Autotuning Multigrid with PetaBricks

Cy Chan[†], Jason Ansel[†], Yee Lok Wong[⋆], Saman Amarasinghe[†], Alan Edelman[†⋆]
[†] *CSAIL, Massachusetts Institute of Technology, Cambridge, MA 02139*
[⋆] *Dept. of Mathematics, Massachusetts Institute of Technology, Cambridge, MA 02139*

ABSTRACT

Algorithmic choice is essential in any problem domain to realizing optimal computational performance. Multigrid is a prime example: not only is it possible to make choices at the highest grid resolution, but a program can switch techniques as the problem is recursively attacked on coarser grid levels to take advantage of algorithms with different scaling behaviors. Additionally, users with different convergence criteria must experiment with parameters to yield a tuned algorithm that meets their accuracy requirements. Even after a tuned algorithm has been found, users often have to start all over when migrating from one machine to another.

We present an algorithm and autotuning methodology that address these issues in a near-optimal and efficient manner. The freedom of independently tuning both the algorithm and the number of iterations at each recursion level results in an exponential search space of tuned algorithms that have different accuracies and performances. To search this space efficiently, our autotuner utilizes a novel dynamic programming method to build efficient tuned algorithms from the bottom up. The results are customized multigrid algorithms that invest targeted computational power to yield the accuracy required by the user.

The techniques we describe allow the user to automatically generate tuned multigrid cycles of different shapes targeted to the user's specific combination of problem, hardware, and accuracy requirements. These cycle shapes dictate the order in which grid coarsening and grid refinement are interleaved with both iterative methods, such as Jacobi or Successive Over-Relaxation, as well as direct methods, which tend to have superior performance for small problem sizes. The need to make choices between all of these methods brings the issue of variable accuracy to the forefront. Not only must the autotuning framework compare different possible multigrid cycle shapes against each other, but it also needs the ability to compare tuned cycles against both direct and (non-multigrid) iterative methods. We address this problem by using an accuracy metric for measuring the effectiveness of tuned cycle shapes and making comparisons over all algorithmic types based on this common yardstick. In our results, we find that the flexibility to trade performance versus accuracy at all levels of recursive computation enables us to achieve excellent performance on a variety of platforms compared to algorithmically static implementations of multigrid.

Our implementation uses PetaBricks, an implicitly parallel programming language where algorithmic choices are exposed in the language. The PetaBricks compiler uses these choices to analyze, autotune, and verify the PetaBricks program. These language features, most notably the autotuner, were key in enabling our implementation to be clear, correct, and fast.

1. INTRODUCTION

While multigrid is currently one of the most popular techniques for efficiently solving partial differential equations over a grid, it has become clear that restricting ourselves to a single technique in any problem domain is rarely the optimal strategy. It is often the case that we want to choose between different algorithms based on some characteristics of the input. For example, we may use the input's magnitude as the criteria in a factoring algorithm or the input's length in a sorting algorithm. The optimal cutoff is almost always dependent on underlying machine properties, and it is the goal of autotuning packages such as FFTW [7, 8], ATLAS [17, 18], and OSKI [16] to discover which situations warrant the application of each available technique.

In some cases, tuning algorithmic choice could simply mean choosing the appropriate top-level technique during the initial function invocation; however, for many problems including multigrid, it is better to be able to utilize multiple techniques within a single function call or solve. For example, in the C++ Standard Template Library's sort routine, the algorithm switches from using the divide-and-conquer $O(n \log n)$ merge sort to $O(n^2)$ insertion sort once the working array size falls below a set cutoff. In multigrid, an analogous strategy might switch from recursive multigrid calls to a direct method such as Cholesky factorization and triangular solve once the problem size falls below a threshold.

This paper analyzes the optimizations of algorithmic choice in multigrid. When confronted with the problem of training the autotuner to choose between a recursive multigrid call and a call to an iterative or direct solver, one quickly realizes that no comparison between methods can be fair without considering the relative accuracies of each. Indeed, we found that in some cases sacrificing accuracy at lower levels of recursion has little impact on the accuracy of the final result, while in other cases improving accuracy at a lower level reduces the number of (more expensive) iterations needed at a higher level.

In this paper we describe a novel dynamic programming strategy that allows us to make fair comparisons between various iterative, recursive, and direct methods, resulting in an efficient, tuned algorithm for user-specified convergence criteria. The resulting algorithms can be visualized as tuned multigrid cycles that apply targeted computational power to meet the accuracy requirements of the user. Our methodology does not tune cycle shapes by manipulating the shapes directly; it instead categorizes algorithms based on the accuracy of the results produced, allowing it to compare *all types* of algorithms (direct, iterative, and recursive) and make tuning decisions based on that common yardstick. Additionally, our tuning algorithm has the flexibility of utilizing different accuracy constraints for various components within a single algorithm, allowing the autotuner to independently trade performance and accuracy at each level of multigrid recursion.

This work on multigrid was developed using the PetaBricks programming language [2]. PetaBricks is an implicitly parallel programming language where algorithmic choice is a first class construct, to help programmers express and tune algorithmic choices and cutoffs such as these to obtain the fastest combination of algorithms to solve a problem. While traditional compiler optimizations can be successful at optimizing a single algorithm, when an algorithmic change is required to boost performance the burden is put on the programmer to incorporate the new algorithm. Programs written in PetaBricks can naturally describe multiple algorithms for solving a problem and how they can fit together. This information is used by the PetaBricks compiler and runtime to create and autotune an optimized multigrid algorithm.

1.1 Outline

We first describe in Section 2 the algorithmic choices available in multigrid and detail the dynamic programming approach to autotuning for accuracy and performance. Section 3 describes the PetaBricks language and implementation of the compiler and autotuning system that makes tuning over algorithmic choice possible. Section 4 presents experimental results. Finally, Sections 5, 6 and 7 describe related work, future work, and conclusions.

1.2 Contributions

We make the following contributions:

- We introduce an autotuner that can tune over algorithmic choice in multigrid problems.
- We describe how an accuracy metric can be used to make reasonable comparisons between direct, iterative, and recursive algorithms in a multigrid setting for the purposes of autotuning.
- We show how the use of dynamic programming can help us efficiently build tuned multigrid algorithms that combine methods with varying levels of accuracy while providing that a final target accuracy is met.
- We demonstrate that the performance of our tuned multigrid algorithms is superior to more basic reference approaches.
- We show that optimally tuned multigrid algorithms can be dependent on machine architecture, demonstrating the utility of a portable solution.

2. AUTOTUNING MULTIGRID

Although multigrid is a versatile technique that can be used to solve many different types of problems, we will use the 2D Poisson's equation as an example and benchmark to guide our discussion. The techniques presented here are generalizable to higher dimensions and the broader set of multigrid problems.

Poisson's equation is a partial differential equation that describes many processes in physics, electrostatics, fluid dynamics, and various other engineering disciplines. The continuous and discrete versions are

$$\bigtriangledown^2 \phi = f \quad \text{and} \quad Tx = b, \tag{1}$$

where T, x, and b are the finite difference discretizations of the Laplace operator, ϕ, and f, respectively.

To build an autotuned multigrid solver for Poisson's equation, we consider the use of three basic algorithmic building blocks: one direct (band Cholesky factorization through LAPACK's DPBSV routine), one iterative (Red-Black Successive Over Relaxation), and one recursive (multigrid). The table below shows the computational complexity of using any single algorithm to compute a solution.

Algorithm	Direct	SOR	Multigrid
Complexity	n^2 (N^4)	$n^{1.5}$ (N^3)	n (N^2)

From left to right, each of the methods has a larger overhead, but yields a better asymptotic serial complexity [6]. N is the size of the grid on a side, and $n = N^2$ is the number of cells in the grid.

2.1 Algorithmic choice in multigrid

Multigrid is a recursive algorithm that uses the solution to a coarser grid resolution as part of the algorithm. We will first address tuning symmetric "V-type" cycles. An extension to full multigrid will be presented in Section 2.4.

For simplicity, we assume all inputs are of size $N = 2^k + 1$ for some positive integer k. Let x be the initial state of the grid, and b be the right hand side of Equation (1).

```
MULTIGRID-V-SIMPLE($x, b$)
1: if $N = 3$ then
2:     Solve directly
3: else
4:     Relax using some iterative method
5:     Compute the residual and restrict to half resolution
6:     Recursively call MULTIGRID-V-SIMPLE on coarser grid
7:     Interpolate result and add correction term to current
       solution
8:     Relax using some iterative method
9: end if
```

It is at the recursive call on line 6 that our autotuning compiler can make a choice of whether to continue making recursive calls to multigrid or take a shortcut by using the direct solver or one of the iterative solvers at the current resolution. Figure 1 shows these possible paths of the multigrid algorithm.

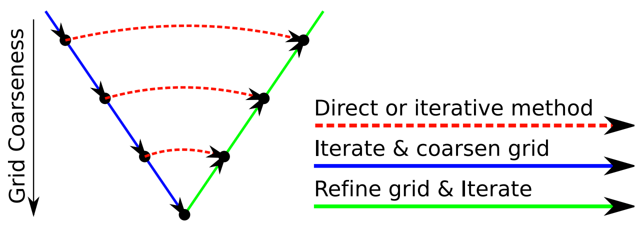


Figure 1: Simplified illustration of choices in the multigrid algorithm. The diagonal arrows represent the recursive case, while the dotted horizontal arrows represent the shortcut case where a direct or iterative solution may be substituted. Depending on the desired level of accuracy a different choice may be optimal at each decision point. This figure does not illustrate the autotuner's capability of using multiple iterations at different levels of recursion; it shows a single iteration at each level.

The idea of choice can be implemented by defining a top level function `MULTIGRID-V`, which makes calls to either the direct, iterative, or recursive solution. The function `RECURSE` implements the recursive solution.

```
MULTIGRID-V($x, b$)
1: either
2:     Solve directly
3:     Use an iterative method
4:     Call RECURSE for some number of iterations
5: end either
```

```
RECURSE($x, b$)
1: if $N = 3$ then
2:     Solve directly
3: else
4:     Relax using some iterative method
5:     Compute the residual and restrict to half resolution
6:     On the coarser grid, call MULTIGRID-V
7:     Interpolate result and add correction term to current
       solution
8:     Relax using some iterative method
9: end if
```

Making the choice in line 1 of `MULTIGRID-V` has two implications. First, the time to complete the algorithm is choice dependent. Second, the accuracy of the result is also dependent on choice since the various methods have different abilities to reduce error (depending on parameters such as number of iterations or weights). To make a fair comparison between choices, we must take both performance and accuracy of each choice into account. To this end, during the tuning process, we keep track of not just a single optimal algorithm at every recursion level, but a *set* of such optimal algorithms for varying levels of desired accuracy.

2.2 Full dynamic programming solution

We will first describe a full dynamic programming solution to handling variable accuracy, then restrict it to a discrete set of accuracies. We define an algorithm's *accuracy level* to be the ratio between the error norm of its input x_{in} versus the error norm of its output x_{out} compared to the optimal solution x_{opt}:

$$\frac{||x_{in} - x_{opt}||_2}{||x_{out} - x_{opt}||_2}.$$

We choose this ratio instead of its reciprocal so that a higher accuracy level is better, which is more intuitive. In order to measure the accuracy level of a potential tuned algorithm, we assume we have access to representative training data so that the accuracy level of our algorithms during tuning closely reflects their accuracy level during use.

Let level k refer to an input size of $N = 2^k + 1$. Suppose that for level $k-1$, we have solved for some set A_{k-1} of optimal algorithms, where optimality is defined such that no optimal algorithm is dominated by any other algorithm in both accuracy and compute time.

In order to construct the optimal set A_k, we try substituting all algorithms in A_{k-1} for step 6 of `RECURSE`. We also try varying parameters in the other steps of the algorithm, including the choice of iterative methods and the number of iterations (possibly zero) in steps 4 and 8 of `RECURSE` and steps 3 and 4 of `MULTIGRID-V`.

Trying all of these possibilities will yield many algorithms that can be plotted as in Figure 2(a) according to their accuracy and compute time. The optimal algorithms we add to A_k are the dominant ones designated by square markers.

The reason to remember algorithms of multiple accuracies for use in step 6 of `RECURSE` is that it may be better to use a less accurate, fast algorithm and then iterate multiple times, rather than use a more accurate, slow algorithm. Note that even if we use a direct solver in step 6, the interpolation in step 7 will invariably introduce error at the higher resolution.

2.3 Discrete dynamic programming solution

Since the optimal set of tuned algorithms can grow to be very large, the PetaBricks autotuner offers an approximate version of the above solution. Instead of remembering the full optimal set A_k, the compiler remembers the fastest al-

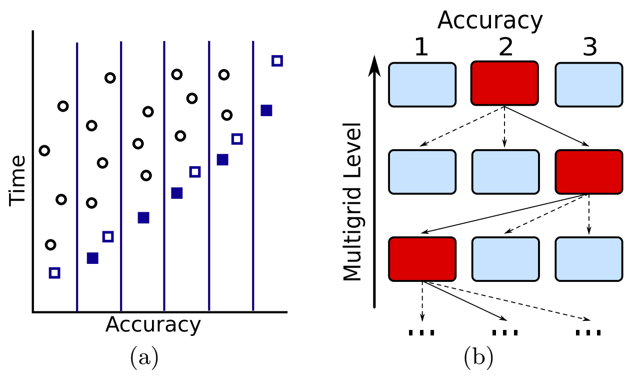


Figure 2: (a) Possible algorithmic choices with optimal set designated by squares (both hollow and solid). The choices designated by solid squares are the ones remembered by the PetaBricks compiler, being the fastest algorithms better than each accuracy cutoff line. (b) Choices across different accuracies in multigrid. At each level, the autotuner picks the best algorithm one level down to make a recursive call. The path highlighted in red is an example of a possible path for accuracy level p_2

gorithm yielding an accuracy of at least p_i for each p_i in some set $\{p_1, p_2, \ldots, p_m\}$. The vertical lines in Figure 2(a) indicate the discrete accuracy levels p_i, and the optimal algorithms (designated by solid squares) are the ones remembered by PetaBricks. Each highlighted algorithm is associated with a function MULTIGRID-V$_i$, which achieves accuracy p_i on all input sizes.

Due to restricted time and computational resources, to further narrow the search space, we only use SOR as the iteration function since we found experimentally that it performed better than weighted Jacobi on our particular training data for similar computation cost per iteration. In MULTIGRID-V$_i$, we fix the weight parameter of SOR to ω_{opt}, the optimal value for the 2D discrete Poisson's equation with fixed boundaries [6]. In RECURSE$_i$, we fix SOR's weight parameter to 1.15 (chosen by experimentation to be a good parameter when used in multigrid). We also fix the number of iterations of SOR in steps 4 and 8 in RECURSE$_i$ to one. As more powerful computational resources become available over time, the restrictions on the algorithmic search space presented here may be relaxed to find a more optimal solution.

The resulting accuracy-aware Poisson solver is a family of functions, where i is the accuracy parameter:

```
MULTIGRID-V$_i(x, b)$
1: **either**
2:     Solve directly
3:     Iterate using SOR$_{\omega_{\text{opt}}}$ until accuracy $p_i$ is achieved
4:     For some $j$, iterate with RECURSE$_j$ until accuracy $p_i$
       is achieved
5: **end either**
```

```
RECURSE$_i(x, b)$
1: **if** $N = 3$ **then**
2:     Solve directly
3: **else**
4:     Compute one iteration of SOR$_{1.15}$
5:     Compute the residual and restrict to half resolution
6:     On the coarser grid, call MULTIGRID-V$_i$
7:     Interpolate result and add correction term to current
       solution
8:     Compute one iteration of SOR$_{1.15}$
9: **end if**
```

The autotuning process determines what choices to make in MULTIGRID-V$_i$ for each i and for each input size. Since the optimal choice for any single accuracy for an input of size $2^k + 1$ depends on the optimal algorithms for *all* accuracies for inputs of size $2^{k-1}+1$, the PetaBricks autotuner tunes all accuracies at a given level before moving to a higher level. In this way, the autotuner builds optimal algorithms for every specified accuracy level and for each input size up to a user specified maximum, making use of the tuned sub-algorithms as it goes.

The final set of multigrid algorithms produced by the autotuner can be visualized as in Figure 2(b). Each of the versions has the flexibility to choose any of the other versions during its recursive calls, and the optimal path may switch between accuracies many times as we recurse down towards either the base case or a shortcut case.

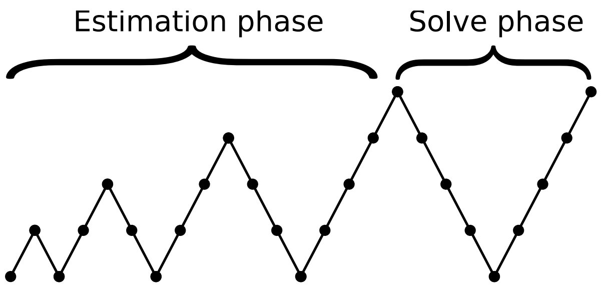


Figure 3: Conceptual breakdown of full multigrid into an estimation phase and a solve phase. The estimation phase can be thought of as just a recursive call to full multigrid up to a coarser grid resolution. We make use of this recursive structure, in addition to our autotuned "V-type" multigrid cycles, in constructing tuned full multigrid cycles.

2.4 Extension to Autotuning Full Multigrid

Full multigrid methods have been shown to exhibit better convergence behavior than traditional symmetric cycle shapes such as the V and W cycles by utilizing an estimation phase before the solve phase (see Figure 3). The estimation phase of the full multigrid algorithm can be thought of as just a recursive call to itself at a coarser grid resolution. We extend the autotuning ideas presented thus far to leverage this structure and produce autotuned full multigrid cycles.

The following simplified code for ESTIMATE and FULL-MULTIGRID illustrates how to construct an autotuned full multigrid cycle.

```
ESTIMATE$_i(x, b)$
1: Compute residual and restrict to half resolution
2: Call FULL-MULTIGRID$_i$ on restricted problem
3: Interpolate result and add correction to $x$
```

```
FULL-MULTIGRID$_i(x, b)$
1: **either**
2:     Solve directly
3:     For some $j$, compute estimate by calling ESTIMATE$_j(x, b)$,
   then **either**:
4:         Iterate using SOR$_{\omega_{\text{opt}}}$ until accuracy $p_i$ is achieved
5:         For some $k$, iterate with RECURSE$_k$ until accuracy
           $p_i$ is achieved
6: **end either**
```

Here we take advantage of the discrete dynamic programming analogue presented in Section 2.3 where we maintain only finite sets of optimized functions FULL-MULTIGRID$_j$ and MULTIGRID-V$_k$ to use in recursive calls. In FULL-MULTIGRID$_i$, there are three choices: the first is just a direct solve (line 2), while the latter two choices (lines 4 and 5) are similar to those given in MULTIGRID-V$_i$ except an estimate is first calculated and then used as a starting point for iteration. Note that this structure is descriptive enough to include the standard full multigrid V or W cycle shapes, just as the MULTIGRID-V$_i$ algorithm can produce standard regular V or W cycles.

The parameters j and k in FULL-MULTIGRID can be chosen independently, providing a great deal of flexibility in the construction of the optimized full multigrid cycle shape. In cases where the user does not require much accuracy in the final output, it may make sense to invest more heavily in the estimation phase, while in cases where very high precision is needed, a high precision estimate may not be as helpful as most of the computation would be done in relaxations at the highest resolution. Indeed, we found patterns of this type during our experiments.

2.5 Limitations

It should be clear that the algorithms produced by the autotuner are not meant to be optimal in any theoretical sense. Because of the compromises made in the name of efficiency, the resulting autotuning algorithm merely strives to discover near-optimal algorithms from within the restricted space of cycle shapes reachable during the search. There are many cycle shapes that fall outside the space of searched algorithms; for example, our approach does not check algorithms that utilize different choices in succession at the same recursion depth instead of choosing a single choice and iterating. Future work may examine the extent to which this restriction impacts performance.

Additionally, the scalar accuracy metric is an imperfect measure of the effectiveness of a multigrid cycle. Each cycle may have different effects on the various error modes (frequencies) of the current guess, all of which would be impossible to capture in a single number. Future work may expand the notion of an "optimal" set of sub-algorithms to include separate classes of algorithms that work best to reduce different types of error. Though such an approach could lead to a better final tuned algorithm, this extension would obviously make the auto-tuning process more complex.

We will demonstrate in Section 4 that although our methodology is not exhaustive, it can be quite descriptive, discovering cycle shapes that are both unconventional and efficient. That section will present actual cycle shapes produced by our multigrid autotuner and show their performance compared to less sophisticated heuristics. We will first describe the PetaBricks language and autotuning compiler in further detail.

3. PETABRICKS LANGUAGE

A key element that made our approach to multigrid possible was the PetaBricks programming language [2]. PetaBricks is a new implicitly parallel programming language in which algorithmic choice is a first class language construct. PetaBricks programs describe many possible ways to solve a problem and how they fit together. The PetaBricks compiler and runtime use these choices to autotune the program in order to find an optimal hybrid algorithm. Our implementation was written in the PetaBricks language, and we use the PetaBricks autotuner to tune our algorithms. For more information about the PetaBricks language and compiler see our prior work [2]; the following summary is included for background.

3.1 PetaBricks Language Design

The main goal of the PetaBricks language was to expose algorithmic choice to the compiler in order to empower the compiler to perform autotuning over aspects of the program not normally available to it. PetaBricks is an implicitly parallel language, where the compiler automatically parallelizes PetaBricks programs.

The PetaBricks language is built around two major constructs, *transforms* and *rule*s. The *transform*, analogous to a function, defines an algorithm that can be called from other transforms or invoked from the command line. The header for a transform defines *to*, *from*, and *through* arguments, which represent inputs, outputs, and intermediate data used within the transform. The size in each dimension of these arguments is expressed symbolically in terms of free variables, the values of which must be determined by the PetaBricks runtime.

The user encodes choice by defining multiple *rule*s in each transform. Each rule computes a region of data in order to make progress towards a final goal state. Rules can have different granularities and intermediate state. The compiler is required to find a sequence of rule applications that will compute all outputs of the program. Rules have explicit dependencies, allowing automatic parallelization and automatic detection and handling of corner cases by the compiler. The rule header references *to* and *from* regions which are the inputs and outputs for the rule. Free variables in these regions can be set by the compiler allowing a rule to be applied repeatedly in order to compute a larger data region. The body of a rule consists of `C++`-like code to perform the actual work.

3.2 PetaBricks Implementation

The PetaBricks implementation consists of three components: a source-to-source compiler from the PetaBricks language to C++, an autotuning system and choice framework to find optimal choices and set parameters, and a runtime library used by the generated code.

3.2.1 PetaBricks Compiler

The PetaBricks compiler works using three main phases. In the first phase, *applicable regions* (regions where each rule can legally be applied) are calculated for each possible choice using an inference system. Next, the applicable regions are aggregated together into *choice grids*. The choice grid divides each matrix into rectilinear regions where uniform sets of rules may legally be applied. Finally, a *choice dependency graph* is constructed and analyzed. The choice dependency graph consists of edges between symbolic regions in the choice grids. Each edge is annotated with the set of choices that require that edge, a direction of the data dependency, and an offset between rule centers for that dependency. The output code is generated from this choice dependency graph.

PetaBricks code generation has two modes. In the default mode, choices and information for autotuning are embedded in the output code. This binary can then be dynamically tuned, generating an optimized configuration file; subsequent runs can then use the saved configuration file. In the second mode, a previously tuned configuration file is applied statically during code generation. The second mode is included since the C++ compiler can make the final code incrementally more efficient when the choices are fixed.

3.2.2 Autotuning System and Choice Framework

The autotuner uses the *choice dependency graph* encoded in the compiled application. This choice dependency graph is also used by the parallel scheduler. This choice dependency graph contains the choices for computing each region and also encodes the implications of different choices on dependencies.

The intuition of the autotuning algorithm is that we take a bottom-up approach to tuning. To simplify autotuning, we assume that the optimal solution to smaller sub-problems is independent of the larger problem. In this way we build algorithms incrementally, starting on small inputs and working up to larger inputs.

The autotuner builds a multi-level algorithm. Each level consists of a range of input sizes and a corresponding algorithm and set of parameters. Rules that recursively invoke themselves result in algorithmic compositions. In the spirit of a genetic tuner, a population of candidate algorithms is maintained. This population is seeded with all single-algorithm implementations. The autotuner starts with a small training input and on each iteration doubles the size of the input. At each step, each algorithm in the population is tested. New algorithm candidates are generated by adding levels to the fastest members of the population. Finally, slower candidates in the population are dropped until the population is below a maximum size threshold. Since the best algorithms from the previous input size are used to generate candidates for the next input size, optimal algorithms are iteratively built from the bottom up.

In addition to tuning algorithm selection, PetaBricks uses an n-ary search tuning algorithm to optimize additional parameters such as parallel-sequential cutoff points for individual algorithms, iteration orders, block sizes (for data parallel rules), data layout, as well as user specified tunable parameters.

All choices are represented in a flat configuration space. Dependencies between these configurable parameters are exported to the autotuner so that the autotuner can choose a sensible order to tune different parameters. The autotuner starts by tuning the leaves of the graph and works its way up. If there are cycles in the dependency graph, it tunes all parameters in the cycle in parallel, with progressively larger input sizes. Finally, it repeats the entire training process, using the previous iteration as a starting point, a small number of times to better optimize the result.

3.2.3 Runtime Library

The runtime library is primarily responsible for managing parallelism, data, and configuration. It includes a runtime scheduler as well as code responsible for reading, writing, and managing inputs, outputs, and configurations. The runtime scheduler dynamically schedules tasks (that have their input dependencies satisfied) across processors to distribute work. The scheduler attempts to maximize locality using a greedy algorithm that schedules tasks in a depth-first search order. Following the approach taken by Cilk [9], we distribute work with thread-private deques and a task stealing protocol.

4. RESULTS

In this section, we present the results of the PetaBricks autotuner when optimizing our multigrid algorithm on three parallel architectures designed for a variety of purposes: Intel Xeon E7340 server processor, AMD Opteron 2356 Barcelona server processor, and the Sun Fire T200 Niagara low power, high throughput server processor. These machines provided architectural diversity, allowing us to show not only how autotuned multigrid cycles outperform reference multigrid algorithms, but also how the shape of optimal autotuned cycles can be dependent on the underlying machine architecture.

To the best of our knowledge, there are no standard data distributions currently in wide use for benchmarking multigrid solvers, so it was not clear what the best choice is for training and benchmarking our tuned solvers. We decided to use matrices with entries drawn from two different random distributions: 1) uniform over $[-2^{32}, 2^{32}]$ (unbiased), and 2) the same distribution shifted in the positive direction by 2^{31} (biased). The random entries were used to generate right-hand sides (b in Equation 1) and boundary conditions (boundaries of x) for the problem. We also experimented with specifying a finite number of random point sources/sinks in the right-hand side, but since the observed results were similar to those found with the unbiased random distribution, we did not include them in interest of space. If one wishes to obtain tuned multigrid cycles for a different input distribution, the training should be done using that data distribution.

4.1 Autotuned multigrid cycle shapes

During the tuning process for the `MULTIGRID-V`$_i$ algorithm presented in Section 2.3, the autotuner first computes the number of iterations needed for the SOR and `RECURSE`$_j$ choices before determining which is the fastest option to attain accuracy p_i for each input size. Representative training data is required to make this determination. Once the number of required iterations of each choice is known, the autotuner times each choice and chooses the fastest option.

Figures 4(a) and 4(b) show the traces of calls to the tuned `MULTIGRID-V`$_4$ algorithms for unbiased and biased uniform random inputs of size $N = 4097$, on the Intel machine. As you can see, the algorithm utilizes multiple accuracy levels throughout the call stack. In general, whenever greater accuracy is required by our tuned algorithm, it is achieved through some repetition of optimal substructures determined by the dynamic programming method. This may be easier to visualize by examining the resulting tuned cycles corresponding to the autotuned multigrid calls.

Figures 5(a) and 5(b) show some tuned "V-type" cycles created by the autotuner for unbiased and biased uniform random inputs of size $N = 2049$ on the AMD Opteron machine. The cycles are shown using standard multigrid notation with some extensions: The path of the algorithm progresses from left to right through time. As the path moves down, it represents a restriction to a coarser resolution, while paths up represent interpolations. Dots represent red-black SOR relaxations, solid horizontal arrows represent calls to the direct solver, and dashed horizontal arrows represent calls to the iterative solver.

As seen in the figure, a different cycle shape is used depending on what level of accuracy is required by the user. Cycles shown are tuned to produce final accuracy levels of $10, 10^3, 10^5$, and 10^7. The leverage of optimal subproblems is clearly seen in the common patterns that appear across cycles. Note that in Figure 5(b), the call to the direct solver in cycle i) occurs at level 4, while for the other three cycles, the direct call occurs at level 5. This is an example of the autotuner trading accuracy for performance while accounting for the accuracy requirements of the user.

Figures 5(c) and 5(d) show autotuned full multigrid cycles for unbiased and biased uniform random inputs of size $N = 2049$ on the AMD Opteron machine. Although similar

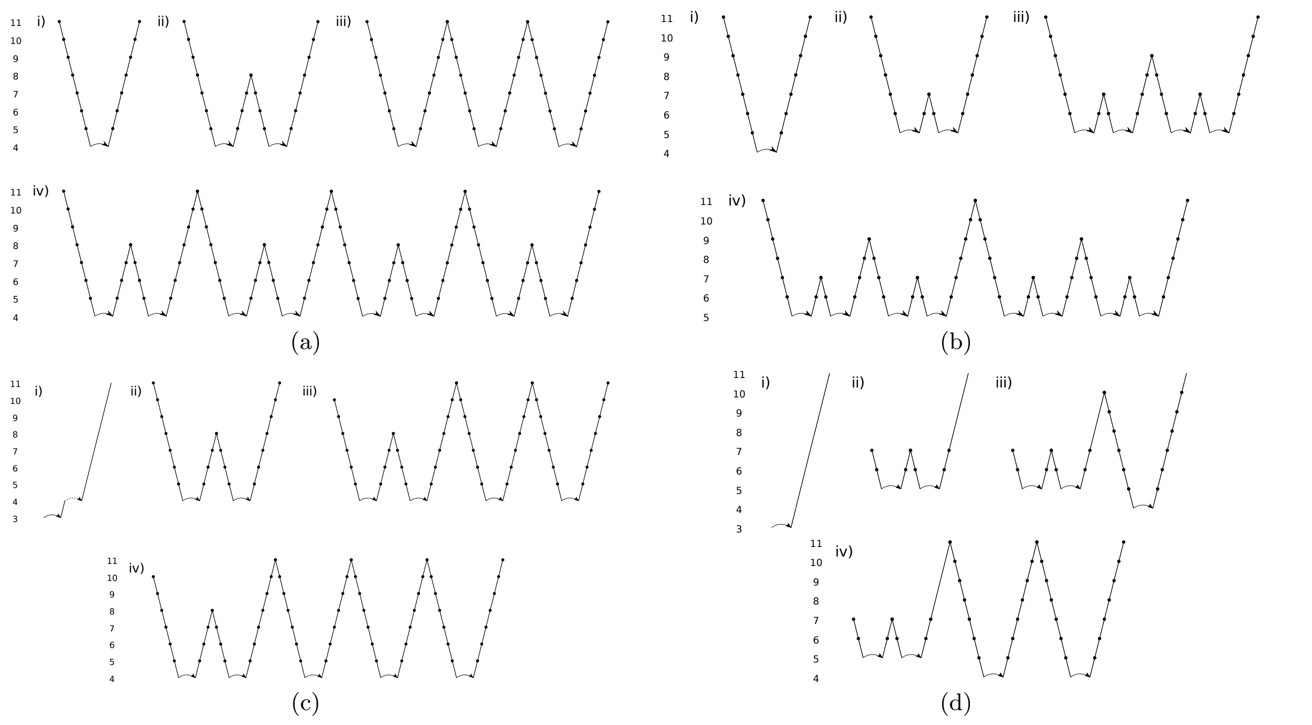


Figure 5: Optimized multigrid V (a and b) and full multigrid (c and d) cycles created by the autotuner for solving the 2D Poisson's equation on an input if size $N = 2049$. Subfigures a) and c) were trained on unbiased uniform random data, while b) and d) were trained on biased uniform random data. Cycles i), ii), iii), and iv), correspond to algorithms that yield accuracy levels of $10, 10^3, 10^5$, and 10^7, respectively. The solid arrows at the bottom of the cycles represent shortcut calls to the direct solver, while the dashed arrow in c)-i) represents an iterative solve using SOR. The dots present in the cycle represent single relaxations. Note that some paths in the full multigrid cycles skip relaxations while moving to a higher grid resolution. The recursion level is displayed on the left, where the size of the grid at level k is $2^k + 1$.

substructures are shared between these cycles and the "V-type" cycles in 5(a) and 5(b), some of the expensive higher resolution relaxations are avoided by allowing work to occur at the coarser grids during the estimation phase of the full multigrid algorithm. The tuned full multigrid cycle in Figure 5(d)-iv) shows how the additional flexibility of using an estimation phase can dramatically alter the tuned cycle shape when compared to Figure 5(b)-iv).

It is important to realize that the call stacks in Figure 4 and the cycle shapes in Figure 5 are all dependent on the specific situation at hand. They would all likely change were the autotuner run on other architectures, using different training data, or solving other multigrid problems. The flexibility to adapt to any of these changing variables by tuning over algorithmic choice is the autotuner's greatest strength.

4.2 Performance

This section will provide data showing the performance of our tuned multigrid Poisson's equation solver versus reference algorithms and heuristics. Test data was produced from the same distributions used for training described in Section 4. Section 4.2.1 describes performance of the autotuned `MULTIGRID-V` algorithm, and Section 4.2.2 describes the performance of the autotuned `FULL-MULTIGRID` algorithm.

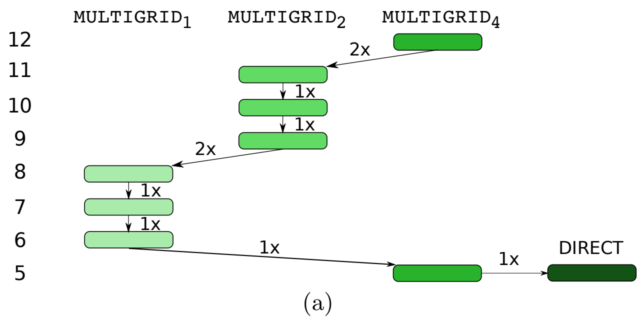
4.2.1 Autotuned multigrid V algorithm

To demonstrate the effectiveness of our dynamic programming methodology, we compare the autotuned `MULTIGRID-V` algorithm against more basic approaches to solving the 2D Poisson's equation to an accuracy of 10^9, including several multigrid variations. Results presented in the section were collected on the Intel Xeon server testbed machine.

Figure 6 shows the performance of our autotuned multigrid algorithm for accuracy 10^9 on unbiased uniform random inputs of different sizes. The autotuned algorithm uses internal accuracy levels of $\{10, 10^3, 10^5, 10^7, 10^9\}$ during its recursive calls. The figure compares the autotuned algorithm with the direct solver, iterated calls to SOR, and iterated calls to `MULTIGRID-V-SIMPLE` (labeled Multigrid). Each of the iterative methods is run until an accuracy of at least 10^9 is achieved.

As to be expected, the autotuned algorithm outperforms all of the simple algorithms shown in Figure 6. At sizes greater than $N = 65$, the autotuned algorithm performs slightly better than `MULTIGRID-V-SIMPLE` because it utilizes a more complex tuned strategy.

Figure 7 compares the tuned algorithm with various heuristics more complex than `MULTIGRID-V-SIMPLE`. The training data used in this graph was drawn from the biased uniform distribution. Strategy 10^9 refers to requiring an accuracy of 10^9 at each recursive level of multigrid until the base case



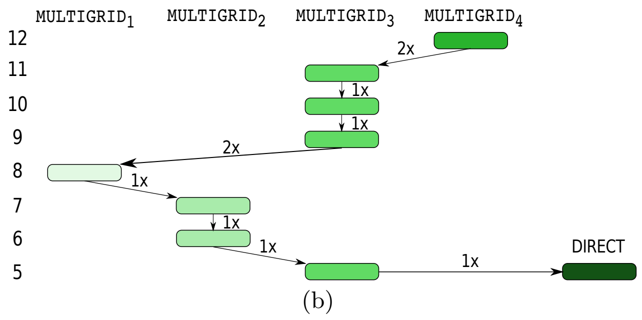


Figure 4: Call stacks generated by calls to autotuned MULTIGRID-V$_4$ for a) unbiased and b) biased random inputs of size $N = 4097$ on an Intel Xeon server. Discrete accuracies used during autotuning were $(p_i)_{i=1..5} = (10, 10^3, 10^5, 10^7, 10^9)$. The recursion level is displayed on the left, where the size of the grid at level k is $2^k + 1$. Note that each arrow connecting to a lower recursion level actually represents a call to RECURSE$_i$, which handles grid coarsening, followed by a call to MULTIGRID-V$_i$.

direct method is called at $N = 65$. Strategies of the form $10^x/10^9$ refer to requiring an accuracy of 10^x at each recursive level below that of the input size, which requires an accuracy of 10^9. Thus, all strategies presented result in a final accuracy of 10^9; they differ only in what accuracies are required at lower recursion levels. All heuristic strategies call the direct method for smaller input sizes whenever it is more efficient to meet the accuracy requirement.

The lines in Figure 7 are somewhat close together and difficult to see on the logarithmic time scale, so Figure 8 presents the same data but showing the ratio of times taken versus the autotuned algorithm. We can more clearly see in this figure that as the input size increases, the most efficient heuristic changes from Strategy $10^1/10^9$ to $10^3/10^9$ to $10^5/10^9$. The autotuner does better than just choosing the best from among these heuristics, since it can also tune the desired accuracy at each recursion level independently, allowing greater flexibility. This figure highlights the complexity of finding an optimal strategy and showcases the utility of an autotuner that can efficiently find this optimum.

Another big advantage of using PetaBricks for autotuning is that it allows a single program to be optimized for both sequential performance and parallel performance. We have observed our autotuner make different choices when running on different numbers of cores. Figure 9 shows the speedup achieved by our tuned MULTIGRID-V algorithms on our Intel

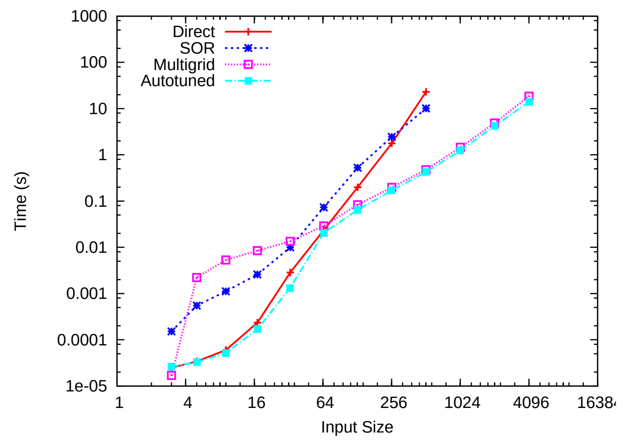


Figure 6: Performance for algorithms to solve Poisson's equation on unbiased uniform random data up to an accuracy of 10^9 using 8 cores. The basic direct and SOR algorithms as well as the standard V-cycle multigrid algorithm are all compared to our tuned multigrid algorithm. The iterated SOR algorithm uses the corresponding optimal weight ω_{opt} for each of the different input sizes

testbed machine.

4.2.2 Autotuned full multigrid algorithm

In order to evaluate the performance of our autotuned MULTIGRID-V and FULL-MULTIGRID algorithms on multiple architectures, we ran them for problem sizes up to $N = 4097$ (up to 2049 on the Sun Niagara) for target accuracy levels of 10^5 and 10^9 alongside two reference algorithms: an iterated V cycle and a full multigrid algorithm. The reference V cycle algorithm runs standard V cycles until the accuracy target is reached, while the reference full multigrid algorithm runs a standard full multigrid cycle (as in Figure 3), then standard V cycles until the accuracy target is reached.

We chose these two reference algorithms since they are generally deemed good starting points for those interested in implementing multigrid for the first time. Since they are easy to understand and commonly implemented, we felt they were a reasonable point of reference for our results. From these starting points, performance tweaks can be manually applied to tailor the solver to each user's specific application domain. The goal of our autotuner is to discover and make these tweaks automatically.

Figure 10 shows the performance of both reference and autotuned multigrid algorithms for unbiased uniform random data relative to the reference iterated V-cycle algorithm on all three testbed machines. Figure 11 shows similar comparisons for biased uniform random data. The relative time (lower is better) to compute the solution up to an accuracy level of 10^5 is plotted against problem size.

On all three architectures, we see that the autotuned algorithms provide an improvement over the reference algorithms' performances. There is an especially marked difference for small problem sizes due to the autotuned algorithms' use of the direct solve without incurring the overhead of recursion. Speedups relative to the reference full multigrid algorithm are also observed at higher problem sizes:

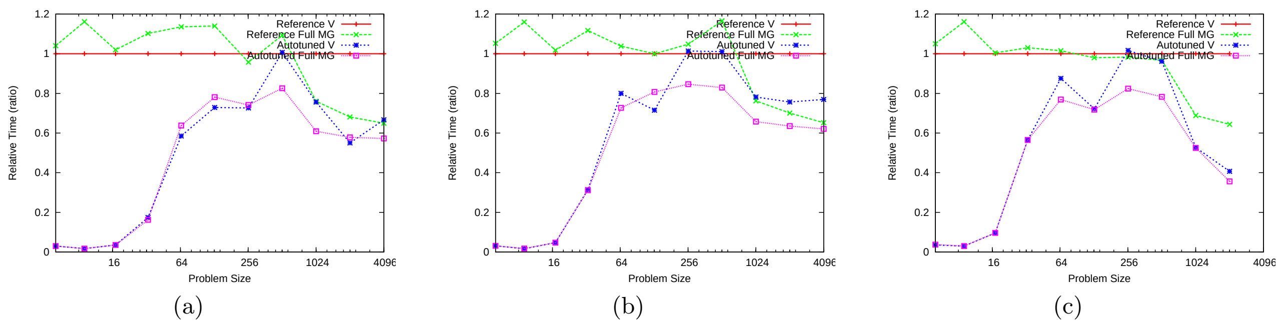


Figure 10: Relative performance of multigrid algorithms versus reference V cycle algorithm for solving the 2D Poisson's equation on unbiased, uniform random data to an accuracy level of 10^5 on a) Intel Harpertown, b) AMD Barcelona, and c) Sun Niagara.

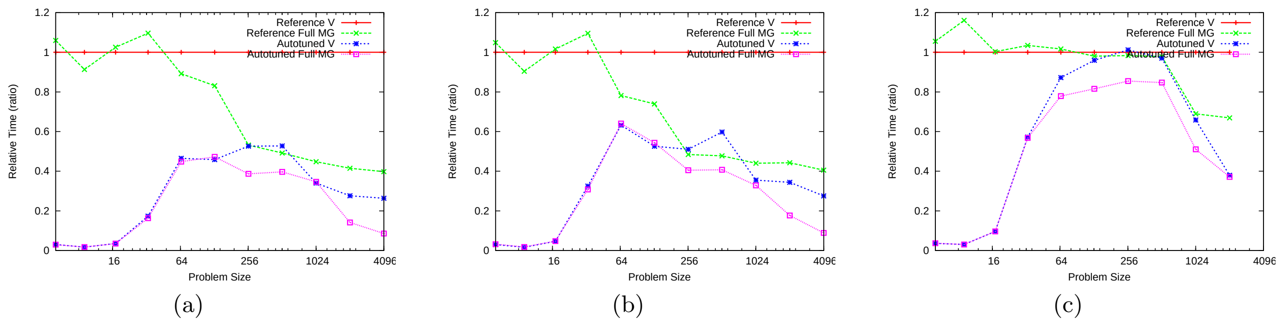


Figure 11: Relative performance of multigrid algorithms versus reference V cycle algorithm for solving the 2D Poisson's equation on biased uniform random data to an accuracy level of 10^5 on a) Intel Harpertown, b) AMD Barcelona, and c) Sun Niagara.

e.g., for problem size $N = 2049$, we observed speedups of 1.2x, 1.1x, and 1.8x on the unbiased uniform test inputs, and 2.9x, 2.5x, and 1.8x on the biased uniform test inputs for the Intel, AMD, and Sun machines, respectively.

Figures 12 and 13 show similar performance comparisons, except to an accuracy level of 10^9. The autotuner had a more difficult time beating the reference full multigrid algorithm when training for both high accuracy and large size (greater than $N = 257$). For sizes greater than 257, autotuned performance is essentially tied with the reference full multigrid algorithm on the Intel and AMD machines, while improvements were still possible on the Sun machine. For input size $N = 2049$, a speedup of 1.9x relative to the reference full multigrid algorithm was observed on the Niagara for both input distributions. We suspect that performance gains are more difficult to achieve when solving for both high accuracy and size in some part due to a greater percentage of compute time being spent on unavoidable relaxations at the finest grid resolution.

4.3 Effect of Architecture on Autotuning

Multicore architectures have drastically increased the processor design space resulting in a large variance in processors currently on the market. Such variance significantly hinders porting efforts of performance critical code.

Figure 14 shows the different optimized cycles chosen by the autotuner on the three testbed architectures. Though all cycles were tuned to yield the same accuracy level of 10^5, the autotuner found a different optimized cycle shape on each architecture. These differences take advantage of the specific characteristics of each machine. For example, the AMD and Sun machines recurse down to a coarse grid level of 2^4 versus 2^5 on the Intel machine. The AMD and Sun's cycles appear to make up for the reduced accuracy of the coarser direct solve by doing more relaxations at medium grid resolutions (levels 9 and 10).

We found that the performance of tuned multigrid cycles can be quite sensitive to where the autotuning is performed in some cases. For example, the use of the autotuned full multigrid cycle for unbiased uniform inputs of size $N = 2049$ trained on the Sun Niagara but run on the Intel Xeon results in a 29% slowdown compared to the natively trained algorithm. Likewise, using the cycle trained on the Xeon results in a 79% slowdown compared to using the natively trained cycle on the Niagara.

5. RELATED WORK

Some multigrid solvers using algorithmic choice have been presented in the past. SuperSolvers [3] is not an autotuner but rather a system for designing composite algorithms that leverage multiple algorithmic choices to solve sparse linear systems reliably. Our approach differs by the use of tuning algorithmic choice at different levels of the multigrid hierarchy and the use of tuned subproblems during recursion.

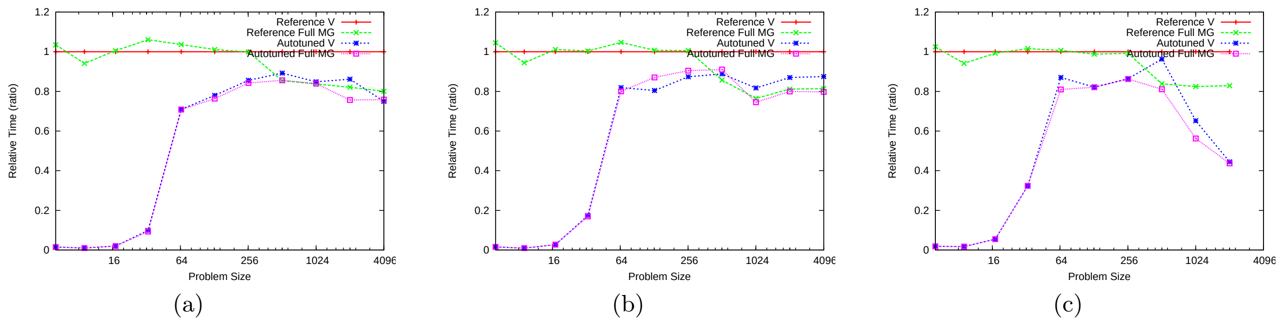


Figure 12: Relative performance of multigrid algorithms versus reference V cycle algorithm for solving the 2D Poisson's equation on unbiased, uniform random data to an accuracy level of 10^9 on a) Intel Harpertown, b) AMD Barcelona, and c) Sun Niagara.

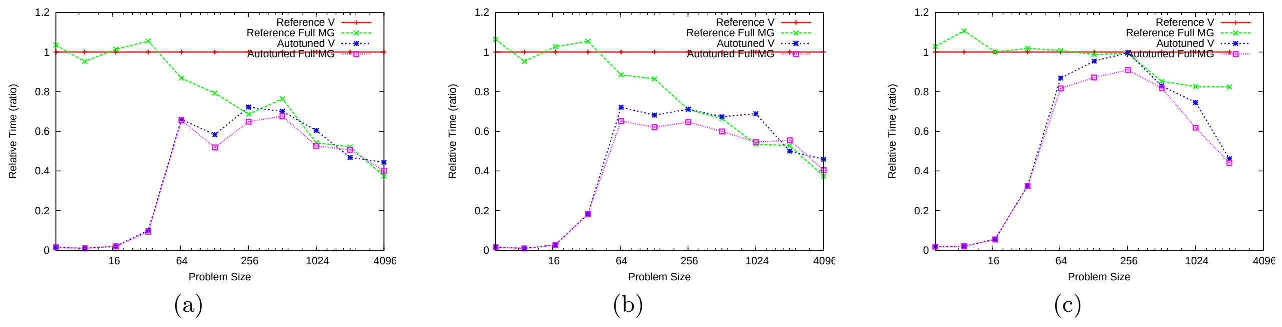


Figure 13: Relative performance of multigrid algorithms versus reference V cycle algorithm for solving the 2D Poisson's equation on biased, uniform random data to an accuracy level of 10^9 on a) Intel Harpertown, b) AMD Barcelona, and c) Sun Niagara.

Unfortunately, no direct performance comparison was possible for this paper due to the lack of availability of source code.

Cache-aware implementations of multigrid have also been developed. In [15], [14], and [11] optimizations improve cache utilization by reducing capacity and conflict misses during linear relaxation and inter-grid transfers. An autotuner was presented in [5] to automatically search the space of cache and memory optimizations for the relaxation step over a variety of hardware architectures. The optimizations presented in these related works are for the most part orthogonal to the approach taken in this paper. There is no reason lower-level optimizations cannot be combined with algorithmic tuning at the level of cycle shape.

A number of empirical autotuning frameworks have been developed for building efficient, portable libraries in other specific domains. PHiPAC [4] is an autotuning system for dense matrix multiply, generating portable C code and search scripts to tune for specific systems. ATLAS [17, 18] utilizes empirical autotuning to produce a cache-contained matrix multiply, which is then used in larger matrix computations in BLAS and LAPACK. FFTW [7, 8] uses empirical autotuning to combine solvers for FFTs. Other autotuning systems include SPARSITY [10] for sparse matrix computations, SPIRAL [13] for digital signal processing, UHFFT [1] for FFT on multicore systems, OSKI [16] for sparse matrix kernels, and an autotuning framework for optimizing parallel sorting algorithms by Olszewski and Voss [12].

6. FUTURE WORK

An interesting direction we wish take this work is in the domain of tuning multi-level algorithms across distributed memory systems. The problem of discovering the best data layout and communications pattern for such a solver is very complex.

One specific problem this framework may help address is when to migrate data between machines. For example, we may want to use a smaller subset of machines once the problem is sufficiently small to reduce the surface area to volume ratio of each machine's working set. Doing so reduces the communications overhead of relaxations, but incurs the cost of the data transfer. We wish to extend the ideas presented here to produce "optimal" algorithms parameterized not just on size and accuracy, but also on data layout. The dynamic programming search could then take data transfers into account when comparing the costs of utilizing various "optimal" sub-algorithms, each with their own associated layouts.

Another direction we plan to explore is the use of dynamic tuning where an algorithm has the ability to adapt during execution based on some features of the intermediate state. Such flexibility would allow the autotuned algorithm to classify inputs and intermediate states into different distribution classes and then switch between tuned versions of itself, providing better performance across a broader range

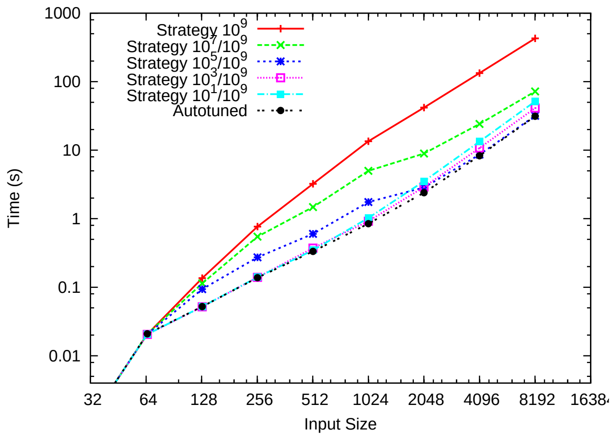


Figure 7: Performance for algorithms to solve Poisson's equation up to an accuracy of 10^9 using 8 cores. The autotuned multigrid algorithm is presented alongside various possible heuristics. The graph omits sizes less than $N = 65$ since all cases call the direct method for those inputs. To see the trends more clearly, Figure 8 shows the same data as this figure, but as ratios of times taken versus the autotuned algorithm.

of inputs. For example, we may want to switch between cycle shapes during execution depending on the dominant error frequencies observed in the residual.

7. CONCLUSIONS

It has become nearly impossible to tune individual algorithms by hand for portable performance, and multigrid algorithms are no exception. No single choice of parameters can yield the best possible result for different user environments, which include problem, machine architecture, and accuracy requirements. The high performance computing community has always known that in many problem domains, the best sequential algorithm is different from the best parallel algorithm. Varying problem size and data sets will also require different algorithms. Currently there is no viable way for incorporating all these algorithmic choices into a single multigrid program to produce portable programs with consistently high performance.

In this paper we introduced a novel dynamic programming approach to autotuning multigrid algorithms. Our approach tunes with an awareness of accuracy that allows fair comparison between various direct, iterative, and recursive algorithmic types such that optimal solutions are built from the bottom up. We demonstrated that the resulting tuned cycles achieve excellent performance compared to algorithmically static implementations of multigrid.

8. ACKNOWLEDGEMENTS

This work is partially supported by NSF Award CCF-0832997 and an award from the Gigascale Systems Research Center. We wish to thank the UC Berkeley EECS department for generously letting us use one of their machines for benchmarking. Finally, we would like to thank the anonymous reviewers for their constructive feedback.

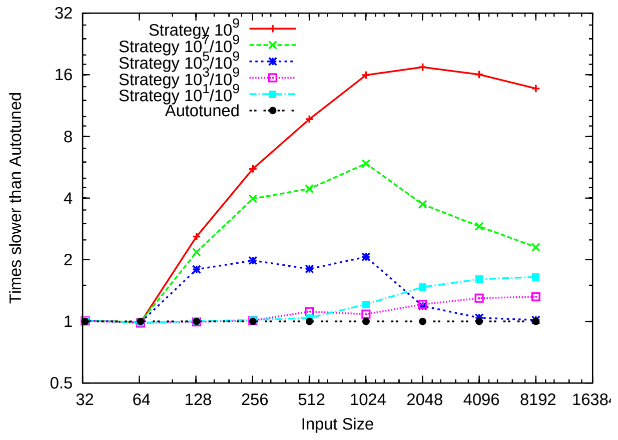


Figure 8: Speedup of tuned algorithm compared to various simple heuristics to solve Poisson's equation up to an accuracy of 10^9 using 8 cores. The data presented in this graph is the same as in Figure 7 except that the ratio of time taken versus the autotuned algorithm is plotted. Notice that as the problem size increases, the higher accuracy heuristics become more favored since they require fewer iterations at high resolution grid sizes.

9. REFERENCES

[1] A. Ali, L. Johnsson, and J. Subhlok. Scheduling FFT computation on smp and multicore systems. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 293–301, 2007.

[2] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. Petabricks: A language and compiler for algorithmic choice. In *PLDI '09: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[3] S. Bhowmick, P. Raghavan, and K. Teranishi. A combinatorial scheme for developing efficient composite solvers. In *ICCS '02: Proceedings of the International Conference on Computational Science-Part II*, pages 325–334, London, UK, 2002. Springer-Verlag.

[4] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ansi c coding methodology. In *ICS '97: Proceedings of the 11th international conference on Supercomputing*, pages 340–347, 1997.

[5] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[6] J. W. Demmel. *Applied Numerical Linear Algebra*. SIAM, August 1997.

[7] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*,

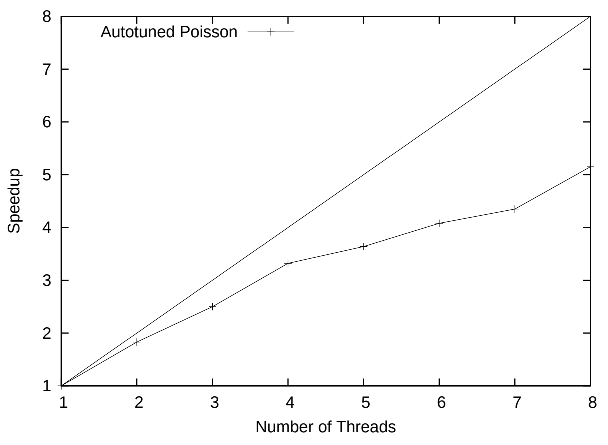


Figure 9: Parallel scalability. Speedup as more worker threads are added. Run on an 8 core (2 processor × 4 core) x86_64 Intel Xeon System.

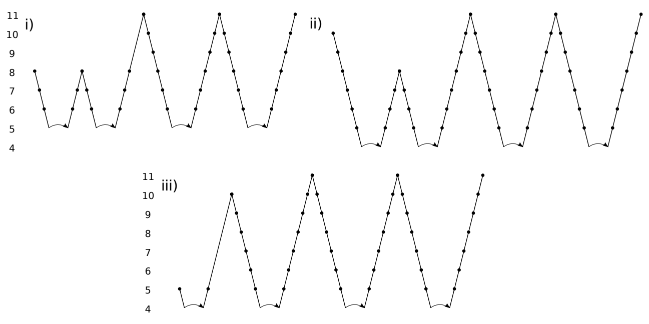


Figure 14: Comparison of tuned full multigrid cycles across machine architectures: i) Intel Harpertown, ii) AMD Barcelona, iii) Sun Niagara. All cycles solve the 2D Poisson's equation on unbiased uniform random input to an accuracy of 10^5 for an initial grid size of 2^{11}.

volume 3, pages 1381–1384. IEEE, 1998.

[8] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, February 2005. Invited paper, special issue on "Program Generation, Optimization, and Platform Adaptation".

[9] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, Jun 1998. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

[10] E. Im and K. Yelick. Optimizing sparse matrix computations for register reuse in SPARSITY. In *In Proceedings of the International Conference on Computational Science, volume 2073 of LNCS*, pages 127–136. Springer, 2001.

[11] M. Kowarschik and C. Weiss. Dimepack – a cache-optimized multigrid library. In *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2001), volume I*, pages 425–430. CSREA, CSREA Press, 2001.

[12] M. Olszewski and M. Voss. Install-time system for automatic generation of optimized parallel sorting algorithms. In *PDPTA*, pages 17–23, 2004.

[13] M. Puschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, A. G. Franz Franchetti, R. W. J. Yevgen Voronenko, Kang Chen, and N. Rizzolo. SPIRAL: Code generation for dsp transforms. In *Proceedings of the IEEE.*

[14] G. Rivera and C.-W. Tseng. Tiling optimizations for 3d scientific computations. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 32, Washington, DC, USA, 2000. IEEE Computer Society.

[15] S. Sellappa and S. Chatterjee. Cache-efficient multigrid algorithms. *Int. J. High Perform. Comput. Appl.*, 18(1):115–133, 2004.

[16] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proceedings of SciDAC 2005*, Journal of Physics: Conference Series, San Francisco, CA, USA, June 2005. Institute of Physics Publishing.

[17] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.

[18] R. C. Whaley and A. Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005.