# Parallelizing the Merge Sorting Network Algorithm on a Multi-Core Computer Using Go and Cilk

Peiyi Tang
Department of Computer Science
University of Arkansas at Little Rock
Little Rock, AR
pxtang@ualr.edu

Doug Serfass
Department of Computer Science
University of Arkansas at Little Rock
Little Rock, AR
djserfass@ualr.edu

## ABSTRACT
We create a scalable parallel algorithm based on parallelizing the merge sorting network algorithm. We implemented our scalable parallel algorithm using two modern shared-memory parallel programming languages, Go [1] and Cilk-5 [2]. We also compare Go and Cilk speedup and performance. Our experimental results show that our parallel algorithm is scalable and provides near linear speedup for all large problem sizes. We compare Go and Cilk scheduling and synchronization efficiency as well as the overall performance of our parallel code.

## Categories and Subject Descriptors
C.1.4 [**Parallel Architecture**]. D.1.3 [**Concurrent Programming**]: Parallel programming. D.3.2 [**Language Classification**]: Concurrent, Distributed and Parallel Languages. F.2.2 [**Non-Numerical Algorithms and Problems**]: Sorting and Searching

## General Terms
Algorithms, Performance, Design, Languages, Theory.

## Keywords
Sorting Network, Multi-Core Multiprocessor, Language Go, Language Cilk, Scalable Parallel Algorithm, Speedup and Performance.

## 1. INTRODUCTION
Sorting is one of the most important problems in computer science. Sorting networks have been studied since the mid 1950s [3] and are considered to be some of the most efficient sorting algorithms. The sequential merge sorting network algorithm [4] is an excellent candidate for parallelization on a parallel computer.

In 2004, parallel computers became a reality as all microprocessor manufacturers began to manufacture multi-core multiprocessor chips. Widespread availability of multi-core computers has made parallel programming both feasible and necessary. Continued use of sequential programs on multi-core computers is a waste of multiprocessor resources. There is an urgent requirement to convert sequential programs to parallel programs. Only then will multi-core computers deliver the gain in performance promised by multiprocessor hardware.

Efficient parallel algorithms and programming languages are necessary in order to optimally execute applications on multi-core computers. In this paper, we parallelize the sequential merge sorting network algorithm to create an efficient scalable parallel merge sorting network algorithm. To prove the efficiency of our scalable parallel merge sorting network algorithm, we implemented it using Go and Cilk-5. We also compare the performance of these two languages.

Our experimental results on an 8-core AMD Opteron computer show that our scalable parallel merge sorting network algorithm delivers close to linear speedup using both Go and Cilk. We found that the speedup of Cilk is greater than Go due to Cilk's efficient work-stealing scheduling and synchronization. We also found that the Go compiler generates more efficient code resulting in faster execution time. While coding, we discovered that Go is more flexible than Cilk in that Go allows a function to be called sequential or parallel. This feature of Go resulted in more compact code.

Our contribution to sorting network research is as follows: (1) We created a new scalable parallel merge sorting network algorithm. (2) We implemented the scalable parallel merge sorting network algorithm in the modern Go and Cilk multi-core parallel programming languages. (3) We conducted a performance evaluation of our implementation and generated data that confirm nearly linear speedup. (4) We compared the performance of Go and Cilk.

This paper consists of the following sections: Section 2 discusses recent related work on sorting network algorithms. Section 3 describes how we parallelized the merge sorting network algorithm. Section 4 comments on the syntax we encountered (or was missing) when implementing the scalable parallel merge sorting network algorithm in Go and Cilk. Section 5 provides a performance evaluation of our implementation. Section 6 is our conclusions and suggestions for future research.

## 2. RELATED WORK
Ye et. al [5] evaluated an efficient comparison-based sorting algorithm for CUDA-enabled GPUs. Their study concluded that although bitonic sorter has a relatively high computation complexity, it is still very efficient when sorting small sequences. The authors offer evidence that their algorithm demonstrates up to 30% better performance than previous optimized comparison-based algorithms for input sequences with millions of elements.


49th ACM Southeast Conference, March 24–26, 2011, Kennessaw, GA, USA. Copyright 2011 ACM 978-1-4503-0686-7/11/03.....$10.00.

Seiferas [6] simplifies Paterson's version of the AKS sorting network by tuning the invariant to be maintained. His study provides a clear and simple modular demonstration of the remarkable big-oh version of the AKS result.

Olariu et. al [7] proposed a simple sorting architecture whose main feature is the pipelined use of a sorting network of fixed I/O size p to sort an arbitrarily large data set of N elements. Their study concluded that the time performance of their design is virtually independent of the cost and depth of the underlying sorting network.

# 3. PARALLELIZING THE MERGE SORTING NETWORK ALGORITHM

The process of parallelizing the merge sorting network algorithm consisted of three steps: identification of the sequential merge sorting network algorithm; modifying the sequential merge sorting network algorithm to create a parallel version; and improving the performance of the parallel merge sorting network algorithm to create the scalable parallel merge sorting network algorithm.

## 3.1 Sequential Merge Sorting Network Algorithm

The sequential merge sorting network algorithm is shown in Figure 1. The problem size, n, is a power of 2.

```
int w[n];
compSwitch(i,j)
   a←w[i]; b←w[j];
   if (a≤b)  then {w[i]←a; w[j]←b}
             else  {w[i]←b; w[j]←a}
halfClean(l,h)
   s←h-l+1;
   for i←0 to s/2-1
      compSwitch(l+i,l+i+s/2);
alterHClean(l,h)
   s←h-l+1;
   for i←0 to s/2-1
      compSwitch(l+i,h-i);
biSort(l,h)
   if l=h then return;
   s←h-l+1;
   halfClean(l,h);
   biSort(l,l+s/2-1); biSort(l+s/2,h);
merge(l,h)
   s←h-l+1;
   alterHClean(l,h);
   biSort(l,l+s/2-1); biSort(l+s/2,h);
sort(l,h)
   if l=h then return;
   s←h-l+1;
   sort(l,l+s/2-1); sort(l+s/2,h);
   merge(l,h);
main()
   sort(0,n-1);
```

**Figure 1. Sequential Merge Sorting Network Algorithm**

It is a divide-and-conquer merge sort algorithm as shown in the function sort(). Sorting of array w between (l, h) is accomplished by first sorting the two sub arrays (l,l+s/2-1) and (l+s/2,h), where s=h-l+1 is the size of the section of w to be sorted. Then the two sub arrays are merged.

The merge function merge() does an alternative half-clean for the whole section (l,h) followed by two bitonic sorts, biSort(), on the two sub arrays (l,l+s/2-1) and (l+s/2,h).

Bitonic sort function biSort() performs a half-clean, halfClean(), on the whole section (l,h), followed by two recursive calls to itself on the two sub arrays (l,l+s/2-1) and (l+s/2,h).

Half-clean function halfClean() and alternative half-clean function alterHClean() compare and switch half of the section with the other half using different patterns.

## 3.2 Parallel Merge Sorting Network Algorithm

To parallelize the sequential merge sorting network algorithm, we need to find independent operations. The divide-and-conquer nature of the functions sort(), merge() and bisort() indicates that the recursive calls to the two subsections are completely independent and, thus, can be executed in parallel.

We borrow the cobegin statement from the parallel programming language Chapel [8] to express parallel executions of the statements in a block {}. The cobegin statement does not finish until all of the parallel statements in a block are finished. Thus, a barrier synchronization for the statements in the block is implied.

All of the compare and switch calls in the functions halfClean() and alterHClean() operate on disjoint sets of elements of array w. Therefore, these functions can be executed in parallel. We use the forall statement (also from Chapel) to express a for loop whose iterations are executed in parallel. There is an implied barrier synchronization at the end of the loop. Thus, the forall loop does not finish until all iterations are finished. The parallel merge sorting network algorithm is shown in Figure 2.

```
int w[n];
compSwitch(i,j)
   a←w[i]; b←w[j];
   if (a≤b)  then {w[i]←a; w[j]←b}
             else  {w[i]←b; w[j]←a}
halfClean(l,h)
   s←h-l+1;
   forall i←0 to s/2-1
      compSwitch(l+i,l+i+s/2);
alterHClean(l,h)
   s←h-l+1;
   forall i←0 to s/2-1
      compSwitch(l+i,h-i);
biSort(l,h)
   if l=h then return;
   s←h-l+1;
   halfClean(l,h);
   cobegin {biSort(l,l+s/2-1); biSort(l+s/2,h)}
merge(l,h)
   s←h-l+1;
   alterHClean(l,h);
   cobegin{biSort(l,l+s/2-1); biSort(l+s/2,h)}
sort(l,h)
   if l=h then return;
   s←h-l+1;
   cobegin{sort(l,l+s/2-1); sort(l+s/2,h)};
   merge(l,h);
main()
   sort(0,n-1);
```

**Figure 2. Parallel Merge Sorting Network Algorithm**

The parallel merge sorting network algorithm in Figure 2 specifies the maximum number of parallel operations (parallelism), resulting in a theoretical complexity of $O(\lg^2 n)$. This is due to the fact that there are lgn merges and each merge has a maximum of lgn alternative half-cleans.

Parallel operations need to be scheduled on parallel processors at run time. The barrier synchronization implicit in the forall and cobegin statements incur run time overhead. The complexity analysis of the parallel merge sorting network algorithm ignores this overhead. Thus, the $O(\lg^2 n)$ time bound is not realistic.

Increased parallelism results in greater overhead. If parallelism is not controlled, the run time scheduling and synchronization overhead will overwhelm the computation. This overhead will result in the execution time of the parallel merge sorting network algorithm being greater than that of the sequential merge sorting network algorithm. Therefore, the parallel merge sorting network algorithm in Figure 2 is not scalable. As a result, as the problem size n increases, there will be a corresponding decrease in parallel performance.

## 3.3 Scalable Parallel Merge Sorting Network Algorithm

To overcome the problems of the parallel merge sorting network algorithm, we create a new variable, np, to represent the number of processors. We also limit the number of parallel operations in each step to be less than or equal to np. We implement these modifications by switching to sequential execution when there are not enough processors available for parallel function calls.

Because the parallelism is restricted, the increased problem size n does not increase the run time scheduling and synchronization overhead. Thus, the performance is scalable. We call this new algorithm the scalable parallel merge sorting network algorithm as shown in Figure 3.

The three functions sort, merge and biSort each contain a divide-and-conquer section and each function call works on a sub problem size of s=h-l+1. The total size of the array w is n. Thus, there are a total of n/s parallel calls to each these three functions. Note also that n, s and n/s are all powers of 2. If $n/s \geq np$, then each function call cannot have more than one processor and we execute divide-and-conquer sequentially. Otherwise, if $n/s < np$, then each of the function calls can be associated with at least one processor and we spawn two parallel divide-and-conquer function calls.

A similar argument exists for functions halfClean and alterHClean. Again, if $n/s \geq np$, then each function call cannot have more than one processor and we execute all of the compare and switch operations sequentially. Otherwise, if $n/s < np$, then $m = \lfloor np/(n/s) \rfloor$ is the number of processors available for the call. We divide the total s/2 compare and switch operations into m chunks and spawn m parallel operations with the calls halfCleanChunk(l,h,j,m) or alterHCleanChunk(l,h,j,m). Both of these calls result in sequential compare and switch on the corresponding block of indexes. In order to divide the sz = s/2 index evenly into m blocks, the starting and ending indexes of block j ($0 \leq j \leq m-1$) are $\lfloor j*sz/m \rfloor$ and $\lfloor (j+1)*sz/m-1 \rfloor$, respectively, according to [9].

```
int w[n];
int np; // number of processors
compSwitch(i,j)
  a←w[i]; b←w[j];
  if (a≤b)  then{w[i]←a; w[j]←b} else{w[i]←b; w[j]←a}
halfCleanChunk(l, h, j, m)
  s←h-l+1; sz←s/2; bz←sz/m;
  for i←j*bz to (j+1)*bz-1 compSwitch(l+i,l+i+sz);
halfClean(l,h)
  s←h-l+1;
  if  (n/s>=np) then //not enough processors
    for i←0 to s/2-1 compSwitch(l+i,l+i+s/2);
  else // use m=np/(n/s) processors
     m←np/(n/s); forall j←0 to m-1 halfCleanChunk(l, h, j, m);
alterHCleanChunk(l, h, j, m)
  s←h-l+1; sz←s/2; bz←sz/m;
  for i←j*bz to (j+1)*bz-1 compSwitch(l+i,h-i);
alterHClean(l,h)
  s←h-l+1;
  if  (n/s>=np) then //not enough processors
    for i←0 to s/2-1 compSwitch(l+i,h-i);
  else // use  m=np/(n/s) processors
    m←np/(n/s); forall j←0 to m-1 alterHCleanChunk(l, h, j, m);
biSort(l,h)
  if (l=h) then return; s←h-l+1;
  halfClean(l,h);
  if  (n/s>=np) then // not enough processors
    biSort(l,l+s/2-1); biSort(l+s/2,h);
  else
    cobegin {biSort(l,l+s/2-1); biSort(l+s/2,h)}
merge(l,h)
  s←h-l+1; alterHClean(l,h);
  if  (n/s>=np) then // not enough processors
    biSort(l,l+s/2-1); biSort(l+s/2,h);
  else
    cobegin {biSort(l,l+s/2-1); biSort(l+s/2,h)}
 sort(l,h)
  if l=h then return; s←h-l+1;
  if  (n/s>=np) then // not enough processors
    sort(l,l+s/2-1); sort(l+s/2,h)
  else
    cobegin {sort(l,l+s/2-1); sort(l+s/2,h)}
  merge(l,h);
main()
  sort(0,n-1);
```

**Figure 3. Scalable Parallel Merge Sorting Network Algorithm**

# 4. IMPLEMENTATION OF SCALABLE PARALLEL MERGE SORTING NETWORK ALGORITHM USING GO AND CILK

We implemented the scalable parallel merge sorting network algorithm in two multi-core parallel programming languages, Go and Cilk-5. Both languages had positive and negative syntax coverage in two parallel programming concepts, synchronization and parallel function calls.

## 4.1 Go

Go provides a CSP channel variable type for communication and synchronization and parallel function calls (termed goroutines) for concurrent computations. The synchronization at the end of a parallel loop (forall in our algorithm) is implemented by a single channel to which each goroutine (working as a parallel iteration) sends a value upon completion. The type of the sent value is not important (in our case, it is an integer). At the end of parallel

loop, there is a second for loop used to receive values from the channel. One value is received from each goroutine, for a total of m received values. For example, consider function halfClean() from Figure 3 implemented in Go as shown in Figure 4.

```
func halfClean(l,h int) {
 s:=h-l+1
 if n/s>=np {
  for i:=0; i<s/2; i++ {
   compSwitch(l+i,l+i+s/2)
  }
 } else {
  m:=s*np/n
  c:=make(chan int,1)
  for j:=0; j<m; j++ {
   go halfCleanChunk(l,h,j,m,c)
  }
  for i:=0; i<m; i++ {
   <-c
  } // synchronize
 }
}
```

**Figure 4. Go Code For Half-Cleaner**

In Figure 4, channel c is created and passed to m halfCleanChunk goroutines as an argument. The final for loop implements barrier synchronization by waiting for m integers to be received from channel c.

Any function in a Go program can be called as a sequential or parallel routine. This feature provides great flexibility in coding parallel algorithms and results in compact parallel code. Additionally, any function (called sequential or parallel) can itself contain both sequential and parallel function calls. For example, in Figure 4, function halfClean can be called as both a parallel and a sequential routine and is allowed to contain both the sequential function call to compSwitch and the parallel function call go halfCleanChunk.

## 4.2 Cilk

Cilk's spawn statement (similar to the Go statement go) is used to start a separate concurrent computation. Cilk also provides the concise sync statement for barrier synchronization. For example, consider function halfClean() from Figure 3 implemented in Cilk as shown in Figure 5.

```
cilk void halfClean(int l, int h) {
 int s, m, j;
 s = h - l + 1;
 m = s*np / n;
 for(j=0; j < m; j++) {
  spawn halfCleanChunk(l,h,j,m);
 }
 sync;
}
```

**Figure 5. Cilk Code For Half-Cleaner**

A spawned routine must be declared as a parallel routine with the keyword cilk. Only cilk functions can spawn other parallel routines. Consequently, in the Cilk implementation of the function halfClean from Figure 3 (and other functions), we were forced to declare two versions of every function. One function was declared as a parallel function and preceded by the cilk keyword. A second (identical) function was declared as a sequential function and was not preceded by the cilk keyword. This limitation caused our Cilk code to be twice as long as our Go code.

# 5. PERFORMANCE EVALUATION

We ran both the Go and Cilk implementations of the scalable parallel merge sorting network algorithm on an 8-core AMD Opteron 2.8GHz CPU with 1MB cache and 32 GB memory. The purpose of the experiment was to evaluate the speedup of parallel execution and to verify the scalability as problem size increases.

Our problem size was n = $2^{13}$, $2^{15}$, … , $2^{27}$. Array w[n] consisted of n distinct integers initially sorted in descending order (worst case). Our Go and Cilk implementations sorted w[n] into ascending order. For each problem size n, we ran our Go and Cilk implementations using number of processors np = 1, 2, … , 8. For each problem size n and number of processors np, we ran our Go and Cilk implementations 5 times to determine the average execution time.

Go execution time is shown in Figure 6. Note that the y axis of Figure 6 is logarithmic.

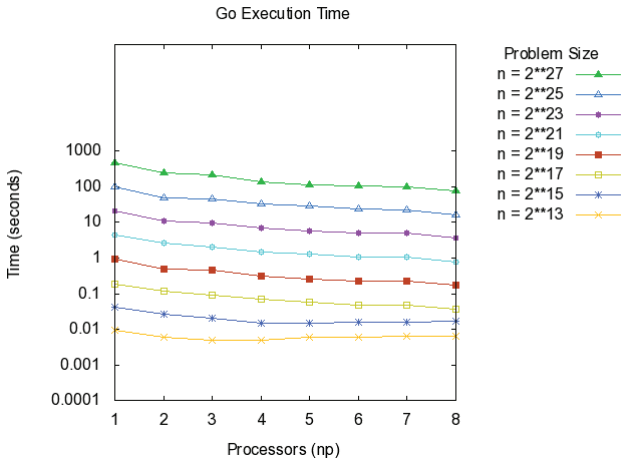


**Figure 6. Go Execution Time**

We calculated the speedup of parallel execution time using p processors over the sequential execution time of 1 processor as

$$S_p = \frac{T_1}{T_p}$$

where $T_p$ and $T_1$ are the execution times using $p$ and 1 processor(s), respectively. Go speedup $S_p$ is shown in Figure 7.

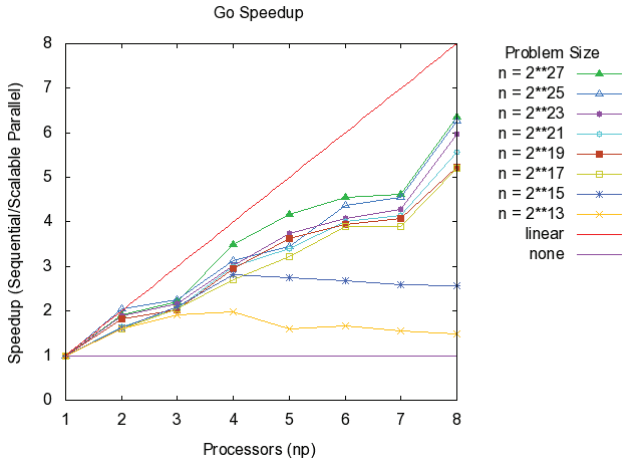


**Figure 7. Go Speedup**

The Cilk execution time and speedup are shown in Figures 8 and 9, respectively.

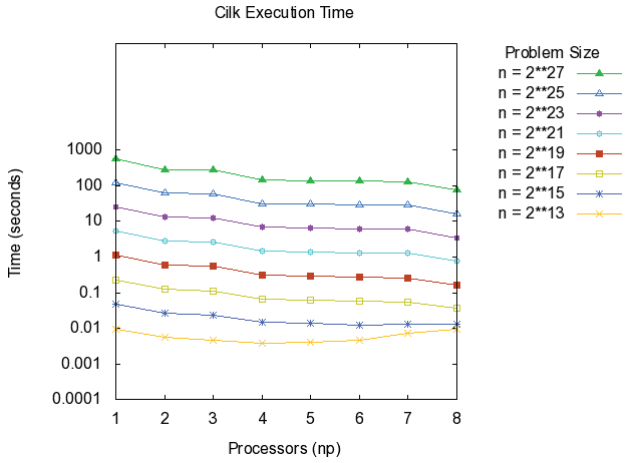


**Figure 8. Cilk Execution Time**

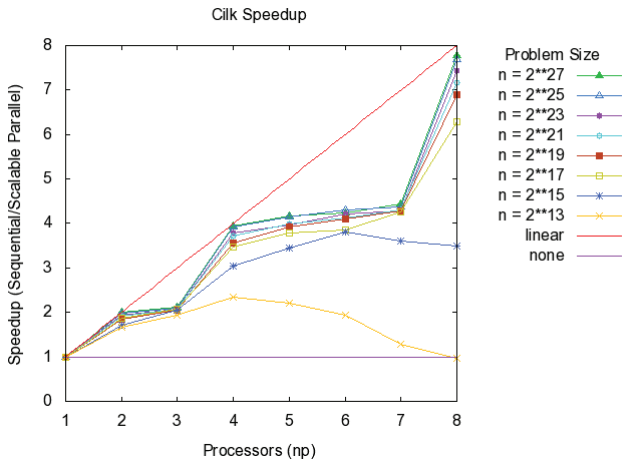


**Figure 9. Cilk Speedup**

Both Go and Cilk demonstrated that when problem size is small, i.e. $n=2^{13}$, speedup decreases as the number of processors np increases from 4 to 8. For small problem size, the size of each parallel task (the number sequential operations) is also small, and the scheduling and synchronization overhead of the parallel tasks is not negligible when compared to the small task execution time. The scheduling and synchronization overhead increases as the number of parallel tasks increases. As a result, increasing the number of processors increases the total execution time because a greater number of processors allow more parallel tasks.

For problem size $n=2^{17}$ and above, the speedup becomes normal. A greater the number of processors results in less execution time. Also, the larger the problem size, the larger the size of each parallel task, while the scheduling and synchronization overhead is determined by the number of parallel tasks and remains constant if the number of processors np is fixed. As a result, the scheduling and synchronization overhead have less impact on the execution time as problem size n increases. Therefore, increased problem size results in increased speedup in both Go and Cilk implementations (see Figures 7 and 9).

Note that when np is a power of 2 (i.e. np=2, 4, 8), the speedup of Cilk (see Figure 9) is almost perfect (i.e. linear) for large problem sizes.

We also observed that when np=5, 6 and 7, the speedup barely increases and is close to that of np=4 for both Go and Cilk implementations. The same can be said about the speedup of np=3, which is close to that of np=2. This is because the number of parallel tasks running parallel sort(), biSort() and merge() (see Figure 3), particularly at the lowest level, is always a power of 2. The number of processors above a power of 2 do not reduce the execution time of these parallel functions. The additional processors only assist in speedup of execution times of the halfClean() and alterHClean() functions.

Comparing Figures 7 and 9, we notice that Cilk has a greater speedup than Go. This indicates that the scheduling and synchronization overhead relative to parallel task computation time in Cilk is less than in Go. This is because synchronization in Cilk is built into the language and scheduling is implemented by the efficient work-stealing algorithm. In contrast, the barrier synchronization in Go is implemented by the programmer as a for loop (see Figure 4) that receives values from a channel.

We compared the execution times of Go and Cilk and found that Go is faster than Cilk in most combinations of problem size n and number of processors np.

Figure 10 shows the ratio

$$R = \frac{T_{cilk}}{T_{go}}$$

where $T_{cilk}$ and $T_{go}$ are the execution times of Cilk and Go, respectively. Note that when np=1, Cilk is 25% slower than Go for larger problem sizes. This proves Go is faster than Cilk as a sequential language, due to efficient compiler optimization and code generation. Cilk remains noticeably slower than Go except for np=4 and np=8.

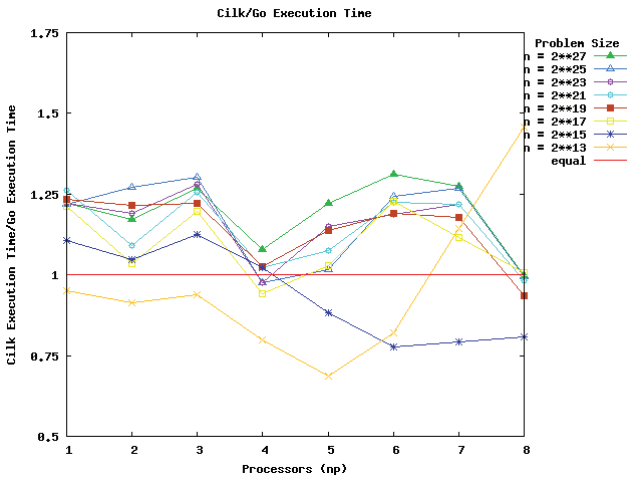


**Figure 10. Cilk/Go Execution Time**

## 6. CONCLUSIONS

A large number of sequential algorithms await conversion to parallel algorithms. Without this effort, the processing power of multi-core computers will be wasted.

We created a scalable parallel merge sorting network algorithm and implemented it using two modern shared-memory parallel programming languages, Go and Cilk-5. Our experimental results show that our parallel algorithm is scalable. Both Cilk and Go implementations increase computation speed for larger problem sizes. We compared Go and Cilk scheduling and synchronization efficiency and found that Cilk is more efficient and, thus, delivers higher speedup than Go. We compare the language efficiency of Go and Cilk and found that Go is faster than Cilk.

We have proven in our study that even decades old algorithms, such as the merge sorting network algorithm, are ripe for improvement when implemented in modern parallel languages such as Go and Cilk.

We would like to see future work using Go and Cilk to parallelize other sequential algorithms. Only then will the multi-core revolution realize its true potential.

## 7. ACKNOWLEDGMENTS

We thank Albert Everett, system administrator of our test computer, for extensive help installing Go and Cilk and for maintaining an environment conducive to accurate results.

This work was supported in part by the National Science Foundation under Grant CRI CNS-0855248, Grant EPS-0701890, Grant EPS-0918970, Grant MRI CNS-0619069, and OISE-0729792.

## 8. REFERENCES

[1] *The Go Programming Language.* [Online]. Available: http://golang.org/

[2] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou (1995): Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 1995.* pp. 207–216.

[3] D. Knuth (1973). *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Addison-Wesley, 1973.

[4] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. (2001). *Introduction to Algorithms.* McGraw-Hill, 2001, pp. 704-724.

[5] X. Ye, D. Fan, W. Lin, N. Yuan, and P. Ienne (2010): High Performance Comparison-Based Sorting Algorithm on Many-Core GPUs. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS), 2010.* pp. 1-10.

[6] J. Seiferas (2009): Sorting Networks of Logarithmic Depth, Further Simplified. In *Algorithmica*, vol. 53, no. 3, pp. 374-384, 2009.

[7] S. Olariu, M. Pinotti, and S. Zheng (1999): How to Sort N Items Using a Sorting Network of Fixed I/O Size. In *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 5, pp. 487-499, May 1999.

[8] *Chapel Programming Language: Overview.* [Online]. Available: http://chapel.cray.com/index.html

[9] M. Quinn (2003). *Parallel Programming in C with MPI and OpenMP.* McGraw-Hill, 2003, p. 119.