Automatic Choice of Scheduling Heuristics for Parallel/Distributed Computing

Clayton S. Ferner[†] and Robert G. Babb II[‡]

Abstract

Task mapping and scheduling are two very difficult problems that must be addressed when a sequential program is transformed into a parallel program. Since these problems are NP-hard, compiler writers have opted to concentrate their efforts on optimizations that produce immediate gains in performance. As a result, current parallelizing compilers either use very simple methods to deal with task scheduling or they simply ignore it altogether. Unfortunately, the programmer does not have this luxury. The burden of repartitioning or rescheduling, should the compiler produce inefficient parallel code, lies entirely with the programmer.

We were able to create an algorithm (called a metaheuristic), which automatically chooses a scheduling heuristic for each input program. The metaheuristic produces better schedules in general than the heuristics upon which it is based. This technique was tested on a suite of real scientific programs written in SISAL and simulated on four different network configurations. Averaged over all of the test cases, the metaheuristic out-performed all eight underlying scheduling algorithms; beating the best one by 2%, 12%, 13%, and 3% on the four separate network configurations. It is able to do this, not always by picking the best heuristic, but rather by avoiding the heuristics when they would produce very poor schedules. For example, while the metaheuristic only picked the best algorithm about 50% of the time for the

[†] Lucent Technologies Inc., 11900 N. Pecos St., Westminster, CO 80234, cferner@lucent.com.
[‡] Dept. Of Mathematics and Computer Science, University of Denver, Denver, CO 80208, babb@cs.du.edu.

100 Gbps Ethernet, its worst decision was only 49% away from optimal. In contrast, the best of the eight scheduling algorithms was optimal 30% of the time, but its worst decision was 844% away from optimal.

1. Introduction

The difficulty with obtaining high performance from a scientific program on a parallel computer arises because several very difficult problems must be addressed simultaneously: (1) how to divide a program into individual tasks that can (potentially) be executed by different processors, which is known as the task partitioning problem, (2) how those tasks will be assigned to processors, which is known as the mapping problem, and (3) in what order will the tasks be executed, which is known as the scheduling problem. All three of these problems are NP-hard. Unfortunately, the only practical means by which compilers can address such difficult problems automatically is through the application of heuristics.

Over the past forty years, the computer science research community has produced a plethora of scheduling heuristics for parallel computing. As if this were not enough, Ieumwananonthachai et al. [25 and 26] show how to automatically generate new scheduling heuristics using Yan's "Post-game Analysis" [52]. This suggests that the number of possible heuristics for static scheduling is essentially unlimited. Ideally, a compiler would be able to choose from among any of a large set of scheduling heuristics to employ when producing executable code for a parallel application. However, it is not clear how a compiler would go about choosing the best scheduling heuristic for a given program. One obvious solution would be for a compiler to generate all possible schedules from all available schedulers and choose the one with the best predicted performance. However, there are at least two problems with this brute force approach:

(1) Many of the best known heuristics take a significant amount of time to compute a schedule.

(2) Predicted best schedule performance does not always correspond to actual best performance due to communication delays and conflicts, unknown or variable task execution times, I/O, etc.

Problem (1) implies that a compiler may need to make the decision of which scheduler to use without knowing what the resulting schedules will look like. Solving (2) could require executing the same program multiple times to determine the optimal schedule, which is obviously not very attractive.

As it turns out, most production compilers do not address all three of these problems. An example is the Fx compiler from Carnegie Mellon University [45, 46, 47], which does mapping but assumes a one-to-one mapping of tasks to sets of processors and therefore ignores scheduling. Even compilers that deal with scheduling, such as the Paradign compiler [38], use a single simple heuristic (e.g. list-scheduling). Part of the reason for this is that the majority of heuristics in the literature are academic research projects that are not industrial strength. Yet, even with more sophisticated scheduling techniques, compiler writers are most likely to implement a single heuristic which will be used on all input.

The goal of this research is to answer the question of: How does one make the choice of which scheduling approach to apply to any given program executed on any given machine? The literature on parallel computing indicates that the choice of a scheduling heuristic can have a significant effect on the parallel speed-up (or slow-down) that is achieved for a given application on a particular machine configuration. Being able to select the best heuristic for a given situation is therefore a very important step. If there exists a single superior scheduling heuristic, it is not apparent from reading the literature on scheduling techniques. A few papers have made comparisons of different approaches; although some perform better than others, there appears to be no consistent winner. Nonetheless, if an almost universally superior heuristic could be found, it would likely be a significant contribution to the current state-of-the-art in scheduling theory.

In this paper, we present one method by which a parallelizing compiler might automatically make the choice of a suitable scheduling heuristic on a per-program basis. The approach we have taken in this research relies on statistical methods. A variety of metrics were taken from sample programs and compared against the performance of the resulting schedules of a particular scheduler. If such correlations exist, then the execution times of schedules produced by a particular heuristic can be predicted to some level of accuracy. If several heuristics can be predicted with good accuracy, then the regression analysis may be used to choose the best heuristic for a particular program based on the metrics.

One major difference between our approach and the methods used in the past is that we treat the scheduling heuristics as black boxes. Previous researchers have used theoretical models based on their analysis of the heuristic algorithms involved. Our model is based solely on the performance of the resulting program code.

The rest of this paper is organized as follows: Section 2 summarizes the related research in this area, Section 3 describes the experiment, Section 4 discusses the results of the experiments, and Section 5 presents conclusions.

2. Related Research

There are many areas of scheduling theory. This research is focused on static scheduling of task graphs onto distributed systems. Table 1 lists many of the static scheduling heuristics from the literature. For a more complete survey of the various scheduling areas, including static scheduling, see [11], [10], [3], [31], [49], and [15].

Table 1: Table of Static Scheduling Heuristics.

Scheduler Heuristic	Abbr.	Reference
List Scheduling		
Hu's Algorithm	HU	[23]
Yu's Algorithm	YU	[56]
Insertion Scheduling Heuristic	ISH	[30]
Earliest Task First	ETF	[24]
Duplication Scheduling Heuristic	DSH	[30]
The Mapping Heuristic	MH	[14], [32]
Dynamic Level Scheduling	DLS	[44]
Prioritized Scheduling Algorithm	PSA	[38]
Selvakumar and Murthy's Algorithm	—	[41]
Preferred Path Selection	PPS	[33]
Critical Path		
Modified Critical Path	MCP	[50]
Mobility-Directed	MD	[50]
Kim & Browne's Algorithm	—	[29]
Sarkar's Algorithm	—	[39], [40]
Heavy Node First	HNF	[42], [43]
Weighted Length	WL	[42], [43]
Dominant Sequence Clustering	DSC	[53], [54], [55]
Change Critical Tasks	CCT	[38]
Task Clustering/Reassignment Algorithm	TCA/TRA	[51]
Cluster-M	—	[9]
Palis et al. Algorithm	—	[36]
Annealing		
Simulated Annealing	(SA)	[12]
Mean Field Annealing	(MFA)	[6]
Miscellaneous		
McCreary et. al. Algorithm	CLANS	[34]
Chaudhary and Aggarwal Algorithm	—	[8]
Hou, Ansari, and Ren Algorithm	—	[22]

One area of scheduling research that has not received a great deal of attention is in the comparison of different static scheduling strategies. Comparisons of static scheduling heuristics can be found in these references: [1], [37], [26], [19], [20], [21], [48], [4], [44], [35], [28], [13], [15], [33], [9], and [2]. Several of these works ([48], [35], [27], [28], and [2]) have suggested that various heuristics will perform better in some situations than others. These papers have attempted to discover under which circumstances one particular heuristic should be used over another.

This survey of the literature on static scheduling suggests several things:

(1) There are many heuristics (not all industrial strength) from which to choose, but there have only been a handful of comparisons of static scheduling heuristics done.

(2) Most of the comparisons have been done with the intention of demonstrating the effectiveness of a particular scheduling heuristic by showing its (supposed) superiority to other heuristics.

(3) Most of these comparison papers used either a small sample of application programs or did not use real applications (i.e., generated task graphs).

(4) There have been only a few papers suggesting that the choice of which scheduler to use can be done automatically.

We designed an experiment to determine if statistical methods can be used to predict the results of various static scheduling heuristics well enough to make a good choice for which heuristic to use for a particular program. We attempted a balance between realism and sample size. The input programs for our experiments were based on real programs and not on randomly generated task graphs.

3. Experimental Plan

This section describes the experiment that was used to verify how well the idea proposed in

Section 1 for automatically choosing a scheduling heuristic will work. This was a preliminary experiment designed to verify how well this approach might work. We used simulation as a means of obtaining estimates of the run-times of the resulting schedules. Our results todate indicate that this approach could be useful in a production compiler sometime in the near future.

3.1 Task Graph Metrics

Ferner [17] proposed a collection of 41 different task graph metrics that could potentially be predictors of schedule lengths. The majority of these metrics were only marginally related to the quality of the resulting schedules. The two most important metrics (which are used throughout the rest of this paper) are the number edges in the task graph (denoted e) and the total graph weight of the node weights only (denoted TGW-w) or the sum of the nodes weights.

Both of these metrics are easy to calculate in terms of the complexity. Once the task graph is computed, TGW-w can be computed in $\mathbf{O}(v + e)$ (where v is the number of nodes), and e obviously can be calculated in $\mathbf{O}(e)$. However, it is trivial to maintain these metrics as the task graph is being built, in which case the complexity of both is $\mathbf{O}(1)$. Nonetheless, the amount of work needed to derive both metrics is insignificant compared to the amount of work already performed by a typical compiler.

3.2 Library of Scheduling Heuristics

Table 2 shows the set of scheduling heuristics used in the experiment. Although this list is not long, it includes several of the more important heuristics known. The choice of which heuristics to include was determined by the availability. Specifically, the authors were willing to provide an implementation.

Table 2: Scheduler Heuristic Approaches Considered in this Project.

Scheduler	Type	Reference
CLANS	graph decomposition	[34]
DSC	critical path	[54]
MH	modified list scheduling	[14]
HU	list scheduling	[23]
ISH	list scheduling	[30]
RAN1	random list scheduling	—
RAN2	random list scheduling	—
SEQ	sequential	—

The two random schedulers are both list schedulers but use a random number generator to assign priorities to nodes in the ready queue. Successive runs of these schedulers on the same input may result in different outputs. The SEQ scheduler is actually not a parallel scheduler. It assigns all tasks to the same processor and therefore provides a sequential version of the program. This scheduler (and the random schedulers) is included in this study to serve as a control. It is very interesting that occasionally more sophisticated algorithms do not fair well against these simpler ones.

3.3 The Metaheuristic

Figure 1 presents an algorithm for choosing which scheduler should be used based on the predicted values of the schedule lengths. This algorithm is referred to as the *metaheuristic*, because it is a heuristic for choosing a heuristic.

Input: Task graph representing a program
Output: Choice of which scheduling heuristic to apply

Method:
1. Compute the appropriate metrics on the task graph
2. Plug the metrics into the regression model for each scheduler to derive a predicted schedule length
3. Choose the scheduler with the smallest predicted schedule length

Figure 1: Algorithm of the Metaheuristic.

Although the scheduling heuristics need to be run and the resulting schedules executed in order to create the initial regression models, this is no longer the case once the models have been derived. In fact, none of the scheduling heuristics needs to be executed to derive the suggestion. The metrics are relatively inexpensive (in terms of complexity) to obtain from the task graph and the regression model used in the next section can be calculated in constant time. The complexity of this algorithm is linear with respect to the number of scheduling heuristics in the library.

3.4 Overview

Figure 2 shows a schematic of the main steps for the experiment. These steps are described in more detail below. The two factors to emphasize in this approach are the importance of a large sample space and a reasonable level of realism. To address this difficulty, the input used in this research are examples of scientific applications written in SISAL (see [16] and [7] for more information about SISAL). The SISAL compiler front-end was used to generate large, realistic task graphs called IF1 graphs.

Unfortunately, IF1 graphs are hierarchical, which means that nodes may be complex or simple. Complex nodes are used to represent such constructs as loops. Not all of the schedulers have been designed to deal with this more abstract graph. Therefore, the IF1 must be converted into a simple task graph using a program called Flatten. Flatten takes as input an IF1 graph, unrolls the loops to specified depths, converts the complex nodes into simple nodes (i.e., removes the graph hierarchy), and assigns weights to the edges and nodes. Each node corresponds to a single instruction. The node weights were assigned from a table of known weights for each instruction. This table was constructed by timing these instructions on a Sun SparcServer 690MP. The edge weight values are the number of bytes associated with the

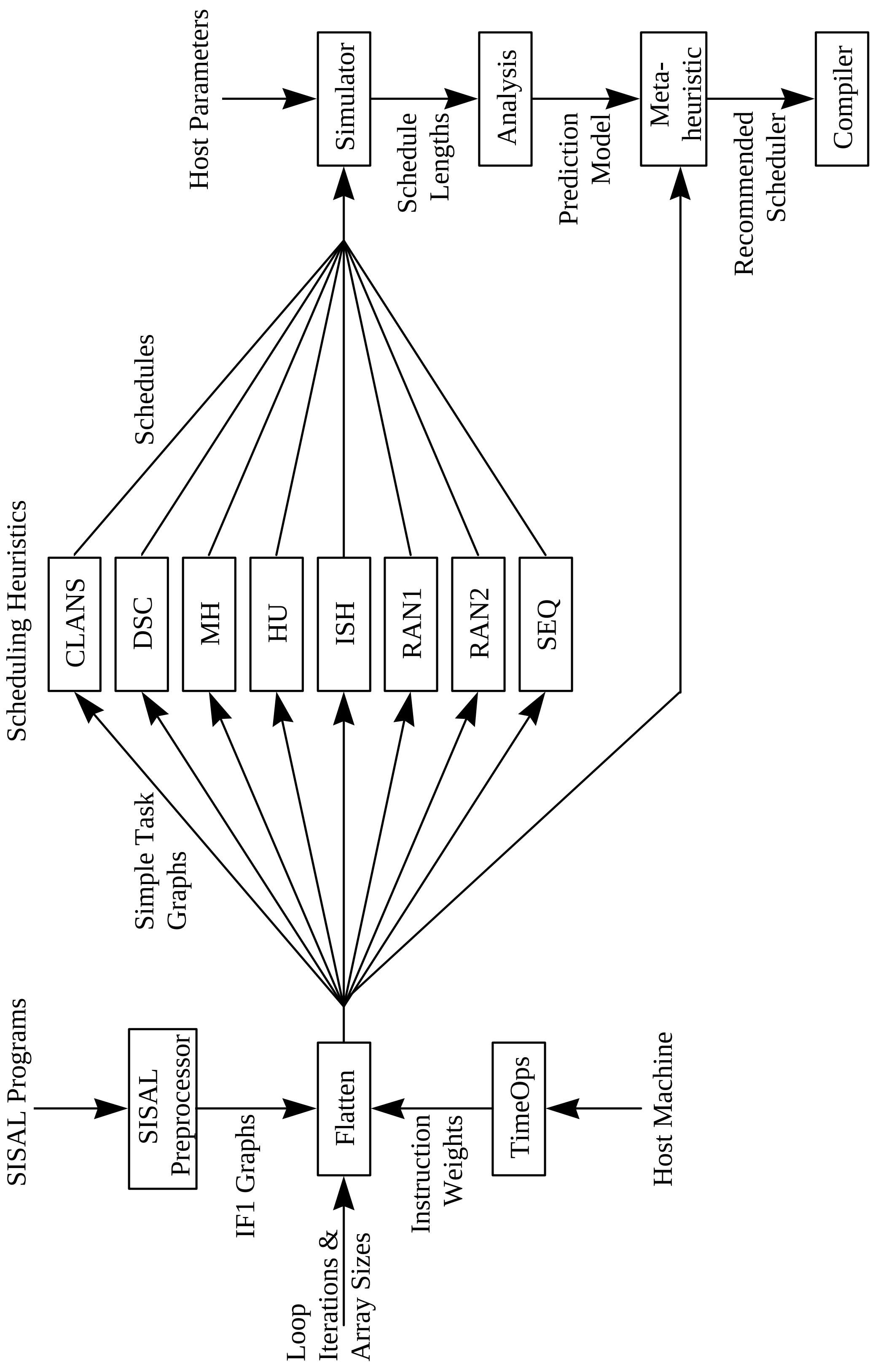


Figure 2: Experimental Plan

data-dependence between instructions, and arrays are assumed to be fixed in size. Since the loops had to be unrolled in order to create a simple task graph, some realism has been lost. Forcing the loop bounds to be static is a rather restrictive requirement. However, various loop-unrolling schemes were tried and their effect on the scheduler performance was investigated (see Section 4.4).

Once the task graphs which represent the scientific programs were produced, they were provided to each of the various schedulers as input to produce different schedules. The metric that was used in this experiment to compare each schedule to determine the best one was the wall-clock execution time, or makespan, of the schedule. However, this measurement can vary from execution to execution, even for the same executable program. This is especially true in a multi-user environment. In order to make a fair comparison of the schedules produced by the eight schedulers (and therefore keep the experiment a controlled experiment) simulation was required. Two distributed-memory architectures were implemented in the simulation: Ethernet and Mesh.

The simulation produced the total execution time of the schedule taking into account processor speeds, network topology, bandwidths, message overhead, and channel contention. All processors were considered to be homogeneous with a scaling factor of 1. Since CLANS assumed an infinite number of available processors, the maximum number of available processors was 256 for all scheduling heuristics. This value was large enough that each heuristic was able to provide a schedule with as many processors as it needed. Unfortunately, we do not know how this approach will perform with a limited number of processors. We plan to investigate the issue of limiting the number of processors later.

Three models of bandwidth were considered: 10 Mbps, 1 Gbps, and 100 Gbps. The communication overhead for the 10 Mbps bandwidth was approximately 800 microseconds. This value was calculated by transmitting messages on a 10Base500 shared Ethernet. The overhead was decreased by a factor of 100 as the bandwidth increased.

Although the 10Mbps bandwidth was studied in this research, the results are not presented in this paper. The reason for this is because the communication costs were too large to make parallel computation practical. All of the parallel schedulers produced worse schedules than sequential execution for this bandwidth on all input. Therefore, only the results of the 1 Gbps and 100 Gbps bandwidth models are presented. The next subsection discusses this issue further.

3.5 Input Data Sets

Two separate data sets were used in the experiments. The first data set was used to determine whether there are any significant correlations between the proposed metrics and schedule length and, if so, to derive the regression models.

A second data set was needed to verify the results. The metaheuristic provided suggestions for which scheduling heuristic should be used for each input in this second set. However, in order to verify the success of the metaheuristic, each input still needed to be run through each heuristic and each schedule simulated. Nonetheless, the metaheuristic made its choices based solely on the analysis done on the first set and not on the results of the simulation of the second set.

The SISAL programs used in this research were provided by the SISAL group at Lawrence Livermore National Lab, and are listed in Table 3. Several of these programs were taken from a test suite described in [7]. Fifteen of the smallest programs in the set of

Table 3: Input Application Programs Written in SISAL.

Subject Area	**Program**	**Subject Area**	**Program**
Approximation	Parpi1	Parsing	Cyk
of Pi	Parpi2	Matrix Arithmetic	Inverse
	Parpi_Babb		Matmult
Fourier Transform	Dft		Mmult2
	Badfft		Transpose
	Cfft		Conv
Game of Life	Life1	Primes	Arsieve
	Life2	Random Num. Gen.	Ranf
Gauss-Jordan	Gauss	Scientific Kernels	Loops
Elimination	Gaussdata	Searching	Minmax
	Gaussjnew	Sorting	Batcher
	Lu		Bubble
	Lunpiv		Insertion1
	Lupiv		Insertion2
	Stand_Alone_Gauss		Mesort
Gel Chromatography	Ricard		Pinsert
Gel Electrophoresis	Kin16		Pinsertdata
Hamming	Ham		Simplebatcher
Hilbert	Data	Sparse Matrix	Sp
Integration	Area		Spinit
Laplace	Laplace	Square Root	Sqrt
Pattern Matching	Noise		Test
	Noisedump	Text processing	Wordcount
		Vector Arithmetic	Vectest

scientific kernels called LOOPS were used as the first data set. These programs were unrolled using six different unrolling schemes shown in Table 4, resulting in a total of ninety input task graphs. Array sizes were fixed at 100 for all programs. The remaining programs[1] were used in Data Set 2 with unrolling schemes one and two.

The values in the table are the number of iterations that the loops at each nesting level were unrolled, and the maximum loop nesting for both suites of programs was 10 levels. For example, with scheme 6, the outermost loops were unrolled 6 iterations, the next level loops were unrolled 5 iterations, and the inner-most loops were unrolled 1 iteration. These values are much

[1] The choice of data sets was somewhat arbitrary. The LOOPS kernels were available from Lawrence Livermore's library of SISAL programs at the beginning of this experiment. Most of the remaining programs only became available later.

smaller than the number of iterations each loop would likely be executed when run on a real problem. Unfortunately, full unrolling of the loops would create an explosion of task nodes in the task graph. Therefore, these limited unrolling schemes were chosen for practical reasons. Furthermore, many of the scheduling heuristics require a large amount of time to compute a schedule from the task graphs produced in the experiment.

Table 4: Loop Unrolling Schemes.

	Scheme					
Loop Level	**1**	**2**	**3**	**4**	**5**	**6**
1	1	2	3	2	5	6
2	1	2	3	2	5	5
3	1	2	3	5	5	4
4	1	2	3	5	5	3
5	1	2	3	5	10	2
6	1	2	3	5	10	1
7	1	2	3	5	10	1
8	1	2	3	5	10	1
9	1	2	3	5	10	1
10	1	2	3	5	10	1

Since the number of iterations was fixed, an important question that arises is whether this had a significant impact on the results. We investigated the influence of the different unrolling schemes on the prediction model that is presented in the next section and found that the model was fairly insensitive to the schemes (see Section 4.4.)

The method used to derive these task graphs automatically yields extremely fine-grained graphs (i.e. each node is a single instruction). One result of this was that the parallel schedulers produced schedules that were slower than sequential execution on a single processors for all input for the 10 Mbps bandwidth. We omitted those results from the next section because we did not learn anything substantial. It is apparent that the input programs need to be re-partitioned into coarser-grained task graphs in order to reduce the amount of interprocessor communication for this slower network. However, partitioning was specifically left out of the scope of this research so that the experiment would be a more manageable size.

One way to increase the effective granularity would have been to increase the weights of the nodes in the task graphs across the board. However, an equivalent solution is to increase the communication speeds of the topologies. This was the path taken in this experiment, since 1 Gbps is almost available with today's technology and 100 Gbps is approximately the latency of shared memory.

Unfortunately, partitioning and scheduling are inter-related problems. Had we chosen a different partitioning scheme, we likely would have produced completely different results. Regardless of how we chose to partition we would not be able to determine if it was the most efficient partitioning short of trying every known scheme. This simple partitioning we used to create input task graphs is a starting point for more experiments of this sort.

4. Results

In this section, the results of the experiment are presented. First, the performance of the eight schedulers on data set 1 are shown. Then the derivation of the prediction model is provided. The predictions on the second data set are presented against the actual results. The last two subsections present two sanity checks.

4.1 Scheduler Performance (Data Set 1)

Figure 3 and Figure 4 compare the performance of the eight scheduling heuristics relative to each other on data set 1. Figure 3 shows the frequency that each scheduler produced an optimal schedule, and Figure 4 shows how far from optimal these schedules were. The term *optimal* is used here to mean *relatively optimal* and refers to the schedule that provided the shortest total execution time from among the eight produced by the scheduling heuristics ($\text{OPT}_i = \min_j \{X_{i,j}\}$, where $X_{i,j}$ is the schedule length for task graph *i* of the schedule produced by heuristic *j*). It is not necessarily the true optimal schedule over all possible schedules, which is unknown in general.

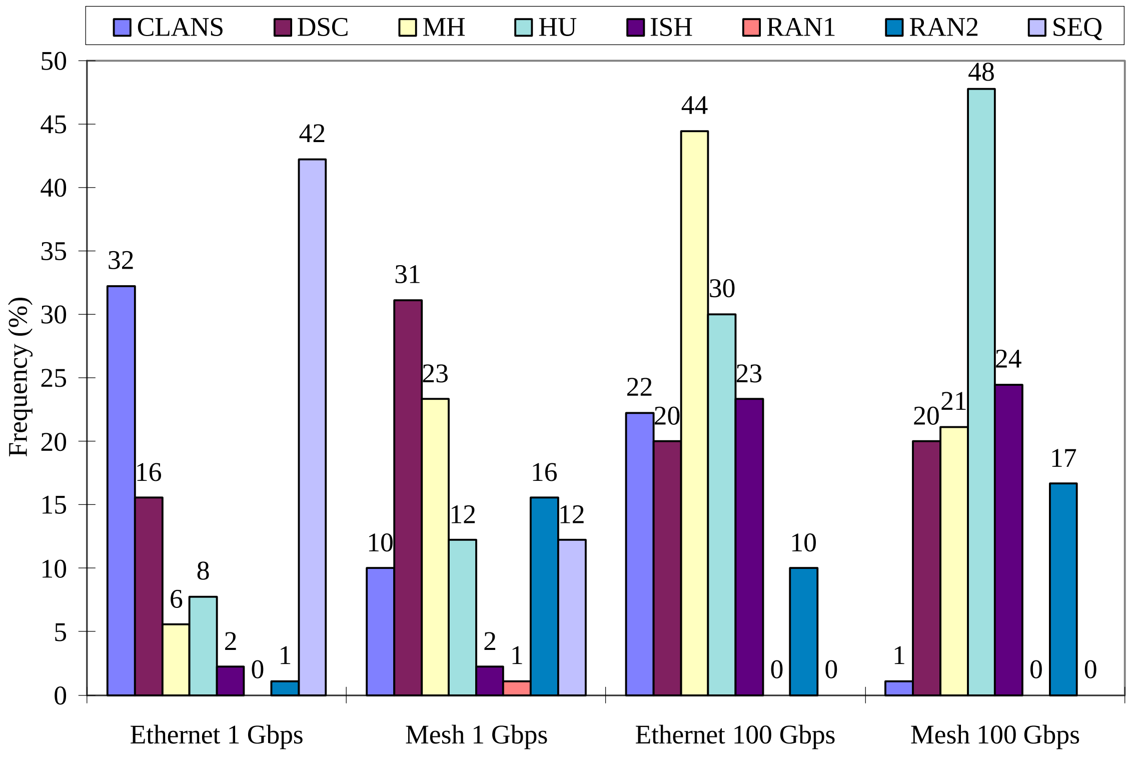


Figure 3: Number of Optimal Schedules for Data Set 1. Since more than one scheduling heuristic can create schedules that are equivalent in length to the best schedule, the percentages can sum to more than 100 percent.

The *slowdown from optimal factor* is calculated as:

$$1 + \frac{\sum_{i=1}^{N} \frac{X_{i,j} - \mathrm{OPT}_i}{\mathrm{OPT}_i}}{N}$$

This value indicates how much worse the schedules were for each heuristic than the optimal schedules. A factor of 1.0 means the scheduler produced the best schedules for all input.

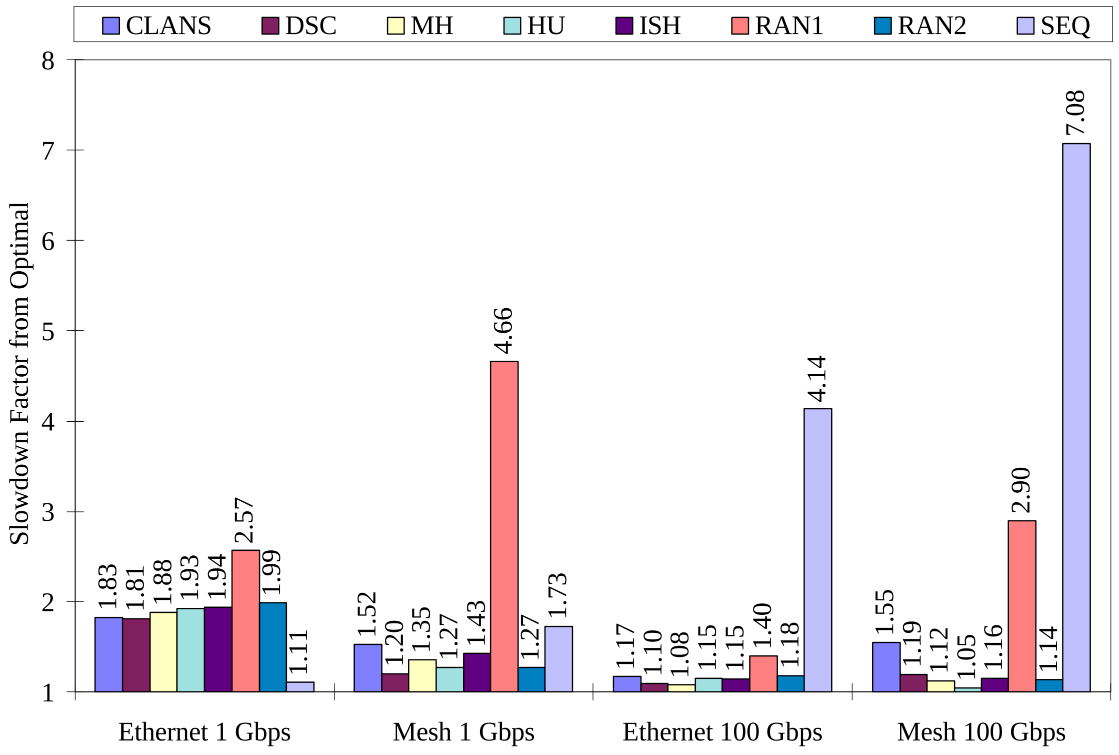


Figure 4: Slowdown Factors From Optimal for Data Set 1.

The RAN1 heuristic was the clear looser. In only one case did it produce a schedule that was equivalent to the optimal schedule. However, for this one input, several of the other heuristics also produced a schedule equivalent to the optimal one. The slowdown factors for RAN1 clearly demonstrate that this heuristic should probably be avoided.

The sequential scheduler does well for the 1 Gbps bandwidths and poorly for the 100 Gbps bandwidths, and even worse for Mesh than for Ethernet. It should not be surprising that the faster bandwidth allows for greater potential speedup for parallel execution, which gives the parallel heuristics an advantage over sequential. The Mesh also provides additional potential speedup due to a greater number of communication links in the topology.

The remaining parallel schedulers (other than RAN1) are fairly competitive relative to each other. In other words, the relative gain in performance of using one of these remaining parallel schedulers over another is fairly small. More importantly, there appears to be no clear winner. Although several heuristics perform well in general (e.g. DSC), the best scheduling heuristic varies from environment to environment (even when ignoring RAN1 and SEQ).

4.2 Prediction Model

This section provides a sample (due to lack of space) of the correlations found between some of the task graph metrics and the schedule lengths produced by the various heuristics. Also presented in this subsection is the model used to predict the schedule lengths on data set 2.

4.2.1 Correlation Coefficients

Each of the 41 metrics was compared to the schedule lengths for each scheduler on each topology for all 90 input programs. Only several of the stronger correlations are shown in Figure 5 for the 1 Gbps Ethernet. For a complete list of these correlation coefficients see [17].

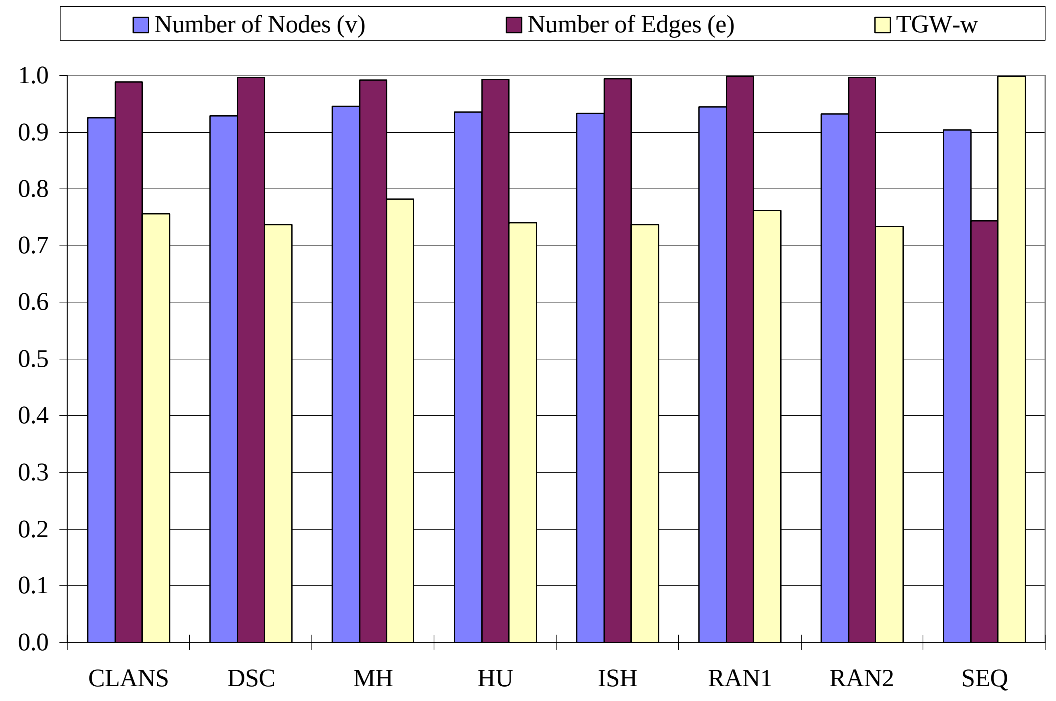


Figure 5: Correlation Coefficients. These are several of the stronger correlated metrics for 1 Gbps Ethernet.

The number of edges is a very good predictor for most of the parallel scheduling heuristics. The strength of these correlations exceeded expectation. In fact, it is very surprising that both random schedulers have correlations very close to 1.0 (although this is not the case for RAN1 on the Mesh).

Notice that the lengths of the sequential schedules are exactly the sum of the node weights. This should not be surprising since there is assumed to be no communication overhead associated with single processor execution, and the node weights are a representation of the amount of work that needs to be done. It is not likely that any other model would be necessary to predict the sequential execution.

Figure 6 shows the scatter graph of the CLANS heuristic on the 1 Gbps bandwidth Ethernet. This graph is typical of the scatter graphs for the other environments and heuristics.

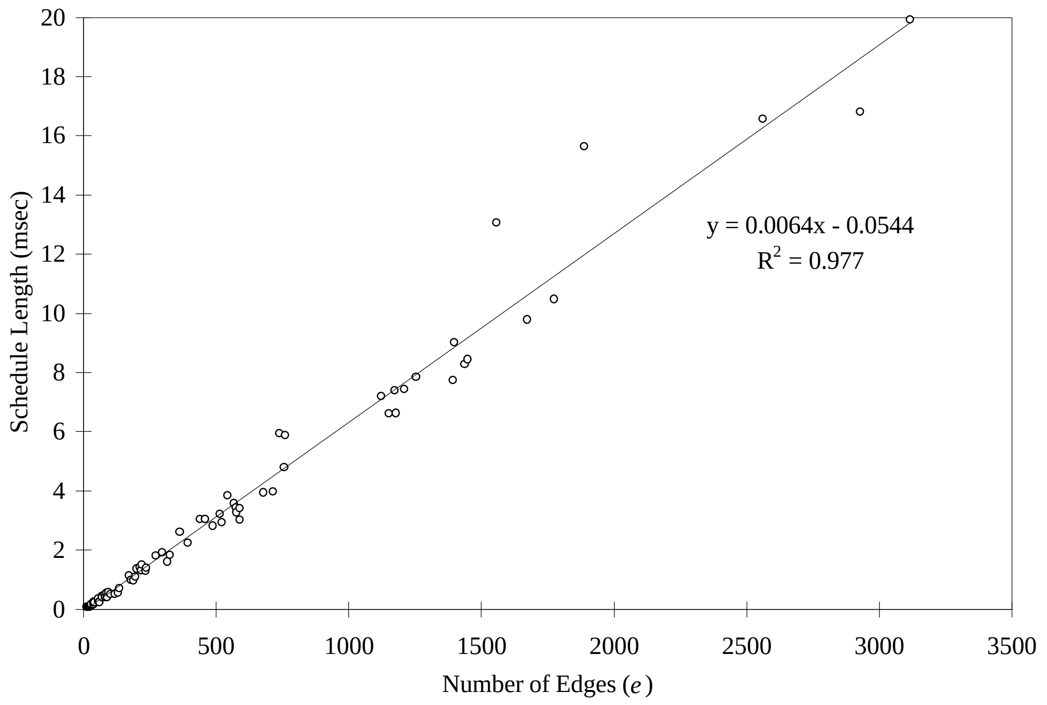


Figure 6: Scatter Graph of CLANS Heuristic for 1 Gbps Bandwidth Ethernet. The line is the least-squares regression line.

The remaining scatter graphs can be found in [17]. This graph is presented here to further emphasize the strength of the correlation between the schedule lengths and the number of edges.

4.2.2 Regression Model

Although several different regression models were tried, only the simplest one will be discussed in detail in this paper, because more complex models (e.g., multivariate) did not offer significant improvement in choosing schedulers. Since the number of edges appears to have the strongest correlations overall for the parallel scheduler, it was chosen as the metric to be used to attempt to predict the schedule lengths. However, the sequential scheduler was predicted by using the sum of the node weights, since the correlation coefficient is 1.

The equations of the least-squares regression lines (between the appropriate metric and the schedule lengths) and resulting R^2 values are shown in Table 5. The purpose of showing this table here is not to provide the equations for predicting these heuristics in general (further research is probably needed to fine tune them), but rather to emphasize of the simplicity of the metaheuristic. Both the number of edges (e) and the sum of the task graph (TGW-w) weights are very easy to calculate. In fact, since a compiler will need to create the task graph in the first place, these two metrics are essentially free. The complexity of the metaheuristic is therefore *linearly* proportional to the number of heuristics available in the library.

As we continue our research, we expect the regression models to change for different categories of input. New categories (other than topology) that we think may be useful (although we do not know at this point) are:

(1) the partitioning algorithm used and/or the granularity,

(2) the problem domain (i.e. Gaussian elimination, weather, Hilbert spaces, etc.),

(3) the general shape of the task graph,

(4) the time of the day (i.e. how loaded the system is),

(5) the programming style of the programmer.

Table 5: Regression Equations and R^2 Values. *e* represents the number of edges and TGW-w represents the sum of the node weights.

Scheduling	**Ethernet**		**Mesh**	
Heuristic	**R^2**	**Equation (µsec)**	**R^2**	**Equation (µsec)**
		1 Gbps		
CLANS	0.977	$6.37 \times e - 54.40$	0.885	$2.98 \times e + 181.23$
DSC	0.995	$6.31 \times e - 48.33$	0.940	$2.28 \times e + 178.31$
MH	0.986	$6.42 \times e + 11.08$	0.848	$2.68 \times e + 146.17$
HU	0.987	$6.59 \times e + 4.78$	0.882	$2.52 \times e + 136.23$
ISH	0.989	$6.67 \times e - 18.26$	0.943	$3.00 \times e + 66.13$
RAN1	0.998	$8.78 \times e + 8.19$	0.748	$7.72 \times e - 384.67$
RAN2	0.994	$6.87 \times e - 30.25$	0.855	$2.30 \times e + 243.19$
SEQ	1.000	$0.10 \times \text{TGW-w} + 0.17$	1.000	$0.10 \times \text{TGW-w} + 0.17$
		100 Gbps		
CLANS	0.871	$0.97 \times e + 54.40$	0.750	$0.64 \times e + 92.10$
DSC	0.933	$0.96 \times e + 20.78$	0.819	$0.47 \times e + 83.94$
MH	0.957	$0.99 \times e - 1.55$	0.882	$0.53 \times e + 36.97$
HU	0.970	$1.02 \times e + 19.68$	0.872	$0.45 \times e + 53.51$
ISH	0.978	$0.99 \times e + 29.28$	0.908	$0.51 \times e + 52.01$
RAN1	0.973	$1.19 \times e + 46.91$	0.631	$1.14 \times e - 36.79$
RAN2	0.977	$1.06 \times e + 12.59$	0.841	$0.52 \times e + 42.43$
SEQ	1.000	$0.10 \times \text{TGW-w} + 0.17$	1.000	$0.10 \times \text{TGW-w} + 0.17$

4.3 Prediction Results (Data Set 2)

The regression equations from the previous subsection were used by the metaheuristic to attempt to predict the schedule lengths of the programs in the second data set for each of the eight schedulers. The metaheuristic made its choices for the best scheduler, then these choices were compared to the actual schedule lengths. The metaheuristic is presented in the following sections as though it were just another scheduler. Figure 7 and Figure 8 present the results of the metaheuristic choices for the best scheduling heuristic on Data Set 2.

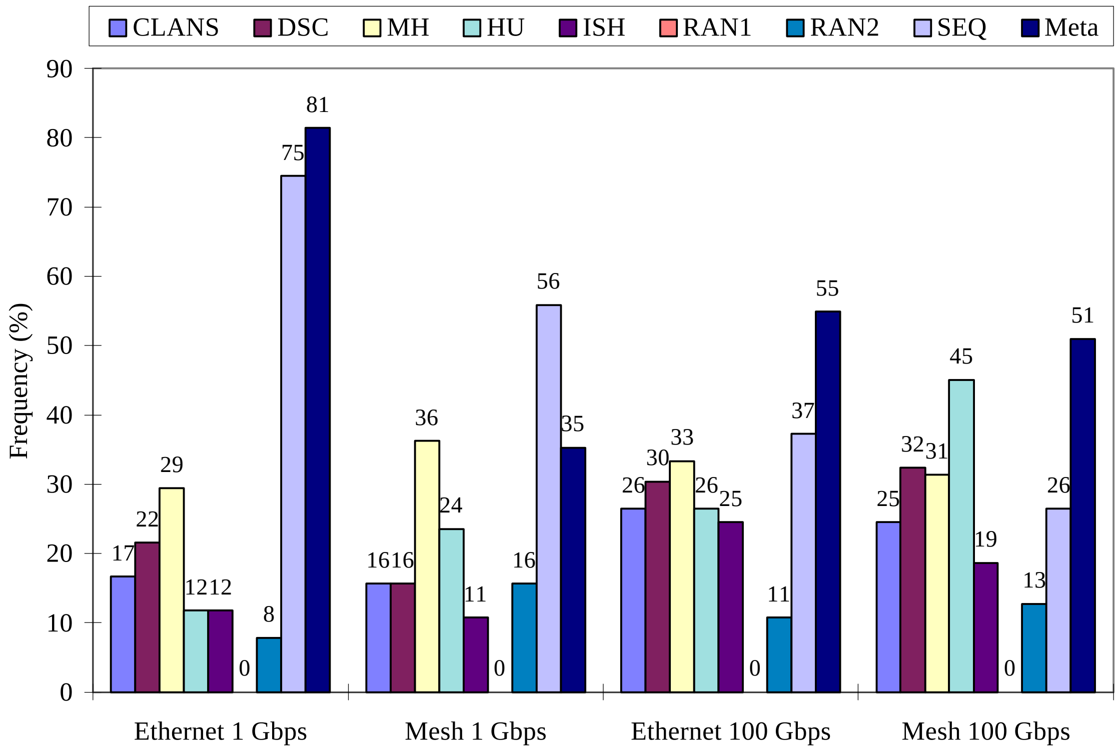


Figure 7: Number of Optimal Schedules for Data Set 2. Since more than one scheduling heuristic can create schedules that are equivalent in length to the best schedule, the percentages can sum to more than 100 percent.

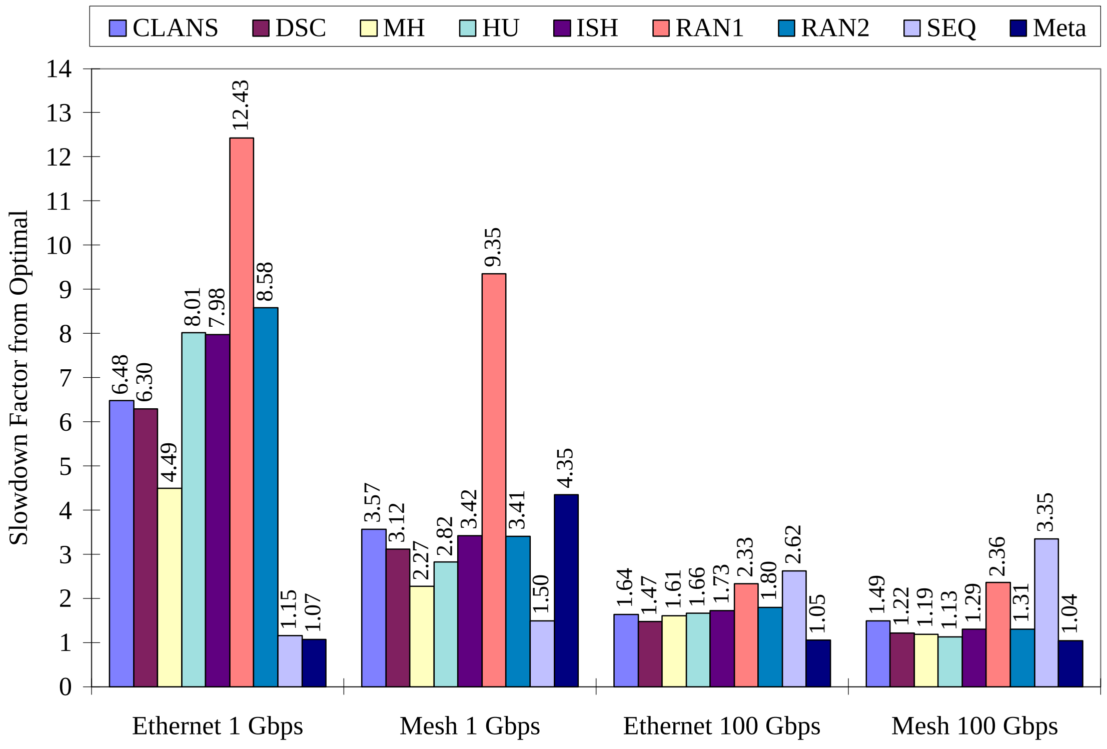


Figure 8: Slowdown Factors From Optimal for Data Set 2.

The metaheuristic provided overall better schedules than all of the other scheduling heuristics, with the exception of the 1 Gbps Mesh topology. The reason the metaheuristic did poorly for that environment is because it chose the RAN1 scheduling heuristic about 35 percent of the time. However, this makes no sense, since the RAN1 scheduler produced such poor schedules for data set 1. A closer examination of the regression for RAN1 revealed the cause of this. The steepness of the least-squares line produced a large negative y-intercept of the regression line (see appendix B of [17]), which in turn produced negative prediction values for RAN1. This brings up an interesting point: What do negative predictions for run-time mean? Nonetheless, it is clear that the RAN1 scheduler should not be used.

Figure 9 shows the results for the 1 Gbps Mesh if the RAN1 scheduling heuristic is removed from the library of available schedulers. Once again, the metaheuristic provided better schedules overall. Perhaps a more sophisticated prediction model should be used to deal with cases like the RAN1 scheduling heuristic. However, the simpler model still does an effective job of choosing the best scheduling heuristic for the most part. Certainly for the other three environments, the metaheuristic produced more optimal schedules and better performance than all the other heuristics. In fact, being able to know what the optimal schedule would be beforehand is only marginally better than using the metaheuristic.

Figure 10 shows the average normalized percent improvement of the metaheuristic over the eight scheduling heuristics upon which it is based. The formula used for this data is:

$$\frac{\sum_{i=1}^{N} \frac{X_{i,j} \text{ - META}_i}{X_{i,j}}}{N} \quad .$$

Instead of using the optimal schedule as the base, the schedule produced by each heuristic is the base.

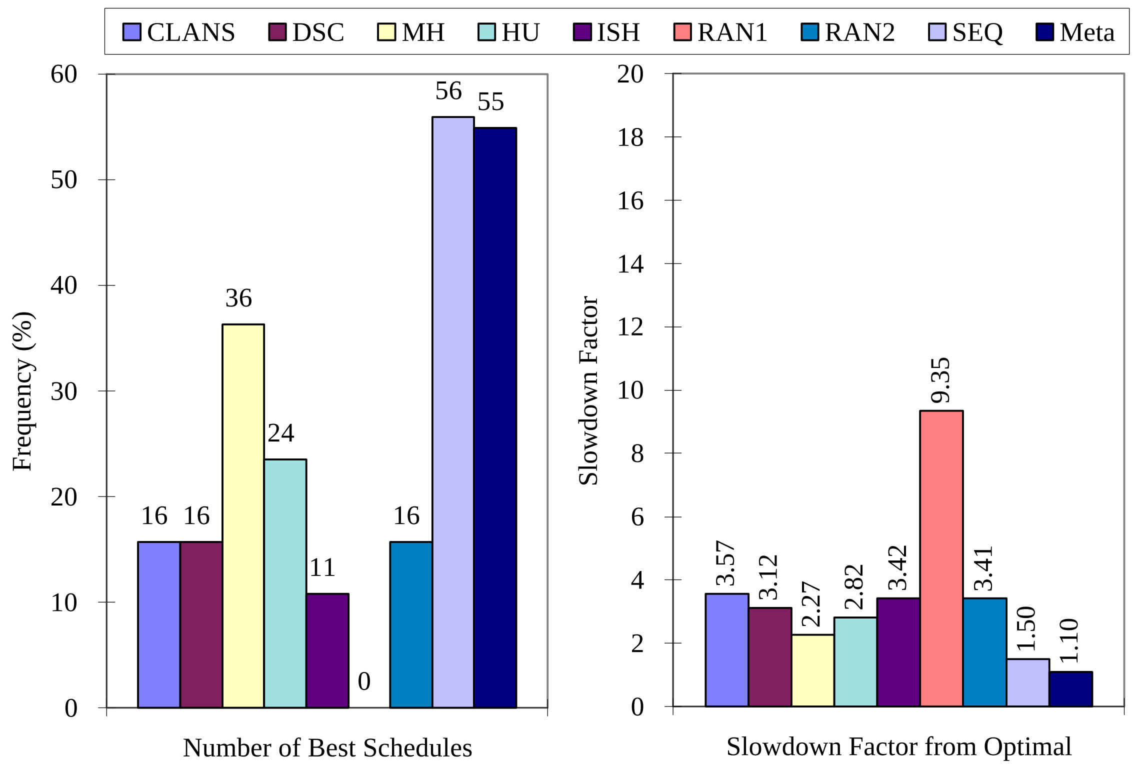


Figure 9: Metaheuristic Results on Mesh 1 Gbps with RAN1 Removed from the Library of Available Heuristics.

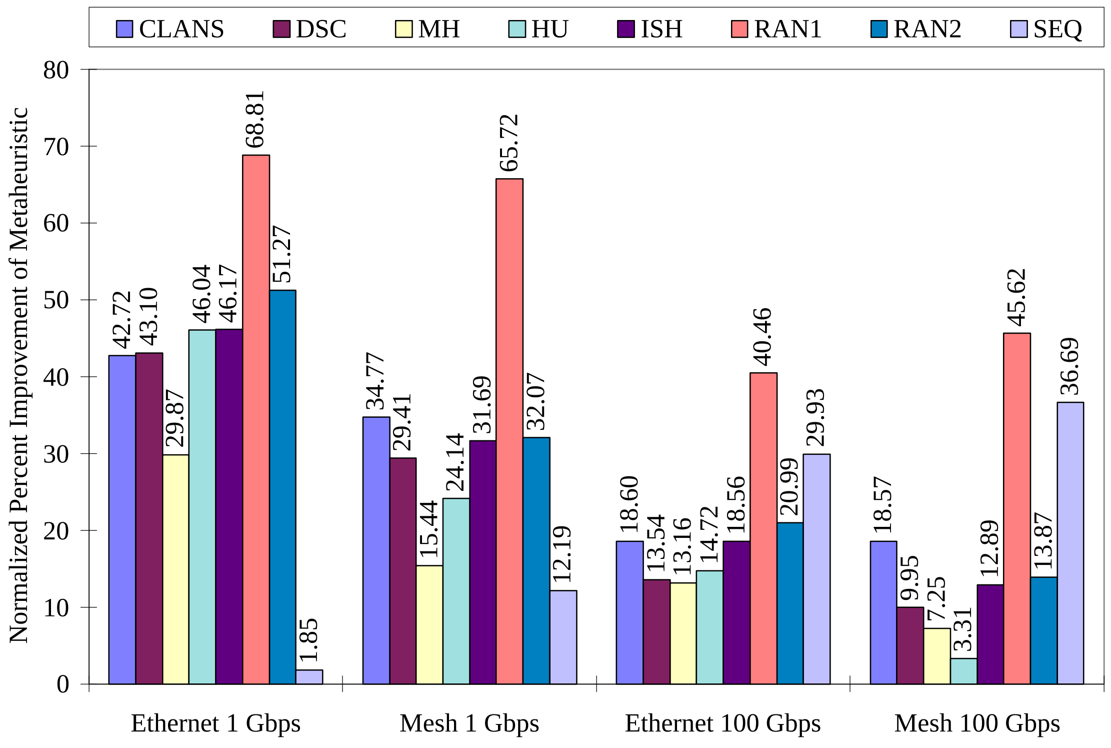


Figure 10: Normalized Percent Improvement of the Metaheuristic over the Other Scheduling Heuristics.

For the 100 Gbps bandwidth, the metaheuristic is correct about which scheduling heuristic will produce the best schedule about half of the time. Although this is more often than the other heuristics, it is not known whether this is a large or small percentage. Figure 11 shows the performance losses of each heuristic for each input in data set 2.

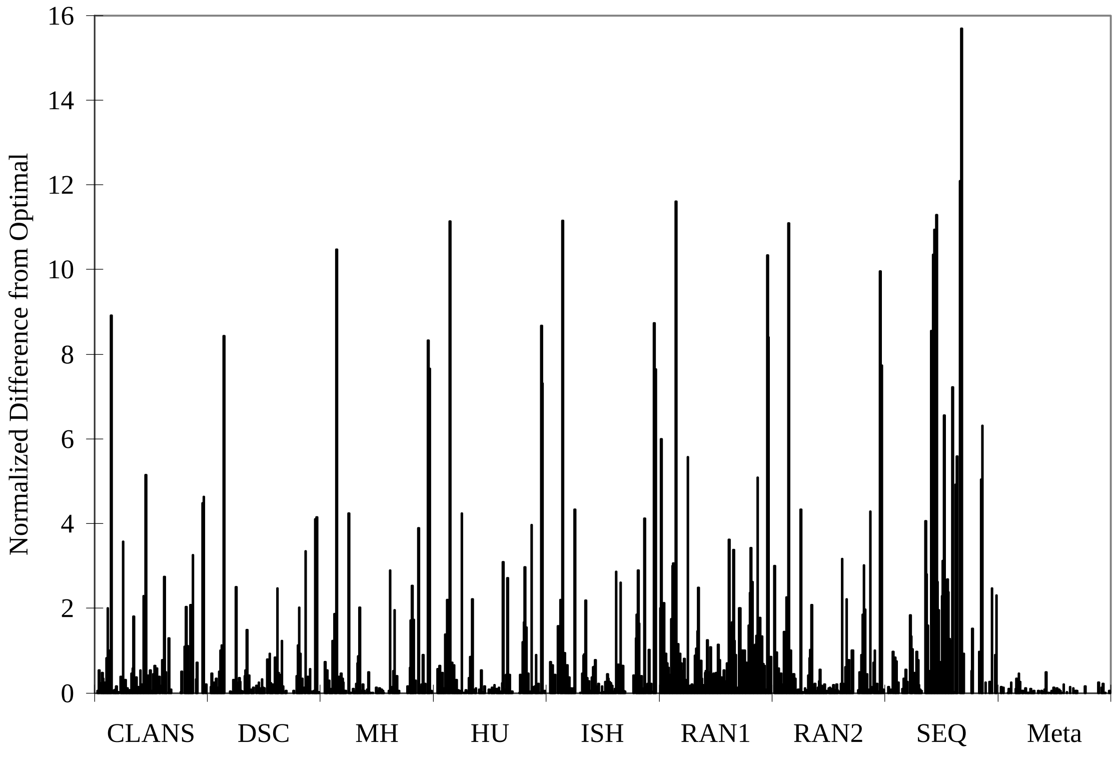


Figure 11: Normalized Differences from Optimal for each Scheduling Heuristic on 100 Gbps Ethernet. The normalized difference from optimal is ($X_{i,j}$ - OPT_i) / OPT_i for each input *i* for each scheduling heuristic *j*.

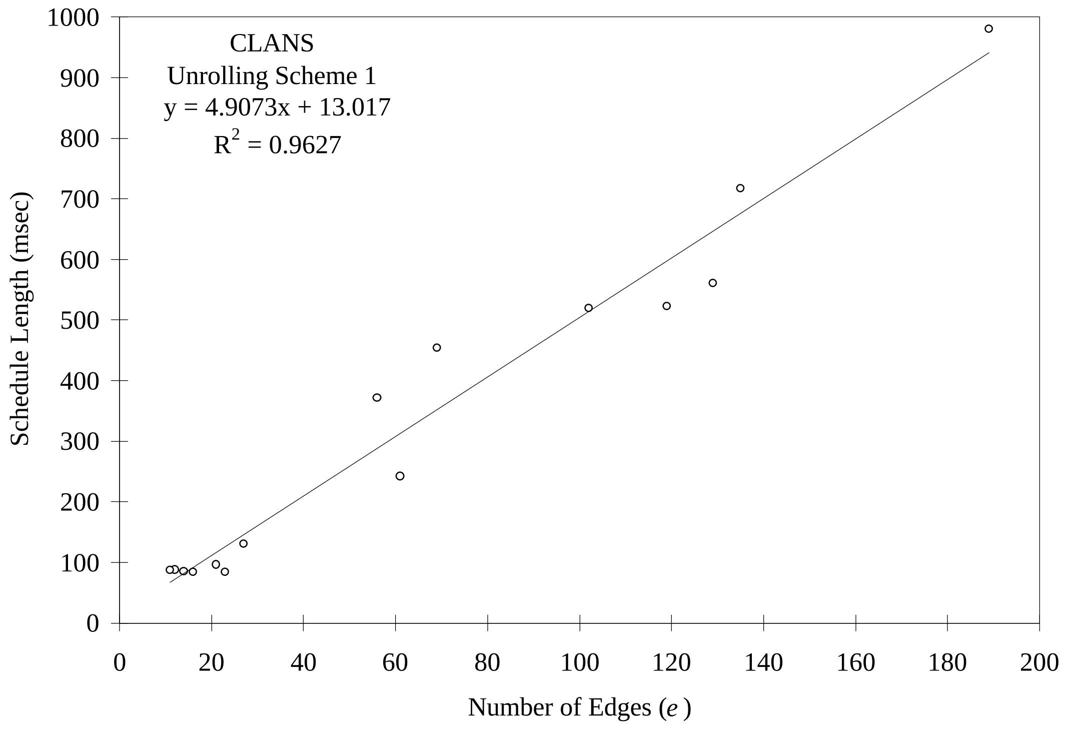
This discussion so far has demonstrated that the metaheuristic is providing good choices for which scheduling heuristic to use. It may be surprising that such a simple algorithm would be successful in doing this. However, when one takes a closer look into the data, it is not as surprising. Figure 11 shows that all of the original eight heuristics produced some poor schedules. However, the metaheuristic does not. Even though the metaheuristic is correct in choosing the best only half of the time, where it gains ground over the other heuristics is by *avoiding* the bad schedules. Furthermore, it does this for the large task graphs as well as the small ones.

4.4 Loop Unrolling

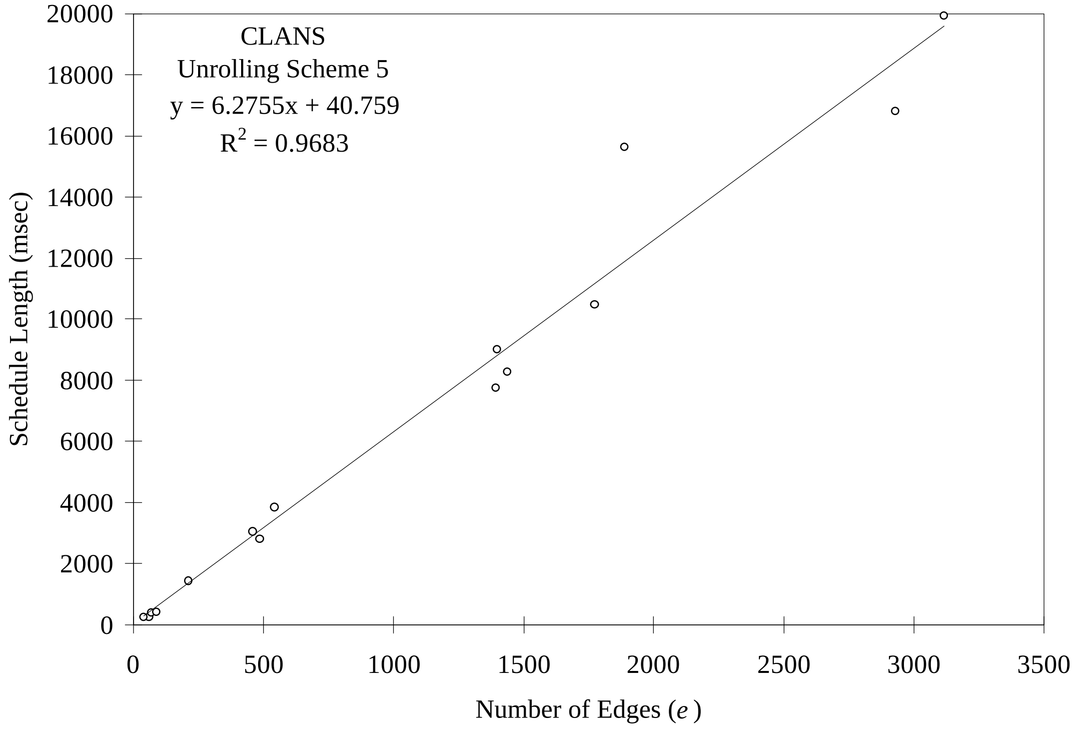
This section, as well as the next, provides results of two sanity checks. The task graphs that were used in this experiment were created from the IF1 graphs produced by the SISAL compiler. These graphs are hierarchical graphs. In other words, many of the complex nodes of the IF1 graphs represent loops, where the internal nodes represent the parts of the loops. The internal nodes may also be complex nodes as well as simple nodes. Since many of the scheduling heuristics for static scheduling do not handle loops (i.e., require a flattened task graph), the hierarchical graphs were converted into simple task graphs by unrolling the loops to pre-defined numbers of iterations.

The arbitrary unrolling of the loops could have an impact on the results of this research. To investigate this, the correlation coefficients were recomputed for each group of programs unrolled for each scheme, as was described in Section 3. The maximum correlation coefficients across the 41 metrics for each scheduling heuristic for each loop-unrolling scheme were computed. The strengths of the correlations fluctuated only slightly across the schemes. For the most part, the correlation coefficients were just as strong as the overall correlations. Although the particular metric that produced the highest correlation varies slightly from scheme to scheme, the number of edges still appears to be the best overall predictor.

Figure 12 shows the scatter graph for CLANS heuristic for 1 Gbps bandwidth Ethernet for the two extreme cases of the six unrolling schemes. (All six graphs for each heuristic are shown in [17].) One can see from these figures that some of the input programs exploded in size as the loops were unrolled to more iterations (due to nested loops), while others did not



(a) Unrolling Scheme 1



(b) Unrolling Scheme 5

Figure 12: Scatter Graphs of the CLANS Heuristic on 1 Gbps Bandwidth Ethernet for Two Loop Unrolling Schemes.

increase in size by much at all. More importantly, it appears that the correlations, and therefore the predictability, of the heuristics scale quite well. The predicted runtimes of these programs vary, but the regression model will not change significantly for larger loop-iteration sizes.

4.5 Simulation Sanity Check

Since the results of this work are entirely based on having fairly accurate estimations of the schedule lengths, one might ask, How good is the simulation? To test the accuracy, one of the schedules was implemented and run on a real Ethernet based network, using PVM 3.3 [see 5 and 18] as the communication medium. Figure 13 shows the results of timing the program during several different trial runs during different times of the day with different numbers of users on the system. Figure 13 also shows the predicted run-time provided by the simulation.

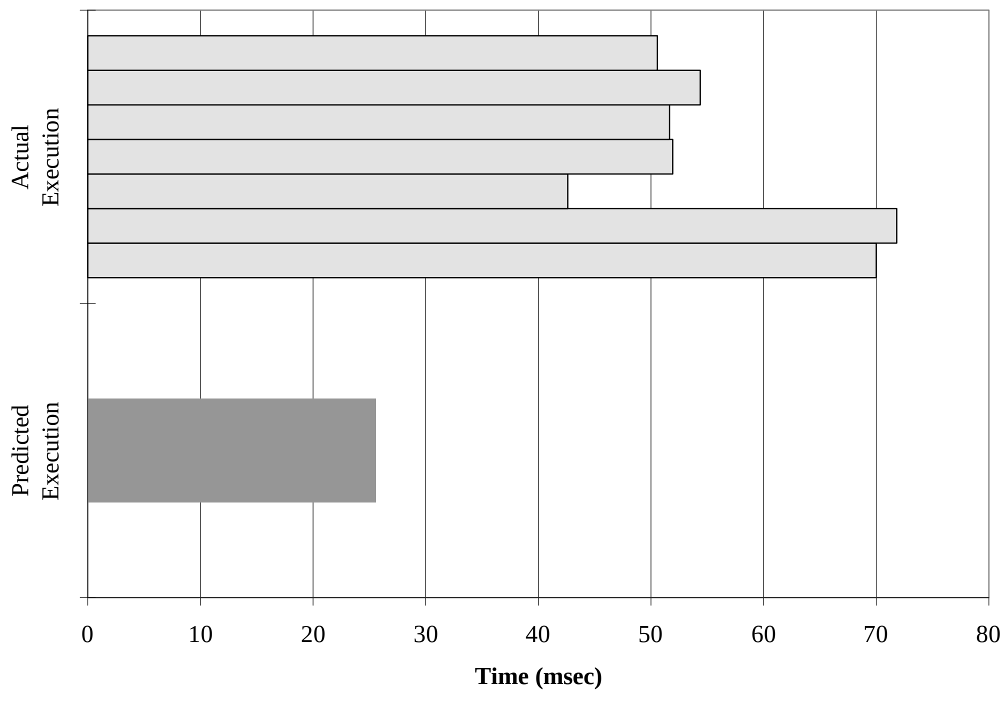


Figure 13: Sanity Check of the Simulation. The schedule produced by the HU heuristic of loop10 using unrolling scheme 1 was implemented using PVM 3.3 and run on a network of workstations connected by a 10Base500 shared Ethernet. The variations in the actual execution times reflect the different loads from other users on the system. The simulation did not model background traffic from other applications or users.

The actual execution times reflect the traffic associated with a multi-user environment. Although the simulation models channel contention (i.e., only one processor may transmit data at one time), it does not model background traffic from other processes. The goal of this experiment was to verify that the predicted execution time of the simulation was a reasonable estimation of the true running time.

5. Conclusions

The first conclusion that can be drawn from this work is that it does indeed matter which scheduling heuristic is used for a particular application and parallel system. The choice of which heuristic is used for any particular situation can have a significant impact on the run-times of the resulting schedules. In fact, using the wrong scheduling heuristic can have a devastating impact on performance.

Second, it appears that there is no clear best scheduler in general, at least not among these eight. With the exception of RAN1, each scheduling heuristic produced some optimal schedules. Furthermore, the slowdown factor from optimal for the schedulers are, for the most part, consistent across the heuristics. This means that the loss in performance of using one heuristic for all input would be approximately the same for most of the heuristics. In other words, there would not be any great performance benefit of using one of the seven parallel scheduling heuristic versus another for all input.

The correlations proposed at the outset of this work did indeed exist. In fact, the strengths of these correlations greatly exceeded expectations. Not only were the more successful heuristics predictable, but so too were the less successful ones such as the random schedulers. The estimates of the schedule lengths for these heuristics were good enough that a very simple regression model sufficed to make an automatic choice of the best candidate to use. The fact that these models *were* so simple *and* were still successful in obtaining better schedules makes a very strong statement about how predictable the heuristics were. After viewing the results of the previous chapter, it appears to be likely that other scheduling heuristics would be at least somewhat predictable. One would expect that as more improved scheduling heuristics are added to the library of heuristics, the quality of the schedules chosen by the metaheuristic will improve.

Although there were more sophisticated models considered in this research, only the simplest one was presented, in order to limit the scope of this paper and because of the fact that the simplest model still worked. However, one would expect that better models would produce better results in the long run. There is significant future research needed to determine which models would be best to use, what the exact parameters of those models should be, and under which circumstances to use them.

Perhaps the most interesting conclusion is not that the metaheuristic was successful in choosing better schedules overall, but why it was better. The metaheuristic did not choose the best heuristic most of the time. However, it was successful in avoiding the really bad schedules, which is how it was able to gain ground against the other schedulers.

We have demonstrated through the experiments presented in this paper that statistical methods can be used to choose among scheduling heuristics on a per-program basis with encouraging results. This is especially true when one considers that the analysis, in the cases we studied, essentially comes for free. The cost of deriving the predictions is very small because it uses variables such as the number of edges and node weights; metrics that are already available when the task graphs are created. Although all of the heuristics except RAN1 produced an occasional best schedule, they also occasionally produced very poor schedules as well. By avoiding the heuristics that would produce poor schedules, the metaheuristic was able to improve the overall quality of the schedules.

We have listed below several of the open problems that need to be addressed in the future:

(1) How does the program partitioning affect the prediction models?

(2) Can the metaheuristic approach be used to automatically choose a good partitioning heuristics on a per-program basis?

(3) How are the prediction models affected by limited resources (e.g. number of processors)?

(4) Are there better prediction models or new metrics that can be used to improve the approach further?

(5) Can this approach be used to solve the combined problems of partitioning, mapping, and scheduling?

6. References

[1] T. L. Adam, K. M. Chandy, and J. R. Dickson. “A comparison of list schedules for parallel processing systems,” Communications of the ACM, 17(12):685-690, December 1974.

[2] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. “A static performance estimator to guide data partitioning decisions.” In the Proceedings of the Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming, Williamsburg, VA, Apr. 21-24, 1991.

[3] K. M. Baumgartner and B. W. Wah. “Computer scheduling algorithms: past, present, and future.” Information Sciences, 57:319-345, 1991.

[4] M. A. Blumrich, C. J. Brownhill, K. Li, and A. Nicolau.”An Empirical Comparison of Loop Scheduling Algorithms on a Shared Memory Multiprocessor.” Technical Report TR-360-92, Princeton University, January 1992.

[5] Arndt Bode, ed. Proceedings of the Parallel virtual machine, EuroPVM ’96: third European PVM conference, Munich, Germany, October 7-9, 1996.

[6] Tevfik Bultan and Cevdet Aykanat. “A new mapping heuristic based on mean field annealing.” Journal of Parallel and Distributed Computing, 16(4):292-305, December 1992.

[7] D. Cann. “Retire FORTRAN? A debate rekindled.” Communication of the ACM, 35(8):81-89,

August 1992.

[8] Vipin Chaudhary and J. K. Aggarwal. “A generalized scheme for mapping parallel algorithms.” IEEE Transactions on Parallel and Distributed Systems, 4(3):328-346, March 1993.

[9] Song Chen, Mary M. Eshaghian, and Ying-Chieh Wu. “Mapping arbitrary non-uniform task graphs onto arbitrary non-uniform system graphs.” In 1995 International Conference on Parallel Processing, II:191-195, 1995.

[10] Philippe Chretienne, Edward G. Coffman, Jr., Jan Karel Lenstra, and Zhen Liu, ed. Scheduling Theory and its Applications. John Wiley & Sons, New York, 1995.

[11] E.G. Coffman, Jr., ed. Computer and Job-Shop Scheduling Theory. John Wiley & Sons, New York, 1976.

[12] E. H. D’Hollander, Y. Devis, “Directed taskgraph scheduling using simulated annealing,” In Proceedings of the 1991 International Conference on Parallel Processing, VII-Software:180-185, 1991.

[13] M. D. Dikaiakos, A. Rogers, and K. Steiglitz. “A Comparison Study of Heuristics for Mapping Parallel Algorithms to Message-Passing Multiprocessors.” Technical Report TR-446-94, Princeton University, February 1994.

[14] Hesham El-Rewini and T. G. Lewis. “Scheduling parallel program tasks onto arbitrary target machines.” Journal of Parallel and Distributed Computing, 9:138-153, 1990.

[15] H. El-Rewini, T. G. Lewis, and H. H. Ali. Task Scheduling Parallel and Distributed Systems. Prentice Hall, Englewood Cliffs, NJ, 1994.

[16] J. T. Feo, D. C. Cann, and R. R. Oldehoeft. “A report on the SISAL language project.” Journal of Parallel and Distributed Computing, 10:349-366, 1990.

[17] Clayton S. Ferner “Automatic Choice of Scheduling Heuristics for Parallel/Distributed Computing.” Ph.D. Thesis, Department of Mathematics and Computer Science, University of Denver, 1997.

[18] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. PVM: Parallel Virtual Machine: A Users’ Guide and Tutorial for Networked Parallel Computing. MIT Press, Cambridge, MA, 1996.

[19] A. Gerasoulis, S. Venugopal, and T. Yang. “Clustering task grpahs for message passing architectures.” In Proceedings of the 4th ACM International Conference on Supercomputing, 447-456, 1990.

[20] Apostolos Gerasoulis and Tao Yang. “A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors.” Journal of Parallel and Distributed Computing, 16(4):276-291, December 1992.

[21] Apostolos Gerasoulis and Tao Yang. “On the granularity and clustering of directed acyclic task graphs.” IEEE Transactions on Parallel and Distributed Systems, 4(6):686-701, June 1993.

[22] Edwin S.H. Hou, Nirwan Ansari, and Hong Ren. “A Genetic Algorithm for Multiprocessor Scheduling.” IEEE Transactions on Parallel and Distributed Systems, 5(2):113-120, February 1994.

[23] T. Hu. “Parallel sequencing and assembly line problems.” Operations Research, 9(6):841-848, 1961.

[24] Jing-Jang Hwang, Yuan-Chieh Chow, Frank D. Angers, and Chung-Yee Lee. “Scheduling precedence graphs in systems with interprocessor communicaton times.” SIAM Journal of Computing, 18(2):244-257, 1989.

[25] A. Ieumwananonthachai, A. N. Aizawa, S. R.. Schwarz, B. W. Wah, and J. C. Yan. “Intelligent mapping of communication processes in distributed computing systems.” In Proceedings of Supercomputing ‘91, 1991.

[26] Authur Ieumwananonthachai, Akiko N. Aizawa, Steven R. Schwartz, Benjamin W. Wah, and Jerry C. Yan. “Intelligent process mapping through systematic improvement of heuristics.” Journal of Parallel and Distributed Computing, 15(2):118-142, June 1992.

[27] A. A. Khan, C. L. McCreary, and Y. Gong. “A Numerical Comparative Analysis of Partitioning Heuristics for Scheduling Task Graphs on Multiprocessors.” Technical Report CSE93-12. Dept. of Computer Science and Engineering, Auburn University, 1993.

[28] A. A. Khan, C. L. McCreary, and M. S. Jones, “A comparison of multiprocessor scheduling heuristics.” In Proceedings of the 23rd International Conference on Parallel Processing, Aug. 1994.

[29] S. J. Kim and J. C. Browne. “A general approach to mapping of parallel computations upon multiprocessor architectures.” In Proceedings of the 1988 International Conference on Parallel Processing, 3:1-8, 1988.

[30] B. Kruatrachue. “Static Task Scheduling and Grain Packing in Parallel Processing Systems.” Ph.D. Thesis, Department of Computer Science, Oregon State University, 1987.

[31] J. K. Lenstra and A. H. G. Rinnooy Kan. “Complexity of scheduling under precedence constraints.” Operations Research, 26(1):22-35, January-February 1978.

[32] Ted Lewis and Hesham El-Rewini. “Parallax: A tool for parallel program scheduling.” IEEE Parallel & Distributed Technology, 1(2):62-72, May 1993.

[33] Brian A. Malloy, Errol L. Lloyd, and Mary Lou Soffa. “Scheduling DAG’s for asynchronous multiprocessor execution.” IEEE Transactions on Parallel and Distributed Systems, 5(5):498-508, May 1994.

[34] C. McCreary and H. Gill. “Automatic determination of grain size for efficient parallel processing.” IEEE Transactions on Software Engineering, SE-2(4):308-320, September 1989.

[35] C. L. McCreary, A. A. Khan, J. J. Thompson, and M. E. McArdle. “A comparison of heuristics for scheduling DAGS on multiprocessors.” In Proceedings of the 8th International Parallel Processing Symposium, 446-451, April 1994.

[36] Michael A. Palis, Jing-Chiou Liou, and David S.L. Wei. “Task clustering and scheduling for distributed memory parallel architectures.” IEEE Transactions on Parallel and Distributed Systems, 7(1):46-55, January 1996.

[37] Douglas M. Pase, “A Comparative Analysis of Static Parallel Schedulers Where Communication Costs Are Significant.” Ph.D. Thesis, Department of Computer Science and Engineering, Oregon Graduate Center (now Oregon Graduate Institute), July 1989.

[38] Shankar Ramaswamy and Prithviraj Banerjee. “Processor allocation and scheduling of macro

dataflow graphs on distributed memory multicomputers by the PARADIGM compiler." In Proceedings of the 22nd International Conference on Parallel Processing, VII:134-138, St. Charles, IL, 1993.

[39] Vivek Sarkar. Partitioning and Scheduling Parallel Programs for Multiprocessors. MIT Press, Cambridge, MA, 1989.

[40] V. Sarkar. "Automatic partitioning of a program dependence graph into parallel tasks." IBM Journal of Research and Development, 35(5/6): 779-804, September 1991.

[41] S. Selvakumar and C. Siva Ram Murthy. "Scheduling precedence constrined task graphs with non-negligible intertask communication onto multiprocessors." IEEE Transactions on Parallel and Distributed Systems, 5(3):328-336, March 1994.

[42] Behrooz Shirazi and A. R. Hurson. "Special issue on scheduling and load balancing." Journal of Parallel and Distributed Computing, 16(4):271-275, December 1992.

[43] B. Shirazi and M. Wang, "Analysis and evaluation of heuristic methods for static task scheduling." Journal of Parallel and Distributed Computing, 10:222-232, 1992.

[44] Gilbert C. Sih and Edward A. Lee. "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures." IEEE Transactions on Parallel and Distributed Systems, 4(2):175-187, February 1993.

[45] Jaspal Subhlok. "Automatic mapping of Task and Data Parallel Programs for Efficient Execution on Multicomputers." Technical Report CMU-CS-93-212. School of Computer Science, Carnegie Mellon University, 1993.

[46] Jaspal Subhlok, James M. Stichnoth, David R. O'Hallaron, and Thomas Gross. "Exploiting task and data parallelism on a mutlicomputer." In Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming, San Diego, CA, May 1993.

[47] Jaspal Subhlok, David R. O'Hallaron, Thomas Gross, Peter A. Dinda, and Jon Webb. "Communication and memory requirements as the basis for mapping task and data parallel programs." In Proceedings Supercomputing '94, 330-339, 1994.

[48] A. Sussman. "Model-driven mapping onto distributed memory parallel computers." In Proceedings Supercomputing '92, 818-829, November 1992.

[49] J. D. Ullman. "NP-complete scheduling problems." Journal of Computer and System Sciences, 10:384-393, 1975.

[50] Min-You Wu and Daniel D. Gajski. "A programming aid for hypercube architectures." The Journal of Supercomputing, 2:349-372, 1988.

[51] Shen Shen Wu and David Sweeting. "Heuristic algorithms for task assignment and scheduling in a processor network." Parallel Computing, 20:1-14, 1994.

[52] J. C. Yan. "New 'post-game analysis' heuristics for mapping parallel computations to hypercubes." In Proceedings of the 1991 International Conference on Parallel Processing, VII-Software: 236-242, 1991.

[53] Tao Yang and Apostolos Gerasoulis. “PYRROS: static task scheduling and code generation for message passing multiprocessors.” In Proceedings of the 6th ACM International Conference on supercomputing, Washington, D. C., 428-437, July 1992.

[54] Tao Yang and Apostolos Gerasoulis. “DSC: Scheduling parallel tasks on an unbounded number of processors.” IEEE Transactions on Parallel and Distributed Systems, v5(9):951-967, 1993.

[55] Tao Yang and Apostolos Gerasoulis. “List scheduling with and without communication delays.” Parallel Computing, 19(12):1321-1344, December 1993.

[56] Wang Ho Yu. “LU Decomposition on a Multiprocessing System with Communications Delay.” Ph.D. Thesis, Department of Engineering, University of California, Berkeley, 1984.