Improving the Performance of Independent Task Assignment Heuristics MinMin, MaxMin and Sufferage

E. Kartal Tabak, B. Barla Cambazoglu, Cevdet Aykanat

Abstract—MinMin, MaxMin, and Sufferage are constructive heuristics that are widely and successfully used in assigning independent tasks to processors in heterogeneous computing systems. All three heuristics are known to run in $O(KN^2)$ time in assigning N tasks to K processors. In this paper, we propose an algorithmic improvement that asymptotically decreases the running time complexity of MinMin to $O(KN \log N)$ without affecting its solution quality. Furthermore, we combine the newly proposed MinMin algorithm with MaxMin as well as Sufferage, obtaining two hybrid algorithms. The motivation behind the former hybrid algorithm is to address the drawback of MaxMin in solving problem instances with highly skewed cost distributions while also improving the running time performance of MaxMin. The latter hybrid algorithm improves the running time performance of Sufferage without degrading its solution quality. The proposed algorithms are easy to implement and we illustrate them through detailed pseudocodes. The experimental results over a large number of real-life datasets show that the proposed fast MinMin algorithm and the proposed hybrid algorithms perform significantly better than their traditional counterparts as well as more recent state-of-the-art assignment heuristics. For the large datasets used in the experiments, MinMin, MaxMin, and Sufferage, as well as recent state-of-the-art heuristics, require days, weeks, or even months to produce a solution, whereas all of the proposed algorithms produce solutions within only two or three minutes.

Index Terms—Parallel processors, heterogeneous systems, load balancing, independent task assignment, MinMin, MaxMin, Sufferage, constructive heuristics

1 Introduction

The focus of this work is on the independent task assignment problem, which often arises in applications related to heterogeneous computing systems. In this problem, we have a set $\mathcal{T} = \{T_1, T_2, \ldots, T_N\}$ of N independent tasks, a set $\mathcal{P} = \{P_1, P_2, \ldots, P_K\}$ of K heterogeneous processors, and an expected-time-to-compute matrix $E = \{x_{i,k}\}_{N \times K}$, where $x_{i,k}$ denotes the expected execution cost of task T_i on processor P_k. The objective is to find a task-to-processor assignment that results in the minimum turnaround time (makespan). In other words, the objective is to minimize the load of the maximally loaded (bottleneck) processor. This problem is known to be NP-complete [1].

The `MinMin` heuristic is first introduced in [1] and since then it is used many times for solving the independent task assignment problem, which commonly emerges in the context of heterogeneous systems [1]–[13]. `MinMin` is a constructive heuristic with some desirable properties. It is free of parameters that require tuning and is easy to implement. Moreover, it is reported to produce "high quality" solutions. Since its first proposal, the running time of the `MinMin` algorithm is reported to be $O(KN^2)$ in the literature [1], [4], [5], [8]–[13]. Despite its success, the quadratic running time complexity of the heuristic prevents its use in problem instances where the number of tasks to be assigned is very large. Recently, the `MinMin` algorithm is parallelized to enable the application of the algorithm to large datasets [14]. This parallel version runs in $O(N^2K/P + N^2 + N \log P)$ time, where P denotes the number of homogenous processors used in parallelization of the `MinMin` algorithm (P may be different than K).

We believe that the computational complexity of `MinMin` is overlooked in the parallel and distributed computing literature. This mainly stems from the task-oriented view of `MinMin`, constituting a lower bound of $\Omega(KN^2)$ on the running time. In this paper, we propose an $O(KN \log N)$-time algorithm that improves this quadratic lower bound by switching from the task-oriented view to a processor-oriented view. The proposed `MinMin` algorithm, which is referred to herein as `MinMin+`, attains exactly the same solution quality as `MinMin` without degrading the ease of implementation. The results of our experiments over a wide range of problem instances indicate that `MinMin+` runs several orders of magnitude faster than `MinMin`. For a large dataset that contains about 2.5 million tasks, `MinMin` finds a 16-way assignment

- *E. K. Tabak and C. Aykanat are with the Department of Computer Engineering, Bilkent University, Turkey, 06800. E-mail: tabak,aykanat@cs.bilkent.edu.tr*
- *B. B. Cambazoglu is with Yahoo! Research, Barcelona, Spain. E-mail: barla@yahoo-inc.com*

Digital Object Indentifier 10.1109/TPDS.2013.107

in about 22 days, whereas MinMin+ finds the same assignment in about a minute.

Two other well-known constructive heuristics used for solving the independent task assignment problem are MaxMin (MaxMin) [1], [2], [8], [15] and Sufferage (Suff) [9]. These heuristics differ from MinMin in the task selection policy adopted during the task-to-processor assignment process. In this work, we propose improvements over these two heuristics as well. We combine MaxMin with MinMin+ as well as Suff with MinMin+ to obtain the hybrid algorithms MaxMin+ and Suff+, respectively.

The assignment of large tasks to their favorite processors[1] is important to obtain a good makespan, especially in skewed datasets. Although the MaxMin heuristic assigns the largest task to its favorite processor, its inherent mechanism is likely to fail to assign remaining large tasks to their favorite processors. The motivation behind MaxMin+ is to address this drawback of MaxMin in solving problem instances with highly skewed cost distributions while also improving the running time performance of MaxMin.

Suff is reported to be among the algorithms that yield high-quality solutions [9], [16], [17]. Despite its success, the quadratic running time prevents the application of this heuristic to large datasets. The motivation behind Suff+ is to improve the running time performance of Suff without degrading the solution quality.

Although both MaxMin+ and Suff+ are, in the worst case, still $O(KN^2)$-time algorithms, our experimental results show that they run considerably faster than the traditional MaxMin and Suff heuristics, respectively. The experimental results also indicate that MaxMin+ finds considerably better solutions than MaxMin while Suff+ finds slightly better solutions than Suff, on average.

MinMin is also used as a component in the design of more complex algorithms [2], [18], [19]. Genetic algorithm (GA) [2], [18] is a typical example of such complex algorithms. In this work, we also demonstrate that the running time performance of the GA algorithm can be significantly improved simply by replacing MinMin with MinMin+, without affecting the original solution quality at all.

The rest of the paper is organized as follows. Table 1 summarizes the notation used throughout the paper. Section 2 describes the existing algorithms. The proposed MinMin+, MaxMin+, Suff+ algorithms, and the improved GA algorithm are discussed in Section 3. In Section 4, our experimental setup and results are presented. The paper is concluded in Section 5.

2 Existing Algorithms

MinMin: The MinMin heuristic [1] proceeds in N iterations. At each iteration, a previously unassigned task is selected and assigned to a processor. The selected task is removed from further consideration in the remaining iterations. The task-to-processor assignment in each iteration is decided based on a two-step procedure. In the first step, MinMin computes the minimum completion time (MCT) of each unassigned task over the processors to find the best processor, which can complete the processing of that task at earliest time. This decision is made taking into account the current loads of processors (e_k) and the execution time of the task on each processor ($x_{i,k}$). In the second step, MinMin selects the task with the minimum MCT among all unassigned tasks and assigns the task to its best processor found in the first step. Due to the task selection policy adopted in the second step, MinMin favors the assignment of tasks with lower costs in earlier iterations, and hence the assignment of tasks with higher costs are usually performed during the later iterations. The two-step selection algorithm is provided in Algorithm 1. An $O(KN^2)$-time algorithm for MinMin is given in Algorithm 2.

MaxMin: MaxMin [1], [2], [8], [15] differs from MinMin in the task selection policy adopted in the second step of the task-to-processor assignment procedure. Unlike MinMin, which selects the task with the minimum MCT, MaxMin selects the task with the maximum MCT and then assigns it to the best processor found in the first step (Algorithm 3). Due to this task selection policy, MaxMin performs the assignment of tasks with higher costs in earlier iterations. The algorithm for MaxMin is presented in Algorithm 4.

RASA: In [20], the drawbacks of MaxMin and MinMin are analyzed and a hybrid algorithm, referred to as RASA, is proposed. RASA alternates between MaxMin and MinMin in its iterations. In particular,

1. A processor P_k is said to be a favorite processor for a task T_i if the expected cost of T_i is minimum on P_k, i.e., $k=\text{argmin}_\ell\, x_{i,\ell}$.

TABLE 1
The notation used throughout the paper

Notation	Explanation
A	task-to-processor assignment vector
E	expected-time-compute matrix
G	number of chromosomes used in the GA algorithm
H	number of iterations of the GA algorithm
K	number of processors
M	makespan
M^*	ideal makespan
N	number of tasks
$\mathcal{P}$	set of processors
P_k	kth processor
Q_k	priority queue of P_k in the MinMin+ algorithm
R	machine heterogeneity constant
$\mathcal{T}$	set of tasks
T_i	ith task
U	a set of tasks
e_k	current load of processor P_k
i, j	indices that refer to tasks
k, ℓ	indices that refer to processors
m	number of MaxMin-based assignments in MaxMin+
$x_{i,k}$	computation cost of task T_i on processor P_k
α	exponent constant for power-law distribution
γ	relative cost for the RC algorithm

Algorithm 1 MinMinSelect(U, e, x, K)

$min' \leftarrow \infty$
for all $i \in U$ **do**
  $min \leftarrow \infty$
  for $k \leftarrow 1$ **to** K **do**
    if $e_k + x_{i,k} < min$ **then**
      $min \leftarrow e_k + x_{i,k}$
      $kmin \leftarrow k$
  if $min < min'$ **then**
    $min' \leftarrow e_{kmin} + x_{i,kmin}$
    $k' \leftarrow kmin$
    $i' \leftarrow i$
return $\langle i', k' \rangle$

Algorithm 2 MinMin(x, K, N)

$U \leftarrow \{1, 2, \ldots, N\}$
for $k \leftarrow 1$ **to** K **do**
  $e_k \leftarrow 0$
while U is not empty **do**
  $\langle i', k' \rangle \leftarrow$ MinMinSelect(U, e, x, K)
  $A[i'] \leftarrow k'$
  $e_{k'} \leftarrow e_{k'} + x_{i',k'}$
  $U \leftarrow U - \{i'\}$
return A

Algorithm 3 MaxMinSelect(U, e, x, K)

$max \leftarrow 0$
for all $i \in U$ **do**
  $min \leftarrow \infty$
  for $k \leftarrow 1$ **to** K **do**
    if $e_k + x_{i,k} < min$ **then**
      $min \leftarrow e_k + x_{i,k}$
      $kmin \leftarrow k$
  if $min > max$ **then**
    $max \leftarrow e_{kmin} + x_{i,kmin}$
    $k' \leftarrow kmin$
    $i' \leftarrow i$
return $\langle i', k' \rangle$

Algorithm 4 MaxMin(x, K, N)

$U \leftarrow \{1, 2, \ldots, N\}$
for $k \leftarrow 1$ **to** K **do**
  $e_k \leftarrow 0$
while U is not empty **do**
  $\langle i', k' \rangle \leftarrow$ MaxMinSelect(U, e, x, K)
  $A[i'] \leftarrow k'$
  $e_{k'} \leftarrow e_{k'} + x_{i',k'}$
  $U \leftarrow U - \{i'\}$
return A

Algorithm 5 Rasa(x, K, N)

$U \leftarrow \{1, 2, \ldots, N\}$
for $k \leftarrow 1$ **to** K **do**
  $e_k \leftarrow 0$
for $r \leftarrow 1$ **to** N **do**
  if r is odd **then**
    $\langle i', k' \rangle \leftarrow$ MaxMinSelect(U, e, x, K)
  else
    $\langle i', k' \rangle \leftarrow$ MinMinSelect(U, e, x, K)
  $A[i'] \leftarrow k'$
  $e_{k'} \leftarrow e_{k'} + x_{i',k'}$
  $U \leftarrow U - \{i'\}$
return A

Algorithm 6 SuffSelect(U, x, K, N)

$sufferage' \leftarrow 0$
for all $i \in U$ **do**
  $min \leftarrow \infty$
  $second\_min \leftarrow \infty$
  for $k \leftarrow 1$ **to** K **do**
    if $e_k + x_{i,k} < min$ **then**
      $second\_min \leftarrow min$
      $min \leftarrow e_k + x_{i,k}$
      $kmin \leftarrow k$
    else if $e_k + x_{i,k} < second\_min$ **then**
      $second\_min \leftarrow e_k + x_{i,k}$
  $sufferage \leftarrow second\_min - min$
  if $sufferage > sufferage'$ **then**
    $sufferage' \leftarrow sufferage$
    $k' \leftarrow kmin$
    $i' \leftarrow i$
return $\langle i', k' \rangle$

Algorithm 7 Suff(x, K, N)

$U \leftarrow \{1, 2, \ldots, N\}$
for $k \leftarrow 1$ **to** K **do**
  $e_k \leftarrow 0$
while U is not empty **do**
  $\langle i', k' \rangle \leftarrow$ SuffSelect(U, e, x, K)
  $A[i'] \leftarrow k'$
  $e_{k'} \leftarrow e_{k'} + x_{i',k'}$
  $U \leftarrow U - \{i'\}$
return A

`MaxMin` is used in odd rounds while `MinMin` is used in even rounds. The `RASA` algorithm, which runs in $O(KN^2)$ time, is displayed in Algorithm 5.

Sufferage: The main difference between `Suff` [9] and `MinMin` is the task selection policy. In the first step of the process, `Suff` computes the second MCT value in addition to the MCT value for each task. In the second step, the sufferage value, which is defined as the difference between the MCT and the second MCT values of a task, is taken into account. `Suff` selects the task with the largest sufferage and assigns it to the best processor found in the first step. The algorithm for `Suff` is presented in Algorithm 7.

Relative Cost (RC): `RC` [17] is a constructive heuristic similar to `MinMin`, but it uses a different selection criterion which does not lead to a bias between small tasks and large tasks. At each iteration of the algorithm, `RC` selects the task with the lowest relative cost, which is calculated as

$$\gamma = \left(\frac{\min_k \{x_{i,k} + e_k\}}{\mathrm{avg}_k \{x_{i,k} + e_k\}} \right) + \left(\frac{x_{i,k^*(i)}}{\mathrm{avg}_k \{x_{i,k}\}} \right)^{\xi}, \quad (1)$$

where $k^*(i) = \mathrm{argmin}_k \{x_{i,k} + e_k\}$ in the current iteration. The selected task is assigned to processor $k^*(i)$. ξ is a parameter in the $[0, 1]$ range and is used to control the effects of the first and second terms in Eq. 1. `RC` is reported as a high-quality algorithm and runs in $O(KN^2)$ time. The `RC` algorithm is displayed in Algorithm 8.

Algorithm 8 RC(x, K, N)

$U \leftarrow \{1, 2, \ldots, N\}$
for $k \leftarrow 1$ **to** K **do**
    $e_k \leftarrow 0$
for $i \leftarrow 1$ **to** N **do**
    $avg \leftarrow \text{avg}_k\{x_{i,k}\}$
    for $k \leftarrow 1$ **to** K **do**
        $\gamma_s[i,k] \leftarrow x_{i,k}/avg$
for $j \leftarrow 1$ **to** N **do**
    $\gamma\_min \leftarrow \infty$
    for $i \leftarrow 1$ **to** N **do**
        $avg \leftarrow \text{avg}_k\{e_k + x_{i,k}\}$
        $k \leftarrow \text{argmin}_k\{e_k + x_{i,k}\}$
        $min \leftarrow x_{i,k}$
        $\gamma \leftarrow min/avg \times \gamma_s[i,k]^{\xi}$
        if $\gamma < \gamma\_min$ **then**
            $\gamma\_min \leftarrow \gamma$
            $i' \leftarrow i$
            $k' \leftarrow k$
    $A[i'] \leftarrow k'$
    $e_{k'} \leftarrow e_{k'} + x_{i',k'}$
    $U \leftarrow U - \{i'\}$
return A

Genetic Algorithm (GA): `GA` [2], [18] is an example of more complex algorithms that use `MinMin` as a component. `GA` uses `MinMin` as an initial chromosome and improves the solution of `MinMin` using genetic algorithm techniques. In this approach, each chromosome represents a different task-to-processor assignment. Assuming G chromosomes, one of the chromosomes is initially populated with `MinMin` while the remaining G–1 chromosomes are populated with random assignments. Maintaining the best assignment (elitism) guarantees that the solution quality of `GA` is not worse than the quality of `MinMin`. Crossover is implemented as a single random cross on the paired chromosomes. Mutation is defined as reassigning a random task to a random processor. The initial population runs in $O(KN^2 + G\log G + NG)$ time. Each iteration of `GA` runs in $O(NG + G^2)$ time. Hence, `GA` runs in $O(KN^2 + HNG + HG^2)$ time, where H is the number of iterations.

3 Proposed Algorithms

3.1 MinMin+

The high running time complexity of the `MinMin` algorithm stems from the $O(KN)$-time cost that is incurred while computing the MCT values for every unassigned task and processor pair. Note that the MCT values and the best processor of an unassigned task may change at each iteration of the loop in Algorithm 2. This is because the $e_k + x_{i,k}$ value associated with an unassigned task T_i and processor P_k may change as the e_k values are updated throughout the iterations. Without any loss of generality, let us assume that a task is assigned to a processor P_k in the previous iteration. This assignment increases the e_k value. Therefore, in the next iteration, the $e_k + x_{i,k}$ values for all unassigned tasks need to be recomputed for processor P_k. This task-oriented view of the `MinMin` algorithm forms a lower bound of $\Omega(KN^2)$ on the running time of the algorithm.

In this work, we demonstrate that the above-mentioned quadratic lower bound can be avoided by switching from the task-oriented view to a processor-oriented view. To this end, we propose a novel algorithm, referred to as `MinMin+`. In this algorithm, the MCT values that are associated with each processor are separately maintained, instead of being unnecessarily recomputed at each iteration for every unassigned task. In particular, we use a priority queue Q_k for each processor P_k to maintain the completion times of all tasks on that processor. More specifically, each task T_i is maintained in K different priority queues, keyed by their $x_{i,k}$ values. Each priority queue Q_k supports the MIN, DELETE, and BUILD operations. MIN(Q_k) is a query operation that returns the id of the unassigned task that has the minimum completion time on processor P_k. DELETE(Q_k, i) is an update operation that removes task T_i from Q_k. The BUILD(k) operation initializes the data structures. We also maintain a boolean array F of size N. Each array element $F[i]$ indicates whether task T_i is yet assigned to a processor or not. Initially, we set all $F[i]$ values to FALSE since no task is assigned to a processor at the beginning.

The proposed `MinMin+` algorithm is given in Algorithm 9. The `MinMin+Init` function (Algorithm 10) is called in the first line of the algorithm to perform the necessary initializations. The following main loop (lines 2–8) performs N iterations, assigning a task to a processor at each iteration. The `MinMin+Select` function (Algorithm 11) invokes a MIN(Q_k) operation on each priority queue Q_k to find a candidate task for processor P_k. The candidate task T_i selected for processor P_k is effectively the task that will increase the current completion time of P_k (i.e., e_k) by the smallest amount if T_i is assigned to P_k. For each processor P_k, the execution time of the candidate task T_i on P_k is added to e_k to compute the updated e_k value for P_k if T_i is assigned to P_k. A running-min operation performed over these K updated e_k values gives the minimum MCT value (min) for the current iteration as well as the task-to-processor assignment (i', k') that achieves this minimum MCT value. At the end of each iteration of the main loop, the assigned task $T_{i'}$ is deleted from all priority queues (lines 7 and 8).

For the implementation of the priority queue, we have considered two alternatives: binary heap and sorted linear array. Although both implementations lead to the same worst-case running time complexity, our empirical results indicate that the sorted linear array implementation yields significantly lower execution times compared to the binary-heap implemen-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

Algorithm 9 MINMIN+(x, K, N)

$\langle e, F, Q\rangle \leftarrow$ MINMIN+INIT(x, K)
for $j \leftarrow 1$ **to** N **do**
  $\langle i', k'\rangle \leftarrow$ MINMIN+SELECT(Q, e, K)
  $A[i'] \leftarrow k'$
  $e_{k'} \leftarrow e_{k'} + x_{i',k'}$
  $F[i'] \leftarrow$ TRUE
  for $k \leftarrow 1$ **to** K **do**
    DELETE(Q_k, i')
return A

Algorithm 10 MINMIN+INIT(x, K)

for $k \leftarrow 1$ **to** K **do**
  $e_k \leftarrow 0$
for $i \leftarrow 1$ **to** N **do**
  $F[i] \leftarrow$ FALSE
for $k \leftarrow 1$ **to** K **do**
  $Q_k \leftarrow$ BUILD(k)
  $\triangleright$ Q_k contains records of $\langle i, x_{i,k}\rangle$.
return $\langle e, F, Q\rangle$

Algorithm 11 MINMIN+SELECT(Q, e, K)

$min \leftarrow \infty$
for $k \leftarrow 1$ **to** K **do**
  $\langle i, x\rangle \leftarrow$ MIN(Q_k)
  if $e_k + x < min$ **then**
    $min \leftarrow e_k + x$
    $k' \leftarrow k$
    $i' \leftarrow i$
return $\langle i', k'\rangle$

tation. Hence, in what follows, we present the running time analysis of the `MinMin+` algorithm only for the sorted linear array implementation.

In the sorted linear array implementation, for each processor P_k, we maintain a linear array Q_k, which contains N tuples of the form $\langle i, x_{i,k}\rangle$. The BUILD operation sorts the tuples in Q_k in increasing order of the $x_{i,k}$ values. For each Q_k, we maintain an index b_k, indicating the unassigned task that currently has the smallest completion time on processor P_k. The BUILD operation initializes the b_k value to 1. The overall running time of the BUILD operation is $O(N \log N)$. The MIN(Q_k) operation can be realized in $O(1)$ time, simply by returning the task id of the b_k-th tuple in Q_k. After a task T_i is assigned to a processor, it is deleted by setting $F[i]$ to TRUE and running a DELETE(Q_k) operation on every Q_k. Since $Q_k[1, \ldots, b_k - 1]$ contains the tasks that are already assigned, the DELETE(Q_k) operation can be realized by advancing the b_k index on Q_k until an unassigned task is encountered. Although the worst-case running time of an individual DELETE(Q_k) operation is $O(N)$, the amortized cost of DELETE(Q_k) operation is $O(1)$. This is because N DELETE operations performed on Q_k can lead to at most N increments on b_k. This simple yet efficient implementation of the DELETE operation makes the sorted linear array implementation preferable over the binary heap implementation. The proposed `MinMin+` algorithm involves K BUILD(k), $K \times N$ MIN(Q_k), and $K \times N$ DELETE(Q_k) operations. Hence, the overall running time complexity is $O(KN \log N + KN + KN) = O(KN \log N)$.

3.2 MaxMin+

In some problem instances, the task sizes follow a power-law distribution, i.e., there are a small number of very large tasks and a very large number of small tasks. In such cases, the assignment of large tasks can have a significant impact on the load of the most heavily loaded processor (i.e., makespan) and determine the resulting solution quality. In case of the `MinMin` heuristic, due to the adopted task selection policy, smaller tasks are assigned in earlier iterations, delaying the assignment of larger tasks to later iterations. The solution quality obtained in the earlier iterations is likely to deteriorate due to the late assignment of very large tasks. In case of the `MaxMin` heuristic, the larger tasks are assigned in earlier iterations, but not necessarily to their favorite processors. To demonstrate the issue, let us consider the first few iterations of `MaxMin`. The first iteration assigns the largest task to its favorite processor. Let us assume that the second largest task has the same favorite processor as the largest task. In the second iteration, the task selection policy of `MaxMin` prevents the assignment of the second largest task to its favorite processor. In the next iteration, the third largest task loses the flexibility of being assigned to the favorite processors of the largest two tasks and so on.

To alleviate the above-mentioned drawbacks of the `MinMin` and `MaxMin` heuristics, we combine these two heuristics under a hybrid heuristic, which we refer to as `MaxMin+`. Like `MinMin` and `MaxMin`, the `MaxMin+` heuristic involves a main loop that assigns a selected task to a processor at each iteration. Within an iteration, the heuristic first computes a task-to-processor assignment according to the `MinMin` heuristic. The computed assignment is realized only if it does not lead to an increase in the makespan of the previous iteration. If, however, the computed assignment increases the makespan, the task-to-processor assignment is recomputed according to the `MaxMin` heuristic.

The `MaxMin+` algorithm is presented in Algorithm 12, using the asymptotically faster `MinMin+` algorithm proposed in Section 3.1 instead of the standard `MinMin` algorithm. In the algorithm, `MinMin+Init` (line 3) performs the necessary initializations as in `MinMin+`. Line 5 computes the task-to-processor assignment according to `MinMin+`. The if statement at line 6 checks whether the computed assignment increases the current makespan. Line 7 computes the task-to-processor assignment according to `MaxMin`.

As described in Section 2, the `RASA` heuristic also combines `MinMin` and `MaxMin`. In `RASA`, `MinMin` is

Algorithm 12 MAXMIN+(x, K, N)

```
1: U ← {1, 2, ..., N}
2: makespan ← 0
3: ⟨e, F, Q⟩ ← MINMIN+INIT(x, K)
4: while U is not empty do
5:     ⟨i', k'⟩ ← MINMIN+SELECT(Q, e, K)
6:     if e_{k'} + x_{i',k'} > makespan then
7:         ⟨i', k'⟩ ← MAXMINSELECT(U, e, x, K)
8:         makespan ← e_{k'} + x_{i',k'}
9:     A[i'] ← k'
10:    e_{k'} ← e_{k'} + x_{i',k'}
11:    U ← U − {i'}
12:    F[i'] ← TRUE
13:    for k ← 1 to K do
14:        DELETE(Q_k, i')
15: return A
```

executed in odd-numbered iterations while `MaxMin` is executed at even-numbered iterations. The proposed `MaxMin+` heuristic differs from `RASA` in that the choice between `MinMin` and `MaxMin` at each iteration is made in an adaptive manner, considering the current processor loads. The experimental results reported in Section 4 shows the success of this adaptive policy with respect to the policy adopted in `RASA`.

The running time of `MaxMin+` depends on the frequency of `MaxMin`-based assignments. In practice, `MaxMin+` is expected to run slower than `MinMin+` since line 7 is executed when the assignment is performed according to `MaxMin`. `MaxMin+` is expected to run faster than `MaxMin`. The performance of `MaxMin+` depends on the ratio of the `MaxMin`-based assignments to the total number of assignments.

In the following lemmas, we describe the theoretical behavior of the `MaxMin+` algorithm and find the expected number of `MaxMin`-based assignments for some statistical distributions. We present the proofs of our lemmas and theorems in the appendix not to interrupt the flow of the paper.

Lemma 3.1: `MaxMin+` makes one `MaxMin`-based assignment in the best case, and makes N `MaxMin`-based assignments in the worst case.

Lemma 3.2: `MaxMin+` runs in $O(KN \log N + KNm)$ time, where m is the number of `MaxMin`-based assignments.

In general, the number of `MaxMin`-based assignments is expected to decrease with both increasing heterogeneity and increasing K. The former expectation is due to the higher variation in task execution costs with increasing heterogeneity, which generally results in an increase in the ratio between the weights of larger tasks and smaller tasks. Hence, a `MaxMin`-based assignment of a large task will be amortized by a large number of `MinMin`-based assignments of smaller tasks. The latter expectation is due to the extra processing power provided by the additional processors, which results in more room for the `MinMin` selections until the makespan changes. The experimental results reported in Section 4.2.1 support this expectation.

We present the following theorems for the special and possibly the worst case of $K = 2$ homogenous processors.

Theorem 3.1: For $K = 2$ homogenous processors, if the task weights of a dataset have a power-law distribution with the probability density function $f(x) = Cx^{-\alpha}$ for $x > x_{\min}$ and $\alpha > 2$, the expected number of `MaxMin`-based assignments is $\left(\frac{1}{2}\right)^{\frac{\alpha-1}{\alpha-2}} N$.

Note that, if α gets closer to 2, the number of `MaxMin`-based assignments decreases.

Theorem 3.2: For $K = 2$ homogenous processors, if the task weights of a dataset are uniformly distributed between $x_{\min}$ and $x_{\max}$, the expected number of `MaxMin`-based assignments is $\frac{2r-\sqrt{2r^2+2}}{2r-2}N$, where $r = x_{\max}/x_{\min}$.

Corollary 3.1: For $K = 2$ homogenous processors, if the task weights of a dataset are uniformly distributed between $x_{\min}$ and $x_{\max}$, the expected number of `MaxMin`-based assignments is greater than $0.28N$.

According to Theorem 3.1, for a skewed dataset with a typical α value of 2.33 [21], the expected upper bound on the number of `MaxMin`-based assignments to be performed by `MaxMin+` is $0.061N$. That is, at most 6.1% of the assignments will be expensive `MaxMin`-based assignments. This approximately corresponds to a speedup of 16 with respect to `MaxMin`.

According to Theorem 3.2, for a uniform dataset with $x_{\max}/x_{\min} = 2$, the expected number of `MaxMin`-based assignments to be performed by `MaxMin+` is 41% of the total number of assignments. These theoretical findings show that the relative speedup of `MaxMin+` over `MaxMin` is expected to be much higher on skewed datasets. The experimental results given in Section 4.2.1 validate this expectation.

3.3 Suff+

Despite the success of `Suff` in producing high quality solutions [9], [16], [17], its quadratic running time prevents the application of `Suff` to large datasets. To make `Suff` applicable to large datasets, we combine it with `MinMin+`, under a new heuristic referred to as `Suff+`. The main idea behind the `Suff+` heuristic is to perform critical assignment decisions by `Suff` so that the solution quality is not significantly degraded and perform non-critical assignment decisions by the fast `MinMin+` algorithm. With this approach, we expect a considerable decrease in the execution time of `Suff` with a small potential degradation in the solution quality.

In `Suff+`, the criticality of an assignment decision is determined by the effect of a possible `MinMin+` assignment on the makespan. At each assignment iteration, `Suff+` first computes a task-to-processor assignment according to `MinMin+`. The computed assignment is realized only if it does not lead to an

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

Algorithm 13 Suff+(x, K, N)

1: $U \leftarrow \{1, 2, \ldots, N\}$
2: $makespan \leftarrow 0$
3: $\langle e, F, Q\rangle \leftarrow$ MinMin+Init(x, K)
4: **while** U is not empty **do**
5: $\quad\langle i', k'\rangle \leftarrow$ MinMin+Select(Q, e, K)
6: $\quad$**if** $e_{k'} + x_{i',k'} > makespan$ **then**
7: $\qquad\langle i', k'\rangle \leftarrow$ SuffSelect(U, e, x, K)
8: $\qquad makespan \leftarrow e_{k'} + x_{i',k'}$
9: $\quad A[i'] \leftarrow k'$
10: $\quad e_{k'} \leftarrow e_{k'} + x_{i',k'}$
11: $\quad U \leftarrow U - \{i'\}$
12: $\quad F[i'] \leftarrow$ TRUE
13: $\quad$**for** $k \leftarrow 1$ **to** K **do**
14: $\qquad$Delete(Q_k, i')
15: **return** A

increase in the makespan of the previous iteration. If, however, the `MinMin+`-based assignment increases the makespan, the task-to-processor assignment is recomputed according to the `Suff` heuristic.

The algorithm for `Suff+` is provided in Algorithm 13. As in `MaxMin+`, the `MinMin+Init` function (line 3) performs the necessary initializations. Line 5 computes the assignment according to `MinMin+`. The comparison operation at line 6 checks whether makespan will change if the computed assignment is used. Line 7 computes the task-to-processor assignment according to `Suff`.

3.4 GA+

Traditionally, the `MinMin` heuristic is used as a submodule in more complex task assignment algorithms. As mentioned in Section 2, `GA` is such an algorithm since it uses `MinMin` to find an initial solution. In the literature, `GA` is reported as a slow algorithm, compared to $O(KN^2)$ algorithms such as `MaxMin` and `RC` [2], [17].

Herein, we consider `GA` to illustrate the impact of using `MinMin+` instead of `MinMin` on the performance of complex task assignment algorithms. Incorporation of the `MinMin+` heuristic into `GA` leads to an asymptotically faster algorithm, which we refer to as `GA+`. This combination retains the original solution quality of `GA`. `GA+` runs in $O(KN \log N + HNG + HG^2)$ time, making it run much faster than $O(KN^2)$ algorithms and rendering it practical even for large datasets.

4 Experimental Results

4.1 Datasets

The datasets used in the experiments belong to different application areas: social-network analysis, distributed web crawling, image-space-parallel direct volume rendering (DVR), and row-parallel sparse matrix vector multiplication (SpMxV). In these contexts, the independent task assignment problem arises in load balancing of parallel/distributed applications. These datasets are displayed in Table 2.

TABLE 2
Properties of the datasets

Dataset	N	Task weights Max.	Avg.	α
Social networks				
coauthorship	725,344	672	6.81	3.43 ± 0.04
commonJob	241,233	10,270	7.08	2.30 ± 0.01
Distributed web crawling				
ClueWeb-B	799,115	$6.1{\times}10^6$	61.56	2.23 ± 0.00
ClueWeb-A	2,483,726	$1.5{\times}10^9$	1010.50	2.16 ± 0.00
Image-space-parallel direct volume rendering (DVR)				
blunt	20,611	171	90.95	6.51 ± 0.29
comb	32,238	149	64.58	3.84 ± 0.22
Row-parallel sparse matrix vector multiplication (SpMxV)				
barrier2-1	113,076	7,031	33.65	3.78 ± 0.20
language	399,130	11,555	3.05	2.59 ± 0.01
k3plates	11,107	58	34.12	6.42 ± 0.92
big	13,209	12	6.92	7.42 ± 1.57
olafu	16,146	89	62.87	6.29 ± 0.81
mark3jac060	27,449	44	6.22	3.06 ± 0.16
Zhao1	33,861	6	4.92	4.60 ± 2.31
dawson5	51,537	33	19.61	3.02 ± 0.63
epb3	84,617	6	5.48	1.79 ± 0.79
lung2	109,460	8	4.50	2.33 ± 0.26
hood	220,542	77	48.83	6.56 ± 2.16
Lin	256,000	7	6.90	1.16 ± 0.10
pre2	659,033	628	9.04	2.50 ± 0.07

(*) Rows in gray indicate skewed datasets.

Our social network datasets (`coauthorship` and `commonJob`) are in the form of sparse graphs. In `coauthorship`, each vertex represents an author and an edge represents the coauthorship relation between two authors. In `commonJob`, each vertex represents an employee and there is an edge between two vertices if the respective employees have ever worked in the same company. The `coauthorship` and `commonJob` datasets are obtained from DBLP[2] and LinkedIn[3], respectively. In both of these graphs, a vertex represents a task to be processed. The degree of a vertex corresponds to the cost of executing the task.

In distributed web crawling datasets (`ClueWeb-A` and `ClueWeb-B`), the tasks represent the web sites and the processors represent the crawlers that will download the pages in the web sites. The weight of a task is set to the number of pages in the respective web site. The `ClueWeb-A` and `ClueWeb-B` datasets, which are obtained from the ClueWeb-09 collection [22], are the largest two datasets among our datasets.

In row-parallel DVR datasets (`blunt` and `comb`), rendering each rectangular pixel block of an image forms a separate task. The weight of a task is set to the expected number of ray-face intersections to be performed while rendering the pixels in the respective pixel block [23]. `blunt` (blunt fin) and `comb` (combustion) are two curvilinear datasets obtained from the NASA Ames Research Center [24].

2. http://www.informatik.uni-trier.de/~ley/db/
3. http://www.linkedin.com/

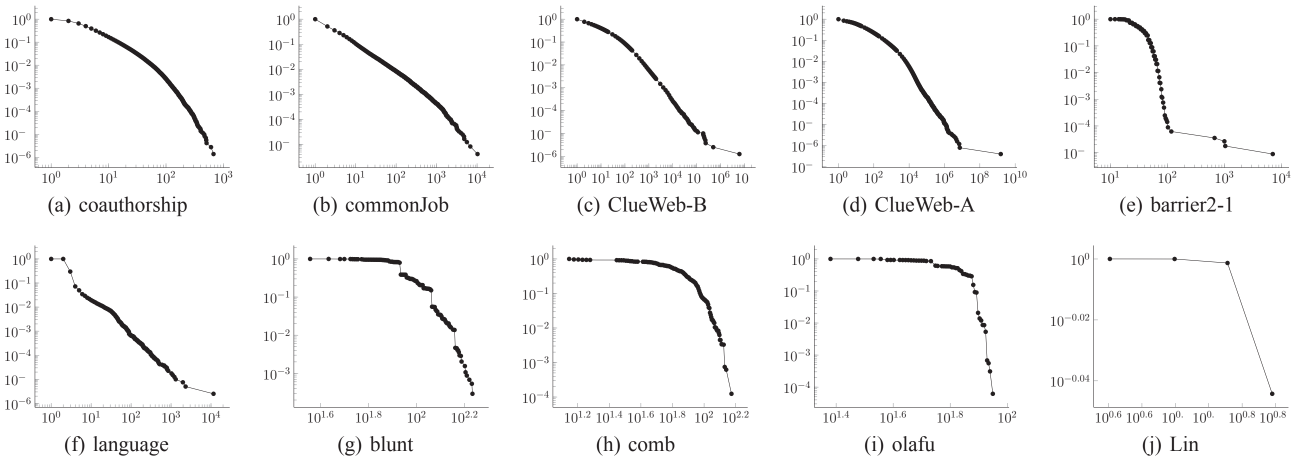


Fig. 1. Log-log plots of the cumulative density distribution of task weights for skewed datasets ((a)–(f)) and non-skewed datasets ((g)–(j)). x-axis: weights of tasks, y-axis: cumulative density distribution, i.e., $P(X \geq x)$.

In row-parallel SpMxV datasets, each task corresponds to computing the inner product of a distinct row of the sparse matrix with a dense column vector. The weight of a task is equal to the number of nonzeros in the respective row. We use 13 sparse matrices that are selected from the University of Florida sparse matrix collection [25].

For the distributed web crawling datasets, the ETC value of each task on each crawler is calculated using the techniques described in [26]. For the other datasets, the ETC matrices are constructed using the high machine heterogeneity method discussed in [27]. For each $x_{i,k}$, we multiply the weight of the corresponding task with a random integer in the range $[1 \ldots R]$, where R is the machine heterogeneity constant. Following [27], we selected R as 100 to reflect high machine heterogeneity. For all datasets, the ETC matrices are generated for $K \in \{4, 8, 16, 24, 32\}$ processors. Each dataset and K value combination forms a different assignment instance for our experiments. Since we have 19 datasets and five different K values, we have a total of 95 assignment instances.

In Table 2, the Max and Avg columns display the maximum and average task weights, respectively. The α column shows the exponent constant of the power-law distribution $p(w) = Cw^{-\alpha}$ of task weights, together with their error margins. The α values are computed by using the linear least squares method on log-log distributions of the datasets and are used here to identify the datasets with power-law distributions. The datasets that have α values with low error margin and high max/avg ratio are good candidates to have power-law distributions. In this respect, `coauthorship`, `commonJob`, `ClueWeb-B`, `ClueWeb-A`, `barrier2-1`, and `language` datasets are considered to have a power-law distribution. In the remaining tables, the rows are colored in gray to indicate skewed datasets.

Fig. 1 displays the log-log plots of the cumulative density distribution of task weights for the datasets. In the figure, the plots for skewed and non-skewed datasets are presented in (a)–(f) and (g)–(j), respectively. Note that the plots for only four datasets out of 13 SpMxV datasets are displayed in Fig. 1. The complete list of plots can be found in Appendix.

4.2 Performance Analysis

All of the algorithms are implemented in Java programming language. All experiments were carried out on a Linux workstation equipped with six 2100-MHz quad-core CPUs and 132 GB of memory.

The load balancing quality of the assignment algorithms are compared according to the percent load imbalance ratio defined as

$$\%\mathrm{LI} = 100 \times \frac{M - M^*}{M^*}, \quad (2)$$

where M denotes the makespan of an assignment produced by an algorithm and M^* denotes the ideal makespan for the given assignment instance. M^* is computed as

$$M^* = \frac{W^*_{\mathrm{tot}}}{K} = \frac{\sum_i \min_k \{x_{i,k}\}}{K}, \quad (3)$$

where W^*_{tot} is the execution time obtained when the tasks are assigned to their favorite processor. This value forms a rather loose lower bound for the makespan. The optimal makespan is potentially greater than M^*.

Tables 3–6 display the load imbalance values for 4-, 8-, 16-, 24-, and 32-way assignments obtained by the existing (baseline) and proposed heuristics for different types of datasets. Table 7 displays load imbalance averages for different K values over all datasets. In these tables, we display the results of `MinMin` and `MinMin+` in the same column, since these heuristics attain the same results. The results of `GA` and `GA+` are displayed in the same column due to the same reason.

Tables 8–11 display the running times of the heuristics for different types of datasets. Table 12 displays running time averages for different K values over all

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

TABLE 3
Percent load imbalance values for social network datasets

		Original heuristics				Proposed heuristics			
Dataset	K	MxM	Suff	RC	RASA	MM MM+	MxM+	Suff+	GA GA+
coauthorship	4	204.94	0.08	**0.01**	163.35	0.10	0.13	0.02	0.04
	8	280.50	0.58	**0.02**	229.68	0.13	0.07	0.06	0.07
	16	316.25	1.86	**0.10**	263.86	0.48	0.11	0.24	0.30
	24	315.95	2.43	0.22	266.97	0.76	**0.11**	0.34	0.43
	32	310.82	2.66	**0.19**	262.12	1.69	0.30	0.80	0.96
commonJob	4	163.19	0.71	1.07	143.40	1.87	0.72	**0.53**	0.81
	8	218.67	2.51	**0.63**	192.47	8.97	1.46	1.86	3.75
	16	239.99	5.26	**3.28**	212.25	18.92	3.87	10.14	9.24
	24	235.61	5.62	**5.08**	213.56	23.90	8.77	17.85	14.50
	32	227.71	6.97	**4.58**	204.58	37.28	14.51	16.81	23.42

TABLE 4
Percent load imbalance values for distributed web crawling datasets

		Original heuristics				Proposed heuristics			
Dataset	K	MxM	Suff	RC	RASA	MM MM+	MxM+	Suff+	GA GA+
ClueWeb-B	4	81.49	22.28	19.91	80.06	41.82	**17.24**	18.52	37.62
	8	175.88	102.35	103.61	173.05	168.72	99.63	**99.02**	159.99
	16	230.77	162.19	161.96	227.42	319.82	160.48	**155.16**	306.18
	24	286.10	**222.08**	224.38	282.29	476.91	230.77	230.77	458.82
	32	**323.97**	**323.97**	324.50	**323.97**	607.80	**323.97**	**323.97**	589.19
ClueWeb-A	4	**172.02**	**172.02**	173.35	**172.02**	205.18	**172.02**	**172.02**	204.82
	8	**436.41**	**436.41**	436.85	**436.41**	482.96	**436.41**	**436.41**	482.31
	16	**802.88**	**802.88**	802.92	**802.88**	891.27	**802.88**	**802.88**	889.61
	24	**1286.95**	**1286.95**	1286.98	**1286.95**	1393.57	**1286.95**	**1286.95**	1388.59
	32	**1763.49**	**1763.49**	1763.53	**1763.49**	1868.91	**1763.49**	**1763.49**	1862.57

TABLE 5
Percent load imbalance values for parallel DVR datasets

		Original heuristics				Proposed heuristics			
Dataset	K	MxM	Suff	RC	RASA	MM MM+	MxM+	Suff+	GA GA+
blunt	4	185.24	0.10	0.06	102.04	0.11	1.12	0.07	**0.04**
	8	253.94	0.65	0.27	155.03	0.29	0.65	0.23	**0.12**
	16	276.43	1.86	0.52	175.31	0.60	0.60	0.54	**0.34**
	24	275.48	2.25	1.37	176.10	1.02	0.78	1.02	**0.47**
	32	269.72	2.52	2.07	172.12	1.42	1.07	1.18	**0.74**
comb	4	187.83	0.10	0.05	116.36	0.08	0.67	0.09	**0.03**
	8	252.82	0.74	0.12	169.19	0.16	0.48	0.17	**0.08**
	16	278.85	1.83	0.43	195.78	0.49	0.31	0.35	**0.24**
	24	276.05	2.56	0.86	191.02	0.85	0.66	0.83	**0.47**
	32	271.01	2.81	1.40	189.24	0.94	0.70	0.92	**0.55**

datasets. These averages are obtained by normalizing the running time values with those attained by the MinMin+ heuristic.

In Tables 6 and 11, the performance results for row-parallel SpMxV datasets are presented only for four sample sparse matrices out of 13 matrices. The complete results for this particular type of datasets are reported in Appendix. The average performance results displayed in Tables 7 and 12, however, are computed by considering the performance results of all datasets.

In Tables 3–6, the bold value(s) in each row indicate the best solution(s) in terms of load balancing performance for the respective assignment instance. In all tables, the MinMin, MinMin+, MaxMin, and MaxMin+ heuristics are abbreviated as MM, MM+, MxM, and MxM+, respectively.

4.2.1 Comparison with Traditional Counterparts

In this subsection, we discuss the performance of each proposed heuristic against its traditional counterpart.

MinMin+ versus MinMin: As mentioned in Section 3.1, MinMin+ finds exactly the same solutions as MinMin. However, MinMin+ is several orders of magnitude faster than MinMin in all assignment instances. On average, MinMin+ is 5603-, 3703-, 4192-, 3214-, and 2947-times faster than MinMin in 4-, 8-, 16-, 24-, and 32-way assignments, respectively.

As expected, the speedup of MinMin+ over MinMin increases with increasing number of tasks. For the 16-way assignment of the largest dataset ClueWeb-A, which contains about 2.5 million tasks, MinMin finds a solution in about 22 days while MinMin+ finds the same solution in about a minute, i.e., MinMin+ runs

TABLE 6
Percent load imbalance values for parallel SpMxV datasets

		Original heuristics				Proposed heuristics			
Dataset	K	MxM	Suff	RC	RASA	MM MM+	MxM+	Suff+	GA GA+
`barrier2-1`	4	202.22	0.12	**0.01**	119.77	0.89	0.46	0.04	0.15
	8	278.87	0.59	**0.03**	180.31	2.25	0.26	0.09	0.51
	16	310.18	1.38	**0.09**	208.49	0.36	0.19	0.12	0.18
	24	311.27	2.10	0.30	210.86	7.56	0.28	**0.21**	2.42
	32	303.48	2.25	0.30	207.41	1.39	**0.26**	0.30	0.77
`language`	4	198.73	0.38	0.03	121.62	1.68	0.27	**0.03**	0.63
	8	286.72	2.59	0.33	186.64	7.30	**0.27**	0.44	3.23
	16	315.71	3.98	1.31	214.60	27.98	**1.15**	2.01	22.87
	24	319.59	2.51	**0.57**	219.26	5.72	0.57	4.49	3.20
	32	308.00	4.37	**1.79**	212.24	58.74	4.49	3.70	51.44
`olafu`	4	184.48	0.16	0.11	104.11	0.09	1.04	0.11	**0.04**
	8	247.81	0.80	0.34	152.60	0.27	0.58	0.28	**0.12**
	16	269.75	1.78	0.88	172.10	0.81	0.69	0.63	**0.37**
	24	267.79	2.79	1.47	172.49	0.96	0.90	1.22	**0.57**
	32	258.30	2.98	2.30	171.92	1.14	1.15	1.23	**0.76**
`Lin`	4	218.44	0.01	**0.01**	115.61	**0.01**	0.41	0.01	**0.01**
	8	324.60	0.13	0.01	193.65	0.01	0.29	0.03	**0.01**
	16	361.38	0.62	0.05	223.68	0.05	0.17	0.05	**0.02**
	24	358.59	1.01	0.07	223.51	0.07	0.14	0.06	**0.04**
	32	349.33	1.12	0.09	219.01	0.10	0.13	0.09	**0.09**

TABLE 7
Averages of percent load imbalance values over all datasets

		Original heuristics				Proposed heuristics			
Dataset	K	MxM	Suff	RC	RASA	MM MM+	MxM+	Suff+	GA GA+
Skewed	4	170.43	32.60	32.40	133.37	41.92	31.81	31.86	40.68
	8	279.51	90.84	90.25	233.09	111.72	89.68	89.65	108.31
	16	369.30	162.92	161.61	321.58	209.80	161.45	161.76	204.73
	24	459.25	253.61	252.92	413.32	318.07	254.57	256.77	311.33
	32	539.58	350.62	349.15	495.64	429.30	351.17	351.51	421.39
Non-skewed	4	195.60	0.10	0.04	113.24	0.08	0.74	0.06	0.04
	8	270.44	0.60	0.16	170.93	0.20	0.45	0.18	0.09
	16	298.36	1.62	0.52	195.76	0.58	0.46	0.44	0.30
	24	295.70	2.16	1.15	195.37	1.04	0.64	0.73	0.56
	32	288.46	2.39	1.16	192.02	1.22	0.79	1.14	0.72

about 31,400 times faster than `MinMin`.

`MaxMin+` versus `MaxMin`: `MaxMin+` finds drastically better solutions than `MaxMin` in all assignment instances, except for the 32-way assignment of `ClueWeb-B` and the assignment instances of `ClueWeb-A`, where both heuristics find solutions with the same makespan. The averages displayed in Table 7 demonstrate the large quality difference between `MaxMin+` and `MaxMin`. On average, `MaxMin+` attains average load imbalance values of 177.74% and 0.62% compared to 363.61% and 269.71% of `MaxMin`, for skewed and non-skewed datasets, respectively. Moreover, `MaxMin+` is several orders of magnitude faster than `MaxMin` in all assignment instances. On average, `MaxMin+` runs 6917- and 404-times faster than `MaxMin` for skewed and non-skewed datasets, respectively. Note that the performance gaps between `MaxMin+` and `MaxMin` in load balancing and running time are much higher in non-skewed datasets compared to skewed datasets in favor of `MaxMin+`. The former is expected since `MaxMin` is highly tuned for skewed datasets and fails to find good solutions for non-skewed datasets, whereas `MaxMin+` is a more balanced heuristic. The latter is also expected since skewed datasets generally contain much larger number of tasks than non-skewed datasets.

Table 13 displays the number of `MaxMin`-based assignments performed by `MaxMin+`. As seen in this table, in general, the number of `MaxMin`-based assignments considerably decreases with increasing K values, thus conforming with the expectation given in Section 3.2. This behavior explains the decrease in the running time performance gap between `MaxMin+` and `MinMin+` with increasing K as shown in Table 12. Even for the smallest K value of four, the number of `MaxMin`-based assignments is much smaller than the number of `MinMin`-based assignments for each instance. For $K = 4$, the worst case occurs for the `big` matrix, where only 9.25% of the assignments are `MaxMin`-based assignments. These results show that the expected number of `MaxMin`-based assignments given in Theorem 3.1 for $K = 2$ homogenous processors is a rather loose upper bound for $K \geq 4$ heterogeneous processors.

As seen in Table 13, `MaxMin+` makes only one `MaxMin`-based assignment for the 32-way assign-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

TABLE 8
Running times (seconds) of heuristics for social network datasets

Dataset	K	Original heuristics						Proposed heuristics			
		MM	MxM	Suff	RC	RASA	GA	MM+	MxM+	Suff+	GA+
coauthorship	4	53,859.2	63,053.1	67,678.7	89,023.9	64,896.3	54,884.8	5.7	172.5	387.4	1,031.2
	8	70,204.6	66,434.0	97,158.5	1.2×10^5	81,245.7	72,146.3	11.5	71.8	218.3	1,953.2
	16	1.2×10^5	1.4×10^5	2.0×10^5	1.9×10^5	1.4×10^5	1.3×10^5	20.9	66.3	168.2	4,407.1
	24	1.8×10^5	2.0×10^5	2.8×10^5	2.4×10^5	1.7×10^5	1.9×10^5	33.4	85.7	235.6	4,277.8
	32	2.1×10^5	2.1×10^5	3.5×10^5	2.8×10^5	1.9×10^5	2.2×10^5	40.9	84.8	171.4	4,414.9
commonJob	4	8,276.9	6,810.9	5,346.2	4,883.6	7,059.9	8,781.3	1.3	2.3	2.7	505.6
	8	8,242.8	9,522.5	9,031.0	9,810.2	8,604.5	9,375.7	2.4	3.0	3.7	1,135.3
	16	13,506.1	13,627.6	12,932.8	13,657.7	13,847.8	14,905.9	2.6	5.9	4.2	1,402.5
	24	18,835.1	18,537.7	20,593.0	26,190.4	17,578.4	20,346.4	7.7	9.7	9.6	1,519.0
	32	24,104.4	37,576.2	26,927.1	26,379.2	21,281.3	25,619.6	9.7	10.5	9.2	1,524.9

TABLE 9
Running times (seconds) of heuristics for distributed web crawling datasets

Dataset	K	Original heuristics						Proposed heuristics			
		MM	MxM	Suff	RC	RASA	GA	MM+	MxM+	Suff+	GA+
ClueWeb-B	4	73,814.2	75,260.0	78,577.7	1.1×10^5	90,773.2	77,284.4	4.1	5.3	7.7	3,474.3
	8	1.2×10^5	88,850.7	79,415.0	1.4×10^5	1.1×10^5	1.2×10^5	9.5	11.5	13.6	4,386.9
	16	2.3×10^5	1.4×10^5	1.9×10^5	2.8×10^5	1.3×10^5	2.4×10^5	18.2	17.6	22.5	5,144.0
	24	2.9×10^5	2.6×10^5	2.9×10^5	3.7×10^5	1.8×10^5	2.9×10^5	36.5	42.4	28.2	4,059.6
	32	4.1×10^5	3.2×10^5	3.6×10^5	4.3×10^5	2.2×10^5	4.1×10^5	47.3	41.6	46.0	4,169.8
ClueWeb-A	4	6.7×10^5	8.1×10^5	7.3×10^5	9.3×10^5	6.5×10^5	6.9×10^5	19.6	19.4	20.8	12,573.3
	8	8.4×10^5	1.1×10^6	1.0×10^6	1.4×10^6	7.9×10^5	8.5×10^5	39.2	38.8	51.1	11,473.6
	16	1.9×10^6	1.7×10^6	1.8×10^6	2.8×10^6	1.2×10^6	2.0×10^6	60.5	84.2	89.7	12,936.7
	24	2.7×10^6	2.6×10^6	3.0×10^6	2.9×10^6	1.8×10^6	2.7×10^6	106.2	112.3	141.0	14,059.2
	32	3.3×10^6	2.9×10^6	3.2×10^6	3.5×10^6	2.8×10^6	3.4×10^6	183.9	174.5	193.1	14,231.3

TABLE 10
Running times (seconds) of heuristics for parallel DVR datasets

Dataset	K	Original heuristics						Proposed heuristics			
		MM	MxM	Suff	RC	RASA	GA	MM+	MxM+	Suff+	GA+
blunt	4	15.3	15.1	17.8	21.7	18.9	25.6	0.0	0.7	2.5	10.3
	8	26.0	23.9	34.0	43.1	29.9	40.7	0.1	0.5	2.1	14.7
	16	69.2	59.7	100.8	107.8	132.4	90.7	0.2	0.4	1.8	21.7
	24	208.5	228.7	163.1	174.5	164.8	234.4	0.3	0.8	4.3	26.2
	32	259.2	287.1	334.6	246.1	231.6	291.8	0.3	0.8	3.8	32.9
comb	4	56.3	39.1	47.0	188.5	85.8	70.2	0.1	1.4	5.0	14.0
	8	88.0	124.3	93.5	113.0	186.7	114.7	0.2	0.8	3.9	26.8
	16	159.5	191.2	279.7	236.9	256.7	200.6	0.3	1.0	3.8	41.4
	24	314.0	289.2	356.3	466.2	350.3	360.7	0.6	1.7	6.7	47.3
	32	437.3	445.9	446.2	457.5	436.5	475.7	0.6	1.5	6.8	38.9

ment of `ClueWeb-B` and all K-way assignments of `ClueWeb-A`. `ClueWeb-A` has an extremely large task whose weight is greater than the sum of the weights of all other tasks. The assignment of such a large task to its favorite processor avoids the need for a second `MaxMin`-based assignment in future iterations. A similar reasoning holds for the 32-way assignment of `ClueWeb-B`. In fact, `MaxMin` is also expected to find a “good” solution in such assignment instances. As seen in Tables 3–6, these are the only assignment instances where `MaxMin` was able to find a solution with the same makespan as `MaxMin+`.

`MaxMin+` versus `RASA`: Although `RASA` finds slightly better solutions than `MaxMin`, `MaxMin+` finds significantly better solutions than `RASA` in all assignment instances, except for the 32-way assignment of `ClueWeb-B` and the assignment instances of `ClueWeb-A`, where all three heuristics find solutions with the same makespan. On average, `MaxMin+` attains average load imbalance values of 177.74% and 0.62% compared to 319.40% and 173.46% of `RASA`, for skewed and non-skewed datasets, respectively. These results validate the success of the proposed adaptive selection policy of `MaxMin+` over that of `RASA`. `MaxMin+` is several orders of magnitude faster than `RASA` in all assignment instances. On average, `MaxMin+` runs 5953- and 333-times faster than `RASA` for skewed and non-skewed datasets, respectively.

`Suff+` versus `Suff`: Out of 95 assignment instances, `Suff+` finds better solutions than `Suff` in 83 instances, whereas `Suff` finds better solutions than `Suff+` in only six instances. In the remaining six assignment instances (five assignment instances of `ClueWeb-A` and the 32-way assignment of `ClueWeb-B`), both `Suff` and `Suff+` find solutions with the same makespan. As seen in Table 7, in terms

TABLE 11
Running times (seconds) of heuristics for parallel SpMxV datasets

		Original heuristics						Proposed heuristics			
Dataset	K	MM	MxM	Suff	RC	RASA	GA	MM+	MxM+	Suff+	GA+
barrier2-1	4	1,044.0	1,245.0	1,978.2	1,065.2	1,505.8	1,176.6	0.6	20.6	74.8	133.1
	8	1,809.8	1,835.4	2,295.3	2,343.4	2,008.1	2,134.5	1.1	15.9	61.9	325.8
	16	3,356.1	3,138.0	3,961.0	4,697.8	3,254.9	3,636.5	2.2	20.4	62.2	282.7
	24	4,534.0	4,893.5	5,511.7	5,959.1	4,099.6	5,150.5	3.7	15.1	73.9	620.3
	32	5,078.4	5,810.1	6,360.4	6,551.5	6,345.0	5,418.7	3.5	14.1	83.4	343.9
language	4	15,081.9	16,432.9	16,562.0	26,088.5	25,826.6	16,413.0	2.3	65.9	166.8	1,333.3
	8	25,924.4	23,470.0	24,928.3	34,444.6	34,882.4	28,029.7	4.7	15.9	40.0	2,110.1
	16	39,780.2	34,559.5	47,030.2	71,420.6	51,280.7	42,259.3	11.5	12.6	12.0	2,490.6
	24	74,398.5	76,276.2	75,045.1	90,750.8	70,951.8	77,000.6	22.4	32.3	50.2	2,624.6
	32	71,323.0	67,528.2	71,835.9	77,366.0	77,218.7	73,990.1	18.2	21.9	15.9	2,685.3
olafu	4	9.1	9.8	11.0	13.4	13.2	15.7	0.0	0.4	1.5	6.7
	8	14.0	13.6	49.8	22.0	18.9	22.6	0.1	0.3	1.2	8.7
	16	35.8	40.7	38.5	58.1	54.3	54.9	0.2	0.3	1.0	19.3
	24	81.9	129.2	97.7	106.2	84.4	101.7	0.3	0.4	2.4	20.1
	32	180.4	156.1	136.8	143.1	131.6	196.1	0.3	0.5	2.1	15.9
Lin	4	6,987.9	8,185.9	7,401.1	10,191.5	7,977.8	7,234.2	1.2	200.8	684.1	247.5
	8	8,309.2	10,809.7	11,219.6	16,430.8	10,947.9	8,688.3	2.0	129.2	556.3	381.1
	16	15,901.8	24,876.3	20,575.5	28,687.5	16,427.6	16,374.7	6.0	117.8	784.2	478.9
	24	23,305.1	23,062.5	25,086.5	31,325.5	23,233.7	23,847.7	8.5	109.9	794.6	551.1
	32	28,725.5	29,544.6	31,139.2	45,157.7	29,805.8	29,184.5	10.5	119.4	766.7	469.5

TABLE 12
Normalized running time averages over all datasets

		Original heuristics						Proposed heuristics			
Dataset	K	MM	MxM	Suff	RC	RASA	GA	MM+	MxM+	Suff+	GA+
Skewed	4	12,813.9	14,307.1	13,863.1	17,964.0	14,379.7	13,294.8	1.0	16.7	46.8	481.9
	8	8,357.6	9,069.9	8,878.2	12,122.8	8,653.1	8,711.3	1.0	4.5	14.5	354.7
	16	10,134.9	8,614.4	9,924.2	14,029.0	7,595.1	10,397.5	1.0	3.0	6.9	263.6
	24	7,604.2	7,363.6	8,610.4	8,867.2	5,623.8	7,745.3	1.0	1.9	5.4	142.1
	32	6,643.6	6,186.7	6,981.4	7,304.4	5,482.1	6,755.3	1.0	1.7	5.3	112.8
Non-skewed	4	2,274.3	2,556.8	2,501.8	2,988.4	2,488.6	2,435.1	1.0	38.0	130.5	161.8
	8	1,555.4	1,902.1	1,858.3	2,553.1	1,907.3	1,703.4	1.0	13.1	59.1	149.0
	16	1,449.6	1,577.0	1,766.1	2,168.1	1,539.7	1,565.2	1.0	5.9	29.5	116.6
	24	1,187.6	1,558.1	1,625.2	1,620.5	1,171.9	1,250.5	1.0	4.1	21.5	63.9
	32	1,241.6	1,618.3	1,639.7	1,814.9	1,343.0	1,298.0	1.0	3.8	20.4	57.4

of average load balancing quality, Suff+ shows comparable performance with Suff for skewed datasets, whereas Suff+ performs better than Suff for non-skewed datasets. On average, Suff+ attains average load imbalance values of 178.31% and 0.51% compared to 178.12% and 1.37% of Suff, for skewed and non-skewed datasets, respectively. As seen in Table 12, Suff+ is a few orders of magnitude faster than Suff in all assignment instances. On average, Suff+ runs 6078- and 194-times faster than Suff for skewed and non-skewed datasets, respectively.

GA+ versus GA: As mentioned in Section 3.4, GA+ finds exactly the same solutions as GA. However, GA+ is significantly faster than GA in all assignment instances. On average, GA+ is 19-, 16-, 23-, 22-, and 38-times faster than GA in 4-, 8-, 16-, 24-, and 32-way assignments, respectively. For the 16-way assignment of the largest dataset ClueWeb-A, GA finds a solution in about 23 days while GA+ finds the same solution in less than four hours, i.e., GA+ runs about 154 times faster than GA for that assignment instance.

4.2.2 General Comparison

For general performance comparison, we will only consider MinMin+, MaxMin+, Suff+, GA+, and RC since the improved versions perform better than their traditional counterparts and MaxMin+ performs significantly better than RASA.

For the six skewed datasets, both of the proposed hybrid algorithms, MaxMin+ and Suff+, find considerably better solutions than MinMin+, in terms of load balancing quality. Out of 30 assignment instances of skewed datasets, RC, MaxMin+, and Suff+ find the best solutions in 14, 11, and 11 assignment instances, respectively. As seen in Table 7, MaxMin+ and Suff+ respectively attain load imbalance values of 177.74% and 178.31% compared to 177.26% of RC, on average. Hence, MaxMin+ and Suff+ display comparable performance with RC in terms of load balancing quality. However, both MaxMin+ and Suff+ are significantly faster than RC in all of these 30 assignment instances. On average, MaxMin+ and Suff+ respectively run 2657- and 1588-times faster than RC. Hence, the use of RC in large datasets is not feasible.

For skewed datasets, we recommend the use of MaxMin+. Because, as seen in Tables 7 and 12,

TABLE 13
Number of MaxMin-based assignments performed by MaxMin+

Social network			Distributed web crawling			Parallel DVR			Parallel SpMxV			Parallel SpMxV		
Dataset	K	m	Dataset	K	m	Dataset	K	m	Dataset	K	m	Dataset	K	m
coauthorship (N=725,344)			**ClueWeb-B (N=799,115)**			**blunt (N=20,611)**			**barrier2-1 (N=113,076)**			**olafu (N=16,146)**		
	4	13,528		4	257		4	1,840		4	8,233		4	1,416
	8	3,631		8	289		8	696		8	2,987		8	535
	16	1,190		16	9		16	282		16	1,198		16	227
	24	686		24	2		24	172		24	668		24	138
	32	444		32	1		32	128		32	496		32	103
commonJob (N=241,233)			**ClueWeb-A (N=2,483,726)**			**comb (N=32,238)**			**language (N=399,130)**			**Lin (N=256,000)**		
	4	441		4	1		4	2,466		4	8,986		4	23,376
	8	93		8	1		8	912		8	1,093		8	8,882
	16	23		16	1		16	370		16	114		16	3,666
	24	11		24	1		24	226		24	137		24	2,246
	32	9		32	1		32	165		32	11		32	1,634

MaxMin+ is considerably faster than Suff+ and yields comparable performance in terms of load balancing quality.

For the 13 non-skewed datasets, GA+ finds the best solutions in 51 assignment instances out of 65 assignment instances in terms of load balancing quality. GA+ performs better than the other heuristics in assignment instances where MinMin+ already shows good performance (e.g., SpMxV and DVR datasets). This can be attributed to the fact that GA+ improves the initial assignment provided by MinMin+. Furthermore, GA+ is approximately two orders of magnitude slower than MinMin+. Hence, to analyze the performance of MinMin+, we exclude GA+ in the statistics given in the following paragraph to show the relative performance of the algorithms in finding the best assignments.

Out of 65 assignment instances of the non-skewed datasets, RC, MinMin+, MaxMin+, and Suff+ find the best assignments in 17, 17, 18, and 17 assignment instances, respectively. As seen in Table 7, MinMin+, MaxMin+ and Suff+ respectively attain load imbalance values of 0.62%, 0.62%, and 0.51% compared to 0.61% of RC, on average. Hence, MinMin+, MaxMin+, and Suff+ display comparable load-balancing performance with RC for non-skewed datasets. However, for these 65 assignment instances, MinMin+, MaxMin+, and Suff+ respectively run 2229-, 499-, and 236-times faster than RC, on average. Hence, the use of RC is not feasible also for large non-skewed datasets. For these 65 assignment instances, MinMin+ runs 13- and 52-times faster than MaxMin+ and Suff+, respectively, on average. We observe a trade-off between the solution quality and running times of MinMin+ and GA+. GA+ displays better load balancing performance than MinMin+, whereas MinMin+ is significantly faster (110-times, on average).

For non-skewed datasets, we recommend the use of MinMin+, since MinMin+ runs significantly faster than both MaxMin+ and Suff+ while achieving comparable load balancing performance. The use of GA+ should be considered only if the significantly higher running time of GA+ can be amortized by the improved load balancing on the target application.

5 Conclusion

We presented certain performance improvements over the popular independent task assignment heuristics MinMin, MaxMin, and Suff. In particular, we proposed the MinMin+ heuristic which improves the worst-case runtime complexity of MinMin from $O(KN^2)$ to $O(KN \log N)$ in assigning N independent tasks to K processors. Moreover, we proposed the MaxMin+ and Suff+ heuristics, which are hybrid versions of MaxMin and Suff, obtained by combining the latter heuristics with MinMin. We evaluated the performance of all heuristics over a large number of real-life datasets. The experiments indicate that each of our heuristics runs considerably faster than their traditional counterparts, MinMin+ being the fastest. In terms of the solution quality, both MaxMin+ and Suff+ are found to perform considerably better than MinMin+ for skewed datasets while MinMin+ is found to perform comparable for non-skewed datasets. Considering the tradeoffs between the solution quality and the running times of the proposed assignment algorithms, we recommend the use of MinMin+ for non-skewed datasets and recommend MaxMin+ for skewed datasets.

References

[1] O. H. Ibarra and C. E. Kim, "Heuristic algorithms for scheduling independent tasks on nonidentical processors," *J. ACM*, vol. 24, no. 2, pp. 280–289, 1977.

[2] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *J. Parallel Distrib. Comput.*, vol. 61, no. 6, pp. 810–837, 2001.

[3] H. J. Siegel and S. Ali, "Techniques for mapping tasks to machines in heterogeneous computing systems," *J. Syst. Archit.*, vol. 46, no. 8, pp. 627–639, 2000.

[4] R. Duan, R. Prodan, and T. Fahringer, "Performance and cost optimization for multiple large-scale grid workflow applications," in *Proc. 2007 ACM/IEEE Conf. Supercomputing*, 2007, pp. 12:1–12:12.

[5] P. Luo, K. Lü, and Z. Shi, "A revisit of fast greedy heuristics for mapping a class of independent tasks onto heterogeneous computing systems," *J. Parallel Distrib. Comput.*, vol. 67, pp. 695–714, 2007.

[6] E. Davis and J. M. Jaffe, “Algorithms for scheduling tasks on unrelated processors,” *J. ACM*, vol. 28, pp. 721–736, 1981.

[7] P. C. SaiRanga and S. Baskiyar, “A low complexity algorithm for dynamic scheduling of independent tasks onto heterogeneous computing systems,” in *Proc. 43rd Annual Southeast Regional Conf.*, 2005, pp. 63–68.

[8] R. Armstrong, D. Hensgen, and T. Kidd, “The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions,” in *7th IEEE Heterogeneous Computing Workshop)*, 1998, pp. 79–87.

[9] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund, “Dynamic mapping of a class of independent tasks onto heterogeneous computing systems,” *J. Parallel Distrib. Comput.*, vol. 59, pp. 107–131, 1999.

[10] C. Liu and S. Baskiyar, “A general distributed scalable grid scheduler for independent tasks,” *J. Parallel Distrib. Comput.*, vol. 69, pp. 307–314, 2009.

[11] A. J. Page, T. M. Keane, and T. J. Naughton, “Multi-heuristic dynamic task allocation using genetic algorithms in a heterogeneous distributed system,” *J. Parallel Distrib. Comput.*, vol. 70, pp. 758–766, 2010.

[12] S. S. Chauhan and R. C. Joshi, “QoS guided heuristic algorithms for grid task scheduling,” *Int. J. Comput. Appl.*, vol. 2, no. 9, pp. 24–31, 2010.

[13] K. Kaya and C. Aykanat, “Iterative-improvement-based heuristics for adaptive scheduling of tasks sharing files on heterogeneous master-slave environments,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, no. 8, pp. 883 –896, 2006.

[14] F. Pinel, B. Dorronsoro, and P. Bouvry, “Solving very large instances of the scheduling of independent tasks problem on the GPU,” *J. Parallel Distrib. Comput.*, 2012, in press.

[15] R. F. Freund, M. Gherrity, S. Ambrosius, M. Campbell, M. Halderman, D. Hensgen, E. Keith, T. Kidd, M. Kussow, J. D. Lima, F. Mirabile, L. Moore, B. Rust, and H. J. Siegel, “Scheduling resources in multi-user, heterogeneous, computing environments with SmartNet,” in *Proc. 7th Heterogeneous Computing Workshop*, 1998, p. 3.

[16] K. Kaya, B. Uçar, and C. Aykanat, “Heuristics for scheduling file-sharing tasks on heterogeneous systems with distributed repositories,” *J. Parallel Distrib. Comput.*, vol. 67, no. 3, pp. 271–285, 2007.

[17] M.-Y. Wu and W. Shu, “A high-performance mapping algorithm for heterogeneous computing systems,” in *Parallel and Distributed Processing Symposium., Proceedings 15th International*, Apr 2001.

[18] L. Wang, H. J. Siegel, V. R. Roychowdhury, and A. A. Maciejewski, “Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach,” *J. Parallel Distrib. Comput.*, vol. 47, no. 1, pp. 8–22, Nov. 1997.

[19] F. Xhafa, E. Alba, B. Dorronsoro, and B. Duran, “Efficient batch job scheduling in grids using cellular memetic algorithms,” *Journal of Mathematical Modelling and Algorithms*, vol. 7, pp. 217–236, 2008.

[20] S. Parsa and R. Entezari-Maleki, “Rasa-a new grid task scheduling algorithm,” *JDCTA: International Journal of Digital Content Technology and its Applications*, vol. 3, no. 4, pp. 91–99, 2009.

[21] M. Hardy, “Pareto’s law,” *The Mathematical Intelligencer*, vol. 32, pp. 38–43, 2010.

[22] CMU-LTI, “The ClueWeb09 dataset,” http://boston.lti.cs.cmu.edu/Data/clueweb09/, 2009.

[23] H. Kutluca, T. M. Kurç, and C. Aykanat, “Image-space decomposition algorithms for sort-first parallel volume rendering of unstructured grids,” *The Journal of Supercomputing*, vol. 15, no. 1, pp. 51–93, 2000.

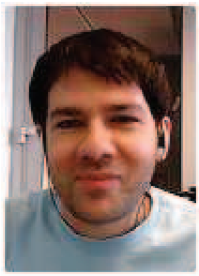
[24] “NASA advanced supercomputing division (NAS) dataset archive,” http://www.nas.nasa.gov/Research/Datasets/datasets.html.

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

[26] B. B. Cambazoglu, E. Varol, E. Kayaaslan, C. Aykanat, and R. Baeza-Yates, “Query forwarding in geographically distributed search engines,” in *Proceeding of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval*. New York, NY, USA: ACM, 2010, pp. 90–97.

[27] S. Ali, H. J. Siegel, M. Maheswaran, S. Ali, and D. Hensgen, “Task execution time modeling for heterogeneous computing systems,” in *Proc. 9th Heterogeneous Computing Workshop*, 2000, pp. 185–199.

E. Kartal Tabak received his BS degree, in computer engineering, from the Computer Engineering Department of Bilkent University. He is currently a PhD candidate at Computer Engineering Department of Bilkent University. His research interests include parallel computing and algorithms, high performance Web Search Engines, computer vision, and software engineering.



Dr. B. Barla Cambazoglu received his BS, MS, and PhD degrees, all in computer engineering, from the Computer Engineering Department of Bilkent University in 1997, 2000, and 2006, respectively. He has then worked as a postdoctoral researcher in the Biomedical Informatics Department of the Ohio State University. He is currently employed as a senior researcher in Yahoo! Research. He has worked in several research projects, funded by the Scientific and Technological Research Council of Turkey, the European Union Sixth and Seventh Framework Programs, and the National Cancer Institute. In 2007, he received the Embodying the Vision award as a developer in the caBIG project. His research interests mainly include information retrieval, web search, and distributed computing. He has papers published in prestigious journals including IEEE TPDS, JPDC, ACM TWEB, Inf. Syst., and IP&M, as well as top-tier conferences such as WWW, SIGIR, KDD, WSDM, and CIKM.

Dr. Cevdet Aykanat received his B.S. and M.S. degrees from Middle East Technical University, Ankara, Turkey, both in Electrical Engineering, and his Ph.D. degree from Ohio State University, Columbus, US, in Electrical and Computer Engineering. He was a Fulbright scholar during his Ph.D. studies. He worked at the Intel Supercomputer Systems Division, Beaverton, Oregon, US, as a research associate. Since 1989, he has been affiliated with the Department of Computer Engineering, Bilkent University, Ankara, Turkey, where he is currently a professor. His research interests mainly include parallel computing, parallel scientific computing and its combinatorial aspects, parallel computer graphics applications, parallel data mining, graph and hypergraph theoretic models for load balancing, high performance information retrieval systems, parallel and distributed databases, and grid computing. He has (co)authored more than 70 technical papers published in academic journals indexed in the ISI, and his publications have received more than 600 citations in ISI indexes. He is the recipient of the 1995 Young Investigator Award of The Scientific and Technological Research Council of Turkey and 2007 Parlar Science Award. He was appointed a member of IFIP Working Group 10.3 (Concurrent System Technology) in April 2004, a member of the EU-INTAS Council of Scientists in June 2005, and an Associate Editor of IEEE Transactions of Parallel and Distributed Systems in December 2008.