An Empirical Analysis of Parallel Random Permutation Algorithms on SMPs

Guojing Cong
IBM T.J. Watson Research Center

David A. Bader*
College of Computing
Georgia Institute of Technology

February 25, 2006

Abstract

We compare parallel algorithms for random permutation generation on symmetric multiprocessors (SMPs). Algorithms considered are the sorting-based algorithm, Anderson's shuffling algorithm, the dart-throwing algorithm, and Sanders' algorithm. We investigate the impact of synchronization method, memory access pattern, cost of generating random numbers and other parameters on the performance of the algorithms. Within the range of inputs used and processors employed, Anderson's algorithm is preferable due to its simplicity when random number generation is relatively costly, while Sanders' algorithm has superior performance due to good cache performance when a fast random number generator is available. There is no definite winner across all settings. In fact we predict our new dart-throwing algorithm performs best when synchronization among processors becomes costly and memory access is relatively fast.

We also compare the performance of our parallel implementations with the sequential implementation. It is unclear without extensive experimental studies whether fast parallel algorithms beat efficient sequential algorithms due to mismatch between model and architecture. Our implementations achieve speedups up to 6 with 12 processors on the Sun E4500.

1 Introduction

Random permutation generation is a fundamental problem in computer science. It is particularly useful in designing randomized algorithms. One popular usage is to perturb the input so that worst-

*This work was supported in part by NSF Grants CAREER ACI-00-93039, NSF DBI-0420513, ITR ACI-00-81404, DEB-99-10123, ITR EIA-01-21377, Biocomplexity DEB-01-20709, and ITR EF/BIO 03-31654; and DARPA Contract NBCH30390004.

case scenarios occur with minuscule probability. Formally, the random permutation problem is to generate permutations on $1,\ldots,n$ and each permutation has probability $\frac{1}{n!}$ of being generated.

The classic sequential shuffling algorithm by Knuth [9] runs in linear time with a very small constant factor. There are several fast parallel algorithms in the literature. Reif [17] presents a randomized work-time optimal CRCW PRAM algorithm based on integer sorting. Dart-throwing is another popular technique for random permutation on CRCW PRAM [12, 16, 14, 7, 5]. Czmuaj *et al.* [4] present an $O(\log\log n)$ time algorithm on EREW PRAM that uses $O(n^{1+o(n)})$ processors. In addition to PRAM algorithms, researchers also study the problem on more realistic models. Anderson [1] proposed an algorithm similar to the sequential algorithm for small shared-memory parallel computers. With p processors the algorithm achieves a theoretic speedup of almost p. Sanders [18] considered the problem on distributed, hierarchical, and external memory settings, and his algorithm runs in time $O(n/p+T_{comm}(n/p,p)+T_{prefix}(p))$ on p processors w.h.p, where $T_{comm}(k,p)$ is the time for randomly sending or receiving k elements on each processor and $T_{prefix}(p)$ is the time for computing a prefix sum. Lassous and Thierry [10] present algorithms for the CGM model with a small number of supersteps and low communication cost.

In spite of fast theoretic algorithms, there have been few studies that investigate various implementation choices when adapting the algorithms designed for theoretic models to current architectures. Bridging the gap between models and real architectures to bring maximum performance is not a trivial matter. The few prior experimental studies either argue in favor of a specific model [5] or focus on the performance of one certain algorithm [18]. No extensive results are available for comparing the performance of the algorithms. These algorithms have different features regarding parallel algorithmic overhead, memory access behavior, synchronization, and scalability. It is

hard to predict the performance of the algorithms solely from complexity analysis, and experimental studies should bring deeper insight into the comparison of the algorithms. More importantly, prior studies (both theoretic and experimental) largely ignore the impact of pseudo-random number generators on the performance. In our study we find that the choice of random number generator affects our decision on which permutation algorithm gives highest performance.

We implement four parallel algorithms. They are the sorting-based algorithm [17] (denoted as *Rand_Sort*), the dart-throwing algorithm [5, 6] (denoted as *Rand_Dart*), the shuffling algorithm [1] (denoted as *Rand_Shuffle*), and Sanders' algorithm for the distributed-memory setting [18] (denoted as *Rand_Dist*). When implementing these algorithms, we evaluate different design choices, e.g., mutex-lock vs. spin-lock synchronization, the trade-off between cache-friendly design and design that requires fewer random numbers. Three parallel pseudo-random number generators are used to study their impact: the Scalable Parallel Random Number Generator (SPRNG) [13], a linear congruential generator (LCG) [11], and TWISTER– a fast generator that is frequently used in Monte Carlo simulations [15].

Our target architecture is symmetric multiprocessor (SMP). We test our implementations on Sun E4500, a uniform-memory-access (UMA) shared memory parallel machine with 14 UltraSPARC II 400MHz processors and 14 GB of memory. Each processor has 16 Kbytes of direct-mapped data (L1) cache and 4 Mbytes of external (L2) cache. We implement the algorithms using POSIX threads and software-based barriers [2].

The rest of the paper is organized as follows. Section 2 introduces the algorithms and presents in detail the tuning of the algorithms for performance optimization; Section 3 gives the performance results and analysis; Section 4 is our conclusion and future work.

2 Algorithms and Implementations

Of the four random permutation algorithms, *Rand_Sort* and *Rand_Dart* are PRAM algorithms, while *Rand_Shuffle* and *Rand_Dist* are based on more realistic models that acknowledge the fact that for most parallel architectures only a limited number of processors are available. *Rand_Shuffle* and *Rand_Dist* are closely related to the sequential shuffling algorithm in that shuffling is employed to generate permutations. The design of *Rand_Sort* and *Rand_Dart* are very different from the sequential approaches. The four algorithms have different performance properties regarding parallel overhead, memory access pattern, synchronization cost, and the number of random bits needed (as random bits can be a costly resource). Oftentimes optimizations for these aspects are mutually exclusive to each other, e.g., reducing the number of random bits needed may be in direct conflict of reducing the amount of synchronization. Experimental studies bring deeper understanding of the algorithms and their relationship to each other so that combined with asymptotic analysis we can extrapolate with confidence for inputs that are not tested. The insight gained from experimental studies also helps the design of new efficient algorithms. We next introduce the algorithms and discuss the pros and cons of different design choices. Section 2.1 briefly introduces the implementation of *Rand_Sort*; section 2.2 gives our new "dart-throw" algorithm that is twice as fast as the popular "dart-throw" algorithm in the literature; section 2.3 presents the implementation of *Rand_Shuffle* using spinlocks for synchronization; and section 2.4 presents our cache-friendly implementation of *Rand_Dist*.

2.1 *Rand_Sort*

Rand_Sort starts by generating for each element a random number between 1 and $O(n)$. The generated sequence is then sorted. *Rand_Sort* produces a random permutation by moving each element to its corresponding sorted location. Implementing *Rand_Sort* on SMPs is simple if fast sorting routines are available. Sorting may be too heavy a machinery to use for the random permutation problem. In addition to moving elements, sorting may incur overhead of multiple rounds of comparisons. Because of that overhead, it is unlikely that *Rand_Sort* beats other algorithms we consider. However, as sorting is arguably one of the most studied problems in computer science, on architectures with hardware dedicated to sorting, *Rand_Sort* can have superior performance. In our implementations, we use Helman and JáJá's cache-friendly parallel sample sort implementation [8].

2.2 *Rand_Dart*

The generic "dart-throwing" algorithm has two steps. First n elements are placed at random into an array with size linear in n. Then the array is compacted and the relative order of the elements gives an implicit random permutation.

There are several flavors of the"dart-throwing" algorithm. They differ slightly in the way darts are thrown and how the array is compacted. In this paper we consider the algorithm described in [5] that is adapted from a randomized CRCW "processor allocation" algorithm by Gil [6]. For $c \log\log n$ ($c \geq 1$) rounds, each unplaced element selects a random slot from a contiguous region of an array A. If a collision occurs (i.e., multiple elements try to claim the same slot), the placement fails and waits for the next round. The size of the region is $d \cdot n$ (d is a constant) in the first round,

and decreases by half at each round. The algorithm is re-run if after $c \log\log n$ rounds there are still unplaced elements. The algorithm runs w.h.p. in $O(\log n)$ time with linear work on both CRCW and QRQW PRAM [5]. This "dart-throwing" algorithm is denoted as *Rand_Dart_G* in this paper.

Rand_Dart_G requires the detection of collision among processors for dart-throwing. This usually involves two phases. First the processors trying to claim a certain slot write their *id*s into a tag associated with the cell. After synchronization, the processors then check to see if there was a collision in cells that they tried to claim by comparing the tag with their *id*s. If the values are different, then a collision is detected. Collision detection can be costly especially when the input is large, due to the added cost of more memory accesses. We propose a new "dart-throwing" algorithm (*Rand_Dart_CB*) that obviates the need for collision detection. In fact, the algorithm uses mutual exclusion to prevent collision. The basic idea of the algorithm is as follows. When an element is to be placed at a certain slot *s*, the processor first gains ownership of *s* through some mutual exclusion scheme. The element is placed at *s* if no other processor has claimed it, otherwise the processor chooses another random slot until it finds an empty slot. Next we show that upon termination the algorithm generates each possible permutation with probability $\frac{1}{n!}$. First we consider throwing n darts (elements) into $m = \alpha n$ ($\alpha > 1$) slots sequentially.

Lemma 1 *Let $\pi = \pi_1, \pi_2, \cdots, \pi_n$ be any permutation of $\{1, 2, \ldots, n\}$. Consider placing $\pi_1, \pi_2, \ldots, \pi_n$ one by one at random into an array A of m slots. In case of a collision, the element is placed again at a random slot of A until it lands in an empty slot. After all elements are placed, the relative order of the elements defines a random permutation. Also each permutation is generated with probability.*

Proof: We define the *identity permutation* as $1, 2, \ldots, n$. We define a *configuration* to be the

mapping of the n elements, with at most one element per slot, are placed into array A. With a *configuration* we do not differentiate between elements. We first show that for any *configuration* κ, each possible permutation is generated with probability $1/n!$. That is, we show $Pr(\hat{\pi} = \pi' | \kappa) = 1/n!$, where $\hat{\pi}$ is the resulting permutation, and $\pi' = \pi'_1, \pi'_2, \ldots, \pi'_n$ is any possible permutation of the *identity permutation*. Let $i_1, i_2, \ldots, i_n$ be the indices for elements $\pi_1, \pi_2, \ldots, \pi_n$ in π', respectively.

$$\begin{aligned} Pr(\hat{\pi}_{i_1} = \pi'_{i_1} | \kappa) &= Pr(\hat{\pi}_{i_1} = \pi_1 | \kappa) \\ &= \frac{Pr(\hat{\pi}_{i_1} = \pi_1 \wedge \kappa)}{Pr(\kappa)} \\ &= \frac{1}{n} \end{aligned}$$

Using the above equation as the base case, we prove by induction that

$$Pr(\hat{\pi} = \pi' | \kappa) = \frac{1}{n!}.$$

As the induction step, assume

$$Pr\left(\bigwedge_{j=1}^{k} \hat{\pi}_{i_j} = \pi_j \middle| \kappa \right) = \frac{1}{n(n-1)\cdots(n-k+1)}, \; k \geq 1.$$

For $k+1$, let ε be the event $\bigwedge_{j=1}^{k}(\hat{\pi}_{i_j} = \pi_j)$, η be the event $(\hat{\pi}_{k+1} = \pi_{k+1})$,

$$\begin{aligned} Pr\left(\bigwedge_{j=1}^{k+1} (\hat{\pi}_{i_j} = \pi_j) \middle| \kappa \right) &= Pr(\varepsilon \wedge \eta | \kappa) \\ &= \frac{Pr(\eta | \varepsilon \wedge \kappa) Pr(\varepsilon \wedge \kappa)}{Pr(\kappa)} \end{aligned}$$

$$
\begin{aligned}
&= Pr(\eta|\varepsilon \wedge \kappa) Pr(\varepsilon|\kappa) \\
&= \frac{1}{n-k} \frac{1}{n(n-1)\cdots(n-k+1)} \\
&= \frac{1}{n(n-1)\cdots(n-(k+1)+1)}
\end{aligned}
$$

So $Pr(\hat{\pi} = \pi'|\kappa) = \frac{1}{n!}$. It follows that each random permutation is generated with probability $\frac{1}{n!}$. $\square$

Theorem 1 *The parallel algorithm with p processors generates each possible random permutation with probability* $\frac{1}{n!}$. *Furthermore, the expected number of dart-throws used by our algorithm is* $\frac{\alpha}{\alpha-1}n$.

Proof: We can impose an imaginary serialized order on the throws from different processors. The throws that do not cause collisions can be serialized arbitrarily with regard to each other. For throws that collide, we arbitrate and impose an order by using atomic operations for claiming the slots, so that the "dart-throws" from the parallel algorithm can be viewed thrown by a sequential algorithm. Note that throws from different processors are independent. By lemma 1 the parallel algorithm generates each random permutation with probability $\frac{1}{n!}$.

Let random variable X be the number of throws needed to generate a permutation. Let X_i be the number of throws it takes to get the $(i+1)^{th}$ new slot after we get the i^{th} new slot.

$$
E(X) = E\left(\sum_{i=1}^{n-1} X_i\right) = 1 + \frac{m}{m-1} + \frac{m}{m-2} + \cdots + \frac{m}{m-n+1}
$$

Note that $\frac{m}{m-k} < \frac{m}{m-n+1}$ for $1 \leq k < n-1$, and the above simplifies into:

$$E(X) < n\frac{m}{m-n+1} \approx \frac{\alpha}{\alpha-1}n$$

As the X_i's are independent, using Chernoff bound we can further show that

$$Pr(X > (1+\delta E(X))) < e^{-\frac{n\delta^2}{4}} = o(1), 0 < \delta \leq 2e-1.$$

□

Fig. 1 compares the performance of the two "dart-throwing" algorithms we implemented. *Rand_Dart_G* implements the algorithm described in [5], while *Rand_Dart_CB* is our algorithm using atomic operations (in this case, *compare&swap*) to claim slots. For the input size of 20M integers, *Rand_Dart_CB* is roughly twice as fast as *Rand_Dart_G* with different number of processors. Similar results are also observed for input sizes that fit in cache, e.g., 1M integers.

The function $\frac{\alpha}{\alpha-1}$ ($\alpha > 1$) decreases as α increases. Larger α reduces the expected number of "throws", hence reduces the cost of generating random numbers. When α gets large enough, there is essentially no conflict among processors. Yet the performance advantage of increasing α is very likely to be offset by poor memory access behavior that comes with the larger array size.

For the remainder of this paper, we use *Rand_Dart_CB* as our *Rand_Dart* implementation.

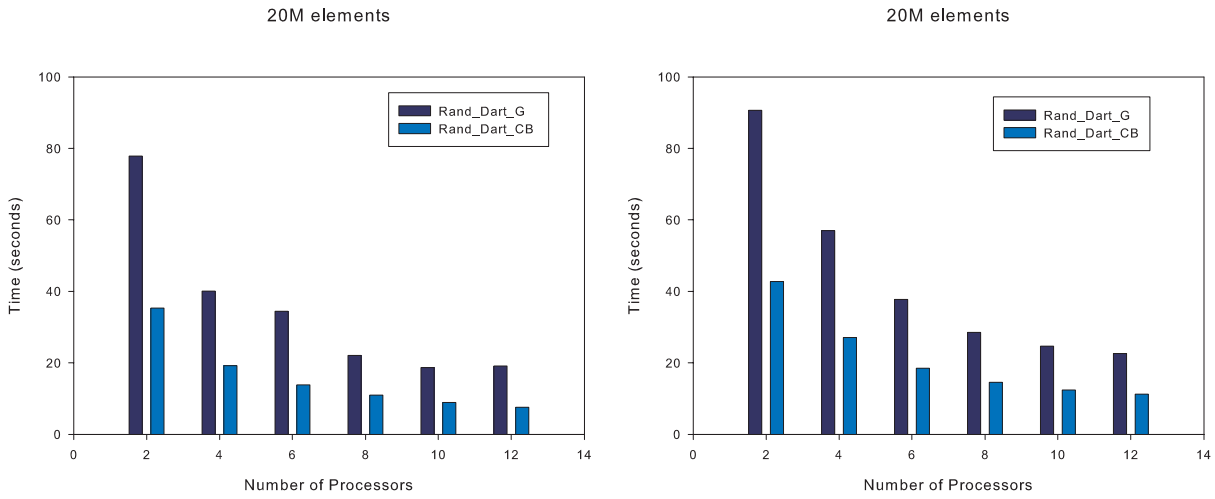

Figure 1: Comparison of the performance of *Rand_Dart_G* and *Rand_Dart_CB* with different number of processors. The input size is 20M integers. For the left plot, SPRNG is used. LCG is used in the algorithms in the right plot. *Rand_Dart_CB* uses $\alpha = 2$.

2.3 *Rand_Shuffle*

Rand_Shuffle solves the random permutation problem on "small" parallel computers, i.e., the number of the processors is much smaller than the problem size. *Rand_Shuffle* is similar to Knuth's sequential algorithm. The sequential algorithm conducts a sequence of swapping operations between element j and a random element s with $j \leq s \leq n$, $1 \leq j \leq n-1$. Anderson [1] proves that the order in which the swapping operations occur does not affect the distribution of the permutations. Assuming no conflicts among processors, i.e., no two processors attempt to shuffle the same element simultaneously, with Anderson's theorem, we can divide the swapping operations among processors arbitrarily. Intuitively, *Rand_Shuffle* runs p copies of Knuth's algorithm in parallel.

Implementing *Rand_Shuffle* on SMPs is straightforward. Anderson suggests using locks to resolve conflicts among processors. System mutex locks incur large overhead and are not scalable for large data sets. Instead we use spinlocks implemented through atomic operations (e.g., *compare&swap*, and *load-link, store-conditional*). Our prior studies [3] with spanning tree and

minimum spanning tree algorithms show that when the work inside a critical section can be performed relatively fast (that is, as quickly as the time it takes other processors to spin a few times) such as in the cases of incrementing a shared location or of setting a flag, we can use spinlocks that are much simpler than system mutex locks and avoid large overhead.

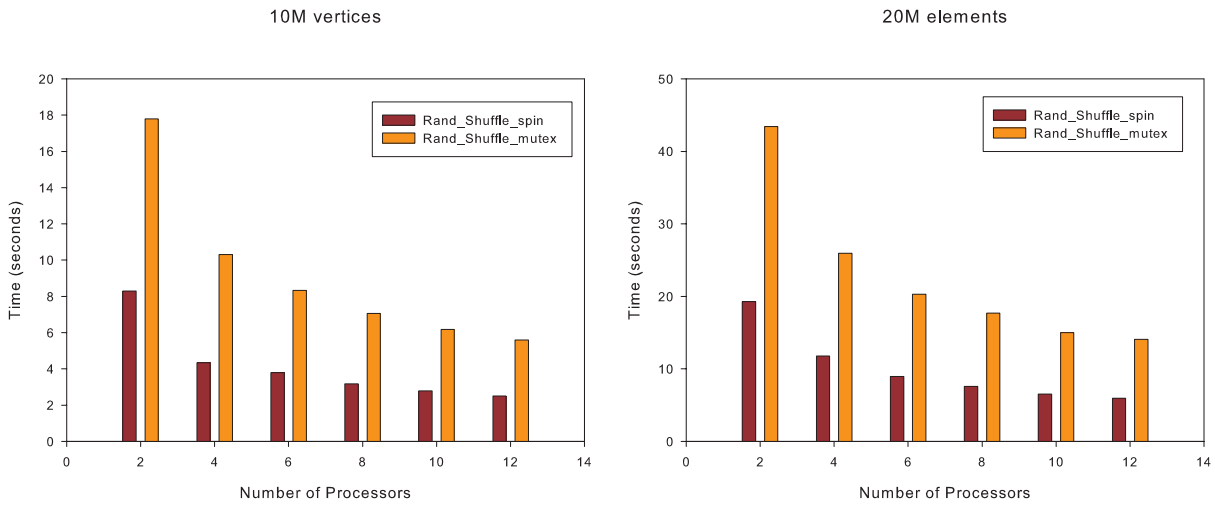

Figure 2: Performance comparison of *Rand_Shuffle_spin* and *Rand_Shuffle_mutex* for different input sizes. The random number generator used is SPRNG.

Fig. 2 shows the performance comparison between two implementations of the shuffling algorithm. *Rand_Shuffle_mutex* uses system mutex locks, while *Rand_Shuffle_spin* uses spinlocks. *Rand_Shuffle_spin* is much faster than *Rand_Shuffle_mutex*. The random number generator used is SPRNG. Similar results are observed for implementations with other random number generators. Hence, we use *Rand_Shuffle_spin* as our implementation for *Rand_Shuffle* in the remainder of this paper.

The performance advantage of *Rand_Shuffle* is obvious for shared-memory parallel computers with a moderate number of processors due to its simplicity. *Rand_Shuffle* is much simpler than *Rand_Sort*. It is interesting to compare *Rand_Shuffle* with *Rand_Dart*. Both algorithms employ some kind of atomic operations to resolve conflicts among processors, and have random memory

access patterns. *Rand_Dart* generates expected $\frac{\alpha}{\alpha-1}n$ random numbers while *Rand_Shuffle* generates exactly $n-1$ random numbers. It seems *Rand_Dart* has larger overhead for reasonable choices of α and *Rand_Shuffle* should perform better than *Rand_Dart*. While this is generally true, the disadvantage of *Rand_Shuffle* over *Rand_Dart*, however, is that the number of conflicts increase with the number of processors. For each element, *Rand_Shuffle* "locks" the element itself and some other element in order to swap. Synchronization causes more overhead when more processors compete for locks. In the extreme case with n processors available, for each swap operation there is a conflict for *Rand_Shuffle*.

2.4 *Rand_Dist*

Rand_Dist considers the problem in the distributed, external and hierarchical memory settings. The algorithm starts with p processors with each one containing n/p elements. Each processor randomly chooses a destination processor for each element, and sends the element to that processor. After receiving the incoming elements, each processor computes a local random permutation. The last step is to concatenate the local permutations into a global one. Sanders [18] shows that the algorithm runs in time $O(n/p + T_{comm}(n/p, p) + T_{prefix}(p))$ on p processors w.h.p, where $T_{comm}(k, p)$ is the time for randomly sending or receiving k elements on each processor and $T_{prefix}(p)$ is the time for computing a prefix sum. This algorithm can be classified as using the divide-and-conquer approach according to Hagerup [7]. Actually one flavor of the sorting-based PRAM algorithm [17] has similar design: each of the elements $1, \cdots, n$ chooses a random *key* and the elements are grouped (sorted) according to the *key*; each group is then sequentially permuted. Sanders [18] is the first to study the impact of memory hierarchy on permutation algorithms with

modern architectures.

Sanders implemented the algorithm on the MasPar MP-1, and observed that the algorithm runs faster than the QRQW dart-throwing algorithm. The main motivation of *Rand_Dist*, however, was that a faster sequential algorithm on machines with a hierarchical memory system can be derived by simulating p processors with a single processor.

We adapt *Rand_Dist* to run on SMPs, and give a parallel implementation that coordinates "send" and "receive" for better cache performance. Alg. 1 describes our detailed implementation of *Rand_Dist* so that we can analyze the cache performance of the algorithm. In Alg. 1, a random destination processor is first generated for each element. Each processor then computes the number of elements to send to and receive from other processors. Each processor copies its elements to appropriate destination locations, and permutes its received elements. The simulated "send" and "receive" on an SMP take place in a coordinated manner, i.e., our implementation guarantees that elements "sent" from one processor to the same destination be stored in consecutive locations, which may not be the case in the distributed-memory setting. The coordinated "send" and "receive" do not skew the distribution of the random permutations. To see this, note that the distribution of elements sent to each processor is not changed, and only the ordering of the received elements is affected. As these elements are then locally randomly permuted, the probability of generating any certain sequence is the same no matter how the initial sequence is ordered. The coordinated "send" and "receive", however, do make a difference in the memory access pattern of the implementation.

In Alg. 1, steps 1 and 2 incur only contiguous memory accesses. Step 3 computes values for several small auxiliary data structures. As these data structures are of size $O(p^2)$ that fit in cache

Input: 1) Shared array A with n elements
2) Processor P_i $(0 \leq i \leq p-1)$

Output: Array B that is a random permutation of A

1. generate a random destination processor P_j for each element;

for $\frac{in}{p} \leq k \leq \frac{(i+1)n}{p} - 1$ **do**
$dest[k] \leftarrow rand(0, p-1)$;

2. compute *snd_cnt* (shared $p \times p$ array), the number of elements to send to each processor;

for $\frac{in}{p} \leq k \leq \frac{(i+1)n}{p} - 1$ **do**
$snd\_cnt[i][dest[k]]$ ++;

3. with *snd_cnt*, compute the following:
the number of elements to receive from each processor, the total number of elements received, and finally the starting location (*start_offset*) for elements that are routed to each processor.;

4. copy elements to appropriate locations; $local[i]$ is temporary storage for processor i and *cntr* is a one dimension array with size p;

for $\frac{in}{p} \leq k \leq \frac{(i+1)n}{p} - 1$ **do**
$offset \leftarrow start\_offset[dest[k]][i]$;
$local[dest[k]][cntr[dest[k]]+offset] \leftarrow A[k]$;
$cntr[dest[k]]$++;

5. run sequential algorithm on $local[i]$ and concatenate them to get B;

Algorithm 1: Algorithm for processor P_i $(0 \leq i \leq p-1)$ for generating a random permutation using *Rand_Dist*. *snd_cnt* and *cntr* are initialized to 0. The function $rand(i, j)$ generates an integer uniformly at random between i and j inclusive.

and the computation is simple, step 3 does not have significant impact on the overall performance of the algorithm, and hence we do not show the details. In step 4 each processor copies its local elements into appropriate destination locations. As consecutive elements may be routed to different processors (hence locations far away from each other in memory), at first glance, step 4 seems to incur a lot of cache misses. However, as our implementation decrees that elements sent to the same processor from the same source be copied to consecutive locations, there are only a few choices (in fact, p choices) with regard to where an element may end up. Furthermore, if elements A_j and A_k are sent to the same destination processor by one source processor in successive order, A_j will be stored right next to A_k. As long as the cache on each processor is able to hold the p cache

lines without conflicts, most cache misses incurred in step 4 are compulsory misses. Step 5 runs p copies of Knuth's algorithm on smaller scales. As the two elements in each swap operation are from an array segment (i.e., the elements received by a certain processor) that is much smaller than the whole range of the array, a larger portion of the memory accesses are cache hits compared with Knuth's algorithm.

We can further reduce cache misses in step 5 by simulating $p' > p$ virtual processors with p physical processors until each *local* fits in cache. Larger p' generally means smaller *slot* size, however, the choice of p' is not without restriction. To achieve good load balance, we want p to evenly divide p' so that each physical processor is assigned relatively same amount of work.

3 Performance Results and Analysis

This section summarizes the experimental results of our implementation. Figs. 3 and 4 show the performance comparison of *Rand_Shuffle*, *Rand_Dart*, *Rand_Sort* and *Rand_Dist* with different random number generators for various input sizes. In our implementation, we set $\alpha = 2$ for *Rand_Dart*, and the number of virtual processors adapts with the problem size and the number of physical processors available so that the expected size of slot for each processor fits in cache.

First notice that in all plots, *Rand_Sort* is inferior to other algorithms in performance. For different input sizes, *Rand_Sort* is about 2-4 times slower than the fastest implementation. The performance of *Rand_Shuffle* is best if LCG is used as the random number generator. LCG with a small period is normally very fast. As we are dealing with large input sizes, the LCG we use is much slower than other random number generators as it generates random numbers with large enough periods. Among all algorithms, *Rand_Shuffle* requires the fewest random bits. When randomness

becomes an expensive resource, *Rand_Shuffle* is usually the better choice. *Rand_Dist* performs best if fast pseudo-random number generators, e.g., SPRNG and TWISTER, are used. Compared with *Rand_Dart* and *Rand_Shuffle*, *Rand_Dist* is more cache-aware. The disadvantage of *Rand_Dist* is the need to generate more random numbers than both *Rand_Shuffle* and *Rand_Dart*. *Rand_Dist* requires $2n$ random numbers while *Rand_Dart* and *Rand_Shuffle* require $\frac{\alpha}{\alpha-1}n$ (expected) and $n-1$ random numbers, respectively. The cost of generating random numbers, in addition to cache scheme, is a significant factor in determining the performance ranking of algorithms and the tuning of parameters. When randomness is a costly resource, it dominates performance and overshadows the importance of cache performance. Another important factor to performance is synchronization cost. If an algorithm requires frequent synchronization among processors, it is not likely to scale well. Although within the range of processors available, *Rand_Shuffle* consistently beats *Rand_Dart*, we predict a cross point in the performance curve when more processors are available. In Fig. 3 we can see the trend. With more processors, the performance of *Rand_Dart* gets closer to that of *Rand_Shuffle*.

Figs. 3 and 4 also show the performance of the sequential random permutation algorithm. Here we implement Knuth's shuffling algorithm. Sanders [18] reported that the simulation of *Rand_Dist* for multiple processors with one physical processor actually runs faster than Knuth's sequential algorithm due to better cache performance of *Rand_Dist*. The speedup is up to 2 on UltraSparc processor and 4 on MIPS 10000. We reproduced his results with his code on UltraSparc, however, our own implementation of *Rand_Dist* on one processor (that simulates multiple processors) does not run faster than Knuth's algorithm. This is largely due to the difference in the cost of generating random numbers. Sanders' own simple random number generator is much faster than that we use

and may have a smaller period. A pseudo-random number generator with a small period works reasonably well with *Rand_Shuffle*, *Rand_Sort* and *Rand_Dist* (they generate a permutation although the distribution may be highly skewed), but poses serious problem for *Rand_Dart*. *Rand_Dart* falls into an infinite loop as darts start to fall consistently into claimed slots when the random numbers generated wrap around. If choices of random number generators are limited and only generators with small periods are available, *Rand_Sort*, *Rand_Shuffle* and *Rand_Dist* are preferable. With our random number generators and our target architecture, the shuffling algorithm is the fastest. While the sequential shuffling algorithm is faster than simulating *Rand_Dist* with one processor, the parallel version of the shuffling algorithm (*Rand_Shuffle*) is slower than *Rand_Dist* when fast parallel random number generators (SPRNG and TWISTER) are used. This is because of the parallel overhead incurred by *Rand_Shuffle* over the sequential algorithm. In addition to synchronization, *Rand_Shuffle* uses at least twice as much memory usage (array of locks) and thrice as many memory accesses (locking and unlocking) as that of the sequential algorithm. Our fastest parallel implementation (*Rand_Shuffle* or *Rand_Dist*) achieve speedups around 5-6 with 12 processors.

4 Conclusions and Future Work

We compare parallel random permutation algorithms on SMPs. We give fast implementations of four algorithms, i.e., the sorting-based algorithm, Anderson's shuffling algorithm, the dart-throwing algorithm, and Sanders' algorithm for distributed-memory environments. The performance ranking of the four algorithms depend on the input size, cost of random number generators, cost of synchronization, and memory hierarchy. Within the range of input sizes used, *Rand_Shuffle* is superior in performance when LCG is used for random number generation, while *Rand_Dist*

is fastest when SPRNG and TWISTER are used as random number generators. Our parallel implementation achieves speedups up to 6 with 12 processors. For future work, we would like to compare the performance of the algorithms on other architectures, e.g., the multi-threaded architecture. We expect a different performance ranking from the one on architectures with deep cache organization.

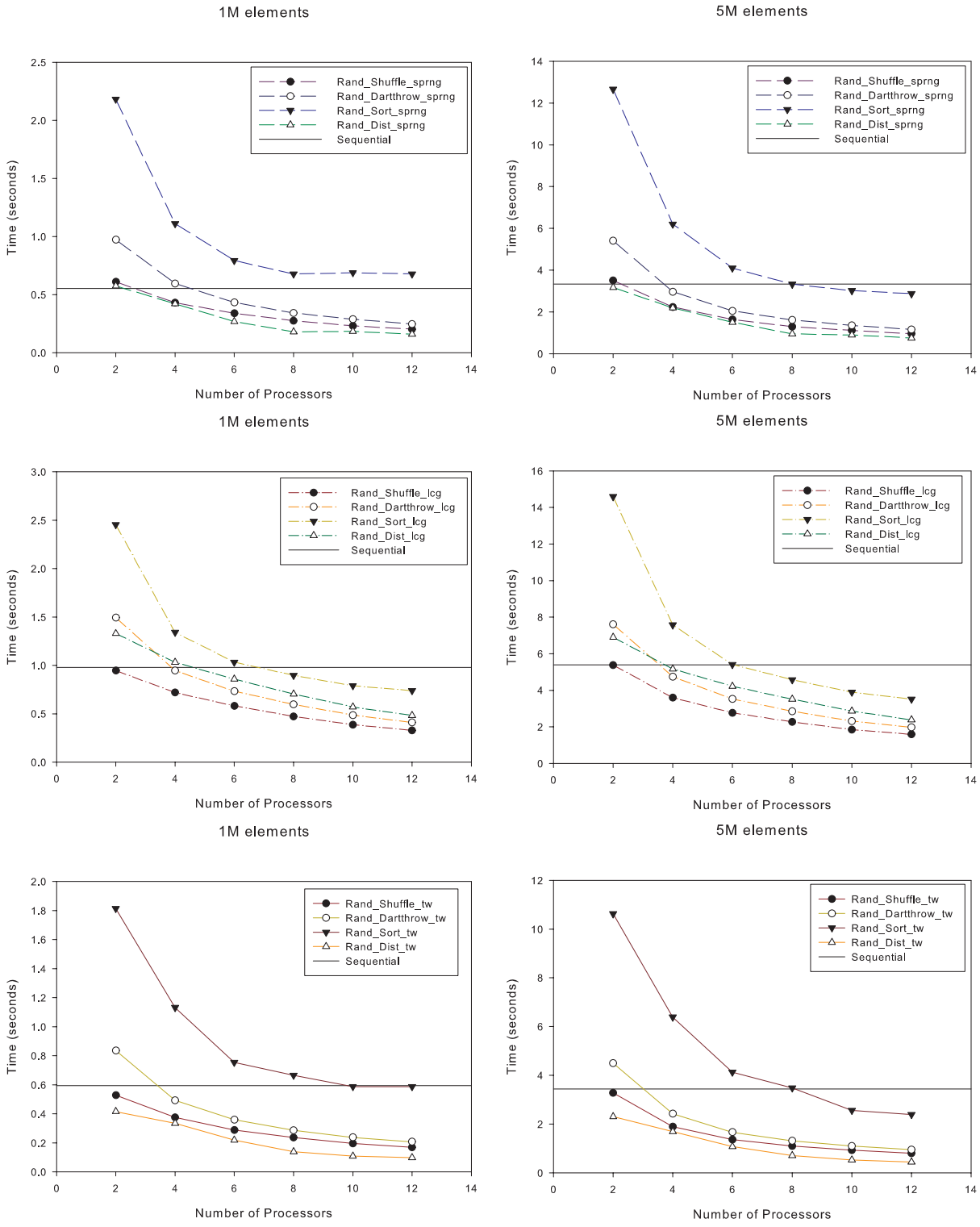

Figure 3: Comparison of the performance of *Rand_Shuffle*, *Rand_Dart*, *Rand_Sort* and *Rand_Dist* for input size 1M and 5M integer elements. The suffix shows which random number generator is used. SPRNG, LCG, and TWISTER are used in the top, middle, and bottom rows, respectively. *Rand_Shuffle* uses spinlocks for synchronization. *Rand_Dart* implements our algorithm with $\alpha = 2$. "Sequential" is the time taken for Knuth's algorithm for the same problem instance.

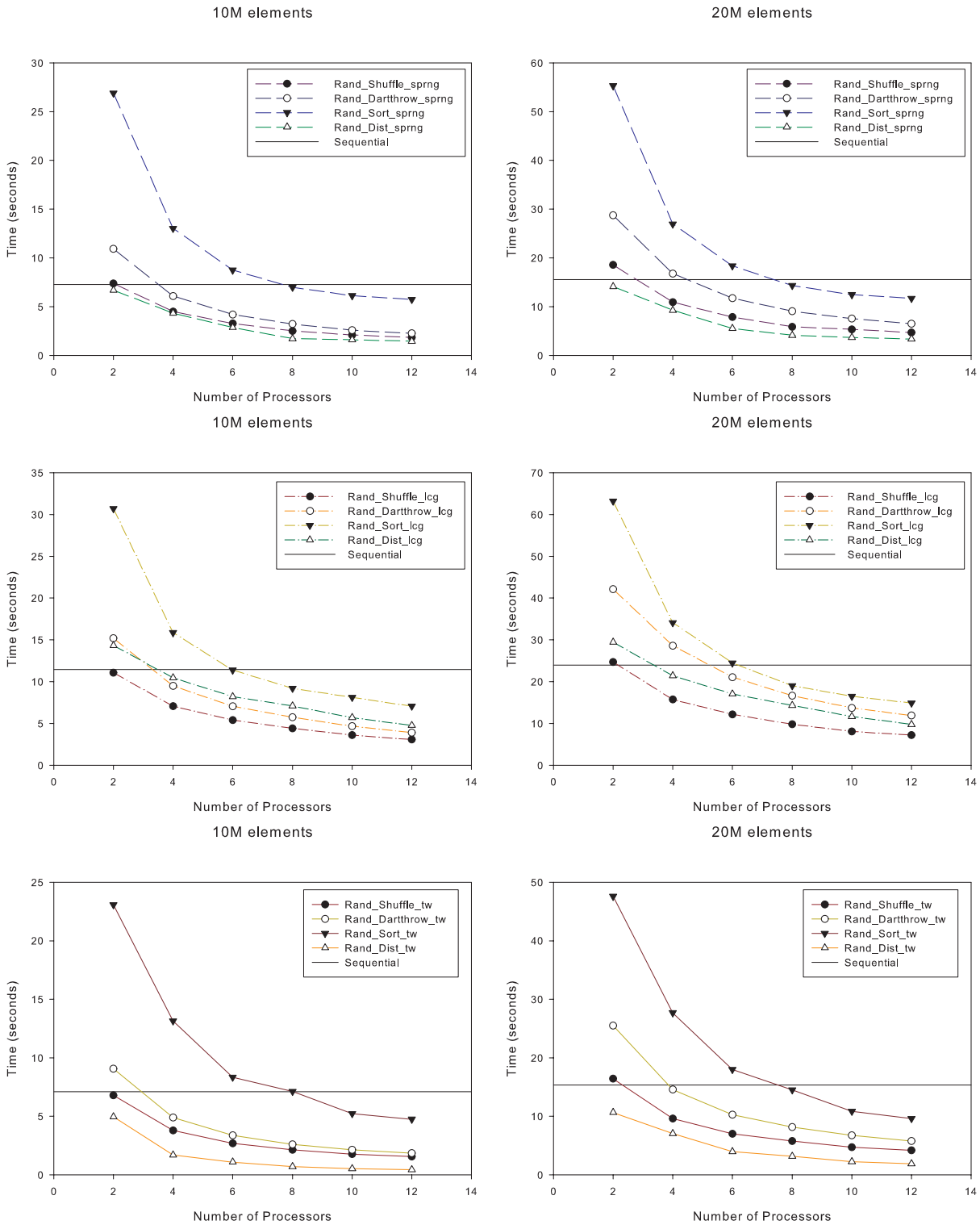

Figure 4: Comparison of the performance of *Rand_Shuffle*, *Rand_Dart*, *Rand_Sort* and *Rand_Dist* for input size 10M and 20M integer elements. The suffix shows which random number generator is used. SPRNG, LCG, and TWISTER are used in the top, middle, and bottom rows, respectively. *Rand_Shuffle* uses spinlocks for synchronization. *Rand_Dart* implements our algorithm with $\alpha = 2$."Sequential" is the time taken for Knuth's algorithm for the same problem instance.

References

[1] R.J. Anderson. Parallel algorithms for generating random permutations on a shared memory machine. In *Proc. 2nd Ann. Symp. Parallel Algorithms and Architectures (SPAA-90)*, pages 95–102, Island of Crete, Greece, 1990. ACM.

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

[3] G. Cong and D. A. Bader. Lock-free parallel algorithms: an experimental study. In *Proc. Int'l Conf. on High-performance Computing (HiPC 200 4)*, Banglore, India, December 2004.

[4] A. Czumaj, P. Kanarek, M. Kutylowski, and K. Lorýs. Fast generation of random permuations via networks simulation. *Algorithmica*, 21(1):2–20, 1998.

[5] P.B. Gibbons, Y. Matias, and V.L. Ramachandran. Efficient low-contention parallel algorithms. *J. of Computer and System Sciences*, 53:417–442, Dec 1996.

[6] J. Gil. Fast load balancing on a PRAM. In *Proc. 3rd IEE Symp. on Parallel and Distributed Computing*, pages 10–17, December 1991.

[7] T. Hagerup. Fast parallel generation of random permutations. In *Proc. 18th Int'l Colloquium on Automata Languages and Programming*, volume 510 of *Lecture Notes in Computer Science*, pages 405–416. Springer-Verlag, 1991.

[8] 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, January 1999. Springer-Verlag.

[9] D.E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison-Wesley Publishing Company, 1981.

[10] I.G. Lassous and E. Thierry. Generating random permutations in the framework of coarse grained models. In *Proc. Int'l. Conf. on Principles of Distributed Systems*, volume 2, pages 1–16, 2000.

[11] D.H. Lehmer. Mathematical methods in large-scale computing units. In *Proc. 2nd Symp. on Large-Scale Digital Calculating machinery*, pages 141–146, Cambridge, MA, May 1951.

[12] M. Manohar and H. K. Ramapriyan. Connected component labeling of binary images on a mesh connected massively parallel processor. *Computer Vision, Graphics, & Image Processing*, 45(2):133–149, 1989.

[13] M. Mascagni, A. Srinivasan, S.M. Ceperley, and F. Saied. *Scalable Parallel Random Number Generators (SPRNG) Library*. Florida State University, 2.0 edition, 1995. `sprng.cs.fsu.edu`.

[14] Y. Matias and U. Vishkin. Converting high probability into nearly-constant time – with applications to parallel hashing. In *Proc. 23rd Ann. ACM Symp. on Theory of Computing*, pages 307–316. ACM, May 1991.

[15] M. Matsumoto and Y. Kurita. Twisted GFSR generators. *ACM Transactions on Modeling and Computer Simulation*, 2(3):179–194, 1992.

[16] S. Rajasekaran and J.H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Computing*, pages 594–607, 1989.

[17] J.H. Reif. An optimal parallel algorithm for integer sorting. In *Proc. 26th Ann. IEEE Symp. Foundations of Computer Science*, pages 496–503. IEEE Press, 1985.

[18] P. Sanders. Random permutations on distributed, external and hierarchical memory. *Information Processing Letters*, 67(6):305–309, 1998.