J. Parallel Distrib. Comput. 68 (2008) 1473–1486

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

One-dimensional partitioning for heterogeneous systems: Theory and practice☆

Ali Pınar[a], E. Kartal Tabak[b], Cevdet Aykanat[b,*]

[a] High Performance Computing Research Department. Lawrence Berkeley National Laboratory, United States
[b] Department of Computer Engineering, Bilkent University, Turkey

ARTICLE INFO

Article history:
Received 8 February 2007
Received in revised form
3 July 2008
Accepted 12 July 2008
Available online 25 July 2008

Keywords:
Parallel computing
One-dimensional partitioning
Load balancing
Chain-on-chain partitioning
Dynamic programming
Parametric search

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 solution times remain comparable.

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 [5,6,9,15]. Similarly, processors can be mapped to a 1D array so that communication is relatively faster between close processors in this processor chain [10]. 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 power. We have surveyed the rich literature on this problem, proposed novel methods as well as improvements 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

☆ This work is partially supported by The Scientific and Technological Research Council of Turkey (TÜBİTAK) under projects EEEAG-105E065 and EEEAG-106E069.
* Corresponding author.
E-mail addresses: apinar@lbl.gov (A. Pınar), tabak@cs.bilkent.edu.tr (E. Kartal Tabak), aykanat@cs.bilkent.edu.tr (C. Aykanat).

0743-7315/$ – see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2008.07.005

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 comparable. On average, optimal solutions provide 4.9 and 8.7 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 [7]. 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 processor 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 one. Pınar and Aykanat [17] proved that RB produces partitions with bottleneck values no greater than $B^* + w_{\max}(P-1)/P$.

Miguet and Pierson [12] 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 [12] 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 [2], and Manne and Olstad [11] 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.

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., $\lg x = \log_2 x$

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 in only $O(P \lg N)$ 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 [16,17] 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.

```
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, s_{p-1}, N, w);
```

Fig. 1. Heterogeneous CCP heuristics.

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 as close to the ratio $E_{p,q}/E_{q+1,r}$ 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 s_q in the prefix-summed $\mathcal{W}$ array.

We also propose a generalization of Miguet and Pierson's heuristic, *MP* [12]. *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

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

4.2. Dynamic programming

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

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

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 [2] 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., $j_i^p \geq j_{i-1}^p$. 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., $B_j^{p-1} \geq W_{j+1,i}/e_p$. 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:

$$B_i^p = \max \left\{ B_{j_i^p}^{p-1}, \frac{W_{j_i^p+1,i}}{e_p} \right\},$$

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

```
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
        B[p, i] ← W_{j,i}/e_p;
        j ← j − 1;
      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
        B[p, i] ← W_{j,i}/e_p;
        j ← j − 1;
      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.

```
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.

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.

In the homogenous case, Manne and Olstad [11] 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.

Alternatively, 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 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

```
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.

m, such that $W_{1,m} \le w$. Similarly, in *RL-PROBE*, *BINSEARCH* $(\mathcal{W}, w)$ searches $\mathcal{W}$ for the smallest index m such that $W_{1,m} \ge w$.

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* $\Pi^1 = \langle {h^1}_0, {h^1}_1, \ldots, {h^1}_P\rangle$, $\Pi^2 = \langle {l^2}_0, {l^2}_1, \ldots, {l^2}_P\rangle$, $\Pi^3 = \langle {l^3}_0, {l^3}_1, \ldots, {l^3}_P\rangle$ *and* $\Pi^4 = \langle {h^4}_0, {h^4}_1, \ldots, {h^4}_P\rangle$ *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* $SL_p \le s_p \le SH_p$ *for all* $1 \le p \le P$, *where* $SL_p = \max\{l_p^2, l_p^3\}$ *and* $SH_p = \min\{h_p^1, h_p^4\}$.

```
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.

```
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.

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 task subchain $\mathcal{T}_{j,N}$ to the processor subchain $\mathcal{P}_{2,P}$, 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.

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

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

and thus $2^{k_1} + 2^{k_2} + \cdots + 2^{k_{P-1}} \leq \frac{UB-LB}{w_{\min}} + P - 1$. 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 $k_1 = k_2 = \cdots = k_{P-1} = k$ for some k. In that case,

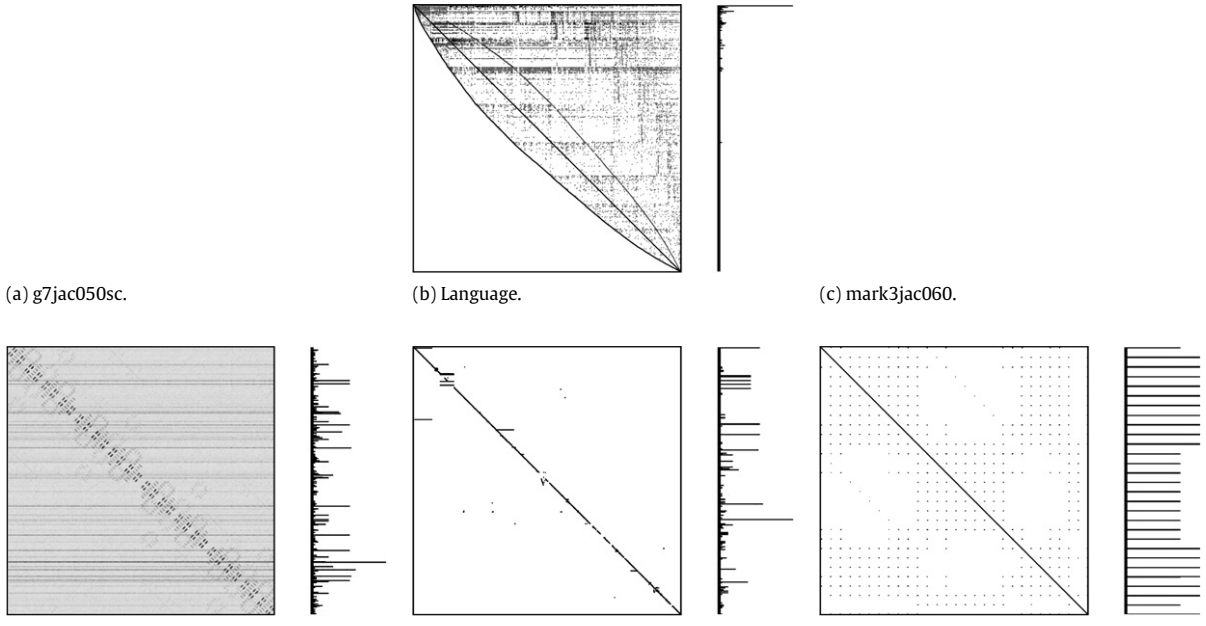
$$(P-1)2^k \leq \frac{UB - LB}{w_{\min}} + P - 1$$

and thus

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

(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: histograms showing weight distributions of 1D task chains.



(a) g7jac050sc. (b) Language. (c) mark3jac060.

(d) Stanford. (e) Stanford Berkeley. (f) torso1.

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

Table 2
Properties of the test set

Name	No. of tasks N	Workload			
		Total	Per task		
		W_{tot}	$w_{\text{avg.}}$	$w_{\min}$	$w_{\max}$
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	11 555
mark3jac060	27.4 K	0.2 M	6.22	2	44
Stanford	261.6 K	2.3 M	8.84	1	38 606
Stanford_Berkeley	615.4 K	7.6 M	12.32	1	83 448
torso1	116.2 K	8.5 M	73.32	9	3 263

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

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

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 $Pe_{\min} =$