A Framework for Adaptive Algorithm Selection in STAPL

Nathan Thomas Gabriel Tanase Olga Tkachyshyn Jack Perdue
Nancy M. Amato Lawrence Rauchwerger
Parasol Lab, Dept. of Computer Science, Texas A&M University
{nthomas,gabrielt,olgat,jkp2866,amato,rwerger}@cs.tamu.edu

ABSTRACT

Writing portable programs that perform well on multiple platforms or for varying input sizes and types can be very difficult because performance is often sensitive to the system architecture, the run-time environment, and input data characteristics. This is even more challenging on parallel and distributed systems due to the wide variety of system architectures. One way to address this problem is to adaptively select the best parallel algorithm for the current input data and system from a set of functionally equivalent algorithmic options. Toward this goal, we have developed a general framework for adaptive algorithm selection for use in the Standard Template Adaptive Parallel Library (STAPL). Our framework uses machine learning techniques to analyze data collected by STAPL installation benchmarks and to determine tests that will select among algorithmic options at run-time. We apply a prototype implementation of our framework to two important parallel operations, sorting and matrix multiplication, on multiple platforms and show that the framework determines run-time tests that correctly select the best performing algorithm from among several competing algorithmic options in 86-100% of the cases studied, depending on the operation and the system.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Parallel programming; C.1.4 [**Processor Architectures**]: Parallel Architectures

General Terms

Algorithms, Performance, Experimentation, Languages

Keywords

Adaptive Algorithms, Parallel Algorithms, Machine Learning, Sorting, Matrix Multiplication

*This research supported in part by NSF Grants EIA-0103742, ACR-0081510, ACR-0113971, CCR-0113974, ACI-0326350, and by the DOE.

PPoPP'05, June 15–17, 2005, Chicago, Illinois, USA.

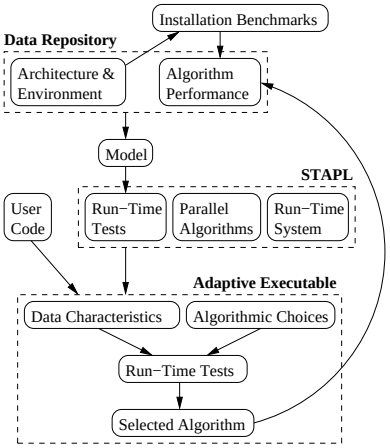


Figure 1: Adaptive framework.

1. INTRODUCTION

For many important operations there exist multiple, functionally equivalent, algorithms that could be used to perform them. Common examples include sorting (insertion sort, quicksort, mergesort, etc.), computing minimum spanning trees (Kruskal's and Prim's algorithms), matrix multiplication (the naive $O(n^3)$ method, or Strassen's, Winograd's, and Cannon's algorithms, etc.), and computing convex hulls (incremental, giftwrapping, divide-and-conquer, quickhull, etc.).

When there exist many functionally equivalent algorithmic options, the decision of which to use can depend on many factors such as type and size of input, ease of implementation and verification, etc. In a parallel and distributed setting, the situation becomes even more complex due to the wide variety of system architectures — and performance can even vary significantly for different sizes (e.g., number of processing elements) of a given architecture. These factors make it difficult even for an experienced programmer to determine which algorithm should be used in a given situation. Moreover, and perhaps more importantly, they make it very difficult to write portable programs that perform well on multiple platforms or for varying input data sizes and types. Ideally, the programmer should simply specify the desired operation (e.g., sort) and the decision of which algorithm to use should be made later, possibly even at run-time, once the environment and input characteristics are known.

In this paper, we describe a general framework for adaptive algorithm selection that we have developed for use in the Standard Template Adaptive Parallel Library (STAPL) [2, 1, 28, 31]. An overview of the process is sketched in Figure 1. Briefly, when STAPL is in-

stalled on the system, statically available information about the architecture and environment is collected and performance characteristics for the algorithmic options available in the library are computed by installation benchmarks. This data is stored in a repository and machine learning techniques are used to analyze it and to determine tests that will be used at run-time for selecting an algorithmic option. At run-time, any necessary characteristics are measured and then a decision about which algorithmic option to use is made.

To study the utility of the approach, we apply the prototype implementation of our framework to two parallel operations, sorting and matrix multiplication. In our validation tests on multiple platforms, our framework provided run-time tests that correctly selected the best performing algorithm from among several competing algorithmic options in 93-100% of the cases studied for sorting and in 86-92% of the cases studied for matrix multiplication, depending on the system. Our framework can also be used to tune algorithmic and run-time parameters, such as the granularity of interprocessor communication, for optimal program performance. These parameters can be modeled empirically using measurements collected during previous runs of the program.

As described in detail in Section 5, our work differs from previous work in several significant aspects. First, unlike some existing specialized sequential sorting libraries (e.g., [21]), our objective is to select among parallel algorithms whose behavior is more difficult to model than their sequential counterparts due to the additional complexity of modeling the system and data distribution. Second, while our framework can also be used for parameter tuning of a particular algorithm (e.g., ATLAS [35]), our focus is on the higher-level automatic selection among multiple functionally equivalent algorithmic options. Finally, to the best of our knowledge, our framework provides the first general methodology for automatically developing a model for selecting among multiple algorithmic options. All other work of which we are aware relies upon some level of manual modeling (e.g., [6, 21, 35] or is domain specific (e.g., [26]).

The framework presented here is a generalization of previous work done by our group for the adaptive selection among multiple algorithmic options for parallel sorting [32, 1] and for performing parallel reduction operations [36, 37]. In this paper, we generalize and refine the strategies applied to these specific operations and develop a general framework for the adaptive and dynamic (run-time) selection of one algorithm from a library containing multiple algorithmic options. We exercise the framework on a new operation (matrix multiplication) and on new implementations of the sorting algorithms, for which we also present a new run-time computed metric to characterize the input data with respect to its initial disorder.

We begin in Section 2, with an overview of the framework and a description of its major components. In Section 3 we briefly describe STAPL and discuss how the framework is integrated into it. In Section 4, we present two case studies with experimental results for parallel sorting and parallel matrix multiplication, respectively. We finish with a discussion of related work (Section 5), and conclusions and future work (Section 6).

2. THE ADAPTIVE ALGORITHM SELECTION FRAMEWORK

Figure 1 shows the major components and flow of information in the adaptive framework. During STAPL installation, the framework collects statically available information from standard header files and vendor system calls about the architecture and environment, such as available memory, cache size, and number of processors. This information is stored in a host information table in the *data repository* (currently implemented as a MySQL database). Also stored in the repository is *algorithm performance* data gathered from the installation benchmarks for the algorithmic options available in STAPL. Next, machine learning techniques are used to analyze the data in the repository and to determine tests that will be used at run-time to select the appropriate algorithm from algorithmic options in STAPL. At run-time, any necessary characteristics are measured and then a decision about which algorithmic option to use is made.

In the remainder of this section, we describe the major components of the framework in more detail.

2.1 Definitions

The algorithm selection problem can be expressed as an attempt to optimize a *fitness function* over a multivariate space of program *attributes* and *parameters*. We define these terms as follows:

- **Fitness Function** - dependent variable which is most often defined as (including for the algorithms discussed in this paper) the minimization of execution time. However, there are other metrics such as power usage and memory consumption that might be also be of interest.
- **Attributes** - defining characteristics of the problem which have *fixed* values for the purpose of adaptation. These include both input sensitive metrics such as number of elements and data type as well as system resource values such as the allotted degree of parallelism.
- **Parameters** - values that can be *tuned* at the invocation of a problem instance in an effort to maximize the fitness function. Examples include arguments to the algorithm that affect its behavior (e.g., the number of digits in radix sort). Also included here are arguments to system/run-time calls that though not directly related to the algorithm, have a clear effect on its execution (i.e., granularity of communication in parallel systems).

2.2 Modeling

Attempts to quantify the effects of attributes and parameters on the value of the fitness function can be described by two broad categories: *general empirical models* and *analytical models*. Empirically driven techniques use (fitness, attributes, parameter) tuples derived from previous algorithm runs to create a model to predict optimal parameter values for future problem instances. They are desirable because of generality and minimization of user effort, but can have difficulty in modeling complex problems. Analytical models are manually generated and implicitly contain algorithm and architecture information in their formulation. Running instances of the problem may be used to determine coefficients. Though usually requiring more effort from the algorithm developer, analytical models can take advantage of programmer provided knowlege which can sometimes result in increased accuracy. For this reason, they can also be used for other purposes, such as validating actual versus predicted performance.

While we are also developing analytical models for STAPL components, we have yet to fully integrate these techniques into STAPL and will focus on empirical techniques for the experiments in this paper. That said, the interfaces employed are general, allowing for transparent inclusion later. In this work, we use several techniques common in machine learning that have proved quite capable in modeling algorithm performance. These approaches are described in the following section.

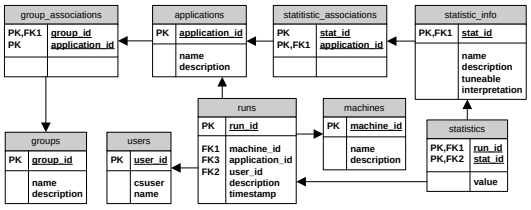


Figure 2: Database schema.

2.2.1 Learner Descriptions

We have implemented three empirical learning approaches within our framework. The first is a decision tree learner, based on Quinlan's ID3 algorithm [27]. After creation of the tree model, a tunable parameter's value can be found by traversing the tree from the root to a leaf node. Non-leaf nodes of the tree contain a function taking the instance's value of one problem's attributes which determines which child the traversal visits next. Leaf nodes contain one of the parameter's possible values. The decision tree is currently the most mature of our learners, implementing both Common Error and Chi Square tree pruning [22] to reduce tree size and prevent the common problem of over-fitting the training data set. We further favored use of this approach in initial testing, since the outputted model (if-then-else nest in C++ syntax) is easily understandable by the human eye.

The second learner is a standard back propagation neural network [30, 22]. It has two internal layers (adjustable) in addition to the input layer with one node per attribute and the output layer with one node for each possible parameter value. Each internal neuron is a linear system of its input (nodes from the previous layer). After training, the prediction is gained by application of the attributes to the first layer, allowing values to propagate to the output layer and selecting the output node with the maximum value.

The third learner is a Bayes naive classifier [13, 22], which, as its name suggests, is relatively simple, basing its predictions on conditional probabilities. For the case studies below, it proved incapable of adequately learning the problem, but remains useful as proof of the generality of the framework.

2.2.2 Choosing a Learner

The above learners all share a common input format and interface for prediction querying once they have been trained. Hence, so long as they provide the same functionality, they can be interchanged with each other automatically, if we choose to allow that. This interchange can be seamlessly accomplished by having the query API placed in a library file which is dynamically linked at execution time with the application.

Each learner is tested by employing a standard 10-way learning validation test in which the input cases are divided into 10 equal segments, with ten models created using 9 segments and analyzed by a different segment each time to measure its accuracy. A general understanding of the learner's ability to model the problem is given by examining the average accuracy of the ten models and the size of the 95% confidence interval for this average. We select the most accurate model from the most accurate learner as the one used for algorithm selection.

2.3 Performance Database

One of the key components of the adaptive framework is a database to store the knowledge gained from the execution of various problem instances. Though space doesn't permit a detailed discussion, Figure 2 shows the database schema. Listed below are the design requirements for a database to be used in a general framework such as this.

- **Generality** – In short, all problems, with varying attributes and parameters of different types must be supported without any change in the database schema. Although this may require complex queries, they can be automatically generated with wrapper scripts.
- **Attribute & parameter additions** – Attributes and parameters for a problem cannot be static. Additions must be allowed as deemed necessary by the user. Furthermore, this must be done in manner that doesn't require dropping previously collected data.
- **Space efficient** – We expect the database to grow quite large, so redundancy and unused records fields must be minimized.

All common interactions of the framework with the database (e.g., problem creation and instance insertion) are handled through a simple API that automatically generates any SQL statements and properly formats the data. Full type checking and other data validations are employed to minimize errors and ensure only "clean" data is placed in the database.

3. INTEGRATION OF ADAPTIVE FRAMEWORK IN STAPL

3.1 STAPL Overview

STAPL (the Standard Template Adaptive Parallel Library) is a framework for parallel C++ code [2, 1, 28, 31]. Its core is a library of ISO Standard C++ components with interfaces similar to the (sequential) ISO C++ standard library [23]. STAPL offers the parallel system programmer a shared object view of the data space. The objects are distributed across the memory hierarchy which can be shared and/or distributed address spaces. Internal STAPL mechanisms assure an automatic translation from one space to another, presenting to the less experienced user a unified data space. For more experienced users the local/remote distinction of accesses can be exposed and performance enhanced. STAPL supports the SPMD model of parallelism with essentially the same consistency model as OpenMP. To exploit large systems such as IBM's BlueGene/L, STAPL allows for (recursive) nested parallelism (as in NESL [5]).

The STAPL infrastructure consists of platform independent and platform dependent components. These are revealed to the programmer at an appropriate level of detail through a hierarchy of abstract interfaces. The platform independent components include the core parallel library, a view of a generic parallel/distributed machine, and an abstract interface to the communication library and run-time system.

The core STAPL library consists of `pAlgorithms` (parallel algorithms) and `pContainers` (distributed data structures). The `pContainer` is the parallel equivalent of the STL container. Its data is distributed but the programmer is offered a shared object view. The `pContainer` distribution can be user specified or computed automatically. Currently, STAPL has two `pContainers` that do not have STL equivalents: parallel matrix (`pMatrix`) and parallel graph (`pGraph`). A `pAlgorithm` is the parallel equivalent of an STL algorithm. STAPL's ARMI (Adaptive Remote Method Invocation) communication library [31] uses the remote method invocation (RMI) communication abstraction that assures mutual exclusion at the destination but hides the lower level implementation (e.g., MPI, OpenMP).

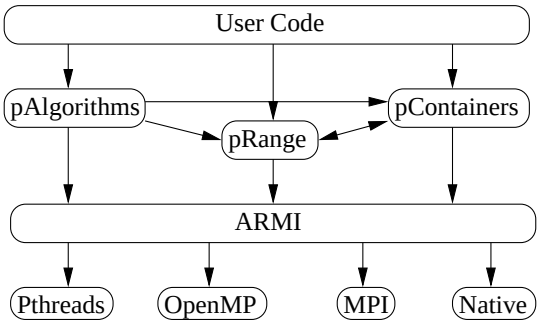


Figure 3: STAPL **components.**

One of the key design objectives of STAPL is to provide *portable performance* — STAPL programs should run efficiently on multiple platforms without user code modification. Since the performance of parallel algorithms is sensitive to system architecture, to application data, and to run-time conditions, this objective can only be obtained if STAPL programs have the ability to automatically adapt to the current system conditions and the input data. Adaptive behavior by `pAlgorithms` is supported by providing a number of algorithmic options that have the same interface and functionality, but which perform differently depending on the architecture and data involved. The general framework for the run-time selection of the most appropriate algorithm option for the current conditions is the subject of this paper.

3.2 Integration

The instantiation of the adaptive framework begins during the installation of the STAPL library on a new platform. A configuration file contains problem creation calls to the framework API for each algorithm that will contain adaptive mechanisms. This series of calls creates appropriate entries in the database to facilitate the storing and validation of empirical test runs. STAPL actually provides a wrapper around this call to algorithms so that STAPL can transparently include common parameters (e.g., number of processors) or parameters of STAPL components it knows might affect performance. The current structure of this call is shown in Figure 1.

Algorithm 1 Framework Problem Creation API

1: learner.create_problem("problem_name", fitness, attribute, parameter)
Where:
Fitness = (execution_time, int, min())
Attribute = {
(num_processors, int),
(num_elements, int),
(data_type, enum{int,double,...}),
(data_size, int),
(algorithm_specified_metrics) }
Parameter = {
(algorithm, enum{$algo_1$,$algo_2$,...}),
(algorithm_specified_parameter),
(armi_aggregation, int) }

After STAPL is installed on the system, the next step is to run the installation benchmarks and insert the performance data into database. The STAPL installation can perform these tests itself. Alternatively, STAPL could pass a "training executable" along with ranges of attributes and parameters and the framework will run the tests itself.

Once empirical data collection has completed, the models are created using the techniques described in Section 2.2. For each parameter an algorithm needs predicted, a function call named with the form *predict_AlgorithmName_ParameterName* is created which takes in as arguments attributes and returns the predicted value of the parameter. These function definitions can be placed in a dynamically linked library file which could be included in every compilation of a STAPL application. Calls to these functions would occur inside the adaptive algorithm, wherever such decisions are required. An example function definition is shown in the experimental results.

4. EXPERIMENTAL RESULTS

We tested our adaptive framework using two important parallel operations: sorting and matrix multiplication. In Section 4.2, we study parallel sorting. We first describe the individual parallel sorting algorithms currently available in STAPL and then discuss how the adaptive algorithm selection framework has been applied to them to obtain STAPL's adaptive `psort()` `pAlgorithm`. Then we examine performance results on various systems comparing STAPL's adaptive `psort` to the individual algorithms. Matrix multiplication is examined in a similar manner in Section 4.3. As will be seen, for both operations, STAPL's adaptive algorithm selection framework performs quite well, selecting the best performing algorithm from the available algorithmic options in 93-100% of the cases studied for sorting and 86-92% of the cases studied for matrix multiplication, depending on the system.

4.1 Systems studied

We examined the behavior of the algorithms on three systems.

Altix: an 128 processor SGI Altix 3700, with two 1.3Ghz Intel Itanium 4GB RAM and two processors per board, each with 3MB of L3 cache. Its hypercube interconnect implements hardware DSM with CC-NUMA.

Cluster: an 1152 node Linux cluster located at Lawrence Livermore National Laboratory. Each node has two 2.4Ghz Pentium 4 Xeon processors each with a 512KB L2 cache and 4GB of memory. The Quadrics interconnect uses a fat tree topology.

SMPCluster: a 68 node IBM cluster located at Lawrence Livermore National Laboratory. Each node has sixteen 375Mhz POWER 3 Nighthawk-2 processors, each with an 8MB L2 cache and 16GB of memory. The interconnect is an internal IBM SP multistage switch network.

4.2 Parallel Sorting

Sorting is a fundamental operation that is used in a large range of applications. In this section, we consider three popular parallel algorithms for sorting: *sample*, *radix*, and *column* sort. Each behaves differently when the architecture and input characteristics are varied, making sorting a good candidate for our adaptive framework. We next provide descriptions of each sorting algorithm and discuss how we applied the adaptive algorithm selection framework in this case. Finally, we present results obtained using the resulting adaptive sorting routine on the various machines.

4.2.1 Algorithms

Sample sort [4] is a two pass technique that first samples the data to create discrete buckets (one per processor). Next, the algorithm distributes the elements into these buckets. These buckets are then sorted independently. Some implementations vary the ratio of elements sampled to the number of processors, termed the *over-*

sampling ratio, attempting to ensure even distribution of elements into the buckets. However, we leave this ratio fixed at 128. Our implementation is outlined in Algorithm 2.

Algorithm 2 sample_sort($pRange$)

```
1: locally, select s random keys from local elements
2: globally, sort all selected keys
3: select p − 1 splitters to define p buckets
4: for each local element on each processor do
5:     determine correct bucket for element using splitters
6:     send the element to the appropriate bucket
7: end for
8: each processor sorts the local bucket
```

Radix sort [4] is a version of the commonly known linear time sequential sort for integers, implemented using parallel counting sort. The sort considers elements in chunks of r bits at a time from least to greatest. Processors count the number of times each value $0..2^r - 1$ occurs in their local data and then the prefix sum of the counts is computed globally. Each processor inserts its elements appropriately into the global list. This approach is repeated over r bits at a time until all bits have been considered. Our implementation, shown in Algorithm 3, employs a simple optimization that scans the input set to determine the minimum and maximum values, so that only bits that can differ are processed. Also, for experiments, we fix r at 16 bits.

Algorithm 3 radix_sort($pRange$)

```
1: for the i-th block of r bits do
2:     for each possible value in 0..2^r − 1 do
3:         count local elements with this value
4:     end for
5:     exchange count arrays
6:     compute prefix sum of value counts
7:     compute processor's offset for each value
8:     add prefix sum and offset for final position
9:     for each element in parallel do
10:        place element in predetermined position
11:    end for
12: end for
```

Column sort [20, 8] is a version of one of the first parallel sorts to prove a $O(\log n)$ upper bound on the parallel time. However, it requires four local sorts of data and four communications steps, two all-to-all and two one-to-one. Hence, although theoretically optimal, it is often outperformed by other parallel sorting algorithms. The STAPL implementation uses the `pMatrix` `pContainer` described in Section 4.3 as the base underlying data structure. The implementation is summarized in Algorithm 4.

Algorithm 4 column_sort($pRange$)

```
1: local element sort
2: transpose and reshape the pMatrix
3: local element sort
4: reshape and transpose the pMatrix
5: local element sort
6: circularly shift pMatrix down by (column_height)/2
7: local element sort
8: circularly shift pMatrix up by (column_height)/2
```

4.2.2 Training data for adaptive pSort

We began the creation of the adaptive sorting models by determining quantifiable attributes that might affect the performance of the algorithms on different architectures. For these experiments, we include the following metrics: *number of processors, cache size, number of elements*, and *data type of elements*. All of these parameters are available to the application at execution time.

In addition to these metrics, we also consider the range of possible values in the input, fixing the minimum value at 0 and varying the maximum value. We analyze this property of the data due to the optimization in the radix sort implementation that allows it to ignore bits that are identical in all input elements. Hence, when the maximum is relatively small, radix sort is expected to have a competitive edge.

Finally, it is well known that the initial ordering of the input data will affect which algorithm performs best. Inputs with varying levels of disorder will likely cause the performance of each algorithm to change. To capture this attribute, we include an additional metric, a measure of *input presortedness*.

Algorithm 5 pSort(pRange pr)

```
1: array S = sample of pr
2: sort S and track displacement distances
3: compute average distance
4: normalize average by size(S)
5: sort = learn_pSort(p, n, type, size, norm_avg_dist)
6: call sort(pr)
```

To quantify *input presortedness*, we perform a run-time sampling of the input and sort it, tracking for each element the difference between its initial position and its sorted position. We then compute the average distance, normalized by the size of the sample. Hence, the *normalized average distance* is expected to be near 0 when the entire input set is sorted or nearly sorted, to be near 0.5 when it is nearly reverse sorted, and somewhere in between for random sequences between these two extreme cases. This metric is passed on as a parameter to the learner. Algorithm 5 shows pseudocode for STAPL's adaptive `pSort`.

Parameters	Values
Processors (P)	2 4 8 16 32 64*
Data Type (T)	int double
Input Size (N)	(100K..20M)*P/sizeof(T)
Max Value	N/1000 N/100 N/10 N 3N MAX_INT
Input Order	sorted reverse sorted random
Displacement	0..20% of N**

*only up to 32 processors on the Altix
**only for sorted and reversed sorted

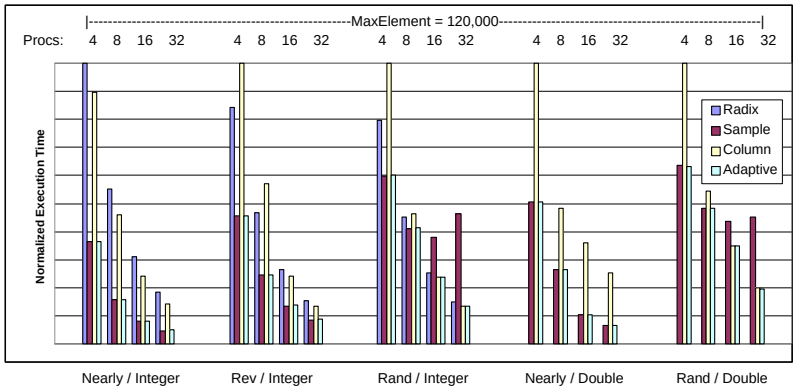
Table 1: Parameters for parallel sorting

```
1: if p ≤ 8 then
2:     sort = "sample"
3: else
4:     sort = "sample"
5: end if
```

Table 1 summarizes the attribute values used to generate training inputs for the adaptive framework. Note that some parameters are a function of others. For instance, the range of input sizes was scaled by the number of processors used and the size of the chosen data type. This was done to enable larger input sizes to be sorted on the larger processor configurations than would have been possible on fewer processors. We also ensured that our range would range

if $p \leq 8$ **then**
  $sort =$ "sample"
else
  if $dist\_norm \leq 0.117188$ **then**
    $sort =$ "sample"
  else
    if $dist\_norm \leq 0.370483$ **then**
      $sort =$ "column"
    else
      $sort =$ "sample"
    end if
  end if
end if

(a)



(b)

Figure 4: (a) p_sort decision tree and (b) adaptive sorting 120M elements on Altix.

from inputs that would fit entirely within local level two cache to inputs that cannot fit in the combined level three cache. Varying presortedness is created by the *Input Order* and *Displacement* parameters. Using randomly generated combinations of these values, 1000 inputs were generated for each of the machines. Each input consists of the attribute set and best algorithm for it. Note that for attributes such as those dealing with presortedness, the associated runtime test (i.e., *Normalized Average Distance*) is passed to the learner in place of the value(s) used to generate data. These instances are used to create the learner model and validate it using the approach discussed in Section 2.2.2.

4.2.3 Results

Using the training data described above, we generated models for algorithm selection on the SGI Altix, SMPCluster, and Linux Cluster. On all platforms, a decision tree prediction model was selected. The estimated accuracy of the models (based on the 10-way test) for the three platforms was 94%, 98%, and 94%, respectively. We only show detailed results for the Altix and Linux Cluster in this section.

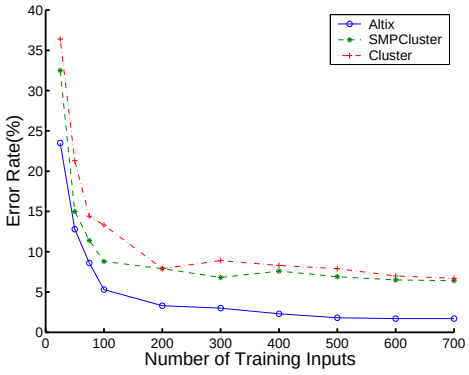


Figure 5: Model error rates for sort.

Figure 5 shows the change in model accuracy on the three platforms as the number of training inputs increases. On all platforms, the accuracy of the model increases rapidly with the first 100 training inputs, and then increases at a more modest pace for the next 100 training inputs. The two clusters obtain minimal increase in accuracy after about 200 training inputs, while the Altix continues to obtain modest, but noticeable increases until about 500 training inputs. The rapid increase over the first 100-200 training inputs covers the stage where the basic structure of the tree is set. Thereafter, increases in accuracy are obtained by refining the thresholds in the decision nodes.

Figure 4(a) shows the decision tree created by the learner on the Altix. It is interesting to note that on this platform radix sort is never selected by the decision tree. If it were, as it is on the other platforms, then there would be some split based on input type because radix sort can only be used for integer inputs. Instead, note the importance of presortedness in the model, as it first splits on nearly sorted data with an implied threshold of element displacement and later splits again using *normalized average distance*, on a value that represents a split between reversed and random data. The trees for the two cluster machines are larger than on the Altix, with 29 and 33 nodes versus 7, likely due to the increasing complexity of the machines' memory topologies.

Validation. To validate the models on these platforms, we tested their accuracy on a set of problem instances not provided in the training phase. Specifically, we considered input sizes of 80M and 120M elements for both integer and doubles. Three presortedness measures were used: sorted and reversed (both with 5% element displacement) in addition to random data sets. These instances had either maximum element value of N/1000 or N and were run on the same processor counts ranging from 2 to 64.

The accuracy of the adaptive algorithm on the validation set was 100% on the SGI Altix and 93.3% on the Linux Cluster. For the cluster, the average relative performance penalty of mispredictions was 25.4% of the optimal algorithm.

Figure 4(b) and Figure 6 show representative inputs from the validation testing on the SGI Altix and Linux Cluster, respectively. Each set of bars shows the performance of our adaptive algorithm and of the three base algorithms. Execution times for each category have been normalized by the maximum time from that category. Algorithm mispredictions (selecting the wrong algorithm), are denoted by "E." There are several observations that can be made from Figure 6. First, by comparing the adaptive method and the fastest algorithm when the model was correct (i.e., no "E", when the adaptive method did select the fastest algorithm), it can be seen that the overhead incurred by the adaptive method is relatively small. It can also be seen that algorithm performance can be quite sensitive to input type (sortedness) and magnitude (maximum absolute value of an element). For example, sample sort performs well on nearly sorted data and column sort does well on random and nearly reverse inputs. While performance is input sensitive on both plat-

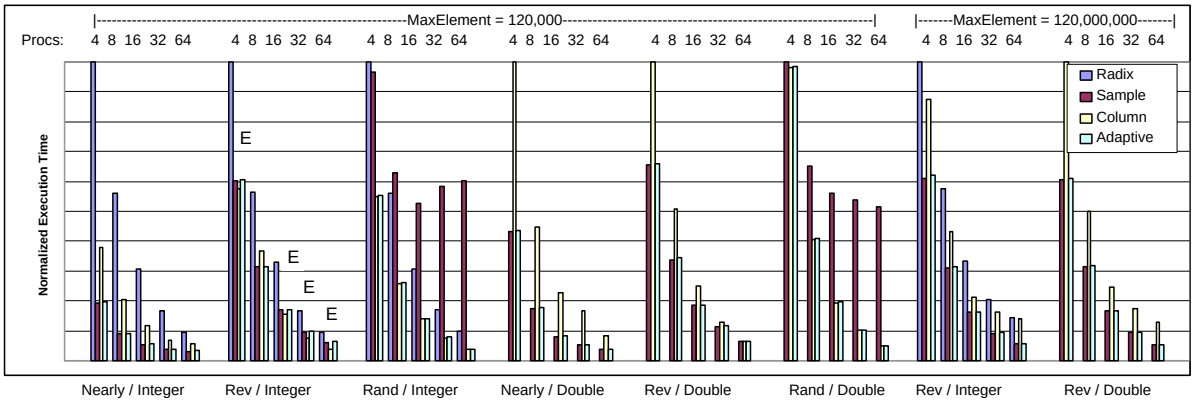


Figure 6: Adaptive sorting 120M elements on the Linux Cluster.

forms, there are some differences between platforms. For example, sample sort performance changes for random data sets on the two platforms, and although column sort usually is best for reverse sorted inputs, sample sort performs better on the Linux Cluster for small magnitude inputs. Indeed, this is the cause for four of the eight mispredictions — sample sort was fastest, but our scheme selected column sort. This problem is caused by an incompletely refined threshold on the *input range* in the tree. Better sampling around such thresholds during training may help avoid such problems. This effect is also noticeable with doubles, but is less pronounced. Finally, we see that in some cases the relative performance changes with the number of processors – initially sample sort wins, but column sort overtakes it as the number of processors increases.

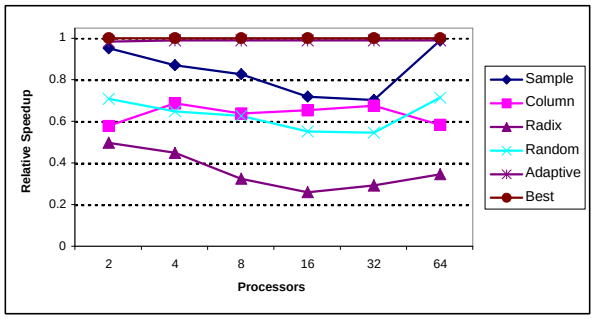


Figure 7: Relative speedup of sorting algorithms on Linux Cluster.

Figures 7 and 8 show the *average relative speedup* and *average relative error*, respectively, of the various algorithms on the Linux Cluster. So that we can include radix sort, we have only used integer instances. However, both the validation tests and predicted tree accuracy show that the model is equally effective on the double data type. In both graphs, the ”random” algorithm randomly selects either sample, column, or radix. Also, the adaptive algorithm does not include the overhead of the method (measuring statistics and evaluating the decision tree). However, recall we saw that this overhead is generally minimal. Figure 7 shows that our predictive model works almost as well as the “perfect predictive model”, obtaining 98.8% of the best possible performance, whereas the next best algorithm (sample) only provides 83%. Figure 8 shows the performance degradation incurred by the various

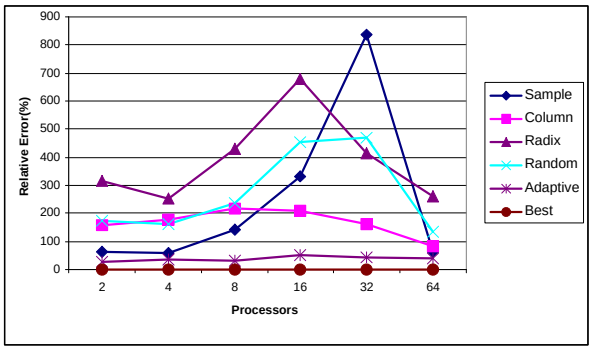


Figure 8: Relative error of sorting algorithms on Linux Cluster.

algorithms as compared to the optimal one for each instance. When incorrect, our model incurs on average a 35% performance penalty, whereas all competing approaches incur penalties of at least 170%. Finally, note that sample sort, the best performer among the base algorithms in terms of relative speedup, incurs a large performance penalty of 210% when incorrectly used.

4.3 Parallel Matrix Multiplication

Matrix multiplication is a fundamental numerical computation in many scientific applications. In this section, we look at three parallel matrix multiplication algorithms: Cannon’s algorithm [19], Broadcast Multiply Roll (BMR) [16], and Scalable Universal Matrix Multiplication Algorithm (SUMMA) [17]. These algorithms have different memory requirements and communication patterns which stress the system in different ways, making them good candidates for adaptive selection and inclusion in STAPL.

All of the algorithms use STAPL’s `pMatrix`, a parallel matrix container based on the Matrix Template Library (MTL) [33] developed at Indiana University. We have used the specialization features of MTL to optimize the version used in STAPL to use the ATLAS implementation of the BLAS library [35] or the Math Kernel Library (MKL) from Intel, for sequential operations (thanks to Trevor Blackwell and Gunter Winkler). A 2D block cyclic data distribution for the `pMatrix` was selected for these operations due to its observed benefit for the performance of matrix algorithms like parallel LU decomposition [9] and parallel matrix multiplication [10].

4.3.1 Algorithms

All algorithms perform the operation $C = C + A \cdot B$. The matrices have a blocked layout, of size $b \times b$, blockwise.

Cannon's algorithm [19] starts by skewing blocks of A leftward and blocks of B upwards followed by a multiplication/shift phase (see Algorithm 7). The algorithm does not efficiently overlap communication with computation, often yielding poor performance.

Algorithm 6 Cannon

1: left circular shift row i of A by amount i
2: upward circular shift column j of B by amount j
3: **for** $k = 0..b-1$ **do**
4: $C(,) = C(,) + A(,) * B(,)$
5: left circular shift each row of A by 1
6: upward circular shift each column of B by 1
7: **end for**

The **Broadcast Multiply Roll** (BMR) [16] algorithm replaces the skewing step in Cannon's algorithm with a row-wise broadcast for blocks of A. The blocked cyclic distribution version we implemented in STAPL is similar to PUMMA [10]. The broadcast is implemented in an asynchronous fashion thus allowing computation and communication to overlap.

Algorithm 7 BMR

1: **for** $k = 0..b-1$ **do**
2: Broadcast block $A(i,(i+k) \mod b)$ row-wise
3: //Assume that data received trough
4: //broadcast is stored in T
5: $C(,) = C(,) + T * B(,)$
6: Upward circular shift each column of B by one
7: **end for**

SUMMA [17] (Algorithm 9) replaces the upward shift in BMR with a column-wise broadcast for the rows of B. The broadcast can be implemented asynchronously, thus providing more opportunities than BMR to overlap communication with computation. The upward shift in BMR implies a synchronization step in the main loop of the algorithm. Overall, the performance of SUMMA is better than the performance of the other two algorithms for large numbers of processors.

Algorithm 8 SUMMA

1: **for** $k = 0..b-1$ **do**
2: **for** $I = 1 \ldots b$ in parallel **do**
3: broadcast block $A(I,k)$ rowwise
4: //$A(I,k)$ will be stored into Acol
5: **end for**
6: **for** $J = 1 \ldots b$ in parallel **do**
7: broadcast block $B(k,J)$ columnwise
8: //$B(k,J)$ will be stored into Brow
9: **end for**
10: $C(,) = C(,) + Acol * Brow$
11: **end for**

4.3.2 Training data for adaptive pMatrixMultiply

We again determine those metrics that describe the problem instance and may affect the performance of the matrix multiplication algorithms. The attributes include *number of elements*, *data type of elements*, *number of processors* and *data distribution*. The block cyclic distribution used is characterized by two attributes: *stencil* and *block size*.

The `pMatrix` container can control how its data is mapped onto the system through its distribution manager. There are two parameters we study that have an affect on performance: *block size* and a *stencil* denoting how the block cyclic distribution is mapped to the computation threads. For example, when a matrix is distributed across 16 computational threads there are five possible stencils: 16x1, 8x2, 4x4, 2x8 or 1x16. For mesh architectures, the best performance will be obtained when the stencil for the block cyclic distribution matches the physical network topology.

Parameters	Values
Processors*	P =rand$(2, 4, 8, 16, 32, 64)$
Data Type	DT = rand$(double, complex(double))$
Input Size**	N = sqrt(rand$(200K..40M)P/(3$DataTypeSize$))$
Stencil	(px, py) =rand(all possible stencils for P)
Block Size***	B =rand$(MinN/max(px,py)..N/max(px,py))$

*$P = 64$ linux cluster, frost
**dimension (i.e., NxN)
***MinN=sqrt(200K * P/(3*DataTypeSize))

Table 2: Parameters for matrix multiplication.

Table 2 summarizes the attribute values used to generate training inputs for the adaptive framework. The input size is randomly generated within an interval that ranges from a size where the local data for each thread for all three matrices will fit in a level two cache to a size that exceeds level three cache on all platforms considered. We also scale the interval in which the input size can vary with the number of processors, this way covering realistic situations where the input problems solved are bigger on higher number of processors. The stencil used was chosen randomly among all possible stencils for a certain number of processors. For example when $P = 16$ we randomly select the stencil from 1x16, 2x8, 4x4, 8x2 and 16x1. The block size was varied randomly in the range where three blocks will fit in level two cache and where they would exceed level three cache. The block size is correlated with N and the stencil used. It has the restriction that it should not be bigger than N and moreover it should not lead to an empty block on some threads thus leaving them unused. This restriction is expressed in Table 2 by the fact that the block size can be at most $N/\max(px, py)$, where px and py describe the stencil used. Using various combinations of these values, 1000 inputs were generated by the framework for each platform. Each input consists of the attribute set and the best algorithm for it, and the learner created the model and validated the results using the approach discussed in Section 2.2.2.

4.3.3 Results

Using the training data described above, we generated models for algorithm selection on the two platforms considered, Altix and SMPCluster. As with sorting, in both cases, the decision tree learner proved the most accurate. The estimated accuracy of the models (based on the 10-way test) for the two platforms was 90.5% on the SGI Altix and 93.8% on the SMPCluster.

Figure 9 shows the change in model accuracy on the two platforms as the number of training inputs increases. As with the sorting models, we see that initially there is a rapid increase in accuracy as the number of training inputs increases, but then the improvements level off. In contrast to sorting, for matrix multiplication we see that accuracy is worse on the Altix than it is on the SMPCluster.

Validation. To validate the models on these platforms, we tested their accuracy on a set of problem instances not provided in the training phase. Specifically, we considered inputs of 4096x4096

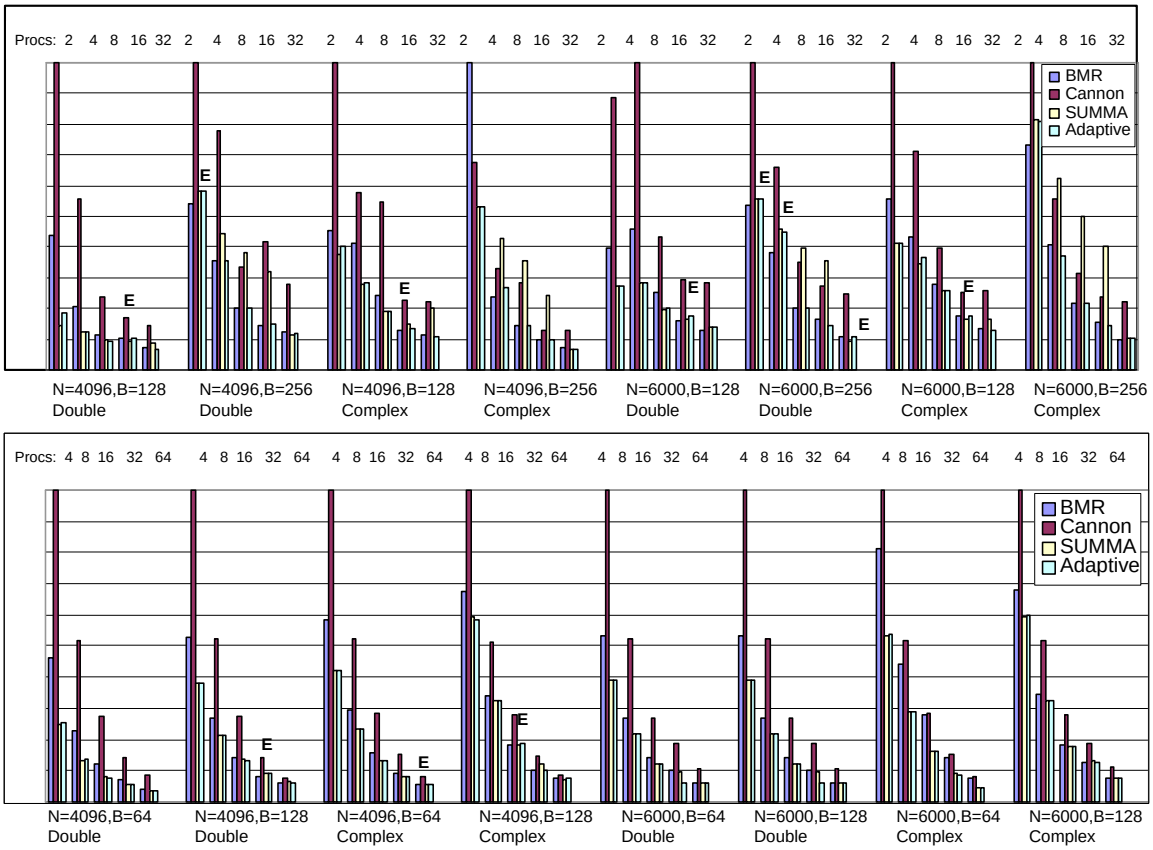


Figure 10: Adaptive matrix multiplication on Altix (top) and SMPCluster (bottom). Algorithm mispredictions are marked by an 'E'.

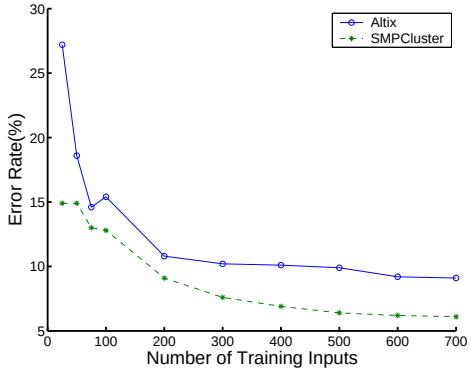


Figure 9: Model error rates for matrix multiplication.

and 6000x6000 matrices of double and complex data types. Two block sizes were considered for each machine: 128 and 256 on Altix, 64 and 128 on SMPCluster. We also chose two types of stencils to generate different communication patterns. One that minimizes communication by minimizing stencil boundaries (standard domain decomposition): for 2, 8 and 32 threads we used 1x2, 4x2, 8x4 stencils, respectively. And another that maximizes communication by maximizing stencil boundaries: Px1 for all but the largest number of threads on the machine for which we chose (P/2)x2 stencils. The number of processors was varied from 2 to 32 on Altix and from 2 to 64 on SMPCluster. This generated a total of 80 test cases on Altix and 96 test cases on SMPCluster.

The accuracy of the adaptive algorithm on the validation set was 86.25% on the SGI Altix (11 mispredictions) and 92% on the SMPCluster (7 mispredictions), which is comparable to the accuracy predicted by the learner. Moreover, when mispredictions occur, the performance penalty was relatively small with an average relative performance error over the mispredicted samples of 14.8% for Altix and 7% for SMPCluster.

Figure 10 shows representative inputs from the validation testing on the SGI Altix (top) and SMPCluster (bottom). Each set of bars shows the performance of our adaptive algorithm and of the three base algorithms. Execution times for each category have been normalized by the maximum time from that category. Algorithm mispredictions (selecting the wrong algorithm) are denoted by "E". We included only the results for the cases where the stencil generates the maximum communication, which also turn out to be interesting because the algorithms interchange often in terms of best performance, especially on Altix.

For the most part, the results are comparable to those for the sorting algorithms. For example, by comparing the adaptive method and the fastest algorithm when the model was correct (i.e., no "E", when the adaptive method did select the fastest algorithm), it can be

seen that the overhead incurred by the adaptive method is relatively small. As was the case with the sorting algorithms, we believe the accuracy will improve when we use more training data and/or better methods for determining the threshold values between algorithms. For instance, the algorithm misprediction for block size 128 on Altix for 16 threads occurs as BMR begins to outperform SUMMA.

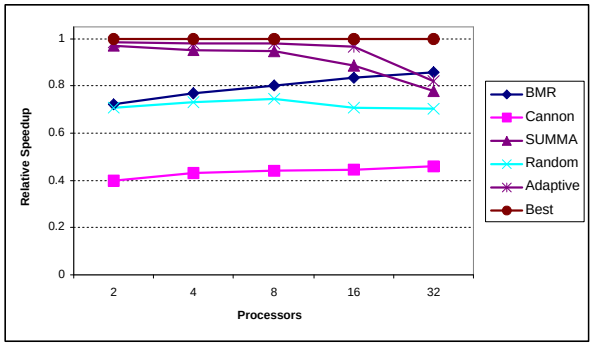


Figure 11: Relative speedup of matrix multiplication algorithms on Altix.

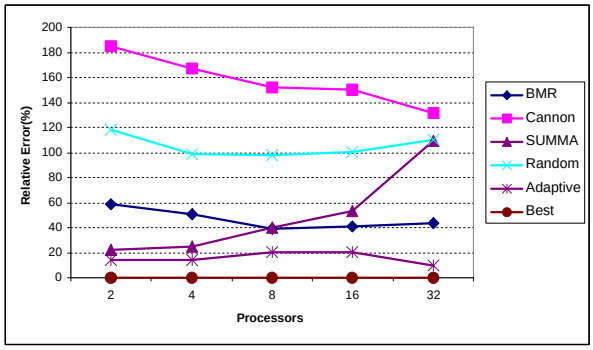


Figure 12: Relative error of matrix multiplication algorithms on Altix.

Figures 11 and 12 show the *average relative speedup* and *average relative error* of the various algorithms for the Altix. The relative speedup plot shows that our model obtains 96.07% of the ideal performance. The next best algorithm, SUMMA, achieves only 92%. Note the behavior of the algorithms at 32 processors. None provide near the ideal case, as there is a clear performance switch between Summa and BMR, which our model does not immediately recognize, likely because it lies on the boundary of the problem space explored during the training phase. The relative error plot shows the performance degradation incurred by the various algorithms as compared to the optimal one for each instance. When incorrect, our model incurs on average of 17.3% performance penalty, whereas the next best algorithm suffers a 47.7% penalty.

5. RELATED WORK

There has been a fair amount of work done in the area of automated algorithm tuning and selection, much of it focusing on numerical libraries. The case for algorithm selection was first made by Rice [29], and the utility of an empirically based approach was recently reaffirmed by Vuduc [34]. In this section, we discuss several projects and how they relate to the STAPL framework presented in this paper. Table 3 gives a brief summary of this comparison.

The ATLAS [35] library aims to provide automated tuning of BLAS [18], a widely used collection of linear algebra routines. ATLAS makes two important contributions to the field. First, it defines a good taxonomy of software adaptation, dividing tuning between parameterized adaptation and source code adaptation. Source code adaptation is further split between multiple implementation (choosing among several implementations) and code generation techniques (i.e., loop unrolling). Another important contribution is a strong proof of the viability of automated library customization — ATLAS equals or exceeds the performance of vendor supplied BLAS implementations on a wide variety of platforms.

Similar to our approach, ATLAS uses empirical data gathered at installation to drive customization. Their modeling approach, however, is clearly specialized for the BLAS library and no attempt is made to discuss a general framework. Furthermore, no characteristics of the input data, other than problem size, are employed in the optimization process. Though this may be unimportant for the studied routines, other previous work [21, 36, 37] emphasizes the importance of input dependent adaptation. For example, [36, 37] develops a collection of parallel algorithms for the reduction operation and conclude that on a given system, various inputs of equal size require a different algorithm for optimal performance.

SPIRAL [26] is targeted at signal processing algorithms and is unique in that it maintains internally high-level formula-based descriptions of algorithms from which a large set of concrete code instantiations can be generated for consideration during the installation/learning phase. Although domain specific, the framework does provide a generic interface in which many learners can be used and employs a feedback mechanism to guide the selection of code instantiations to attempt. It is also insensitive to characteristics of the input data.

Brewer [6] is probably the most extensive previous work aimed at making a framework for parallel algorithm selection. Linear system performance models are provided, along with code annotations, by the user, and then a benchmarking phase determines the models' coefficients. The models are then used to estimate the running time of each algorithm, and the algorithm is selected that corresponds to the model that had the minimum estimated running time. The process was shown to be effective for selecting among sorting algorithms and determining data distributions for an equation solver. The use of individual performance models appears to require more effort by the user than our approach. We have not seen cases where answering the more difficult question of what is the expected running time of each algorithm yields a more accurate selection then simply determining which algorithm will perform best in the current situation.

Li, Garzaran, and Padua [21] present a sequential sorting library that is tuned to individual platforms and also employs basic statistical information about the data (standard deviation, entropy) to provide some customization that is input sensitive. They begin by tuning the individual parameters of the considered sorting algorithms (quicksort, radix, merge), such as the span and depth of the heap created by merge sort and the approach used by quicksort and merge sort implementations to sort the subsets created. These optimized versions are then considered by a two phase algorithm process that first selects between merge and quicksort based on input size. The first phase selection is then compared to radix sort using the Winnow learning algorithm which uses empirical cases and data entropy to create a model to select the best among these algorithms. Although this work is specialized for adaptive sequential sorting, it is similar in philosophy to our work on adaptive parallel

	STAPL	ATLAS	SPIRAL	Brewer	Li	Olszewski
Scope	General	Linear Algebra	DSP	General	Sorting	Sorting
Input Sensitive	Yes	Size Only	Size Only	No	Yes	Yes
Model Variation	Generic API	Fixed	Multiple Provided	Fixed	Fixed	Fixed
Parallel	Yes	No	No	No	No	Yes
Data Types	Arbitrary	Int/Float/Complex	Float/Complex	Int	Int	Arbitrary

Table 3: Comparison with related work.

sorting [32, 1].

Olszewski and Voss [24] present a system for automatic generation of optimized sequential and parallel sorting algorithms. The authors propose a novel approach to generate optimized sequential sort algorithms by using a divide-and-conquer technique. The input problem is recursively partitioned and a decision on what algorithm to use is made at each recursive call. The same approach is extended to generate optimized parallel algorithms. The authors focused on shared memory parallel sorts for small scale SMP systems where the overhead of maintaining a shared work queue is low.

The importance of disorder in the sorting problem has long been studied in the literature [7, 15]. Many sequential algorithms exist that attempt to smoothly transition from a theoretical complexity of $O(n)$ to $O(n * logn)$ as disorder increases [11, 12, 25, 14]. These techniques often modify existing algorithms (i.e., insertion, heap, merge) with complex auxiliary data structures to approach optimality with respect to one or more metrics of disorder. Examples of these disorder measures include the maximum distance of an inverted pair of elements (Dis), the minimum number of elements removed that yields a sorted sequence (Rem), the minimum exchanges requires to sort the sequence (Exc), and the number of distinct sorted runs in the list ($Runs$). Cook and Kim [11] performed a series of empirical tests showing the effect of disorder on the performance of several practical algorithms.

Our approach to quantifying disorder differs in several ways. First, we are using disorder to choose among several competing algorithms based on their actual performance on a given architecture, not as the basis of implementation or theoretical analysis of a single algorithm. In addition, we employ a sampling based measure of disorder that though possibly imprecise, provides an inexpensive runtime test that enables correct algorithm selection for the cases we studied.

There is also work that focuses on key-based record sorting in distributed databases. Arpaci-Dusseau [3] present an algorithm for use on networks of workstations that tunes parameters such as data striping based on information inferred about the disk topology, I/O bandwidth, and available memory.

6. CONCLUSIONS AND FUTURE WORK

We have presented a framework for adaptive algorithm selection and have described its integration and use within the STAPL environment. We have shown that it is capable of dynamically optimizing the execution of two important parallel operations, sorting and matrix multiplication, by choosing among competing algorithmic options and tuning both algorithm and run-time parameters. We have further shown the generality of the approach by providing implementations of multiple learners that compete for use and by describing a general database for performance statistics that can be used by any application.

For future work, there are several improvements we plan to incorporate into the framework. As discussed previously, we believe improvements in accuracy can be obtained by using better training data and/or better methods for determining the threshold values between algorithms. We also plan to investigate better automating the generation of training instances by providing the framework with the capability to intelligently select parameters based on architectural and algorithmic features.

7. REFERENCES

[1] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL: A standard template adaptive parallel C++ library. In *Proc. of the International Workshop on Advanced Compiler Technology for High Performance and Embedded Processors (IWACT)*, Bucharest, Romania, Jul 2001.

[2] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL: An adaptive, generic parallel programming library for C++. In *Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Cumberland Falls, Kentucky, Aug 2001.

[3] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. High-performance sorting on networks of workstations. In *Proc. ACM Conference on the Management of Data (SIGMOD)*, pages 243–254, 1997.

[4] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the Connection Machine CM-2. In *Proc. ACM Symp. Par. Alg. Arch. (SPAA)*, pages 3–16, 1991.

[5] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. In *PPOPP*, pages 102–111, 1993.

[6] Eric A. Brewer. High-level optimization via automated statistical modeling. In *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPoPP)*, pages 80–91, 1995.

[7] W.H. Burge. Sorting, trees, and measures of order. *Information and Control*, 1(3):181–197, 1958.

[8] Geeta Chaudhry, Wisniewski Wisniewski, and Thomas H. Cormen. Columnsort lives! an efficient out-of-core sorting program, June 12 2001.

[9] Jaeyoung Choi, J. J. Dongarra, L. S. Ostrouchov, Petitet, A. P., D. W. Walker, and R. C. Whaley. Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Scientific Programming*, 5(3):173–184, Fall 1996.

[10] Jaeyoung Choi, Jack J. Dongarra, and David W. Walker. PUMMA: Parallel Universal Matrix Multiplication Algorithms on distributed memory concurrent computers. *Concurrency: Practice and Experience*, 6(7):543–570, 1994.

[11] Curtis R. Cook and Do Jin Kim. Best sorting algorithm for nearly sorted lists. *Communications of the ACM*, 23(11):620–624, 1980.

[12] Edsger W. Dijkstra. Smoothsort, an alternative for sorting in situ. *Science of Computer Programming*, 1(3):223–233, 1982.

[13] Pedro Domingos and Michael Pazzani. On the optimality of the simple bayesian classifier under zero-one loss. *Machine Learning*, 29:103–130, 1997.

[14] Amr Elmasry and Michael L. Fredman. Adaptive sorting and the information theoretic lower bound. In *STACS*, pages 654–662, 2003.

[15] Vladimir Estivill-Castro and Derick Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys*, 24(4):441–476, December 1992.

[16] G. C. Fox and S. W. Otto. Matrix algorithms on a hypercube I: matrix multiplication.

[17] R. A. Van De Geijn and J. Watts. SUMMA: scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.

[18] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, 1979.

[19] Hyuk-Jae Lee, James P. Robertson, and Jos A. B. Fortes. Generalized cannon's algorithm for parallel matrix multiplication. In *Proceedings of the 11th international conference on Supercomputing*, pages 44–51. ACM Press, 1997.

[20] T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Trans. Comput.*, c-34(4):344–354, 1985.

[21] X. Li, M. J. Garzaran, and D. Padua. A dynamically tuned sorting library. In *Proc. of the International Symposium on Code Generation and Optimization*, pages 111–124, March 2004.

[22] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[23] David Musser, Gillmer Derge, and Atul Saini. *STL Tutorial and Reference Guide, Second Edition*. Addison-Wesley, 2001.

[24] Marek Olszewski and Michael Voss. Proceedings of the international conference on parallel and distributed processing techniques and applications, pdpta '04, june 21-24, 2004, las vegas, nevada, usa, volume 1. In Hamid R. Arabnia, editor, *PDPTA*. CSREA Press, 2004.

[25] Ola Petersson and Alistair Moffat. A framework for adaptive sorting. In *SWAT '92, Third Scandinavian Workshop on Algorithm Theory*, pages 422–433, 1992.

[26] Of Signal Processing. SPIRAL: A generator for platform-adapted libraries.

[27] J. Ross Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[28] L. Rauchwerger, F. Arzu, and K. Ouchi. Standard Templates Adaptive Parallel Library. In *Proc. of the 4th International Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers (LCR)*, Pittsburgh, PA, May 1998.

[29] J. R. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.

[30] D. Rumelhart, G. Hinton, and R. Williams. Learning internal representations by error propagation. *IEEE Transactions on Parallel and Distributed Systems: Explorations in the Microstructure of Cognition*, 1, 1986.

[31] Steven Saunders and Lawrence Rauchwerger. ARMI: an adaptive, platform independent communication library. In *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPoPP)*, pages 230–241. ACM Press, 2003.

[32] Steven Saunders, Nathan Thomas, Nancy Amato, and Lawrence Rauchwerger. Adaptive parallel sorting in the STAPL library. Technical Report TR01-005, Parasol Laboratory, Texas A&M University, November 2001.

[33] Jeremy G. Siek and Andrew Lumsdaine. The matrix template library: A generic programming approach to high performance numerical linear algebra. In *ISCOPE*, pages 59–70, 1998.

[34] R. Vuduc, J. Demmel, and J. Bilmes. Statistical models for empirical search-based performance tuning. *Int. Journal of High Performance Computing Applications*, 18(1):65–94, February 2004.

[35] R. Clint Whaley, Antoine Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, January 2001.

[36] Hao Yu and Lawrence Rauchwerger. Adaptive reduction parallelization techniques. In *ICS '00: Proceedings of the 14th International Conference on Supercomputing*, pages 66–77, New York, NY, USA, 2000. ACM Press.

[37] Hao Yu, Dongmin Zhang, and Lawrence Rauchwerger. An adaptive algorithm selection framework. In *Proceedings of the Parallel Architecture and Compilation Techniques, 13th International Conference on (PACT'04)*, pages 278–289. IEEE Computer Society, 2004.