Parallel Algorithm for Integer Sorting with Multi-core Processors

S. Stoichev, S. Marinova

Key Words: *Parallel algorithm; sorting; counting sort; multi-core processor; shared-memory.*

Abstract. *This paper investigates the problem of optimizing sorting of large data sets by applying parallelism. Six variants of a parallel algorithm based on Counting-sort, designed for shared-memory parallel architectures are implemented. Counting-sort is applicable for sorting integers in a given range and is faster than all known sorting algorithms. A speedup from 1.1 times to 4.8 times with respect to the sequential algorithm, achieved by the implemented algorithms, which we called P-CountingSort, PaddingSort, StepPaddingSort, BlockSort, LoopBSort, PNLoopBSort, is demonstrated. Optimization techniques for multi-core processors of the contemporary PCs and investigated the speedup after applying the proposed techniques are applied. The performance and scalability of the implemented algorithms are measured under different conditions and distributions of the input data. Based on the experimental results, the PaddingSort algorithm achieves maximal speed-up for significant key ranges and a multi-processor system. We made recommendations for future research based on the conducted experiments.*

1. Introduction

This paper investigates the problem of designing an efficient parallel sort algorithm for shared-memory parallel computer systems. The development of multi-core processors determines the motivation for this research and the focus on optimizing the speed of SQL queries with parallel architectures. At the present time, working with multi-core processors of the personal computers is becoming a customary practice, which determines the future development of programming in order to obtain the maximal benefit from the organization of parallel architectures.

Sorting is a common operation in database systems. Accelerating the process of sorting is important in the area of Online Analytical Processing (OLAP), where the usage of a data warehouse suggests processing tens of millions of records, due to the presence of historic data. The basic operations that are typical for OLAP are rollup, drill-down, slice and dice, pivot, ranking, sorting and selection. As the tables in the warehouse grow in size, the time needed to perform the operations and answer user queries grows as well. An approach to optimizing the query processing is applying parallelism to the most time-consuming database operations.

The parallel sort algorithms, presented in this paper, are based on Counting-sort, which is one of the most efficient operative algorithms known at the moment [17]. The high speed of sorting makes the algorithm interesting both from theoretical and practical point of view. Research has shown that Counting-sort is faster that all existing sort algorithms when certain conditions are met (integer keys in a given range)[17]. Many sort algorithms exist at the present time, and QuickSort is one of the fastest, with many developments for its optimization. According to the authors knowledge at the current stage there are no developments or attempts to parallelize Counting-sort, based on distribution of records by key value, although such algorithm can find wide practical application in database systems and OLAP.

Counting-sort is sorting by key or sorting by key and number. Sorting by key and number is applicable for sorting records in a database. The records consist of a *key* and a *number* – integer in a given range. The number is used for unique identification of the record and it is defined as *identifier*.

Summary of Contributions:

The paper offers the following contributions:

– In Section 2 we present a survey of the existing parallel sort algorithms in comparison to the algorithm, proposed in this paper.

– In Section 3 we describe the general organization of multi-core processors and the experimental processor. We present the notation, the algorithm and the implementation of Counting-sort and we illustrate the sorting mechanism with an example – sorting an array of ten records.

– In Section 4 we propose a parallel sort algorithm in six variants, which we call *P-CountingSort, PaddingSort*, *StepPaddingSort*, *BlockSort*, *LoopBSort* and *PNLoopBSort*. We illustrate the parallel sort with an example – sorting an array of 4**N* records with four processors.

– In Section 5 we present results of benchmarking the variants of the algorithm with multi-core and multi-processor systems, based on running time and achieved speedup with respect to the sequential algorithm. We investigate the influence of the design and the scale of the parallel architecture and the characteristics of the input data over the execution time. The scalability of the proposed variants of the parallel algorithm is evaluated.

– In Section 6 we make general conclusions about the applicability of the proposed algorithm based on the experimental results and recommendations for future work.

2. Related Work

Comparison of Keys: Many parallel sort algorithms have been invented, that are based on comparison of keys – [4,7,11,14,18]. The comparisons create a large amount of conditional branches that cause stalls in the processor's pipeline. The contemporary processors, which gain huge benefits from the branch prediction hardware, cannot use this hardware for sorting due to the execution of comparisons of keys with unpredictable outcome [6]. Publication [7] approaches the problem of decreased performance by applying SIMD instructions for eliminating the conditional branches.

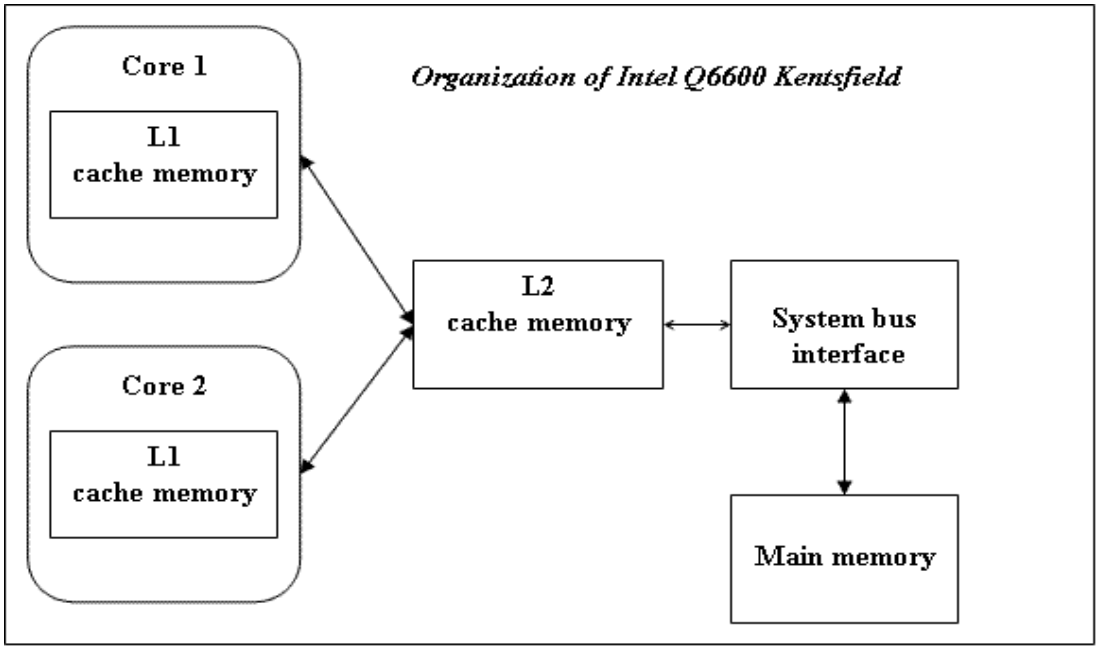

Figure 1. Organization of the experimental processor

Distribution by Keys: There are several developments concerning parallel sort algorithms based on distribution or digital properties of the keys rather than comparison of keys – [1,2,5,8,10]. The existing parallel algorithms are based on Radix Sort, Sample Sort and Bucket Sort.

Finding the Optimal Algorithm: Publication [5] investigates the problem of finding and using the sort algorithm that is optimal for the given system and input data. The authors claim that there is not a single sort algorithm optimal for every situation and propose a library of algorithms. A special attention is paid to sorting with shared memory systems due to the increasing importance of multi-core processors.

Sorting with Shared-nothing Architectures and External Sorting: A significant part of the research in the area of parallel sort algorithms is dedicated to parallel „shared nothing" architectures – [20,19] and external sorting [3,18].

Counting-sort: The aim of the authors is to implement a parallel algorithm based on an unexplored sort algorithm called Counting-sort, which has a complexity $O(N+K_{max})$, where N is the length of the input array, K_{max} is the length of the key range[17]. Therefore the parallel algorithm presented in this paper is a novel approach to parallel sorting, designed for shared memory parallel systems and is expected to be faster than the existing algorithms in the case of integer sorting. It is not based on comparison of keys and therefore does not suffer from a decreased performance due to the presence of many conditional branches.

3. Preliminaries

3.1. Organization of the Experimental Multi-core Processor

Multi-core processors are implemented as one physical CPU with several cores embedded in it. Each core has a decoder, ALU units, registers and other computational resources and the cores can share Translation Lookaside Buffers (TLBs). Depending on the design, the cores share a cache memory or each core uses its own cache memory. Hybrid approaches exist, where each core uses its own L1 cache memory and core pairs share L2 or L3 cache memory. The multi-core processor used for the experiments is Intel Q6600 Kensfield. It is a quad core CPU that contains two dies, with two processing cores located on each die. *Figure 1* depicts the organization of the die.

3.2. Counting-sort

In this paper we examine a sorting algorithm known as Counting-sort that is one of the most efficient algorithms, known by this moment.

3.2.1. Notation

A is an input array of N records to be sorted. Each record $A[i]$ $(1 \le i \le N)$ from the input array is defined as a structure, containing a key K in $[K_{min}..K_{max}]$ and an identifier n in $[N_{min}..N_{max}]$.

For clarity, in the expose it is assumed that $K_{min}=1$ and $N_{min}=1$, although the algorithm is applicable for ranges, starting from any integer number. The records $A[1]$, $A[2]$,...$A[N]$ are sorted in ascending order according to the key K, where a permutation $p=\{p_1, p_2,...p_N\}$ (p_i in $[1..N_{max}]$) is found such that:

$A(p_1).K \le A(p_2).K \le A(p_3).K \le ... \le A(p_N).K$

is satisfied and

$A(p_i).n = p_i$ for $1 \le i \le N$.

3.2.2. Sequential Algorithm

The algorithm for Counting-sort is divided into two steps:

(1) Distribution, which is a distribution of the records from the input array by key into buckets;

(2) Output, which is traversing the buckets in ascending order according to the key and writing the sorted sequence of records to an output array (which could be the input array).

An implementation of step *(1)* in the programming language C is shown in *figure 2*. The compilation of arrays *start* and

```
void CountingSort(element A[], int start[],
 int next[])
{
        int number,k,p;
        for(i=0;i<Kmax;i++) start[i]=-1;
        for(i=0;i<N;i++)
        {
                k=A[i].key;
                p=start[k-1];
                number=A[i].number;
                start[k-1]=number;
                next[number]=p;
        }
}
```

Figure 2. Program implementation of step *(1) Distribution*

next is a base for implementing other operations such as searching, computing aggregation functions and performing join operations that are applied in SQL queries. The parallel implementation of Counting-sort, which we propose in this publication, distributes the input array into multiple arrays *start* and one array *next* (except the *PNLoopBSort* algorithm). The compiled arrays do not need to be merged, since they are the basis for other parallel operations like parallel search, parallel computing of aggregation functions and parallel join operations. The execution of Step(2) *Output* is required only for outputting a list of records, sorted by key. A focus of the research is optimizing the execution time of step *(1) Distribution*.

3.2.3. Implementation

There are two known methods for implementing Counting-sort – static and dynamic implementation. In this paper we will examine the static implementation that leverages static memory allocation.

There are two data structures in the static implementation of Counting-sort:

– an array *start* of size K_{max};

– an array *next* of size N_{max};

The element *start*[*i*] is a pointer to a list of identifiers of records with key *K*=*i*. The array *next* stores lists of identifiers and *next*[*i*] stores the index of the next identifier in the list. *Figure 4* illustrates the graphic of sorting for array of ten records (*figure 3*) and a key range K_{max}=5.

4. Parallel Algorithm

This section presents six variants of a parallel sort algorithm based on Counting-sort. The variants are based on three approaches for implementing Counting-sort on a parallel architecture – a direct implementation (*P-CoountingSort*), an implementation based on the architecture's characteristics and an implementation based on the mechanism for domain decomposition. The *PaddingSort*, *StepPaddingSort*, *BlockSort* and *PNLoopBSort* variants are implementations of the base parallel algorithm *P-CountingSort* that make skillful use of the specific features of the hardware. The *P-CountingSort*, *PaddingSort*, *StepPaddingSort and BlockSort* variants distribute code blocks over processors, the *LoopBSort* and *PNLoopBSort* algorithms distribute loop iterations over processors. The time complexity of the parallel sort algorithm is determined by the amount of work, performed by one processor and the time necessary to process *p* arrays of size K_{max}:

$$T(N, K_{\max}, p) = \mathrm{O}(N / p + p.K_{\max}), \quad (1)$$

where *N* is the size of the input array; K_{max} is the key range, *p* is the number of processors, due to the domain decomposition and the absence of communication and synchronization between the processors. Formula (1) shows that the execution time of Counting-sort grows slower or equally fast to $N/p + p.K_{max}$.

4.1. P-CountingSort

The *P-CountingSort* variant is based on domain decomposition of the input data. The input array *A* is divided into the necessary number of sequential segments *p* that corresponds to the number of processors in the system. Each segment is associated with a processor and a parallel sort of the segments by the processors is executed. The description determines the problem of parallel implementation of Counting-sort as „trivially parallel" due to the parallel and independent execution of multiple copies of a function (*figure 2*) with different input data. The parallel sorting of an array containing 4**N* records with *P-CountingSort* is illustrated by *figure 5*. In order to avoid critical sections, which cause serious decrease in performance, each processor uses its own copy of the array *start*. A method for associating processors with data is shown on *table 1*.

i	0	1	2	3	4	5	6	7	8	9
A[i].key	1	5	2	2	3	4	4	1	5	3
A[i].number	8	4	2	5	1	3	9	7	6	10

Figure 3. Example of an input array A with ten records

The experiments with *P-CountingSort*, conducted on a multicore processor during the development, illustrated higher execution time of the parallel algorithm compared to the sequential algorithm. The speedup achieved with the parallel algorithm is

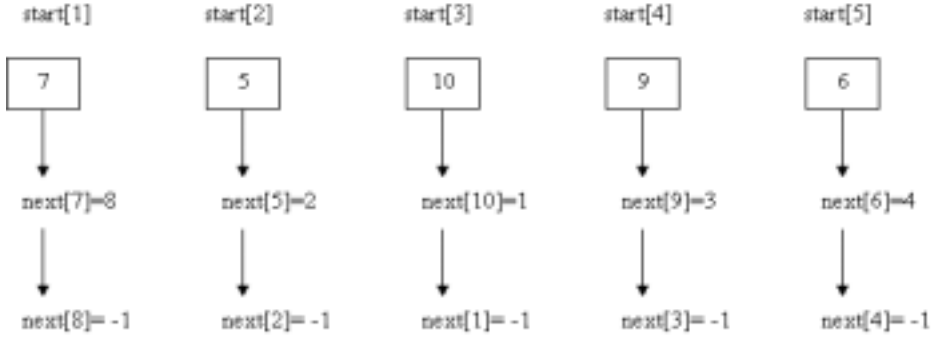

Figure 4. Static implementation of sorting ten records

Table 1. Associating processors with data in *P-CountingSort*

Processor	Reads from	Writes in
1	A[0,N], start1	start1, next
2	A[N+1,2*N], start2	start2, next
3	A[2*N+1,3*N], start3	start3, next
4	A[3*N+1,4*N], start4	start4, next

in direct ratio to the key range. The experimental results led to the creation of a variant of *P-CountingSort* – algorithm *PaddingSort* in order to improve the parallel sort time for small key ranges.

4.2. *PaddingSort*

The reason for the decrease in the performance of the parallel algorithm on a multi-core parallel system for a small key range is the false sharing between cores of the processor ([9,16]). By assumption every core has a private L1 cache memory. At a certain moment in time the L1 cache memory of the core becomes unsynchronized with the cache memory of the other core. Example: When executing two threads on two separate cores, which read and write in adjacent memory cells, the data values, although independent of each other, are written in one cache line. As a result, the memory management system marks the cache line „invalid", no matter that the data, used by the thread has not changed, because the other thread had written in the same cache line. The situation is known as false sharing and increases the access to the main memory which leads to a decreased performance. The performance of the parallel sort algorithm is improved by appending empty memory blocks of size 128 bytes (128 bytes is the size of the sector transferred between the cache memory and the main memory for Intel Q6600 Kensfield) to the arrays *start*. The threads are not writing into adjacent memory cells due to the separating empty blocks. A variant of the parallel sort algorithm with arrays *start* padded by 128 bytes is called *PaddingSort*.

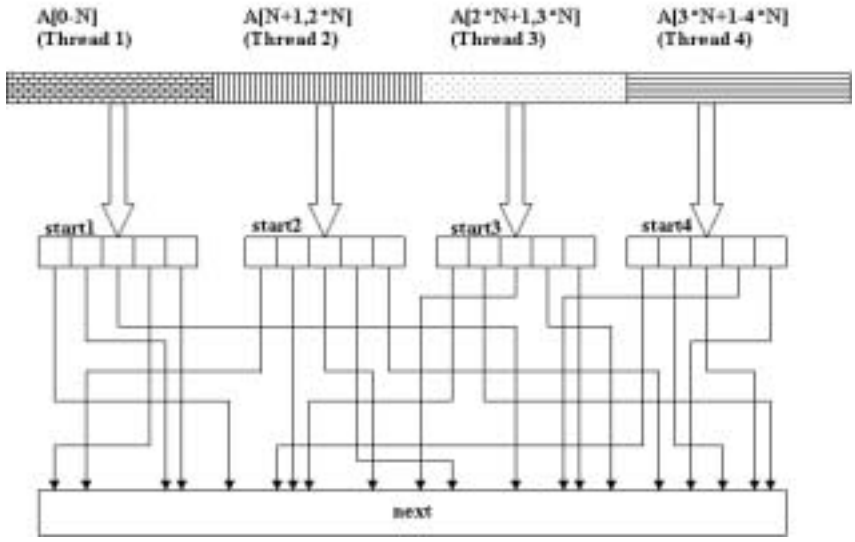

Figure 5. Parallel sorting with *P-CountingSort*

4.3. *StepPaddingSort*

The variant of the parallel sort algorithm called *StepPaddingSort* is a modification of *PaddingSort* with the purpose of optimizing the shared cache memory between core pairs in the processor. The input array *A* is divided into *p*/2 sequential segments, where *p* is the number of processing cores. Each core pair, which shares L2 cache memory, sorts one segment by processing intervals from it. The length of the sequential interval of elements, sorted by each core at one step is defined as *length of the step s*. Processor core 1 and Processor core 2 share a cache memory. If *s*=8, Processor core 1 sorts elements *A*[1-8], *A*[17-32] etc., Processor core 2 sorts elements *A*[9-16], *A*[33-48] etc. The idea is: when Processor core 1 reads element *A*[1], it writes into the shared cache memory elements *A*[1-16] due to the sector transmission of data between cache and main memory. Therefore, when Processor core 2 sorts elements *A*[9-16], they will most probably be located in the L2 cache memory, in a sector that was transferred by the other processor. *StepPaddingSort* is designed to be executed on multi-core parallel systems that have core pairs sharing a cache memory.

4.4. *BlockSort*

The idea of variant *BlockSort* is dividing the data into blocks that fit in the processor's L2 cache memory. The data used by each core is located in arrays *A*, $start_p$ and *next*. The arrays are segmented into blocks with fixed size. A loop is organized – each iteration sorts in parallel two blocks with four processor cores, and two processor cores, which share L2 cache memory, sort one block.

4.5. *LoopBSort*

The idea of the *LoopBSort* variant (LoopB is defined as Loop Bucket) is to execute in parallel the iterations of the main loop in CountingSort() by processors cores where each core uses its own copy of array *start*. Therefore, multiple cores are concurrently distributing adjoining records of the input array. The planning of iterations between the processor cores is modified by applying OpenMP's clauses. Example: Assigning the block size of iterations, processed at once by each core. If the block size is maximal, the algorithm is similar to *PaddingSort*. The conducted experiments show that a block size of 100 000

iterations is optimal. Replacing the OpenMP clause *schedule (static,100000)* with two nested loops with strides 100 000 and 1 respectively decreased the execution time of *LoopBSort*. The *LoopBSort* algorithm uses techniques for avoiding the false sharing.

4.6. *PNLoopBSort*

A variant of the parallel sort algorithm called *PNLoopBSort* is analogical to *LoopBSort*. An exception is using private copies of array *next* for each core (PN -Private Next). The idea of the modification is applying the recommendation for minimal usage of shared data between parallel threads [9]. A disadvantage of the *PNLoopBSort* algorithm is using additional memory of size *p* N**sizeof(*next*) bytes for maintaining copies of the array *next*.

5. Experimental Evaluation

5.1. Description of the Experimental Systems

The experimental evaluation of the algorithms is performed on two multi-core systems:

– *System 1* characteristics: one quad core 2.6GHz processor with 2GB memory. Cores with their own L1 cache memory of size 32KB. Core pairs share L2 cache memory of size 4 MB (totally 8 MB L2 cache memory per processor, 8MB L2 cache memory per System 1). Architecture of the core and the memory – *figure 1*. Operating system – Windows Vista Enterprise.

– *System 2* characteristics: two quad core 2GHz processors with 2GB memory. Cores with their own L1 cache memory of size 8KB. Core pairs share L2 cache memory of size 4 MB (totally 8 MB L2 cache memory per processor, 16 MB cache memory per System 2). Operating system – Windows Server 2003.

5.2. Experimental Data Sets

The key distributions, used for experimental evaluation of the algorithms' behavior are:

– K=*1*: all records have the same key K=1;

– K=*1*(K_{max}): all records have K=1, except twenty records, that have key K= K_{max};

– K=*1(1):1*(K_{max}): half of the records have K=1, the other half – K= K_{max}.

The worst case is expected to be the distribution K=*1(1):1*(K_{max}), where the locality of memory access is highly decreased (there is an alternation of access to elements *start[1]* and *start*[K_{max}]). The random key distributions were implemented with pseudorandom functions from C/C++ libraries and allow using the same sequence of keys in all experiments, which guarantees correctness when comparing the results.

If the records of the input array correspond to the rows of a database's table, the key – to a column of the table and the identifier is defined as the primary key, it could be stated, that in most cases the following distributions of identifiers are observed:

• *Sorted ascending identifiers*: observed in tables without deletions. The identifiers n of the records in array *A[N]* are unique integers, where $n \in [1..N]$ and

$A[1].n \leq A[2].n \leq A[3].n \leq ... \leq A[N].n$

• *Randomly distributed identifiers*: the identifiers n of the records in array *A[N]* are unique integers, where $n \in [1..N]$. The records are not sorted by identifier.

We generated the randomly distributed identifiers by generating sorted ascending identifiers and applying a shuffling function that swaps random identifiers millions of times until their distribution becomes approximately random. The shuffling function, which we developed, uses pseudorandom number generators and generates the same final distribution of identifiers every time for a given N.

5.3. Benchmarking

The benchmarking is performed between the parallel algorithms *P-CountingSort*, *PaddingSort*, *StepPaddingSort, BlockSort*, *LoopBSort* and *PNLoopBSort*. The algorithms are compared based on running time, speedup to the sequential algorithm, influence of the false sharing, key range, key distribution, shared cache memory, identifier distribution, scalability to key range, input data size and processing cores.

5.3.1. Setup

The parallel algorithms are written in C with OpenMP directives [13]. The criterion for comparison of the algorithms *P-CountingSort*, *PaddingSort*, *StepPaddingSort, BlockSort*, *LoopBSort* and *PNLoopBSort* is based on five experiments. We use a technology, known as processor affinity, to associate each program thread with a separate core of the processor.

5.3.2. Results

Influence of False Sharing

Experiments have shown that for small key ranges, the speedup of *PaddingSort* is from 1.4 times to 4 times higher than the speedup of *P-CountingSort*, due to the elimination of the false sharing. The difference between the speedups of the algorithms is in inverse ratio to the key range because the false sharing decreases with the increase of the size of arrays *start* (*figure 6*).

Influence of the Key Range

The experiments have shown that the execution time of all parallel algorithms increases significantly when the key range reaches 100 000 for all values of N and the amount of execution time increase is in direct ratio to the size of the input array. The influence of key range on the execution time in the case of 8 cores is illustrated by *figure 7*. The influence of key range on the execution time is analogical for the rest of the parallel algorithms. The execution time increases because the operational arrays cannot fit into the cache memory.

Influence of the Key Distribution

The experimental results show that in a situation of randomly distributed identifiers and a significant key range K_{max}=10 000 000:

– K=*1* increases the speedup achieved with *PaddingSort* and *LoopBSort* by 0.159(average) and 0.179(average) respectively related to the random key distribution.

– K=*1(m)* decreases the speedup achieved with *PaddingSort* by 0.02(average) and increases the speedup achieved with *LoopBSort* 0.16(average) related to the random key distribution.

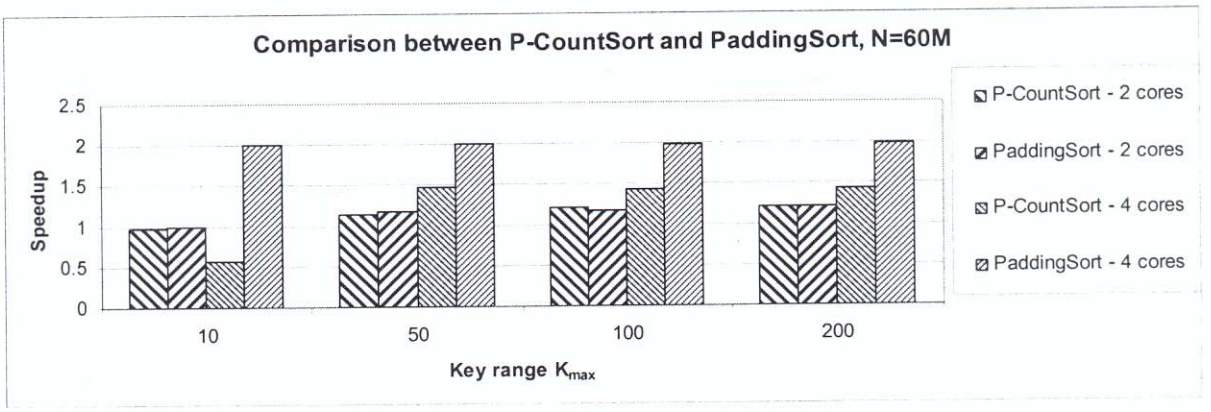

Figure 6. Speedup of *P-CountingSort* and *PaddingSort* when sorting an array of 60 million records with System 1

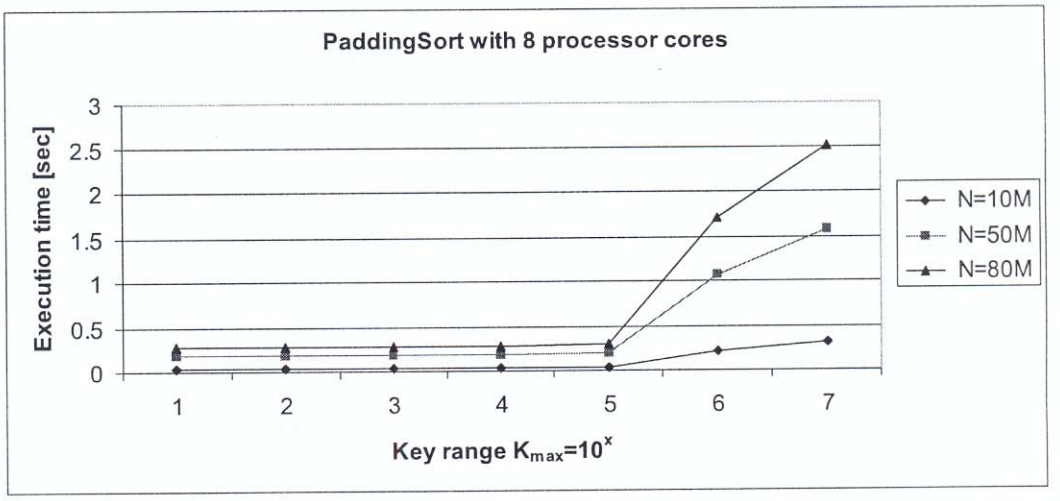

Figure 7. Running time of *PaddingSort* for different key ranges with System 2

– *K=1(1):1(m)* increases the speedup achieved with *PaddingSort* and *LoopBSort* by 0.190(average) and 0.083(average) respectively related to the random key distribution.

In a situation of sorted ascending identifiers and a significant key range K_{max}=10 000 000:

– *K=1* increases the speedup achieved with *PaddingSort* by 0.085 on the average and decreases the speedup achieved by *LoopBSort* by 0.034 on the average related to the random key distribution.

– *K=1(m)* increases the speedup achieved with *PaddingSort* by 0.086 on the average and decreases the speedup achieved by *LoopBSort* by 0.039 on the average related to the random key distribution.

– *K=1(1):1(m)* increases the speedup achieved with *PaddingSort* by 0.068 on the average and decreases the speedup achieved with *LoopBSort* by 0.01 on the average related to the random key distribution.

The key distribution has little influence on the speedup achieved with *PaddingSort*, *LoopBSort*, *StepPaddingSort* and *PNLoopBSort* for a small key range. *Table 2* illustrates the influence of the key distribution on speedup achieved with *StepPaddingSort*, where *y* denotes the standard deviation of the difference between the speedup for a random key distribution α and the speedup for skewed key distribution α_K.

Influence of the Shared Cache Memory

As discovered in our experiments, the only situation when execution with two cores that share a cache memory is beneficial compared to execution with two cores that use separate cache memory is sorting with *PaddingSort*, sorted ascending identifiers and a small key range. In all other situations the execution of the algorithms *PaddingSort*, *StepPaddingSort*, *LoopBSort*, *PNLoopBSort* with two cores that share L2 cache memory achieves lower speedup compared to the execution with two cores that use a separate L2 cache memory.

Influence of the Identifier Distribution

The experiments show that the speedup achieved with the algorithms *PaddingSort*, *StepPaddingSort*, *BlockSort*, *LoopBSort*, *PNLoopBSort* is higher in a situation of randomly distributed identifiers compared to a situation of sorted ascending identifiers, with one exception – sorting with *PaddingSort*, implemented with two cores that share a cache memory and a small key range (*figure 13*).

Scalability to the Key Range

The speedup achieved with the algorithms *PaddingSort*, *StepPaddingSort*, *BlockSort*, *LoopBSort* and *PNLoopBSort* does not increase as the key range increases except for the implementation with 2 processor cores. *PaddingSort* achieved a

Table 2. Influence of the key distribution on the speedup achieved with *StepPaddingSort* for randomly distributed identifiers and key range K_{max}=5 with System 1

	α_K - α	2 cores	4 cores
K=1	Average	-0,211	-0,128
	σ	0,155	0,119
	Max	-0,156	-0,127
K=1(m)	Average	-0,189	-0,116
	σ	0,146	0,115
	Max	-0,143	-0,104
K=1(1):1(m)	Average	-0,193	-0,118
	σ	0,153	0,116
	Max	-0,102	-0,107

speedup of nearly 4.8 times compared to the sequential algorithm when executed with 2 cores on separate processors for key range larger than 1 000 000 and different size of the input array (*figure 8*).

Scalability to Data Size

The speedup achieved with the algorithms *PaddingSort, StepPaddingSort and BlockSort* is in a direct ratio to the input array size *N* in a situation of randomly distributed identifiers, 4 cores and small key range (*figure 15*). The speedup achieved with algorithm *PNLoopBSort* is in a direct ratio to the input array size *N* in a situation of randomly distributed identifiers and a significant key range (*figure 15* and *figure 16*).

Scalability to the Number of Processing Cores

Experiments with System 1 have shown that the speedup achieved with *PaddingSort, StepPaddingSort and LoopBSort* increases with the increase of the number of cores for randomly distributed identifiers and a small key range (*figure 15* and *figure 16*). The speedup achieved with *PaddingSort and StepPaddingSort* increases with the increase of the number of cores for randomly distributed identifiers and a significant key range (*figure 15* and *figure 16*). In other situations, the examined parallel algorithms are not found to be scalable to the number of processing cores(*figure 13* and *figure 14*). We observe from *figure 8* that the speedup achieved with *PaddingSort* and 8 cores is not tangibly higher than the speedup achieved with *PaddingSort* and 4 cores for different key ranges with System 2. In a situation of a significant key range, sorting with *PaddingSort* and 4 cores led to higher speedup compared to sorting with *PaddingSort* and 8 cores. The algorithm's scalability to number of processing cores is not optimal because the cores compete for memory bandwidth and there is a contention between the processors over the memory bus [15].

Comparison Based on Running Time

It was discovered empirically that the step length s = 100 000 is optimal for the execution time of the *StepPaddingSort* algorithm. While experiments have shown that the size of the block is a not a factor with a significant influence over the execution time of *BlockSort* for sizes from 1 MB to 8 MB, the maximal results were achieved with a block size of 7 MB. The experiments have shown that planning the iterations

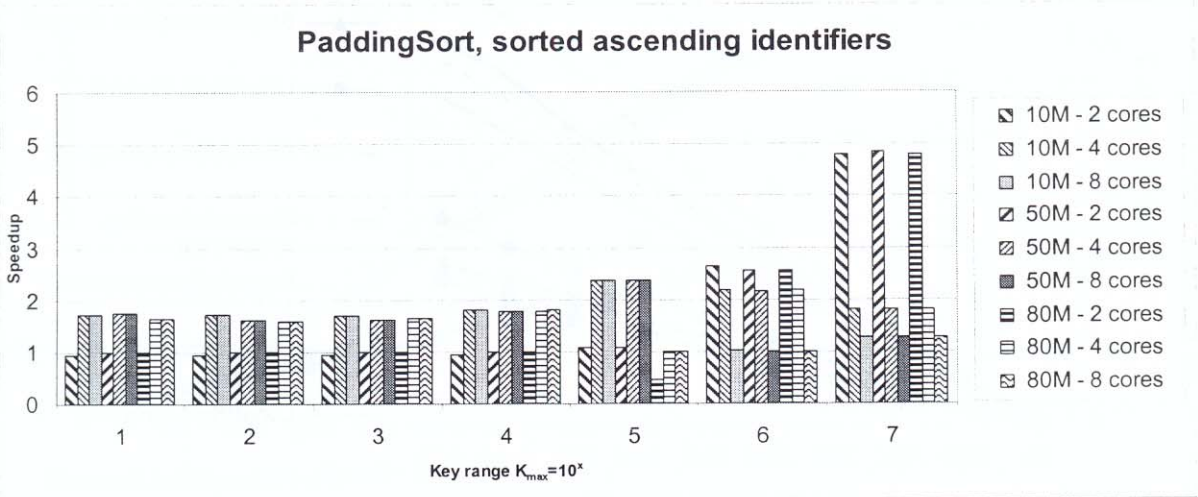

Figure 8. Speedup of *PaddingSort* for different key range, number of cores and input array size with System 2

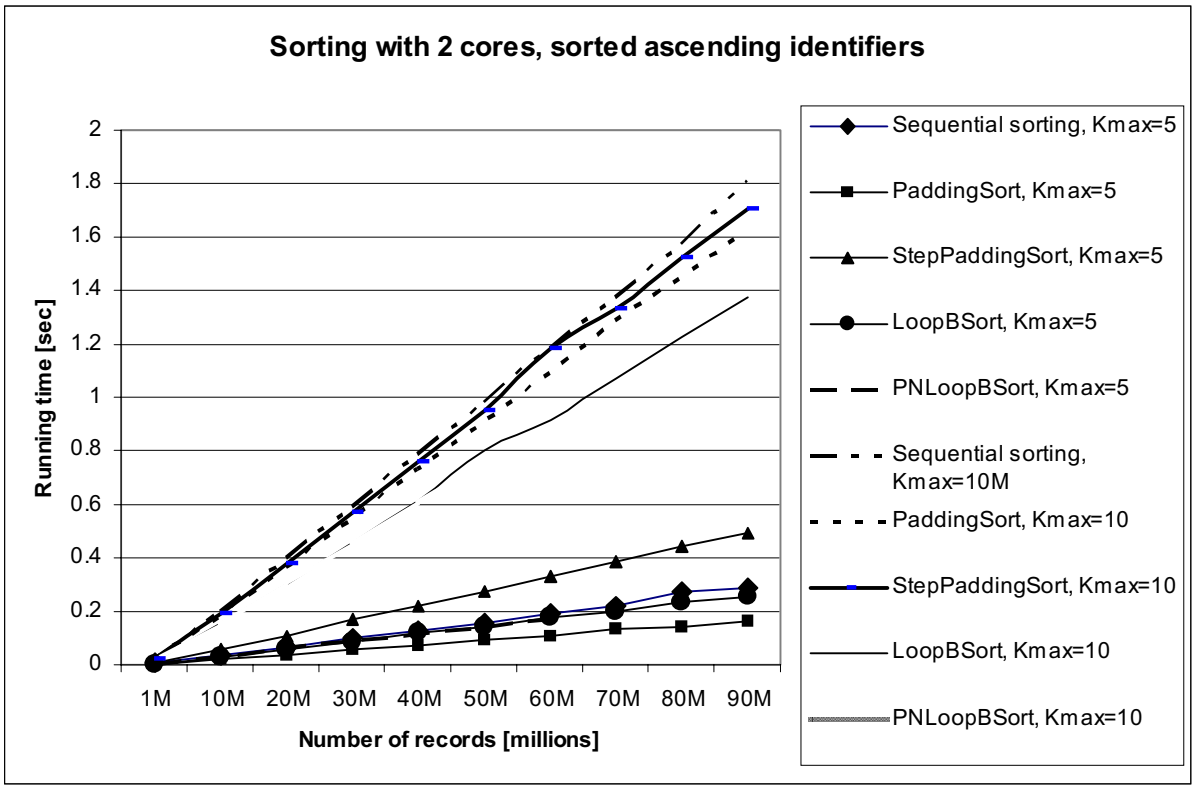

Figure 9. Running time for sorting with two cores and sorted ascending identifiers with System 1

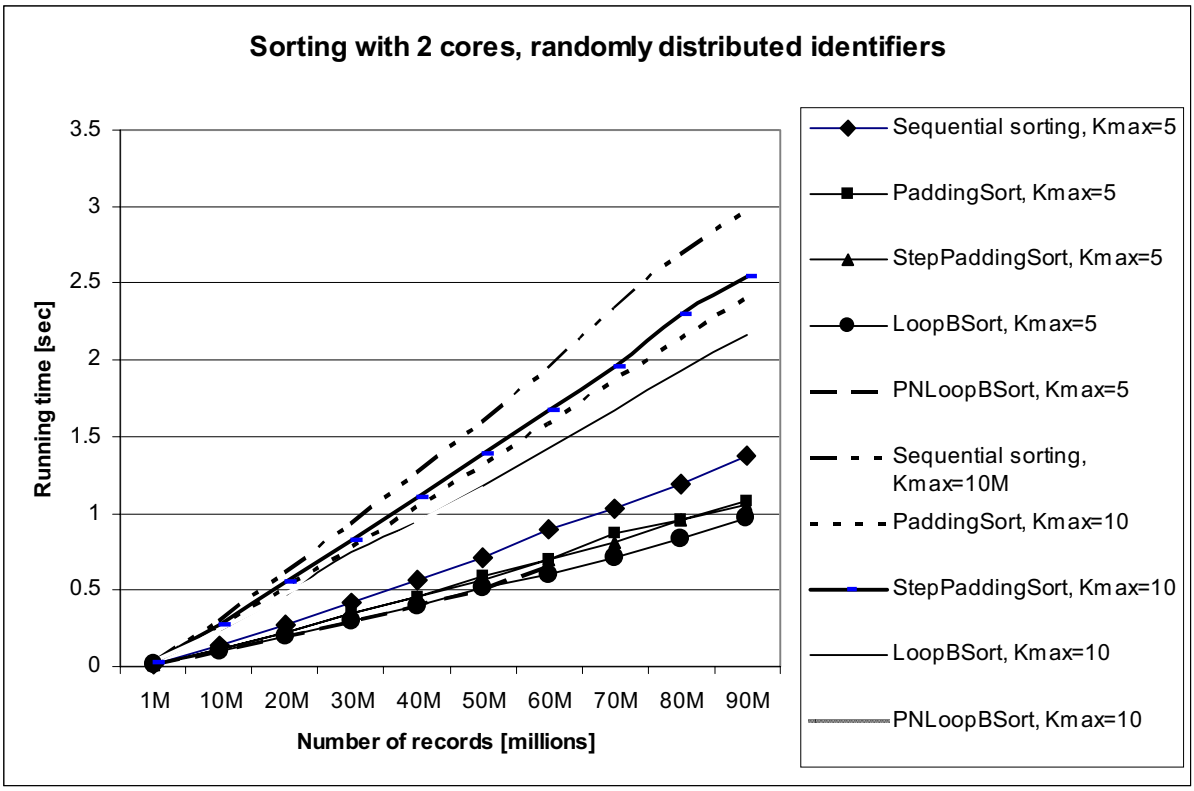

Figure 10. Running time for sorting with two cores and randomly distributed identifiers with System 1

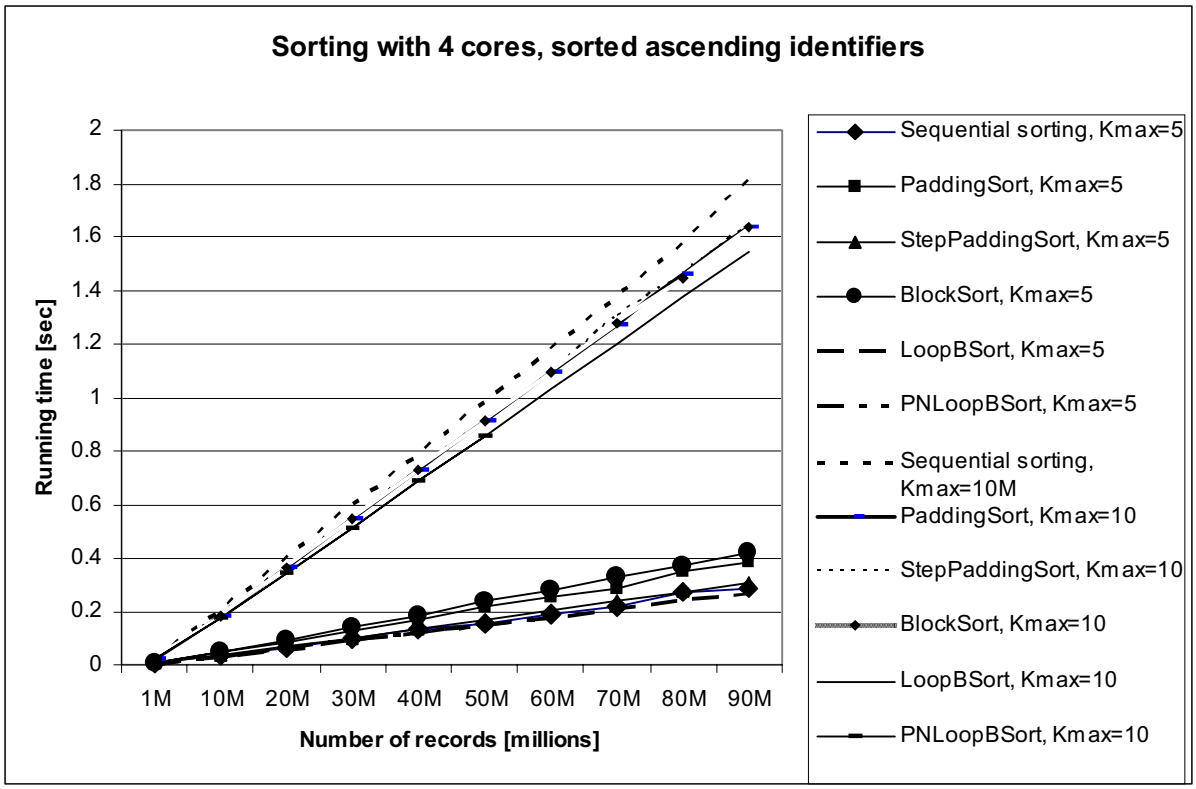

Figure 11. Running time for sorting with four cores and sorted ascending identifiers with System 1

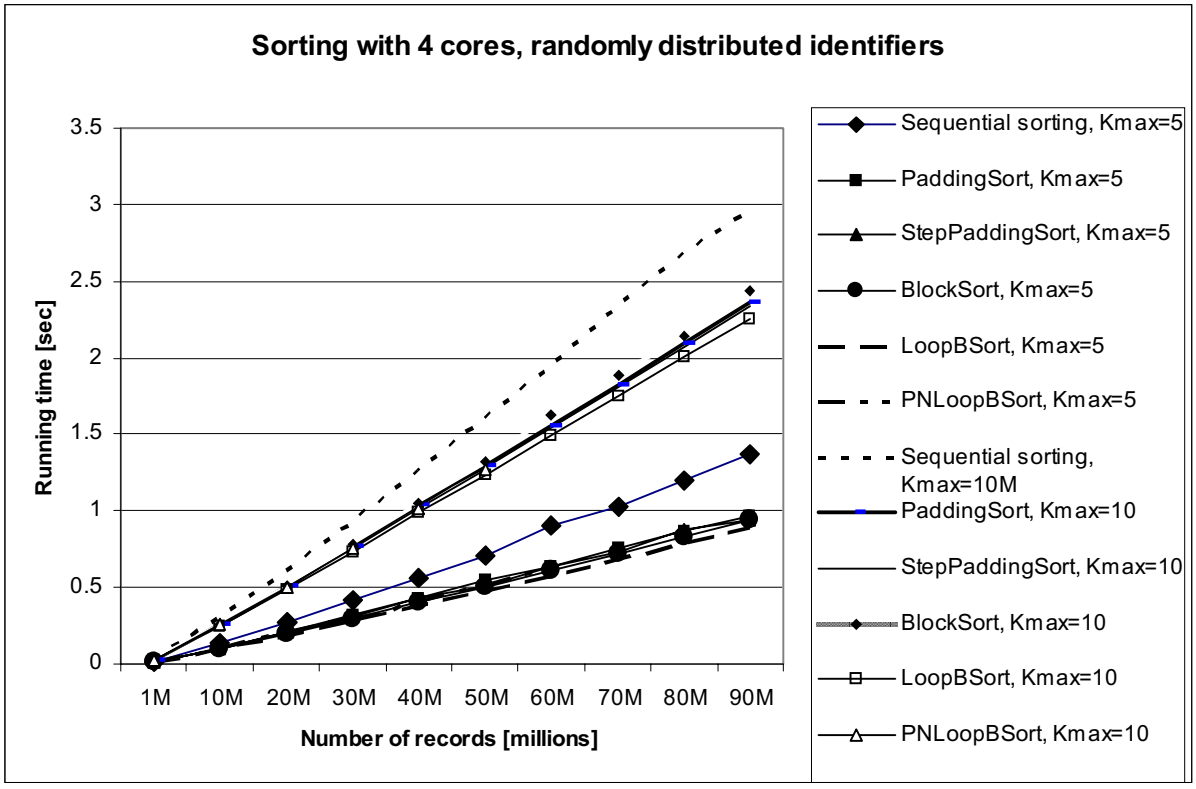

Figure 12. Running time for sorting with four cores and randomly distributed identifiers with System1

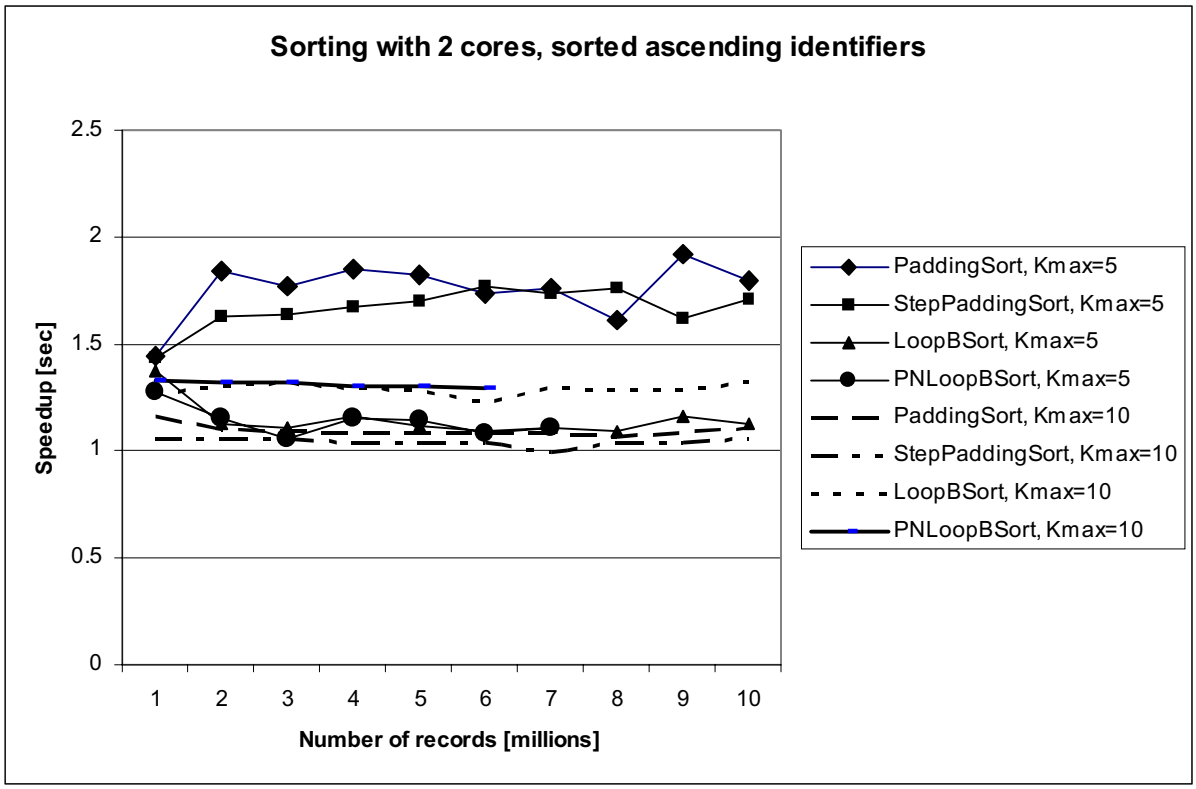

Figure 13. Speedup for sorting with two cores and sorted ascending identifiers with System 1

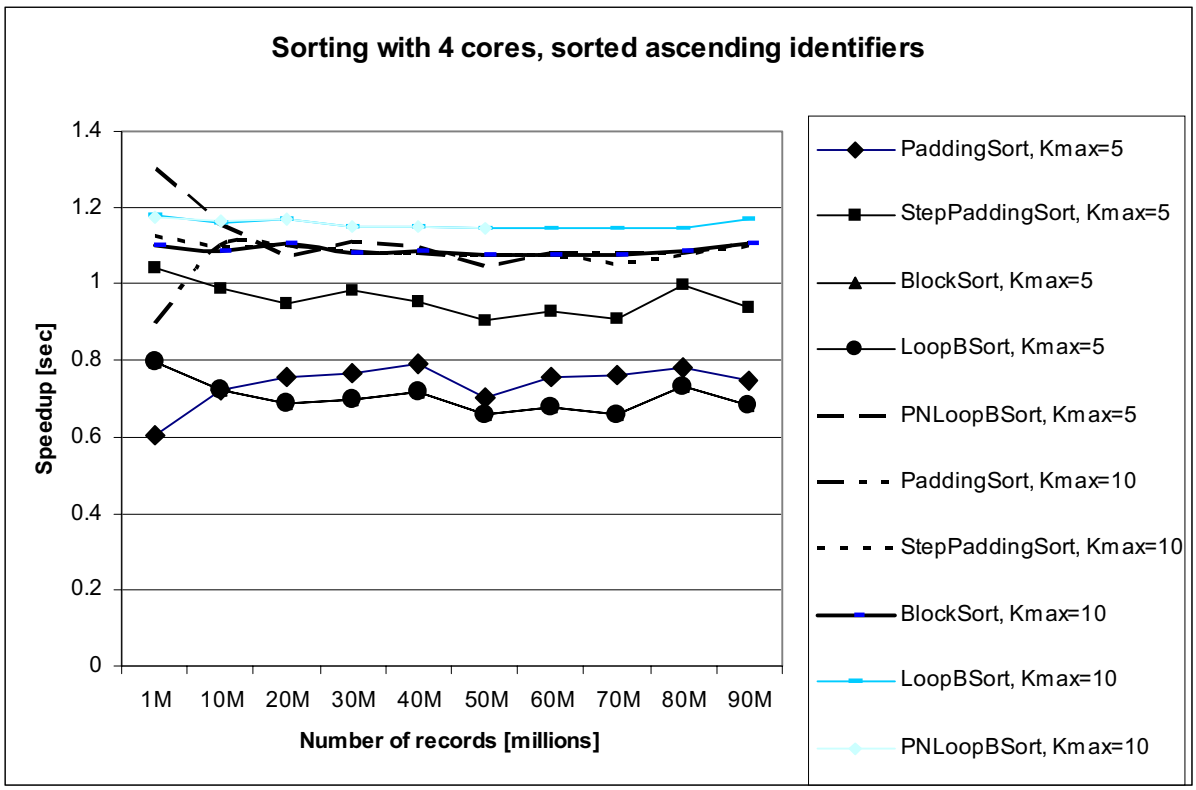

Figure 14. Speedup for sorting with four cores and sorted ascending identifiers with System 1

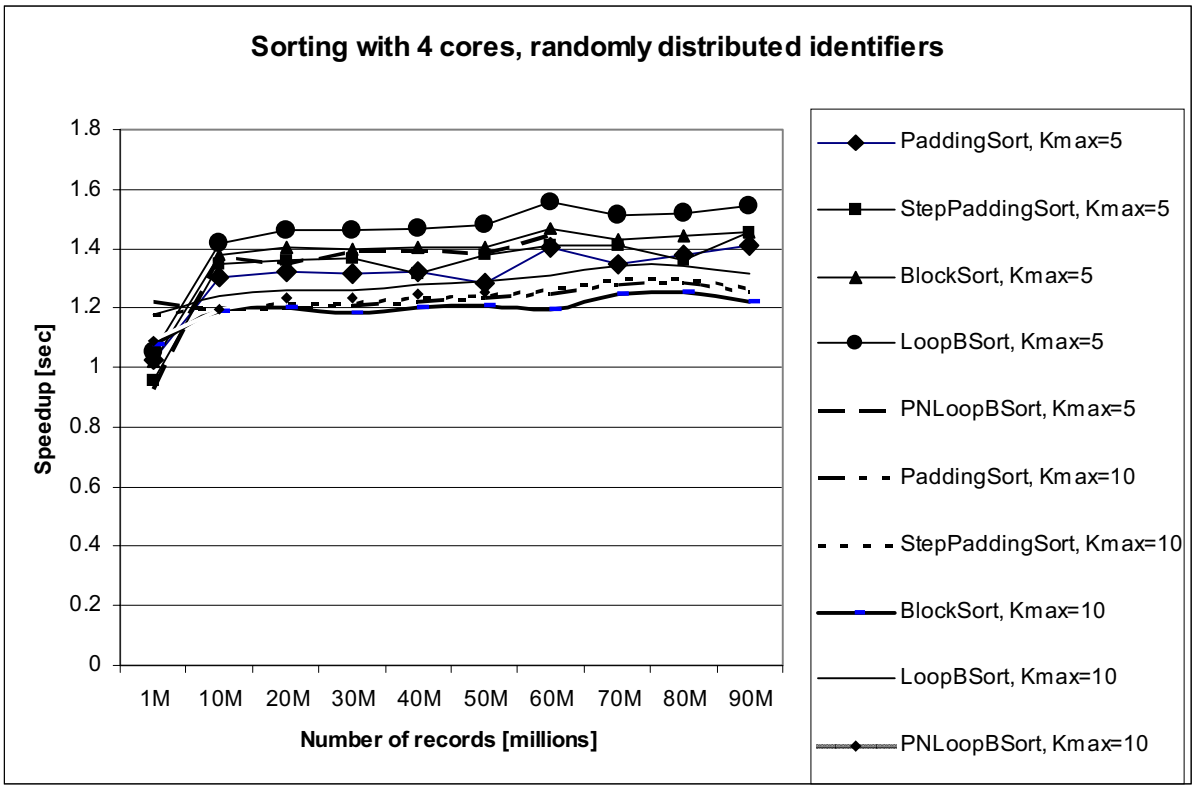

Figure 15. Speedup for sorting with four cores and randomly distributed identifiers with System 1

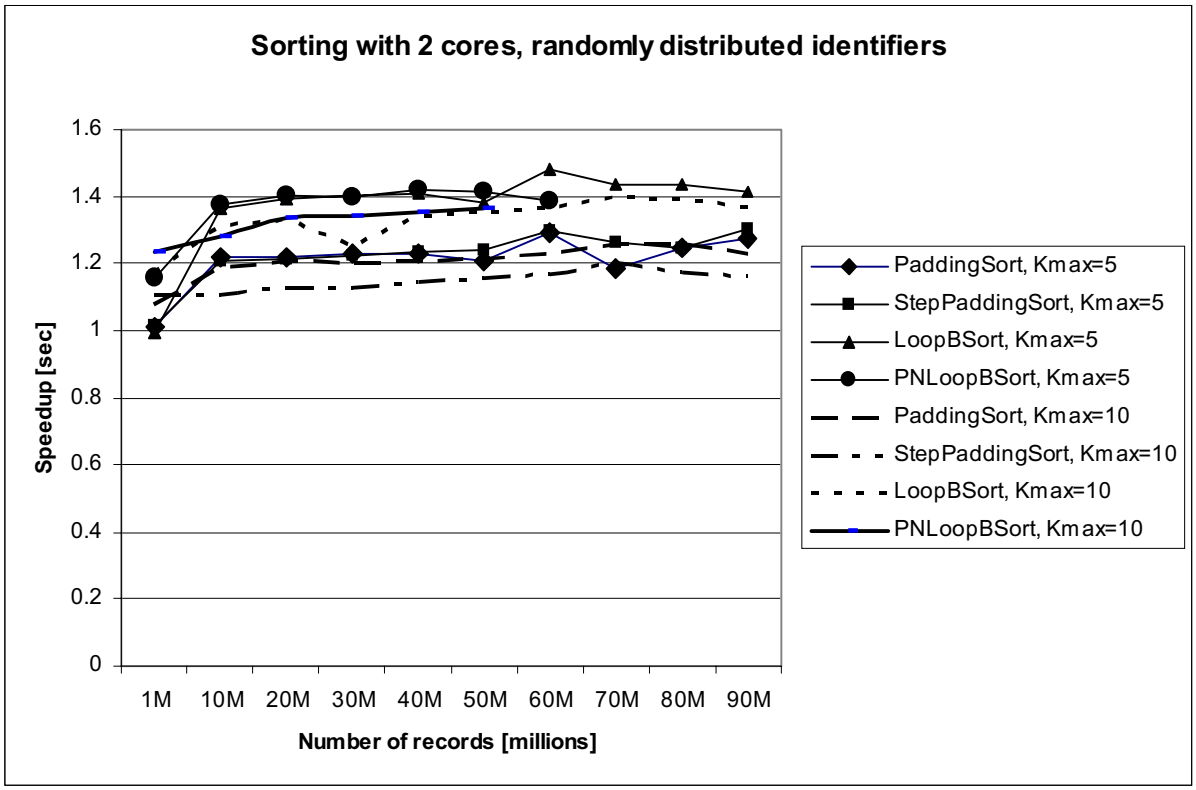

Figure 16. Speedup for sorting with two cores and randomly distributed identifiers with System 1

in chunks of 100 000 is optimal for the execution of the algorithms *LoopBSort* and *PNLoopBSort*. As illustrated by *figures 9 –12*, the running time of all sort algorithms increases linearly with the size of the input array. The linear dependency is anticipated because the segments of the input array are sorted in parallel by distributing the records into buckets and there is no communication or synchronization between the processor cores.

Sorting with *PaddingSort* is fastest in a situation of sorted ascending identifiers, a small key range and implementation with two cores. In all other situations *LoopBSort* is the fastest sorting algorithm for any key range and input array size. *PNLoopBSort* is slightly faster than *LoopBSort* in a situation of a significant key range and implementation with two cores that share a cache memory.

StepPaddingSort has better performance compared to *PaddingSort* in a situation of four processor cores and:

– a significant key range and randomly distributed identifiers;

– a small key range and sorted ascending identifiers.

Sorting with algorithm *BlockSort* is faster compared to *PaddingSort* only in a situation of a small key range, four cores and randomly distributed identifiers.

6. Conclusions and Future Work

PaddingSort is recommended for sorting records that are arranged by identifier in ascending order with keys in a small range and for systems that have two cores. Using *PNLoopBSort* is beneficial in a situation of a significant key range and two cores, although it requires considerable amount of additional memory and *LoopBSort* can be used instead. In all other situations sorting with *LoopBSort* is recommended. The investigated variants of the parallel algorithm perform faster when they process the input array in blocks of 100 000 records. All variants except *PaddingSort* and *StepPaddingSort* are recommended for systems with separate L2 caches and all variants except *PaddingSort* perform better in a situation of randomly distributed identifiers. *LoopBSort* is less stable to unfavorable key distribution compared to *PaddingSort* but the decrease in speedup is not significant. The algorithms' scalability related to the key range, the input data size and the number of processing cores is not optimal, due to the cache memory and the memory bus sharing.

The variant *PaddingSort* does not have an optimal scalability on multi-core systems with shared L2 cache memory. However, sorting with *PaddingSort* is very fast (nearly 4.8 times faster than sequential sort) when performed with two separate processors on a multiprocessor system.

The future research in the area of parallel sort algorithms for integers could be directed towards:

– Experimental evaluation with multiprocessor systems equipped with 2 and more processors.

– Extending the parallel algorithm to clusters of workstations and MPI.

– Applying existing techniques for optimization of OpenMP applications on architectures with shared caches, such as large page size [12] and partitioning and privatizing the shared arrays *A* and *next* [21].

– Employing the producer-consumer model, specifically designed for shared cache architectures. We are currently researching a variant of *PaddingSort – PCPaddingSort*, where one core is processing elements from the input array A, while the other core is writing to arrays *start* and *next* [9]. The initial experimental results are encouraging. The workload imbalance between the reader and the writer needs optimization.

References

1. Alsabti, K. and S. Ranka. *I*nteger Sorting Algorithms for Coarse-Grained Parallel Machines. Fourth International Conference on High-Performance Computing, 1997 Proceedings, Dec. 1997, 159-164.

2. Ananth, Grama, Anshul Gupta, George Karypis, Vipin Kumar. Introduction to Parallel Computing, Second Edition. Addison Wesley, 2003.

3. Taniar, David and J. Wenny Rahayu. *S*orting in Parallel Database Systems, HPC '00: Proceedings of the Fourth International Conference on High-Performance Computing in the Asia-Pacific Region, 2, IEEE Computer Society, 2000, 830.

4. Ezequiel, Herruzo and Guillermo Ruiz and J. Ignacio Benavides and Oscar Plata. A New Parallel Sorting Algorithm Based on Odd-Even Mergesort. PDP '07: Proceedings of the 15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, 2007, 18-22.

5. Garber, B. A. and D. Hoeflinger and Li Xiaoming and M. J. Garzaran, and D. Padua. Automatic Generation of a Parallel Sorting Algorithm. IEEE International Symposium on Parallel and Distributed Processing, 2008. IPDPS 2008, April 2008, 1-5.

6. Goetz, Graefe. Implementing Sorting in Database Systems. – *ACM Comput. Surv.*, New York, NY, USA, 8, 2006,10.

7. Inoue Hiroshi and Takao Moriyama and Hideaki Komatsu and Toshio Nakatani. AA-Sort: A New Parallel Sorting Algorithm for Multi-Core SIMD Processors. PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques, 2007, 189-198.

8. Hongzhang, Shan and Jaswinder Pal Singh. Parallel Sorting on Cache-coherent DSM Multiprocessors. Supercomputing '99: Proceedings of the 1999 ACM/IEEE Conference on Supercomputing (CDROM), 1999, 40.

9. Intel 64 and IA-32 Architectures Optimization Reference Manual. http://www3.intel.com/design/processor/manuals/248966.pdf

10. Kodama, Y., M. Sato, H. Sakane, S. Sakai, K. Hanpei, Y. Yamaguchi. Parallel Execution of Radix Sort Program Using Fine-grain C

ommunication. International Conference on Parallel Architectures and Compilation Techniques, 1997. Proceedings, Nov 1997, 136-145.

11. Süβ, Michael and Claudia Leopold. A User's Experience with Parallel Sorting and OpenMP. Proceedings of the Sixth Workshop on OpenMP (EWOMP'04), 2004.

12. Noronha, R., D. K. Panda. Improving Scalability of OpenMP Applications on Multi-core Systems Using Large Page Support. IEEE International Parallel and Distributed Processing Symposium, 2007, IPDPS 2007, March 2007, 1-8.

13. OpenMP Application Program Interface. http://www.openmp.org/mp-documents/spec30.pdf

14. Parallel Bubble-sort and Parallel Quick-sort Implementations. http://www.metz.supelec.fr/metz/recherche/ersidp/Projects/Paralgo/Parsort/parbqsort1/Root.html

15. Sandia National Laboratories More Chip Cores Can Mean Slower Supercomputing. Sandia Simulation Shows. http://www.sandia.gov/news/resources/releases/2009/multicore.html

16. Akhter, Shameem and Jason Roberts. Multi-Core Programming: Increasing Performance through Software Multithreading. Intel press, 2006.

17. Stoichev, Stoicho D. Synthesis and Analysis of Algorithms. Sofia, 2007

18. Tsigas, P. and Yi Zhang. A Simple, Fast Parallel Implementation of Quicksort and its Performance Evaluation on SUN Enterprise 10000. Eleventh Euromicro Conference on Distributed and Network-Based Processing, 2003. Proceedings, 2003, 372-381.

19.Zhao, X. and N. J. Martin and R. G. Johnson. PPS-A Parallel Partition Sort Algorithm for Multiprocessor Database Systems. DEXA '00: Proceedings of the 11th International Workshop on Database and Expert Systems Applications, IEEE Computer Society, 2000, 635.

20. Yu-lung, Lo and Yu-chen Huang. Effective Skew Handling for Parallel Sorting in Multiprocessor Database Systems. International Conference on Parallel and Distributed Systems, IEEE Computer Society, 2002, 151.

21. Zhiyuan, Li. Reducing Cache Conflicts by Partitioning and Privatizing Shared Arrays. 1999 International Conference Parallel Architectures and Compilation Techniques, 1999, Proceedings, 1999, 183-190.

Manuscript received on 24.10.2008

*Engineer **Svetlana P. Marinova** is a PhD candidate at Technical University of Sofia, Bulgaria, advised by prof. Stoicho Stoichev. Svetlana received BSc in Computer systems and Control from Technical University of Sofia in 2003 and expects to receive her PhD in 2010. She interned at Microsoft Research in Redmond, USA and has worked as a research associate in Sofia University St. Kliment Ohridski.*

Svetlana's current research interests are in intelligent systems, data warehousing and cryptography. Her PhD work has been focused on designing efficient parallel algorithms in the context of the contemporary information systems and database management systems.

Svetlana has experience as a teaching assistant in databases, programming environments and cryptographic methods for protecting data in databases.

Contacts:

Department of Computer Systems and Control
Technical University-Sofia
8, Kliment Ohridski St.,
1000 Sofia, Bulgaria
tel. +359 885000477
e-mail: a_doct@tu-sofia.bg

*Professor **Stoicho D. Stoichev** received MSc in Radio engineering at Technical University-Sofia in 1958, PhD in Computer Science at St. Petersburg University of Electrical Engineering „LETI" in 1965 and Doctor of Technical Sciences in Computer Science at Technical University-Sofia in 1991. His research interests include synthesis and analysis of algorithms, graph isomorphism algorithms and artificial intelligence. He is senior member of IEEE.*

Contacts:

Department of Computer Systems and Control
Technical University-Sofia
8, Kliment Ohridski St.,
1000 Sofia, Bulgaria
tel. +359 2 965 3385
fax +359 2 68 53 43
e-mail: stoi@tu-sofia.bg