

## Multi-Core Program Optimization: Parallel Sorting Algorithms in Intel Cilk Plus

Sabahat Saleem<sup>1</sup>, M. IkramUllah Lali<sup>1</sup>, M. Saqib Nawaz<sup>1\*</sup> and Abou Bakar Nauman<sup>2</sup>

<sup>1</sup>*Department of Computer Science & IT, University of Sargodha, Sargodha, Pakistan*

<sup>2</sup>*Department of CS & IT, Sarhad University of Science & IT, Peshawar, Pakistan*

\* [saqib\\_dola@yahoo.com](mailto:saqib_dola@yahoo.com)

### Abstract

*New performance leaps has been achieved with multiprogramming and multi-core systems. Present parallel programming techniques and environment needs significant changes in programs to accomplish parallelism and also constitute complex, confusing and error-prone constructs and rules. Intel Cilk Plus is a C based computing system that presents a straight forward and well-structured model for the development, verification and analysis of multi-core and parallel programming. In this article, two programs are developed using Intel Cilk Plus. Two sequential sorting programs in C/C++ language are converted to multi-core programs in Intel Cilk Plus framework to achieve parallelism and better performance. Converted program in Cilk Plus is then checked for various conditions using tools of Cilk and after that, comparison of performance and speedup achieved over the single-core sequential program is discussed and reported.*

**Keywords:** Single-Core, Multi-Core, Cilk, Sorting Algorithms, Quick Sort, Merge Sort, Parallelism, Speedup

### 1. Introduction

With current technology, improving the number of transistors does not increase the computing capability of a system. One of the new techniques used to increase the performance of a system is the development of multi-core processors. In multi-core processors, two or more processors are used in order to increase efficiency and enhance performance [7]. Multi-core processing enhances user experience in many ways: improving the performance of activities that are bandwidth-intensive and compute, boosting the number of PC tasks that can be performed simultaneously and increasing users (workers) total number that can utilize the same server or computer and the architectures flexibility will scale out to meet new usage models that are bound to arise in the future as digitized data continues to proliferate [2].

Nowadays, multi-core computing is a flourishing area as computers having single-core processor are reaching the physical limits of possible speed and complexity. Various companies have already produced and are working on multi-core products include Intel, ARM, AMD, VIA and Broadcom [7]. Multi-core architecture is used both in personal systems and embedded systems. “Flynn's taxonomy” classifies computer architecture into four classes that are Single Instruction stream, Single Data stream (SISD), Single Instruction stream, Multiple Data stream (SIMD), Multiple Instruction stream, Single Data stream (MISD) and Multiple Instruction stream, Multiple Data stream (MIMD). Out of these four classes, only MIMD machine can execute multiple instructions simultaneously in

combination of working on a separate and independent data stream [23]. That's why MIMD model is the most suitable for parallel computing.

New performance leaps will be achieved as the industries are continuously exploiting various new parallel programming techniques [10]. In parallel computation more than one calculation are executed simultaneously [5]. Parallel computing is moving into the standard with a fast increase in the multi-core processors adoption. Parallel programming enables to fully use all the cores of the system. Shifting towards multi-core processors will have huge impact on nowadays and future software [26]. Intel Cilk Plus language is designed for the writing of multi-core programs. Cilk offers a model that is well-structured and simple for program development, analysis and verification of written program [10]. Any program that works on an Intel single-core processor will work with an Intel multi-core processor. Moreover, Intel Cilk Plus is considered the fastest and easiest way to tackle multi-core and vector processing power [8].

A program performance running on a single-core can be enhanced over multi-core by splitting the whole program into different threads which can concurrently run on multiple processors. Due to the increasing popularity of multi-core technologies, many attempts of parallelism have been reported in literature. Apart from existing techniques, there is still enough room for developing efficient technique for better parallelism. In this article, multi-core program optimization is done where two single-core sorting algorithms are converted to parallel programming code using Intel Cilk Plus technology. Single-core sorting (quick sort and merge sort) programs are converted to Intel Cilk Plus programs to achieve efficient parallelism.

## 2. Intel Cilk Plus Technology

Intel Cilk Plus language, developed at Cilk Arts is built on the Cilk technology at M.I.T. over the past two decades. With Cilk Plus, C language is extended for writing of parallel applications that efficiently make use of multiple processors. The Cilk Plus language is well suited for divide and conquer algorithms. Divide and conquer algorithms are tackled mostly with recursive technique and Cilk support these techniques [4]. Intel Cilk Plus is a part of the C compiler; therefore no extra steps are required while working with any code of C or C++ in Cilk Plus [8].

Intel Cilk Plus maps an arbitrary number of tasks to a limited set of workers and parallelism is achieved without supervising call chains and also without oversubscribing resources. In Intel Cilk Plus, parallelism is achieved by exploiting the number of workers available in the pool instead of usage of unbound resource during scheduled tasking [4]. For parallel programming in Cilk Plus, three keywords offer a powerful model.

### 2.1. Cilk Plus Keywords

In order to achieve task parallelism, Cilk Plus offers three powerful expressions, which are: *cilk\_for*, *cilk\_spawn* and *cilk\_sync*. In Cilk, *cilk\_for* is the replacement for C++ for loop, where iterations in loop are allowed to run in parallel [4]. In *cilk\_for*, body of loops executes in parallel, so in order to avoid race conditions, it must not modify control variable as well as any non-local variable. *Cilk\_spawn* keyword is used to start parallel computations. It informs the runtime system that spawned function can run in parallel. *Cilk\_sync* keyword is used to wait for the spawned procedures to complete. *Cilk\_sync* function cannot run in parallel with spawned functions.

A race condition occurs when two or more process/threads access shared data at same time and often creates bugs in parallel programming. When a memory location is accessed by two

or more strands at the same time then determinacy race occurs. Data race is considered as a special kind of determinacy race [4]. Various techniques like semaphores, locks, mutex are used, which allow only one process or thread to access shared data at a time. If one process is using shared data and other process wants access then second process will have to wait. Cilk offers race detector tool *cilkscreen* that monitors the execution of the Cilk program.

The *cilkscreen* tool detects all data races that are encountered during execution. It reports only those races that occur due to Cilk keywords. For efficient parallel program, all race condition detected by *cilkscreen* tool should be resolved first. *Cilkview* scalability and performance analyzer tool reports statistics about a parallel Cilk program. It predicts how the performance of program in Cilk scales on multi-cores. It also presents a graphical view of performance and scalability. Analyzer measure and evaluate the expected speedup on 2, 4, 8, 16, and 32 multi-processors [4].

### 3. Sorting Algorithms

In most frequent operations that computer performs, sorting operation is the one in it. Sorting algorithms are kind of algorithms that put elements in certain order. Methods for sorting are generally based on the comparison between elements and record moves. Widely used sorting algorithms are: bubble sort, quick sort, simple sort, bucket sort, merge sort and heap sort. All sorting algorithms are different from one another on the basis of their performance and efficiency. Sorting algorithms allow us to put information in some meaningful order and are used in many applications. Database systems make huge use of sorting operations [27]. They are also used in sparse matrix multiplication, image processing, computer graphics, statistical methodology and Map Reduce [22, 23]. Due to their importance, efficient sorting algorithms are needed for various parallel architectures. Along with advantages, sorting methods have disadvantages too. Main disadvantage is that the best sorting algorithms have performance of  $O(n \log n)$ . It means that large data will take long time to sort. Best, average and worst time complexity of above mentioned sorting algorithms is given in table 1.

**Table 1. Time Complexity of Sorting Algorithms**

Sorting algorithms	Best Case	Average Case	Worst case	Space
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	1
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$	1
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	1
Bucket sort	$O(n)$	$O(n)$	$O(n)$	1
Quick sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$\log n$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$n$
Heap sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	1

Insertion sort, bubble sort and selection sort are the simple sorting algorithms with average running time of  $O(n^2)$ . In three of them, insertion sort is the most simple one and insertion sort works well when the records sequence is in order or when the value of  $n$  small [24]. In all of the sorting methods given in the table, quick sort is usually the faster one and has the best average performance of  $O(n \log n)$ . But in worst case when the record is in order, the quick sort has the worst case running time of  $O(n^2)$ . Both in average case and in worst case, heap sort running time is  $O(n \log n)$ . In average case, quick sort is little faster than heap sort. In worst case, heap sort has the advantage over quick sort algorithm and it is the biggest edge of heap sort over the quick sort, but heap sort is not a stable sort as heap operations may change the order of equal items. In merge sort, two or more than two lists or tables are merged in

order to make two or more new ordered lists or tables. On comparison of merge sort with quick sort and heap sort, merge sort is a stable sort. Main drawback of the merge sort is that it requires more memory as compared to quick sort. Most implementation of merge sort has to be done in  $O(2n)$  space, whereas quick sort implementation can be done in place.

### 3.1. Related Work

In the past, various sorting algorithms have been studied, but only few algorithms have the capability to effectively make use of Single Instruction Multiple Data (SIMD) instructions and thread-level parallelism. Inoue *et al.*, [3] in their article, proposed a parallel sorting algorithm for memory-sharing multi-cores, known as Aligned-Access sort (AA-sort). This algorithm exploits the Single Instruction Multiple Data (SIMD) instructions. Main factor in achieving high performance through this algorithm is removing unaligned memory accesses that may reduce SIMD instructions effectiveness. Rashidy *et al.*, [6] developed a parallel bubble sort algorithm using streaming model approach. In stream program, the input data to the program comes in stream form. Writing program in stream model is called stream processing. In stream processing with a given set of data (stream), various kinds of operations can be applied to every element in the stream.

In [6], they have written the code for parallel bubble sort in Java language and used Java library *Jstream*. Stream programs can be mapped easily to the architecture of multi-core or distributed systems [14]. All stream programs exhibit various features like these programs can work with large data streams and has stable pattern of computations. One of the basic properties of a stream program is that it can work with massive data easily. Logically, data are finite in size and usually data comes from external source. Each data item is executed for short time period and after processing, it is sent to the output [14].

In order to speed up sorting process, multiprocessors are used for parallel sorting. Different parallel sorting algorithm such as bi-tonic sort [15, 16], column sort [20], parallel merge sort [12], parallel radix sort [21] and randomized parallel sorting [18] has been developed. In [9] a parallel sorting algorithm using Graphics Processing Units (GPU's) has been devised. Man *et al.*, [11] developed an efficient parallel algorithm *psort* in C language that is compatible with standard *qsort*. They have implemented *psort* on a server having two processors of Intel Quad-Core. Any program that uses *qsort* from standard C library can be accelerated by simply changing *qsort* call with *psort*. For local sequential sorting, *psort* make use of standard *qsort* as sub-routine. So if the *qsort* performance is improved, *psort* performance will automatically be improved. In [25], they have presented a parallelized algorithm called Map Sort to accelerate Electronic Design Automation (EDA) software's performance on multi-core architecture.

In [19], various quick sort algorithms are parallelized using OpenMP 3.0 method and the embedded environment of OMAP-4430. They study and inspect the effect of parallelization in these algorithms with the help of profiler *GNU Gprof*. Qian and Xu [24] analyze the parallelism of sequential algorithms of sorting that are based on multi-core systems. Nadathur *et al.*, [1] designed high-performance parallel routines of merge and radix sort for many-core GPU's in CUDA.

Bader *et al.*, [13] in their article identified some main issues in the design of algorithm for multi-core processors and they proposed a framework for multi-core processors and this framework is called Software and Algorithms for Running on Multi-

core (SWARM). SWARM is an open-source parallel library consisting of basic primitives that make use of multi-core processors. SWARM framework offers basic functionality for multithreaded programming like memory management, synchronization and collective operations. SWARM is built on POSIX threads, in which user has the choice to either use the developed primitives or direct thread primitives.

#### 4. Parallel Quick Sort in Cilk Plus

Among all sorting algorithms, quick sort is considered the most popular and vastly used sorting algorithms. Quick sort algorithm developed by Tony Hoary, is a divide and conquer algorithm. In quick sort algorithm, array is partitioned into two by a pivot element [17]. Those elements that are smaller compared to pivot are moved before it and all elements that are greater than pivot are moved after it. Moving of the elements before or after pivot can be done in linear time [17].

In this section, sequential quick sort algorithm is converted to parallel quick sort in Cilk Plus. We have used Visual C++ express edition 2008 with Cilk technology. In parallel quick sort, recursive calls are spawned with *cilk\_spawn* keywords so they can run in parallel. The *cilk\_sync* keyword indicates that the recursive function cannot be allowed to continue its operation before all the requests of *cilk\_spawn* in the same recursive function have completed. *Cilkscreen* tool is used to detect any data races created by Cilk keywords during execution and *cilkview* is used for performance and speed up comparisons. Parallel quick sort algorithm in Cilk is given below:

```
// Cilk parallel program for Quick Sort Algorithm....

#include <cilk.h>
#include <cilkview.h>
#include <algorithm>
#include <functional>
using namespace std;

void quick_sort(int * beg, int * end)
{
    if (beg < end)
    { end = end--;

int * pivot = partition(beg, end, bind2nd(less<int>(), *end));
    swap(*end, *pivot);

    cilk_spawn quick_sort(begin, pivot);
        pivot++;
        end++;
    quick_sort(pivot, end);
    cilk_sync; }}

int cilk_main()
{
    int n;
    cout<<"Enter total number of elements that are to be sorted:";
    cin>>n;
```

```
int* a = new int[n];

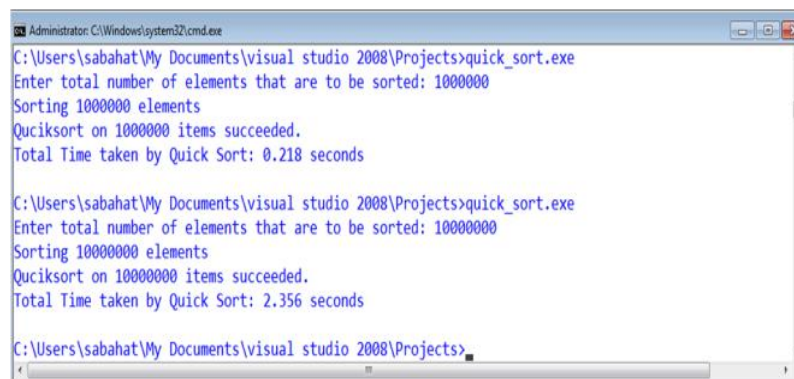
cilk_for (int x = 0; x< n; x++)
{
    a[x] = x; }
random_shuffle(a, a + n);
cout << "Sorting"<< n << "elements" << endl;

cilk::cilkview cilkview;
cilkview.start();
    quick_sort(a, a + n);
cilkview.stop();

cilkview.dump("Qucik_Sort_Results", false);
cout << "Quciksort on" << n <<"items succeeded." << endl;
cout << "Total Time taken by Quick Sort:"
    << cilkview.accumulated_milliseconds()/1000.f <<"Seconds"
    << endl;
delete[]a;
return 0;}
```

Every program starts as a single-threaded program. To begin parallel activities, the context of Cilk runtime must be created and initialized and begin execution at a precise entry point in the program. This can be achieved in two ways. One approach is to replace the main with *cilk\_main*, while the second approach is to first create an explicit context and use *cilk::run* to enter in this context. In above code, we have replaced the main with *cilk\_main*. In the code, *cilk::cilkview* object is created. Methods of *start()*, *stop()* and *dump()* are declared in the code that will produce the performance measurements. Above program will be run *N* times. *Cilkview* will run the program one extra time with the option of parallel performance analyzer to predict the scaling of the performance.

Above program is run with visual studio 2008 command prompt. In order to build and compile a Cilk program on a Windows system, *cilkpp* command is used. After successful compilation, *cilkpp* command makes an executable file of the program with the same name. Figure 1 shows how to run the program.



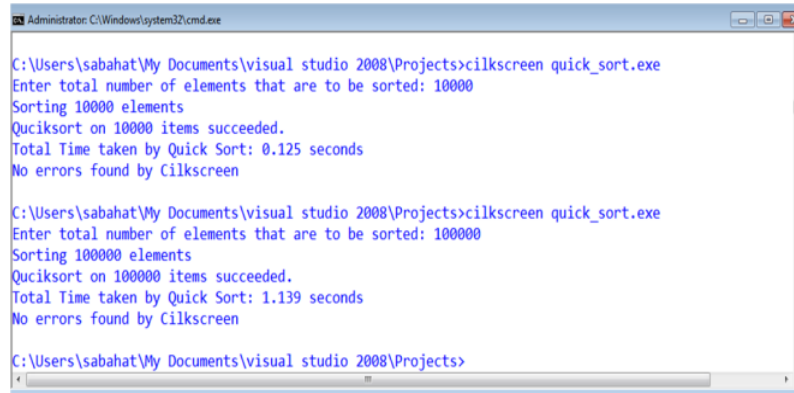
```
Administrator: C:\Windows\system32\cmd.exe
C:\Users\sabahat\My Documents\visual studio 2008\Projects>quick_sort.exe
Enter total number of elements that are to be sorted: 1000000
Sorting 1000000 elements
Quciksort on 1000000 items succeeded.
Total Time taken by Quick Sort: 0.218 seconds

C:\Users\sabahat\My Documents\visual studio 2008\Projects>quick_sort.exe
Enter total number of elements that are to be sorted: 10000000
Sorting 10000000 elements
Quciksort on 10000000 items succeeded.
Total Time taken by Quick Sort: 2.356 seconds

C:\Users\sabahat\My Documents\visual studio 2008\Projects>
```

**Figure 1. Running the Parallel Quick Sort Program**

As discussed in Section 2, the *cilkscreen* tool checks the program operation while program is running with some test input. *Cilkscreen* detects all data races that are detected during execution of the program. For a reliable parallel program, all races that *cilkscreen* reports should be review and resolved [4]. *Cilkscreen* run the program on single worker and all the reads and writes to memory are monitored by *cilkscreen*. After program finishes, *cilkscreen* shows all the conflicts information that are found during write/read and any write/write. If any possible program schedule output result that is different from the serial program execution then a race condition has occurred. Figure 2 show that there are no data races in our parallel quick sort program.



```
Administrator: C:\Windows\system32\cmd.exe

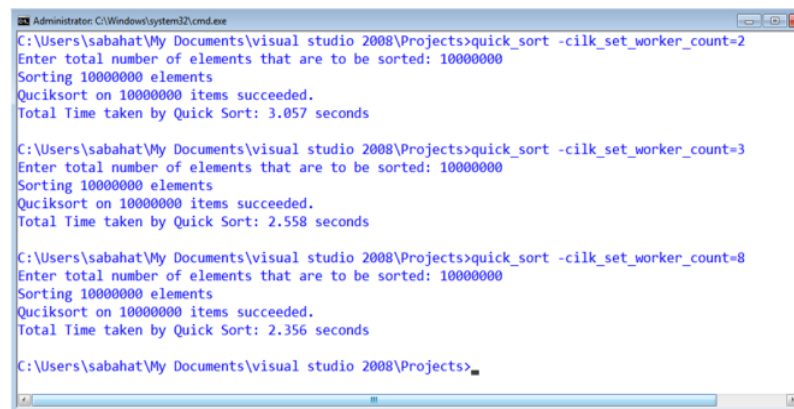
C:\Users\sabahat\My Documents\visual studio 2008\Projects>cilkscreen quick_sort.exe
Enter total number of elements that are to be sorted: 10000
Sorting 10000 elements
Quicksort on 10000 items succeeded.
Total Time taken by Quick Sort: 0.125 seconds
No errors found by Cilkscreen

C:\Users\sabahat\My Documents\visual studio 2008\Projects>cilkscreen quick_sort.exe
Enter total number of elements that are to be sorted: 100000
Sorting 100000 elements
Quicksort on 100000 items succeeded.
Total Time taken by Quick Sort: 1.139 seconds
No errors found by Cilkscreen

C:\Users\sabahat\My Documents\visual studio 2008\Projects>
```

**Figure 2. Cilkscreen Race Detection of Parallel Quick Sort**

By default, the number of worker threads is equal to the number of cores on the system on which Cilk program is running. We can increase or decrease the number of workers from command line using *cilk\_set\_worker\_count* command. Figure 3 shows the execution and time taken by the quick sort with different counts values of the worker to sort 1000000 elements.



```
Administrator: C:\Windows\system32\cmd.exe

C:\Users\sabahat\My Documents\visual studio 2008\Projects>quick_sort -cilk_set_worker_count=2
Enter total number of elements that are to be sorted: 10000000
Sorting 10000000 elements
Quicksort on 10000000 items succeeded.
Total Time taken by Quick Sort: 3.057 seconds

C:\Users\sabahat\My Documents\visual studio 2008\Projects>quick_sort -cilk_set_worker_count=3
Enter total number of elements that are to be sorted: 10000000
Sorting 10000000 elements
Quicksort on 10000000 items succeeded.
Total Time taken by Quick Sort: 2.558 seconds

C:\Users\sabahat\My Documents\visual studio 2008\Projects>quick_sort -cilk_set_worker_count=8
Enter total number of elements that are to be sorted: 10000000
Sorting 10000000 elements
Quicksort on 10000000 items succeeded.
Total Time taken by Quick Sort: 2.356 seconds

C:\Users\sabahat\My Documents\visual studio 2008\Projects>
```

**Figure 3. Parallel Quick Sort Program Execution with different Workers Count**

*Cilkview* analyzer helps in understanding the parallel performance of a program. Result of *cilkview* appears on Visual Studio command prompt window after program finish execution and *cilkview* also display a graph that shows the predicted speedup and parallelism. Report of *cilkview* is partitioned in two sections. One is called the Parallelism Profile and the other is called the Speedup Estimate [4]. Parallelism profile is shown in Figure 4. The Parallelism profile is shown on command prompt demonstrates all the statistics of program execution in Cilk. To get whole

picture of the program parallel operation, *cilkview* scalability and performance analyzer run the program on a single worker and tracks all the spawns. Statistics about work, span, burdened spawn, parallelism, burdened span etc. are shown in Parallelism profile.

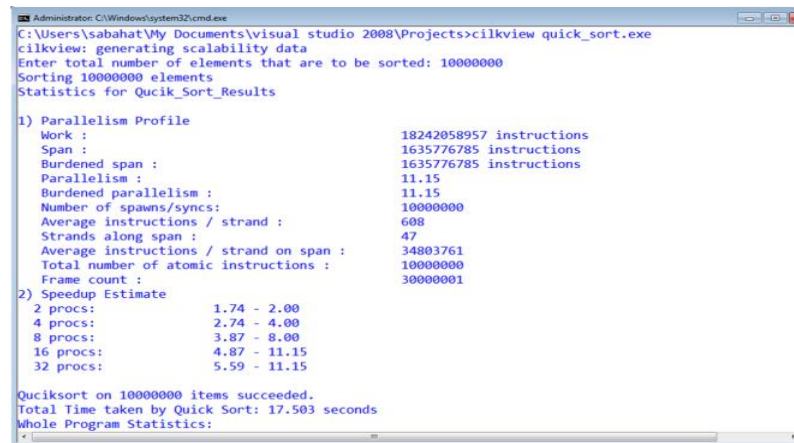


Figure 4. Parallelism Profile of Parallel Quick Sort

*Cilkview* also measures the anticipated speedup on number of processors such as on 2, 4, 8, 16 and 32 processors. In the graph, estimates of speedup are displayed with ranges of lower bounds and upper bounds. Upper bound is the smaller number of workers and the program parallelism and the lower bound in the graph stands for estimated overhead. Total overhead depends on various factors, which include the program parallel structure and the total number of workers. Lower bound that is less than 1 shows that instead of speed up, the program may slow down when it is run on more than one processor [4]. Figure 5 shows the estimated speed up of program in the form of a graph.

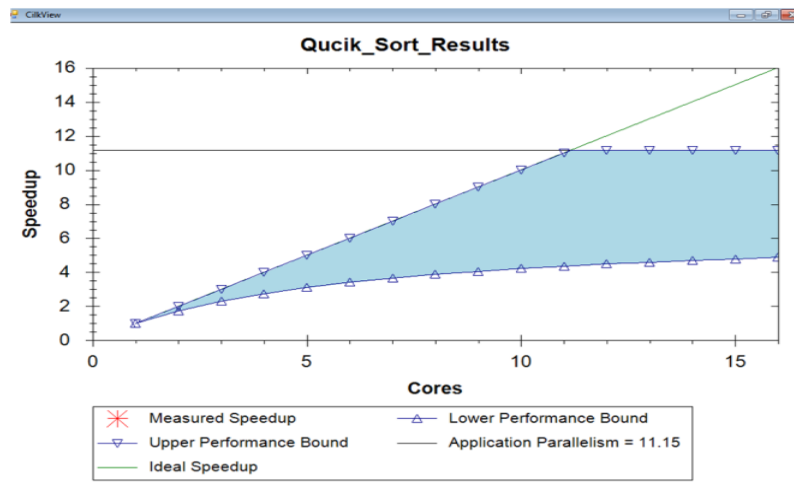


Figure 5. Speedup Estimate of Parallel Quick Sort

## 5. Parallel Merge Sort in Cilk Plus

In computer sorting, John von Neumann first proposed the method for merge sort. Merge Sort algorithm like quick sort algorithm is based on the method of divide and conquer.



Unsorted array of elements or unsorted list of elements is break into small sub-arrays. The elements of unsorted sub-arrays are sorted and then after sorting they are merge together into a sorted list or array. Merge sort algorithm is linear in time with time complexity of  $O(n)$  [17]. We have converted a sequential merge sort program written in C to a Cilk program. Converted program is shown below.

```
// Parallel Merge Sort Algorithm....

void merge(int arr[], int pp, int qq, int rr);
void mergesort(int arr[], int pp,int rr)
{
    int qq;
        if (pr<rr){
            qq=((pp+rr)/2);

cilk_spawn mergesort(arr,pp,qq);
    mergesort(arr,qq+1,rr);
    cilk_sync;
    merge(arr,pp,qq,rr); } }
    int b[10000000];

void merge(int arr[], int pp, int qq, int rr)
{
    int i,j,k;
    k=0, i=p, j=q+1;
    while(i<=q && j<=r)
    {
        if (arr[i]<arr[j])
            b[k++]= arr[i++];
        else b[k++]= arr[j++]; }
    while(i<=q)
        b[k++]= a[i++];
    while(j<=rr)
        b[k++]= arr[j++];
        for(i=rr;i>=pp;i--){
            arr[i]=b[--k]; }}

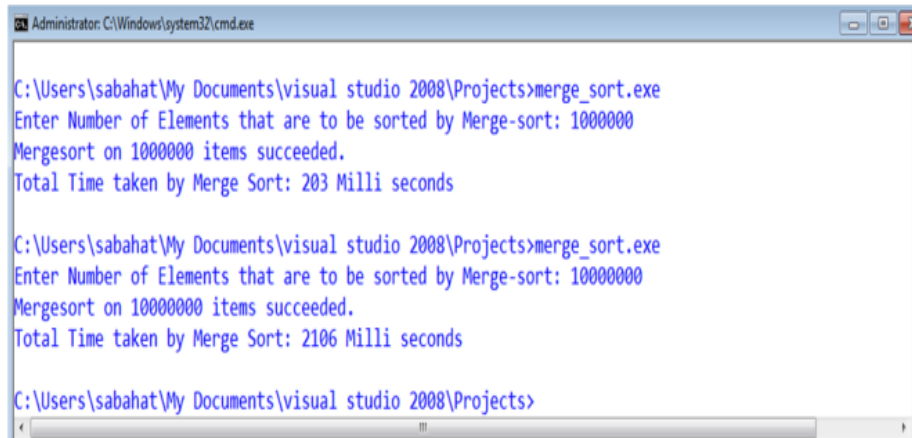
int cilk_main(){
    int size;
    cout<<"Enter Number of Elements that are to be sorted by
Merge-sort:";
    cin>>size;
    int a[10000000];
    cilk_for (int x = 0; x < size; x++){
        a[x] = x; }
    random_shuffle(a, a + size);

    cilk::cilkview cilkview;
    cilkview.start();
        mergesort(a,0,size);
```

```
cilkview.stop();

cilkview.dump("Merge_Sort_Results", false);
cout << "Mergesort on " << size << " items succeeded." <<
endl;
cout << "Total Time taken by Merge Sort:"
    <<cilkview.accumulated_milliseconds()
    << "Milli seconds"<< endl;
return 0;}
```

For the compilation of above merge sort program, same procedure of quick-sort is carried out. *Cilkpp* command is used to compile the parallel merge sort program written in Cilk. In Figure 6, parallel merge-sort program is run on 1000000 and 10000000 random elements. Total time taken by program for merging specified number of elements is also shown in Figure 6. The Figure 6 shows that the parallel merge sort program sorted 1000000 random elements in 203 milliseconds and it took 2106 milliseconds to sort 10000000 unsorted elements. Execution time for parallel merge sort is shown in milliseconds, while execution time for parallel quick sort is calculated in seconds. By default, *cilkview* tool measure the total time in milliseconds.



```
Administrator: C:\Windows\system32\cmd.exe

C:\Users\sabahat\My Documents\visual studio 2008\Projects>merge_sort.exe
Enter Number of Elements that are to be sorted by Merge-sort: 1000000
Mergesort on 1000000 items succeeded.
Total Time taken by Merge Sort: 203 Milli seconds

C:\Users\sabahat\My Documents\visual studio 2008\Projects>merge_sort.exe
Enter Number of Elements that are to be sorted by Merge-sort: 10000000
Mergesort on 10000000 items succeeded.
Total Time taken by Merge Sort: 2106 Milli seconds

C:\Users\sabahat\My Documents\visual studio 2008\Projects>
```

**Figure 6. Running the Parallel Merge Sort Program**

As discussed above, in Cilk, *cilkview* analyzer helps in understanding the parallel performance of a program. Parallelism profile for parallel merge sort is shown in Figure 7 while speedup estimate (graph) is shown in figure 8. Both parallelism profile and speedup estimate were discussed in Section 2.1 and 4. For performance estimation, if grain size is not set then *cilkview* monitors the *cilk\_for* loops with assumed granularity of 1. *Cilkview* also monitors the single execution of the program with a predefined input data. Generally, the performance of the program will change with different input data. We have left out the data race detection in parallel merge sort as the procedure for it is same to the procedure described in Section 4 for parallel quick sort algorithm.

```

Administrator: C:\Windows\system32\cmd.exe

C:\Users\sabahat\My Documents\visual studio 2008\Projects>cilkview merge_sort.exe
cilkview: generating scalability data
Enter Number of Elements that are to be sorted by Merge-sort: 10000000
Statistics for Merge_Sort_Results

1) Parallelism Profile
Work : 14785256603 instructions
Span : 810010582 instructions
Burdened span : 810010582 instructions
Parallelism : 18.25
Burdened parallelism : 18.25
Number of spawns/syncs: 10000000
Average instructions / strand : 492
Strands along span : 24
Average instructions / strand on span : 33750440
Total number of atomic instructions : 10000000
Frame count : 30000001

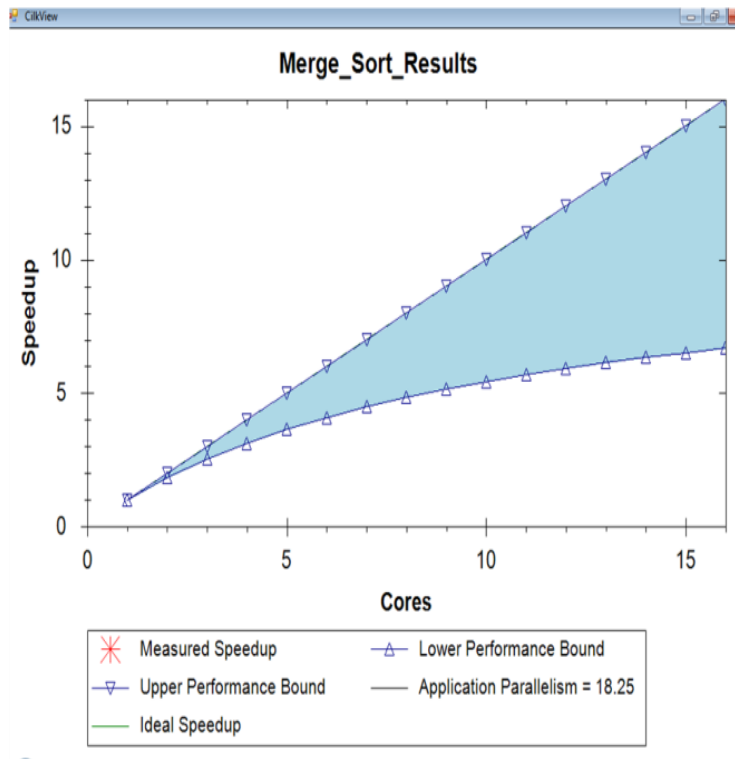
2) Speedup Estimate
2 procs: 1.83 - 2.00
4 procs: 3.13 - 4.00
8 procs: 4.84 - 8.00
16 procs: 6.67 - 16.00
32 procs: 8.23 - 18.25

Mergesort on 10000000 items succeeded.
Total Time taken by Merge Sort: 14306 Milli seconds
Whole Program Statistics:

Cilkview Scalability Analyzer V1.1.0, Build 8504

```

**Figure 7. Parallelism Profile for Parallel Merge Sort**



**Figure 8. Speedup Projection of Parallel Merger Sort**

Figure 8 shows the estimated speedup of program in the form of a graph. As it is cleared from Figure 8, speedup increases with increase in the total number of available cores.

*Cilkview* performance and scalability analyzer help in understanding the parallel performance of the program written in Cilk. It shows the parallel statistics of the program, prediction of the program performance over multiple processors; benchmark the program on one or more processors and it also presents a graphical view of speedup and scalability.

## 6. Conclusion

Technology of multi-core has and is changing the basic concepts in the field of software research, specifically those applications that runs on personal computers and on servers. Programs and applications that are developed with ideas of parallel programming not only improved the performance dramatically, it is considered as key factor for the evolution and popularity of multi-core. Cilk is an algorithmic multi-threaded language. Objective of the Intel Cilk is to achieve parallelism, enhance performance and speedup. In Cilk, programmers can develop their applications either by writing program in the semantics of call/return and then specifying which call can be executed in parallel or by writing the program in multithreaded environment.

As Intel Cilk Plus is an extension to C/C++ language, original programs does not require significant restructuring and modification in order to add parallelism in programs. Quick sort and merge sort algorithms are divide and conquer algorithm and can easily be converted to Cilk code as Cilk technology is suitable for this kind of algorithms. In this article, popular sequential quick sort algorithm is converted to parallel quick sort algorithm in Intel Cilk Plus. Performance and speedup is achieved over sequential quick sort by converting sequential program into parallel Cilk program. After quick sort, sequential merge sort algorithm in C is converted to Cilk program. Speedup and performance achieved by conversion of sequential quick sort and merge sort algorithm to Intel Cilk program is discussed. Parallel quick sort and parallel merge sort algorithms that we developed in Cilk framework has been evaluated on Intel core i3 system with Microsoft Windows 7 Operating System and results are shown in figures.

## References

- [1] S. Nadathur, M. Harris and M. Garland, "Designing Efficient Sorting Algorithms for many Core GPU's", IEEE International Symposium on Parallel & Distributed Processing, (2009), pp. 23-29.
- [2] G. Koch, "Multi-Core Introduction", Intel Developer Zone, <http://software.intel.com/enus/articles/multi-core-introduction>, (2013) March 5.
- [3] H. Inoue, T. Moriyama, H. Komatsu and T. Nakatani, "AA-Sort: A New Parallel Sorting Algorithm for Multi-Core SIMD Processors", IEEE 16<sup>th</sup> International Conference on Parallel Architectures and Compilation Techniques, (2007), pp. 189-198.
- [4] Intel Cilk++ SDK Programmer's Guide, Document Number 322581-001 US. Intel Corporation, (2009).
- [5] G. S. Almasi and A. Gottlieb, "Highly Parallel Computing", Benjamin Cummings Publishers, CA, USA, (1989).
- [6] R. Rashidy, S. Yousefpour and M. Koohi. "Parallel Bubble Sort Using Stream Programming Paradigm", 5<sup>th</sup> International Conference on Application of Information and Communication Technologies (AICT), (2011), pp. 1-5.
- [7] I. Fasiku, "Performance Evaluation of Multi-Core Processors, M. Tech Thesis", Federal University of Technology, Akure, Nigeria, (2012).
- [8] A. D. Robinson, "Cilk Plus: Language Support for Thread and Vector Parallelism", (2012).

- [9] E. Sintroon and U. Assarsson, "Fast Parallel GPU Sorting using a Hybrid Algorithm", *Journal of Parallel and Distributed Computing*, vol. 66, (2008), pp. 1381-1388.
- [10] Intel Cilk Plus, Intel Developer Zone, <http://software.intel.com/en-us/intel-cilk-plus>, (2013) October 4.
- [11] D. Man, Y. Ito and K. Nakano, "An Efficient Parallel Sorting Compatible with Standard qsort", *International Conference on Parallel and Distributed Computing, Applications and Technologies*, (2009), pp. 512-517.
- [12] M. Joen and D. Kim, "Parallel Merge Sort with Load Balancing". *International Journal of Parallel Programming*, vol. 31, (2003), pp. 21-33.
- [13] D. A. Bader, V. Kanade and K. Madduri, "SWARM: A Parallel Programming Framework for Multi-Core Processors", *First Workshop on Multithreaded Architectures and Applications (MTAAP)*, (2007), pp. 21-33.
- [14] J. H. Spring, J. Privat, R. Guerraoui and J. Vitek, "StreamFlex: High-Throughput Stream Programming in Java", *ACM SIGPLAN Notices*, vol. 42, no. 10, (2007), pp. 211-228.
- [15] K. Batcher, "Sorting Networks and their Applications", *Proceedings of the AFIPS Spring Joint Computer Conference*, (1968), pp. 307-314.
- [16] M. F. Ionsecu and K. E. Schauser, "Optimizing Parallel Bi-tonic Sort", In *Proceedings of the 11<sup>th</sup> International Symposium on Parallel Processing*, (1997), pp. 303-309.
- [17] Leiserson, E. Charles, R. L. Rivest and C. Stein, "Introduction to Algorithms", Ed. Thomas H. Cormen, The MIT press, (2001).
- [18] D. R. Helman, D. D. Bader and J. Jaja. "A Randomized Parallel Sort Algorithm with an Experimental Study", *Journal of Parallel and Distributed Computing*, vol. 53, (1998), pp. 1-23.
- [19] K. J. Kim, S. J. Cho and J. -W. Jeon, "Parallel Quick Sort Algorithms Analysis using OpenMP 3.0 in Embedded System", *11<sup>th</sup> International Conference on Control, Automation and Systems*, (2011), pp. 757-761.
- [20] A. C. Dusseau, D. E. Kuller, K. E. Schauser and R. P. Martin, "Fast Parallel Sorting under Log P: Experience with the CM-5", *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, (1996), pp. 791-805.
- [21] A. Sohnans and Y. Kodama, "Load Balanced Parallel Radix Sort", In *Proceedings of the 12<sup>th</sup> International conference on Supercomputing*, (1998), pp. 305-312.
- [22] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", *Communications of the ACM*, vol. 51, no. 1, (2008), pp. 107-113.
- [23] B. He, W. Fang, N. K. Govindaraju, Q. Luo and T. Wang. "Mars, A Map Reduce Framework on Graphics Processors", In *Proceedings of the 17<sup>th</sup> Intl. Conference on Parallel Architectures and Compilation Techniques*, (2008), pp. 260-269.
- [24] X. -j. Qian, J. -b. Xu, "Optimization and Implementation of Sorting Algorithm based on Multi-Core and Multi-Thread", *IEEE 3<sup>rd</sup> International Conference on Communication Software and Networks*, (2011), pp. 29-32.
- [25] E. Masato, "Parallelizing Fundamental Algorithms such as Sorting on Multi-core Processors for EDA Acceleration", *Proceedings of the 2009 Asia and South Pacific Design Automation Conference*, (2009), pp. 230-233.
- [26] J. Shirako, H. Kasahara and V. Sarkar, "Language Extensions in Support of Compiler Parallelization", *Languages and Compilers for Parallel Computing*, Springer, vol. 5234, (2008), pp. 78-94.
- [27] G. Graefe, "Implementing Sorting in Database Systems", *ACM Computer Surveys*, vol. 38, no. 3, (2006), pp. 1-37.

## Authors



### Sabahat Saleem

She received her B.S. degree in Computer Science from University of Sargodha, Pakistan. She is currently pursuing her M.S. in Computer Science from University of Sargodha, Pakistan. Her research interest includes Parallel Algorithms.



**M. IkramUllah Lali**

He is working as Assistant Professor in the department of Computer Science & IT, University of Sargodha, Pakistan. He received his M.Sc. & Ph.D. degrees from COMSATS Institute of Information Technology, Islamabad, Pakistan. His research interests include Software Engineering, Formal Methods, Parallel and Distributed Computing. He is an author of several research articles in the different Journals of international repute.



**M. Saqib Nawaz**

He received his B.S. degree in Computer Systems Engineering from University of Engineering and Technology, Peshawar, Pakistan. He is currently pursuing his M.S. in Computer Science from University of Sargodha, Pakistan. His research interests include Formal Methods and Parallel Computing.



**Abou Bakar Nauman**

He received his M.Sc. degree in Computer Science from Bahauddin Zakariya University, Multan, Pakistan. He is pursuing his Ph.D. from COMSATS Institute of Information Technology, Islamabad, Pakistan. He is currently working as Assistant Professor in Department of CS & IT, Sarhad University of Science & IT, Peshawar, Pakistan. His research interests include Software Engineering (Requirement Engineering and Software Testing), Parallel and Distributed Computing.