High-Level Optimization via Automated Statistical Modeling

Eric A. Brewer*
University of California at Berkeley

We develop the use of statistical modeling for portable high-level optimizations such as data layout and algorithm selection. We build the models automatically from profiling information, which ensures robust and accurate models that reflect all aspects of the target platform.

We use the models to select among several data layouts for an iterative PDE solver and to select among several sorting algorithms. The selection is correct more than 99% of the time on each of four platforms. In the few cases it selects suboptimally, the selected implementation performs nearly as well; that is, it always makes at least a very good choice. Correct selection is platform and workload dependent and can improve performance by nearly a factor of three.

We also use the models to optimize parameters of these applications automatically. In all cases, the models predicted the optimal parameter setting, resulting in improvements ranging up to factor of three.

Finally, we use the models to construct portable high-level libraries*, which contain multiple implementations and support for automatic selection and parameter optimization of the fastest implementation for the target platform and workload.*

1 Introduction

Traditionally, libraries achieve portability through the portability of their implementation language, which is usually C or Fortran. Unfortunately, such low-level languages require the developer to encode many platform-specific decisions into the source code. Although these decisions are usually made correctly for the initial platform, they are unlikely to remain optimal over time or after porting.

This dilemma is particularly painful for supercomputers, because the platforms exhibit a wide variance in the relative costs of computation and communication. This variance greatly reduces the chance that algorithm and layout decisions remain even acceptable as an application moves to a new machine. Figure 1 shows the huge variance in communication costs. At the same time, the CPU performance varies less; the slowest is the J-Machine [DAL+89], which actually has solid communication performance.

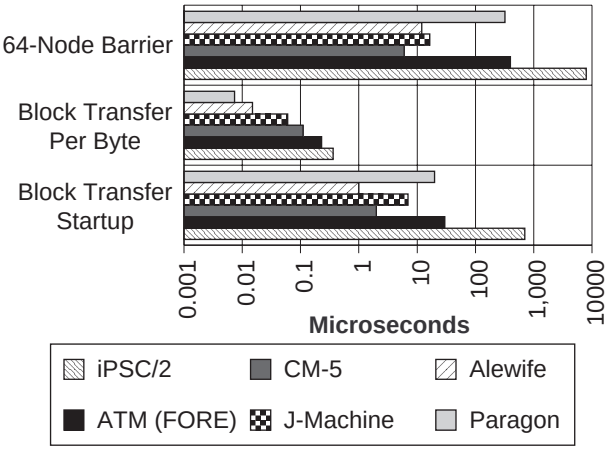

Figure 1: Variance in the costs of communication.

In practice, users end up rewriting their applications for each new platform. By providing several coordinated implementations in one library, we greatly increase the chance that at least one of them performs well on each target platform. The framework also simplifies code reuse over time and distance by providing a complete specification and a method for incorporation of new implementations.

2 Overview

A *high-level library* consists of a set of implementations of a single specification, as shown in Figure 2. Each implementation has a model that predicts its performance on the target platform. The models predict the execution time given a set of input parameters, such as the problem size. To ensure robustness and accuracy, the models are generated automatically by the *auto-calibration toolkit* based on profiling information for each platform, as shown in Figure 3.

The library designer provides the code, the input parameters and their ranges, and a list of terms that form the basis for the statistical models. Given the list of terms and the profiling data, the toolkit 1) uses linear regression to find coefficients for each term, 2) throws out statistically insignificant terms and recomputes the remaining coefficients, and 3) validates the model against an independent set of profile samples. Figure 4 shows an example. Because the toolkit removes irrelevant terms, designers can always add in terms that might be relevant, which allows one list to cover all platforms. Finally, the toolkit computes an accuracy metric, called the *mean relative error* (MRE):

*: Contact: brewer@cs.berkeley.edu, http://www.cs.berkeley.edu/~brewer. This work is supported in part by Project SCOUT, ARPA Contract MDA972-92-J-1032; by ARPA Contract N00014-91-J-1698; by an equipment grant from Digital Equipment Corporation; and by grants from AT&T and IBM. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed on implied, of the U.S. government.

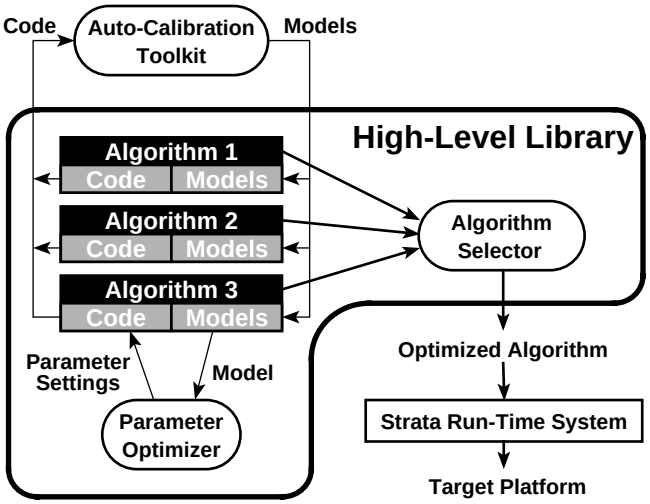

Figure 2: A high-level library consists of a set of algorithms each with code and models. The models used are automatically generated by the auto-calibration toolkit and are used for algorithm selection and parameter optimization. The selected algorithm is then compiled for the target platform using the optimized parameter settings.

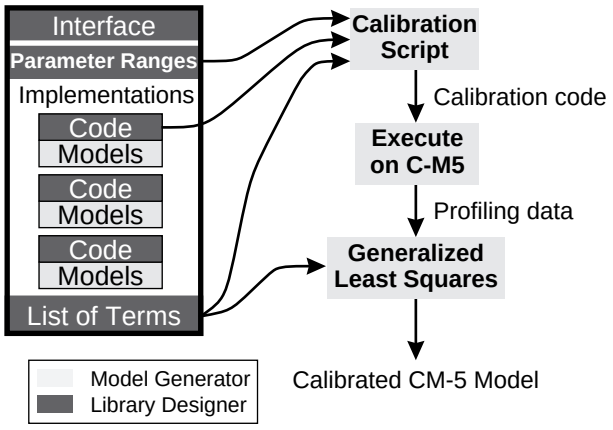

Figure 3: This block diagram describes the automatic generation of models via profiling and linear regression. The dark gray boxes indicate inputs provided by the library designer, while the light gray boxes are provided by the auto-calibration toolkit. Calibration code is generated from the code and the parameter ranges; upon execution this code produces the profiling data used to fit the models terms provided by the designer.

$$\mathrm{MRE} \equiv \sqrt[n]{\prod_{i=1}^{n} \frac{y_i + |y_i - \hat{y}_i|}{y_i}}, \qquad (1)$$

which is just the geometric mean of the relative errors. Intuitively, the MRE tells you how far off the model is on average; we typically express it as a percentage.[1]

Once the calibrated models are available for each platform, the selection is easy: the selector computes a predicted execution time for each implementation and the one with the minimum predicted time is chosen as the winner. Accurate models imply complete success.

Inputs for Stencil Module:

- Profiling Data (automatically collected)
- Parameters: i (iterations), w (width), h (height)
- Ranges: $i \in [1,100]$, $w, h \in [16,10000]$
- List of terms: 1, i, w, h, iw, ih, wh, iwh

Outputs:

- Model: $\mathrm{Stencil}(i, w, h) = 249.7 - 31.09iw - 31.35ih + 4.1iwh$
- MRE: 1.33%

Figure 4: Example input/output pair for the stencil application. The toolkit computes the model based on the list of terms and the profiling data. It throws out irrelevant terms and computes the MRE based on independent samples.

Porting to a new platform thus involves porting Strata and then running the calibration programs for each of the high-level libraries. Porting Strata is not trivial, but it is relatively straightforward and the cost is amortized over all of the libraries, since they only require Strata and C. Occasionally, the list of model terms may have to be extended to capture some unusual behavior of the new platform.

2.1 The Platforms

We evaluate these techniques on four target platforms: the CM-5 and simulated versions of MIT's Alewife [KA93], Intel's Paragon [INTEL91], and a network of workstations based on the FORE ATM switch [FORE92][TLL93]. The three simulated architectures have not been rigorously validated against their actual counterparts, but the Alewife and ATM platforms match their real counterparts on several microbenchmarks. The Paragon version underestimates communication start-up costs.[2] Fortunately, the goal here is not accurate simulation of the platforms, but rather a *wide variety* of costs that stresses the models' ability to track the architectures.

The Alewife has the fastest communication performance, with the CM-5 a close second. The primary difference is Alewife's support for block transfer via DMA. The Paragon is third with a higher start-up cost for communication, but

1: The statistical details are beyond the scope of paper, but we should note that we found MRE to be more useful that the traditional R^2 metric, which emphasizes overall variance. Term relevance is based on the null hypothesis using 95% confidence intervals. Fitting is done via χ^2 using weights that minimize relative rather than absolute error. Complete details are in [BRE94].

2: This is due to numbers from Intel that assumed kernel-level message passing rather than the user-level communication required in practice.

excellent throughput. The local-area network (LAN) has good throughput, but suffers from very high start-up costs. The CM-5 alone has hardware support for global synchronization such as barriers. The CPU performance of the four is about the same; by design, the three simulated architectures have the exact same CPU performance. Thus, these four architectures differ in their relative communication and computation costs: the LAN and Paragon favor few large messages, while the Alewife and CM-5 tend to allow many small messages and more communication in general. The models will in fact capture this knowledge precisely without any guidance from the designer.

3 Stencil Computations

Stencil computations are a technique for iteratively solving partial differential equations. The basic technique is two-dimensional red-black successive over relaxation [FOX+88][BBC+94]. The two-dimensions are discretized into a *virtual grid* and each iteration involves communication with a fixed set of grid neighbors. For example, the basic stencil for successive over relaxation is:

$$x_{i+1} = \frac{\omega}{4}(\text{North} + \text{South} + \text{East} + \text{West}) + (1 - \omega)\, x_i \quad (2)$$

where x_i is the value of the center of the stencil at the i^{th} iteration, the directions refer to grid neighbors, and ω is the degree of overrelaxation. Convergence is detected by tracking the largest update difference, or *delta*, for each node; if all of the nodes had delta values less than some predetermined convergence threshold, then we have a solution.

Conceptually, the stencil is applied to every grid point on every iteration. However, in practice concurrent updates of neighboring grid points can prevent convergence. Thus, the parallel versions use *red-black* ordering, which means that we two-color the grid into red and black regions similar to a checkerboard. We then break each iteration into red and black halves: the corresponding halves are updated in parallel. This prevents concurrent updates, since red vertices only have black neighbors and vice versa. Virtual-grid edges that cross node boundaries require communication, so the choice of data layout greatly affects the performance of the solver.

3.1 Data-Layout Options

There are many ways to partition a virtual two-dimensional grid among the physical processors. We would like a partitioning that ensures load balancing and minimizes communication. However, after requiring that these criteria be met, there remain many reasonable choices. In this section, we examine four choices, which are shown graphically in Figure 5 for a 16-node platform.

The simplest layout is to place all of the data on one processor, called the "uniprocessor" layout. It is primarily useful for small problems that would otherwise be dominated by the communication costs.

The second layout option is the most common: partition the grid into equal-sized blocks. The processors form a square grid; for example, an 800x1600 virtual grid leads to blocks that are 100x200 on each node in a 64-node machine. Note that a square virtual grid results in square blocks. Square blocks minimize communication, since a square has the minimum perimeter for a given rectangular area, and only the elements on the perimeter must be communicated.

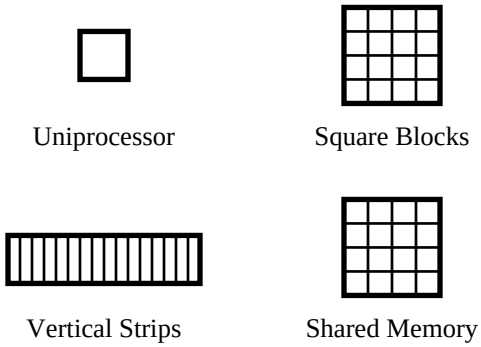

Figure 5: Data Layout Options for Stencil Computations

The third option is vertical strips. Each strip covers the full vertical range of the virtual grid, but only covers a small portion horizontally. Thus, a 1600x1000 virtual grid on a 16-node machine leads to 100x1000 blocks on each node. The strips version minimizes the *number* of messages, rather than the amount of data. For a two-dimensional grid, the strip version only needs to communicate with two neighbors, compared with 4 for the square-block layout. For platforms with high start-up costs for communication, we expect the strips version to perform relatively well.

Finally, the fourth option is to use shared-memory hardware to handle the communication implicitly. The CM-5 does not support shared memory, but the PROTEUS versions can. This is not really a layout option, since all of the previous layouts could use shared memory, but it does represent a different implementation and is subject to selection.

3.2 Results

Figure 6 show the predicted winners for each of the four 64-node platforms for a given width and height of the virtual grid. There are many important facets to this graph. First, the square version seems to be the overall winner. The strips version wins when the aspect ratio is skewed towards short and wide, which makes sense since each node gets the full vertical range. The strips version wins much more on the ATM due to the its high start-up cost for communication.

The uniprocessor version wins only for either small width or small height. However, nothing discussed so far explains why the uniprocessor version would win for very tall but thin grids, since such grids encompass a substantial amount of work. This behavior reflects an important aspect of automatic selection, which is that not all of the implementations need to cover the entire input range. In this case, the square version requires a width and height of at least 16 (a red and black point for each node), since otherwise some

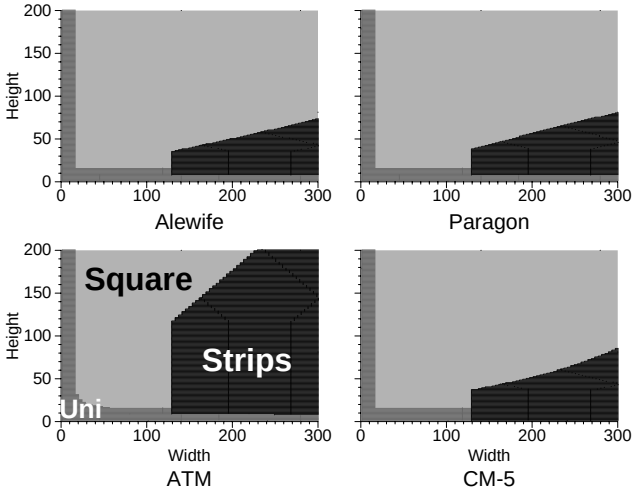

Figure 6: This graph shows the predicted winner for each platform based on the width and height of the virtual grid. The predictions are quite accurate: the models produce the correct answer more than 99% of the time.

nodes would have zero points. The square implementation could be made to handle this boundary case, but only at a significant increase in complexity. Instead, the square version simply prohibits part of the input range. The models support this by returning infinity for unsupported inputs, as shown in Figure 7. The strips version requires a minimum width of 128, but has no restrictions on the height; the width restriction is visible in the figure.

The models thus provide an elegant way to implement only part of the input space without limiting the end user: as long as each input point is covered by at least one implementation, the selector will always pick a valid implementation. In this case, the uniprocessor version covers the entire space and thus acts as a fall back for inputs that are avoided by the parallel implementations. It is hard to overstate the practical importance of this combination: all of the implementations become simpler because they can avoid parts of the input space that are painful to implement. The use of automatic selection hides this simplification from the user by never selecting inappropriate implementations.

Perhaps surprisingly, the shared-memory version *never* wins. This occurs because nodes alternate ownership of the shared edges. Thus, the shared-memory version requires round trips for every cache line along the boundary.[3]

Of course, the figure only shows the *predicted* regions of victory; the key issue is the accuracy of these regions. Fortunately, the predictions are almost always correct, as shown by Table 1. Each of platforms achieved a prediction accuracy of about 99%, with the overall average at 99.42%.[4] The smaller machines did slightly worse, primarily because the implementations tend to be closer in performance.

Each entry in the table represents thousands of executions; the 16- and 32-node numbers represent 500 different virtual-grid sizes and the 64-node numbers represent about 3000 sizes. Each grid size requires four executions, one for each layout. However, even with this many samples the coverage is spotty: the Alewife prediction graph in Figure 6 represents 60,000 grid sizes and thus 240,000 model evaluations, compared with 12,000 simulations. To increase the accuracy of the estimates, we concentrated most of the selected grid sizes along the boundaries in the prediction graph. This leads to a conservative estimate of prediction accuracy, since we expect most of the errors to occur near the boundaries.

Platform	16 Nodes	32 Nodes	64 Nodes	Aggregate
Alewife	99.4%	99.5%	99.51%	99.50%
Paragon	99.3	99.5	99.45	99.39
ATM	98.9	99.2	99.21	99.13
CM-5	99.6	99.8	99.71	99.72
Aggregate	**99.3%**	**99.5%**	**99.52%**	**99.42%**

Table 1: This table shows the accuracy of the model predictions for a range of platforms and machine sizes; in particular, these numbers reflect the percentage of trials in which the selected implementation was indeed the best.

The correlation of errors and boundaries has a very important benefit: when the selector is wrong, it picks an implementation whose performance is near optimal. For example, if the actual execution-time surfaces must be within 4% for the selector to make an error, then the performance penalty for that error can be at most 4%. Thus, in the few cases where the selector is wrong, it will always pick a good implementation. Table 2 presents the performance penalties incurred due to incorrect selections. The average penalty *when wrong* is only 2.12%.

Of course, this number totally neglects the benefit of correct selection! We can not really use the same data to compute the expected gain, since we intentionally concentrated our samples along the boundary. We measure of gain as the average difference between the selected implementation and the square-block implementation. Table 3 shows the average gains for random grids up to 5000x5000.

The ATM version shows significantly more gain simply because the square-block layout wins a smaller proportion of the time. Note that these numbers reflect the possibility of incorrect selections, but none of the 80 samples included an incorrect selection, which matches our expectations. (Table 2 shows that the weighted impact of incorrect selections is only about 0.01%.)

3: An update protocol would help, but it would still not match the efficiency of block transfers.

4: Surprisingly, the CM-5 had the best models, with a noticeable edge over the three simulated platforms. This may be due to systematic inaccuracies in the simulations that are avoided by the real machine, but there is no real evidence for this.

Platform	Mean Penalty When Wrong	Expected Penalty	Worst-Case Penalty
Alewife	2.88%	0.0141%	16.25%
Paragon	3.09	0.0170	13.63
ATM	1.43	0.0113	7.86
CM-5	1.10	0.0032	5.45
Aggregate	**2.12%**	**0.0114%**	**16.25%**

Table 2: This table reveals the penalty for selection errors for the 64-node machines. The second column gives the average penalty *when the wrong implementation is selected*. If we weight the penalty by its probability from Table 1 then we get the expected penalty, shown in the third column. The fourth column gives the worst penalty seen over all of the runs.

Platform	Net Gain
Alewife	8%
Paragon	21%
CM-5	11%
ATM	90%

Table 3: Net Gain from Automatic Selection (20 samples). Individual samples showed gains ranging from close to zero all the way up to 291% for a skewed grid on the ATM network.

$$\text{Uni}(w, h, i) = \quad \text{(MRE=6.84\%)}$$
$$249.7i - 31.09iw - 31.35ih + 4.1iwh$$

$$\text{Strips}(w, h, i) = \quad \text{(MRE=2.34\%)}$$
$$\text{if } (w < 128) \text{ then } \infty$$
$$\text{else } 6.919 + 46.42h + 77.91i + 12.16ih + 0.04393iwh$$

$$\text{Square}(w, h, i) = \quad \text{(MRE=1.00\%)}$$
$$\text{if } (w < 16 \text{ or } h < 16) \text{ then } \infty$$
$$\text{else } 9.04 + 6.186w + 5.478h + 123.1i + 2.716iw + 1.205ih - 0.04406iwh$$

Figure 7: Alewife Models for Stencil: w and h describe the size of the virtual grid, i is the number of iterations

3.3 Models

Figure 7 presents the models for Alewife for the three primary stencil layouts, which are the most complicated and least accurate of the four platforms. Although none of the models are extremely complicated, there are a fair number of terms. The list of terms is just the combinations of *i*, *w*, and *h*. In this case, our initial list was sufficient and the toolkit returned simple and accurate models.

```
for each digit { [start with LSD]
    Locally count each numeral
    Compute the global starting (node,
       offset) pair for each numeral
    Transfer each key to its (node, offset)
    Barrier for completion of reordering
}
```

Figure 8: Pseudo-code for Radix Sort

3.4 Summary

The stencil library achieves performance gains ranging from 8-90% because of its ability to automatically select among multiple implementations, and it always selects implementations that are optimal or near optimal.

4 Sorting

For the second high-level library, we look at parallel sorting, for which there are many algorithms. The goal here is to select the fastest algorithm for the current platform and workload. We look at two algorithms: radix sort and sample sort. Both implementations are based on a sorting study for Thinking Machines' CM-2 by Guy Blelloch *et al.* [BLM+92].

4.1 Radix Sort

Figure 8 gives pseudo-code for parallel radix sort. The main loop covers each of the digits, normally a particular set of contiguous bits from a larger integer key. For example, a 32-bit integer can be thought of as 8 4-bit digits, 4 8-bit digits, or even 3 11-bit digits. Increasing the width of a digit reduces the number of loop iterations, but increases the work within each iteration. The next section examines how to set the digit size optimally and automatically for the target platform.

The first step for each digit is to count the number of keys with each digit value, or numeral. The result for the first step for a base-*r* radix sort is a set of *r* counts, one for each numeral.

In the second step, we determine the numeral counts globally by combining all of the local counts. This is done using a vector-scan operation with a vector of *r* words.[5] Thus, each node knows the total number of keys to its left for each numeral; we refer to this count as the *offset*.

At the same time, we also compute the total number of each numeral, using a vector reduction. We can combine the total and offset to determine where to send each key. For example, if there 1000 keys with numeral zero, then we say that the first 1000 elements of the target array hold numeral zero; numeral one starts at index 1001. Combining this with the offsets, we get exact target positions for each key. If the

5: In a *scan* operation, each node i contributes an element x_i and receives the value $x_0 \oplus x_1 \oplus \ldots \oplus x_{i-1}$, where "⊕" denotes an associative operator, in this case addition. Thus, each node gets the sum of the values to its left, with node 0 receiving 0. A *reduction* is similar, except that each node gets the total sum for all nodes. In a *vector scan*, each node contributes a vector and the result is the vector of the element-wise scans.

numeral-one offset for node 3 was 225, then the overall starting position for node 3's numeral-one keys would be $1001 + 225 = 1226$. This array is then evenly blocked across all of the processors.

We make one pass through the data, sending the data according to the index for its numeral, and then incrementing the index. We repeat this process for each digit.

4.2 Sample Sort

The primary drawback of radix sort is that it rearranges all of the data for every digit. Ideally, we would just move the key directly to its final destination. This is the goal of sample sort, which uses sampling to guess the boundary keys for each node. Reif and Valiant developed the first sampling sort, called *flashsort*, in 1983 [RV87]. Another early version was implemented by Huang and Chow [HC83].

For n keys and p processors we get the following basic algorithm:

1. Find $p-1$ keys, called *splitters*, that partition the key space into p buckets,
2. Rearrange the data into the p buckets, which correspond to the p processors,
3. Sort the keys within each bucket.

Unfortunately, it is possible for the splitters to be far enough off that one of the processors is the target for more keys than it can physically hold in memory. Furthermore, the wider the variance in keys per node, the longer the final local sorting phase will take, since the time of the phase is limited by the processor with the most keys.

Fortunately, there is a nice solution to both of these problems: we can use s buckets per processor, where s is called the *oversampling ratio*. For example, if $s = 10$ there is still a chance that one of the buckets will have many more keys than the average, but it very unlikely that all ten of the buckets on one processor will contain many more keys than average. The expected number of keys per node is $\frac{n}{p}$; if we call the maximum number of keys per node L, then the we can define the work ratio as $\alpha \equiv \frac{L}{(n/p)}$. Blelloch *et al.* [BLM+92] showed that $\Pr[\alpha > 2]$ decreases exponentially with s: with $s = 64$ and 10^6 keys, $\Pr[\alpha > 2.5] < 10^{-6}$. In practice, oversampling ratios of 32 or 64 ensure even load balancing of the keys, essentially always resulting in $\alpha < 2$.

Figure 9 shows the pseudo-code for sample sort with oversampling. There is some small chance that a node will have insufficient memory for the incoming keys. We assume space for $\frac{2n}{p}$ keys plus the buffer storage. When there is insufficient space for an incoming buffer, we forward it randomly; we know that there is sufficient space somewhere. We then simply restart the algorithm after the transfer phase,

```
Pick s keys at random on each node,
    for a total of ps keys
Sort the ps keys using parallel radix sort
Send every s^th key to node 0;
    [these are the splitters]

Node 0 broadcasts the splitters

For each of the n/p keys (on each node) {
    Find the target node via binary search
    Group the keys into buffers by proc
    When a buffer becomes full,
        send it to its node
}

Transfer all non-empty buffers
Block until all transfers are complete
    [by counting outstanding transfers]

Repeat for load balancing if needed
    [discussed in text]

Sort data locally to complete the sort
```

Figure 9: Pseudo-code for Sample Sort

which is not as bad as it sounds. First, the chance that there is insufficient storage is near zero. Second, when we repeat the algorithm we get independent splitters, so we expect the second iteration to succeed. Third, *most* of the data is already on the correct node.

Given that sample sort almost always rearranges the data only once, it should beat radix sort for large numbers of keys or for platforms with expensive communication. However, it is more complicated than radix sort, requires more memory, and has substantially more set-up work to perform. Thus, we expect radix sort to do well on smaller problems and perhaps on platforms with high-performance communication. Furthermore, the time for radix sort depends linearly on the width of the keys, while sample sort is essentially independent of the size of the key space.

4.3 The Sorting Module

The sorting module sorts integer keys with widths from 4 to 32 bits. It assumes each node contributes a fixed-size buffer that is at most half full of keys; after the sort, the buffers contain each node's chunk of the sorted keys, with keys strictly ordered by processor number. The module also returns a list of $p-1$ keys that can be used to binary search for the node containing a given key.

The module has two parameters: the number of keys per node and the width of a key in bits. We select among four algorithms:

1. Sample Sort
2. Radix Sort with 4 bits per digit, radix = 16
3. Radix Sort with 10 bits per digit, radix = 1024
4. Radix Sort with 14 bits per digit, radix = 16384.

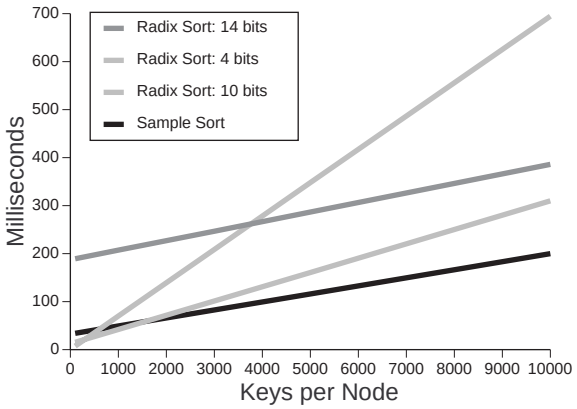

Figure 10: Sorting Performance on a 64-Node CM-5. This graph shows the performance of the four sorting algorithms for a 64-node CM-5 and 28-bit keys. Sample sort is the best for large problems, while the small-radix radix sorts work best for small problems.

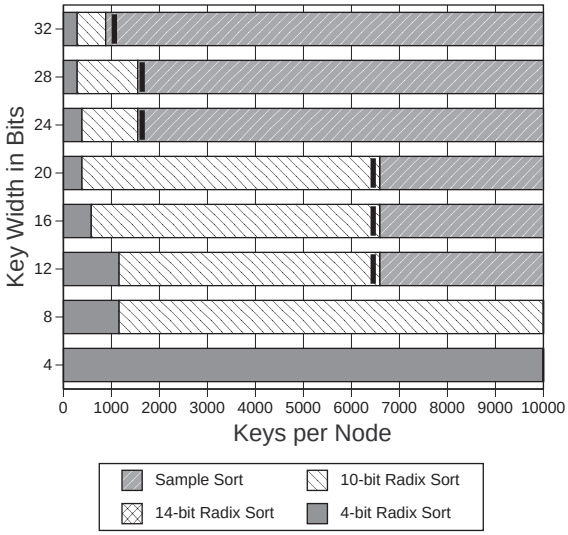

Figure 11: This plot shows the best algorithm for a given key width and keys per node for a 64-node CM-5. These regions are the predictions: the black bars denote the actual crossover point. The bars are omitted for clarity if the predicted crossover is within 0.5% of the actual crossover point. Thus, the models accurately predict the best algorithm. Note that 14-bit Radix Sort never wins.

4.4 Results

We start by looking at the relative performance on a 64-node CM-5 with 28-bit keys. Figure 10 shows the performance of the four algorithms. Figure 11 shows the prediction graph for the 64-node CM-5. As expected, the small-radix radix sorts work best for small problems, while sample sort is the best asymptotically. Radix sort with a 14-bit radix never wins: although it is the best version of radix sort for larger problems, sample sort is even better for those same inputs.

Several bars contain black bars. These bars mark the actual crossover point for the corresponding region boundary. To clarify the predicted crossovers, the bars have been omitted if the actual crossover point is within 50 keys per node of the prediction, which is [one-fortieth?] of an inch graphically.

Platform	Key Width	4-bit Radix	10-bit Radix	Sample Sort	Error Ratio
Alewife: pred	16	1–2584	2585–	—	$\frac{20}{10000}$
Alewife: act		1–2604	2605–	—	
Alewife: pred	24	1–1295	—	1296–	$\frac{7}{10000}$
Alewife: act		1–1288	—	1289–	
Alewife: pred	32	1–817	—	818–	$\frac{2}{10000}$
Alewife: act		1–819	—	820–	
CM-5: pred	16	1–580	581–6597	6598–	$\frac{49}{10000}$
CM-5: act		1–583	584–6551	6552–	
CM-5: pred	24	1–386	387–1551	1552–	$\frac{61}{10000}$
CM-5: act		1–391	392–1607	1608–	
CM-5: pred	32	1–290	291–879	880–	$\frac{5}{10000}$
CM-5: act		1–291	292–875	876–	
Paragon: pred	8	1–661	662–	—	$\frac{31}{10000}$
Paragon: act		1–692	693–	—	
Paragon: pred	16	1–553	—	554–	$\frac{6}{10000}$
Paragon: act		1–547	—	548–	
Paragon: pred	32	1–215	—	216–	$\frac{0}{10000}$
Paragon: act		1–215	—	216–	
ATM: pred	8	1–383	384–	—	$\frac{8}{10000}$
ATM: act		1–391	392–	—	
ATM: pred	16	1–312	—	313–	$\frac{6}{10000}$
ATM: act		1-318	—	319–	
ATM: pred	32	1–126	—	127–	$\frac{1}{10000}$
ATM: act		1-127	—	128–	

Table 4: This table gives the percentage of incorrect predictions for keys per node in the range of 1 to 10,000 on 64-node platforms. The overall mean accuracy is 99.84%. The middle columns give the predicted and actual key ranges for which each algorithm is best.

Table 4 gives the predicted and actual regions for all of the platforms, along with a summary of the prediction errors. Given the predicted and actual intervals, we can compute the number of different keys-per-node values for which the prediction was wrong. For example, if a crossover point was off by ten keys, then we count that error as ten incorrect predictions. We take the total number of incorrect predictions and divide by 10000, which is our nominal range of interest, to get the "Error Ratio" column. We leave the ratio as a fraction because the denominator is rather arbitrary. Nonetheless, the number of errors is much smaller than the range of interest in practice.

In general, radix sort does much better on the CM-5 than on the other platforms. This is primarily due to two factors: the hardware support for vector-scan operations, and the absence of an integer multiplication instruction, which slows down the random-number generation required by sample

$$\text{RadixSort}(keys, bpd, width) = 11.41 \cdot bins + 9.92 \cdot iter \cdot keys + 77.36 \cdot logP$$

where:

$keys \equiv$ Number of keys per node

$bpd \equiv$ Radix width in bits (bits per digit)

$width \equiv$ Key width in bits

$iter \equiv \left\lceil \frac{width}{bpd} \right\rceil \equiv$ Number of iterations

$bins \equiv 2^{bpd} \equiv$ Number of buckets (the radix)

$logP \equiv$ Log base 2 of the number of processors

Figure 12: CM-5 Radix Sort Model

sort.

In general, radix sort is competitive only for small problems or small key widths. However, radix sort wins if it only requires one iteration. The more interesting case is the Alewife platform with 16-bit keys, in which 10-bit radix sort requires two passes to sample sort's one. However, the communication performance of the Alewife is fast enough that the cost of a second pass for radix sort is less than the additional overhead for sample sort.

For the stencil module, the penalty for prediction errors was small because they only occur when there are multiple viable options. For sorting, it is nearly impossible to measure a penalty that is statistically significant. This is due to the fact that the performance of sample sort varies depending on the random-number generator: the crossover points from radix sort to sample sort are always nebulous. As we move away from the crossover point, the difference in performance starts to dominate the white noise and there should be a measurable penalty. But there are no errors away from the crossover points!

4.5 Models

These models are slightly simpler than those for the stencil module, with the CM-5 model for radix sort, shown in Figure 12, standing out as exceptionally simple and accurate. The first term is the cost of the buckets, which require global vector scans and reductions. The second term is the cost of rearranging the data, which depends of the number of iterations and the number of keys per node. The last term gives the marginal cost for the machine size.

Thus, this model captures the performance of radix sort on the CM-5 with only three terms and yet achieves excellent accuracy (MRE = 1.33%). The original list of terms included many other combinations that seemed important, such as the number of processors, but the toolkit threw them out as superfluous.

In general, the sample-sort models are less accurate than those for radix sort. This makes sense given the variance in execution times due to the random-number generator: some runs are more load balanced than others.

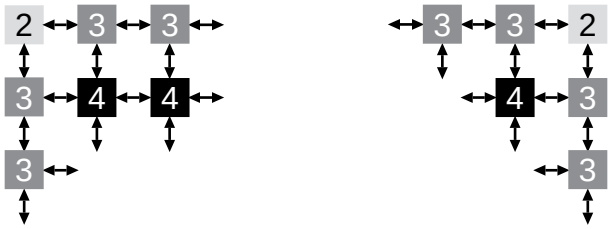

Figure 13: Boundary processors have fewer neighbors to communicate with than nodes in the center.

5 Parameter Optimization

The key idea in this section to is combine two powerful techniques: automatically calibrated statistical models and numerical root finding. Given models that accurately predict the cost of the code for a given parameter, we can use root finding to give us the optimal value. We do this in two different ways depending on the nature of the models. For both methods, the designer specifies the range of interest of the parameter.

First, if we can set up the problem as the equality of two models, then we can build a new model that computes the difference of the models. The root of the new model is the equality point of the original models. The root finder is:

```
int IntegerRoot(int bottom, int top,
  float f(int param))
```

This procedure finds the root of `f` in the interval [`bottom`, `top`]. If there is no root in the given range, then the root finder returns `bottom`-1 if the curve is strictly negative, and `top`+1 if the curve is strictly positive. If there is more than one root, the root finder returns one of them.

In the second method, we assume that the model has a single minimum and use a technique similar to root finding to locate the minimum. Essentially, the algorithm is binary search, using the numerical derivative at each point to indicate in which half the minimum resides. The minimum finder is:

```
int IntegerMinimum(int bottom, int
  top, float f(int param))
```

This procedure finds *a* minimum of `f` in the interval [`bottom`, `top`].[6]

The overall methodology, then, is to build a composite model, $f(x)$, that either has a root or a minimum, and then to use one of the above procedures to find the optimal value of x. This can be done either at compile time or run time.

5.1 Stencils: Extra Rows and Columns

A simple but useful optimization for stencil programs is to give the boundary processors extra work to do to make up for their reduced communication. Figure 13 shows the num-

6: It may be that other optimizations require a more sophisticated tool for finding the global minimum, but for these applications the single-minimum requirement was sufficient.

ber of communication directions for different nodes in the square-block layout. Nodes with fewer directions sit idle part of the time. Thus, we can give the boundary processors more grid points, and the middle nodes correspondingly less. This improves load balancing. The benefit of this optimization is small for large machines, since there are relatively few boundary nodes. Nonetheless, it never hurts and is a good example of the methodology.

5.1.1 The Composite Model

To build our composite model, we combine a model for the cost of the missing communication with a model for the cost of the additional grid points. The model for the missing communication is:

$$\text{Comm}(w) = \textit{BlockTransfer}\left(\frac{8 \cdot w}{2(8)}\right) \quad (3)$$

where *BlockTransfer(x)* is the cost for a block transfer of x bytes. The size is multiplied by 8 to convert doubles to bytes, divided by two because of red-black ordering, and divided by 8 because the width is divided among 8 nodes. The statistical fitting is hidden within the *BlockTransfer* function, which is one of the Strata models.

The model for the marginal communication cost uses the Square model from Figure 7:

$$\text{Computation}(w, h, x) = \frac{\text{Square}(w, h + 8x, 1000) - \text{Square}(w, h, 1000)}{1000} \quad (4)$$

We exploit the fact that the marginal cost for x extra rows per node is the same as the marginal cost for $8x$ extra rows for the whole problem, since there are 8 processors along the vertical dimension (assuming 64 nodes). We compute the cost for 1000 iterations and then divide by 1000 to get the marginal cost per iteration, which effectively throws out initialization costs. Also, note that the marginal cost depends on the existing size of the virtual grid, but the Square model already captures this behavior.

Given our two submodels, we define the composite model as the difference between them:

$$f(x) \equiv \text{Comm}(w) - \text{Computation}(w, h, x) \quad (5)$$

We give this function to the `IntegerRoot` procedure to calculate the optimal number of extra rows. A similar function is used for the optimal number of extra columns.

5.1.2 Results

Table 5 shows the predictions for the optimal number of extra rows and columns. Overall, we tried 100 different configurations of platforms and problem sizes: in all 100 trials, the predicted value was in fact optimal.

We can also use the models to predict the *benefit* of the optimization. In particular, if we assume that the extra rows and columns simply fill up idle time on the boundary processors, then the performance of the optimized version is nearly identical to the performance of a problem with that many fewer rows and columns. Table 5 gives the predicted and actual benefit of the optimization. The predicted value is uniformly high, which is due to the fact that the implementation requires an integer number of additional rows, and thus the cost of a whole row can not always be hidden in the idle time. As expected, the benefit of the optimization is relatively small, and has less impact on bigger problems.

Platform, Size 64 Processors	Rows	Cols	Opt	Benefit	
				Pred	Actual
Alewife, 200x200	1	1	✓	1.6%	1.4%
CM-5, 200x200	2	1	✓	3.1%	2.5%
ATM, 50x50	9	2	✓	8.9%	8.1%
Paragon, 200x200	3	1	✓	3.3%	2.6%

Table 5: This table shows the predicted number of extra row and columns required to provide load balancing. All of the predictions were correct. The models can also predict the benefit of the optimization.

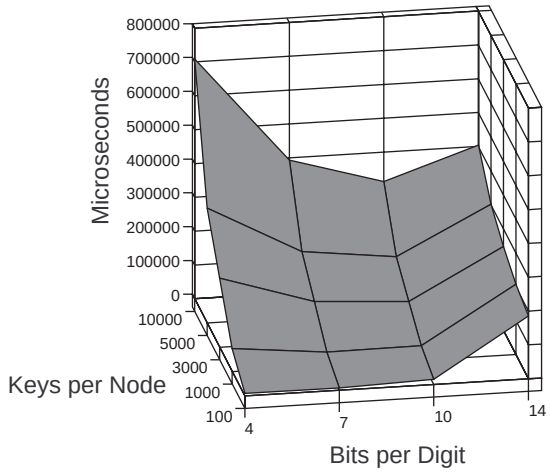

Figure 14: This graph plots the performance of radix sort on a 64-node CM-5 with 28-bit keys, versus the width of the digit in bits and the number of keys per node. For less than 1000 keys per node, the optimal width is 4 bits, around 1000 it is 7 bits, and above 1000 it is 10 bits. For large enough sorts, 14-bit digits would be optimal.

5.2 Radix Sort: Optimal Digit Size

In our second example, we attempt to pick the optimal radix for radix sort. The best radix depends on the platform, the problem size, and the width of the keys. Figure 14 shows the effect of the digit size and problem size on the performance of radix sort for a 64-node CM-5 with 28-bit keys. By using parameter optimization, we can build one implementation that covers all of the different radix values.

5.2.1 The Composite Model

In fact, our existing models for radix sort already take the digit size as a parameter (Figure 12). Thus, we can build a composite model quite trivially:

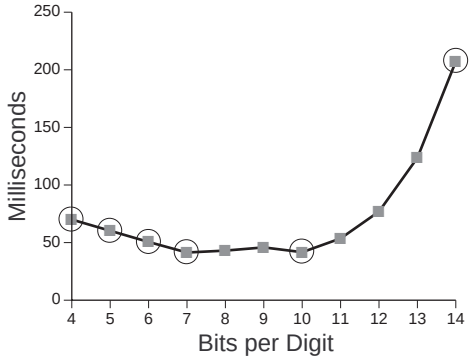

Figure 15: This graph plots the proposed composite model from equation (6) against the parameter (bits per digit). The graph has multiple minima and thus could lead to suboptimal values. Only the circled nodes are considered by the revised model of equation (7), which only has one minimum (at 7).

$$f(x) \equiv \text{Radix}(keys, x, key\_width). \quad (6)$$

To find the optimal radix, we simply invoke `IntegerMinima` with *f* and a reasonable interval, such as [1, 16].

Unfortunately, the specified *f* has multiple minima. Figure 15 shows the graph of $f(x)$ versus *x* for a 64-node CM-5 with 28-bit keys and 1000 keys per node. The drop from 9 to 10 occurs because a 9-bit radix requires four iterations, while a 10-bit radix needs only three.

The solution is to build a model that only looks at the minimum radix size for a given number of iterations. Thus, we parameterize the composite model in terms of iterations, from which we can compute the best radix. In particular, for *i* iterations, the best digit width is $\left\lceil \frac{key\_width}{i} \right\rceil$:

$$g(i) \equiv \text{Radix}\left(keys, \left\lceil \frac{key\_width}{i} \right\rceil, key\_width \right). \quad (7)$$

The viable sizes are circled in Figure 15; connecting the only the circled nodes leads to a curve with exactly one minimum at $i = 4$, which corresponds to a digit width of 7 bits.[7]

5.2.2 Results

As with the stencil example, the predicted optimal radix was correct in all trials (40). We checked optimality by testing all of the radix values.

It is difficult to access the average benefit of this optimization. The overall best choice is probably 8-bit digits, but it is not obvious. In any case, any single choice will be very bad for some inputs. For example, with a key width of 9, 9-bit digits can perform up to twice as well as 8-bit digits. Similarly, with a key width of 4, 4-bit digits on small problems can outperform 8-bit digits by more than a factor of three.

Although the optimal value is important for good performance, it is very difficult to determine manually. The platform, problem size, and key width all have substantial impact on the best digit width, yet it is not obvious how to combine these influences. Optimizing any single one alone leads to inputs with poor performance. The beauty of model-based parameter optimization is the ease with which it perfectly combines these influences: the composite model precisely captures the importance of each factor.

Given this family of radix sorts that use the optimal digit width, we can now build a better sorting module. In particular, to perform algorithm selection, we first determine the optimal digit width. We then select between radix sort with the optimal radix and sample sort. This results in a module with fewer implementations and better overall performance.

Metric	Stencils	Sorting
Implementations	5	3
Average code expansion	290%	210%
Selection (CM-5)	10.2 μs	6.0 μs
Optimization iterations	5	4
Optimization (CM-5)	51 μs	42 μs

Table 6: Overheads for run-time selection/optimization.

6 Run-Time Model Evaluation

Although it has only been hinted at so far, the model-based decisions can be made at run-time as well as at compile time. The auto-calibration toolkit simplifies run-time evaluation by generating C models that we can link in with the application.

Table 6 covers the two drawbacks to run-time evaluation: code expansion and selection overhead. Expansion is measured against the average of the individual implementations.[8] The memory requirements should be significantly better, since only one implementation will be paged in. The overhead for run-time selection is about two microseconds per implementation, although it depends on the application and the platform. This is trivial compared to the typical execution times, which range from milliseconds to minutes.

Parameter optimization can be also be done at compile time or run time, but if done at compile time, some of the relevant inputs, such as the number of keys per node for radix sort, might have to be filled in with generic values. Unlike algorithm selection, run-time parameter optimization avoids code expansion. The overhead is about 10 microseconds per iteration, which again is in the noise.

7 Related Work

The closest related work is that of Alan Sussman [SUS91][SUS92]. Sussman's system chooses the best mapping of data from a fixed set of choices for compilation onto a linear array of processors. He shows that the choice of mapping

7: The 10-bit radix is 0.31 milliseconds slower than the 7-bit version.

8: The fifth stencil implementation is horizontal strips. The 14-bit version of radix sort was removed from the sorting module.

greatly affects the performance of two image processing algorithms and that his system picks the best mapping by using execution models. This work extends Sussman's work by combining execution models with parameterized modules by looking at general topologies, and by addressing algorithm decisions in addition to layout decisions.

Predictors for Fortran include PTRAN [ABC+88] and Parafrase-2 [POL+89]. PTRAN determines the maximum available parallelism for a semantically single-threaded Fortran program and can use this information to establish a lower bound on execution time for a given number of processors. Parafrase-2 extends this by breaking the execution time into categories based on various operations such as floating-point operations and memory accesses.

Cytron [CYT85] looks at determining the optimal number of processors to use for a fork-join task graph such as those generated by a "doall" loop. Chen, Choo and Li [CCL88] present a simple performance model based on the number of processors and the times for unit computation and communication. Their optimization phase uses the model to evaluate transformations. Culler *et al.* [CKP+93] propose the *LogP* model as tool for algorithm development and analysis. The models used here subsume all three of these and implicitly handle many additional aspects, such as bisection-bandwidth limitations, OS overheads, and caching effects. In general, the models used in this work are more accurate and robust, and support automatic recalibration.

Gallivan *et al.* [GJMW91] investigate model-based Fortran performance prediction in two steps. First they build a database of "templates" that represent common loop patterns. A template captures the load-store behavior and the vector behavior of the Alliant FX/8 multiprocessor. The templates are simple models based on profiling code for a representative loop. The second step is to map application loops onto those in the database, via compiler analysis. There will be inaccuracies depending on how close a match the compiler can find. The loops must avoid conditional statements, procedure calls, and nonlinear array indices. Looking at phases allows us to avoid their restrictions: we amortize the effects of conditional statements over time. We also automate the modeling, which allows portability.

A. Dain Samples' doctoral dissertation [SAM91] presents a optimizer that selects among implementations of a sequential module based on profile information. Our system generalizes this technique, applies it to parallel computing, uses statistical models as the medium for feedback, and separates the modeling from the decision making, which extends and simplifies the use of the profiling information.

Some systems use programmer-supplied annotations to control the implementation of a module. High Performance Fortran [HPF] uses programmer directives to control the layout of arrays. Prelude [WBC+91] used annotations to control the migration of data and threads. The annotations are a short-term solution that side-steps the difficulties of automatic layout or migration.

Many languages and libraries provides support for modularity, including Multipol [YEL92], Modula-3 [NEL91] and Standard ML [MTH90], but none address automatic selection or optimization of multiple implementations.

There are a myriad of adaptive algorithms. These algorithms use feedback to adjust the key parameter dynamically, which may take some time to reach the optimal value. A good example is the "slow start" mechanism used from TCP [JAC88]. For short messages, as seen with the World Wide Web, the optimal value is never reached, and the connection achieves very low bandwidth. In contrast, model-based parameter optimization starts with the optimal value. The feedback-based algorithms are more robust: they can handle dynamic or unknown environments. A hybrid may work best: compute the starting value and adapt over time.

8 Conclusions

The performance of the stencil module strongly depends on the layout of the virtual grid onto physical processors. The square-block layout is the best overall, but the uniprocessor and vertical strips versions are often better. The shared-memory version never wins.

The performance of the sorting algorithms depends on the size of the problem, the width of the keys, and the target platform. The radix sorts work best for small problems or small key widths, while sample sort is usually better asymptotically.

The models for both modules were quite accurate: the selector picked the best implementation more than 99% of the time on all of the platforms. In the few cases in which a suboptimal implementation was selected, that implementation was nearly as good as the best choice: only a few percent slower on average for the stencils and nearly identical for sorting. The system only makes errors near the boundaries of the performance surfaces, from which it follows that the penalty for these errors is small.

In contrast, the benefit of picking the right implementation was often very significant: averaging 8-90% for stencils and often more than a factor of two for sorting. We also used the models for parameter optimization. We found no errors in the prediction of the optimal parameter value (140 trials), and benefits ranging from a few percent to a factor of three.

An important benefit of automatic selection is the tremendous simplification of the implementations that arose from the ability to ignore painful parts of the input range. By adding preconditions into the models, we ensure that the selector never picks inappropriate implementations. Thus, an immediate consequence of automatic selection is the ability to combine several simple algorithms that only implement part of the input range.

In general, porting a high-level library requires much less work than porting an application. The implementations depend on only C and Strata. The libraries are essentially self-tuning: they combine the flexibility of multiple implementations with an automatic method for selecting the best one in the new environment. Applications that use the librar-

ies are also more portable, because many of the performance-critical decisions are embedded within the libraries.

Model-based parameter optimization provides a form of adaptive algorithm. In fact, one way to view this technique is as a method to turn algorithms with parameters that are difficult to set well into adaptive algorithms that compute the best value. This brings a new level of performance to portable applications: the key performance parameters are set optimally and automatically as the environment changes.

Selection can be done at run time as well as compile time. The code expansion is small and the overhead is insignificant compared to the execution time. Run-time parameter optimization has low overhead, no code expansion, and can exploit workload information, such as the problem size, that is only available dynamically.

Finally, parameter optimization can be combined with automatic algorithm selection to produce simple, more powerful modules, with better overall performance. The selection is done in two passes: first, we find the optimal parameter values for each of the parameterized implementations, and then we select among the optimized versions. For both applications, there are inputs for which the selected and optimized version is more than three times faster than best overall (optimized) implementation.

Acknowledgments: Thanks to Bill Weihl for advising this work. Charles Leiserson, Butler Lampson and Greg Papadopoulos provided helpful feedback as well. Thanks also to Lisa Sardegna, Sanjay Ghemawat, and David Chaiken.

[ABC+88] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An Overview of the PTRAN Analysis System for Multiprocessing. *Journal of Parallel and Distributed Computing*, Volume 5, Number 5, pages 617–640, October 1988.

[ACP95] T. E. Anderson, D. E. Culler and D. A. Patterson. The Case for NOW. *IEEE Micro*, to appear, 1995.

[BBC+94] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* Society for Industrial and Applied Mathematics, 1994.

[BK94] E. A. Brewer and B. C. Kuszmaul. How to Get Good Performance from the CM-5 Data Network, *Proceedings of the 1994 International Parallel Processing Symposium (IPPS '94)*, April 1994.

[BLM+92] 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*. Technical Report 222, Thinking Machines Corporation, 1992.

[BRE94] E. A. Brewer. *Portable High-Performance Supercomputing: High-Level Platform-Dependent Optimization.* MIT Ph.D. Dissertation, September 1994.

[CCL88] M. Chen, Y. I. Choo, and J. Li. Compiling Parallel Programs by Optimizing Performance. *Journal of Supercomputing*, Volume 2, Number 2, pages 171–207, October 1988.

[CKP+93] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of Principles and Practice of Parallel Processing (PPoPP '93)*, May 1993.

[CYT85] R. Cytron. Useful Parallelism in a Multiprocessor Environment. In the *Proceedings of the 1985 International Conference on Parallel Processing*, August 1985.

[DAL+89] W. J. Dally *et al.* The J-Machine: A Fine-Grain Concurrent Computer. In G.X. Ritter, editor, *Proceedings of the IFIP Congress*, pages 1147–1153. North-Holland, August 1989.

[FOX+88] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems on Concurrent Processors, Volume 1: General Techniques and Regular Problems*. Prentice Hall, 1988.

[INTEL91] Intel Corporation. *Paragon XP/S Product Overview*, 1991.

[FORE92] FORE Systems. *TCA-100 TURBOchannel ATM Computer Interface: User's Manual*. 1992.

[GJMW91] K. Gallivan, W. Jalby, A. Malony, and H. Wijshoff. *Performance Prediction for Parallel Numerical Algorithms*. Technical Report, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 1991.

[HC83] J. S. Huang and Y. C. Chow. Parallel Sorting and Data Partitioning by Sampling. In the *Proceedings of the 7th Computer Software and Applications Conference (COMPSAC '83)*, pages 627–631, November 1983.

[HPF] D. Loveman, *High-Performance Fortran Proposal.* Presented at the Rice University High-Performance Fortran Forum, January 27, 1992. [ftp: `titan.cs.rice.edu`]

[JAC88] V. Jacobson. Congestion Avoidance and Control. In the *Proceedings of SIGCOMM '88*, August 1988.

[KA93] J. Kubiatowicz and A. Agarwal. Anatomy of a Message in the Alewife Multiprocessor. *Proceedings of the 7th ACM International Conference on Supercomputing*, July 1993.

[LEI+92] C. E. Leiserson *et al*. The Network Architecture of the Connection Machine CM-5. *Proceedings of the 1992 Symposium on Parallel Algorithms and Architectures (SPAA '92)*, revised March 21, 1994.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[NEL91] G. Nelson. *Systems Programming with Modula-3.* Prentice Hall, 1991.

[POL+89] C. D. Polychronopolous *et al.* Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing, and Scheduling Programs on Multiprocessors. In the *Proceedings of the 1989 International Conference on Parallel Processing*, Volume II, pages 39–48, August 1989.

[RV87] J. H. Reif and L. G. Valiant. A Logarithmic Time Sort for Linear Size Networks. *Journal of the ACM*, Volume 34, Number 1, pages 60–76, January 1987.

[SAM91] A. Dain Samples. *Profile-Driven Compilation.* Ph.D. Dissertation, University of California at Berkeley Technical Report UCB/CSD 91/627, April 1991.

[SUS91] A. Sussman. *Model-Driven Mapping onto Distributed-Memory Parallel Computers.* Ph.D. Dissertation, Carnegie-Mellon University Technical Report CMU-CS-91-187, September 1991.

[SUS92] A. Sussman. Model-Driven Mapping onto Distributed-Memory Parallel Computers. In the *Proceedings of Supercomputing '92*, pages 818–829, August 1992.

[TLL93] C. A. Thekkath, H. M. Levy and E. D. Lazowska. *Efficient Support for Multicomputing on ATM Networks*. University of Washington Technical Report 93-04-03. Department of Computer Science and Engineering, April 12, 1993.

[VE+92] T. von Eicken *et al*. Active Messages: A Mechanism for Integrated Communication and Computation. *Proceedings of the 19th International Symposium on Computer Architecture (ISCA '92)*, May 1992.

[WBC+91] W. E. Weihl, E. A. Brewer, A. Colbrook, C. Dellarocas, W. C. Hsieh, A. Joseph, C. A. Waldspurger, and P. Wang. *Prelude: A System for Portable Parallel Software.* MIT Technical Report MIT/LCS/TR-519, October 1991.

[YEL92] K. Yelick. Programming Models for Irregular Applications. *Proceedings of the Second Workshop on Languages, Compilers, and Run-Time Environments for Distributed-Memory Multiprocessors*, in *ACM SIGPLAN Notices*, 28(1), 17–20, January 1993.