ISSN:0975-9646

An Efficient Parallel Sorting Algorithm for Multicore Machines

D.Abhyankar, M.Ingle
School of Computer Science, D.A. University, Indore M.P. India

***Abstract*— Sorting an array of integers is one of the most basic problems in Computer Science. Also it is an issue in high performance database applications. Though literature is imbued with a variety of sorting algorithms, different architectures need different optimizations to reduce sorting time. This paper presents a Multicore ready parallel sorting algorithm which has been designed with Multicore/Manycore architecture in mind. Our study shows that the proposed algorithm is excellent for large input size and multiple free cores. In essence algorithm has potential to be a success in situations when one has large input and machine is a Multicore machine. The paper does not neglect overhead involved with parallel programming and suggests two system calls to check the availability of free cores and to reserve a core for a fixed time quantum.**

***Keywords*— Multicore.**

I. INTRODUCTION

There have been abundant computer applications which need sorting as a key component. Since SQL operations use it as an internal database subroutine, all database applications gain advantage of an efficient sorting algorithm. Also sorting is a must for some rudimentary database operations like a creation of indices and binary searches. Sorting is functional in operations like finding closest pair, determining an element's uniqueness, finding k^{th} largest element, and identifying membership. Many practical applications in computational geometry need sorting. For instance sorting is used to find the convex hull in computational geometry. Applications that use sorting include supply chain management, bioinformatics and computer graphics [13].

Multi-core processors are extensively used across many application areas including general-purpose, embedded, network, digital signal processing (DSP), and graphics. However existing softwares are not able to exploit the multiple cores available in the machine. This is partly because of sequential algorithms implemented by existing software. For instance Libraries of C, C++ and Java implement variant of Quicksort which is a sequential sorting algorithm. Designing a Multicore ready or Manycore ready sorting to exploit multiple cores is one of the central ideas of the paper. Ofcourse there are other Multicore ready sorting algorithms but the presented algorithm is superior to existing Multicore algorithms in terms of load balancing and space. In addition paper observes that there are certain situations where it is wise to drop the idea of parallel sorting. For small input size the idea of parallel or Multicore sorting should be relinquished. Proposed algorithm exploits the multiple cores in a better way because it has excellent load balancing features. Also it offers a solution that is space efficient for large arrays.

II. LITERATURE SURVEY

Last 50-60 years have produced a surprisingly large number of sorting algorithms. Focus of this section is on the algorithms that have the potential to exploit the Multicore capability skilfully. Quicksort is considered to be one of the fastest sequential sorting algorithms, but the implementations which can exploit parallel architectures efficiently are not feasible because of the load balancing characteristics of Quicksort especially in the initial stage of sorting. In contrast parallel Mergesort, parallel Radix sort and Mapsort are three decent alternatives on Multicore machines, but they have their problems too. Radix Sort can also be parallelized effectively, but its performance depends on the support for handling simultaneous updates to a memory location within the same SIMD register. Also, parallel Mergesort and Map sort though achieve good load balance, but are not in place and demand more memory. On the other hand proposed algorithm is in place plus it has excellent load balance. Proposed algorithm is an outcome of a perfect selection of lower level sequential algorithms [9, 10, 11, 12, 13, and 14].

III. BACKGROUND

Transistor density has grown steadily in last few years to meet the rising performance requirements of business applications, but this increasing density leads to an extremely sharp increase in power consumption of processor, which generates tremendous amount of heat. Though faster processors are one way to improve the performance, other approaches produce the performance without any upsurge in the clock speed, power consumption and heat. Indeed excellent overall performance can be achieved by reducing the clock speed while increasing the number of processing units called cores. For instance, excellent performance can be gained by a shift from single core to many cores which has an obvious advantage of keeping the heat considerably low [9].

Since the proposed algorithm is designed for Multicore architecture a decent understanding of Multicore architecture will enable us to understand the context in which proposed algorithm will be relevant. A Multicore machine is a machine with two or more independent actual CPUs (called "cores"), which are the units that read and execute instructions. Cores are integrated onto a single integrated circuit die (chip multiprocessor), or onto multiple dies in a single chip package. In many-core machine number of cores is much larger than what traditional Multicore machines have. In Manycore machine CPU count is roughly in the range of several tens; above this range network on chip technology is more effective [15].

The improvement in performance gained by the use of a multi-core processor depends largely on the software algorithms used and their implementation. In particular, possible gains are limited by the fraction of the software that can be parallelized to run on multiple cores simultaneously; this effect is described by Amdahl's law. Massively parallel problems may realize speedup factors near the number of cores, or even more if the problem is split up enough to fit within each core's cache(s), avoiding use of much slower main memory. Most applications, however, are not accelerated so much unless programmers devote a great deal of effort in re-designing the whole solution. Thus it is crystal clear that parallelization of software (algorithms) is a significant current topic of research [15].

Due to power consumption and other reasons, microprocessors are being built with multiple processors on chip. Multi-cores are already on most desktops, and number of cores is expected to increase steeply in the near future. Computer science research needs to address the multitude of challenges that we need to face come with this shift to the Multicore era. Our focus is on developing a sorting algorithm which can exploit the full potential of Multicore and Manycore machines. Also we have tried to understand in which situation Multicore ready sorting algorithm should be more efficient and where we should prefer to avoid Multicore ready sorting algorithms.

For an algorithm designer it would be beneficial to understand some architectural details of Multicore machines. In Multicore machine at one hand we observe that CPU's have their own private caches and at the same time they happen to share a common cache. So an algorithm needs to achieve decent performance in both types of caches. If an algorithm designer achieves overlap in data and code than s/he can achieve excellent shared cache performance. But achieving excellent shared cache performance causes the private cache performance to deteriorate. On the other hand if s/he manages independent data and code for each core then private chaches will deliver superior performance but the performance of shared cache is guaranteed to go down. So an algorithm designer ought to satisfy competing needs [15].

IV. Proposed Algorithm

This section formulates a novel Multicore ready sorting algorithm which has been discussed in subsection 5.1 and 5.2. Subsection 5.1 is an informal outline of the algorithm, whereas 5.2 treats the algorithm at more formal level.

A. Informal Description Algorithm1 and Algorithm2

Algorithm1

It is assumed that Machine has n free cores. If input array is small then Program will invoke Heapsort. Avoiding parallelism for small inputs is an intensely practical idea. Else it will divide the array in n equal parts and sort them parallel on n different cores using Heapsort. Repeatedly use free cores to merge consecutive subarrays using an adaptive in place merges (Algorithm 2) until there are sub arrays to merge. Detailed java like psuedocode of the proposed algorithm is present in subsection 5.2. Algorithm 1 needs algorithm 2 as a lower level routine. Algorithm 2 is a variation of merge algorithm used in standard C++.

Algorithm2

Algorithm 2 assumes we have 2 sorted consecutive sub arrays A and B. If at least one of the A and B is empty then terminate trivially. A can be divided into two types of sub arrays A1 and A2. A1 will have elements which do not involve inversions with elements of B. A2 will have inversions with elements of B. One of the A1 and A2 may be empty. In the same way B can be divided into B1 and B2. B1 will have elements which do not involve inversions with elements of A. B2 will have inversions with elements of A. One of the B1 and B2 may be empty. Then Transfer A2 into B and transfer B2 into A. Restore the sorting order of transferred elements. Now we have at most two A components and 2 B components to merge which can be merged by calling Algorithm 2 on free cores.

B. Formal Description of Algorithm1

Algorithm 1 has been presented using Java like psuedocode. This psuedocode presentation has been derived from the multithreaded Java implementation of the algorithm. Java was a natural choice for implementation since Java offers multithreading.

```
class x implements Runnable{
    int[] a;     int p;     int r;
    public x(int[] b, int p1, int r1)
    {
        a = b;
        p = p1;
        r = r1;
    }
    public void run()
    {
      /* this is expected to run parallel in
favourable circumstances */
      ManyCore1.HeapSort(a,p,r);
    }
}

class M implements Runnable{
    int[] a;     int p;     int Chunk;
    public M(int[] b, int p1, int Chunk1)
    {
        a = b;
        p = p1;
        Chunk = Chunk1;
    }
    public void run()
    {
        /* expected to run parallel in
favourable circumstances */
        ManyCore1.Merge(a, p, Chunk);
    }
}

public class ManyCore1 {

   private final static Random generator =
new Random();
   public static void Merge1(int[] a,int p,
int Chunk, int r )
   {
       // Already known
   }
   public static void Merge(int[] a, int p,
int Chunk)
   {
       //Already known
```

```
   }

   public static void MergeRound(int[] a,
int n, int ChunkSize)
    {
        int MergeCount = n/(2*ChunkSize);
        int j = 0; int p = 0;
        Thread[] t = new Thread[MergeCount];
        while(j<MergeCount)
        {
            M ob = new M(a,p,ChunkSize);
            t[j] = new Thread(ob);
            // causes parallel execution
            t[j].start();
            p = p + (2*ChunkSize);
            j++;
        }
         j = 0;
         while(j < MergeCount)
         {
             t[j].join();
             j++;
         }
         p = p - (2*ChunkSize);
         Merge1(a,p,ChunkSize,n-1);

    }
    public static void ParallelMerge(int
a[], int n, int m)
    {
        int ChunkSize = (n/m);
         while(ChunkSize < (n))
         {
         MergeRound(a,n,ChunkSize);
         ChunkSize = ChunkSize*2;
         }

    }
    public static void ManyCoreSort(int[] y,
int n,  int m)
   {
       if(n < m)
       {
          InsertionSort(y, 0,n-1);
          return;
       }
       int Chunk = (n/m);
       int p = 0;
       int r = Chunk -1;
       Thread[] t = new Thread[m];
       int i = 0;
       while(i < (m-1))
           {
           x ob = new x(y, p, r);
           t[i] = new Thread(ob);
            t[i].start();
            i++;
            p = p + Chunk;
            r = r+Chunk;
          }
        HeapSort( y,p,n-1);
        i = 0;
        while(i<(m-1))
        {
           t[i].join();
           i++;
        }
   }

}
```

V. Results

Experiments on parallel sorting have convinced us that parallel sorting should be used when input size is large, and when you have enough free CPUs to realize actual parallelism. To implement the proposed algorithm Java threads were instrumental. Because Multithreading has overheads, parallelism on small input will be unable to outweigh the multithreading overheads. Multithreading for small input size will almost always be counterproductive. In our opinion parallelism should be opted where input size is large and problem has enough inherent parallelism. In addition to that there are software engineering concerns. Parallel programs are genuinely complex and are certain to end up in a code that is not neither compact nor elegant. Even when problem has inherent parallelism issue is how many cores or CPU's are free. If there are too few CPUs free then creating new threads is counterintuitive and counterproductive. It would have been a lot better situation if operating system or platform can inform about the availability of free CPUs and a program can reserve a free CPU for some fixed time quantum. If Operating system informs that no free CPU's then a program should avoid creating new threads. This study proposes two system calls FreeCPUCount and ReserveCPU. It seems that these system calls have the enormous potential to reduce the multithreading overheads.

VI. Conclusion

Proposed algorithm invokes Heapsort and in place Merge algorithm as a lower level routine and that ascertains that suggested algorithm remains in place. Selected merge variation is an in place Merge algorithm, which is excellent at load balancing. It can be observed that often available cores will be busy in merging. It is easy to see that performance of private cache of the cores will be excellent because cores will work on disjoint data sets. The only concern is the performance of the shared cache because most of the times core will work on disjoint data sets. Fortunately only data is disjoint; Code can be shared. Shared code is one of the salient plus point of the proposed algorithm. Proposed algorithms avoids multithreading for small input size, because parallelism on small inputs leads to the performance which is modest at best. Avoiding parallelism or multithreading for small input is one of the pragmatic plus point of the presented algorithm.

For a multithreaded algorithm to be successful it has to achieve excellent load balancing, excellent private cache performance and excellent shared cache performance, plus overall memory demand should be low. In addition to that an algorithm needs large input size to outweigh the overheads involved in parallel programming. Moreover at run time it should have enough free CPUs to exploit the parallelism of the algorithm. In this light one can see that the algorithm needs large input and a lot of free cores. Ultimately performance of any parallel algorithm depends on the input size and the availability of free cores. The proposal of system calls FreeCPUCount and ReserveCPU has the potential to reduce the overheads associated with parallelism.

REFERENCES

[1] D. E. Knuth, *The Art of Computer Programming*, Vol. 3, Pearson Education, 1998.

[2] C. A. R. Hoare, *"Quicksort"* Computer Journal5 (1), 1962, pp. 10-15.

[3] S. Baase and A. Gelder, *Computer Algorithms:Introduction to Design and Analysis*, Addison-Wesley, 2000.

[4] J. L. Bentley, *"Programming Pearls: how to sort" Communications of the ACM*, Vol. Issue 4, 1986, pp. 287- ff.

[5] R. Sedgewick, *"Implementing quicksort Programs" Communications of the ACM*, Vol. 21, Issue10, 1978, pp. 847-857.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001.

[7] G. S. Brodal, R. Fagerberg and G. Moruz, *"On the adaptiveness of Quicksort"* Journal of Experimental Algorithms ACM, Vol. 12, Article 3.2, 2008.

[8] N. Wirth, *Algorithms and Data Structures*, © N. Wirth 1985 (Oberon version: August 2004)

[9] Intel 64 and IA-32 architectures optimization reference manual. http://www.intel.com/products/processor/manuals.

[10] W. A. Martin, *ACM Comp Surv*., 3(4):147-174, 1971.

[11] P. Tsigas and Y. Zhang. "*A Simple Fast Parallel Implementation of Quicksort and its Performance Evaluation* on SUN Enterprize 10000, "pdp, 00:372, 2003.

[12] H. Inoue, T. Moriyama, H. Komatasu and T. Nakatani, *"AA-Sort: A New Parallel Sorting Algorithm for Multi-Core SIMD Processors"* in PACT 07 pp. 189-198, 2007.

[13] J. Chhugani, W. Macy, A. Baransi, A.D. Nguyen, M. Hagog, S. Kumar, V. W. Lee, Y. K. Chen, P. Dubey, "*Efficient Implementation of Sorting on Multi-Core SIMD CPU Architecture*" Journal Proceedings of the VLDB Endowment, Volume 1, Issue 2, August 2008.

[14] M. Edahiro, "*Parallelizing fundamental algorithms such as sorting on multi-core processors for EDA acceleration"* Design Automation Conference, 2009.

[15] V. Ramchandran, "*CS398T presentation (PDF file)*" on http://www.cs.utexas.edu/~vlr/.