Comparing Parallel Performance of Go and C++ TBB on a Direct Acyclic Task Graph Using a Dynamic Programming Problem

Doug Serfass
Department of Computer Science
University of Arkansas at Little Rock
Little Rock, Arkansas 72204
djserfass@ualr.edu

Peiyi Tang
Department of Computer Science
University of Arkansas at Little Rock
Little Rock, Arkansas 72204
pxtang@ualr.edu

ABSTRACT

Concurrent programming languages Go and C++ Threading Building Blocks (TBB) offer high level parallel programming mechanisms built on top of threads. Go goroutines and TBB task classes are used as the computation units that are mapped to physical threads on multi-core processors. The synchronization mechanisms in Go and TBB are the channel and the task scheduler, respectively. We utilized these mechanisms to implement a parallel version of the optimal binary search tree dynamic programming algorithm in Go and TBB. Both implementations tile the iteration space and construct and evaluate a direct acyclic task graph for optimal parallelism without over constraints. We compared Go and TBB speedup and performance to create a benchmark of how efficient these two languages are at evaluating a direct acyclic task graph. Our experimental results show that the overhead of task scheduling and synchronization in TBB is much smaller than Go and that the overall performance of TBB is 1.6 to 3.6 times faster than Go. TBB provided super linear speedup under certain conditions, which we attribute to the majority of the test data being cached and the negative cost of task scheduling and synchronization. We conclude that TBB task scheduling and synchronization is faster than Go and that the top speedup of TBB is greater than that of Go.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Parallel programming; D.3.2 [**Language Classification**]: Concurrent, Distributed and Parallel Languages; I.2.8 [**Problem Solving, Control Methods, and Search**]: Dynamic Programming

General Terms

Algorithms, Performance, Design, Languages, Theory

Keywords

Optimal Binary Search Tree, Multi-Core Multiprocessor, Language Go, Language C++ TBB, Direct Acyclic Task Graph, Speedup and Performance

1. INTRODUCTION

Multithreading parallel programming using raw threads is similar to assembly programming in that it is low level, error prone and results in low programmer productivity. Race conditions may occur if concurrent threads accessing the same data are not properly synchronized. OpenMP [1] features only a parallel loop mechanism and tends to over constrain parallel computations, resulting in potential parallelism not being fully exploited. Recent research has attempted to solve these problems and to raise parallel programming to a new level of productivity.

Go [2] provides the goroutine as a parallel and concurrent computation unit. A goroutine is spawned by prepending the syntax "go" to the invocation of an otherwise normal sequential routine. The compiler generates code to map goroutines to individual physical threads, which are managed by the run-time system. This results in every core of a multi-core processor normally running one physical thread. Mapping a goroutine to a physical thread and switching between goroutines on a physical thread is fast and lightweight. The slower techniques of dispatching a thread to a core and thread context switching are used by older languages and libraries.

TBB [3] is similar to Go in this respect in that TBB uses the task class as the parallel computation unit. Mapping and scheduling of TBB tasks to physical threads is done by the TBB scheduler. The TBB scheduler runs on every physical thread and uses a depth first work and breadth first theft work stealing and scheduling algorithm. As in Go, mapping tasks to a physical thread and switching between tasks on a physical thread is fast and lightweight.

Go channels are the only synchronization mechanism available to goroutines. A Go channel is a high level, CSP synchronization mechanism. As a result, implementing any type of parallelism in Go, for example, parallel loop or direct acyclic task graph, requires channels.

TBB task synchronization is accomplished through a reference count. A TBB task will not be spawned and put into the ready queue of the physical thread until the task reference count is decremented to zero. It is the responsibility of a task's predecessor tasks to decrement its reference count.

A successor task will not be put on the ready queue until all of its predecessor tasks are complete. The scheduler picks up a task from the ready queue and invokes it (or steals a task from other threads if the ready queue is empty).

One of the most efficient parallel computations is to evaluate a direct acyclic task graph without over constraints. For this type of parallel computation, parallel programming simply consists of defining and specifying a task graph. Go and TBB facilitate specification and coding of task graphs. The performance of this type of parallel program is determined by the efficiency of task scheduling. Performance is also affected by synchronization among such variables as the grain size of each task and the cache locality achieved by grouping computations using nearby data into a task.

The purpose of this study is to compare the parallel performance, task scheduling and synchronization of Go and TBB. We selected the optimal binary search tree dynamic programming algorithm for our comparison. The parallel implementation of this algorithm uses a non trivial direct acyclic task graph. We implemented the algorithm in Go [4] and TBB and executed our implementations on an 8-core AMD Opteron processor. Our experimental results show that the overhead of task scheduling and synchronization is less in TBB. TBB has better performance and greater speedup for small grain size (larger parallelism) configurations. The overall performance of TBB is 1.6 to 3.6 times faster than Go. We observed super linear speedups of TBB for small grain size configurations, which we attribute to the majority of the test data being cached and the negative cost of task scheduling and synchronization.

Our contribution to parallel computing research is as follows: (1) We implemented the optimal binary search tree dynamic programming algorithm in the modern Go [4] and TBB multi-core parallel programming languages.; (2) We compared the performance of Go and TBB.; (3) We observed that the scheduling and synchronization cost of TBB is less than that of Go and that TBB is faster than Go.; (4) We discovered the grain size of a task which results in the greatest speedup in Go [4] and TBB.

This paper consists of the following sections: Section 2 is recent related work on comparative parallel language performance.; Section 3 is a brief description of the direct acyclic task graph parallel algorithm for the optimal binary search tree problem.; Sections 4 and 5 describe the scheduling, synchronization and implementation of the direct acyclic task graph in Go [4] and TBB.; Section 6 compares and analyzes the performance of our implementations.; Section 7 is our conclusions and suggestions for future research.

2. RELATED WORK

Bhattacharjee et al. [5] assessed the overheads that manifest at high core counts and small task sizes in TBB and OpenMP. Their study provides valuable insights for creating robust, scalable runtime libraries.

Zhao et al. [6] address the problem of reducing the total amount of overhead incurred by a program due to excessive task creation and termination. The authors introduce a transformation framework to optimize task parallel programs with finish, forall and next statements. The authors conclude that, for medium grained parallelism, the benchmarks studied in their paper provide evidence of significant improvement obtained by their transformation framework.

Chen and Johnson [7] illustrate common cache issues [of multi threaded programs] and show how current tools aid programmers in reliably diagnosing these issues. The authors promote discussions on whether such cache issues should be included in the Berkeley Our Pattern Language.

3. DIRECT ACYCLIC TASK GRAPH OF OPTIMAL BINARY SEARCH TREE PROBLEM

Dynamic programming is an efficient method for solving optimization problems with overlapping subproblems [8]. It utilizes a tabular approach from the bottom up to solve in polynomial time problems that would otherwise require exponential time. The optimal binary search tree problem can be solved by dynamic programming methods.

Given n keys, $a_1, \cdots, a_n$, and the probability distribution, $p_1, \cdots, p_n$, of their occurrence, the optimal binary search tree problem is to create the binary search tree of the keys with the minimum average search time. For each key in a binary search tree, the time needed to locate the key is equal to its level number. Let l_i be the level number of key a_i with the level number of the root node being 1. Then, the average search time of a key in the tree is $\sum_{i=1}^{n} l_i p_i$. The problem is to build an optimal binary search tree where this search cost is minimized.

If the optimal binary search tree for $a_1, \cdots, a_n$ has a_r $(1 \leq r \leq n)$ as its root, then its left sub tree containing $a_1, \cdots, a_{r-1}$ and its right sub tree containing $a_{r+1}, \cdots, a_n$ must both also be optimal and so are any of its sub trees. Let the cost (average search time) of the optimal binary search tree containing the keys $a_i, \cdots, a_j$ be $c(i, j)$.

Then, we must have

$$c(i,j) = \min_{i \leq r \leq j} (c(i, r-1) + c(r+1, j)) + \sum_{k=i}^{j} p_k \qquad (1)$$

where $c(i, i) = p_i$ for all $1 \leq i \leq n$ and $c(i, i-1) = 0$ for all $1 \leq i \leq n+1$.

The value of r that gives the minimum of the sums $c(i, r-1) + c(r+1, j)$ determines a_r as the root of the optimal binary search tree containing keys $a_i, \cdots, a_j$. Dynamic programming methods will compute and store $c(i, j)$ for small sub trees of $j - i = 1$ prior to computing and storing $c(i, j)$ of larger sub trees of $j - i = 2$ and so on.

The data structure to store $c(i, j)$ is the upper right triangular sub array of the matrix $cost[n+1][n+1]$. $c(i, j)$ is stored in $cost[i-1][j]$. The root of the optimal binary search tree containing $a_i, \cdots, a_j$ is stored in $root[i-1][j]$ of the matrix $root[n+1][n+1]$. The probability distribution of the keys is stored in the array $prob[n]$. The sequential dynamic programming algorithm for finding the optimal binary search tree [9] in Go [4] is shown in Figure 1, where $c(i, j)$ in equation (1) is calculated by the function `mst(i,j)`.

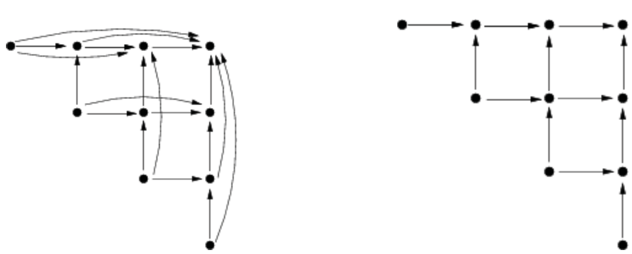
The computation of $cost[i][j]$ is a task and the data dependencies between tasks are shown in Figure 2(a). The computation of $cost[i][j]$ depends on all elements to the left and below $cost[i][j]$. Transitive edges are deleted, resulting in the reduced dependency graph shown in Figure 2(b).

If the grain size of parallel tasks is too small, then scheduling and synchronization costs can overwhelm parallel computation time. To control the grain size of parallel tasks, we partition the iteration space into $vp(vp+1)/2$ tiles. Each tile is a parallel task of larger grain size $\frac{n}{vp} \times \frac{n}{vp}$. The data

```
...
var (
  cost [n+1][n+1]float
  root [n+1][n+1]int
  prob [n]float
}

func mst(i,j int) {
  var   bestCost float = 1e9 + 0.0
  var   bestRoot int = -1
  switch  {
    case i >= j:
      cost[i][j] = 0.0
      root[i][j] = -1
    case i+1==j:
      cost[i][j] = prob[i]
      root[i][j] = i+1
    case i+1 < j:
      psum := 0.0
      for k := i; k <= j-1; k++ {
        psum += prob[k]
      }
      for r := i; r <= j-1; r++ {
        rcost := cost[i][r] + cost[r+1][j]
        if rcost < bestCost {
          bestCost = rcost
          bestRoot = r+1
        }
      cost[i][j] = bestCost + psum
      root[i][j] = bestRoot
      }
  }
}
func main()  {
  ...// initialize prob[]
  for i:=n; i>=0; i-- {
     for j:=i; j <= n; j++ {
       mst(i,j)
     }
  }
}
```

Figure 1: Sequential Dynamic Programming Algorithm for Finding the Optimal Binary Search Tree in Go



(a) Dependency Graph　　(b) Reduced Dependency Graph

Figure 2: Fine Grain Dependency Graphs

dependencies between tasks are shown in Figure 3. Tiles are tasks and a dot within a tile is the computation of $c(i, j)$. Arrows are a dependency between tiles. This graph is a *direct acyclic task graph*.

This graph results in maximum parallelism and minimum parallel execution time. While dependencies will vary by dynamic programming algorithm, an efficient dynamic programming parallel program must be based on this type of graph. The use of an over specified parallel construct, such as a parallel for loop and barrier synchronization, will result in a less efficient dynamic programming parallel program.

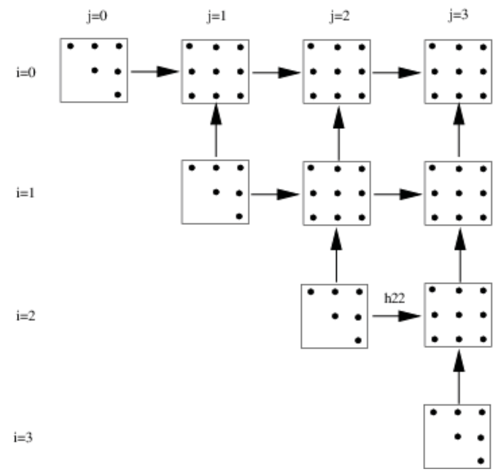


Figure 3: Direct Acyclic Task Graph

4. GO TASK GRAPH

Go is a concurrent, garbage collected systems programming language developed by Google. Go simplifies parallel programming by implementing concurrency through goroutines. Any Go function can be invoked as a normal routine or as a goroutine by adding the keyword go before a function call. A goroutine is executed concurrently with its calling routine.

The Go compiler generates code to map goroutines to individual physical threads which are managed by the run time system. Every core of a multi-core processor is normally running one physical thread. Mapping a goroutine to a physical thread and switching between goroutines on a physical thread is fast and lightweight. This technique is much faster than the traditional method of dispatching a thread to a core and thread context switch on a core.

The tasks of a task graph are implemented as goroutines. Dependency arrows of a task graph are implemented by channels. Channels are the only high level synchronization mechanism provided by the Go language. A channel is an extended CSP communication channel [10]. A channel with a non zero buffer size is used to implement asynchronous send (write). The implementation of the task graph of the optimal binary search tree in Figure 3 in Go was reported in [4].

5. TBB TASK GRAPH

TBB is a C++ library developed by Intel for multi-core parallel programming. Tasks are implemented through the tbb::task class. Task computation is specified by overriding the execute() method.

Mapping and scheduling of tasks to physical threads is done by the scheduler. The scheduler runs on each of the physical threads and uses a depth first work and breadth first theft work stealing scheduling algorithm for load balancing. Synchronization between tasks is implemented through the reference count, ref_count, of a task. A task will not be spawned and put into the ready queue of the physical thread by the scheduler until the task reference count is equal to zero.

To implement dependency arrows in a task graph, a predecessor task decrements the reference count of its successor task when the predecessor task completes its execute() method. The reference count of a successor task is initialized to a value equal to the number of its predecessors. A successor task will not be put in the ready queue until all of its predecessor tasks are complete and its reference count is equal to zero.

The implementation of the task graph of the optimal binary search tree problem in Figure 3 is shown in Figure 4. The class DagTask implements a task in the task graph. Each task has up to two successors stored in array successor[]. A task with no predecessors is called a *front* node. Front nodes in Figure 3 are the tiles on the main diagonal. A front node pointer DagTask* a is used to link front nodes so that all front nodes can be spawned in succession starting with tile $(0, 0)$.

The execute() method of the DagTask class is composed of three steps: (1) Spawn the next front node if the task is on the main diagonal.; (2) Call the function chunk(i, j, ...), which is similar to the chunk function of the Go code in [4].; (3) Decrement the reference count of the task successor(s).

The task graph is built and evaluated by the function build_evaluate(). This function first allocates and initializes data arrays *cost*, *root* and *prob*. Next, the function creates $vp * vp/2$ DagTask tasks (x[][]) and initializes each task reference count by calling set_ref_count(). Then, the function initializes successor node and front node pointers. Graph building is now complete. The first front node $x[0][0]$ (upper left corner of the graph) is spawned and the last node, $x[0][vp - 1]$ (upper right corner of the graph), waits for its predecessor nodes to complete.

6. PERFORMANCE COMPARISON

We chose problem size $n = 2000$ for our experiment in order to generate execution times large enough to provide meaningful data. We executed the Go code [4] and the TBB code in Figure 4 on an 8-core AMD Opteron processor. We varied the number of tiles in both dimensions, vp, from $2^0, 2^1, \cdots, 2^{10}$. The grain size of a task is $\frac{n}{vp} \times \frac{n}{vp}$. The larger the vp, the smaller the grain size. $vp = 1$ corresponds to the largest grain size with only one tile and no parallelism. $vp = n$ has the smallest grain size of 1 and maximum parallelism.

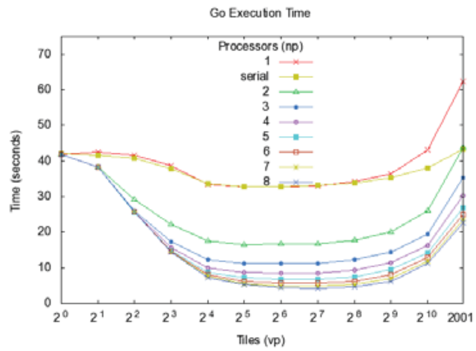
We also varied the number of physical threads, np, from $1, 2, \cdots, 8$. For each configuration of vp and np, we ran the code 5 times in single user mode and calculated the average. The execution time of the Go code was reported in [4] and is repeated in Figure 5(a) for comparison. The execution time of the TBB code is shown Figure 5(b).

$np = 1$ in Figure 5(a) and Figure 5(b) reaches its lowest

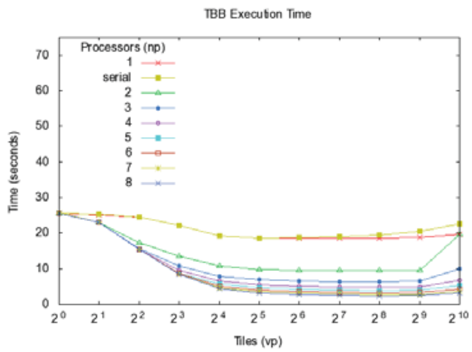
```
class DagTask: public tbb::task {
  const int i,j, vp, n;  float* const prob;
  float** const cost;  int** const root;
public:
  DagTask* successor[2];
  DagTask* a; //next task of the front
  DagTask( int i_, int j_, int vp_, int n_,
  float* prob_, float** cost_, int** root_ )
    : i(i_), j(j_), vp(vp_), n(n_),
    prob(prob_), cost(cost_), root(root_) {}
  tbb::task* execute() {
    if( i==j && i!=vp-1 ) { //main diagonal
      spawn( *a );          //front node
    }
    chunk( i, j, vp, n, prob, cost, root );
    for( int k=0; k<2; ++k )
      if( DagTask* t = successor[k] )
if( t->decrement_ref_count()==0 )
  spawn( *t );
    return NULL;
  }
};

float build_evalute( int vp ) {
  int n;  float *prob;
  float **cost;  int **root;
  DagTask* x[vp][vp];
   .
   .//allocate,initialize arrays cost,root,prob
   .
  //create tasks
  for(int d = 0; d < vp; d++)
    for(int i = 0; i+d < vp; i++) {
      x[i][i+d] =
      new( tbb::task::allocate_root() )
      DagTask(i, i+d, vp, n, prob, cost, root);
      x[i][i+d]->set_ref_count( d==0 ? 0 : 2 );
    }
  //set up successor links and front link
  for(int d = 0; d < vp; d++)
    for(int i = 0; i+d < vp; i++) {
      x[i][i+d]->successor[0] =
      i-1>-1 ? x[i-1][i+d] : NULL;// up
      x[i][i+d]->successor[1] =
      i+d+1<vp ? x[i][i+d+1] : NULL;// right
      if( d==0 && i<vp-1 ) //main diagonal
        x[i][i+d]->a = x[i+1][i+1];
    }
  x[0][vp-1]->increment_ref_count();
  x[0][vp-1]->spawn_and_wait_for_all(*x[0][0]);
  x[0][vp-1]->execute();
  tbb::task::destroy(*x[0][vp-1]);
  return cost[0][n];
}
```

Figure 4: TBB Task Graph Implementation



(a) Go Execution Time



(b) TBB Execution Time

Figure 5: Go and TBB Execution Times

point at $vp = 2^5$ (tile grain size 64×64) and $vp = 2^6$ (tile grain size 32×32), respectively. This is due to cache locality introduced by tiling. This is parallel code running on one thread, but, still creating goroutines and tasks and enforcing synchronization.

We also ran sequential, but still tiled, code (plotted as "serial" in Figure 5(a) and Figure 5(b)). Where $np = 1$ is greater than "serial", the difference between these times is the cost of scheduling and synchronization. Observe this difference increases in Figure 5(a) when vp changes from 2^8 to 2^{10}. However, in Figure 5(b), note $np = 1$ is less than "serial" except for $vp = 2^0$, and, the difference between these times increases as vp increases. Therefore, on one physical thread, parallel code is faster than sequential, and the cost of task scheduling and synchronization in TBB is negative.

Greater scheduling and synchronization cost in Go is also reflected by the increase of $np = 2, \cdots, 8$ as vp goes from 2^7 to 2^{10} in Figure 5(a). The corresponding times in Figure 5(b) actually decrease up to $vp = 2^7$ or $vp = 2^8$ and then increase gradually (except for $np = 2$).

To compare overall performance, we plotted the ratio

$$R = \frac{T_{Go}}{T_{TBB}}$$

where T_{Go} and T_{TBB} are the execution times of Go and TBB, respectively, in Figure 6. The ratio of the "serial" (sequential tiled) code between Go and TBB is about 1.6. This indicates the base language of TBB is 1.6 times faster than Go.

This ratio increases abruptly when vp increases from 2^7

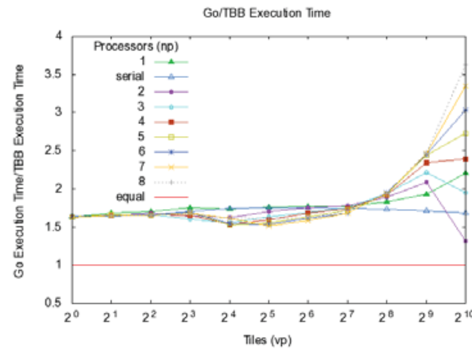


Figure 6: Go/TBB Execution Time

to 2^{10}. This indicates the difference in scheduling and synchronization costs between Go and TBB increases as the number of tasks increases. The greater the number of physical threads (cores) np, the greater the ratio. For $vp = 2^{10}$ and $np = 8$, TBB is more than 3.5 times faster than Go.

$np = 1$ includes the unnecessary overhead of task scheduling and synchronization not required by sequential execution. We use this tiled sequential time as the base to calculate speedup. The speedup of using $np \geq 1$ processors is then

$$S_{np} = \frac{T_{ts}}{T_{np}} \quad (2)$$

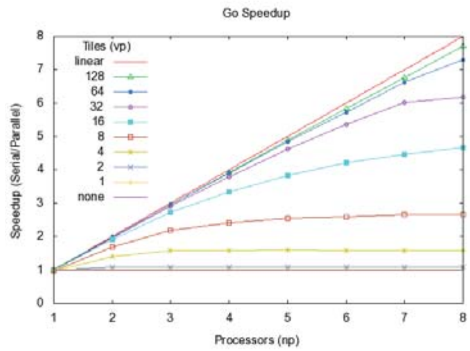
where T_{ts} and T_{np} are the tiled sequential time and the parallel time using np processors, respectively.

Go speedup was reported in [4] and is shown in Figure 7 for comparison. TBB speedup is shown in Figure 8. The greatest Go speedup is achieved when $vp = 128$ (grain size 16×16) and is almost linear. The speedup of $np = 8$ cores is 7.70. As vp decreases from 128, the speedup decreases due to diminished parallelism. As vp increases from 128, speedup decreases due to increased overhead of task scheduling and synchronization.

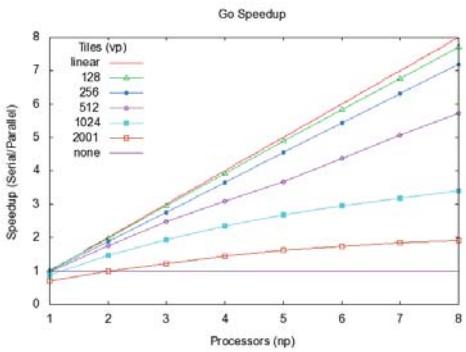
For TBB, when vp is less than or equal 128, speedup decreases due to diminished parallelism. For $vp = 256$ and $vp = 512$, we observed super linear speedup for TBB across all number of cores np. This is due to caching of data and the negative cost of task scheduling and synchronization. The tiled sequential time is greater than parallel code running on one core ($np = 1$). The greatest speedup is $vp = 512$ (grain size 4×4).

7. CONCLUSIONS

We presented implementations of a dynamic programming problem in Go [4] and TBB. To explore the maximum efficiency of parallel execution, both implementations were based on a direct acyclic task graph. Tasks graphs were implemented in Go [4] using goroutines and channels and in TBB using task classes and reference counts. We measured the execution time of both implementations and concluded: (1) Task scheduling and synchronization of TBB is faster than Go.; (2) The base language of C++ TBB is about 1.6 times faster than Go.; (3) The greatest Go speedup is achieved when the grain size of a task is about 16×16 and in TBB when the grain size is about 4×4.; (4) The top speedup of TBB is greater than Go.

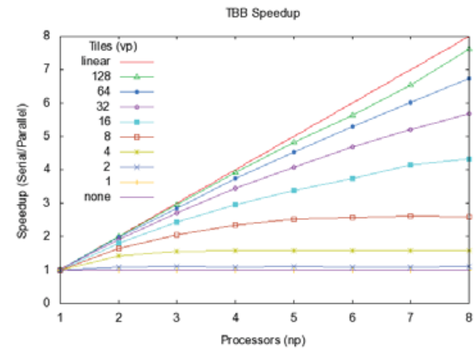


(a) Go Speedup $vp = 1, \cdots, 128$

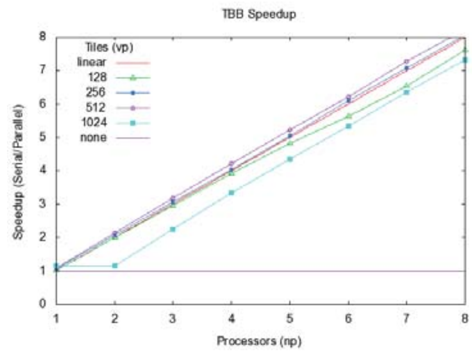


(b) Go Speedup $vp = 128, \cdots, 1024$

Figure 7: Go Speedup Over Sequential Time



(a) TBB Speedup $vp = 1, \cdots, 128$



(b) TBB Speedup $vp = 128, \cdots, 1024$

Figure 8: TBB Speedup Over Sequential Time

Future work is to simplify the implementation of dynamic programming algorithms in TBB. This can be achieved through the creation of a TBB template for parallel dynamic programming. This template could be used to implement other dynamic programming algorithms in order to generate additional direct acyclic task graph performance data.

8. ACKNOWLEDGMENTS

We thank Albert Everett, system administrator of our test computer, for extensive help installing Go and TBB 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.

9. REFERENCES

[1] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, Hightstown, New Jersey, 2003.

[2] Google. The Go Programming Language Specification - The Go Programming Language. `http://golang.org/doc/go_spec.html`, 2011. [Online; accessed 14-July-2011].

[3] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, Sebastopol, California, 2007.

[4] Peiyi Tang. Multi-core parallel programming in go. In *Proceedings of the First International Conference on Advanced Computing and Communications*, pages 64–69. Curran Associates, September 2010.

[5] Abhishek Bhattacharjee, Gilberto Contreras, and Margaret Martonosi. Parallelization libraries: Characterizing and reducing overheads. *ACM Transactions on Architecture and Code Optimization*, 8(1):5:1–5:29, April 2011.

[6] Jisheng Zhao, Jun Shirako, V. Krishna Nandivada, and Vivek Sarkar. Reducing task creation and termination overhead in explicitly parallel programs. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 169–180. Parallel Architectures and Compilation Techniques, September 2010.

[7] Nicholas Chen and Ralph Johnson. Patterns for cache optimizations on multi-processor machines. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, pages 2:1–2:10. ParaPLoP, March 2010.

[8] Thomas H. Corman, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 2nd Edition*. McGraw-Hill Book Company, Hightstown, New Jersey, 2001.

[9] Sara Baase and Allen Van Gelder. *Computer Algorithms: Introduction to Design and Analysis (3rd Ed)*. Addison-Wesley, Boston, Massachusetts, 2000.

[10] C.A.R. Hoare. *Communication Sequential Processes*. Prentice Hall, Saddle River, New Jersey, 1985.