Segmented Min-Min: A Static Mapping Algorithm for Meta-tasks on Heterogeneous Computing Systems

Min-You Wu and Wei Shu
Department of Electrical and Computer Engineering
University of New Mexico

Hong Zhang
Department of Electrical and Computer Engineering
University of Central Florida

Abstract

The Min-min algorithm is a simple algorithm. It runs fast and delivers good performance. However, the Min-min algorithm schedules small tasks first, resulting in some load imbalance. In this paper, we present an algorithm which improves the Min-min algorithm by scheduling large tasks first. The new algorithm, Segmented min-min, balances the load well and demonstrates even better performance in both makespan and running time.

1. Introduction

A heterogeneous computing environment utilizes a suite of different machines interconnected by high-speed networks to execute different computationally intensive applications that have diverse computational requirements [8, 12, 13]. The general problem of mapping tasks to machines has been shown to be NP-complete [10]. Many useful heuristics to perform this mapping function have been developed. Among many sophisticated algorithms, the Min-min algorithm [10] is a simple algorithm which runs fast and delivers satisfactory performance. It selects from all tasks the task that minimizes the completion time on a machine. In most situations, it maps as many tasks as possible to their first choice of machine. However, the Min-min algorithm is unable to balance the load well since it usually schedules small tasks first. In this paper, we propose a simple alternative of the Min-min algorithm by scheduling large tasks first. The proposed algorithm retains the advantage of the Min-min algorithm and achieves good load balance at the same time.

This paper presents the new algorithm, named the *Segmented min-min* algorithm. In section 2, previous heuristic algorithms are reviewed. Section 3 presents the new algorithm. Section 4 exhibits the simulation model and experimental results. Section 5 concludes the paper.

2. Previous Heuristics

In this section, we review a set of heuristic algorithms which schedule meta-tasks to heterogeneous computing systems. A meta-task is defined as a collection of independent tasks with no data dependences. Meta-tasks are mapped onto machines statically; each machine executes a single task at a time. For static mapping, it is assumed that the number of tasks, t, and the number of machines, m, are known a priori.

A large number of heuristic algorithms have been designed to schedule tasks to machines on heterogeneous computing systems. In [2], eleven commonly used algorithms have been evaluated, listed as follows.

OLB : Opportunistic Load Balancing (OLB) assigns each task, in arbitrary order, to the next available machine [1, 7, 8].

UDA : User-Directed Assignment (UDA) assigns each task, in arbitrary order, to the machine with the *best expected execution time* for the task [1, 7].

Fast Greedy : Fast Greedy assigns each task, in arbitrary order, to the machine with the *minimum completion time* for that task [1].

Min-min : In Min-min, the minimum completion time for each task is computed respect to all machines. The task with the *overall minimum completion time* is selected and assigned to the corresponding machine. The newly mapped task is removed, and the process repeats until all tasks are mapped [1, 7, 10].

0-7695-0556-2/00 $10.00 © 2000 IEEE

Max-min : The Max-min heuristic is very similar to the Min-min algorithm. The set of minimum completion times is calculated for every task. The task with *overall maximum completion time* from the set is selected and assigned to the corresponding machine [1, 7, 10].

Greedy : The Greedy heuristic is literally a combination of the Min-min and Max-min heuristics by using the better solution [1, 7].

GA : The Genetic algorithm (GA) is used for searching large solution space. It operates on a population of chromosomes for a given problem. The initial population is generated randomly. A chromosome could be generated by any other heuristic algorithm. When it is generated by Min-min, it is called "seeding" the population with Min-min [15, 14].

SA : Simulated Annealing (SA) is an iterative technique that considers only one possible solution for each meta-task at a time. SA uses a procedure that probabilistically allows solution to be accepted to attempt to obtain a better search of the solution space based on a system temperature [5, 11].

GSA : The Genetic Simulated Annealing (GSA) heuristic is a combination of the GA and SA techniques [3].

Tabu : Tabu search is a solution space search that keeps track of the regions of the solution space which have already been searched so as not to repeat a search near these areas [6, 9].

A* : A* is a tree search beginning at a root node that is usually a null solution. As the tree grows, intermediate nodes represent partial solutions and leaf nodes represent final solutions. Each node has a cost function, and the node with the minimum cost function is replaced by its children. Any time a node is added, the tree is pruned by deleting the node with the largest cost function. This process continues until a complete mapping (a leaf node) is reached [4].

The experimental results from [2] show that OLB, UDA, Max-min, SA, GSA, and Tabu do not produce good schedules in general. Min-min, GA, and A* are able to deliver good performance. The difference between the completion times of the schedules (makespans) generated by these three algorithms is within 10%. GA is consistently better than Min-min by a few percents, since it is seeding the population with a Min-min chromosome. A*, on the other hand, produces better or worse schedules than Min-min and GA in different situations. Among the three algorithms, Min-min is the fastest algorithm, GA is much slower, and A* is very slow. For 512 tasks and 16 machines, the running time of Min-min is about 1 second, GA 30 seconds, and A* 1200 seconds [2].

Min-min is a simple algorithm, fast, and able to deliver good performance. Even GA has to be "seeding" the population with a Min-min chromosome to obtain its good performance. Min-min schedules the "best case" tasks first and generates relatively good schedules. The drawback of Min-min is that it assigns the small task first. Thus, the smaller tasks would execute first and then a few larger tasks execute while several machines sit idle, resulting in poor machine utilization. We propose a simple method to enforce large tasks to be scheduled first. Tasks are partitioned into segments according to their execution times. The segment with larger tasks is scheduled first with the Min-min algorithm being applied within the segment. This is called *Segmented min-min (Smm)*.

3. The Segmented Min-Min Algorithm

Every task has a *ETC (expected time to compute)* on a specific machine. If there are t tasks and m machines, we can obtain a $t \times m$ *ETC matrix*. $ETC(i, j)$ is the estimated execution time for task i on machine j.

The *Segmented min-min* algorithm sorts the tasks according to ETCs. The tasks can be sorted into an ordered list by the average ETC, the minimum ETC, or the maximum ETC. Then, the task list is partitioned into segments with the equal size. The segment of larger tasks is scheduled first and the segment of smaller tasks last. For each segment, Min-min is applied to assign tasks to machines. The algorithm is described as follows.

Segmented min-min (Smm)

1. Compute the sorting key for each task:

SUB-POLICY 1 — *Smm-avg*: Compute the average value of each row in ETC matrix

$$key_i = \sum_j ETC(i, j)/m.$$

SUB-POLICY 2 — *Smm-min*: Compute the minimum value of each row in ETC matrix

$$key_i = \min_j ETC(i, j).$$

SUB-POLICY 3 — *Smm-max*: Compute the maximum value of each row in ETC matrix

$$key_i = \max_j ETC(i, j).$$

2. Sort the tasks into a task list in decreasing order of their keys.

3. Partition the tasks evenly into N segments.

4. Schedule each segment in order by applying Min-min.

Different from the Min-min algorithm, Segmented min-min performs task sorting before scheduling. Sorting implies that larger tasks are promoted to be scheduled earlier. Then, Min-min is applied locally within each segment. The problem here is how to define the sorting key. Tasks with long execution time deserve promotion to early scheduling. However, in a heterogeneous system, the execution time of a task varies in different machines. Therefore, we test three sub-policies by defining the execution time of a task as the average, the minimum, or the maximum of its ETCs.

The third step of the Segmented min-min algorithm partitions tasks into N segments. Determining the optimal value of N is a trade-off. More segments result in better load balance. On the other hand, too many segments will lose advantages of the Min-min algorithm. Intuitively, as long as we partition the tasks into a few segments, such as large, medium, and small tasks, the load can be balanced fairly well. Experimental results confirm this as shown in Figure 1 where the curves show the improvement of *Smm-avg* over Min-min for different values of N. Each point in these curves is the average of five runs. In general, the optimal value of N is relevant to the ratio $c = \frac{t}{m}$. When c is large, Min-min performs well. For small c, which means the number of tasks per machine is not large, the optimal value of N is about 4 or 5. Therefore, we fix the value of N to 4, which means that we always partition the tasks into four segments.

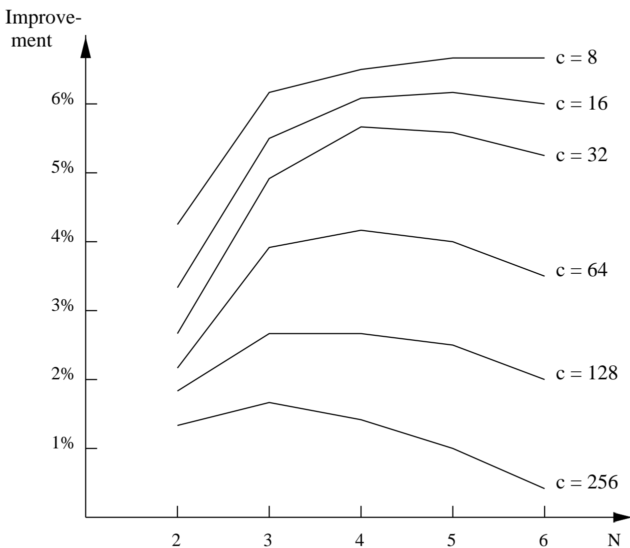

Figure 1. The N Value.

4. Experiments

4.1. Performance Comparison

For the experimental studies, we use the same method in [2] to generate the test set. The parameters include *Consistant, Inconsistant,* or *Semi-Consistant*; *High* or *Low Task Heterogeneity*; and *High* or *Low Machine Heterogeneity*. For details, see [2]. All experiment results are based on 512 tasks, 16 or 32 machines, 100 trails and N = 4. The results for 16 machines are shown in Tables I to XII and that for 32 machines are shown in Tables XIII to XXIV. In these tables, the second column shows the utilization of machines which is defined as $1 - \frac{\sum idle\ time}{m \times makespan}$. The third column is the makespan (the completion time) of schedules. The fourth column is the improvement of each Segmented min-min algorithm over the Max-min algorithm and the fifth column is that over the Min-min algorithm. The last column shows the running time of each algorithm.

4.2. Discussion

From these results, we found that the Segmented min-min algorithm is able to balance the load very well compared to the Max-min and the Min-min algorithms. The system utilization of Min-min is relatively low while that of Segmented min-min is very high. This is because Segmented min-min schedules larger tasks first and smaller tasks can run in parallel with large tasks. Although the Max-min algorithm produces very good load balancing, it does not schedule tasks to their "best case." Thus, its performance is far worse than that of the Segmented min-min algorithm. Higher system utilization makes three Segmented min-min algorithms better than Min-min in almost all cases. *Smm-avg* enhances the performance of Min-min from 2% to 12%. *Smm-min* shows better performance than *Smm-avg* in some cases but is worse than *Smm-avg* in most cases. *Smm-max* is worse than Smm-avg in almost all cases. Thus, we use *Smm-avg* for the Segmented min-min algorithm, which improves the Min-min algorithm by 6.1% in average.

In addition, the running time of the Segmented min-min algorithm is much less than Min-min. This is not difficult to explain because Min-min spends the large amount of time to search entire matrix to map one task each time, while Segmented min-min, taking advantage of the divide-and-conquer strategy, only searches the minimum value within a single partition. In summary, this partitioning method improves the makespan and running time simultaneously.

Table I. 16 Machines, Inconsistent, Low Task, Low Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^3$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.9%	5.425	–	–	1.19
Min-min	91.0%	2.915	–	–	1.06
Smm-avg	98.1%	2.767	96.0%	5.3%	0.33
Smm-min	98.4%	2.746	96.3%	6.1%	0.33
Smm-max	97.8%	2.784	94.9%	4.7%	0.33

Table II. 16 Machines, Inconsistent, Low Task, High Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^5$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.8%	2.513	–	–	1.19
Min-min	83.3%	1.214	–	–	1.06
Smm-avg	96.9%	1.113	125.8%	9.1%	0.33
Smm-min	98.2%	1.064	136.2%	14.2%	0.33
Smm-max	95.9%	1.135	121.4%	7.0%	0.33

Table III. 16 Machines, Inconsistent, High Task, Low Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^4$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.9%	15.943	–	–	1.20
Min-min	91.0%	8.588	–	–	1.07
Smm-avg	98.2%	8.139	95.9%	5.5%	0.33
Smm-min	98.5%	8.087	97.1%	6.2%	0.33
Smm-max	97.9%	8.190	94.7%	4.8%	0.33

Table IV. 16 Machines, Inconsistent, High Task, High Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^6$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.8%	7.375	–	–	1.20
Min-min	83.4%	3.573	–	–	1.07
Smm-avg	96.8%	3.279	124.9%	8.9%	0.33
Smm-min	98.3%	3.131	135.5%	14.1%	0.33
Smm-max	95.9%	3.344	125.5%	6.9%	0.33

Table V. 16 Machines, Consistent, Low Task, Low Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^3$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.9%	7.415	–	–	1.22
Min-min	94.0%	5.857	–	–	1.07
Smm-avg	98.6%	5.705	30.0%	2.7%	0.33
Smm-min	98.2%	5.813	27.6%	0.7%	0.33
Smm-max	98.4%	5.749	29.0%	1.9%	0.33

Table VI. 16 Machines, Consistent, Low Task, High Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^5$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.9%	4.125	–	–	1.23
Min-min	89.0%	2.866	–	–	1.07
Smm-avg	97.7%	2.805	47.1%	2.1%	0.33
Smm-min	96.7%	2.910	42.3%	-2.0%	0.33
Smm-max	97.2%	2.867	43.9%	0.0%	0.33

Table VII. 16 Machines, Consistent, High Task, Low Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^5$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	100.0%	2.181	–	–	1.24
Min-min	93.9%	1.725	–	–	1.08
Smm-avg	98.6%	1.679	29.9%	2.8%	0.33
Smm-min	98.2%	1.710	27.5%	0.9%	0.33
Smm-max	98.4%	1.693	28.8%	1.9%	0.33

Table VIII. 16 Machines, Consistent, High Task, High Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^6$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.9%	12.152	–	–	1.24
Min-min	88.9%	8.437	–	–	1.07
Smm-avg	97.7%	8.258	47.2%	2.2%	0.33
Smm-min	96.7%	8.564	41.9%	-1.5%	0.33
Smm-max	97.4%	8.430	44.2%	0.0%	0.33

Table IX. 16 Machines, Semi-Consistent, Low Task, Low Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^3$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.9%	6.339	–	–	1.21
Min-min	91.8%	3.745	–	–	1.07
Smm-avg	98.2%	3.595	76.3%	4.2%	0.33
Smm-min	98.1%	3.624	74.9%	3.3%	0.33
Smm-max	98.0%	3.624	74.9%	3.3%	0.33

Table X. 16 Machines, Semi-Consistent, Low Task, High Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^5$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.8%	3.199	–	–	1.21
Min-min	84.4%	1.664	–	–	1.07
Smm-avg	96.8%	1.569	103.9%	6.1%	0.33
Smm-min	96.5%	1.593	100.8%	4.5%	0.33
Smm-max	96.3%	1.590	101.2%	4.6%	0.33

Table XI. 16 Machines, Semi-Consistent, High Task, Low Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^5$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.9%	1.862	–	–	1.21
Min-min	91.7%	1.104	–	–	1.07
Smm-avg	98.2%	1.058	76.0%	4.4%	0.33
Smm-min	98.1%	1.066	74.7%	3.5%	0.33
Smm-max	98.0%	1.067	74.5%	3.4%	0.33

Table XII. 16 Machines, Semi-Consistent, High Task, High Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^6$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.9%	9.370	–	–	1.21
Min-min	84.6%	4.882	–	–	1.07
Smm-avg	96.9%	4.619	102.9%	5.7%	0.33
Smm-min	96.6%	4.693	99.7%	4.0%	0.33
Smm-max	96.5%	4.673	100.5%	4.7%	0.33

Table XIII. 32 Machines, Inconsistent, Low Task, Low Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^3$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.5%	1.954	–	–	2.23
Min-min	85.1%	1.294	–	–	2.16
Smm-avg	93.1%	1.199	63.0%	7.9%	1.16
Smm-min	93.9%	1.188	64.5%	8.9%	1.10
Smm-max	92.6%	1.206	62.0%	7.3%	1.10

Table XIV. 32 Machines, Inconsistent, Low Task, High Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^4$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	98.8%	6.395	–	–	2.24
Min-min	68.0%	3.959	–	–	2.16
Smm-avg	81.8%	3.523	81.5%	12.4%	1.16
Smm-min	79.3%	3.678	73.9%	7.6%	1.10
Smm-max	82.1%	3.502	82.6%	13.0%	1.10

Table XV. 32 Machines, Inconsistent, High Task, Low Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^4$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.5%	5.755	–	–	2.23
Min-min	85.2%	3.804	–	–	2.16
Smm-avg	93.2%	3.525	63.3%	7.9%	1.16
Smm-min	93.9%	3.498	64.5%	8.7%	1.10
Smm-max	92.4%	3.556	61.8%	7.0%	1.10

Table XVI. 32 Machines, Inconsistent, High Task, High Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^6$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.8%	1.882	–	–	2.21
Min-min	67.9%	1.167	–	–	2.16
Smm-avg	81.7%	1.038	74.3%	12.4%	1.16
Smm-min	79.5%	1.079	74.4%	8.2%	1.09
Smm-max	81.2%	1.044	80.3%	11.8%	1.09

Table XVII. 32 Machines, Consistent, Low Task, Low Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^3$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.8%	3.502	–	–	2.26
Min-min	88.4%	3.129	–	–	2.18
Smm-avg	94.8%	2.982	17.4%	4.9%	1.17
Smm-min	93.4%	3.025	15.8%	3.4%	1.09
Smm-max	94.1%	3.005	16.5%	4.1%	1.09

Table XVIII. 32 Machines, Consistent, Low Task, High Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^5$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.7%	1.707	–	–	2.27
Min-min	76.4%	1.296	–	–	2.18
Smm-avg	89.3%	1.245	37.1%	4.1%	1.17
Smm-min	87.0%	1.279	33.5%	1.3%	1.10
Smm-max	87.9%	1.260	35.5%	2.9%	1.09

Table XIX. 32 Machines, Consistent, High Task, Low Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^4$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.9%	10.305	–	–	2.27
Min-min	88.5%	9.196	–	–	2.19
Smm-avg	94.8%	8.775	17.4%	4.8%	1.18
Smm-min	93.6%	8.887	16.6%	3.5%	1.10
Smm-max	94.1%	8.849	16.5%	3.9%	1.09

Table XX. 32 Machines, Consistent, High Task, High Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^6$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.8%	5.016	–	–	2.26
Min-min	76.5%	3.814	–	–	2.18
Smm-avg	89.3%	3.668	36.8%	4.0%	1.18
Smm-min	87.0%	3.768	33.1%	1.2%	1.09
Smm-max	87.9%	3.717	34.9%	2.6%	1.09

Table XXI. 32 Machines, Semi-Consistent, Low Task, Low Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^3$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.6%	2.586	–	–	2.23
Min-min	85.1%	1.773	–	–	2.19
Smm-avg	92.8%	1.674	54.5%	5.9%	1.17
Smm-min	92.2%	1.679	54.0%	5.6%	1.09
Smm-max	92.3%	1.683	53.7%	5.3%	1.09

Table XXII. 32 Machines, Semi-Consistent, Low Task, High Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^4$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.3%	10.230	–	–	2.22
Min-min	66.4%	6.121	–	–	2.20
Smm-avg	84.3%	5.604	82.5%	9.2%	1.19
Smm-min	84.6%	5.714	79.0%	7.1%	1.09
Smm-max	82.6%	5.682	80.0%	7.7%	1.09

Table XXIII. 32 Machines, Semi-Consistent, High Task, Low Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^4$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.7%	7.603	–	–	2.23
Min-min	85.1%	5.226	–	–	2.20
Smm-avg	92.8%	4.925	54.3%	6.1%	1.19
Smm-min	92.4%	4.937	54.2%	5.9%	1.10
Smm-max	92.1%	4.967	53.1%	5.2%	1.10

Table XXIV. 32 Machines, Semi-Consistent, High Task, High Machine Heterogeneity

Algorithm	System Utilization	Makespan ($\times 10^6$ Sec.)	Improvement over Max-min	Improvement over Min-min	Running Time (Sec.)
Max-min	99.3%	3.012	–	–	2.22
Min-min	66.3%	1.797	–	–	2.19
Smm-avg	84.1%	1.645	83.1%	9.2%	1.18
Smm-min	84.5%	1.682	79.0%	6.8%	1.09
Smm-max	82.8%	1.674	80.0%	7.3%	1.10

5. Concluding Remarks

The Segmented min-min algorithm starts from a set of large tasks while Min-min starting from small tasks. *Smm* can balance the load very well and runs faster. We will compare it in the near future to the Genetic algorithm that delivered the best performance among eleven selected algorithms.

Acknowledgments

The authors would like to thank the anonymous reviewers for their thorough comments which caused us to improve the presentation and level of detail. This research was partially supported by NSF grants CCR-9505300 and CCR-9625784.

References

[1] 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 (HCW '98)*, pages 79–87, Mar. 1998.

[2] T. Braun, H. Siegel, N. Beck, L. Boloni, M. Maheswaran, A. Reuther, J. Robertson, M. Theys, B.Yao, D. Hensgen, and R. Freund. A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing systems. In *8th IEEE Heterogeneous Computing Workshop (HCW '99)*, pages 15–29, Apr. 1999.

[3] H. Chen, N. S. Flann, and D. W. Watson. Parallel genetic simulated annealing: a massively parallel SIMD approach. *IEEE Transactions on Parallel and Distributed Computing*, 9(2):126–136, Feb. 1998.

[4] K. Chow and B. Liu. On mapping signal processing algorithms to a heterogeneous multiprocessor system. In *ICASSP 91*, pages 1585–1588, May 1991.

[5] M. Coli and P. Palazzari. Real time pipelined system design through simulated annealing. *Journal of Systems Architecture*, 42(6-7):465–475, Dec. 1996.

[6] I. D. Falco, R. D. Balio, E.Tarantino, and R. Vaccaro. Improving search by incorporating evolution principles in parallel tabu search. In *IEEE Conference on Evolutionary Computation*, pages 823–828, 1994.

[7] R. Freund, M. Gherrity, S. Ambrosius, M. Campbell, M. Halderman, D. Hensgen, E. Keith, T. Kidd, M. Kussow, J. Lima, F. Mirabile, L. Moore, B. Rust, and H. Siegel. Scheduling resources in multi-user, heterogeneous, computing environments with SmartNet. In *7th IEEE Heterogeneous Computing Workshop (HCW '98)*, pages 184–199, Mar. 1998.

[8] R. F. Freund and H. J. Siegel. Heterogeneous processing. *IEEE Computer*, 26(6):13–17, June 1993.

[9] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.

[10] O. Ibarra and C. Kim. Heuristic algorithms for scheduling independent tasks on nonidentical processors. *Journal of the ACM*, 77(2):280–289, Apr. 1977.

[11] S. Kirkpatrick, J. C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[12] M. Maheswaran, T. D. Braun, and H. J. Siegel. *Encyclopedia of Electrical and Electronics Engineering*, chapter Heterogeneous Distributed Computing. John wiley & sons, 1999.

[13] H. J. Siegel, H. G. Dietz, and J. K. Antonio. *The Computer Science and Engineering Handbook*, chapter Software support for heterogeneous computing, pages 1886–1909. CRC Press, 1997.

[14] H. Singh and A. Youssef. Mapping and scheduling heterogeneous task graphs using genetic algorithms. In *5th IEEE Heterogeneous Computing Workshop (HCW '96)*, pages 86–97, Apr. 1996.

[15] L. Wang, H. J. Siegel, V. P. Roychowdhury, and A. A. Maciejewski. Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach. *Journal of Parallel and Distributed Computing*, 47(1):1–15, Nov. 1997.

Biographies

Min-You Wu is an Associate Professor in the Department of Electrical and Computer Engineering at the University of New Mexico. He received the M.S. degree from the Graduate School of Academia Sinica, Beijing, China, and the Ph.D. degree from Santa Clara University, California. He has held various positions at University of Illinois at Urbana-Champaign, University of California at Irvine, Yale University, Syracuse University, State University of New York at Buffalo, and University of Central Florida. His research interests include parallel and distributed systems, compilers for parallel computers, programming tools, VLSI design, and multimedia systems. He has published over 80 journal and conference papers in the above areas and edited two special issues on parallel operating systems. He is a senior member of IEEE and a member of ACM. He is listed in International Who's Who of Information Technology and Who's Who in America.

Wei Shu received the Ph.D. degree from the University of Illinois at Urbana-Champaign in 1990. Since then, she worked at Yale University, the State University of New York at Buffalo, and University of Central Florida. She is currently an Associate Professor in the Department of Electrical and Computer Engineering, University of New Mexico. Her current interests include dynamic scheduling, resource management, runtime support systems for parallel and distributed processing, multimedia networking, and operating system support for large-scale distributed simulation. She

is a senior member of IEEE and a member of ACM.

Hong Zhang is a graduate student and Teaching Assistant in the Department of Electrical and Computer Engineering at the University of Central Florida. She received her Bachelor of Science degree in Electrical Engineering from Zhejiang University, Hangzhou, P.R. China, in 1986. She worked as a software engineer in the Computer Center of the Institute of High Energy Physics from 1987-1997, mainly engaged in design and maintenance of the computer system. Her research interests include distributed algorithms, computer networks and database management. She will get her master degree in May, 2000.