Automatic Generation of a Parallel Sorting Algorithm*

Brian A. Garber, Dan Hoeflinger, Xiaoming Li, María Jesús Garzarán, and David Padua
Department of C.S.
University of Illinois at Urbana-Champaign
{garber,hoefling, xli15, garzaran, padua}@cs.uiuc.edu

Abstract

In this paper, we discuss a library generator for parallel sorting routines that examines the input characteristics (and the parameters they affect) to select the best performing algorithm. Our preliminary experimental results show that the automatic generation of a distributed memory parallel sorting routine provides up to a four fold improvement over standard parallel algorithms with typical parameters. With the recent importance of multicore processors, we are extending this work to shared memory. This provides new challenges specific to multicore systems. However, with their increasing popularity, this extension becomes very valuable.

1 Introduction

Automatic Program Optimization is a difficult problem. Compilers are designed to deal with general purpose programs and many times they lack the semantic information necessary to optimize the source program. At the same time, computers are becoming increasingly complex. This makes the interactions between hardware and software difficult to understand, and hence difficult to determine the transformations and program parameter values that lead to the best performance on a given platform. As a result of this situation, library generators such as ATLAS [14], FFTW [3], and SPIRAL [11] have emerged. They have access to the domain specific semantic information that compilers lack and therefore can be more effective than compilers in generating fast code. To determine the best transformations for each target platform these generators make use of empirical search, which proceeds by executing different versions of the target code on the target machine. The version that obtains the best performance is selected as the library routine to be generated for that particular platform.

Recently, a library generator for sequential sorting routines was developed [9, 10] . In [9], empirical search is used to tune a set of sorting algorithms taking into account the fact that performance of sorting depends not only on the architectural features of the target machine, but also on input characteristics such as number of keys and standard deviation. During a training phase this sorting routine generator gathers information about the performance of different versions of sorting as a function of the characteristics of the input data. This information is used at runtime, when the input characteristics are known, to select the version of the code that is to be executed.

In our library generator, we are following an approach similar to that of [9, 10] to generate parallel sorting routines. Thus, in this work we evaluate how the input characteristics affect the performance of several sorting algorithms and develop new strategies for the tuning of the parallel sorting routines. We found that as was the case for sequential sorting, a single algorithm is not the best across the board for parallel sorting. Therefore our strategy, like in the case of the sequential sorting generator is to select at execution time an algorithm based on a table mapping input characteristics to algorithms computed offline.

We compare the preliminary performance of our generated routine with others like the Parallel Sorting using Regular Sampling $PSRS$ [12], and the Charm++ Radix Sort [8]. We found that our tuned algorithm runs significantly faster than PSRS or Charm++. Throughout this paper, we use standard deviation for the runtime algorithm selection. Entropy is a another metric to used measure the variation of digits in a key and was the one used in earlier work [9, 10].

In Section 2, we provide motivation for this research. We then briefly discuss our implementation of the sorting library generator in Section 3. Next, in Section 4, we provide some initial experimental results and compare our sorting library to two distributed memory sort implementations. In Section 5, we discuss similar projects having to do with distributed sorting and automatically tuned libraries. Finally we will present our conclusions in Section 6.

*This material is based upon supported by the National Science Foundation under Awards CCF 0702260 and CNS 0509432

2 Motivation

Unlike many dense numerical algorithms, sorting depends on factors other than the characteristics of the target machine. In [10], it is shown that the standard deviation as well as the number of keys to sort can significantly influence the performance of various sorting algorithms. Depending on the number of keys and standard deviation, different algorithms may perform best.

The problem of developing fast, adaptive sorting routines becomes even more complex with the addition of another orthogonal design space: parallel computers. There are additional factors that influence performance when multiple computers are involved such as the number of processors, the network connectivity, the latency, and the bandwidth.

This is illustrated in Figure 1, which shows the execution times of two different sorting routines (Parallel CC-Radix and one that we call CCSample Sort) when sorting sequences of 4M keys with the same average value and different values of standard deviation on four processors of a cluster known as Turing. The figure shows that for low values of standard deviation CCSample Sort runs faster, while for high values of standard deviation it is better to use Parallel CC-Radix Sort.

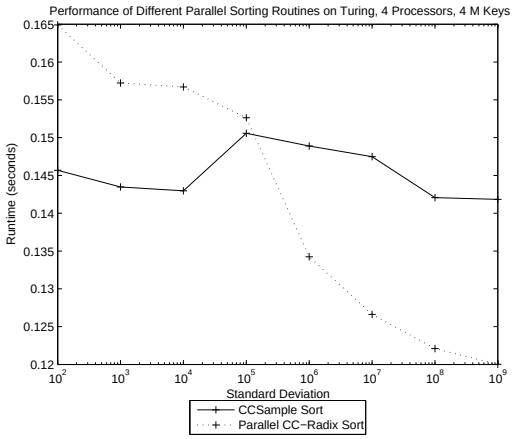

Figure 1: Performance of CCSample Sort and Parallel CC-Radix Sort when sorting 4M keys on Turing with 4 processors.

Figure 2 shows that within Parallel CC-Radix Sort, a significant improvement in runtime can be achieved by changing the values of the parameters based on the standard deviation. In this figure we show the benefit gained from a runtime selection of an optimal radix found by exhaustive search for each standard deviation, compared to using the a single radix for all standard deviations. In the figure, the best single radix choice was identified using exhaustive search as the one that gives the best average performance for all standard deviations. In Figure 2 the internal radix is that used within each processor and the external radix is that used to partition the data for distribution across processors. The figure shows that selecting only one pair for all the standard deviations can hinder run times if no tuning is performed for each standard deviation. Each algorithm has a set of parameters which may be tuned to increase performance.

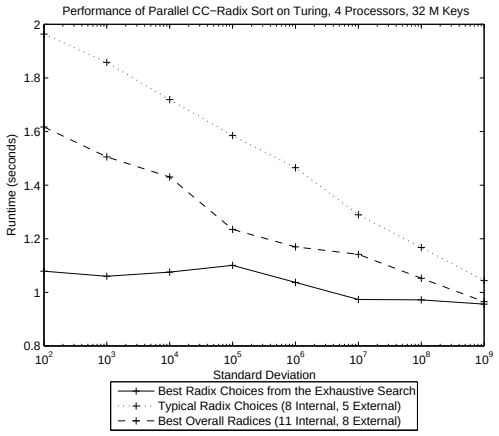

Figure 2: Performance of Parallel CC-Radix Sort when sorting 32M Keys on Turing with 4 processors.

With multicore systems being the new standard for personal computing, there is new need for common algorithms to be designed for shared memory. This presents new challenges to the generator as multicore systems bring new architectures and problems. However, shared memory also allows us more freedom with languages and algorithms as communication between processors is faster. We are extending this library generator to target multicore systems. Other plans for extension include supporting generic data types.

3 Generation of a Sorting Routine

As mentioned above, in [9, 10] it was shown that performance of each sorting routine depends not only on the architectural features of the target machine, but also on the input characteristics such as number of keys and standard deviation. As a result, the sorting routine generation was carried out in two phases. In the first phase, the *training* phase, empirical search is used to determine the values of those parameters on the target architecture as discussed below. In addition, the training phase collects information about the performance of different sorting routines as the input characteristics change. Then, at runtime, tables will be used to select an algorithm and the parameter values for each algorithm. The collected information is used during the next phase, the *runtime* phase, where the input is examined for certain characteristics, and the appropriate sorting routine is selected. In this work, we follow a strategy similar to that of [9, 10], but produce parallel algorithms instead of serial.

The training phase is completed offline. We use a pseudo-random number generator to produce input data

sets with different numbers of keys and standard deviations. We create a lookup table according to our input characteristics. At runtime, the standard deviation of the input must be determined. Then, the table can be used to determine the best algorithm at runtime.

We evaluate our library generator using two different clusters, each using up to 16 processors. One cluster, called Turing, is composed of dual G5 processors. The other cluster, called Mercury, is composed of dual Itanium 2 processors. The computers within each cluster are connected via Myrinet-2000 hardware. Communication between the processors is implemented using MPI. For all the figures shown in this section we use 64M keys and 4 processors, unless it is specified differently. More extensive experiments can be found in [15].

3.1 Tuning Parallel CC-radix

The challenge in Parallel CC-Radix is balancing the load. To do this we must select good radices. Different standard deviations call for different radices. Large radices help separate highly concentrated data, but cause higher cache miss ratios due to the size of the histogram used to count the data going into each bucket. Small radices will have better cache performance, and will work well if data is evenly spread out. Our generator searches for best internal and external radices due to the effects of the standard deviation.

To evaluate the performance of parallel CC-Radix, we run an exhaustive search over all possible choices for external and internal radices. Table 1 shows the best choices for each standard deviation when sorting 64M keys in 4 processors.

Standard Deviation	Int. Turing/Mercury	Ext. Turing/Mercury
100	20/20	15/15
1,000	20/20	14/13
10,000	20/20	13/21
100,000	21/20	18/18
1,000,000	21/20	18/16
10,000,000	17/20	13/12
100,000,000	14/12	11/10
1,000,000,000	11/11	8/5

Table 1: Optimal internal and external radix sizes on 4 processors of the Turing and Mercury clusters for various standard deviations and 64M keys.

3.2 Tuning Sample Sort

In our implementation of Sample Sort, each processor locally sorts the keys, and selects s regularly spaced splitters. These are then broadcast to all other processors so that the the global splitters can be selected. After the global splitters are chosen, the keys are sent to the appropriate processor, where a final sort is performed. Given this implementation, there are three sources of variation:

1. The routine to perform the local sort. In our implementation, we evaluate both Quicksort and CC-Radix Sort.
2. The number of splitters (s) each processor selects.
3. The routine to perform the final sort .

We use empirical search to determine the best choices along these three dimensions for each standard deviation.

We found that for our training data, CC-Radix Sort always outperforms quicksort as the local sort on Turing and Mercury.

For both Turing and Mercury, we found that no matter the standard deviation, the best choice for load balance on P processors is $nP - 1$ samples (n is a positive integer). This evenly balances the load between processors as $nP - 1$ samples implies n groups of data per processor. Equal number of groups on a processor provides good load balance.

It is also necessary to choose whether to use Merge Sort or CC-Radix . Our experimental results show that as the number of processors increases Merge Sort becomes slower, while CC-Radix becomes faster.

3.3 Parallel CC-Radix versus Sample Sort

After tuning parallel CC-Radix and Sample Sort, we need to determine which algorithm performs better. Figure 3 shows the running times of both algorithms when sorting 8M keys in 8 processors. The figure shows that at low standard deviations, Sample Sort performs better than Parallel CC-Radix. The reason is that with low standard deviation, Parallel CC-Radix chooses a large radix to achieve acceptable load balancing. When a large radix is used, the histogram used to accumulate counts of data keys per bucket is very large, causing many cache misses, and lower performance. As standard deviation increases, smaller radix are the best, which results in the better performance of Parallel CC-Radix versus Sample Sort. It is important to note that the execution time of Sample Sort increases with standard deviation, while Parallel CC-Radix decreases.

We must also note that while we see a large benefit in selecting one of these two algorithms on Mercury, parallel CC-Radix sort generally dominates the results on Turing in Figure 3. Results depend on many aspects of the system like number of data items, number of processors and specifications of the system. There are scenarios where large benefit is gained from the two algorithm option on Turing as well.

4 Comparisons

In this Section we compare the performance of our generated library to other parallel algorithms.

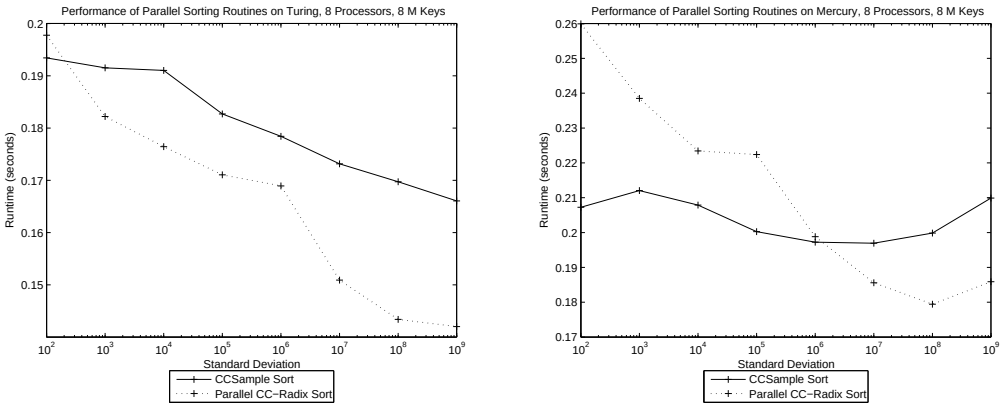

Figure 3: Eight processors using different sorting methods for 8 M keys.

	Turing					Mercury				
Number of Processors	1	2	4	8	16	1	2	4	8	16
Sample Sort Low Std Dev	3.1424	3.6596	2.3352	1.4268	0.8597	4.5038	4.6154	2.5776	1.6359	0.9496
Sample Sort Middle Std Dev	3.2417	3.6945	2.4079	1.5329	0.9447	4.8906	4.8086	2.7415	1.6822	1.0020
Sample Sort High Std Dev	3.0983	3.9883	2.5398	1.5635	0.9514	6.0430	5.2539	2.9246	1.7528	.9849
Parallel CC-Radix Low Std Dev	1.9707	2.8725	2.1367	1.3209	0.7341	4.4872	4.7097	3.2793	1.8641	3.0689
Parallel CC-Radix Middle Std Dev	3.2132	3.1115	1.9339	1.2803	0.7223	4.8805	4.3345	2.4344	1.4206	0.7933
Parallel CC-Radix High Std Dev	3.2513	3.2851	1.9536	1.1943	0.6702	5.9413	4.6098	2.7440	1.5677	0.8104
Tuned Low Std Dev	2.1428	3.0002	2.0787	1.4738	0.7449	4.6633	4.5956	2.5512	1.6424	0.9431
Tuned Middle Std Dev	3.3911	3.0318	2.0214	1.3853	0.8222	5.0497	4.3028	2.3954	1.3582	0.7749
Tuned High Std Dev	3.4571	3.2969	2.0437	1.2608	0.6829	6.1068	4.5735	2.6998	1.5150	0.8062
PSRS Low Std Dev	7.8954	5.3401	3.0889	2.2065	1.7483	13.7438	7.7984	4.0913	2.4694	1.6116
PSRS Middle Std Dev	8.8595	6.3425	3.7854	2.5623	1.5947	13.4193	7.7487	4.1704	2.4726	1.5400
PSRS High Std Dev	15.0808	9.1946	4.7162	2.9312	1.7777	13.6752	7.8042	4.1649	2.5935	1.5262
Charm++ Radix Sort Low Std Dev	16.3861	8.1631	4.1942	1.9898	0.9946	17.3033	6.0751	3.0310	1.5029	0.7773
Charm++ Radix Sort Middle Std Dev	58.9232	29.2309	11.7779	6.8832	3.3404	24.2807	7.5268	3.7946	1.9222	0.9198
Charm++ Radix Sort High Std Dev	99.8141	41.7781	20.0149	9.7275	4.8124	25.1906	9.1257	3.9310	1.9875	0.9209

Table 2: The run times of various sorting algorithms on Turing and Mercury when sorting 64M keys. Low standard deviation: 100, Middle standard deviation: 10^6, High standard deviation: 10^9.

Figure 4 compares the performance of the sorting routine generated using the techniques described in this paper (*Optimally Tuned Sorting Algorithm*) versus *Charm++ Radix Sort* and *Parallel Sorting using Regular Sampling* $PSRS$. Table 2 compares the run times of various sorting algorithms for Turing and Mercury. We include the performance of the tuned Sample Sort and Parallel CC-Radix implementations as well as the run times of the generated sorting routine that we call Tuned. Tuned is either Sample Sort or Parallel CC-Radix (selected by a function based on offline training as discussed above) plus the overhead of computing the standard deviation. Notice that in the table the time reported for Tuned in Mercury is often smaller than both Sample Sort and Parallel CC-Radix. This is because the execution times of tuned and the best of the two algorithms on which Tuned is based are fairly close so that the natural variation in timing causes this effect. Also, the overhead of computing standard deviation was very small in Mercury, around 1%, while in Turing was around 5%. We also include the performance of PSRS and the Charm++ Radix Sort. As can be seen, Tuned runs significantly faster than PSRS and Charm++ Radix Sort.

5 Related Work

The closest related work is that of [9, 10]. However, they generate sequential sorting routines while we generate parallel ones. In [9], they follow a similar strategy to the one described in this paper in that several algorithms are tuned separately and the best algorithm is chosen based on the platform and input characteristics. In [10], they explore the generation of hybrid sorting routines using genetic algorithms. Some of the techniques described in [10] have been used in our library generator when optimizing the sequential algorithms.

Bidan and Toledo [1] present a framework to generate hybrid sequential sorting routines. They tune the algorithm to take into account number of keys of the input, but other input characteristics such as standard

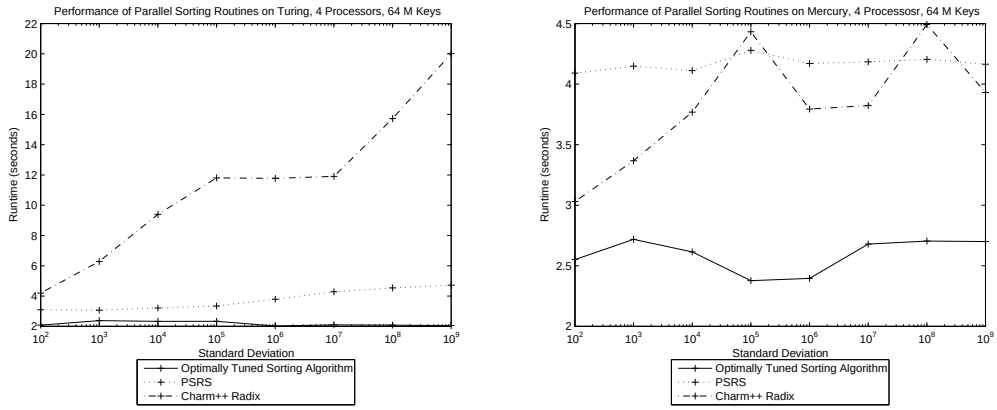

Figure 4: The best performance of various parallel sorting routines with four processors.

deviation are not considered.

Brewer [2] presents an extensive framework for parallel algorithm selection by predicting algorithmic runtimes and selecting the fastest.

Thomas et al. [13] present a framework for the generation of a standard template parallel library. Their framework uses machine learning in addition to empirical experimentation to train a parallel system. While they perform tuning at runtime, their emphasis is on algorithm selection, and fine tuning of individual algorithms is not performed.

Finally, Jiménez et al. [5, 6, 7] have performed significant work to increase data locality when performing a radix sort. They take steps to improve cache usage and load balancing. We use CC-radix and Parallel CC-radix as described by Jiménez et al. In their research, they take limited information about input charactaristics while determining a radix, while we use an empirical search.

6 Conclusion

We have presented some results from our ongoing study on automatic generation of parallel sorting routines. It was shown that dynamic adaptation and offline training are effective for the automatic generation of parallel sorting algorithms. The results of our experiments show that automatic tuning results in most of the cases in a parallel sorting routine that is faster than any of the routines that we could find in the public domain. This however, is not necessarily universally true. It must be pointed out that our objective was not to generate the fastest implementation, but to show that automatic tuning is an effective strategy for parallel sorting. Thus, when an implementation outperforms those considered by the generator, the generator can be extended by incorporating the new algorithm guaranteeing in this way that automatically generated routines can always produced the best known results.

References

[1] E. Bida. An Automatically-Tuned Sorting Library. http://www.cs.tau.ac.il/ stoledo/pubs.html.

[2] E. A. Brewer. High-Level Optimization via Automated Statistical Modeling. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 80–91, 1995.

[3] M. Frigo. A Fast Fourier Transform Compiler. In *Proc. of Programing Language Design and Implementation*, 1999.

[4] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI (2nd ed.): Portable Parallel Programming with the Message-Pas sing Interface"*. MIT Press, 1999.

[5] D. Jiménez. Algoritmos de Ordenación Conscientes de la Arquitectura y las Características de los Datos, Thesis. Technical report, Department of Computer Architecture, Universitat Politècnica de Catalunya, Spain, 2004.

[6] D. Jimenez-Gonzalez, J. L. Larriba-Pey, and J. J. Navarro. Communication Conscious Radix Sort. In *Proc. of the International Conference on Supercomputing*, pages 76–82, New York, NY, USA, 1999. ACM Press.

[7] D. Jiménez-González, J. Navarro, and J. Larriba-Pey. CC-Radix: A Cache Conscious Sorting Based on Radix Sort. In *Euromicro Conference on Parallel Distributed and Network based Processing*, pages 101–108, February 2003.

[8] S. Kumar. Optimizing Communication for Massively Parallel Processing. *Parallel Programming Laboratory Technical Report #05-17*, 2005.

[9] X. Li, M. J. Garzarán, and D. Padua. A Dynamically Tuned Sorting Library. In *In Proc. of the Int. Symp. on Code Generation and Optimization*, pages 111–124, 2004.

[10] X. Li, M. J. Garzarán, and D. Padua. Optimizing Sorting with Genetic Algorithms. In *In Proc. of the Int. Symp. on Code Generation and Optimization*, pages 99–110, 2005.

[11] M. Püschel et al. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232–275, 2005.

[12] H. Shi and J. Schaeffer. Parallel Sorting by Regular Sampling. *Journal of Parallel and Distributed Computing*, 19(4):361–372, 1992.

[13] N. Thomas, G. Tanase, O. Tkachyshyn, J. P. and Nancy M. Amato, and L. Rauchwerger. A Framework for Adaptive Algorithm Selection in STAPL. In *Principles and Practice of Parallel Programming (PPoPP)*, volume 10, pages 277–288, 2005.

[14] R. Whaley, A. Petitet, and J. Dongarra. Automated Empirical Optimizations of Sofware and the ATLAS Project. *Parallel Computing*, 27(1-2):3–35, 2001.

[15] B. Garber. The Necessity of Tuning to Achieve Top Performance Parallel Sorting Algorithms. MS thesis. University of Illinois, 2006.