function succeeds (fails) for some B, then probe function will succeed (fail) for any $B' \geq (\leq)\ B$. Therefore by keeping the smallest B that succeeded and the largest B that failed, unnecessary probing is eliminated, which drastically improves runtime performance [17].

3.3.2 Bidding Algorithm

The bidding algorithm [17,16] starts with a lower bound and proceeds by gradually increasing this bound until a feasible solution value is reached. The increments are chosen to be minimal so that the first feasible bottleneck value is optimal. Consider the partition generated by a failed probe call that loads the first $P-1$ processors maximally not to exceed the specified probe value. To find the next bottleneck value, processors bid with the bottleneck value that would add one more task to their domain, and the minimum bid among the processors is chosen to be the next bottleneck value. The bidding algorithm moves each one of the P separators for $O(N)$ positions in the worst case, where choosing the new bottleneck value takes $O(\lg P)$ time using a priority queue. This makes the complexity of the algorithm $O(NP \lg P)$.

3.3.3 Bisection Algorithms

The bisection algorithm starts with a lower and an upper bound on the solution value and uses binary search in this interval. If the solution value is known to be an integer, then the bisection algorithm finds an optimal solution. Otherwise, it is an ϵ-approximation algorithm, where ϵ is the user defined accuracy for

the solution. The bisection algorithm requires $O(\lg(w_{\max}/\epsilon))$ probe calls, with $O(N+P\lg N\lg(w_{\max}/\epsilon))$ overall complexity.

Pınar and Aykanat [17] enhanced the bisection algorithm by updating the lower and upper bounds to realizable bottleneck values (subchain weights). After a successful probe, the upper bound can be set to be the bottleneck value of the partition generated by the probe function, and after a failed probe, the lower bound can be set to be the smallest value that might succeed, as in the bidding algorithm. These enhancements transform the bisection algorithm to an exact algorithm, as opposed to an ϵ-approximation algorithm.

4 Proposed CCP Algorithms for Heterogeneous Systems

The algorithms we propose in this section extend the techniques for homogenous CCP to heterogeneous CCP. All algorithms discussed in this section require an initial prefix-sum operation on the task-weight array $\mathcal{W}$ for 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 weight W_{ij} of a task subchain $\mathcal{T}_{ij}$ 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 $O(N)$ cost of this initial prefix-sum operation is considered in the complexity analysis. Similarly, $E_{a,b}$ can be computed in $O(1)$ time on a prefix-summed processor-speed array. In all algorithms, we focus only on finding the optimal solution value, since an optimal solution can be easily constructed, once the optimal solution value is known.

Unless otherwise stated, *BINSEARCH* represents a binary search that finds the index to the element that is closest to the target value. There are variants of *BINSEARCH* to find the index of the greatest element not greater than the

target value, and we will state whenever such variants are needed. *BINSEARCH* takes four parameters: the array to search, the start and end indices of the sub-array, and the target value. The range parameters are optional, and their absence means that the search will be performed on the whole array.

4.1 Heuristics

We propose a heuristic, *RB*, based on the recursive bisection idea. During each bisection, *RB* performs a two step process. First, it divides the current processor chain $\mathcal{P}_{p,r}$ into two subchains $\mathcal{P}_{p,q}$ and $\mathcal{P}_{q+1,r}$. Then, it divides the current task chain $\mathcal{T}_{h,j}$ into two subchains $\mathcal{T}_{h,i}$ and $\mathcal{T}_{i+1,j}$ in proportion to the computational powers of the respective processor subchains. That is, the task separator index i is chosen such that the ratio $W_{h,i}/W_{i+1,j}$ is close to the ratio $E_{p,q}/E_{q+1,r}$, as much as possible. RB achieves optimal bisections at each level, however, the quality of the overall partition may be far away from that of the optimal solution.

We have investigated two metrics for bisecting the processor chain: chain length and chain processing power. The chain length metric divides the current processor chain $\mathcal{P}_{p,r}$ into two equal-length processor subchains, whereas the chain processing power metric divides $\mathcal{P}_{p,r}$ into two equal-power subchains. Since the first metric performed slightly better than the second one in our experiments, we will only discuss the chain length metric here. The pseudocode of the RB algorithm is given in Fig. 1, where the initial invocation takes its parameters as $(\mathcal{W}, \mathcal{E}, 1, P)$ with $s_0 = 0$ and $s_P = N$. Note that s_{p-1} and s_r are already determined at higher levels of recursion. *Wtot* is the total weight of current task subchain, and *Wfirst* is the weight for the first processor subchain in proportion to its processing speed. We need to add $W_{1,s_{p-1}}$ to *Wfirst* to seek

```
RB (W, E, p, r)
if p = r then
   return;
Wtot ← W_{s_{p-1}+1, s_r};
q ← (p + r − 1)/2;
Wfirst ← Wtot × E_{p,q}/E_{p,r};
W ← Wfirst + W_{1, s_{p-1}};
s_q ←BINSEARCH(W, s_{p-1}, s_r, W);
RB(W, E, p, q);
RB(W, E, q + 1, r);
```

```
MP (W, N, E, P)
for p ← 1 to P do
   w ← W_{1,N} × E_{1,p}/E_{1,P};
   s_p ←BINSRCH(W, w, s_{p-1}, N);
```

Fig. 1. Heterogeneous CCP heuristics

it in the prefix-summed $\mathcal{W}$ array.

We also propose a generalization of Miguet and Pierson's heuristic, *MP* [13]. *MP* computes the separator index of each processor by considering that processor as a division point for the whole processor chain. In our version, the load assigned to the processor chain $\mathcal{P}_{1,p}$ is set to be proportional to the computational power $E_{1,p}$ of this subchain, as shown in Fig. 1.

Both *RB* and *MP* can be implemented in $O(N+P\lg N)$ time, where the $O(N)$ time is due to the initial prefix-sum operation on the task-weight array.

Below, we investigate the theoretical bounds on the quality of these two heuristics. We assume P is a power of 2 for simplicity.

Lemma 4.1 *B_{RB} is upper bounded by $B^*+w_{\max}/e_{\min} - w_{\max}/(Pe_{\min})$.*

Proof: We use induction, and the basis is easy to show for $P = 2$. For the inductive step, assume the hypothesis holds for any number of processors less than P. Consider the first bisection, where the processors are split into two subchains, each containing $P/2$ processors. Let the total processing power in the left subchain be E_{left}. *RB* will distribute the workload array between the left and right processor subchains as evenly as possible. There will be a task t_i such that the left processor subchain will weigh more than the right

subchain if t_i is assigned to the left subchain, and vice versa. Without loss of generality, assume that t_i is assigned to the left subchain. In the worst case, t_i is the maximum weighted task, and the total task weight assigned to the left subchain, W_{left}, can be upper bounded by

$$W_{\text{left}} \leq \frac{(W_{\text{tot}} + w_{\max})E_{\text{left}}}{E_{\text{tot}}}.$$

Using the inductive hypothesis, the bottleneck value among the processors of the left processor subchain can be upper bounded as follows.

$$\begin{aligned} B_{RB} &\leq \frac{W_{\text{left}}}{E_{\text{left}}} + \frac{w_{\max}}{e_{\min}} - \frac{w_{\max}}{e_{\min}P/2} &&\leq \frac{W_{\text{tot}} + w_{\max}}{E_{\text{tot}}} + \frac{w_{\max}}{e_{\min}} - \frac{w_{\max}}{e_{\min}P/2} \\ &= B^* + \frac{w_{\max}}{E_{\text{tot}}} + \frac{w_{\max}}{e_{\min}} - \frac{w_{\max}}{e_{\min}P/2} &&\leq B^* + \frac{w_{\max}}{e_{\min}P} + \frac{w_{\max}}{e_{\min}} - \frac{w_{\max}}{e_{\min}P/2} \\ &= B^* + \frac{w_{\max}}{e_{\min}} - \frac{w_{\max}}{Pe_{\min}} \end{aligned}$$

The same bound applies to the right processor subchain directly by the inductive hypothesis, since right processor subchain is already underloaded. ■

Lemma 4.2 *B_{MP} is upper bounded by $B^* + w_{\max}/e_{\min}$.*

Proof: Let the sequence $\langle s_0, s_1, \ldots, s_P \rangle$ be the partition constructed by *MP*. For a processor $\mathcal{P}_p$, s_p is chosen to be the separator that best divides $\mathcal{P}_{1,p}$ and $\mathcal{P}_{p+1,P}$. Based on our discussion of bipartitioning quality in the proof of Lemma 4.1, W_{1,s_p} is bounded by

$$E_{1,p}B^* - \frac{w_{\max}}{2} \leq W_{1,s_p} \leq E_{1,p}B^* + \frac{w_{\max}}{2}$$

So, the load of processor p is upper bounded by

$$\begin{aligned} \frac{W_{1,s_p} - W_{1,s_{p-1}}}{e_p} &\leq \frac{E_{1,p}B^* + w_{\max}/2 - E_{1,p-1}B^* + w_{\max}/2}{e_p} \\ &= B^* + \frac{w_{\max}}{e_p} \leq B^* + \frac{w_{\max}}{e_{\min}} \end{aligned}$$

■

The overlapping subproblems and the optimal substructure properties of the homogenous CCP can be extended to the heterogeneous CCP, and thus enabling dynamic programming solutions. The recursive definition for the bottleneck value of an optimal partition can be derived as

$$B_i^p = \min_{0 \le j \le i} \left\{ \max \left\{ B_j^{p-1}, \frac{W_{j+1,i}}{e_p} \right\} \right\} \tag{4}$$

for the heterogeneous case. As in the homogenous case, B_i^p denotes the optimal solution value for partitioning the first i tasks onto the first p processors. This definition results in an $O(N^2P)$-time DP algorithm.

We generalize the observations of Choi and Narahari [3] to develop an $O(NP)$-time algorithm for heterogeneous systems as follows. Their first observation relies on the fact that the optimal position of the separator for partitioning the first i tasks cannot be to the left of the optimal position for the first $i-1$ tasks, i.e., $j_i^p \ge j_{i-1}^p$. Their second observation is that we need to advance a separator index only when the last part is overloaded and can stop when this is no longer the case, i.e., $B_j^{p-1} \ge W_{j+1,i}/e_p$. Then an optimal j_i^p can be chosen to correspond to the minimum of $\max\{B_j^{p-1}, W_{j+1,i}/e_p\}$ and $\max\{B_{j-1}^{p-1}, W_{j,i}/e_p\}$. That is, the recursive definition becomes:

$$B_i^p = \max \left\{ B_{j_i^p}^{p-1}, \frac{W_{j_i^p+1,i}}{e_p} \right\}, \text{ where } j_i^p = \operatorname*{argmin}_{j_{i-1}^p \le j \le i} \left\{ \max \left\{ B_j^{p-1}, \frac{W_{j+1,i}}{e_p} \right\} \right\}.$$

It is clear that the search ranges of separators overlap at only one position, and thus we can compute all B_i^p entries for $1 \le i \le N$ in only one pass over the task subchain. This reduces the complexity of the algorithm to $O(NP)$. Fig. 2(a) presents this algorithm.

```
DP (W, N, P, E)
for p ← 1 to P do
  B[p, 0] ← 0;
for i ← 1 to N do
  B[1, i] ← W_{1,i}/e_1;
for p ← 2 to P do
  j ← 0;
  for i ← 1 to N do
    if W_{j+1,i}/e_p ≤ B[p − 1, j] then
      B[p, i] ← B[p − 1, j];
    else
      repeat
        j ← j + 1;
      until W_{j+1,i}/e_p ≤ B[p − 1, j] or j ≥ i;
      if W_{j,i}/e_p < B[p − 1, j] then
        j ← j − 1;
        B[p, i] ← W_{j,i}/e_p;
      else
        B[p, i] ← B[p − 1, j];
return B_opt ← B[P, N];
```

(a)

```
DP+ (W, N, E, P, SL, SH)
for p ← 1 to P do
  B[p, 0] ← 0;
for i ← SL_1 to SH_1 do
  B[1, i] ← W_{1,i}/e_1;
for p ← 2 to P do
  j ← SL_{p−1};
  for i ← SL_p to SH_p do
    if W_{j+1,i}/e_p ≤ B[p − 1, j] then
      B[p, i] ← B[p − 1, j];
    else
      repeat
        j ← j + 1;
      until W_{j+1,i}/e_p ≤ B[p − 1, j] or j ≥ i;
      if W_{j,i}/e_p < B[p − 1, j] then
        j ← j − 1;
        B[p, i] ← W_{j,i}/e_p;
      else
        B[p, i] ← B[p − 1, j];
return B_opt ← B[P, N];
```

(b)

Fig. 2. DP algorithms for heterogeneous systems: (a) basic DP algorithm, and (b) DP algorithm (*DP+*) with static separator index bounding.

Manne and Olstad [12] reduced the complexity further to $O((N-P)P)$ by observing that there is no merit in leaving a processor empty, and thus the search for j_i^p can start at p instead of 1. However, this does not apply to the heterogeneous CCP, since it might be beneficial to leave a processor empty.

We propose another DP algorithm by extending the *DP+* algorithm (*DP* algorithm with static separator-index bounding) of Pınar and Aykanat [17] for the heterogeneous case. *DP+* limits the search space of each separator to avoid redundant calculation of B_i^p values. *DP+* achieves this separator index bounding by running left-to-right and right-to-left probe functions with the upper and lower bounds on the optimal bottleneck value.

We extend the probing operation to the heterogeneous case as shown in

```
LR-PROBE (W, N, E, P, B)
sum ← 0;
for p ← 1 to P − 1 do
   myB ← B × e_p;
   Bsum ← sum + myB;
   m ← BINSEARCH(W, Bsum);
   sum ← W_{1,m};
   s_p ← m;
if sum + B × e_P ≥ W_{1,N} then
   return TRUE;
else
   return FALSE;
```

(a)

```
RL-PROBE (W, N, E, P, B)
sum ← W_{1,N};
for p ← P downto 2 do
   myB ← B × e_p;
   Bsum ← sum − myB;
   m ← BINSEARCH(W, Bsum);
   sum ← W_{1,m};
   s_{p−1} ← m;
if sum − B × e_1 ≤ 0 then
   return TRUE;
else
   return FALSE;
```

(b)

Fig. 3. Greedy *PROBE* algorithms for heterogeneous systems: (a) left-to-right, and (b) right-to-left.

Fig. 3. In the figure, *LR-PROBE* and *RL-PROBE* denote the left-to-right probe and right-to-left probe, respectively. These algorithms not only decide whether a candidate value is a feasible bottleneck value, but they also set the separator index (s_p) values for their greedy approach. In *LR-PROBE*, *BINSEARCH*$(\mathcal{W}, w)$ refers to a binary search algorithm that searches $\mathcal{W}$ for the largest index m such that $W_{1,m} \leq w$. Similarly, in *RL-PROBE*, *BINSEARCH*$(\mathcal{W}, w)$ searches $\mathcal{W}$ for the smallest index m such that $W_{1,m} \geq w$.

DP+, as presented in Fig. 2(b), uses Lemma 4.3 to limit the search space of s_p values.

Lemma 4.3 *For a given heterogeneous CCP instance* $(\mathcal{W}, N, \mathcal{E}, P)$*, a feasible bottleneck value UB and a lower bound on the bottleneck value LB; let the sequences* $\Pi^1 = \langle h_0^1, h_1^1, \ldots, h_P^1 \rangle$*,* $\Pi^2 = \langle l_0^2, l_1^2, \ldots, l_P^2 \rangle$*,* $\Pi^3 = \langle l_0^3, l_1^3, \ldots, l_P^3 \rangle$ *and* $\Pi^4 = \langle h_0^4, h_1^4, \ldots, h_P^4 \rangle$ *be the partitions constructed by LR-PROBE*(UB)*, RL-PROBE*(UB)*, LR-PROBE*(LB) *and RL-PROBE*(LB)*, respectively. Then, an optimal partition* $\Pi_{\text{opt}} = \langle s_0, s_1, \ldots, s_P \rangle$ *satisfies* $SL_p \leq s_p \leq SH_p$ *for all* $1 \leq p \leq P$*, where* $SL_p = \max\{l_p^2, l_p^3\}$ *and* $SH_p = \min\{h_p^1, h_p^4\}$*.*

Proof: We know that any feasible bottleneck value is greater than or equal to the optimal bottleneck value, i.e., $UB \geq B_{\text{opt}}$. Consider h_p^1, which is the largest index such that the first h_p^1 tasks can be partitioned over p processors without exceeding UB. Then $s_p > h_p^1$ implies $B_{\text{opt}} > UB$, which is a contradiction. So, $s_p \leq h_p^1$. Since, *RL-PROBE* is just the symmetric algorithm of *LR-PROBE*, the same argument proves $s_p \geq l_p^2$.

Consider the optimal partition constructed by *RL-PROBE*(B_{opt}). Since $B_{\text{opt}} \geq LB$, by the greedy property of *RL-PROBE*, $s_p \leq h_p^4$. Assume $s_p < l_p^3$ for some p, then another partition obtained by advancing the s_p value to l_p^3 does not increase the bottleneck value, since the first l_p^3 tasks are successfully partitioned over the first p processors without exceeding LB and thus B_{opt}. An optimal partition $\Pi_{\text{opt}} = \langle s_0, s_1, \ldots, s_P \rangle$ satisfies $l_p^3 \leq s_p \leq h_p^4$. ■

The lower bound LB can be initialized to the optimal lower bound when all processors are equally loaded as

$$LB = B^* = \frac{W_{\text{tot}}}{E_{\text{tot}}}. \tag{5}$$

An upper bound UB can be computed in practice with a fast and effective heuristic, and Lemma 4.1 provides a theoretically robust bound as

$$UB = B^* + \frac{w_{\max}}{e_{\min}} - \frac{w_{\max}}{Pe_{\min}}. \tag{6}$$

4.3 Parametric Search

Parametric search algorithms can be constructed with a *PROBE* function (either LR-PROBE or RL-PROBE given in Fig. 3), and a method to search the space of candidate values. Below, we describe several algorithms to search the space of bottleneck values for the heterogeneous case.

4.3.1 Nicol's Algorithm

We revise Nicol's algorithms for heterogeneous systems as follows. The candidate B values become task subchain weights divided by processor subchain speeds. The algorithm starts with searching for the smallest j so that probing with $W_{1,j}/e_1$ succeeds, and probing with $W_{1,j-1}/e_1$ fails. This means $W_{1,j-1}/e_1 < B_{\text{opt}} \leq W_{1,j}/e_1$, and thus in an optimal solution the probe function will assign the first j tasks to the first processor if it is the bottleneck processor, and the first $j-1$ tasks to the first processor if not. Then the optimal solution value is the minimum of $W_{1,j}/e_1$ and the optimal solution value for partitioning the remaining subchain $\mathcal{T}_{j,N}$ to $P-1$ processors, since any solution with a bottleneck value less than $W_{1,j}/e_1$ will assign only the first $j-1$ tasks to the first processor. Finding the j value requires $\lg N$ probes, and we repeat this search operation for all processors in order. This version of Nicol's algorithm runs in $O(N+(P\lg N)^2)$ time. Fig. 4(a) displays this algorithm.

4.3.2 Nicol's Algorithm with Dynamic Bottleneck-Value Bounding

By keeping the largest B that succeeded and the smallest B that failed, we can improve Nicol's algorithm by eliminating unnecessary probing. Let LB and UB represent the lower bound and upper bound for B_{opt}, respectively. If a processor cannot update LB or UB, that processor does not make any *PROBE* calls. This algorithm, presented in Fig. 4(b), is referred to as *NICOL+*.

In the worst case, a processor makes $O(\lg N)$ *PROBE* calls. But, as we will prove below, the number of probes performed by *NICOL+* cannot exceed $P\lg(1+w_{\max}/(Pe_{\min}w_{\min}))$. This analysis also improves known complexities of homogeneous version of the algorithm. Lemma 4.4 describes an upper bound on the number of probes performed by *NICOL+* algorithm.

```
NICOL (W, E, 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_{i_{b−1},imid}/e_b;
    if PROBE(B) then
      ihigh ← imid;
    else
      ilow ← imid + 1;
  i_b ← ihigh;
  B_b ← W_{i_{b−1},i_b}/e_b;
B_P ← W_{i_{P−1},N}/e_P;
return B_opt ← min_{1≤b≤P}{B_p};
```

(a)

```
NICOL+ (W, E, N, P)
i_0 ← 1;
LB ← B* ← W_{1,N}/E_{1,P};
UB ← LB + w_max × (1/e_min − 1/E_tot);
for b ← 1 to P − 1 do
  ilow ← i_{b−1};  ihigh ← N;
  while ilow < ihigh do
    imid ← (ilow + ihigh)/2;
    B ← W_{i_{b−1},imid}/e_b;
    if LB ≤ B < UB then
      if PROBE(B) then
        ihigh ← imid;
        UB ← B;
      else
        ilow ← imid + 1;
        LB ← B;
    else if B ≥ UB then
      ihigh ← imid;
    else
      ilow ← imid + 1;
  i_b ← ihigh;
  B_b ← W_{i_{b−1},i_b}/e_b;
B_P ← W_{i_{P−1},N}/e_P;
return B_opt ← min_{1≤b≤P}{B_p};
```

(b)

Fig. 4. Nicol's algorithms for heterogeneous systems: (a) Nicol's basic algorithm, (b) Nicol's algorithm (*NICOL+*) with dynamic bottleneck-value bounding.

Lemma 4.4 *The number of probes required by NICOL+ is upper bounded by* $P \lg (1+(UB - LB) / (P w_{\min}))$.

Proof: Consider the first step of the algorithm, where we search for the smallest separator index that makes the first processor the bottleneck processor. We can restrict this search in a range that covers only those indices for which the weight of the first chain will be in the $[LB, UB]$ interval. If there are n_1 tasks in this range, *NICOL+* will require $\lg n_1$ probes. This means that the $[LB, UB]$ interval is narrowed by at least $(n_1 - 1)w_{\min}$ after the first step.

Let k_p be the number of probes by the pth processor. Since k_p probes narrows

the $[LB, UB]$ interval by $\left(2^{k_p} - 1\right) w_{\min}$, we have

$$\left(\left(2^{k_1} - 1\right) + \left(2^{k_2} - 1\right) + \ldots + \left(2^{k_{P-1}} - 1\right)\right) w_{min} \leq UB - LB,$$

and thus $2^{k_1} + 2^{k_2} + \ldots + 2^{k_{P-1}} \leq \dfrac{UB - LB}{w_{\min}} + P - 1$. The corresponding total number of probes is $\sum_{p=1}^{P-1} k_p$, which reaches its maximum when $\sum_{p=1}^{P-1} 2^{k_p}$ is maximum and $k_1 = k_2 = \ldots = k_{P-1} = k$ for some k. In that case,

$$(P-1)2^k \leq \frac{UB - LB}{w_{\min}} + P - 1$$

and thus

$$k \leq \lg\left(1 + \frac{UB - LB}{w_{\min}(P-1)}\right).$$

So, the total number of probes performed by *NICOL+* is upper bounded by:

$$\sum_{p=1}^{P-1} k_p \leq (P-1)k \leq (P-1)\lg\left(1 + \frac{UB - LB}{w_{\min}(P-1)}\right) < P\lg\left(1 + \frac{UB - LB}{w_{\min}P}\right)$$

■

Corollary 4.5 *NICOL+ requires at most* $P\lg(1 + w_{\max}/(Pe_{\min}w_{\min}))$ *probes for heterogeneous, and* $P\lg(1 + w_{\max}/(Pw_{\min}))$ *probes for homogeneous systems.*

NICOL+ runs in $O(N + P^2 \lg N \lg(1 + w_{\max}/(Pe_{\min}w_{\min})))$ time, with the $O(P \lg N)$ cost of a *PROBE* call. In most configurations, $w_{\max}/(e_{\min}w_{\min}P)$ is very small, and is $O(1)$ if $Pe_{\min} = \Omega(w_{\max}/w_{\min})$. In that case, the runtime complexity of *NICOL+* reduces to $O(N + P^2 \lg N)$.

4.3.3 Bidding Algorithm

For heterogeneous systems, the bidding algorithm uses the lower bound given in Eq. 5 for optimal bottleneck value, and gradually increases this lower bound. The bid of each processor $\mathcal{P}_p$, for $p = 1, 2, \ldots, P-1$, is calculated

```
BIDDING (W, N, E, P)
minBid ← W_{1,N}/E_{1,P};
LR-PROBE(W, N, E, P, minBid);
for p ← 1 to P − 1 do
   bids[p] ← W_{s_{p−1}+1,s_p+1}/e_p;
Q ←BUILD-HEAP(P);
repeat
   minP ←EXTRACT-MIN(Q);
   wlast ← W_{s_{P−1}+1,N}/e_P;
   minBid ← bids[minP];
   if minBid < wlast then
      for p ← minP to P − 1 do
         s_p ← BINSEARCH(W, minBid × e_p + W_{1,s_{p−1}});
         previousBid ← bids[p];
         bids[p] ← W_{s_{p−1}+1,s_p}/e_p;
         if bids[p] > previousBid then
            INCREASE-KEY(Q, p);
         else if bids[p] < previousBid then
            DECREASE-KEY(Q, p);
until minBid ≥ wlast;
```

Fig. 5. Bidding algorithm for heterogeneous systems.

as $W_{s_{p-1}+1,s_p+1} \,/\, e_p$, which is equal to the load of $\mathcal{P}_p$ if it also executes the first task of $\mathcal{P}_{p+1}$ in addition to its current load. Then, the algorithm selects the processor with the minimum bid value so that this bid value becomes the next bottleneck value to be considered for feasibility. The processors following the bottleneck processor in the processor chain are processed in order, except the last processor. The separator indices of these processors are adjusted accordingly so that the processors are maximally loaded not to exceed that new bottleneck value. The load of the last processor determines the feasibility of the current bottleneck value. If current bottleneck value is not feasible, the process repeats. Fig. 5 presents the bidding algorithm, which uses a min-priority queue that maintains the processors keyed according to their bid values. In the figure, *BUILD-HEAP*, *EXTRACT-MIN*, *INCREASE-KEY* and *DECREASE-KEY* functions refer to the respective priority queue operations [4].

In the worst case, the bidding algorithm moves P separators for $O(N)$ positions. Choosing a new bottleneck value takes $O(\lg P)$ time using a binary heap implementation of the priority queue. Totally the complexity of the algorithm is $O(NP \lg P)$ in the worst case. Despite this high worst-case complexity, the bidding algorithm is quite fast in practice.

4.3.4 Bisection Algorithm

For heterogeneous systems, the bisection algorithm can use the LB and UB values given in Eqs. 5 and 6. A binary search on this $[LB, UB]$ interval requires $O(\lg(w_{\max}/(\epsilon E_{\text{tot}})))$ probes, thus leading to an $O(\lg(w_{\max}/(\epsilon E_{\text{tot}}))P \lg N)$-time algorithm, where ϵ is the specified accuracy of the algorithm. Fig. 6(a) presents this ϵ-approximation bisection algorithm. We should note that, although the homogenous version of this algorithm becomes an exact algorithm for integer-valued workload arrays by setting $\epsilon = 1$, this is not the case for heterogeneous systems.

We enhance this bisection algorithm to be an exact algorithm for heterogeneous systems by extending the scheme proposed by Pınar and Aykanat [17] for homogenous systems. After each probe, we move lower and upper bounds to realizable bottleneck values, as opposed to the probed value. In heterogeneous systems, realizable bottleneck values are subchain weights divided by appropriate processor speeds. After a successful probe, we decrease UB to the bottleneck value of the partition constructed by the probe, and after a failed probe we increase LB to the bid value as described for the bidding algorithm in Section 4.3.3. Each probe eliminates at least one candidate bottleneck value, and thus the bisection algorithm terminates in a finite number of steps with an optimal solution. Fig. 6(b) displays the exact bisection algorithm.

```
BISECTION (W, N, E, P, ε)
LB ← W_{1,N}/E_{1,P};
UB ← LB + w_max/e_min;
while UB − LB ≥ ε do
   midB ← (UB + LB)/2;
   if PROBE(midB) then
      UB ← midB;
   else
      LB ← midB;
return UB;
```

(a)

```
EXACT-BISECTION (W, N, E, P)
LB ← W_{1,N}/E_{1,P};
UB ← LB + w_max/e_min;
while UB > LB do
   midB ← (UB + LB)/2;
   if LR-PROBE(midB) then
      UB ← min_{1≤p≤P} W_{s_{p−1}+1,s_p}/e_p;
   else
      LB ← min_{1≤p≤P−1} W_{s_{p−1}+1,s_p+1}/e_p;
return UB;
```

(b)

Fig. 6. Bisection algorithms for heterogeneous systems: (a) ϵ-approximation bisection algorithm, (b) Exact bisection algorithm.

5 Chain Partitioning (CP) Problem for Heterogeneous Systems

In this section, we study the problem of partitioning a chain of tasks onto a set of processors, as opposed to a chain of processors. The solution to this problem is not only separators on the task chain, but also processor-to-subchain assignments. Thus, we define a mapping $\mathcal{M}$ as a partition $\Pi = \langle s_0 = 0, s_1, \ldots, s_P = N \rangle$ of the given task chain $\mathcal{T} = \langle t_1, t_2, \ldots t_N \rangle$ with $s_p \leq s_{p+1}$ for $0 \leq p < P$, and a permutation $\langle \pi_1, \pi_2, \ldots, \pi_P \rangle$ of the given set of P processors $\mathcal{P} = \{\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_P\}$. According to this mapping, the pth task subchain $\langle t_{s_{p-1}+1}, \ldots, t_{s_p} \rangle$ is executed on processor $\mathcal{P}_{\pi_p}$. The cost $C(\mathcal{M})$ of a mapping $\mathcal{M}$ is the maximum subchain computation time, determined by the subchain weight and the execution speed of the assigned processor, i.e.,

$$C(\mathcal{M}) = \max_{1 \leq p \leq P} \left\{ \frac{W_{s_{p-1}+1,s_p}}{e_{\pi_p}} \right\}.$$

We will prove that the CP problem is NP-complete. The decision problem for the CP problem for heterogeneous systems is as follows.

Given a chain of tasks $\mathcal{T} = \langle t_1, t_2, \ldots, t_N \rangle$, a weight $w_i \in Z^+$ for each $t_i \in \mathcal{T}$, a set of processors $\mathcal{P} = \{\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_P\}$ with $P < N$, an execution speed $e_p \in Z^+$ for each $\mathcal{P}_p \in \mathcal{P}$, and a bound B, decide if there exists a mapping $\mathcal{M}$ of $\mathcal{T}$ onto $\mathcal{P}$ such that $C(\mathcal{M}) \leq B$.

Theorem 5.1 *The CP problem for heterogeneous systems is NP-complete.*

Proof: We use reduction from the 3-Partition (3P) problem. A pseudo-polynomial transformation suffices, because 3P problem is NP-complete in the strong sense (i.e., there is no pseudo-polynomial time algorithm for the problem unless P=NP). The 3P problem is stated in [8] as follows.

Given a finite set $\mathcal{A}$ of $3m$ elements, a bound $B \in Z^+$, and a cost $c_i \in Z^+$ for each $a_i \in \mathcal{A}$, where $\sum_{a_i \in \mathcal{A}} c_i = mB$ and each c_i satisfies $B/4 < c_i < B/2$, decide if $\mathcal{A}$ can be partitioned into m disjoint sets $S_1, S_2, \ldots, S_m$ such that $\sum_{a_i \in S_p} c_i = B$ for $p = 1, 2, \ldots, m$.

For a given instance of the 3P problem, the corresponding CP problem is constructed as follows.

- The number of tasks N is $m(B+1)-1$. The weight of every $(B+1)$st task is B, (i.e., $w_i = B$ for $i \ mod \ (B+1) = 0$), and the weights of all other tasks are 1.
- The number of processors P is $4m-1$. The first $m-1$ processors have execution speeds of B, (i.e., $e_p = B$ for $p = 1, 2, \ldots, m-1$), and the remaining processors have execution speeds equal to the costs of items in the 3P problem (i.e., $e_p = c_{p-m+1}$ for $p = m, \ldots, 4m-1$).

We claim that there is a solution to the 3P problem if and only if there is a mapping $\mathcal{M}$ with cost $C(\mathcal{M}) = 1$ for the CP problem. The following

observations constitute the basis for our proof.

- The processors with execution speeds of B must be mapped to tasks with weight B to have a solution with cost $C(\mathcal{M}) = 1$, because the execution speeds of all other processors are $\leq B/2$. These processors (tasks) serve as divider processors (tasks).
- The total weight of the chain is $3m + (m-1)B = (B+3)m - B$. The sum of execution speeds of all processors is also $(m-1)B + 3m = (B+3)m - B$. This forces each processor to be assigned a load with value equal to its execution speed to achieve a mapping with cost $C(\mathcal{M}) = 1$.

As noted above, the divider processors should be assigned to the divider tasks. Between two successive divider tasks there is a subchain of B unit-weight tasks with total weight B, which must be assigned to a subset of processors with total execution speed B. Since there are m such subchains, the same grouping of the processors is also valid for grouping c_i values in the 3P problem. Thus the 3P problem can be reduced to the CP problem, proving the CP problem is NP-hard.

The cost of a given mapping can be computed in polynomial time, thus the problem is in NP. Thus we can conclude that the chain partitioning problem for heterogeneous systems is NP-Complete. ■

This complexity shows that we need to resort to heuristics for practical solutions to the CP problem. With the nearly perfect balance results and extremely fast runtimes as we will present in Section 6, CCP algorithms can serve as good heuristics for the CP problem. We tried this approach by finding optimal CCP solutions for randomly ordered processor chains of a CP instance. We observed that the sensitivity to processor ordering is quite low.

You can find a description of these studies in Section 6.3. We also tried improvement techniques, where we swapped processors in the chain to decrease the bottleneck value, but the improvements were modest and could hardly compensate for the increase in runtimes.

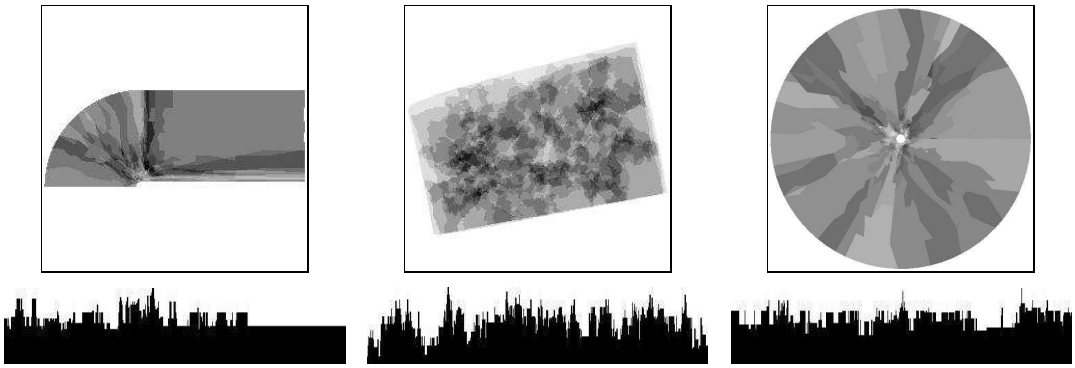
6 Experimental Results

6.1 Experimental Setup

The 1D task arrays used in both CCP and CP experiments were derived from two different applications: image-space-parallel direct volume rendering and row-parallel sparse matrix vector multiplication.

Direct volume rendering experiments are performed on three curvilinear datasets from NASA Ames Research Center [1], namely *Blunt Fin* (blunt), *Combustion Chamber* (comb), and *Oxygen Post* (post). These datasets are processed using the tetrahedralization techniques described in [9] and [18] to produce three-dimensional (3D) unstructured volumetric datasets. The two-dimensional (2D) workload arrays are constructed by projecting 3D volumetric datasets onto 2D screens of resolution 256×256 using the workload criteria of image-space-parallel direct volume rendering algorithm described in [2]. Here, the rendering operations associated with the individual pixels of the screen constitute the computational tasks of the application. The resulting 2D task array is then mapped to a 1D task array using Hilbert space-filling-curve traversal [15]. The workload distributions of the 2D task arrays are visualized in Fig. 7, where darker areas represent more weighted tasks. The histograms at the bottom of the 2D pictures show the weight distributions of the resulting 1D task arrays.

In the sparse matrix experiments, we consider rowwise block partitioning of the matrices obtained from University of Florida Sparse Matrix Collection [5].



a) Blunt Fin b) Combustion Chamber c) Oxygen Post

Fig. 7. Visualization of direct volume rendering dataset workloads. Top: workload distributions of 2D task arrays. Bottom: histogram showing weight distributions of 1D task chains.

In row-parallel matrix vector multiplies, the rows correspond to the tasks to be partitioned, and the number of nonzeros in each row is the weight of the corresponding task. The nonzero distributions of the sparse matrices are shown in Fig. 8. The histograms on the right sides of the visualizations represent the number of nonzeros in each row.

Table 2 displays the properties of the 1D task chains used in our experiments. In the volume rendering dataset, the number of tasks is considerably less than the screen resolution, because zero-weight tasks are omitted. In the sparse matrix dataset, the number of tasks is equal to the number of rows.

In both CCP and CP experiments, $P = 32, 64, 128, 256, 512, 1024$, and 2048-way partitioning of the 1D task arrays were performed. We experimented with different variances of processor speeds, where the processors speeds were chosen uniformly distributed in the 1–4, 1–8, and 1–16 ranges.

In the experiments, the P-way partitioning of a given task chain for a given

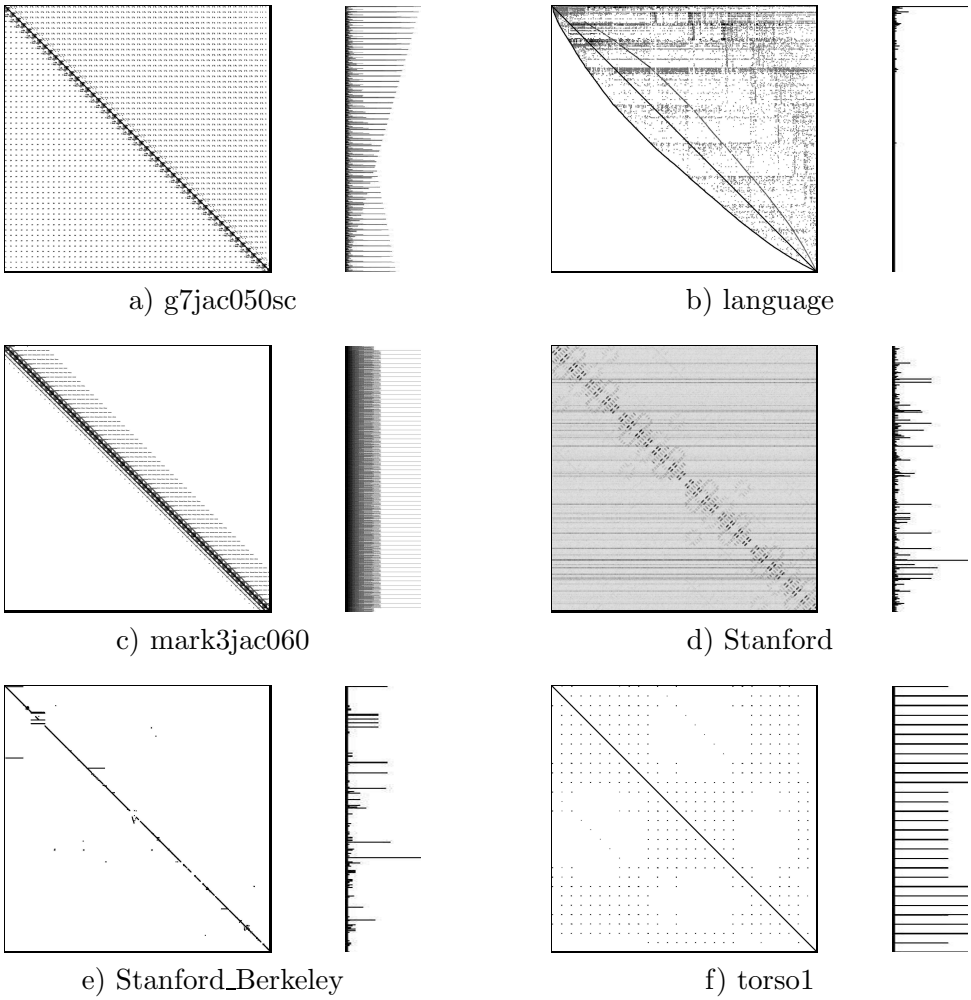


Fig. 8. Visualization of sparse matrix dataset workloads. Left: non-zero distribution of the sparse matrices. Right: histogram showing weight distributions of the 1D task chains.

processor speed range constitutes a partitioning instance. As randomization is used in determining processor speeds, each task chain was partitioned onto 20 different uniformly random processor chains/sets for each speed range, and average performance results are reported for each partitioning instance.

The solution qualities are represented by percent load imbalance values. The percent load imbalance of a partition is computed as $100 \times (B - B^*)/B^*$, where B denotes the bottleneck value of the respective partition.

Table 2
Properties of the test set

Name	No. of tasks N	Workload			
		Total	Per task		
		W_{tot}	w_{avg}	w_{min}	w_{max}
Volume rendering dataset					
blunt	20.6 K	1.9 M	90.95	36	171
comb	32.2 K	2.1 M	64.58	14	149
post	49.0 K	5.4 M	109.73	33	199
Sparse matrix dataset					
g7jac050sc	14.7 K	0.2 M	10.70	2	149
language	399.1 K	1.2 M	3.05	1	11555
mark3jac060	27.4 K	0.2 M	6.22	2	44
Stanford	261.6 K	2.3 M	8.84	1	38606
Stanford_Berkeley	615.4 K	7.6 M	12.32	1	83448
torso1	116.2 K	8.5 M	73.32	9	3263

6.2 CCP Experiments

The proposed CCP algorithms were implemented in the Java language. Tables 3–6 compare the solution qualities of heuristics with respect to those of the optimal partitions obtained by the exact algorithms. In these tables, OPT values refer to the optimal load imbalance values.

Tables 3 and 4 respectively display the percent load imbalance values obtained in mapping the volume rendering and sparse matrix task chains onto processor chains with 1–8 execution speed range. As seen in these two tables, RB performs much better than MP. Out of 63 partitioning instances, RB found better solutions than MP in all but one instance.

As seen in Tables 3 and 4, in general, the quality gap between exact algorithms and heuristics increases with increasing number of processors. For instance, in 2048-way partitioning of the `torso1` matrix, best heuristic finds a solution with 252.44% load imbalance, which means a processor is loaded more than 3.5 times the average load, causing a slowdown as the number of processors increase. An optimal solution however, will have a load imbalance value of

Table 3
Percent load imbalance values for the processor speed range of 1–8 for the volume rendering dataset

CCP instance		Heuristics		OPT
Name	P	RB	MP	
blunt	32	0.27	0.31	0.08
	64	0.62	0.78	0.16
	128	1.35	2.07	0.32
	256	2.94	4.67	0.64
	512	7.27	10.96	1.27
	1024	15.15	21.94	2.83
	2048	36.90	49.23	4.99
comb	32	0.17	0.24	0.06
	64	0.44	0.63	0.11
	128	1.11	1.60	0.23
	256	2.38	3.63	0.45
	512	5.42	7.97	0.92
	1024	12.94	18.24	1.83
	2048	26.61	41.66	3.64
post	32	0.11	0.13	0.03
	64	0.25	0.39	0.07
	128	0.61	0.86	0.13
	256	1.34	2.05	0.27
	512	3.10	4.32	0.54
	1024	6.59	9.21	1.09
	2048	16.21	19.82	2.15

27.61%, providing scalability to thousands of processors.

Tables 5 and 6 display the variation of load balancing performances of heuristics and exact algorithms with varying processor speed ranges for the volume rendering and sparse matrix task chains, respectively. Since RB outperforms MP, only the results for the RB heuristic are displayed in these two tables. The bottom parts of these two tables show the geometric averages of the percent load imbalance values over the number of processors.

As seen in Tables 5 and 6, in general, the performance gap between heuristics and exact algorithms decrease with decreasing processor speed range. However, there exists considerable quality difference between the heuristics and exact algorithms even for the smallest 1–4 speed range.

In constructing the processor chains for the experiments, in addition to the random processor ordering, we also investigated different orderings of the pro-

Table 4
Percent load imbalance values for the processor speed range of 1–8 for the sparse matrix dataset

CCP instance		Heuristics		OPT
Name	P	RB	MP	
g7jac050sc	32	2.21	3.08	0.40
	64	4.88	6.06	0.75
	128	12.21	17.16	1.52
	256	29.06	42.86	3.10
	512	84.54	90.48	6.60
	1024	171.47	289.02	13.59
	2048	261.51	624.91	30.96
language	32	4.58	4.93	0.21
	64	22.60	23.06	0.40
	128	42.06	71.35	1.25
	256	98.08	184.87	35.81
	512	230.49	379.11	171.98
	1024	527.56	1,173.23	443.95
	2048	1,191.77	2,294.59	992.35
mark3jac060	32	0.32	0.54	0.08
	64	0.87	1.01	0.17
	128	2.09	2.75	0.36
	256	5.98	6.90	0.69
	512	15.47	18.17	1.36
	1024	30.23	51.57	2.89
	2048	64.50	127.93	5.92
Stanford	32	12.91	22.85	2.46
	64	42.77	84.14	5.38
	128	110.83	274.42	21.32
	256	204.46	617.98	138.66
	512	435.52	1,058.28	377.97
	1024	1,009.58	2,585.17	855.91
	2048	1,978.18	5,313.99	1,819.63
Stanford_Berkeley	32	10.76	16.91	1.40
	64	49.53	57.69	3.29
	128	89.68	177.24	8.19
	256	160.39	375.68	57.31
	512	315.61	761.14	215.05
	1024	624.98	1,911.41	530.08
	2048	1,248.18	3,949.65	1,165.31
torso1	32	1.74	2.15	0.45
	64	3.82	4.91	0.91
	128	8.75	10.30	1.84
	256	22.46	31.18	3.69
	512	31.68	75.51	7.48
	1024	75.55	75.89	17.86
	2048	252.44	252.44	27.61

cessors having the same speed. In this context, we experimented with the cases where processors having the same speed ordered consecutively, assuming that such processors belong to the same homogenous cluster and hence they are naturally adjacent to each other in the processor chain. We did not observe a considerable sensitivity of the relative load balancing performance between heuristics and exact algorithms to the ordering of processors having the same

Table 5
Percent load imbalance values for different processor speed ranges for the volume rendering dataset

CCP instance		1–4		1–8		1–16	
Name	P	RB	OPT	RB	OPT	RB	OPT
blunt	32	0.21	0.08	0.27	0.08	0.38	0.08
	64	0.39	0.16	0.62	0.16	0.93	0.16
	128	1.06	0.31	1.35	0.32	2.21	0.31
	256	2.19	0.64	2.94	0.64	5.54	0.64
	512	4.62	1.27	7.27	1.27	11.57	1.25
	1024	10.83	2.70	15.15	2.83	26.88	2.61
	2048	22.43	4.93	36.90	4.99	52.25	5.42
comb	32	0.12	0.06	0.17	0.06	0.22	0.06
	64	0.35	0.11	0.44	0.11	0.72	0.11
	128	0.77	0.23	1.11	0.23	1.65	0.23
	256	1.58	0.45	2.38	0.45	3.78	0.45
	512	3.53	0.91	5.42	0.92	9.61	0.91
	1024	7.71	1.82	12.94	1.83	19.75	1.83
	2048	17.53	3.67	26.61	3.64	44.69	3.64
post	32	0.07	0.03	0.11	0.03	0.17	0.03
	64	0.18	0.07	0.25	0.07	0.40	0.07
	128	0.40	0.14	0.61	0.13	0.91	0.13
	256	0.87	0.27	1.34	0.27	2.25	0.27
	512	1.88	0.54	3.10	0.54	4.66	0.54
	1024	4.41	1.09	6.59	1.09	11.42	1.08
	2048	8.87	2.26	16.21	2.15	26.87	2.16
Geometric averages over P							
	32	0.12	0.05	0.17	0.05	0.24	0.05
	64	0.29	0.11	0.41	0.11	0.65	0.11
	128	0.69	0.21	0.97	0.21	1.49	0.21
	256	1.44	0.43	2.11	0.43	3.61	0.43
	512	3.13	0.86	4.96	0.86	8.03	0.85
	1024	7.17	1.75	10.89	1.78	18.23	1.73
	2048	15.16	3.45	25.15	3.39	39.73	3.49

speed.

Tables 7–9 display the execution times of the proposed CCP algorithms on a workstation equipped with a 3 GHz Pentium-IV and 1 GB of memory. In these tables, NC+, BID, and EBS respectively represent the *NICOL+*, *BIDDING*, and *EXACT-BISECTION* algorithms presented in Figs. 4, 5, and 6.

Tables 7 and 8 respectively display the execution times of the CCP algorithms for mapping the volume rendering and sparse matrix task chains onto processor chains with 1–8 execution speed range. In these two tables, relative performance comparison of heuristics shows that MP is slightly faster than RB. Since RB outperforms MP in terms of solution quality as shown in Tables 3 and 4, these results reveal the superiority of RB to MP.

Table 6
Percent load imbalance values for different processor speed ranges for the sparse matrix dataset

CCP instance		1–4		1–8		1–16	
Name	P	RB	OPT	RB	OPT	RB	OPT
g7jac050sc	32	1.22	0.37	2.21	0.40	2.53	0.40
	64	3.53	0.79	4.88	0.75	6.96	0.76
	128	8.94	1.57	12.21	1.52	16.15	1.52
	256	19.62	3.18	29.06	3.10	65.36	3.16
	512	42.24	6.62	84.54	6.60	104.54	6.68
	1024	124.82	14.92	171.47	13.59	162.21	13.56
	2048	307.43	32.67	261.51	30.96	261.88	30.02
language	32	0.36	0.05	4.58	0.21	1.39	0.10
	64	14.09	0.41	22.60	0.40	6.57	0.22
	128	51.77	1.01	42.06	1.25	22.46	1.39
	256	102.08	52.24	98.08	35.81	99.07	27.82
	512	257.83	203.88	230.49	171.98	232.00	156.36
	1024	554.09	506.99	527.56	443.95	519.77	415.09
	2048	1,210.34	1,115.84	1,191.77	992.35	1,088.49	933.33
mark3jac060	32	0.27	0.08	0.32	0.08	0.40	0.08
	64	0.68	0.17	0.87	0.17	1.17	0.16
	128	1.67	0.34	2.09	0.36	3.15	0.35
	256	4.15	0.69	5.98	0.69	10.32	0.69
	512	8.82	1.38	15.47	1.36	22.87	1.40
	1024	20.17	2.85	30.23	2.89	49.73	2.82
	2048	41.26	5.82	64.50	5.92	111.65	5.68
Stanford	32	16.93	2.53	12.91	2.46	20.07	2.61
	64	42.61	5.93	42.77	5.38	48.28	4.88
	128	122.92	32.98	110.83	21.32	90.44	17.79
	256	219.75	167.53	204.46	138.66	215.16	124.62
	512	466.32	434.02	435.52	377.97	427.96	350.50
	1024	1,019.25	966.68	1,009.58	855.91	956.15	805.19
	2048	2,131.61	2,036.65	1,978.18	1,819.63	1,935.93	1,715.91
Stanford_Berkeley	32	7.14	1.29	10.76	1.40	15.32	1.44
	64	26.91	2.51	49.53	3.29	43.39	3.29
	128	85.08	8.96	89.68	8.19	74.51	8.02
	256	191.93	76.34	160.39	57.31	146.90	48.06
	512	331.15	251.99	315.61	215.05	316.54	196.95
	1024	622.85	603.10	624.98	530.08	584.74	496.65
	2048	1,339.44	1,308.36	1,248.18	1,165.31	1,261.41	1,096.94
torso1	32	1.01	0.46	1.74	0.45	1.91	0.45
	64	2.50	0.89	3.82	0.91	4.64	0.88
	128	5.82	1.72	8.75	1.84	14.14	1.85
	256	10.03	3.49	22.46	3.69	22.75	3.73
	512	16.01	5.37	31.68	7.48	65.98	8.26
	1024	40.87	13.12	75.55	17.86	186.70	15.92
	2048	96.14	38.26	252.44	27.61	231.35	32.85
Geometric averages over P							
	32	1.57	0.36	3.04	0.47	3.06	0.42
	64	6.78	0.94	9.59	0.97	8.97	0.86
	128	18.99	2.55	21.30	2.45	21.85	2.41
	256	38.99	13.12	48.21	11.44	60.30	10.51
	512	78.70	32.11	104.64	31.31	130.58	30.67
	1024	181.87	74.04	225.17	72.17	275.55	68.26
	2048	401.91	166.92	481.94	148.31	511.84	146.37

In Tables 7 and 8, relative performances of exact CCP algorithms show that both NICOL+ and EBS are an order of magnitude faster than DP+ and BID for both volume rendering and sparse matrix datasets. As also seen in these

Table 7
Partitioning times (in msecs) for the processor speed range of 1–8 for the volume rendering dataset

CCP instance		Heuristics		Exact algorithms			
Name	P	*RB*	*MP*	DP+	NC+	BID	EBS
blunt	32	0.37	0.36	1	0.58	0.52	0.49
	64	0.39	0.38	1	0.85	0.84	0.66
	128	0.44	0.42	2	1.39	1.91	1.05
	256	0.51	0.47	4	2.42	4.91	1.74
	512	0.64	0.57	14	4.68	13.97	3.28
	1024	0.89	0.76	54	8.67	43.05	6.45
	2048	1.37	1.12	201	15.27	97.54	12.09
comb	32	0.62	0.61	1	0.85	0.80	0.75
	64	0.65	0.64	1	1.15	1.17	0.96
	128	0.69	0.67	2	1.68	2.40	1.37
	256	0.77	0.74	5	2.87	6.04	2.13
	512	0.91	0.84	16	4.84	16.92	3.74
	1024	1.17	1.04	59	9.44	47.19	7.08
	2048	1.68	1.42	230	17.86	130.51	13.30
post	32	1.12	1.11	2	1.36	1.30	1.26
	64	1.15	1.14	2	1.68	1.69	1.46
	128	1.20	1.18	3	2.26	2.91	1.88
	256	1.29	1.26	6	3.52	6.54	2.82
	512	1.45	1.38	16	5.91	16.95	4.51
	1024	1.73	1.59	55	10.36	44.10	7.52
	2048	2.25	1.99	205	20.02	114.60	14.81

two tables, EBS is slightly faster than NICOL+.

It is worth highlighting that for small to medium concurrency, the time it takes EBS and NICOL+ algorithms to find optimal solutions is less than three times the time of the fastest heuristic. More precisely, on overall average, EBS takes only 147% more time than the fastest heuristic for 256-way partitioning. On the other hand, at higher number of processors, the solution qualities of heuristics degrade significantly: on overall average, optimal solutions provide 5.35, 5.47 and 6.00 times better load imbalance values than the best heuristic for 512, 1024 and 2048-way partitionings, respectively. According the these experimental results, we recommend the use of exact CCP algorithms instead of heuristics for heterogeneous systems.

Table 9 displays the variation of running time performances of the CCP al-

Table 8
Partitioning times (in msecs) for the processor speed range of 1–8 for the sparse matrix dataset

CCP instance		Heuristics		Exact algorithms			
Name	P	*RB*	*MP*	DP+	NC+	BID	EBS
g7jac050sc	32	0.31	0.30	1	0.54	0.56	0.46
	64	0.33	0.32	1	0.83	1.08	0.65
	128	0.37	0.35	4	1.31	2.61	1.04
	256	0.44	0.40	13	2.47	7.23	1.80
	512	0.56	0.49	54	4.51	18.88	3.27
	1024	0.80	0.67	234	8.65	48.90	6.07
	2048	1.27	1.02	1730	15.06	100.99	11.96
language	32	7.80	7.80	17	8.19	9.19	8.05
	64	7.84	7.83	22	8.71	14.02	8.47
	128	7.91	7.89	56	9.88	32.63	9.33
	256	8.05	8.01	1999	11.27	8.25	10.63
	512	8.28	8.21	6298	12.38	8.55	11.73
	1024	8.70	8.57	15839	15.96	9.14	16.13
	2048	9.47	9.20	33199	21.82	10.29	20.59
mark3jac060	32	0.47	0.46	1	0.69	0.62	0.60
	64	0.49	0.48	1	0.96	0.94	0.76
	128	0.54	0.52	2	1.48	1.73	1.09
	256	0.62	0.58	7	2.43	3.55	1.78
	512	0.76	0.69	23	4.35	7.95	3.04
	1024	1.01	0.88	90	7.81	19.96	5.95
	2048	1.50	1.25	371	15.91	45.62	11.39
Stanford	32	4.98	4.97	26	5.51	25.10	5.38
	64	5.01	5.00	79	5.99	82.71	5.85
	128	5.08	5.06	841	7.09	437.39	6.67
	256	5.20	5.16	3989	8.42	3022.05	7.80
	512	5.41	5.34	9667	10.77	7524.42	10.08
	1024	5.79	5.65	22472	15.55	16580.61	14.83
	2048	6.48	6.20	49112	25.02	34629.44	23.78
Stanford_Berkeley	32	19.15	19.15	53	19.72	47.08	19.63
	64	19.20	19.18	154	20.60	140.26	20.17
	128	19.27	19.25	558	22.27	460.82	21.16
	256	19.39	19.35	4273	24.34	3722.02	22.24
	512	19.61	19.55	22065	27.82	10742.26	24.53
	1024	20.02	19.89	47607	34.03	22496.33	28.87
	2048	20.78	20.50	100548	46.18	46014.22	37.61
torso1	32	2.12	2.11	5	2.46	4.29	2.38
	64	2.14	2.13	9	2.83	8.80	2.66
	128	2.18	2.16	22	3.55	25.10	3.22
	256	2.26	2.22	83	5.03	76.26	4.45
	512	2.40	2.33	360	7.61	201.48	6.69
	1024	2.68	2.56	1566	13.00	522.08	10.65
	2048	3.24	2.98	6933	23.04	783.22	18.39

gorithms with varying processor speed ranges for the volume rendering and sparse matrix task chains. For a better performance comparison, execution times of the algorithms were normalized with respect to those of the RB heuristic and averages of these normalized values over P are presented in the table. We should mention here that the running time of the RB heuristic does

Table 9
Partitioning time averages (over P) of the exact CCP algorithms normalized with respect to those of the RB heuristic for different processor speed ranges

	1–4				1–8				1–16			
P	DP+	NC+	BID	EBS	DP+	NC+	BID	EBS	DP+	NC+	BID	EBS
Volume rendering dataset												
32	2	1.38	1.28	1.20	2	1.38	1.27	1.21	2	1.40	1.30	1.22
64	2	1.78	1.80	1.44	2	1.76	1.77	1.45	2	1.80	1.88	1.47
128	3	2.42	3.28	1.89	3	2.44	3.33	1.94	3	2.53	3.70	1.96
256	6	3.63	6.94	2.63	6	3.62	7.22	2.73	6	3.65	8.05	2.75
512	15	5.32	15.46	3.79	17	5.45	16.90	3.96	17	5.59	19.07	4.08
1024	43	7.66	32.01	5.21	46	7.77	37.55	5.79	47	7.78	43.59	5.87
2048	114	10.18	53.81	6.95	123	10.15	65.70	7.68	129	10.73	86.03	7.75
Sparse matrix dataset												
32	3	1.25	2.33	1.15	3	1.26	2.30	1.18	3	1.28	2.67	1.17
64	6	1.50	5.34	1.31	6	1.52	5.77	1.34	6	1.54	5.90	1.36
128	34	1.93	24.89	1.59	37	1.93	22.48	1.66	35	2.01	23.37	1.69
256	212	2.58	122.47	2.01	217	2.69	136.83	2.17	219	2.68	147.12	2.12
512	650	3.51	277.96	2.64	649	3.65	340.06	2.89	638	3.75	389.97	2.90
1024	1,422	4.69	565.27	3.51	1,464	4.94	701.30	3.92	1,471	5.07	812.36	3.89
2048	3,136	6.02	1,051.74	4.47	3,243	6.36	1,301.49	5.04	3,234	6.70	1,550.64	5.10

not change with varying processor speed range, as expected. As seen in Table 9, notable performance variation occurs only for the BIDDING algorithm whose running time generally increases with increasing processor speed range.

6.3 CP Experiments

Tables 10 and 11 display the results of our experiments to show the sensitivity of the solution quality of CP problem instances to the processor orderings for the processor speed range of 1–8. In these experiments, we find the optimal CCP solutions for R randomly ordered processor chains of a CP instance, and display geometric averages of the best and average load imbalance values over number of processors. As seen in the tables, for a fixed P, the average imbalance values almost remain the same for different values of R. Although the best imbalance values decrease with increasing R, the decreases are quite small, especially for large P. Moreover, for a fixed R, the relative difference between the best and average imbalance values decreases with increasing P.

Table 10
Geometric averages (over P) of percent load imbalance ratios for R randomly ordered processor chains for the volume rendering dataset with the processor speed range of 1–8

	$R = 10$		$R = 100$		$R = 1000$		$R = 10000$	
P	best	avg	best	avg	best	avg	best	avg
32	0.042	0.050	0.038	0.049	0.036	0.049	0.033	0.048
64	0.097	0.111	0.091	0.112	0.082	0.112	0.077	0.112
128	0.199	0.217	0.189	0.219	0.176	0.218	0.172	0.219
256	0.402	0.427	0.391	0.430	0.377	0.428	0.370	0.428
512	0.852	0.870	0.823	0.870	0.807	0.868	0.791	0.868
1024	1.787	1.849	1.750	1.856	1.727	1.855	1.719	1.855
2048	3.337	3.414	3.245	3.401	3.159	3.402	3.150	3.401

Table 11
Geometric averages (over P) of percent load imbalance ratios for R randomly ordered processor chains for the sparse matrix dataset with the processor speed range of 1–8

	$R = 10$		$R = 100$		$R = 1000$		$R = 10000$	
P	best	avg	best	avg	best	avg	best	avg
32	0.133	0.483	0.104	0.656	0.068	0.588	0.057	0.534
64	0.460	0.906	0.313	0.835	0.257	0.924	0.222	0.935
128	1.304	2.526	1.216	2.484	1.124	2.462	1.020	2.573
256	10.843	11.411	10.291	11.420	10.127	11.427	9.958	11.433
512	31.153	31.694	29.385	31.776	29.078	31.747	28.922	31.735
1024	70.403	71.296	69.160	71.540	68.472	71.530	67.855	71.521
2048	147.792	150.082	146.616	150.360	143.709	150.191	142.917	150.283

These experimental findings show that processor ordering has only a minor effect on solution quality. This is expected since the variance among processor speeds is low, unlike the variance among task weights. Therefore, using an exact CCP algorithm on a number of randomly permuted processor chains can serve as an effective heuristic for the CP problem.

Table 12 displays the results of our experiments to show the sensitivity of the solution quality of CP problem instances to the processor speed range. In these experiments, for each CP instance, we find the optimal CCP solutions for $R = 10000$ randomly ordered processor chains, and display the best load imbalance value. As seen in Table 12, we do not observe a considerable sensitivity of the solution quality of the CP problem instances to the procesor speed range. Notable sensitivity is observed only for the `language`, `Stanford`,

Table 12
Best percent load imbalance ratios for $R = 10000$ randomly ordered processor chains with different processor speed ranges

Volume rendering dataset					Sparse matrix dataset				
CCP instance					CCP instance				
Name	P	1–4	1–8	1–16	Name	P	1–4	1–8	1–16
blunt	32	0.029	0.053	0.051	g7jac050sc	32	0.154	0.146	0.092
	64	0.125	0.134	0.117		64	0.390	0.366	0.371
	128	0.207	0.267	0.241		128	1.003	1.016	0.994
	256	0.628	0.559	0.528		256	2.402	2.226	2.439
	512	1.055	1.193	1.157		512	5.493	5.497	5.297
	1024	2.300	2.992	2.543		1024	13.187	11.727	11.829
	2048	5.000	4.554	4.938		2048	28.115	28.269	26.974
comb	32	0.037	0.034	0.034	language	32	0.004	0.003	0.004
	64	0.076	0.075	0.079		64	0.011	0.010	0.013
	128	0.183	0.180	0.179		128	0.052	0.050	0.040
	256	0.377	0.387	0.380		256	55.560	34.304	24.151
	512	0.818	0.814	0.812		512	206.845	168.371	151.509
	1024	1.707	1.662	1.694		1024	511.078	443.036	407.589
	2048	3.561	3.508	3.522		2048	1,122.157	977.521	915.654
post	32	0.020	0.020	0.020	mark3jac060	32	0.033	0.039	0.041
	64	0.048	0.046	0.047		64	0.095	0.104	0.103
	128	0.109	0.107	0.108		128	0.245	0.232	0.245
	256	0.233	0.234	0.230		256	0.536	0.547	0.544
	512	0.466	0.510	0.479		512	1.173	1.154	1.215
	1024	0.948	1.022	0.988		1024	2.501	2.474	2.504
	2048	2.240	1.957	2.043		2048	5.516	5.255	5.225
					Stanford	32	0.239	0.127	0.128
						64	0.960	0.889	0.525
						128	35.643	12.897	14.879
						256	173.373	136.019	118.176
						512	439.233	371.620	341.987
						1024	973.874	854.300	792.008
						2048	2,047.748	1,793.575	1,684.852
					Stanford_Berkeley	32	0.047	0.063	0.073
						64	0.740	0.554	0.666
						128	2.831	3.307	2.843
						256	80.192	55.570	43.809
						512	255.431	210.865	191.333
						1024	607.837	529.020	487.961
						2048	1,315.674	1,148.137	1,076.473
					torso1	32	0.315	0.229	0.307
						64	0.771	0.639	0.677
						128	1.112	2.240	1.538
						256	1.890	3.087	3.004
						512	4.859	6.996	8.198
						1024	12.046	16.806	15.439
						2048	38.975	28.495	31.079

and `Stanford_Berkeley` sparse matrix datasets, which have high task weight variation (i.e., large $w_{\max}/w_{\text{avg}}$ value). In these datasets, load imbalance values decrease with increasing processor speed range, which possibly because the adverse effect of tasks with large weight on load imbalance can be more easily resolved by mapping them to the processors with larger execution speed.

7 Conclusions

We studied the problem of one-dimensional partitioning of nonuniform workload arrays with optimal load balancing for heterogeneous systems. We investigated two cases: chain-on-chain partitioning, where a chain of tasks is partitioned onto a chain of processors; and chain partitioning, where the task chain is partitioned onto a set of processors (i.e., permutation of the processors is allowed). We showed that chain-on-chain partitioning algorithms for homogenous systems can be revised to solve this partitioning problem for heterogeneous systems, without altering computational complexities of these algorithms. We proved that the chain partitioning problem is NP-complete, and empirically showed that exact CCP algorithms can serve as an effective heuristic, for the CP problem. Our experiments proved the effectiveness of our techniques, as the exact algorithms work much better than heuristics, and balanced work decompositions can be achieved even for high numbers of processors.

8 Availability

The algorithms proposed in this work are implemented in Java language and made publicly available at http://www.cs.bilkent.edu.tr/˜tabak/hetccp/.

References

[1] NASA advanced supercomputing division (NAS) dataset archive, http://www.nas.nasa.gov/Research/Datasets/datasets.html.

[2] B. B. Cambazoglu, C. Aykanat, Hypergraph-partitioning-based remapping

models for image-space-parallel direct volume rendering of unstructured grids, IEEE Transactions on Parallel and Distributed Systems 18 (1) (2007) 3–16.

[3] H.-A. Choi, B. Narahari, Algorithms for mapping and partitioning chain structured parallel computations, in: International Conference on Parallel Processing, 1991.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, Introduction to Algorithms, The MIT Press and McGraw-Hill Book Company, 1989.

[5] T. Davis, University of Florida Sparse Matrix Collection, http://www.cise.ufl.edu/research/sparse/matrices, NA Digest, vol. 97, no. 23 (June 1997).

[6] K. D. Devine, E. G. Boman, R. T. Heaphy, B. A. Hendrickson, J. D. Teresco, J. Faik, J. E. Flaherty, L. G. Gervasio, New challanges in dynamic load balancing, Appl. Numer. Math. 52 (2-3) (2005) 133–152.

[7] K. D. Devine, B. Hendrickson, E. G. Boman, M. M. S. John, C. Vaughan, Zoltan: a dynamic load-balancing library for parallel applications – user's guide, Tech. Rep. SAND99-1377, Sandia National Laboratories (1999).

[8] M. R. Garey, D. S. Johnson, Computers and Intractability; A Guide to the Theory of NP-Completeness, W. H. Freeman & Co., New York, NY, USA, 1990.

[9] M. P. Garrity, Raytracing irregular volume data, in: VVS '90: Proceedings of the 1990 workshop on Volume visualization, ACM Press, New York, NY, USA, 1990.

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

[11] V. J. Leung, E. M. Arkin, M. A. Bender, D. Bunde, J. Johnston, A. Lal, J. S. B. Mitchell, C. Phillips, S. S. Seiden, Processor allocation on Cplant: Achieving general processor locality using one-dimensional allocation strategies, in: CLUSTER '02: Proceedings of the IEEE International Conference on Cluster Computing, IEEE Computer Society, Washington, DC, USA, 2002.

[12] F. Manne, B. Olstad, Efficient partitioning of sequences, IEEE Transactions on Computers 44 (11) (1995) 1322–1326.

[13] S. Miguet, J.-M. Pierson, Heuristics for 1D rectilinear partitioning as a low cost and high quality answer to dynamic load balancing, in: HPCN Europe '97: Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking, Springer-Verlag, London, UK, 1997.

[14] D. M. Nicol, Rectilinear partitioning of irregular data parallel computations, Journal of Parallel and Distributed Computing 23 (2) (1994) 119–134.

[15] J. R. Pilkington, S. B. Baden, Dynamic partitioning of non-uniform structured workloads with spacefilling curves, IEEE Transactions on Parallel and Distributed Systems 7 (3) (1996) 288–300.

[16] A. Pınar, C. Aykanat, Sparse matrix decomposition with optimal load balancing, in: HIPC '97: Proceedings of the Fourth International Conference on High-Performance Computing, IEEE Computer Society, Washington, DC, USA, 1997.

[17] A. Pinar, C. Aykanat, Fast optimal load balancing algorithms for 1D partitioning, Journal of Parallel and Distributed Computing 64 (8) (2004) 974–996.

[18] P. Shirley, A. Tuchman, A polygonal approximation to direct scalar volume rendering, in: VVS '90: Proceedings of the 1990 workshop on Volume visualization, ACM Press, New York, NY, USA, 1990.