Automated Tuning in Parallel Sorting on Multi-core Architectures

Haibo Lin[1], Chao Li[2], Qian Wang[3], Yi Zhao[1], Ninghe Pan[4], Xiaotong Zhuang[5], and Ling Shao[1]

[1] IBM Research - Beijing, China
{linhb,zhaoyizy,shaol}@cn.ibm.com
[2] University of Massachusetts, Boston MA, USA
chaoli@cs.umass.edu
[3] Beijing Institute of Technology, Beijing, China
duck_shmily@hotmail.com
[4] IBM Systems & Technology Group, Beijing, China
panningh@cn.ibm.com
[5] IBM Watson Research Center, Yorktown Heights NY, USA
xzhuang@us.ibm.com

Abstract. Empirical search is an emerging strategy used in systems like ATLAS, FFTW and SPIRAL to find the parameter values of the implementation that deliver near-optimal performance for a particular machine. However, this approach has only proven successful for scientific kernels or *serial* symbolic sorting. Even commercial libraries like Intel MKL or IBM ESSL do not include parallel version of sorting routines. In this paper we study empirical search in the generation of parallel sorting routines for multi-core systems. Parallel sorting presents new challenges that the relative performance of the algorithms depends not only on the characteristics of the architectures and input data, but also on the data partitioning schemes and thread interactions. We have studied parallel sorting algorithms including quick sort, cache-conscious radix sort, multi-way merge sort, sample sort and quick-radix sort, and have built a sorting library using empirical search and artificial neural network. Our results show that this sorting library could generate the best parallel sorting algorithms for different input sets on both x86 and SPARC multi-core architectures, with a peak speedup of 2.2x and 3.9x, respectively.

1 Introduction

Multi-threaded programming is being extended from its current niches to more widespread use with the advent of multi-core processors. One of the major challenges in parallel algorithm implementation is the lack of a comprehensive methodology to drive optimization transformations. Empirical search [1] is an emerging research topic in identifying from a set of algorithms and implementations the one that performs the best on machines where the library is being installed, like ATLAS [1], FFTW [2] and SPIRAL [3]. During the installation phase, they use empirical search to identify the best version of transformations

P. D'Ambra, M. Guarracino, and D. Talia (Eds.): Euro-Par 2010, Part I, LNCS 6271, pp. 14–25, 2010.

based on the characteristics of the target platform and the size of the input data. An automatically tuned sorting library [4][5] is also built using empirical search and machine learning to adapt to different characteristics of input sets.

However, empirical search has only proven successful for algorithms that are used for either serial/parallel scientific kernels or serial symbolic sorting. Even commercial libraries like Intel MKL or IBM ESSL do not include parallel versions in their sorting routines. There are several challenges in tuning parallel sorting for multi-core platforms. First, the lack of precise formulation regarding how the memory hierarchy affects the performance of multiple threads renders it difficult to optimize sorting algorithms. Second, the trade-off between single thread performance (locality) and scalability has not yet been well studied to select optimization parameters. Third, the performance of sorting algorithms relies heavily on the characteristics of input sets, which would, very possibly, introduce load balancing issues for multi-cores. Finally, the overall complexity and the parameter search space has become too large for pure empirical approaches.

In this paper, we present and evaluate a strategy for automatic generation of parallel sorting libraries on multi-core architectures. We evaluate several well known parallel sorting algorithms including quick sort, radix sort, multiway merge sort, sample sort, as well as quick-radix sort. We first extract the most significant factors that influence the performance of multi-threaded sorting algorithms. By characterizing these factors and then partitioning them into orthogonal groups, we are able to reduce the search dimension down to an acceptable range, and accelerate the parameter tuning process using genetic algorithms. We apply work stealing [6] scheduling algorithm to balance the workload on different threads. Based on the tuning results, an artificial neural network (ANN) is built to correlate the input set with the best sorting algorithm and corresponding parameter set. The experimental results show that this sorting library could generate the best parallel sorting algorithms for different input sets on both x86 and SPARC multi-core systems. The library selected algorithm could improve the performance by over 2.2x on Clovertown, and up to 3.9x on Niagara, compared to the performance based on incorrect decisions.

The major contributions of this paper include:

- A self-tuned parallel sorting library that generates the best sorting algorithms adaptive to the characteristics of architectures and input data sets.
- An efficient parameter tuning strategy in seeking the best optimization parameters in parallel sorting algorithms. Particularly, a thread affinity tuning method is presented for multi-cores.
- Detailed experiments and performance analysis of parallel sorting algorithms on different multi-core architectures.

The remainder of the paper is organized as follows. Section 2 presents related work. In Section 3, we introduce parallel sorting algorithms on multi-core. Section 4 presents the empirical search for parameter values. In Section 5, we present the use of artificial neural network to build the sorting library. Section 6 shows experimental results. Concluding remarks are given in Section 7.

2 Related Work

Sorting has been extensively studied in literature. A large number of sorting algorithms have been proposed and their asymptotic complexity, in terms of the number of comparisons or number of iterations, has been carefully analyzed [7]. Parallel sorting algorithms have also been explored since the early 1960's and have been deployed on real parallel machines like CM-2. An early study on parallel sorting was conducted in 1968 when Batcher invented bitonic sorting network [8]. In 1983, Reif and Valiant proposed a more practical randomized algorithm for sorting [9], called flashsort. Many more parallel sorting algorithms were proposed later on, including parallel version of radix sort and quick sort [10][11], a variant of quicksort called hyperquicksort, and other algorithms such as smooth sort, column sort, and parallel merge sort.

In recent year, sorting algorithms have been extended to exploit the SIMD and multicore capability of modern processors, including GPUs, Cell, and others. Bitonic sort is implemented using a sorting network without unpredictable branches, making it well suited for SIMD processors. Mergesort was also implemented and optimized on current SIMD CPU architecture.

ATLAS [1], FFTW [2] and SPIRAL [3] are among the best known autotuned libraries. ATLAS focuses on the implementation of matrix multiplication. It uses almost exhaustive search to find the best parameter values such as tile size and loop unrolling factors. SPIRAL and FFTW generate signal processing libraries. The size of input set is taken into consideration during the algorithm selection phase. The empirical search was further extended to symbolic sorting in [4][5], where the distribution of the input data was considered in the search space. Genetic algorithms was used to map the characteristics of the input to the best sorting algorithms. To our best knowledge, the empirical search has only been applied to serial/parallel scientific kernels and serial symbolic sorting. [12] discusses a library generator for parallel sorting on distributed memory. Our work in this paper is the first attempt to use auto-tuning for parallel sorting on multi-core architectures.

3 Sorting Algorithms

Traditional parallel sorting algorithms for massive parallel machines may not be applicable to multi-core environment directly due to their differences in memory hierarchies and interconnects. In this section, we will introduce five classical parallel sorting algorithms used in our tuning and library building process.

Quick Sort. Quick sort selects a pivot and divides the input set of records into two subsets, the set with records smaller than the pivot and the set with records lager than or equal to the pivot. The parallelization of quick sort is quite straightforward, since the two subsets can be sorted simultaneously. Data partitioning strategy is one of the key factors to be considered in parallel quick sort. We do not use parallel quick sort for small inputs, because the parallelization itself introduces extra overhead such as creating new tasks and synchronizing

the threads. Therefore, we regard *partition size* as one of the parameters to be tuned for quick sort.

Cache-Conscious Radix Sort. Jiménez [13] introduced a cache-conscious radix sort (CC-radix) to improve data locality of radix sort. It used *reverse sort* to partition the entire set into several subsets according to the values of r most significant digits of the integers, repeat the similar operation until the size of subsets are small enough to be fit into the cache. Then radix sort is executed on each of the subsets in parallel. For parallel CC-radix sort, the average size of the subsets after reverse sort (denoted as *partition size*) is in some sense a trade-off between parallelism and locality. A larger partition size reduces the number of reverse sort passes, making the algorithm parallelizable at an earlier stage. While a smaller partition size is more friendly to the cache system. One of the other important parameters in CC-radix sort is the length of radix.

Multiway Merge Sort. Multiway merge sort first partitions the input set of records into several small subsets with the same size and sorts the subsets separately. Although the subsets can be sorted in parallel, the final merge phase turns out to take a long time. We use a hierarchical merge sort in our parallel algorithm. The structure of the merge tree need be tuned. This includes the *fanout* of each node and the *depth* of the tree. The product of these two values determines the number of subsets to be sorted. In sequential merge sort, the fanout of a node mainly depends on the cache line size. In parallel sorting, larger fanout value results in more small subsets but longer sequential merge, whereas smaller fanout value indicates faster sequential merge but worse scalability.

Sample Sort. Assuming n input records are to be sorted on a machine with p processors, sample sort[14] proceeds in four phases: 1) the input data set is divided into p subsets evenly and sorted separately; 2) k sample records are picked from each subset to calculate the $p-1$ splitter records, which could partition the linear order of the n records into p subsets; 3) each processor collects the i^{th} subset from other processors; 4) each processor sorts its subset. Helman [15] suggested that $((x+1)\frac{n}{p^2 s})^{th}$ elements should be selected from each subset as a sample, for $(0 \leq x \leq s-1)$ and a given value of s $(p \leq s \leq \frac{n}{p^2})$. This makes the communication among distributed nodes more regular, leading to a better performance on traditional parallel machines. However, on multi-core architectures with shared memory model, the overhead to access other data partitions is relatively small. So the sampling could be less accurate, but much faster. There are two important parameters for sample sort: the number of subsets and the number of records to be sampled in each subset.

Quick-Radix Sort. Quick-Radix(QR-sort) employs a "top-down" policy which first partitions records into ordered subsets and then sorts the subsets in parallel. Once the subsets are sorted, the whole set of records are sorted, without the need for a final scan. We use quick sort to divide the input set into ordered subsets and then apply other sorting algorithms on each set in parallel. One of the advantages

of QR-sort is that quick sort is an in-place algorithm, which help improve cache locality. QR-sort is similar to the introsort [16], which uses heap sort to optimize the worse-case complexity of quick sort. While QR-sort applies quick sort to quickly generate parallel subtasks. Like merge sort, quick-radix sort is also a combination of data partitioning strategy and radix sorting on partitions. The most important parameter for quick-radix sort is the number of subsets, which depends on the characteristic of the input sets.

4 Tuning Parameters

There are three parameters that are common to all sorting algorithms. we use two register sort algorithms to explore architectural factors for small data sets. *Register Sort Threshold* is the number of elements for register sort switch. *Register Sort Algorithm* is the selection between insertion sort and sorting network. These two parameters are most related to the number of registers in a specific processor, thus they have little to do with the input. *Thread Affinity* is the mapping between software threads and hardware threads. Different data size may prefer different affinity, e.g., it might be good to map threads to adjacent cores to take advantage of the shared cache for small data sets. In contrast, for large data sets, one might want to map threads to other cores for larger cache per core. Therefore, we regard thread affinity as input sensitive.

Four of the five parallel sorting algorithms (excluding quick sort) use radix sort when sorting a partitioned subset. They use a common parameter called *Radix Length*. This parameter depends heavily on the input, especially the standard deviation. In quick sort and CC-radix sort, we use *partition size* to specify the upper bound of a subset size. When a subset is less than the partition size, we do not create new tasks. In sample sort and quick-radix sort, we use the *number of subsets* to describe how many subsets we want to process in parallel. This value varies with the size of the input set.

We use *genetic algorithm* [5] in searching the parameter set. We first begin with an original parameter set and generate nearby parameter sets randomly in each step. Next, we use pre-generated tests to evaluate the performance of sorting algorithms with new parameter sets. The initial parameters for empirical search are selected from recommended parameters by known implementations (e.g. $\log_2 S_{TLB} - 1$ for radix length), theoretical prediction of best values (e.g. cache size for partition size in quick sort and reverse sort phase of CC-radix sort) and parameters in their original application design (e.g. number of subsets for sample sort).

4.1 Tuning Steps

We first split those parameters into orthogonal groups with no or few dependencies, then look for the best parameter set for each group. Fig. 1 illustrates the whole work flow of parameter tuning:

1. We tune the *register sort threshold* and *register sort algorithm* only once, then we use the parameter set as building blocks for other sorting algorithms.

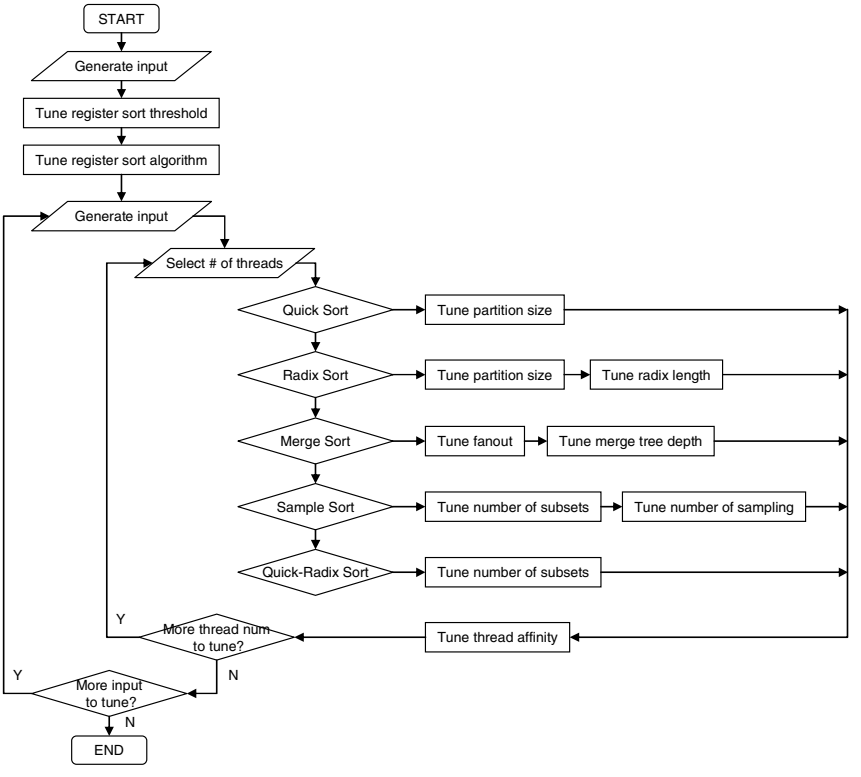

Fig. 1. Flowgraph for parameter tuning

2. After the parameters of register sort have been determined, we generate different input sets to tune the parameters for each sorting algorithm.
3. We also tune the number of threads. Due to the difference in scalability, algorithms may exhibit different performance metrics when running at different number of threads.
4. When the number of threads is greater than one and less than the number of hardware threads, we tune thread affinity. Our experience indicates that tuning parameters for different affinity produces almost identical values. Therefore, we tune thread affinity after other parameters are fixed.

4.2 Thread Affinity

Most multi-core architectures employ hierarchical cache system design. For example, each core may have a private L1 cache and share the L2 cache with neighboring cores. In this case, the mapping of software threads to hardware ones may have impact on performance in terms of cache sharing and cache size per thread. We assume software threads are symmetric. This is largely true for sorting since different threads perform the same job most of the time. Given a

linear ordered hardware thread number, we define the *mapping distance* (*dist* for short) to be the difference among hardware thread IDs where the software threads are mapped.

Let N_h be the total number of hardware threads, and N_s be the number of software threads. The mapping distance needs to be tuned is a set satisfying the following equation.

$$dist \in \{2^d | 0 \leq d \leq \log \frac{N_h}{N_s}\}$$

For example, to map two software threads to eight hardware threads, there are three hardware pair choices: $< 0, 1 >$, $< 0, 2 >$ and $< 0, 4 >$. In this way, we attempt to probe different memory hierarchies of underlying multi-core architecture. We can not guarantee all possible combinations are tuned, but this gives us an reduced search space, which can be realistically implemented.

5 Building the Library

We first characterize the size and standard deviation of input data. Next, we build an Artificial Neural Network (ANN) [17] to learn the best algorithms and their parameter sets through tuning. To use it online, the library calculates characteristics of the input, feeds them to the ANN, and uses the output algorithm and parameter set to sort the input data.

5.1 Characterizing Input

We employ the concept of *entropy* in [4] to characterize the input data set. For a input set with b digits, the entropy is computed as $\sum_i -P_i \log_2 P_i$, where P_i is the ratio between the number of records with value i on this digit and the total number of records. A vector of entropies are obtained. Each element of the vector represents the entropy of a digit position in the record. If the values of a digit position span a wide range, the entropy is high. Therefore, the data will be distributed into many buckets which requires fewer rounds of reverse sort.

In our implementation, the entropy calculation can be further improved by shortening the length of the entropy vector from b to $\lceil b/r \rceil$, where r is the length of radix. We use sampling to calculate the entropy vector instead of scanning the whole input set.

5.2 Artificial Neural Network

ANN is a non-linear statistical data modeling tool. It can be used to model complex relationships between inputs and outputs or to find patterns in data. An ANN can be defined as a highly connected array of elementary processors called *neurons*. A widely used model is the multi-layered perceptron (MLP) ANN, which consists of one input layer, one or more hidden layers and one output layer. Each layer employs several neurons and each neuron in a layer is connected to the neurons in the adjacent layer with different weights.

We use *supervised learning*, in which we are given a set of example pairs $(x, y), x \in X, y \in Y$ and aim to find a function $f : X \rightarrow Y$ that matches the example. In parallel sorting, we define X, Y as follows: $X : < S, E, N >, Y : < A, P, T >$, where S represents the size of input, E the entropy of input, N the number of threads, A the algorithm, P the parameter set, and T thread affinity.

During the library installation time, we generate a set of input sets, and tune each algorithm to obtain the parameter set and thread affinity. By comparing the performance of each algorithm, we can select the best algorithm for a given input set and thread number. In this manner, we can get the training set, i.e., $< X, Y >$ pairs, for the ANN.

6 Performance Evaluation

The experiments were conducted on two multi-core platforms, Intel Clovertown and Sun Niagara. These two platforms were chosen because they represent two types of multi-core design. Clovertown is a quad-core processor, each with its private L1 cache. Every two cores share a L2 cache. Two Clovertown chips are connected via FSB to build an 8-core system. Niagara has 8 cores, each supporting 4 hardware threads. Each core has its private L1 cache, and all cores share one L2 cache.

Our input data are records of 64-bit integers. The training sets in the tuning phase are 512K/16M records (4M/128M bytes) on Clovertown, and 64K/2M records (512K/16M bytes) on Niagara, representing L2 cache-resident and cache-outside setting. The standard deviation is chosen from 2^{25} and 2^{50}. The testing data sets are selected differently from the training sets, which are 32M records (256M bytes) on Clovertown, and 128K records (1M bytes) on Niagara.

6.1 Sensitivity Analysis

Fig. 2 shows the performance improvement of quick sort, CC-radix sort and sample sort on Clovertown with different number of threads for large input sets. The improvements on quick sort is about 5∼15%. The main difference of parameters is the partition size. The initial value is chosen as half the L1 cache size, which is about 2K records. After parameter tuning, the partition size ranges from 16K to 48K records, depending on the number of threads.

CC-radix sort is about 10∼35% faster than the one with initial parameters. As in quick sort, the initial partition size is 2K. This result shows that a larger size (2∼48K) produces better performance. Also, the radix length is reset to 9 from 7 after tuning.

The improvement for sample sort ranges from 10% to 36% for different number of threads. The initial number of subset is set to twice the number of threads. This value matches the tuning results for large number of threads (7 or 8). With fewer threads, sample sort prefers more subsets (12 subsets for 4-thread) since the workload is more balanced. The number of samples is more difficult to select. We limit the total samples to be the half of L1 cache size, which is about 2048 records. The results show that a total of 300∼500 samples is good enough.

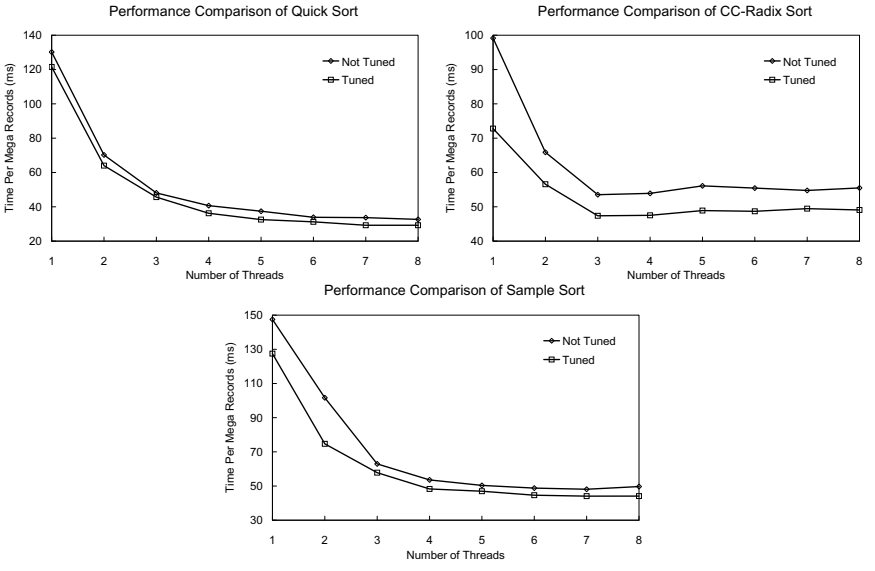

Fig. 2. Performance improvements of quick sort, CC-radix sort and sample sort after parameter tuning on Clovertown

6.2 Thread Affinity

Fig. 3 illustrates the experimental results for thread affinity on Niagara. The dots in this figure are grouped by the number of threads. Given a thread number, we have different schedule distances, as shown in the second x axis (the top line).

We have observed that for a given number of threads, the longer the schedule distance, the faster the algorithm runs. Taking thread number 2 as an example, the performance improves when the schedule distance increases from 2 to 4, and stays at that level for longer schedule distances. This is because these two software threads are scheduled to different hardware cores at a distance of greater than or equal to 4, considering 4-way multi-threading per core. Therefore, the software threads will not compete for processing time of hardware cores. This can be verified by CPU utilization breakdowns in Fig. 3. Using two threads, the CPU is fully utilized at all schedule distances. The behavior of cache system does not make much difference on Niagara, as L1 is very small and L2 is shared.

Then we compare the performance impact of thread affinity for different sorting algorithms. Comparing quick sort and sample sort using 16 threads, the impact of thread affinity on sample sort is much less than quick sort. This can be explained by CPU utilization breakdowns again. At schedule distance of 1, the CPU utilization of sample sort is about 50% already, while quick sort occupies over 80% of the CPU time. Therefore, having more cores is more helpful for quick sort than sample sort.

Fig. 4 also shows that longer schedule distance produces better performance on Clovertown. However, the reason for the performance improvement on

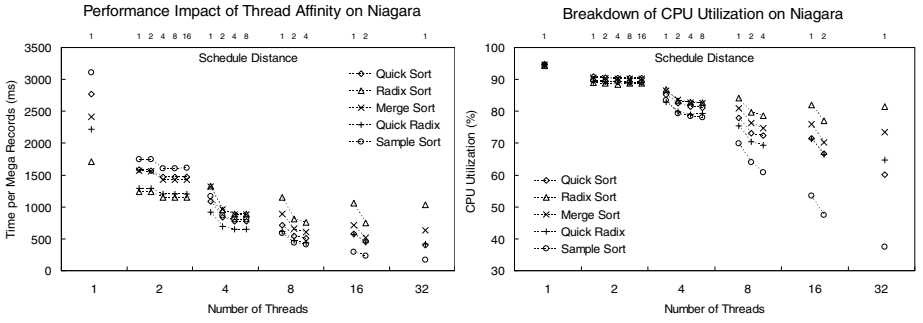

Fig. 3. Experimental results of thread affinity on Niagara

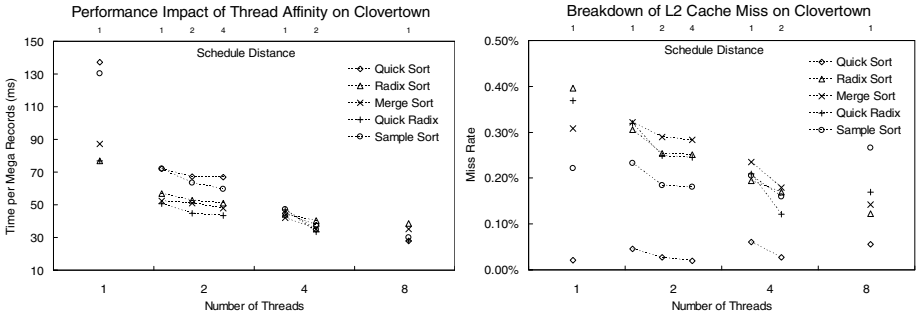

Fig. 4. Experimental results of thread affinity on Clovertown

Clovertown is different from that on Niagara. Since Clovertown does not support SMT, each software thread is mapped to one hardware processor core. The contention for CPU resources can be avoided. As two adjacent cores share one L2 cache, scheduling threads to distant cores having separate L2 cache could yield larger cache size per thread. Considering two threads as an example, the threads are mapped to every other cores at schedule distance of two. Each thread gets 4 MB L2 cache. As a result, the L2 cache miss rate is reduced up to 40%. Also note that the cache miss rate for quick sort is lower than other sorting algorithms. This is because quick sort is an in-place algorithm, while others use radix sort, requiring extra temporary storage.

6.3 Sorting Library

We compare the performance of individual algorithm with our auto-tuned sorting library. We ran these sorting routines on input sets of 32M records with standard deviation of 2^{37} on Clovertown, which is different from the training set. Experiments were conducted with different number of threads to examine both the absolute performance and scalability.

To use our sorting library, a few samples have to be selected to compute entropy, so as to be fed into ANN to produce appropriate sorting algorithm

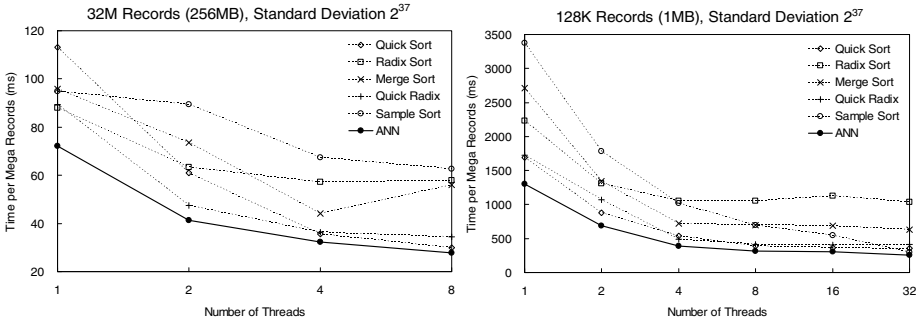

Fig. 5. Performance comparison among sorting library and individual parallel sorting algorithms on Clovertown (left) and Niagara (right)

and corresponding parameter set. As we only used $1/10^4 \sim 1/10^3$ sampling records from the input set, the cost on sampling and entropy computation was negligible in our experiments. In the ANN implementation, we used the FANN library, which was configured as 4 layers, including around 50 hidden neurons. We generated about 1000 training data, and FANN created a standard fully connected back propagation neural network.

Fig. 5 shows the results on Clovertown and Niagara. Dotted lines represent the performance of individual sorting algorithms. Since the test data is different from the training data, parameters used here are not tuned. The solid line represents library selected algorithm and parameter set via ANN. The results show that our sorting library outperforms the best of all five algorithms by up to 22%. Compared to an inappropriate decision, performance improvement could be 2.25x. We notice that quick sort reveals good scalability, however, it is only chosen when the number of threads is large, due to its relatively low performance for single thread. When the number of threads is small, the library prefers radix sort. QR-sort is picked for medium number of threads.

The library also performs well on Niagara. The improvement to the best sorting algorithm ranges from 10% to 33%. The performance gap between the library and the worst sorting algorithm could be 3.9x. Quick sort performs relatively well with small number of threads, and is successfully selected by the library. As the number of threads increases, sample sort is chosen by our library.

7 Conclusions

In this paper, we presented automated tuning for parallel sorting algorithms on modern multi-core architectures. We discussed several practical parameter tuning methodologies for parallel sorting considering both the characteristics of architectures and input data sets. Particularly, thread affinity was taken into account on multi-core architectures. We have built an parallel sorting library based on parameter tuning and machine learning techniques of artificial neural network. Experimental results show that the automated tuning technique is

effective for parallel sorting on multi-core systems. Compared to incorrect decisions, the speedup can be 2.2x on the Intel Clovertown processor and up to 3.9x on the Sun Niagara processor.

References

1. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated empirical optimizations of software and the ATLAS project. Parallel Computing 27(1-2), 3–35 (2001)
2. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proceedings of the IEEE 93(2), 216–231 (2005); Special Issue on Program Generation, Optimization, and Platform Adaptation
3. Xiong, J., Johnson, J., Johnson, R., Padua, D.: SPL: a language and compiler for dsp algorithms. In: Proc. of the International Conference on Programming Language Design and Implementation (PLDI 2001), pp. 298–308 (2001)
4. Li, X., Garzaran, M., Padua, D.: A dynamically tuned sorting library. In: Proc. of the International Symposium on Code Generation and Optimization (CGO 2004), pp. 111–122 (2004)
5. Li, X., Garzaran, M.J., Padua, D.: Optimizing sorting with genetic algorithms. In: Proc. of the International Symposium on Code Generation and Optimization (CGO 2005), pp. 99–110 (2005)
6. Blumofe, R.D.: Executing multithreaded programs efficiently. PhD thesis, Cambridge, MA, USA (1995)
7. Knuth, D.E.: The Art of Computer Programming, 2nd edn., vol. 3/Sorting and Searching. Addison-Wesley Publishing Company, Reading (1982)
8. Batcher, K.: Sorting networks and their applications. In: AFIPS Spring Joint Computer Conference, vol. 32, pp. 307–314 (1968)
9. Reif, J.H., Valiant, L.G.: A logarithmic time sort for linear size networks. Journal of the ACM 34(1), 60–76 (1987)
10. Blelloch, G.E.: Vector models for data-parallel computing. MIT Press, Cambridge (1990)
11. Brown, T., Xiong, R.: A parallel quicksort algorithm. J. Parallel Distrib. Comput. 19(2), 83–89 (1993)
12. Garber, B.A., Hoeflinger, D., Li, X., Garzarán, M.J., Padua, D.: Generation of a parallel sorting algorithm. In: The Next Generation Software Workshop, in Conjunction with IPDPS (2008)
13. Jiménez-González, D., Navarro, J., Larriba-Pey, J.L.: Cc-radix: a cache conscious sorting based on radix sort. In: Proc. of the Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing, pp. 101–108 (2003)
14. Huang, J.S., Chow, Y.C.: Parallel sorting and data partition by sampling. In: Proc. of the IEEE Computer Society's Seventh International Computer Software and Applications Conferences, pp. 627–631 (November 1983)
15. Helman, D.R., Bader, D.A., JáJá, J.: A randomized parallel sorting algorithm with an experimental study. J. Parallel Distrib. Comput. 52(1), 1–23 (1998)
16. Musser, D.R.: Introspective sorting and selection algorithms. Software Practice and Experience 27, 983–993 (1997)
17. Gurney, K.: An Introduction to Neural Networks. University College London (UCL) Press (1997)