An Install-Time System for the Automatic Generation of Optimized Parallel Sorting Algorithms

Marek Olszewski and Michael Voss
Contact Author: Michael Voss (voss@eecg.toronto.edu)
Phone: 416-946-8031, Fax: 416-971-2326
University of Toronto, Dept. of Electrical and Computer Engineering
10 King's College Road, Toronto, Ontario M5S 3G4 Canada

Abstract

Sorting is a fundamental algorithm used extensively in computer science as an intermediate step in many applications. The performance of sorting algorithms is heavily influenced by the type of data being sorted, and the machine being used. To assist in obtaining portable performance for sorting algorithms, we propose an install-time system for automatically constructing sequential and parallel sorts that are highly tuned for the target architecture. Our system has two steps: first a hybrid sequential divide-and-conquer sort is constructed and then this algorithm is parallelized using a shared work-queue model. To evaluate our system, we compare automatically generated sorting algorithms to sequential and parallel versions of the C++STL sort. The generated sorts are shown to be competitive with STL sort on sequential systems and to outperform the parallel STL sort on a 4 processor Xeon server.

Keywords: Parallel sorting algorithms, install-time optimization, Hyperthreading

1 Introduction

Sorting is a fundamental procedure in the modern world. It is used extensively in computer science as an intermediate step in many algorithms, and, in today's information oriented world, sorting facilitates easy and efficient retrieval of data. While much attention has already been devoted to sorting, larger data sets are opening doors for further improvement, especially in the rising domain of parallel computing.

As with most algorithms, the performance of sorting algorithms is heavily influenced by the type of data being sorted, and the machine being used. As such, there is potential for improving sorting times by automatically generating algorithms that are specific to particular data sets and architectures. This approach allows for maximum portability without compromise in performance. In similar work, researchers have created systems to automatically construct machine-specific versions of the BLAS routines [WPD01] and FFT libraries [FJ98].

In this paper, we propose an install-time system for automatically constructing optimized parallel sorting algorithms for small-scale symmetric multiprocessors (SMPs). This construction is performed in two steps. First, a high-performance sequential sort is composed by combining multiple sorting algorithms into a single *divide-and-conquer* algorithm. Next, a parallel sort is created by parallelizing this divide-and-conquer algorithm. In both steps, machine-specific decisions are made to reduce overheads and optimize performance.

This paper makes the following contributions: (1) we present a fully-automatic, install-time system for generating tuned sequential and parallel sorts for small-scale SMPs, (2) we show that our self-tuned sequential sorting algorithm is competitive with the C++STL sorts on 3 target systems, and (3) we show that our parallelized sorting algorithm outperforms parallelized versions of the C++STL sort.

2 Sorting Algorithms

2.1 Sequential Sorting

The problem of sequentially sorting data has been studied for decades. In The Art of Computer Programming V.3, D. Knuth [Knu98] does a comprehensive study of 25 such algorithms, discussing which are most appropriate for different sets of data structures, output requirements, and physical storage media. Sorting algorithms can be categorized into two sets: *comparison* and *non-comparison* based. Using a *decision tree model*, it is possible to prove that there exists a lower bound of $\Omega(nlogn)$ for comparison based sorting algorithms [CLRS01]. Insertion sort, quick sort and merge sort are examples of comparison sorts. Non-comparison sorts, such as bucket sort and radix sort, provide sorting in linear time, but require knowledge of the bounds of the input data. In this paper, we consider only the more general comparison-based sorts.

The simplest comparison algorithm is insertion sort. It has a worst case complexity of $O(n^2)$. *Divide-and-conquer* algorithms perform much better asymptotically than insertion sort. These algorithms take the task of sorting, and divide it into sub-problems. Then, with recursion, these subproblems are sorted using the same algorithm. Both quick sort and merge sort are examples of divide-and-conquer algorithms.

Much of the work done on both sequential and parallel sorting algorithms has focused on finding algorithms with small asymptotic bounds. However, asymptotic bounds only express the order of growth of the execution time of an algorithm, not the hidden constant factors and lower-order terms. For example even though insertion sort is $O(n^2)$, its low constant factor allows it to sort faster than merge sort and quick sort for small data sets.

Furthermore, merge sort and quick sort are of divide-and-conquer type, which means that they can benefit greatly by calling insertion sort when they encounter small data sets after many successive recursive calls. We call sorts that make use of multiple sorting algorithms *hybrid sorting* algorithms. The `std::sort` and `std::stable_sort` functions in the C++ Standard Template Library are often implemented as complex hybrid algorithms. For example, the Gnu versions of these two algorithms use predetermined cutoff points to switch between heap sort, quick sort, merge sort and insertion sort algorithms.

2.2 Parallel Sorting

There have been a number of parallel sorting algorithms proposed specifically for parallel machines. Ideally, a parallel sorting algorithm could achieve an asymptotic complexity of $O(\frac{nlogn}{p})$, where p is the number of processors. Given $p = n$, such a parallel sort would be $O(logn)$. A number of parallel sorts, such as Column sort [Lei85, CCW01], have demonstrated this bound. However, these explicit parallel sorts often have high hidden constants and therefore only perform well with large lists on large numbers of processors. At low numbers of processors or small list sizes, parallelized versions of sequential sorts often outperform these explicitly parallel sorts [AJR+01]. In this paper, we target the small-scale SMP systems that are becoming increasingly commonplace, and therefore focus on shared-memory parallel sorts adapted from sequential sorts, as described in the next section.

2.3 Parallelized Sequential Sorts

A second method of programming parallel sorting algorithms is to parallelize divide-and-conquer algorithms. These algorithms may be parallelized by having subproblems enqueued into a shared work-queue. As threads need work, they remove tasks from the work-queue, and add new tasks to the queue as new subproblems are defined. This shared work-queue provides natural load balancing. Since there is overhead associated with the enqueuing and dequeuing tasks, there is a cutoff point beyond which it is more efficient for a thread to simply execute the sub tasks that it generates instead of adding them to the shared work queue. The optimal point of the change from enqueuing work to continuing locally is data set and architecture specific.

3 An Adaptive Sort

An overview of our proposed system is shown in Figure 1. It has two basic phases: (1) construction of the optimized sequential sort decision tree and (2) selection of a work-sharing cutoff point.

3.1 Sequential Sort Decision Trees

There exist a number of very good sequential sorting algorithms. Because many of these are divide-and-conquer algorithms, it is possible to combine them into one hybrid sorting algorithm. In order to achieve maximum performance, the points at which switching between different algorithms takes place should be chosen such that the most appropriate sorting algorithm be used at each recursive call. These switching points can be empirically determined by running experiments on the targeted architecture with sample data that best matches the expected data in both type and entropy. Once the sampled performance is examined, a decision tree can be constructed. The optimized sorting algorithm will then consult this decision tree during its sorting processes to select the sort that is best to use at each recursive step.

As shown in Figure 1, our system begins by sampling the performance of hybrid sorts by sorting many different random data sets, each varying in size. These hybrid sorts are generated by randomly choosing between the sorting algorithms described in Table 1 at each recursive step. During these runs, the time that it takes to complete the work at each recursive level is measured. This time includes all of

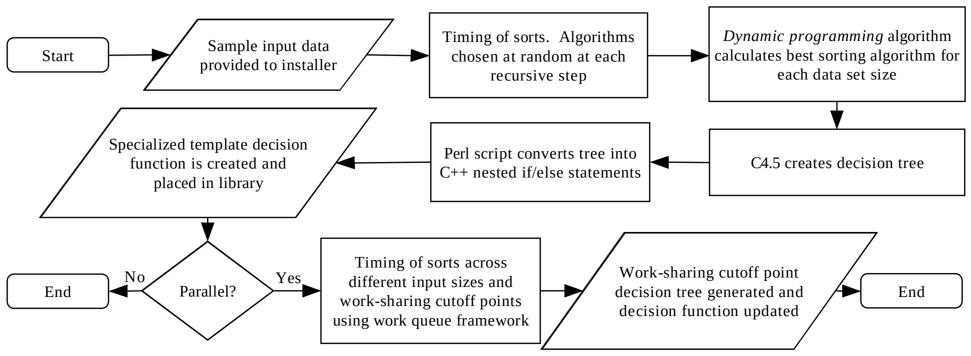

Start
Sample input data provided to installer
Timing of sorts. Algorithms chosen at random at each recursive step
Dynamic programming algorithm calculates best sorting algorithm for each data set size
C4.5 creates decision tree
Perl script converts tree into C++ nested if/else statements
Specialized template decision function is created and placed in library
Parallel?
No
End
Yes
Timing of sorts across different input sizes and work-sharing cutoff points using work queue framework
Work-sharing cutoff point decision tree generated and decision function updated
End

Figure 1: Install-time optimization process

the work performed in the current call but not any of the recursive calls made from it. For example for merge sort, this time would include the merging of the sublists at this step, and for quick sort this would include the time to shuffle the data elements around the pivot point.

In order to ensure an even distribution of timing data for all input array sizes, training is done on 5 different brackets of random input sizes, with a maximum size of 10 million elements. To make the tests feasible, insertion sort–due to its $O(n^2)$ complexity–has to be ignored for arrays with more than one thousand elements. Furthermore, partial sorts are carried out in the largest three brackets (arrays larger than one hundred thousand) preventing many repetitive timing runs for small arrays. The point at which sorting is stopped is varied from bracket to bracket.

Next, for each list size, the best choice to make at each recursive step is determined. To do this, a *dynamic programming* approach [CLRS01] is used to find the best sorting algorithm for each data set size, given that the best choice was carried out in subsequent recursive calls. Once a list of optimal choices is generated, data-mining software (c4.5 [Qui93]) is used to generate a decision tree. c4.5 generalizes the data obtained from the sampling, yielding small decision trees in place of the large mappings between list sizes and best algorithms.

The tree obtained in this step is converted into nested if/else C++ statements. Using template specialization, the decision tree is embedded in a sorting algorithm for the specific data type used in the data gathering process. This generated sorting algorithm is the *optimized sequential Adaptive Sort* for this data type on this architecture, and does not contain the high-overhead decision making and instrumentation code used during the sampling phase.

3.2 Work-Sharing Cutoff Point

The sequential optimized sorting algorithm is a divide-and-conquer sorting algorithm that calls a decision function to select an algorithm at each step, rather than call a fixed sorting algorithm. In the parallel sorting algorithm, the decision function is augmented to make two decisions: (1) what sorting algorithm should be used for this data set size and (2) whether the resulting sub-task(s) should be placed in the shared work-queue for execution by any thread or executed immediately by the current thread.

The first decision is made by referring to the sequential decision tree. This decision tree need not change, since our parallel algorithm continues to use the same sequential algorithms [1].

There is overhead associated with adding and removing tasks from the shared work-queue. In order to ensure balanced work loads between threads while avoiding high overheads, an optimal cutoff point, at which all recursive calls continue within the current thread, has to be obtained. The second phase of our install-time system measures the time it takes to sort sample data at varying work-sharing cutoff points.

Figure 2 displays the effect of changing the cutoff point. It can be seen from the sharp increases in sorting time that there is a small region of acceptable cutoff points. Once enough data is sampled, it is possible to generate a second decision tree that can be used to decide on a optimal cutoff point for given input sizes. Our *optimized parallel sort* uses such a tree before the sorting process to determine the best cutoff point for the input array, which is then used in its decision function to determine whether subtasks should be enqueued or executed immediately by the calling thread.

[1]Some of the parallel versions of sequential sorts include synchronization constructs, but we have found in practice that the overhead of these constructs is minimal and does not change the decision tree found for the sequential algorithm.

Algorithm	Description
Insertion Sort	$O(n^2)$ but with small lower order terms. Efficient for small lists.
Merge Sort	$\Theta(nlgn)$. Subtasks are evenly divided but has higher lower-order terms than quick sort.
Quick Sort	$O(nlgn)$ on average, but is $O(n^2)$ worst-case. Has smaller lower-order terms than merge sort.
In-place Merge Sort	$\Theta(nlgn)$. Higher constant coefficient than merge sort, but can still be competitive, especially when sorting sets of size near an integral power of two.
Heap Sort	$\Theta(nlgn)$. Non-recursive $nlgn$ algorithm. Can do well on medium sized lists. Higher lower-order terms than quick sort.

Table 1: The sorting algorithms available to the optimized hybrid sort.

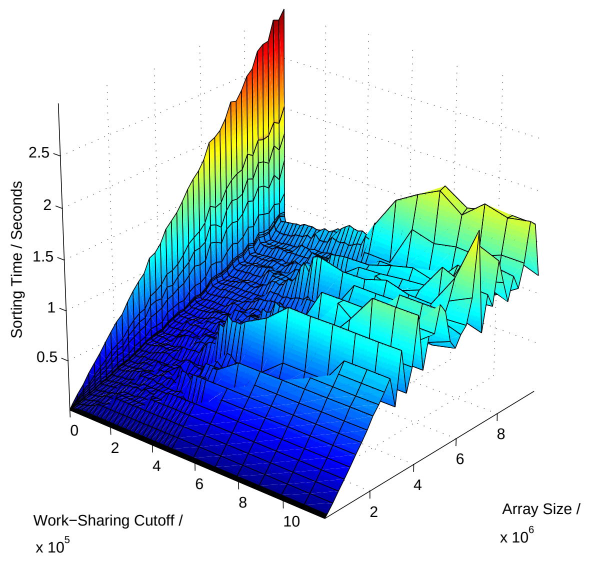


Figure 2: Times for work-sharing cutoff points.

4 Evaluation

4.1 System & Methodology

In the evaluation of our system, we perform both sequential and parallel experiments. Sequential tests were carried out on various architectures and operating systems. These include Linux 2.4.18 on an Intel Pentium IV XEON 1.6GHz processor, Linux 2.4.24 on an AMD Athlon XP 1700+ and SunOS 5.8 on a Sun Sparc workstation 600MHz. All parallel tests were carried out on a four processor Intel Pentium IV XEON 1.6GHz SMP with Hyperthreading support. A modified Linux 2.4.18 kernel was used to allow thread bindings.

Gnu G++ version 2.96 `std::sort` and `std::stable_sort` functions were used in the construction of parallel algorithms for comparison. Since these algorithms are quite complicated, it was not possible to easily employ the same work queue framework used with the self-tuning sort. Instead, seven merge sorts were used to incrementally divide the array by two, passing the result to new threads until 8 equally sized subproblems were created. These were then given to `std::sort` or `std::stable_sort`.

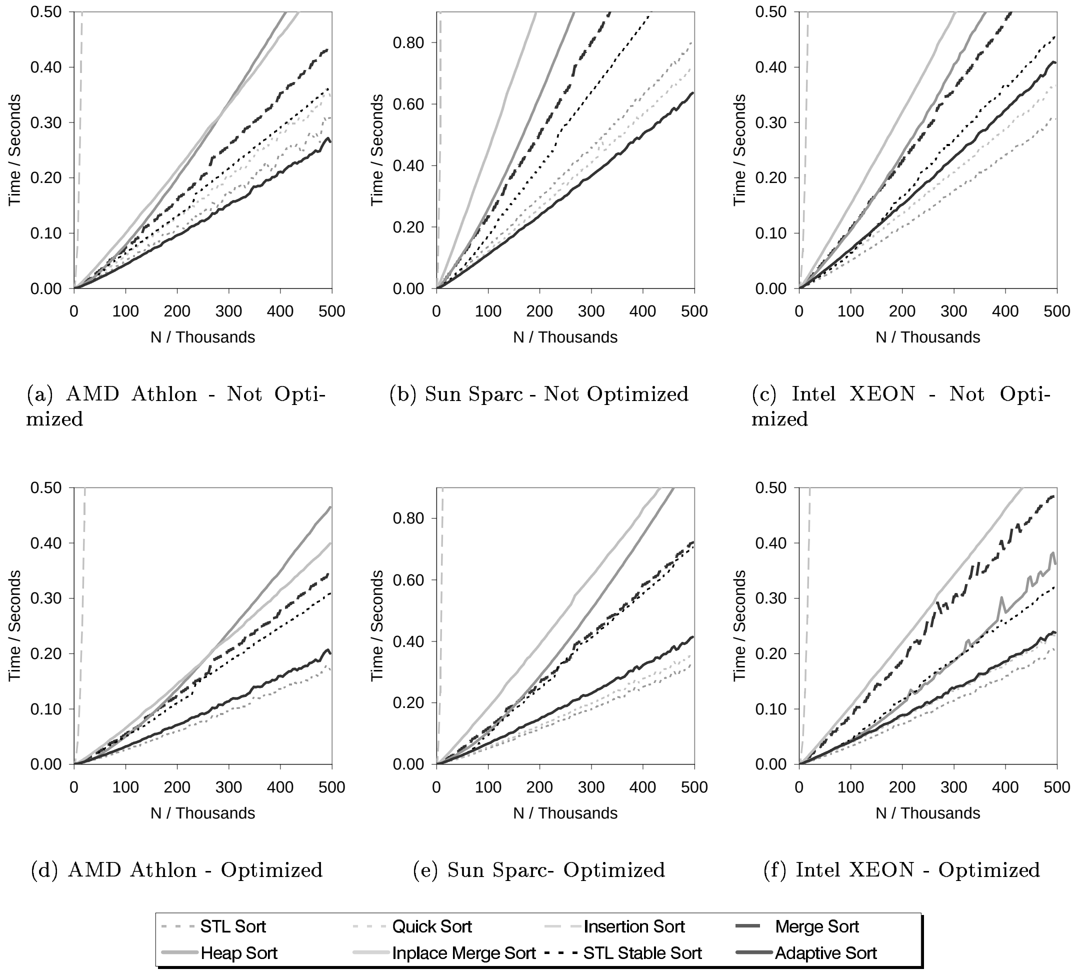
In cases where pre-sorted data was used, it was generated by inserting numbers, proportional to position, into an array that was filled with random data. These numbers ranged from 1 to `RAND_MAX` so that they spanned the same range as the random numbers.

4.2 Sequential Sort Results

Figures 3(a) - 3(f) present an evaluation of our optimized sequential sort compared to several other sequential sorting algorithms. All of the sorts presented in Figure 3 are templated, but due to space constraints we show only the results for doubles. We experimented with other data types as well, and found the trends to be the same. Since these sorts are templated, the algorithms are instantiated in the application code at compile-time. We therefore show both the case when the user is compiling with no optimization and with -O2.

Our install-time created sequential sort is labelled "Adaptive Sort" in Figure 3, and is competitive with STL sort on all platforms. In the non-optimized case, our automatically constructed hybrid sort is better than STL sort on two of the architectures. STL sort is not a simple hybrid sort, but contains highly-tuned and partially iterative versions of quick and merge sorts. This makes STL not only highly efficient, but also difficult to parallelize, as mentioned in Section 4.1. That our straightforward recursive hybrid sort, designed to be easily parallelized using a work-queue model, competes with STL on all sequential architectures is therefore quite impressive.

It is also interesting to note, that our Adaptive Sort is composed of the other non-hybrid sorts presented in Figure 3. However, of the non-hybrid sorts,



(a) AMD Athlon - Not Optimized

(b) Sun Sparc - Not Optimized

(c) Intel XEON - Not Optimized

(d) AMD Athlon - Optimized

(e) Sun Sparc- Optimized

(f) Intel XEON - Optimized

Figure 3: Comparison of Sequential Sorting Algorithms. Random data of type double

only quick sort is competitive with our automatically tuned sorting algorithm. This demonstrates that in most cases the combination of multiple sorts yields an improvement over an one single sorting algorithm.

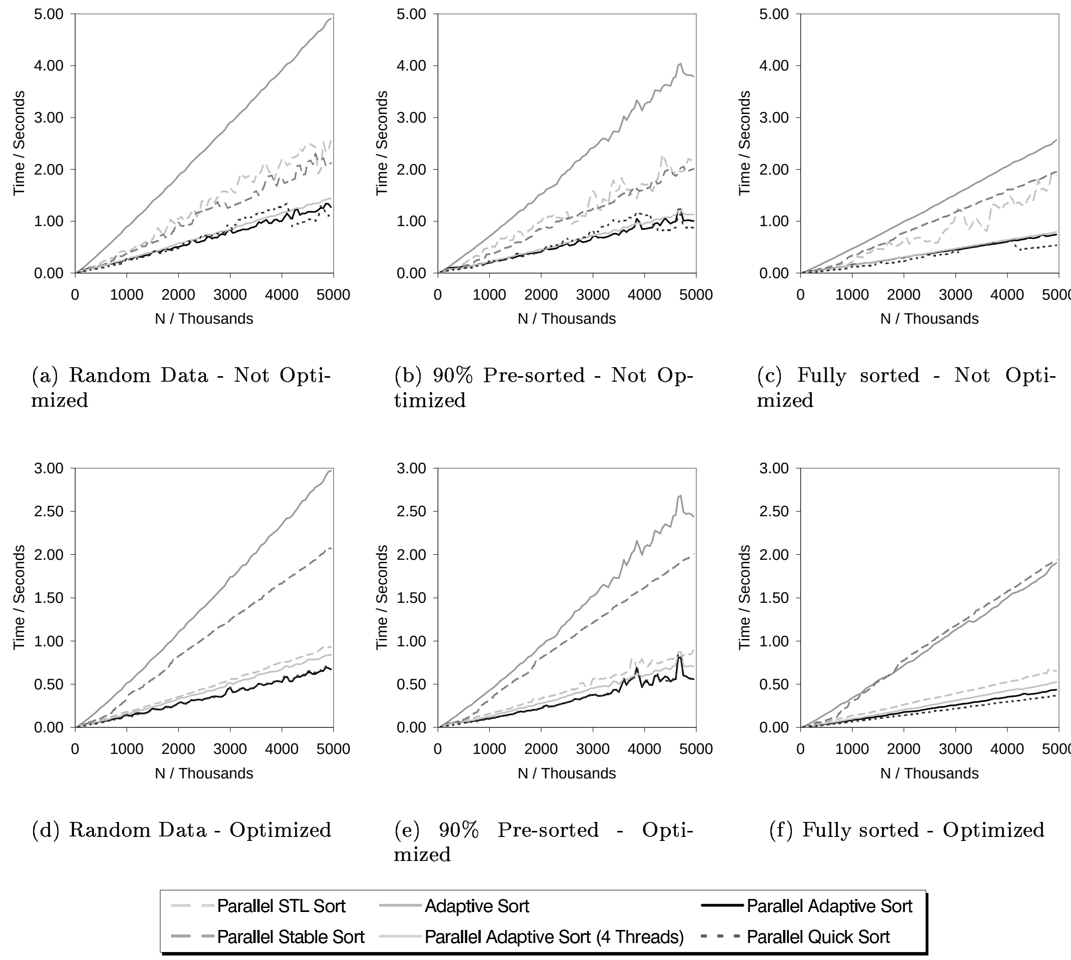
4.3 Parallel Sort Results

Our install-time system presented in Section 3 uses sequential sorts as the basis for its parallel algorithms. In this section, we evaluate the performance of the parallel sorting algorithm generated for our 4 processor Xeon server described in Section 4.1. The performance of the sequential Adaptive Sort on the Xeon server are shown in Figures 3(c) and 3(f). Figures 4(a) - 4(f) show the performance of the parallelized version of this Adaptive Sort.

Figure 4 show results only for doubles, although the trends in the data are the same for other primitive data types. In Figure 4, random, 90% pre-sorted and fully sorted data sets are used for both the non-optimized and optimized cases. In all the non-fully-sorted cases, the Parallel Adaptive Sort outperforms all others.

The scalability of our Adaptive Sort, shown in Figure 5, is excellent. A speedup greater than 4 can be noted in the optimized case when using 8 threads on the 4 physical processors of our Xeon. While a Hyperthreaded processor can execute multiple threads simultaneously, these threads share the functional units of the processor. In computationally-intense applications, this sharing of resources often leads to a degradation when multiple threads are used. Thus, a speedup greater than 4 on 4 physical processors is impressive.

In addition, the parallel STL sorts, which use an 8-way merge sort to call STL and Stable STL sorts, do not scale as well as our Adaptive Sort, as shown in Figure 4. The complicated nature of the STL sort



(a) Random Data - Not Optimized

(b) 90% Pre-sorted - Not Optimized

(c) Fully sorted - Not Optimized

(d) Random Data - Optimized

(e) 90% Pre-sorted - Optimized

(f) Fully sorted - Optimized

Figure 4: Comparison of Parallel Sorting Algorithms. All results are obtained using 8 threads except "Parallel Adaptive Sort (4 Threads)", for which 4 threads are used, and "Adaptive Sort", which is sequential.

does not allow a work-queue model to be easily incorporated, and therefore the parallel versions suffer from load imbalance.

The Parallel Quick Sort shown in Figure 4 is, however, able to employ a work-queue model. This algorithm uses the partially iterative implementation of quick sort used by the Gnu STL sort, and therefore represents a highly tuned algorithm. In addition, we hard-coded the work-sharing cutoff point found by our Parallel Adaptive Sort into Parallel Quick Sort. Without this cutoff point, the Parallel Quick Sort was an order of magnitude slower than our Parallel Adaptive Sort. Therefore, the performance of Parallel Quick Sort shown in Figure 4 is obtained by using machine-specific knowledge not usually available to a static algorithm. Even so, our Parallel Adaptive Sort is able to outperform this highly-tuned quick sort for most data points on non-fully-sorted data. Furthermore, this slight performance hit observed with fully-sorted data can be attributed to the fact that all the Adaptive Sorting algorithms were trained on random data only. Thus the Adaptive and Parallel Adaptive Sorts generated by our install-time system, are shown in Figures 3 and 4 to be highly efficient in comparison to other sequential and parallel sorts.

5 Related Work

There are several other systems that have used composable algorithms to construct a target-specific optimized algorithm at install-time. The ATLAS project [WPD01] samples matrix computations at install-time to construct a tuned version of the BLAS libraries for the target architecture. ATLAS monitors performance to create a cache-contained multiply that is used to construct the larger matrix com-

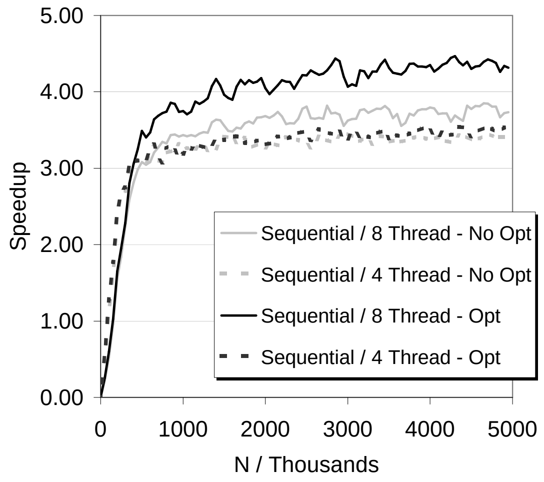


Figure 5: Parallel speedups for optimized and non-optimized versions of both the 8 and 4 threaded versions of the Adaptive Sort using random doubles.

putations provided by the BLAS library. In a similar vein, the FFTW library creates at install-time a plan for combining composable solvers for computing FFTs. FFTW determines and uses machine and compiler characteristics to develop this plan, allowing its FFT computations to perform well across architectures. Our system like ATLAS and FFTW, uses the idea that divide-and-conquer sorting algorithms can be composed from multiple algorithms, and finds the best sorting plan for a given architecture at install-time.

Most closely related to our work is the STAPL Adaptive Parallel C++library [AJR+01]. STAPL provides a range of parallelized template functions from the C++Standard Template Library, including sorts. Similar to our system, a decision tree for selecting an algorithm is created at install-time. However, STAPL does not compose a sorting plan, but instead picks a single sorting algorithm to be used to sort the entire data set based on the size of the data and the number of processors. Our system creates a hybrid algorithm that may switch sorting algorithms as sub-problems decrease in size. STAPL was not available at the time of our evaluation to make performance comparisons.

6 Conclusion

In this paper, we proposed an install-time system for automatically constructing sequential and parallel sorts that are highly tuned for a target architecture. Our system has two steps: first a hybrid sequential divide-and-conquer sort is constructed and then this algorithm is parallelized using a shared work-queue model. To evaluate our system in Section 4, we compared automatically generated sorting algorithms to sequential and parallel versions of the C++STL sort as well as other representative sorting algorithms. The generated sorts were shown to be competitive with STL sort on sequential systems and to outperform the parallel STL sort on a 4 processor Xeon server. On the 4-processor Hyperthreaded server, our Parallel Adaptive Sort was able to achieve a speedup of greater than 4 when using 8 threads. Our automated install-time system was therefore shown to compose highly efficient sorting algorithms for both sequential and parallel systems.

References

[AJR+01] Ping An, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase, Nathan Thomas, Nancy M. Amato, and Lawrence Rauchwerger. STAPL: A Standard Template Adaptive Parallel C++ Library. In *International Workshop on Advanced Compiler Technology*, Romania, 2001.

[CCW01] Geeta Chaudrhy, Thomas H. Cormen, and Leonard F. Wisinewski. Columnsort lives! an efficient out-of-core sorting program. In *SPAA: ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178, Crete Island, Greece, 2001.

[CLRS01] T. Cormen, C. Leiserson, R. Rivest, and Clifford Stein. *Introduction to Algorithms.* The MIT Press, 2 edition, 2001.

[FJ98] Matteo Frigo and Steven G. Johnson. FFTW: An Aadaptive Software Architecture for the FFT. In *Proceedings of ICASSP 3: International Conference on Acoustics, Speech and Signal Procesing*, Seattle, Washington, May 1998.

[Knu98] D. Knuth. *The Art of Computer Programming, Sorting and Searching*, volume 3. Addison-Wesley, Reading, MA, 2 edition, 1998.

[Lei85] Tom Leighton. Tight bounds in the complexity of parallel sorting. *IEEE Transactions on Computers*, 34:318–325, 1985.

[Qui93] J. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, 1 edition, 1993.

[WPD01] R. Clint Whaley, Antoine Petitet, and Jack Dongarra. Automated Empirical Optimization of Software and the ATLAS project. *Parallel Computing*, 27(1):3–35, 2001.