

A Simple, Fast Parallel Implementation of Quicksort and its Performance Evaluation on SUN Enterprise 10000 *

Philippas Tsigas Yi Zhang
Department of Computing Science
Chalmers University of Technology
S-412 96 Gothenburg, Sweden
{tsigas, yzhang}@cs.chalmers.se

Abstract

We have implemented sample sort and a parallel version of Quicksort on a cache-coherent shared address space multiprocessor: the SUN ENTERPRISE 10000. Our computational experiments show that parallel Quicksort outperforms sample sort. Sample sort has been long thought to be the best, general parallel sorting algorithms, especially for larger data sets.

On 32 processors of the ENTERPRISE 10000 the speedup of parallel Quicksort is more than six units higher than the speedup of sample sort, resulting in execution times that were more than 50% faster than sample sort. On one processor parallel quicksort achieved 15% percent faster execution times than sample sorting. Moreover, because of its low memory requirements, parallel Quicksort could sort data sets twice the size that sample sort could under the same system memory restrictions.

1. Introduction

Sorting is an important kernel for sequential and multi-processing computing and a core part of database systems. Donald Knuth in [10] reports that “computer manufacturers of the 1960s estimated that more than 25 percent of the running time on their computers was spend on sorting, when all their customers were taken into account. In fact, there were many installations in which the task of sorting was responsible for more than half of the computing time.” As it was expected, sorting is one of the most heavily studied problems in computer science. Parallel algorithms for sorting have been studied for long, with many major advances in the

area coming from as early as the sixties [10]. Accordingly, a vast number of research articles dealing with parallel sorting have been published; the number is too large to allow mentioning them all, so we will restrict discussion to those that are directly related to our work. Considerable effort has been made by the theoretical community in the design of parallel algorithms with excellent and occasionally optimal asymptotic efficiency. However there has only been limited success in obtaining efficient implementations on actual parallel machines [8, 20]. Similar research effort has also been made with the practical aspects in mind, for a list of work in this area please see [3] and [14]. The latter work has given many exciting results due to interaction between the algorithms research area and the computer architectures research area. Most of the work on high-performance sorting is based on message-passing machines, vector supercomputers and clusters.

This paper looks into the behavior of a simple, fine-grain parallel extension of Quicksort for cache-coherent shared address space multiprocessors. Quicksort has many nice properties: i) it is fast and general purpose; it is widely believed that Quicksort is the fastest general-purpose sorting algorithm, on average, and for a large number of elements [1, 4, 7, 16], ii) it is in-place, iii) it exhibits good cache performance and iv) it is simple to implement. The new generation of hardware-coherent, shared address space multiprocessor systems with their already dominant position on the tightly-coupled multiprocessor systems are our target systems. The implementation of the parallel Quicksort algorithm utilizes the capabilities that these new systems have to offer and uses the following algorithmic techniques:

Cache-efficient: Each processor tries to use all keys when sequentially passing through the keys of a cached-block from the key array.

Communication Overlapping Fine-grain Parallelism: It is a fine-grain parallel algorithm. Although fine-grain parallelism has been thought to be inefficient for computa-

*Performed within the network for Real-Time research and graduate Education in Sweden (ARTES), supported by the Swedish Foundation for Strategic Research (SSF). Also, partially supported by the Swedish Research Council (VR).

tions like sorting due to the synchronization overheads, we achieved efficiency by incorporating non-blocking techniques for sharing data and computation tasks. No mutual locks or semaphores are used in our implementation.

Parallel Partition of Data: A parallel technique for partitioning the data similar to the one presented in [5] is used. We rediscovered this technique when parallelizing Quicksort.

We implemented the algorithm on a SUN ENTERPRISE 10000, a leading example of the tightly-coupled, hardware-coherent architecture and we compared it with sample sort, which has been previously shown to outperform other comparison based, general sorting algorithms, especially for larger data sets [1, 4, 7, 16]. On 32 processors we achieved a speedup that was more than 6 units higher than the speedup of sample sort. This speedup resulted in an execution time that was over 50% faster than sample sort. On one processor of the ENTERPRISE 10000 parallel Quicksort gave 15% percent faster execution times than the sample sort on many large sorting instances. Moreover, parallel Quicksort could sort data sets double the size that sample sort could because of the its low memory requirements. One one processor parallel Quicksort behaves as it sequential parent. The asymptotic number of all comparisons and computation and memory steps used by the algorithm is the same as in quicksort: $O(N \lg(N))$ on average and $O(N^2)$ for the worst case. The average time complexity for the parallel algorithm is $O(\frac{N \lg(N)}{P})$ and the worst case time complexity is $O(\frac{N^2}{P})$. $O(N + P)$ is the asymptotic space complexity.

The remainder of the paper is organized as follows: In Section 2 the algorithm and its analysis are presented. We describe the experimental evaluation in Section 3. The paper concludes with Section 4.

2. The Algorithm

Quicksort [9] is a sequential sorting algorithm that is widely believed to be the fastest comparison-based, sequential sorting algorithm (on average, and for a large input sets) [2, 11, 19]. It is a recursive algorithm that uses the “Divide and Conquer” method to sort all keys. The standard Quicksort first picks a key from the list, the *pivot*, and finds its position in the list where the key should be placed. This is done by “walking” through the array of keys from one side to the other. When doing this, all other keys are swapped into two parts in the memory: i) the keys less than or equal to the *pivot* are placed to “the low side” of the *pivot* and ii) the keys larger than or equal to the *pivot* are placed to “the high side” of the *pivot*. Then the same program is recursively applied on these two parts.

2.1. Description

Assume that we have an array with N keys, indexed from 0 to $N - 1$, to be sorted on a cache-coherent shared memory multiprocessor with P asynchronous processors. Each processor is assigned a unique index, *pid*, from 0 to $P - 1$.

The parallel Quicksort algorithm presented here is a simple parallelization of Quicksort. It is a 3+1-phase algorithm. The first three phases constitute the divide phase and are recursively executed. The last phase is a sequential sorting algorithm that processors execute in parallel, during this phase a helping scheme is used. The four phases are: i) the Parallel Partition of the Data phase, ii) the Sequential Partition of the Data phase, iii) the Process Partition phase and iv) the Sequential Sorting in Parallel with Helping phase. The detailed explanation of the four phases is given below.

Phase One: Parallel Partition of the Data The algorithm sees the array of the data as a set of consecutive *blocks* of size B . B depends on the size of the system’s first-level cache, and is selected so that two *blocks* of length B can fit in cache at the same time. In our system where the size of the first-level cache is 16KB, we selected $B = 2048$ so as to be able to fit two *blocks* of data in the cache at the same time. In order to simplify the description and without loss of generality let us consider first the case where all keys can be divided into *blocks* exactly, i.e. $N \bmod B = 0$. Later on, we will show how to extend this phase for the case where $N \bmod B \neq 0$. The whole array can be viewed as a line of $\frac{N}{B}$ data *blocks*; processors can only choose *blocks* to work on, from the two ends of the line.

The first phase starts with the processor P_0 , the one with the smallest *pid*, picking a *pivot*. After that, each processor in parallel picks the *block* that it finds at the very end of the left side of the line *leftblock* and then the *block* that it finds at the very end of the right side of the line *rightblock* and uses these two *blocks* together with the *pivot* as an input to a function that is called the *neutralize* function. This function is described in pseudo-code in Figure 1. The function takes as input two blocks, *leftblock* and *rightblock*, and the *pivot* and swaps the keys in *leftblock* which are larger than the *pivot* with keys in *rightblock* that are smaller than the *pivot* in an increasing order, as long as this can be done. A call of the *neutralize* function will result into one of the following results: either i) all keys in *leftblock* are going to be less than or equal to the *pivot*, in this case we say that *leftblock* has been *neutralized* or ii) all keys in *rightblock* are going to be larger than or equal to the *pivot* and then we say that *rightblock* has been *neutralized*, or iii) it can also happen that both *leftblock* and *rightblock* have been *neutralized* at the same time.

Each processor will then try to get a fresh *block* from the left side of the array if its *leftblock* was *neutralized* before, or from the right side if its *rightblock* was *neu-*

```

SIDE neutralize (Data *leftblock , Data *rightblock ,
Data pivot)
{
    int i, j;
    do{
        for ( i=0; i<BlockSize; i++ )
            if ( leftblock[i] > pivot )
                break;
        for( j=0; j<BlockSize; j++)
            if ( rightblock[j] < pivot )
                break;
        if ((i== BlockSize) || (j == BlockSize))
            break;
        SWAP( leftblock[i], rightblock[j]);
        i++; j++;
    } while ( i < BlockSize && j < BlockSize )

    if (i == BlockSize && j == BlockSize)
        return BOTH;
    if (i == BlockSize)
        return LEFT;
    return RIGHT;
}

```

Figure 1. The *neutralize* function

tralized before and it will then *neutralize* this *block* with the still charged *block* that it has on hand. If both *blocks* were *neutralized*, the processor gets two fresh *blocks* from both ends. Processes continue the above steps until all *blocks* are exhausted. At this moment, each processor has at most one *block* unfinished in hand and puts it on the *remainingBlocks* shared array that consequently can collect at most P blocks, and exits the *parallel partition* phase. The parallel partition phase is described in pseudo-code in Figure 2. Processors report the number of keys contained on *leftblocks* that have been neutralized by summing the numbers into LN (Left-Neutralized). The number of keys on *rightblocks* that have been neutralized are counted into RN (Right-Neutralized).

For the general case where $N \bmod B = M \neq 0$ we can modify the end condition for the parallel phase so that a processor exits when it finds that the remaining keys are not enough to form a *block*. In this case there are going to be at most P blocks plus M keys left. To process the remaining keys, the processor P_0 , with the smallest pid, will run the *sequential partition* phase.

Phase Two: Sequential Partition of the Data The purpose of **sequential partition** phase is to finish what the parallel partition has started: the placement of keys to the “correct side” of the *pivot*. When the parallel partition finishes all *neutralized blocks* that are between $[0, LN - 1]$ and $[RN, N - 1]$ are correctly placed with respect to the *pivot*. After the Parallel Partition phase, the remaining P blocks that can appear in any position on the array as shown in figure 3a need to be correctly placed and the neutralized blocks that are placed in $[LN - 1, RN]$ have to be swapped in a correct position. During this phase processor P_0 first sorts

```

if (pid == smallestpid)
    pivot = PivotChoose();
barrier(P);
leftblock = Get A Block From LEFT End;
rightblock = Get A Block From RIGHT End;
leftcounter = 0;
rightcounter = 0;
do{
    side = neutralize(leftblock , rightblock , pivot);
    if ((side == LEFT) || (side == BOTH)) {
        leftblock = Get A Block From LEFT End;
        leftcounter ++;
    }
    if ((side == right) || (side == BOTH)) {
        rightblock = Get A Block From RIGHT End;
        rightcounter ++;
    }
} while((leftblock != EMPTY)
    &&(rightblock != EMPTY));
if (leftblock != EMPTY)
    remainingBlocks[pid] = leftblock;
else
    remainingBlocks[pid] = rightblock;
LN = LN + (leftcounter * B);
RN = RN + (rightcounter * B);

```

Figure 2. The procedure that implements the parallel partition phase.

the P blocks using the start indices of these blocks. It then uses this order to pick a *block* from the left (*leftblock*) and a *block* from right (*rightblock*) and give them as input to the *neutralize* function together with the previously selected *pivot*. It does this until all remaining *blocks* are exhausted. In this phase it is not always true that a neutralized *block* can always contribute to LN or RN , for example, a *block* that was picked out from the right end during the parallel partition phase and is neutralized as *leftblock* now will not contribute to LN . All blocks between $[0, LN - 1]$ were picked during the parallel partition phase by some processors from the left end. If they can be neutralized as *leftblocks* in this sequential partition phase, they can contribute to the LN as other neutralized blocks $[0, LN - 1]$. Similar is the case for blocks between $[N - RN, N - 1]$. The procedure of sequential partition is described in figure 5. The first step is sorting the remaining blocks. Then, picking blocks from the two ends and running the *neutralize* function on them as shown in figure 3b. Finally, some blocks are still misplaced between $[LN, N - RN - 1]$. If there are m blocks unfinished between $[0, LN - 1]$, then there should be at least m blocks which are neutralized as *leftblocks* between $[LN, N - RN - 1]$. The Sequential Partition will swap them as shown in figure 3c. The same methods will be applied to the remain blocks between $[N - RN, N - 1]$. Now, all blocks between $[0, LN - 1]$ contain keys less than or equal to the *pivot* and all blocks $[N - RN, N - 1]$ contain keys larger than or equal to the *pivot*. The remaining task is to

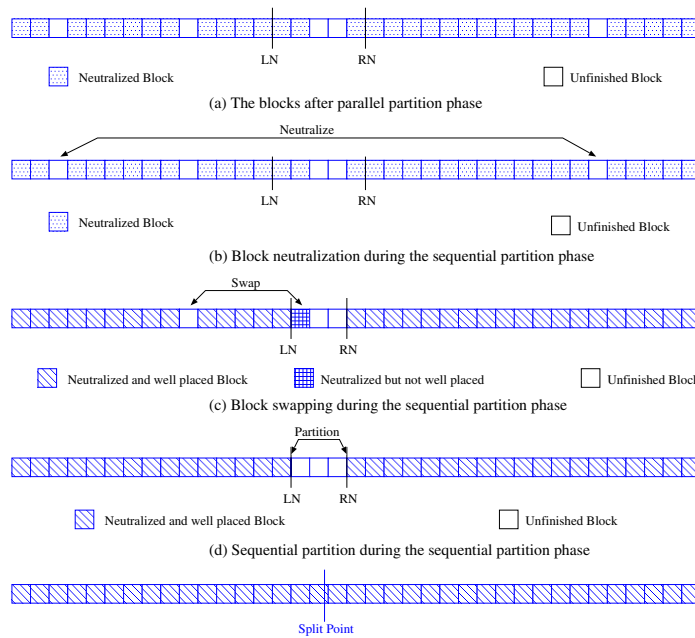


Figure 3. A graphical description of the sequential partition phase.

partition between LN and RN as sequential quicksort.

Next we will demonstrate the behavior of the two phases presented before by a way of an example. The example is also graphically shown in in Figure 4. For input 37 random integers are selected. Our system for this example has 3 processors and the *block* size is 4. First, we need to select a *pivot*. We use the method proposed by Sedgewick in [13] to choose the *pivot*, which is the median of the first, middle and last keys in the array (in this case the *pivot* does not have to be an input key).

$$pivot = \lfloor \frac{(\min(17, 7, 32) + \max(17, 7, 32))}{2} \rfloor = 19$$

After choosing the *pivot*, all processors will pick two *blocks* of keys one *block* from the left end and one *block* from the right end. One possible result is the following: i) processor 0 gets *block* L2 (the second — in input order — *block* from the left end) and R1 (the first — in input order — *block* from the right end), ii) processor 1 gets L1 and R2, iii) processor 2 gets L3 and R3. Then, every processor calls the *neutralize* function. For processor 0, after the return from the *neutralize* function, all keys in *block* L2 are less than or equal to the *pivot* and all keys in *block* R1 are larger than or equal to the *pivot*, i.e. L1 and R1 have been neutralized. *Block* R2 and L3 are neutralized by processors 1 and 2 respectively. To continue the algorithm processor 0 needs to get two more *blocks* one from each end of the array and processor 1 needs to get one *block* from the right end and processor 2 needs to get one from the left end. As the whole system is asynchronous, this time processor 2 gets *block* L4 and processor 0 get *block* L5 and R4.

When processor 1 tries to pick a *block*, it finds out that the remaining keys are not enough to form a *block*, and consequently it exits the parallel partition phase and puts *block* L1 in the *remainingBlocks* array. Processor 0 and processor 2 use the *neutralize* function with input L5 and R4, and L4 and R3, respectively. Processor 0 exits the parallel partition phase with L5 unfinished. Processor 2 exits with no unfinished *block*. The parallel partition phase is over with 2 *blocks* ($\leq P$) marked as unfinished and 1 key ($\leq B-1$) (key 29) left unprocessed in the middle. The algorithm will enter the sequential partition phase to process these keys, in our example there are no neutralized *blocks* in $(LN, N - RN)$ to be processed, $LN = 12$ and $RN = 16$.

Processor 0, as the one with the smallest *pid*, will process first the *blocks* L1 and L5 that are in the *remainingBlocks* array. Processor 0 calls the *neutralize* function with L1 as the *leftblock* and L5 as the *rightblock* and as a result L1 is neutralized. As the start index of L1 is less than LN , LN will be increased by 4. In this example the remaining *block* L5 is between $[LN, N - RN)$ and there is no need to swap it. The processor 0 splits the keys between $[LN, N - RN) = [16, 21) = [16, 20]$ with the *pivot*, 19. The final split point is index 18, as shown in figure 4.

After the two partition phases, the array is split into two subarrays and all keys are placed on the “right side” of the *pivot*.

Phase Three: Process Partition During this phase, the algorithm partitions all processors into two groups. The sizes of these groups are proportional to the sizes of the respective subarrays. If the size of one group is zero, then its size is set to 1 and the other group takes the remaining processors.

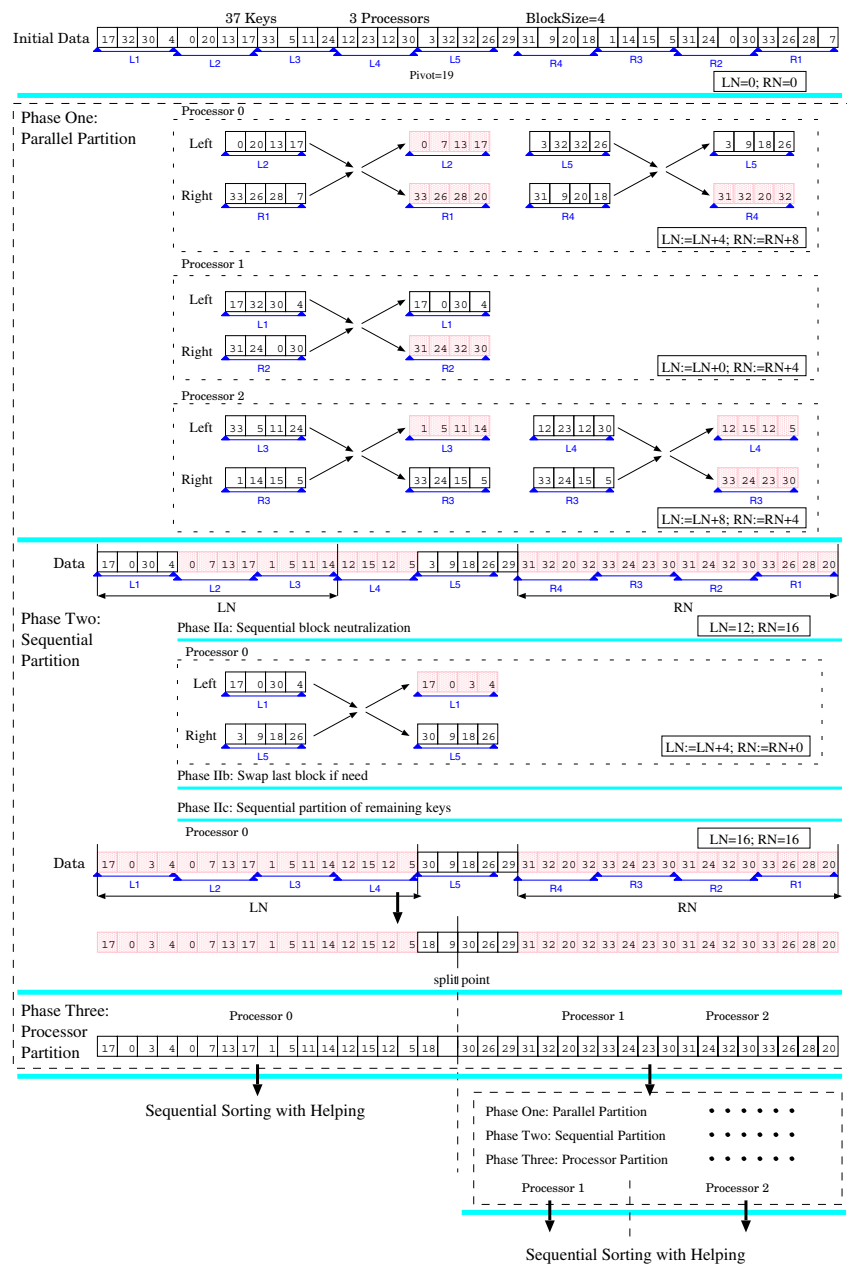


Figure 4. The algorithm by the way of an example.

Each processor group will take a subarray from Phase Two and apply the same parallel partition method, Phase One to Phase Three, on it recursively. When only one processor is assigned to a subarray, the processor will exit the partition phase and enter the *Sequential Sorting with Helping* phase.

Phase Four: Sequential Sorting with Helping During the sequential sorting phase, every processor uses quicksort to sort the subarray it gets from phase three and help other processors' work after it finishes its own. For the sequential quicksort we use, the optimization introduced in [11] is applied. This optimization gives good cache behavior.

Every processor uses an auxiliary stack for itself to keep track of the algorithm's state, and the recursion of Quicksort turns into a loop of *PUSH* and *POP* operations. Whenever a processor encounters a small subarray which can fit in cache, it will use inserting sort to sort it without *PUSH*ing it into the stack.

In our parallel sorting algorithm, we introduce the following helping scheme to achieve good load balance. The stacks of the sequential Quicksort of all processors are implemented as lock-free (non-blocking) stacks shared among all processors. All these stacks are restricted shared con-

```

sort(remainingBlocks , ascend order);
/* p is the number of remain blocks ;  $p \leq P$  */
left = 0; right = p - 1;
/* Treat the remainingBlocks as an array and
do Sequential block Partition */
while(left < right)
{
    /* Neutralize the most left block
    and the most right block */
    side = neutralize( remainingBlocks[ left ],
        remainingBlocks[ right ], pivot);
    if ((side == LEFT) || (side == BOTH)) {
        if (remainingBlocks is between  $[0, LN - 1]$ ){
            /* update LN and remove the block from
            remainingBlocks only if it is between  $[0, LN -$ 
1] */
            LN += B;
            remainingBlocks[ left ] = EMPTY;
        }
        left ++;
    }
    if ((side == right) || (side == BOTH)) {
        if (remainingBlocks is between  $[N - RN, N - 1]$ ){
            /* update RN and remove the block from
            remainingBlocks only if it is between  $[N -$ 
RN, N - 1] */
            RN += B;
            remainingBlocks[ right ] = EMPTY;
        }
        right --;
    }
}
/* For those which still remain in the
remainingBlocks array , we will swap
them with blocks between  $[LN, N - RN)$  */
for ( i=0; i<p; i++) {
    if (remainingBlocks[i] is not EMPTY) {
        Swap remainingBlocks[i] with neutralized
        blocks between  $[LN, N - RN)$ ;
    }
}

```

Figure 5. The procedure that implements the sequential partition phase.

current stacks, because only one processor performs the *PUSH* operations. In the algorithm presented here, we used a variant of [17] that has been optimized for our restricted class of stacks. When one processor finished its job, it will start to help other processors by *poping* out unsorted subarray (one at a time) from their stacks. In this way we can achieve load balance online.

The pseudo code of the complete algorithm is shown in Figure 6¹.

2.2. Analysis of the Algorithm

The parallel Quicksort presented before is a simple parallelization of Quicksort. The parallel algorithm follows the

¹The complete version of the code is available for non-commercial use at: <http://www.cs.chalmers.se/~yzhang/PQuick>

```

PQuicksort(Data *array , int size ,
int ProcessorNumber)
{
    if ( ProcessorNumber == 1 ) {
        /* Parallel Sorting Phase*/
        SequentialSorting(array);
        HelpOtherProcessor();
    }
    /* Parallel Partition Phase; The code is in
    Figure 2*/
    parallel partition;

    /* Sequential Partition Phase */
    if ( pid == smallestpid ) {
        split = sequential partition;
        /* The code is in 5 */

        /* Processor Partition Phase */
        processorsplit = Processor Partition;
    }
    barrier(ProcessorNumber);

    /* Recursive call */
    PQuicksort(&array[0], split+1, processorsplit);
    PQuicksort(&array[split+1], size-split-1,
        ProcessorNumber-processorsplit);
}

```

Figure 6. The complete algorithm

same divide-and-conquer steps as Quicksort and the depth of recursion is the same between the parallel quicksort and the sequential quicksort. Therefore, the amount of comparison and swap operations of the parallel algorithm is the same with the sequential one: $O(N \lg(N))$ for the average case and $O(N^2)$ for the worst case. Now, when looking into the analysis of the speedup of the algorithm we can see that parallelism is introduced in two places: i) the partition phase and ii) the sorting phase. The time cost of the partition phase is $O(N)$. The parallel algorithm finishes the partition of the whole array in two steps. First, all processors neutralize *blocks* of keys in parallel; this will take $O(\frac{N}{P})$ time. Then one processor will process the unfinished P blocks and M keys; this will take $O(P * B + M)$ time. As $M \leq B$, the whole time complexity for the partition phase is $O(\frac{N}{P} + (P + 1) * B)$. If $B \ll N$ then the speedup of the partition phase will be $O(\frac{N}{P})$.

The next parallelism introduced is in the sorting phase. When the whole array is partitioned into P subarrays, each processor will run Quicksort on one of these subarrays. The stacks used by all processors for Quicksort during this phase will be shared by all processors. When a processor is finished with its own subarray, it will access other processors' stacks to help them until all keys are sorted. The speedup for this phase is approximately P with some small synchronization overhead.

From the above analysis, we can see that the time complexity of the algorithm depends on the size of *block*, B , and the number of processors, P and does not depend on

the distribution of keys.

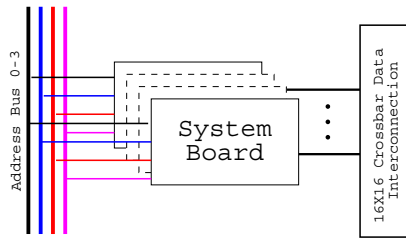


Figure 7. The architecture of the SUN Enterprise 10000

3. Experimental Results

3.1. The SUN ENTERPRISE 10000 Platform

The SUN ENTERPRISE 10000, is a scalable, hardware-supported, cache-coherent, symmetric or uniform memory access (cc-UMA) multiprocessor machine. In ENTERPRISE 10000, every processor has its own cache and all the processors and memory modules attach to the same interconnect. In symmetric shared memory multiprocessor systems, data normally needs to be moved from point to point, while addresses often must be broadcasted throughout the system. Therefore the interconnect of E10000 uses a packet switched scheme with separate address and data paths. Data is transferred with a fast crossbar interconnect, and addresses are distributed with a broadcast router. The crossbar interconnect is constructed with two levels: global and local. In the Global level, there is a 16 byte wide, 16 x 16 crossbar that steers data packets between the 16 system boards. The global data crossbar connects the 16 system boards' ports, as Local level, together. At the Local level, "many-to-one" routers are used on the system boards to gather on-board requests and direct them to one port (per board). The address routing is implemented over a separate set of four global address buses, one for each of the four memory banks that can be configured on a system board. The buses are 48 bits wide including error correcting code bits. Each bus is independent, meaning that there can be four distinct address transfers simultaneously. Figure 7 graphically describes the architecture of the new SUN ENTERPRISE 10000. Main memory is configured in multiple logical units. All memory units, processors, and I/O buses, are equidistant in the architecture and all memory modules comprise a single global shared memory space. The latter means that the machine provides not only a global address space but also the memory access to any memory location is uniform. The machine we used had 36, 249 MHz UltraSPARC processors, which were divided into two logical domains: one with 4 processors as frontend and another one with 32 processors. Each CPU had a 16 KB first-level data cache and a 4 MB

second-level cache. Our experiments were done on the 32 processor domain.

3.2. Parallel Sorting with Sample Sort

We compared parallel Quicksort with sample sort. Sample sort has been shown to be the best comparison-based and consequently general sorting algorithm for larger data sets [1, 4, 7, 16].

Sample sorting is a five-phases algorithm. First it divides the keys evenly among all P processors, and then, during the first phase each processor sorts locally its own keys. During the second phase each processor samples a fixed number of keys from its locally sorted keys. These sample keys in phase three of the algorithm are sent to one processor, that sorts them and selects $P - 1$ out of them as sample splitters for the next phase. During the fourth phase each processor uses these $P - 1$ splitters to partition the sorted input values and to decide locally the appropriate destinations of its partitioned keys. Finally, in the last phase of the algorithm, each processor uses mergesort locally to merge the key sequences sent to it. In [12, 15] description of the algorithm and its analysis can be found together with a study on ways to decide how to perform the sampling of the keys and how to select the sample splitters and how these selection affect the load balance and the behavior of the program. Sample sort is a very efficient parallel sorting algorithm on distributed memory and message passing systems [14].

Sorting Benchmarks:

The performance of sorting depends on the distribution of key values. We used the benchmark data sets, *Uniform*, *Gaussian*, *Zero*, *Bucket* and *Stagger*. A detailed description and justification of the benchmarks can be found in [6].

We used five different input sizes of integers for each of the above five benchmarks: 8M, 32M, 64M, 128M and 256M. For our experiments we had exclusive access to 32 processors of a SUN ENTERPRISE 10000 machine. For parallel Quicksort, we choose the size of *block* to be 2048, with this number two *blocks* could be placed in the first level cache of our system at the same time. For the sample sort, we selected the sample size according to [12, 15] and the number of processors that we had access to. The speedup results are shown in Figures 8, 9, 10 and 11 and Table 1. The results for parallel Quicksort are labeled PQuick. For the experiments with 256M integer keys, we do not have any results for sample sort. This is because the kernel configuration of our systems does not allow single programs to allocate more than 2G bytes of memory and sample sort has higher memory needs than the parallel Quicksort presented here.

From the results obtained, we can see that for any number of processors and for any data size selected parallel

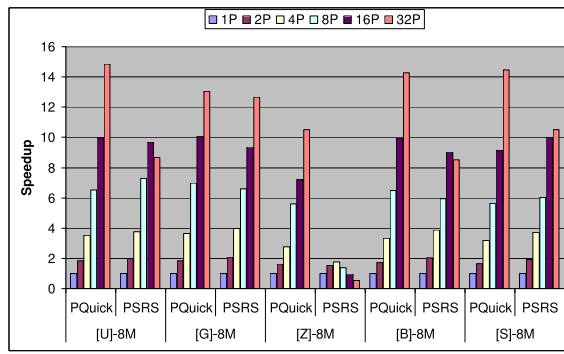


Figure 8. Experiments with 8M Integers

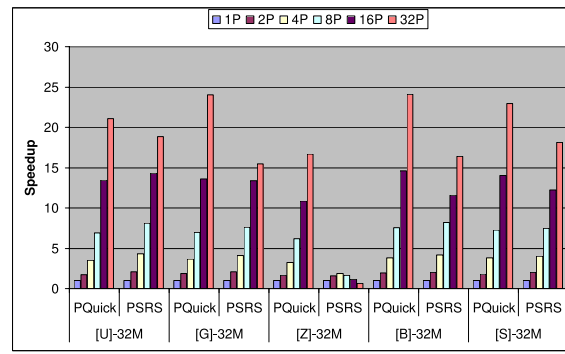


Figure 9. Experiments with 32M Integers

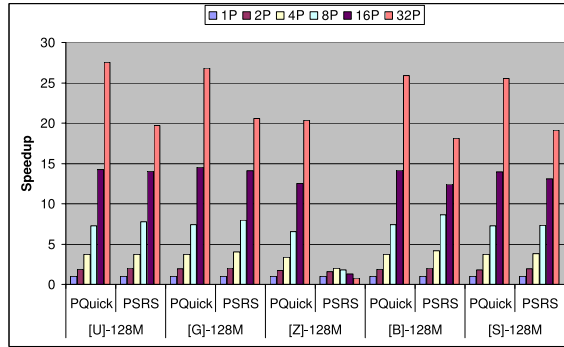


Figure 10. Experiments with 128M Integers

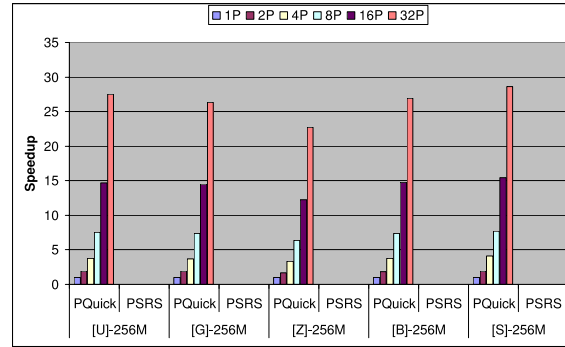


Figure 11. Experiments with 256M Integers

No. Proc.	[U]-64M		[G]-64M		[B]-64M		[S]-64M	
	PQuick	PSRS	PQuick	PSRS	PQuick	PSRS	PQuick	PSRS
1	139221763	157149728	147128678	160416695	139882786	156416659	141337885	157289882
2	79665604	80083084	74646110	77354959	71928127	75946922	82174454	83648007
4	39740592	38210913	37798256	38968839	36426837	36768544	37653127	41555739
8	19799215	20113143	19299378	20098710	18661486	18728796	19496991	21802329
16	10309607	11108192	9900953	11084988	9518032	12768536	9878692	12357691
32	5564001	8015688	5590313	8291338	5221680	9297328	5796658	9318278

Table 1. The execution times (μs) with 64M Integers

Quicksort exhibits the following characteristics:

The execution time of parallel Quicksort is not sensitive to the distribution of the input data Parallel Quicksort produced the same speedup for the the benchmark datasets *Uniform*, *Gaussian*, *Bucket*, and *Stagger*. For the benchmark *Zero*, the speedup that we got is always less than the speedup that we got on the other benchmarks. But, when looking at the absolute execution times, we can see that the sequential time for the *Zero* benchmark is about 30% faster than the sequential execution time for the other benchmarks while all other benchmarks have almost the same sequential execution times. The execution time of parallel Quicksort is not sensitive to the distribution of the data as mentioned in Section 2.2.

Better Execution Times and Speed-ups than Sample Sort The absolute sequential execution times from sam-

ple sort are at least 15% slower than those of the parallel Quicksort in general, with only exception a small number of instances with processors between 2 to 8 and mostly in the benchmarks *Uniform* and *Gaussian*. All the instances in which sample sort outperforms the parallel Quicksort are shown in Figure 3. The excess execution part in sample sort is due to the Mergesort that copies the data from one array to another array. From these figures, we can see that the speedups of the sample sort are better than the parallel Quicksort algorithm when the input sizes are small and the number of processors are small with the only exception the benchmark *Zero* (in *Zero* sample sort performs very poorly). Figure 1 shows the execution times of the benchmark datasets *Uniform*, *Gaussian*, *Bucket*, and *Stagger* for parallel Quicksort and the sample sort algorithm with 64M integers (64 is in the median of 8 and 128, the smallest and

No. Proc.	[Z]-8M		[Z]-32M		[Z]-64M		[Z]-128M	
	PQuick	PSRS	PQuick	PSRS	PQuick	PSRS	PQuick	PSRS
1	9643146	12912753	48593446	62147791	100276211	127420481	211486046	262376511
2	5928310	8383057	28895086	39120591	59113923	79046473	123347231	162749347
4	3508595	7251678	14904826	33235381	30510105	65319136	62769102	131289494
8	1723524	9285370	7848143	37933231	15400692	73537879	32329882	144721896
16	1334066	13362880	4478794	54547186	8475414	104793490	16883436	203698914
32	917237	23342432	2910912	93403783	5020324	178521414	10375268	343280741

Table 2. The execution times (μ s) of Benchmark Zero

the biggest input sizes for sample sort). In sample sort, the local sort phase is well load balanced. The mergesort will also be well load balanced if there are not many duplicate keys, which is the case in benchmarks *Uniform* and *Gaussian*. At the same time, when the number of processors increases, the cache capacity that is used in the algorithm increases. The increase of cache capacity would offset the parallelism overhead in the sample sort algorithm and even introduces superlinear speedup e.g. most experiments on 2 processors for all input data sizes with only exception the benchmark *Zero*. However, when the number of processors and/or the input size of the data is large enough, the cost for parallelism could not be offset by the increase of cache capacity any more. There is no superlinear speedup for sample sort for more than 16 processors and for the experiments with 128M integers only those with 2 processor show superlinear speedup. After that point, the speedup of sample sort lags behind the speedup of the parallel Quicksort. As the sequential execution time of sample sort is longer than the execution time of the parallel Quicksort, the sample sort can only beat the parallel Quicksort when a large superlinear speedup is achieved.

On benchmark *Zero*, sample sort never performed better than the parallel Quicksort. The benchmark *Zero* is the most difficult for sample sort. The execution times of the parallel Quicksort and sample sort for benchmark *Zero* with 8M, 32M, 64M, 128M are shown in Figure 2. When using the benchmark *Zero* the bottleneck is the fifth phase of mergesort. Since all keys have the same value, all keys will be send to one processor at the last phase of the algorithm to be mergesorted. When sorting inputs from *Zero*, only the first phase (local sorting) executes in parallel. On the other hand, parallel Quicksort delivers the best absolute execution times when sorting inputs from benchmark *Zero*. The reason is that the performance of Quicksort is optimal because any *pivot* will partition keys evenly; the best sequential time of parallel Quicksort for benchmark *Zero* also confirm this. Partitioning the keys also help the parallel algorithm by distributing the subarrays evenly among processors.

For double floating point benchmarks, we observed the same trend: parallel Quicksort deliver almost the same speedup all the time; the speedup of the sample sort slow-down when the data size increases and when the number of processor increases.

In the full paper [18], all the experimental results are described.

4. Conclusion

Cache-coherent shared memory multiprocessors offer fruitful ground for algorithmic or programming techniques that were considered impractical before, in the context of high-performance programming, to develop and change a little the way we think about high-performance programming. We have implemented sample sort and a parallel version of Quicksort on a cache-coherent shared address space multiprocessors: the SUN ENTERPRISE 10000. Our computational experiments show that parallel Quicksort outperforms sample sort. Sample sort has been long thought to be the best, general parallel sorting algorithms, especially for larger data sets. The parallel version of Quicksort is a simple fine-grain parallelization of Quicksort. Although fine-grain parallelism has long been thought to be inefficient for computations like sorting due to the synchronization overheads, efficiency was achieved by increased concurrency between communication and computation, This concurrency comes from the incorporation of non-blocking techniques especially when sharing data and sub-tasks. Quicksort might be a practical choice when it comes to general purpose in-place sorting both for uniprocessor and multiprocessor cc-DSM systems.

Acknowledgements

We thank Carl Hallen and Andy Polyakov from our Supercomputing Center, for their help on our inconvenient requests for exclusive use. Many thanks to Marina Papatriantafillou for her support and continuous flow of comments. Niklas Elmqvist and Anders Gidenstam provided us with many comments that made this paper much better to read.

References

- [1] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the connection machine CM-2. In *Proceedings of the 3rd annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'91)*, pages 3–16, Hilton Head, South Carolina, July 1991.

No. Proc.	[B]-8M		[S]-8M		[U]-32M		[U]-64M	
	PQuick	PSRS	PQuick	PSRS	PQuick	PSRS	PQuick	PSRS
1	14269874	16414229	14325041	16452619	68339788	76938437	139221763	157149728
2	8244073	8031141	8614346	8548883	38923895	36649493	79665604	80083084
4	4252418	4238064	4485420	4402723	19327928	17931728	39740592	38210913
8	2204853	2760520	2537265	2722804	9908154	9465147	19799215	20113143

Table 3. The execution times (μs) of instances (in boldface) where sample sort is faster than parallel Quicksort

- [2] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. MIT Press and McGraw-Hill Book Company, 6th edition, 1992.
- [3] R. Dlekmann, J. Gehring, R. Lüling, B. Monien, M. Nübel, and R. Wanka. Sorting large data sets on a massively parallel system. In *Proceedings of the 6th Symposium on Parallel and Distributed Processing*, pages 2–9, Los Alamitos, CA, USA, Oct. 1994. IEEE.
- [4] A. C. Dusseau, D. E. Culler, K. E. Schauer, and R. P. Martin. Fast Parallel Sorting Under LogP: Experience with the CM-5. *IEEE Transactions on Parallel and Distributed Systems*, 7(8):791–805, Aug. 1996.
- [5] P. Heidelberg, A. Norton, and J. T. Robinson. Parallel quicksort using Fetch-and-Add. *IEEE Transactions on Computers*, 39(1):133–137, Jan. 1990.
- [6] D. R. Helman, D. A. Bader, and J. JáJá. A randomized parallel sorting algorithm with an experimental study. Technical Report CS-TR-3669 and UMIACS-TR-96-53, Institute for Advanced Computer Studies, University of Maryland, College Park, MD, Aug. 1996.
- [7] D. R. Helman, D. A. Bader, and J. JáJá. Parallel algorithms for personalized communication and sorting with an experimental study. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 211–222, Padua, Italy, June 1996.
- [8] W. Hightower, J. Prins, and J. Reif. Implementations of Randomized Sorting on Large Parallel Machines. In *Proceedings of the 4th annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'92)*, June 1992.
- [9] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, Apr. 1962.
- [10] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, USA, second edition, 1998.
- [11] A. LaMarca and R. E. Ladner. The influence of caches on the performance of sorting. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 370–379, New Orleans, Louisiana, 5–7 Jan. 1997.
- [12] X. Li, P. Lu, J. Schaeffer, J. Shillington, P. S. Wong, and H. Shi. On the versatility of parallel sorting by regular sampling. *Parallel Computing*, 19(10):1079–1103, Oct. 1993.
- [13] R. Sedgewick. Implementing Quicksort programs. *Communications of the ACM*, 21(10):847–857, Oct. 1978.
- [14] H. Shan and J. P. Singh. Parallel sorting on cache coherent DSM multiprocessors. In *Proceedings of Supercomputing '99*, Portland, Oregon, USA, Nov. 1999.
- [15] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14(4):361–372, 1992.
- [16] A. Sohn and Y. Kodama. Load balanced parallel radix sort. In *Proceedings of the International Conference on Supercomputing (ICS-98)*, pages 305–312, New York, July 1998. ACM press.
- [17] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RC 5118, IBM T. J. Watson Research Center, Yorktown Heights, NY, Apr. 1986.
- [18] P. Tsigas and Y. Zhang. Parallel quicksort seems to outperform sample sort on cache-coherent shared memory multiprocessors: An evaluation on sun enterprise 10000. Technical report, Department of Computing Science, Chalmers University of Technology, Mar. 2002. <http://www.cs.chalmers.se/~tsigas/papers/PQuick-TR.pdf>.
- [19] E. W. Weisstein. Eric Weisstein's world of mathematics. Technical report, Wolfram Research, <http://mathworld.wolfram.com/Quicksort.html>, 1999.
- [20] M. Zagha and G. E. Blleloch. Radix sort for vector multiprocessors. In *Proceedings of Supercomputing '91*, pages 712–721, Albuquerque, New Mexico, Nov. 1991. IEEE.