SWARM: A Parallel Programming Framework for Multicore Processors *

David A. Bader, Varun Kanade and Kamesh Madduri

College of Computing
Georgia Institute of Technology, Atlanta, GA 30332
{bader, varunk, kamesh}@cc.gatech.edu

Abstract

*Due to fundamental physical limitations and power constraints, we are witnessing a radical change in commodity microprocessor architectures to multicore designs. Continued performance on multicore processors now requires the exploitation of concurrency at the algorithmic level. In this paper, we identify key issues in algorithm design for multicore processors and propose a computational model for these systems. We introduce SWARM (**S**oft**W**are and **A**lgorithms for **R**unning on **M**ulti-core), a portable open-source parallel library of basic primitives that fully exploit multicore processors. Using this framework, we have implemented efficient parallel algorithms for important primitive operations such as prefix-sums, pointer-jumping, symmetry breaking, and list ranking; for combinatorial problems such as sorting and selection; for parallel graph theoretic algorithms such as spanning tree, minimum spanning tree, graph decomposition, and tree contraction; and for computational genomics applications such as maximum parsimony. The main contributions of this paper are the design of the SWARM multicore framework, the presentation of a multicore algorithmic model, and validation results for this model. SWARM is freely available as open-source from* `http://multicore-swarm.sourceforge.net/`.

1 Introduction

For the last few decades, software performance has improved at an exponential rate, primarily driven by the rapid growth in processing power. However, we can no longer rely solely on Moore's law for performance improvements. Fundamental physical limitations such as the size of the transistor and power constraints have now necessitated a radical change in commodity microprocessor architecture to multicore designs. Dual and quad-core processors from Intel [14] and AMD [2] are now ubiquitous in home computing. Also, several novel architectural ideas are being explored for high-end workstations and servers. The Sun UltraSparc T1 [17] with eight processing cores and four threads per core, is a design targeting targeting multithreaded workloads and enterprise applications. The Sony-Toshiba-IBM Cell Broadband Engine [16] is a heterogeneous chip optimized for media and gaming applications. A research proposal from the Intel Tera-scale computing [12] project has eighty cores. Continued software performance improvements on such novel multicore systems now requires the exploitation of concurrency at the algorithmic level. Automatic methods for detecting concurrency from sequential codes, for example with parallelizing compilers, have had only limited success. In this paper, we make two significant contributions to simplify algorithm design on multicore systems:

- We present a *new computational model for analyzing algorithms on multicore systems*. This model shares some similarities to existing parallel shared memory multiprocessor models, but also identifies key architectural aspects that distinguish current multicore systems (in particular, the memory sub-system).
- We introduce an *open-source portable parallel programming framework* SWARM (**S**oft**W**are and **A**lgorithms for **R**unning on **M**ulti-core). This library provides basic functionality for multithreaded programming, such as synchronization, control and memory management, as well as collective operations. We use this framework to design efficient implementations of fundamental primitives, and also demonstrate scalable performance on several combinatorial problems.

On multicore processors, caching, memory bandwidth, and synchronization constructs have a considerable effect

*This work was supported in part by NSF Grants CNS-0614915, CAREER CCF-0611589, DBI-0420513 and ITR EF/BIO 03-31654.

on performance. In addition to time complexity, it is important to consider these factors for algorithm analysis. The multicore model we propose takes into account these factors, and can be used to explain performance on systems such as Sun Niagara, Intel and AMD multicore chips. Different models [4] are required for modeling heterogeneous multicore systems such as the Cell architecture.

The SWARM programming framework is a descendant of the symmetric multiprocessor (SMP) node library component of SIMPLE [10]. SWARM is built on POSIX threads that allows the user to use either the already developed primitives or direct thread primitives. SWARM has constructs for parallelization, restricting control of threads, allocation and deallocation of shared memory, and communication primitives for synchronization, replication and broadcast. Built on these techniques, it contains a higher-level library of multicore optimized parallel algorithms for list ranking, comparison-based sorting, radix sort and spanning tree. In addition, SWARM application example codes include efficient implementations for solving combinatorial problems such as minimum spanning tree [9], graph decomposition [6], breadth-first-search [7], tree contraction [8] and maximum parsimony [5].

This paper is organized as follows. Section 2 discusses the computational model we use for analyzing algorithms on multicore systems. We also design a simple benchmark to validate the model. We present SWARM and give a brief overview of its functionality in Section 3. Section 4 discusses some of the fundamental parallel primitives we have implemented in SWARM and presents performance results of a few combinatorial discrete algorithms designed using SWARM.

2 Model for Multicore Architectures

Multicore systems have a number of processing cores integrated on to a single chip [14, 2, 11, 17, 16]. Typically, the processing cores have their own private L_1 cache and share a common L_2 cache [14, 17]. In such a design, the bandwidth between the L_2 cache and main memory is shared by all the processing cores. Figure 1 shows the simplified architectural model we will assume for analyzing algorithms.

Multicore Model.
The multicore model (MCM) consists of p identical processing cores integrated onto a single chip. The processing cores share an L_2 cache of size C, and the memory bandwidth is σ.

(i) Let $T(i)$ denote the local time complexity of the core i for $i = 1, \ldots, p$. Let $T = \max_i T(i)$.

(ii) Let B be the total number of blocks transferred between L_2 cache and the main memory. The requests may arise out of any processing core.

(iii) Let L be the time required for synchronization between the cores. Let $N_S(i)$ be the total number of synchronizations required on core i for $i = 1, \ldots, p$. Let $N_S = \max_i N_S(i)$.

Then the model can be represented by a triple $\langle T, B \cdot \sigma^{-1}, N_S \cdot L \rangle$. The complexity of an algorithm will be represented by the dominant element in this triple.

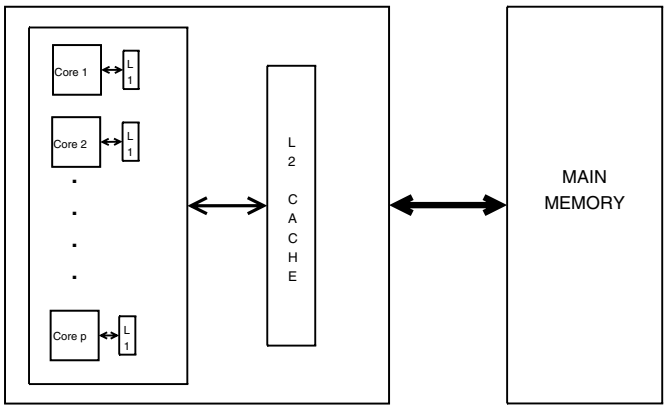


Figure 1. Architectural model for multicore systems

The model proposed above is in many ways similar to the Helman-JáJá model for symmetric multiprocessor (SMP) systems [13], with a few important differences. In the case of SMPs, each processor typically has a large L_2 cache and dedicated bandwidth to main memory, whereas in multicore systems, the shared memory bandwidth will be an important consideration. Thus, we explicitly model the cache hierarchy, and count the block transfers between the cache and main memory in a manner similar to Aggarwal and Vitter's external memory model [1].

In our model, we target three primary issues that affect performance on multicore systems:

1. *Number of processing cores:* Current systems have two to eight cores integrated on a single chip. Cores typically support features such as simultaneous multithreading (SMT) or hardware multithreading, which allow for greater parallelism and throughput. In future designs, we may have up to hundred cores on a single chip.

2. *Caching and memory bandwidth:* Memory speeds have been historically increasing at a much slower rate than processor capacity [15]. Memory bandwidth and latency are important performance concerns for several scientific and engineering applications. Caching is

known to drastically affect the efficiency of algorithms even on single processor systems [18, 19]. In multicore systems, this will be even more important due to the added bandwidth constraints.

3. *Synchronization:* Implementing algorithms using multiple processing cores will require synchronization between the cores from time to time, which is an expensive operation in shared memory architectures.

2.1 Case study: Merge sort

To illustrate the model, we discuss and analyze two sorting algorithms based on p-way merging. The algorithm we may use on an SMP may be different from the one that we design for a multicore system.

Algorithm 1

In the first algorithm, given an input array of length N, we divide this equally among the p processing cores so that each core gets N/p elements to sort. Once the sorting phase is completed, we have p sorted sub-arrays, each of length N/p. Thereafter the merge phase takes place. A p-way merge over the runs will give the sorted array. Each processor individually sorts its elements using some *cache-friendly* algorithm. We do not try to minimize the number of blocks transferred between the L_2 cache and main memory in this approach.

Analysis. Since the p processors are all sorting their respective elements at the same time, the L_2 cache will be shared by all the cores during the sorting phase. Thus if the size of the L_2 cache is C, then effectively each core can use just C/p. Assuming the input size is larger than the cache size, the cache misses will be p times that if only a single core were sorting. Also the bandwidth between the cache and shared main-memory is also shared by all the p cores, and this may be a bottleneck.

We compute the time complexity of each processor. In the sorting phase for each core we have :

$$T_c(\text{sort}) = \frac{N}{p} \cdot \log\left(\frac{N}{p}\right)$$

Then during the merge phase we have :

$$\begin{aligned} T_c(\text{merge}) &= N \cdot \log(p) \\ T_c(\text{total}) &= \frac{N}{p}\log\left(\frac{N}{p}\right) + N\log(p) \end{aligned}$$

Algorithm 2

In this algorithm we divide the given array of length N into blocks of size M where M is less that C, the size of the L_2 cache. Each of such N/M blocks is first sorted using all p cores. This is the sorting phase. When the sorting phase is completed, the array consists of N/M runs each of length M. During the merge phase, we merge p blocks at a time. We keep repeating this till we get a single sorted run. Thus we will need to carry out the merge phase $\log_p\left(\frac{N}{M}\right)$ times.

Analysis. This algorithm is very similar to the I/O model merge sort [1]. Thus this algorithm is optimal in terms of transfers between main memory and L_2 cache. However it will have slightly higher computational complexity. We first compute the time complexity of each core. The p cores sort a total of N/M blocks of size M. We assume the use of a split-and-merge sort for sorting the block of M elements. Thus, during the sorting phase, the time per core is :

$$\begin{aligned} T_c(\text{sort}) &= \frac{N}{M}\cdot\frac{M}{p}\log\left(\frac{M}{p}\right) + \frac{N}{M}\cdot M\log(p) \\ &= \frac{N}{p}\log\left(\frac{M}{p}\right) + N\log(p) \qquad (1) \end{aligned}$$

During any merge phase, if blocks of size S are being merged p at a time, the complexity per core is $\frac{N}{Sp}\cdot Sp\log(p) = N\log(p)$. There are $\log_p\left(\frac{N}{M}\right)$ merge phases, thus we get

$$\begin{aligned} T_c(\text{merge}) &= N\log(p)\cdot\log_p\left(\frac{N}{M}\right) \\ T_c(\text{total}) &= \frac{N}{p}\log\left(\frac{M}{p}\right) + N\log(p)\left(1+\log_p\left(\frac{N}{M}\right)\right) \end{aligned}$$

Comparison. We can see from the analysis of the two algorithms that algorithm 1 clearly has better time complexity than algorithm 2. However, algorithm 2 is optimal in terms of transfers between L_2 cache and shared main memory. On implementing both the algorithms and comparing performance on our multicore test platforms (see Section 4), we found that algorithm 1 outperforms algorithm 2 across most input instances and sizes. However on future systems, with a greater number cores per chip, this may not hold true. Algorithm analysis using this model captures computational complexity as well as memory performance.

2.2 Experimental Evidence Supporting the Multicore Model

The multicore model suggests that one should design algorithms that are *cache-aware*, and minimize transfer of blocks between cache and main memory. This would enable effective use of the shared L_2 cache and memory bandwidth. A common approach to parallelization on an SMP system is to partition a problem among the various processors in a coarse-grained fashion, such that inter-processor communication (synchronization) is reduced as much as possible.

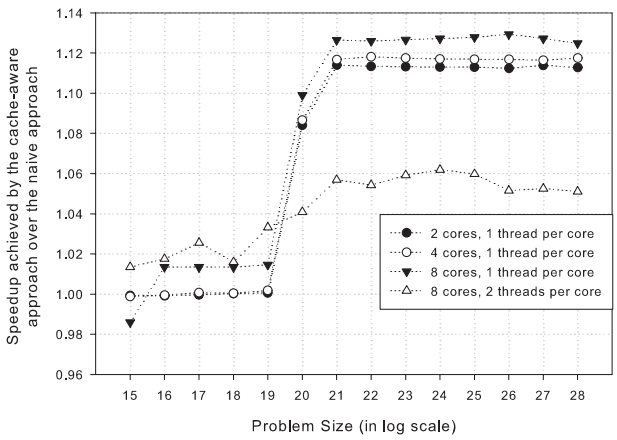


Figure 2. A comparison of the naïve and cache-aware implementations of the benchmark: 2, 4, 8 and 16 threads on the Sun Fire T2000

We would like to demonstrate that this approach may not always lead to good performance on a multicore system, particularly for codes with poor spatial locality.

We use a simple example code to mimic the memory access patterns of a *naïve* and *cache-aware* approach to the divide-and-conquer strategy on multicore systems. The benchmark code reads elements of a large array and performs a fixed number of arithmetic operations on the entire array. Consider a multicore system with m cores, and support for t threads per core (with SMT or hardware multithreading). We can then schedule $p = m * t$ threads concurrently. If there is sufficient bandwidth in the system and no contention for arithmetic and memory units, we should be able to achieve an ideal speedup of p on p threads.

In the naïve approach, the array is split into p equal parts, each thread reads a part of the array and performs some computation on that part of the array. In the cache-aware approach, we read the array in blocks that fit into the L_2 cache, and then all p threads perform operations in parallel on this block. We continue this until the entire array is read in this manner. We also read array elements randomly in both the cases, to mimic poor spatial locality. Note that in both the naïve and cache-aware approaches do equal computational work, and they only differ in their memory access patterns.

In the naïve approach, each thread gets to utilize only a fraction of the L_2 cache, and some array elements may be moved in and out of the cache often. In the cache-aware approach, we have ensured that all the processors in parallel read some part of the array that fits in the L_2 cache. We utilize the L_2 cache and bandwidth better in this approach. To confirm this, we implemented both the approaches on the Sun Fire T2000 server with the UltraSparc T1 processor. With support for up to 32 simultaneous threads of execution, this is the largest multicore platform we could experiment on. Figure 2.2 gives the speedup achieved by the cache-aware approach over the naïve approach, as the problem size is increased from 2^{15} array elements to 2^{28} elements. We run the benchmark on 2, 4, and 8 cores. In all the cases, both approaches work equally well for problem instances smaller than 2^{18} (corresponding to the 3 MB L_2 cache size on the Sun Fire T2000). But for larger problem instances, there is a fixed gap of about 10% between the normalized execution times in all the cases. The execution times differ by 36 seconds for the case of 2^{28} array elements and 2 threads. This benchmark is representative of algorithms with a large memory footprint. The disparity between the two approaches is higher in case of algorithms with low spatial locality. Thus we demonstrate that effectively utilizing the shared memory bandwidth and caching helps in significant performance improvements on multicore systems. We design SWARM primitives and algorithms, optimizing for better cache performance wherever possible.

3 SWARM: Programming Framework for Multicore Systems

SWARM [3] is a portable open-source portable library that provides a programming framework for designing algorithms on multicore systems. Unlike the compiler-based OpenMP approach, we provide a *library-based* parallel programming framework.

A typical SWARM program is structured as follows:

```
int main (int argc, char **argv)
{
    SWARM_Init(&argc, &argv);
    /* sequential code  */
    ....
    ....
    /* parallelize a routine using SWARM */
    SWARM_Run(routine);
    /* more sequential code */
    ....
    ....
    SWARM_Finalize();
}
```

In order to use the SWARM library, the programmer needs to make minimal modifications to existing sequential code. After identifying compute-intensive routines in the program, work can be assigned to each core using an efficient multicore algorithm. Independent operations such as

those arising in *functional parallelism* or *loop parallelism* can be typically threaded. For functional parallelism, this means that each thread acts as a functional process for that task, and for loop parallelism, each thread computes its portion of the computation concurrently. Note that it might be necessary to apply loop transformations to reduce data dependencies between threads.

SWARM contains efficient implementations of commonly-used primitives in parallel programming. We discuss some computation and communication primitives below, along with code snippets:

Data parallel. The SWARM library contains several basic "`pardo`" directives for executing loops concurrently on one or more processing cores. Typically, this is useful when an independent operation is to be applied to every location in an array, for example element-wise addition of two arrays. `Pardo` implicitly partitions the loop among the cores without the need for coordinating overheads such as synchronization of communication between the cores. By default, `pardo` uses block partitioning of the loop assignment values to the threads, which typically results in better cache utilization due to the array locations on left-hand side of the assignment being owned by local caches more often than not. However, SWARM explicitly provides both block and cyclic partitioning interfaces for the pardo directive.

```
/* example: partitioning a "for" loop
   among the cores */
pardo(i, start, end, incr) {
    A[i] = B[i] + C[i];
}
```

Control. SWARM control primitives restrict which threads can participate in the context. For instance, the control may be given to a single thread on each core, all threads on one core, or a particular thread on a particular core.

```
THREADS: total number of execution
threads
MYTHREAD: the rank of a thread,
from 0 to THREADS-1
/* example: execute code on
   thread MYTHREAD */
on_thread(MYTHREAD) {
    ....
    ....
}
/* example: execute code on
   one thread */
on_one_thread {
    ....
    ....
}
```

Memory management. SWARM provides two directives `SWARM_malloc` and `SWARM_free` that, respectively, dynamically allocate a shared structure and release this memory back to the heap.

```
/* example: allocate a shared array
   of size n */
A = (int*)SWARM_malloc(n*sizeof(int),TH);
/* example: free the array A */
SWARM_free(A);
```

barrier. This construct provides a way to synchronize threads running on the different cores.

```
/* parallel code */
    ....
    ....
/* use the SWARM Barrier for
   synchronization */
SWARM_Barrier();
/* more parallel code */
    ....
    ....
```

replicate. This primitive uniquely copies a data buffer for each core.

scan (reduce). This performs a prefix (reduction) operation with a binary associative operator, such as addition, multiplication, maximum, minimum, bitwise-AND, and bitwise-OR. `allreduce` replicates the result from reduce for each core.

```
/* function signatures */
int SWARM_Reduce_i(int myval,
             reduce_t op, THREADED);
double SWARM_Reduce_d(double myval,
             reduce_t op, THREADED);

/* example: compute global sum, using
partial local values from each core */
sum = SWARM_Reduce_d(mySum, SUM, TH);
```

broadcast. This primitive supplies each processing core with the address of the shared buffer by replicating the memory address.

```
/* function signatures */
int SWARM_Bcast_i    (int myval,
```

```
                        THREADED);
int* SWARM_Bcast_ip  (int* myval,
                        THREADED);
char SWARM_Bcast_c   (char myval,
                        THREADED);
```

Apart from the primitives for computation and communication, the thread-safe parallel pseudo-random number generator SPRNG [20] is integrated in SWARM.

4 SWARM: Algorithm Design and Examples

The SWARM library contains a number of techniques to demonstrate key methods for programming on multicore processors.

- The ***prefix-sum*** algorithm is one of the most useful parallel primitives and is at the heart of several other primitives, such as array compaction, sorting, segmented prefix-sums, and broadcasting; it also provides a simple use of balanced binary trees.
- ***Pointer-jumping*** (or path-doubling) iteratively halves distances in a list or graph, thus reducing certain problems in logarithmic time; it is used in numerous parallel graph algorithms, and also as a sampling technique.
- Determining the root for each tree node in a rooted-directed forest is a crucial step in handling equivalence classes—such as detecting whether or not two nodes belong to the same component; when the input is a linked list, this algorithm also solves the parallel prefix problem.
- An entire family of techniques of major importance in parallel algorithms is loosely termed ***divide-and-conquer***—such techniques decompose the instance into smaller pieces, solve these pieces independently (typically through recursion), and then merge the resulting solutions into a solution to the original instance. These techniques are used in sorting, in almost any tree-based problem, in a number of computational geometry problems (finding the closest pair, computing the convex hull, etc.), and are also at the heart of fast transform methods such as the FFT. The `pardo` primitive in SWARM can be used for implementing such a strategy.
- A variation of the above theme is the ***partitioning strategy***, in which one seeks to decompose the problem into independent subproblems—and thus avoid any significant work when recombining solutions; quicksort is a celebrated example, but numerous problems in computational geometry can be solved efficiently with this strategy (particularly problems involving the detection of a particular configuration in 3- or higher-dimensional space).
- Another general technique for designing parallel algorithms is ***pipelining***. In this approach, waves of concurrent (independent) work are employed to achieve optimality.

Built on these techniques, SWARM contains a higher-level library of multicore optimized parallel algorithms for list ranking, comparison-based sorting, radix sort and spanning tree. In addition, SWARM application example codes include efficient implementations for solving combinatorial problems such as minimum spanning tree [9], graph decomposition [6], breadth-first-search [7], tree contraction [8] and maximum parsimony [5].

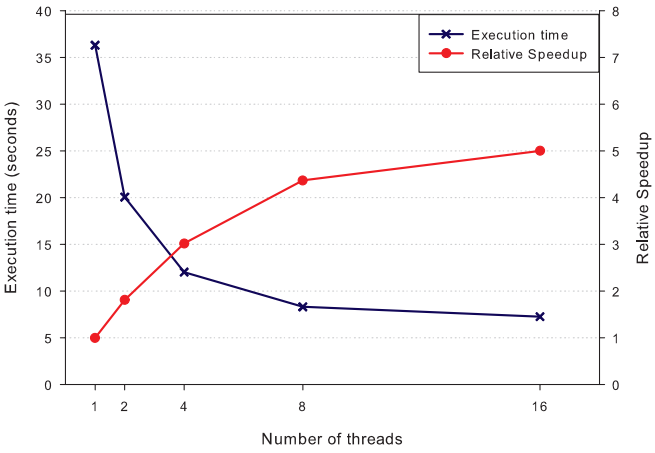
4.1 Performance

We will next discuss parallel performance of a few examples implemented using the SWARM framework – merge sort, radix sort and list ranking – on current multicore systems. Our test platforms are a Sun Fire T2000 server and a dual-core Intel system.

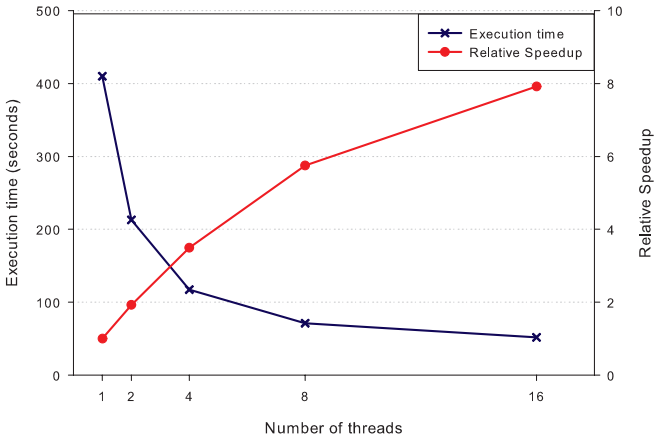
The Sun Fire T2000 server, with the Sun UltraSPARC T1 (Niagara) processor, has eight cores running at 1.0 GHz, each of which is four-way multithreaded. There are eight integer units with a six-stage pipeline on-chip, and four threads running on a core share the pipeline. The cores share 3 MB of L_2 cache, and the system has 6 GB main memory. Since there is only one floating point unit (FPU) for all cores, the Ultra Sparc T1 processor is mainly suited for programs with few or no floating point operations. We compile our codes with the Sun C compiler v5.8 with `-xO2` flag.

The list ranking example was also run on a 2.8 GHz dual-core Intel Xeon processor, with 2 MB L_2 cache and 2 GB main memory. Both the cores support hyper-threading, which gives an impression of four virtual processors. The code is compiled with `icc v9.0` and flags (`-Wall -O2`).

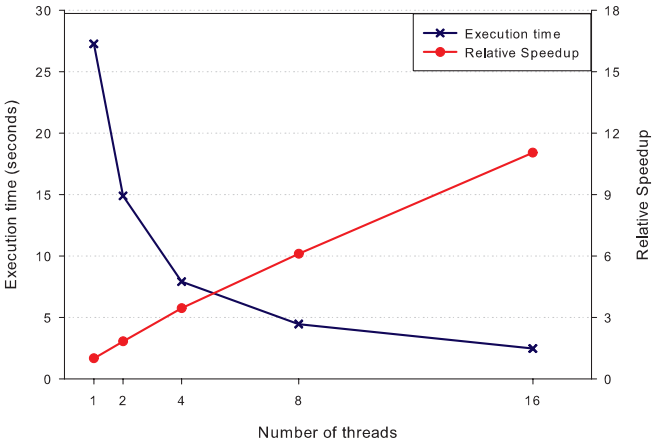
In each case, we run the codes five times and report the average running time. Figures 3(a) and 3(b) plot the execution time and speedup for radix sort and merge sort respectively on the Sun Fire T2000. The input to both the sorting algorithms is a list of 2^{27} randomly ordered integers in [0, 2^{27}). The radix sort algorithm is one order of magnitude faster than merge sort on one thread, but merge sort scales better up to 16 threads. We achieve a speedup of 4.5 for radix sort and 6 for merge sort, on 8 threads. There is little or no speedup with more than 16 threads on the Sun Fire T2000. Figure 3(c) gives the performance of a parallel list ranking algorithm on the Sun Fire T2000 for a random list of size 2^{26}. This particular list ranking instance has poor



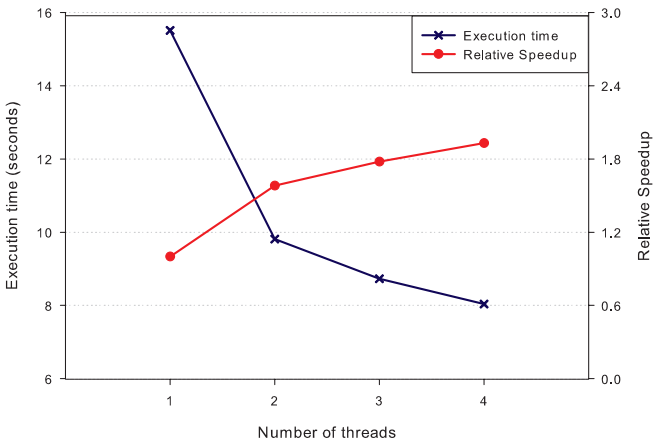
(a) Radix Sort Performance (Sun Fire T2000)



(b) Merge Sort Performance (Sun Fire T2000)



(c) List Ranking Performance (Sun Fire T2000)



(d) List Ranking Performance (Intel Xeon dual-core)

Figure 3. Performance of algorithms implemented in SWARM on the Sun Fire T2000 (3(a), 3(b) and 3(c)) and an Intel Xeon dual-core processor(3(d))

spatial and temporal locality with very little computation, but the execution time scales better with the number of processors than the sorting algorithms. This is because nearly all memory accesses result in cache misses, and latency to main memory decides the running time. The latency can be tolerated to some extent by scheduling concurrent threads. Figure 3(d) plots the execution time and speed-up for the same instance on the dual-core Intel system. We achieve a modest speedup of 2 with four threads in this case. In general, we achieve lower relative speedup on multicore systems in comparison to our previous results on symmetric multiprocessors.

5 Conclusions and Future Work

Our primary contribution in this paper is the design of SWARM, a portable, open-source library for developing efficient multicore algorithms. We have already implemented several important parallel primitives and algorithms using this framework. In future, we intend to add to the functionality of basic primitives in SWARM, as well as build more multicore applications using this library. The second contribution of this paper is a new computational model for the design and analysis of multicore algorithms. While designing algorithms for multicore systems, the cache performance, limited memory bandwidth, and synchronization overhead should also be considered along with time complexity.

References

[1] A. Aggarwal and J. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31:1116–1127, 1988.

[2] AMD Multi-Core Products. `http://multicore.amd.com/en/Products/`, 2006.

[3] D. Bader. SWARM: A parallel programming framework for multicore processors. `https://sourceforge.net/projects/multicore-swarm`, 2006.

[4] D. Bader, V. Agarwal, and K. Madduri. On the design and analysis of irregular algorithms on the Cell processor: A case study of list ranking. In *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS 2007)*, Long Beach, CA, USA, 2007.

[5] D. Bader, V. Chandu, and M. Yan. ExactMP: An efficient parallel exact solver for phylogenetic tree reconstruction using maximum parsimony. In *Proc. 35th Int'l Conf. on Parallel Processing (ICPP)*, Columbus, OH, Aug. 2006.

[6] D. Bader, A. Illendula, B. M. Moret, and N. Weisse-Bernstein. Using PRAM algorithms on a uniform-memory-access shared-memory architecture. In G. Brodal, D. Frigioni, and A. Marchetti-Spaccamela, editors, *Proc. 5th Int'l Workshop on Algorithm Engineering (WAE 2001)*, volume 2141 of *Lecture Notes in Computer Science*, pages 129–144, Århus, Denmark, 2001. Springer-Verlag.

[7] D. Bader and K. Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2. In *Proc. 35th Int'l Conf. on Parallel Processing (ICPP)*, Columbus, OH, Aug. 2006. IEEE Computer Society.

[8] D. Bader, S. Sreshta, and N. Weisse-Bernstein. Evaluating arithmetic expressions using tree contraction: A fast and scalable parallel implementation for symmetric multiprocessors (SMPs). In S. Sahni, V. Prasanna, and U. Shukla, editors, *Proc. 9th Int'l Conf. on High Performance Computing (HiPC 2002)*, volume 2552 of *Lecture Notes in Computer Science*, pages 63–75, Bangalore, India, Dec. 2002. Springer-Verlag.

[9] D. A. Bader and G. Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). In *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS 2004)*, Santa Fe, NM, Apr. 2004.

[10] D. A. Bader and J. JáJá. SIMPLE: A methodology for programming high performance algorithms on clusters of symmetric multiprocessors (SMPs). *Journal of Parallel and Distributed Computing*, 58(1):92–108, 1999.

[11] L. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A scalable architecture based on single-chip multiprocessing. *SIGARCH Comput. Archit. News*, 28(2):282–293, 2000.

[12] J. Held, J. Bautista, and S. Koehl. From a Few Cores to Many: A Tera-scale Computing Research Overview. `ftp://download.intel.com/research/platform/terascale/terascale_overview_paper.pdf`, 2006.

[13] D. R. Helman and J. JáJá. Designing practical efficient algorithms for symmetric multiprocessors. In *Algorithm Engineering and Experimentation (ALENEX'99)*, volume 1619 of *Lecture Notes in Computer Science*, pages 37–56, Baltimore, MD, Jan. 1999. Springer-Verlag.

[14] Multi-Core from Intel – Products and Platforms. `http://www.intel.com/multi-core/products.htm`, 2006.

[15] International Technology Roadmap for Semiconductors, 2004 update. `http://itrs.net`, 2004.

[16] J. Kahle, M. Day, H. Hofstee, C. Johns, T. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, 2005.

[17] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded Sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[18] R. Ladner, J. Fix, and A. LaMarca. The cache performance of traversals and random accesses. In *Proc. 10th Ann. Symp. Discrete Algorithms (SODA-99)*, pages 613–622, Baltimore, MD, 1999. ACM-SIAM.

[19] R. E. Ladner, R. Fortna, and B.-H. Nguyen. A comparison of cache aware and cache oblivious static search trees using program instrumentation. In R. Fleischer, E. Meineche-Schmidt, and B. Moret, editors, *Experimental Algorithmics*, volume 2547 of *Lecture Notes in Computer Science*, pages 78–92. Springer-Verlag, 2002.

[20] M. Mascagni and A. Srinivasan. Algorithm 806: SPRNG: a scalable library for pseudorandom number generation. *ACM Trans. Math. Softw.*, 26(3):436–461, 2000.