OpenMP vs TBB: A survey on Parallel Programming Models

Metarya Ruparel (msr9732) **Safwan Mahmood (sm9453)**

1 Abstract

We compare two parallel programming approaches for multi-core systems: the well-known OpenMP and Threading Building Blocks (TBB) library by IntelR . The comparison is made using the parallelization of different real-world algorithm like MergeSort, Matrix Multiplication, and Two Array Sum. We develop several parallel implementations, and compare them w.r.t. programmability, scalability, run-time overhead, and the amount of control given to the programmer. We show that TBB requires a considerable program re-design, whereas with OpenMP simple compiler directives are sufficient. While TBB appears to be less appropriate for parallelizing existing implementations, it fosters a good programming style and higher abstraction level for newly developed parallel programs. Our experimental measurements on Two Intel Xeon E5-2680 (2.80 GHz) (20 cores) demonstrate that TBB slightly outperforms OpenMP in our implementation.

2 Introduction

Modern CPUs, even on desktop computers, are increasingly employing multiple cores to satisfy the growing demand for computational power. Thus, easy-to-use parallel programming models are needed to exploit the full potential of parallelism. Early parallel programming models (e.g. Pthreads) usually allow for flexible parallel programming but rely on low-level techniques: The programmer has to deal explicitly with processor communication, threads and synchronization, which renders parallel programming tedious and error-prone. Several techniques exist to free programmers from low-level implementation details of parallel programming. One approach is to extend existing programming languages with operations to express parallelism. OpenMP is an example of this approach. Another approach is to introduce support for parallel programming within a library. Threading Building Blocks (TBB), published by IntelR, adds support for high-level parallel programming techniques to C++.

The OpenMP application programming interface is quite popular for shared memory parallel programming. Basically, OpenMP is a set of compiler directives that extend C/C++ and Fortran compilers. These directives enable the user to explicitly define parallelism in terms of constructs for single program multiple data (SPMD), work-sharing and synchronization.

Threading Building Blocks (TBB) is a novel C++ library for parallel programming, which can be used with virtually every C++ compiler. In TBB, a program is described in terms of fine-grained tasks. Threads are completely hidden from the programmer. TBB's idea is to extend C++ with higher-level, task-based abstractions for parallel programming; it is not just a replacement for threads. At runtime, TBB maps tasks to threads.

3 Parallel Programs

Our approach focuses on experimentation and benchmarking. The algorithms chosen are simple and demonstrate a best-case scenario to benchmark parallelization in both OpenMP and TBB. As the workload is easily distributed, any overheads introduced by the parallel algorithms can be made apparent while comparing the performances.

Preprint. Under review.

We implement the following parallel programs to compare the parallel models.

3.1 Two Array Sum

Given two arrays, calculate the sum and store it in an output array. Given the sizes of these arrays we randomly generate the values to be used. This helps us focus on varying the scale of the problem rather than the trouble of array initialization. We recognise that since the loop iterations are independent, we can parallelise the loop that iterates over both the arrays.

We use OpenMP parallel constructs like #pragma omp parallel for to parallelize the loop.

```
#pragma omp parallel for num_threads(no_threads)
for(int i=0;i<N;i++){
    c[i] = a[i] + b[i];
}
```

The parallel parts are timed using the omp_get_wtime() call for performance analysis.

TBB implementation includes a class with TBB construct of operate() which contains the business logic of the Two Array Sum algorithm. The class is passed to the parallel_for construct of TBB. The arrays are passed as pointers to the class.

```
class TwoArraySum {
  int *p_a;
  int *p_b;
  int *p_c;
public:
  TwoArraySum(int * a, int * b, int * c) : p_a(a), p_b(b), p_c(c) {}

  void operator() ( const blocked_range<int>& r ) const {
    for ( int i = r.begin(); i != r.end(); i++ ) {
      p_c[i] = p_a[i] + p_b[i];
    }
  }

};
```

The class is passed into the construct parallel_for

```
parallel_for(blocked_range<int>(0, N),TwoArraySum(a,b,c));
```

The parallel parts are timed using the tick_count::now() for performance comparison analysis.

3.2 Matrix Multiplication

Matrix multiplication is an operation that takes a pair of matrices and produces another matrix. If A is a MxN matrix and B is a NxP matrix, then their matrix product AB is given by:

$$[A,B]_{i,j} = A_{i,1}B_{1,j} + A_{i,2}B_{2,j} + ... + A_{i,n}B_{n,j} = \sum_{r=1}^{n} A_{i,r}B_{r,j},$$

where $1 \le i \le m$ and $1 \le j \le p$.

Figure 1: Matrix Multiplication

After an examination of sequential algorithm for matrix multiplication, it was clear that the main part of the work is carried out within three for loops. First two loops are for initialization of matrices, and the third nested loop, is responsible for multiplication. As such, the primary focus of parallel optimization effort is on making the loop iterations run in parallel. It is evident from the structure of initialization and outer multiplication loops that there are no data dependencies between loop

iterations, unlike the inner loops which have several dependencies. This allows an implementation without the need of any locks. We only considers square matrices for simplicity. Both the matrix are generated randomly using the size provided by the user.

We use the #pragma omp parallel for construct on the outermost loop to parallelize the algorithm.

```
#pragma omp parallel for num_threads(no_threads)
    for (int i = 0; i < N; i++){
      for (int j = 0; j < N; j++){
          for (int k = 0; k < N; k++){
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
          }
       }
    }
```

The parallel parts are timed using the omp_get_wtime() call for performance analysis.

TBB implementation includes a class with TBB construct of operate() which contains the logic of matrix multiplication. The class is passed to the parallel_for construct of TBB. The arrays are passed as pointers to the class.

```
class Multiply
{
    //init
    Multiply(int ** a, int ** b, int ** c, int N): p_a(a), p_b(b), p_c(c), n(N) {}

    void operator()(blocked_range<int> r) const {
            for (int i = r.begin(); i != r.end(); ++i) {
                for (int j = 0; j < n; ++j) {
                    for (int k = 0; k < n; ++k) {
                        p_c[i][j] += p_a[i][k] * p_b[k][j];
                    }
                }
            }
        }
};
```

The class is passed into the construct parallel_for

```
parallel_for(blocked_range<int>(0,N), Multiply(a,b,c,N));
```

The parallel parts are timed using the tick_count::now() call for performance and comparison analysis.

3.3 Merge Sort

MergeSort is a comparison based, stable sorting algorithm with run-time complexity O(nlogn). Parallel mergeSort algorithms are in wide-spread use today being, particularly useful in online sorting. The algorithm sorts a sequence of comparable objects by splitting the input into two pieces and then solves each by applying recursion. The invocations of the recursive calls form a balanced binary-tree with depth log n, where n is the length of the sequence. After the tree has been expanded, a parent node merges the sorted sequences from its two children with the Merge algorithm. The Merge algorithm defines pointers to the start of each sorted sequences given as input. It then proceeds to store the element of whichever pointer points to the smallest element in a new list, after which the pointer is incremented. This is continued until all pointers have traversed their respective input sequences, yielding a combined sequence of all input sequences in sorted order.

High level idea of Merge Sort is as follows:

MergeSort belongs to a set of algorithms known as Divide and conquer algorithms. This makes it an ideal candidate for increased parallel performance since the partitioning of the problem into independent sub problems comes naturally.

$MergeSort(A, p, r)$
if $p < r$ **then**
  $q = floor((p + r)/2)$
  $MergeSort(A, p, q)$
  $MergeSort(A, q + 1, r)$
  $Merge(A, p, q, r)$
end if

Figure 2: Merge Sort

MergeSort parallelizes quite easily. Observe that MergeSort(A, p, q) and MergeSort(A, q+ 1, r) in figure 2 can be executed in parallel. Each node in the recursion tree has a dependency on its children. We tweak the merge part in the algorithm for better parallel implementation. Given two sorted sequences, s1 and s2, to be merged, we define m1 as a pointer to the middle element of s1, and m2 as a pointer to the position m1 would be inserted into s2 by a insertion based sorting algorithm. This splits s1 into two independent pieces a and b, where all elements in a are less than or equal to m1 and all elements of b are greater than m1. c and d are defined similarly for s2. a and c can now be merged on one thread, while b and d gets merged on another in parallel.

The algorithm can also split the sequences further if more than two threads are available. After all merges are complete the sequences are concatenated to yield the complete sorted sequence. The efficiency of the algorithm depends on m2 splitting s2 into similarly sized parts.

The implementation takes array size and threads as inputs and initializes the arrays randomly to save the trouble of large array initializations.

We use the task constructs in OpenMP to implement the merge sort. The array that has to be sorted is handled using the shared construct.

```
void mergeSort(int *X, int n, int *tmp)
{
   if (n < 2) return;

   #pragma omp task shared(X)
   mergeSort(X, n/2, tmp);

   #pragma omp task shared(X)
   mergeSort(X+(n/2), n-(n/2), tmp + n/2);

   #pragma omp taskwait
   mergeSortAux(X, n, tmp);
}
```

The merge sort function is passed to the parallel construct as follows:

```
#pragma omp parallel num_threads(no_threads)
{
    #pragma omp single
    mergeSort(X, N, tmp);
}
```

The parallel parts are timed using the omp_get_wtime() call for performance analysis.

The TBB implementation is done using task group construct. The merge part is tweaked into a parallel merge as discussed above.

The parallelMerge was implemented with task groups too. The parallel parts are timed using the tick_count::now() call for performance analysis.

```
void parallelMergeSort(int A[], int p, int r, int B[], int s) {
        int n = r - p + 1;
        if (n == 1) {
                B[s] = A[p];
        }
        else {
        int *T = new int[n];
        T--;
        int q = (p + r) / 2;
        int qp = q - p + 1;
        task_group tg;
        tg.run([=] { parallelMergeSort(A, p, q, T, 1); });
        tg.run([=] { parallelMergeSort(A, q + 1, r, T, qp + 1); });
        tg.wait();
        parallelMerge(T, 1, qp, qp + 1, n, B, s);
        T++;
        }
}
```

4 Programmability

To contrast the programmability of OpenMP and TBB we used the Two Array Sum (section 3.1) and Array Sum (a variant of Two Array Sum).

4.1 Loop Parallelization

We use Two Array Sum experiment for contrasting Loop Parallelization. To parallelize the summation part of the array, we need to parallelize the for loop of the Two Array Sum algorithm. OpenMP and TBB both offer constructs to define this kind of data parallelism.

In OpenMP, two constructs are used to parallelize a loop: the parallel and the loop construct. The parallel construct, introduced by a parallel directive, declares the following code section to be executed in parallel by a team of threads.

```
#pragma omp parallel num_threads(no_threads)
#pragma opm for
for(int i=0;i<N;i++){
        c[i] = a[i] + b[i];
}
```

Figure 3: OpenMP: Two Array Sum Loop Parallelization

Additionally, the loop construct, introduced by a for directive, is placed within the parallel region to distribute the loop's iterations to the threads executing the parallel region. Figure 3 shows parallelizing the loop using these two constructs.

```
#pragma omp parallel for num_threads(no_threads)
for(int i=0;i<N;i++){
        c[i] = a[i] + b[i];
}
```

Figure 4: OpenMP: Two Array Sum Loop Parallelization

Alternatively, we can also parallelize these loops without any need of these initializations. Therefore, we use the parallel for directive, which is a shortcut declaration of a parallel construct with just one nested loop construct (see Figure 4). Apart from the additional compiler directives, no considerable

changes were made to the sequential program. Thus, an OpenMP-based parallel implementation of the Two Array Sum Program is easily derived from a sequential implementation. OpenMP also supports several strategies for distributing loop iterations to threads. The strategy is specified via the schedule clause, which is appended to the for directive.

```
#pragma omp for schedule(static,chunk_size)
for(int i=0;i<N;i++){
        c[i] = a[i] + b[i];
 }
```

Figure 5: OpenMP: Two Array Sum Strategy

In our application, we expect that the workload is evenly distributed in iteration ranges of reasonable size. Therefore, a default directive that will result even distribution of iterations to threads will be sufficient for our application. OpenMP's default scheduling strategy suits our needs. It creates evenly sized blocks of loop iterations for all threads of a parallel region. Thereby, the block size implicitly is defined as n/p, where n is the number of iterations and p is the number of threads. A static scheduling directive would also suit our need since the workload is even. If chunk size is specified then each thread executes work equal to chunk size in round-robin fashion.

In TBB, parallel loops are defined using the parallel for template, which is applied in two steps as proposed by Reinders:

1. A class is created, which members correspond to the variables that are outside the scope of the loop to be parallelized. The class has to overload the () operator method so that it takes an argument of type blocked_range<T> that defines a chunk of the loop's iteration space which the operator's caller should iterate over. The method's body takes the original code for the loop. Then, the arguments for the loop's iteration bounds are replaced by calls to methods that return the iteration space's beginning and end.
2. The code of the original loop is replaced by a call to the parallel for pattern. The pattern is called with two arguments, the loop's iteration space and an instance of the previously defined class, called the body object. The iteration space is defined by an instance of the blocked_range<T> template class. It takes the iteration space's beginning and end and a grain size value. A parallel loop construct incurs overhead cost for every chunk of work that it schedules. TBB chooses grain sizes automatically if not defined, depending upon load balancing needs. This attempts to limit overheads while still providing ample opportunities for load balancing. The default automatic chunking is recommended for most uses.

```
class TwoArraySum {
  int *p_a;
  int *p_b;
  int *p_c;
  public:
    TwoArraySum(int * a, int * b, int * c) : p_a(a), p_b(b), p_c(c) {}
    void operator() ( const blocked_range<int>& r ) const {
        for ( int i = r.begin(); i != r.end(); i++ ) {
            p_c[i] = p_a[i] + p_b[i];
        }
    }

};

int main(int argc, char *argv[]) {
.
.
 task_scheduler_init init;
 parallel_for(blocked_range<int>(0, N, grain_size),TwoArraySum(a,b,c));
.
.
}
```

Figure 6: TBB: Parallel Two Array Sum

A parallel implementation of Two Array Sum using TBB thus requires us to re-implement the loop (see Figure 6). TBB's parallel for construct is a C++ template and takes a C++ class as parameter. Actually, this technique is a workaround for C++'s missing support of lambda expressions. With lambda expression, blocks of code can be passed as parameters. Thus, the code of the body object could be passed to the parallel for template in-place without the overhead of a separate class definition.

4.2 Thread Coordination

We use Array Sum experiment for contrasting Thread Coordination as Two Array Sum has no critical section as discussed above in Section 3.1. Array Sum is a trivial summations of all the elements in the given array. We have a global sum variable which is used by all the threads. As we see, that within the loops all threads perform multiple additions to determine the final sum. Hence, possible race conditions have to be prevented. There are two basic techniques for this:

1. Mutexes: The summation part is declared mutually exclusive, such that only one thread at a time is able to work on it.
2. Atomic operations: The summation is performed as an atomic operation.

In OpenMP, both techniques are declared by appropriate directives. Mutexes are declared by using the critical construct. Similarly to the aforementioned parallel construct, it specifies a mutual exclusion for the successive code section (See Figure 7).

```
for(int i=0;i<N;i++){
    #pragma omp critical
    //other updates
    x = 0;
        sums += a[i];
}
```

Figure 7: OpenMP: Critical Construct

OpenMP's atomic construct is used similarly to the critical construct (see Figure 8). It ensures that a specific storage location is updated without interruption (atomically). Effectively, this ensures that the operation is executed only by one thread at a time. However, while a critical region semantically locks the entire element the loop is working on, an atomic operation just locks the bin hash_map it wants to update. Thus atomic operations in our case may be regarded as fine-grained locks.

```
for(int i=0;i<N;i++){
    #pragma omp atomic
    sums += a[i];
}
```

Figure 8: OpenMP: Atomic Construct

Besides, mutual exclusion can be implemented explicitly by using low-level library routines. With these routines, in general, a lock variable is created. Afterwards a lock for this variable is acquired and released explicitly. This provides a greater flexibility than using the critical or atomic construct.

TBB provides several mutex implementations that differ in properties like scalability, fairness, being re-entrant, or how threads are prevented from entering a critical section. Scalable mutexes do not perform worse than a serial execution, even under heavy contention. Fairness prevents threads from starvation and, for short waits, spinning is faster than sending waiting threads to sleep. The simplest way of using a mutex in TBB is shown in Fig 9.

The mutex lock variable is created explicitly, while a lock on the mutex is acquired and released implicitly. Alternatively, a lock can be acquired and released explicitly by appropriate method calls, similarly to using the OpenMP library functions. Atomic operations can only be performed on a template data type that provides a set of methods (e.g. fetchAndAdd) for these operations.

```
{
 mutex::scoped_lock lock(countMutex);
 sum += a[i];
}
```

Figure 9: TBB: A simple Mutex Construct

4.3 Comparison

The previous code examples clearly show that TBB requires a thorough redesign of our program, even for a relatively simple pattern like parallel for. Our TBB-based implementations differ greatly from the original version. We had to create additional classes for the loop to be parallelized, and replace the parallelized program parts by calls to TBB templates. Basically, this is because TBB uses library functions that depend on C++ features like object orientation and templates. The most delicate issue was identifying the variables that had to be included within the class definition of TBB's parallel for body object.

Parallelizing the loop of the original Two Array Sum implementation using OpenMP is almost embarrassingly easy. Inserting a single line with compiler directives parallelizes a loop without any additional modifications. Another single line implements a mutual exclusion for a critical section or the atomic execution of an operation. Also, in contrast to TBB, we do not have to take any measures to change variable scopes. OpenMP takes care of most details of thread management.

Regarding the number of lines of code, OpenMP is far more concise than TBB. The parallel constructs are somewhat hidden in the program. TBB, on the other hand, improves the program's structure. Parallel program parts are transferred into separate classes. Though this increases the number of lines of code, the main program becomes smaller and more expressive through the use of the parallel for construct.

The parallel constructs of TBB and OpenMP offer a comparable level of abstraction. In neither case we have to work with threads directly. TBB's parallel for template resembles the semantics of OpenMP's parallel loop construct. OpenMP's parallel loop construct can be configured by specifying a scheduling strategy and a block size value, whereas TBB relies solely on its task-scheduling mechanism. However, OpenMP's parallel loop construct is easier to use, because the schedule clause may be omitted.

Regarding thread coordination, OpenMP and TBB offer a similar set of low-level library routines for locking. However, OpenMP additionally provides compiler directives which offer a more abstract locking method. Also, TBB has a serious drawback regarding atomic operations: these operations are only supported for integral types (and pointers).

In abridgement, using OpenMP is much easier and takes fewer lines of code as compared to TBB.

5 Scalability

For comparing the scalabililty of OpenMP and TBB, we use the Square Matrix Multiplication, and MergeSort benchmark. The aim is to test the algorithm on different number of threads and different data-set size. All the tests are run on Two Intel Xeon E5-2680 (2.80 GHz) (20 cores), with 128GB Memory running CentOS 7.

5.1 Methodology

In order to obtain the correct performance, each benchmark is repeated several times and the standard deviation is checked to be negligible. Average value is taken as representative of the test.

For the 1st experiment, we fix the input size and time the benchmarks on different number of threads. The idea is to capture the relative speed-ups of OpenMP and TBB. The number of threads is specified by an environment variable in OpenMP or by a parameter of the task scheduler's constructor in TBB, respectively.

For experiment 1, we fix the size of matrix to 512x512 and array size to 100,000. We run the parallel implementations of the Matrix Multiplication and MergeSort benchmark in TBB and OpenMP with the following number of threads:

1. 2
2. 4
3. 8
4. 16
5. 32
6. 40

For experiment 2, we will analyze the efficiency. We time both the benchmarks on different input sizes and different number of threads. Apart from running these benchmarks on the different number of threads as mentioned above, we will use the following input sizes:

1. Matrix Multiplication:
   a. 32x32
   b. 64x64
   c. 128x128
   d. 256x256
   e. 512x512
   f. 1024x1024
2. Merge Sort:
   a. 1000
   b. 10,000
   c. 100,000
   d. 1,000,000
   e. 10,000,000

5.2 Comparison - Matrix Multiplication

The table and graph below shows the comparison between speed-ups (taking the sequential performance as the baseline) achieved by OpenMP and TBB for the 512x512 Matrix Multiplication benchmark. The sequential implementation takes 0.827790 sec.

Speed-Up is calculated using the following equation:

$$speed\_up = \frac{time\ take\ by\ 1\ thread}{time\ take\ by\ t\ threads}$$

Model	2	4	8	16	32	40
OpenMP	1.82	3.15	7.05	9.08	10.02	11.28
TBB	1.75	3.36	9.31	14.00	13.93	13.41

Figure 10: Speed-Ups: OpenMP vs TBB

The graph demonstrates that all tested models show significant improvements in decreasing execution time compared to sequential execution. While it may appear that OpenMP performs poorly when compared to TBB, it must be noted that matrix multiplication algorithm is embarrassingly task-parallel problem that is more suited to TBB, which are task-parallel models, whereas OpenMP is better suited to data-parallel problems and may prove easier to implement and perform better when faced with data-parallel problems.

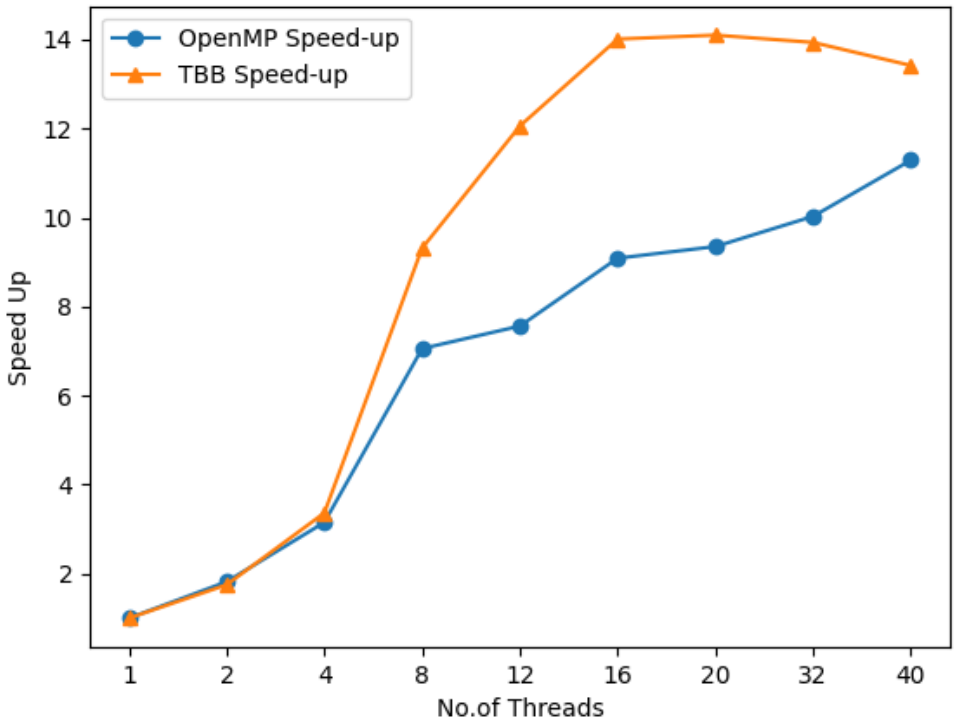

Figure 11: Speed-Up Graph: OpenMP vs TBB

The below table shows efficiency of OpenMP and TBB for the Matrix Multiplication benchmark, using different matrix sizes and number of threads. The efficiency is calculated using the following equation:

$$Efficiency = \frac{speed\_up}{no.\ of\ threads}$$

	2	4	8	16	32	40
32x32	0.64	0.58	0.14	0.007	0.017	0.002
64x64	0.92	1.55	0.51	0.05	0.073	0.04
128x128	1.01	2.32	0.97	0.31	0.041	0.15
256x256	0.92	1.90	0.80	0.70	0.58	0.34
512x512	0.90	1.02	0.62	0.58	0.46	0.28
1024x1024	0.75	0.88	0.67	0.54	0.45	0.37

	2	4	8	16	32	40
32x32	0.64	0.75	0.14	0.05	0.04	0.03
64x64	0.91	1.36	0.45	0.15	0.12	0.07
128x128	0.97	1.93	1.08	0.58	0.46	0.24
256x256	1.01	0.99	0.87	0.70	0.45	0.58
512x512	0.93	0.91	0.86	0.74	0.45	0.36
1024x1024	0.96	0.92	0.89	0.84	0.49	0.40

Figure 12: Left: OpenMP Efficiency; Right: TBB Efficiency

We see that as the problem size increases the efficiency of both the models also increases. However, for larger problem size with lesser number of threads the efficiency starts to drop. This could be because as the problem size increases we can benefit from using more number of threads working in

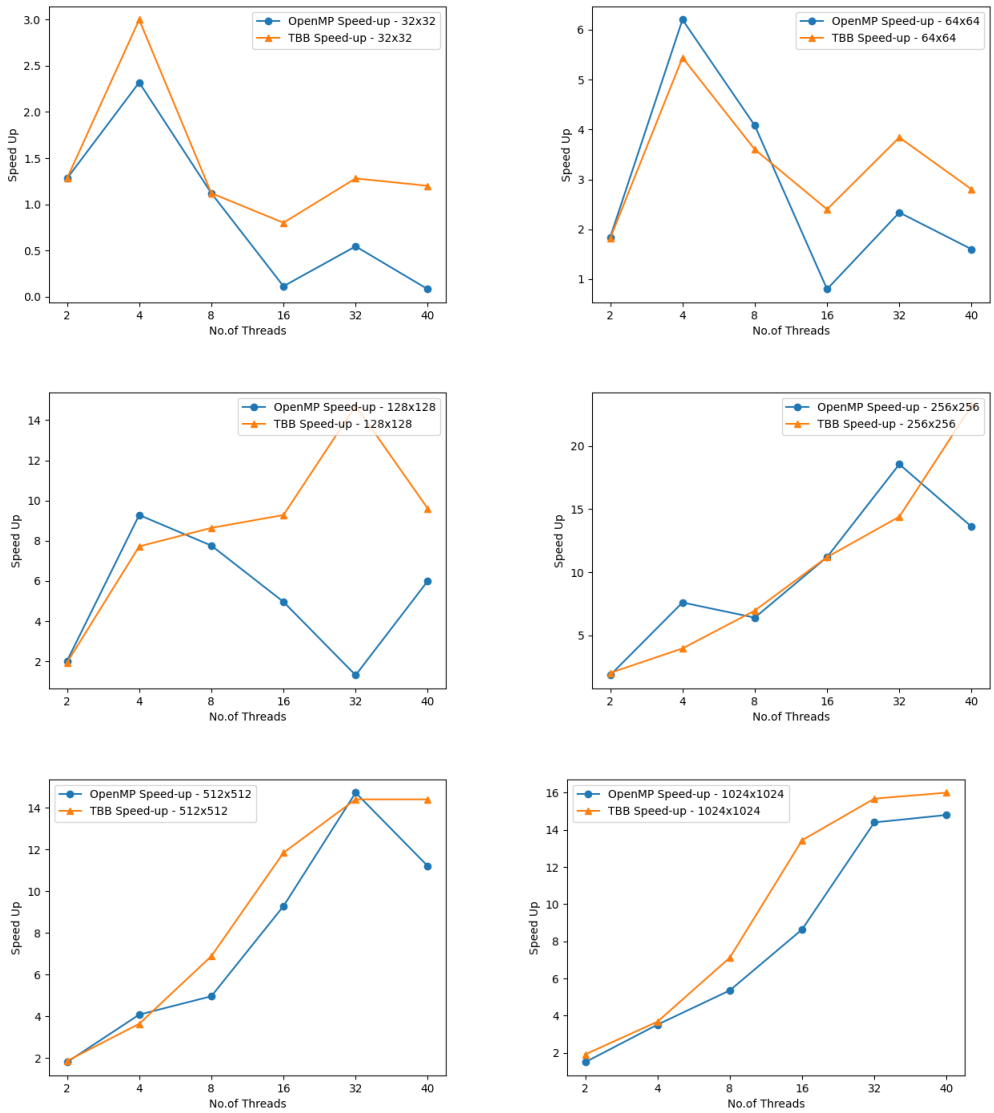

Figure 13: OpenMP vs TBB: Speed-up for different dataset sizes

parallel. The overhead of creating/maintaining more threads is overshadowed by the speed-up in case of larger data-set.

Looking at the efficiency numbers of TBB and OpenMP, we can conclude that TBB on average is more scalable and efficient than OpenMP for Matrix Multiplication benchmark. The reason being task based parallelism in TBB vs data parallelism used in OpenMP. Also, unlike OpenMP, TBB benefits from using logical threads. Upon instantiation, the task scheduler initializes several logical threads. Note the distinction between a logical thread, specified with a threading API, and a physical thread executing on a processor. These logical threads are placed in a global thread pool. By utilizing a thread pool, the scheduler removes the overhead of thread allocation and destruction between each parallel region.

In abridgement, TBB slightly over-performs OpenMP when tested on larger threads and data-sets.

5.3 Comparison - MergeSort

The table and graph below shows the comparison between speed-ups (taking the sequential performance as the baseline) achieved by OpenMP and TBB for the MergeSort benchmark with 100,000 data points generated randomly. The sequential implementation takes 0.033285 sec.

Model	2	4	8	16	32	40
OpenMP	1.22	2.72	3.06	5.41	3.61	2.86
TBB	1.36	3.23	3.54	5.87	6.69	7.85

Figure 14: Speed-Ups: OpenMP vs TBB

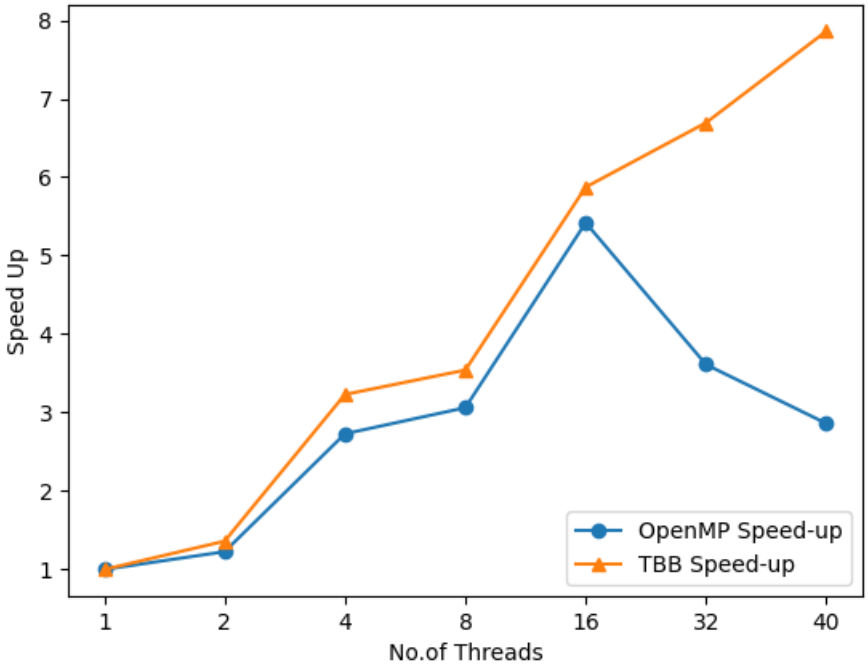

Figure 15: Speed-Up Graph: OpenMP vs TBB

We observed that with a significant large number of data-points both OpenMP and TBB performed faster than its sequential counter-part. However, the speed-up in TBB is comparatively higher than that of OpenMP.

In case of OpenMP we see a drop in speed-up after 16 threads. A possible reason for this drop can be the overhead of creating/maintaining threads. This overhead does not over shadow the corresponding speed-up. On the other hand, the parallel overhead in TBB is not much. A possible reason for this could be that, TBB uses the thread pool to divide the tasks. These task are light weight and there's no context switching between them.

The below table shows efficiency of OpenMP and TBB for the MergeSort benchmark, using different sizes of array and number of threads. The efficiency is calculated using the following equation:

$$Efficiency = \frac{speed\_up}{no.\ of\ threads}$$

	2	4	8	16	32	40
1000	0.44	0.18	0.078	0.028	0.009	0.0041
10000	0.86	0.45	0.24	0.04	0.02	0.018
100000	0.94	0.53	0.39	0.12	0.07	0.066
1000000	0.84	0.59	0.34	0.15	0.06	0.05
10000000	0.85	0.69	0.39	0.21	0.09	0.06

	2	4	8	16	32	40
1000	0.44	0.38	0.088	0.078	0.019	0.0081
10000	0.84	0.55	0.34	0.14	0.11	0.098
100000	0.93	0.67	0.45	0.21	0.18	0.15
1000000	0.88	0.78	0.54	0.25	0.16	0.13
10000000	0.90	0.89	0.69	0.28	0.20	0.18

Figure 16: Left: OpenMP Efficiency; Right: TBB Efficiency

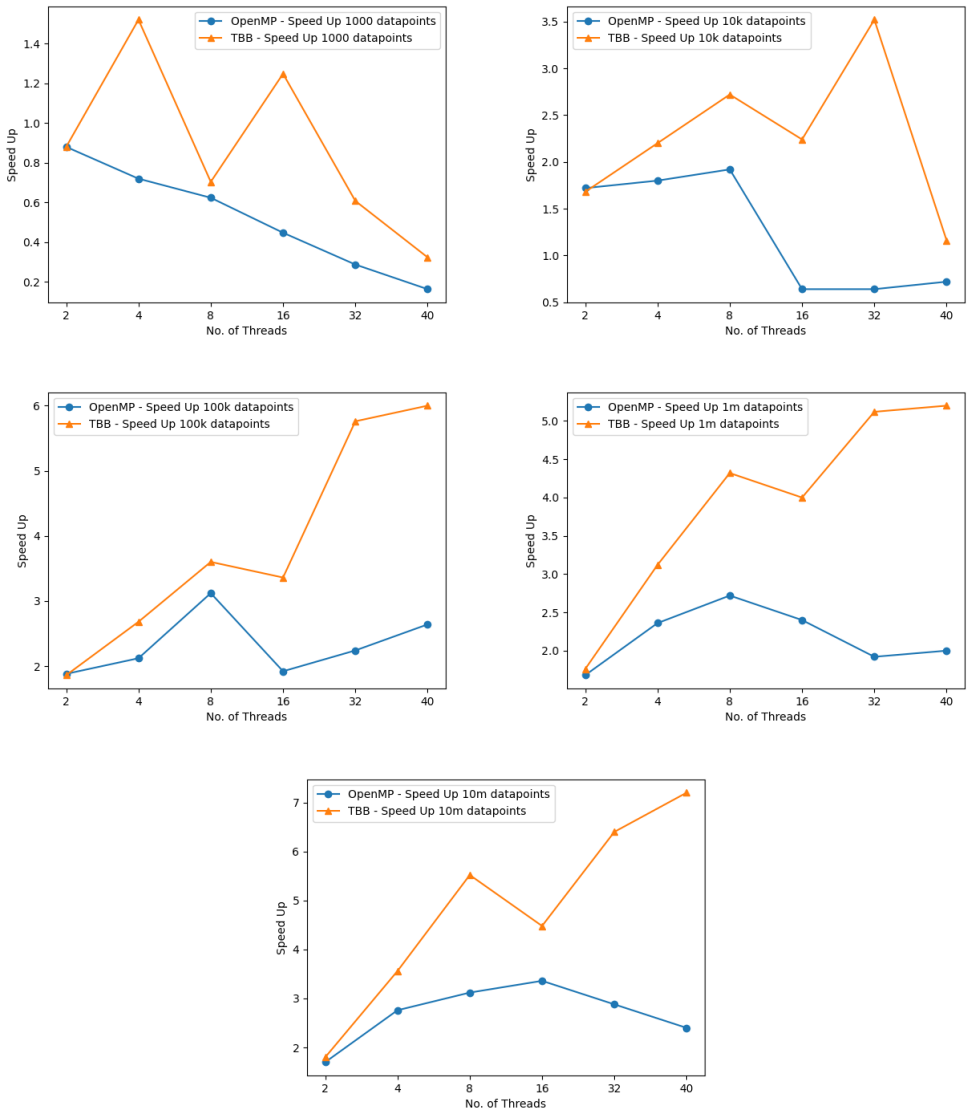

Figure 17: OpenMP vs TBB: Speed-up for different dataset sizes

Looking at the efficiency numbers of TBB and OpenMP, we can conclude that TBB on average is more scalable and efficient than OpenMP for MergeSort benchmark too. This seem to suggest that the task scheduler is more efficient than the tasking model of OpenMP.

In abridgement, from the two benchmarks, we can conclude that TBB scales better as the data-set size and number of thread increases.

6 Runtime overhead

Scheduling and synchronization can be sources of significant overhead in shared memory parallel programs. Overhead analysis is an extended view of Amdahl's Law. Assume T_s is the time spent by a serial implementation of a given algorithm and T_p is the time spent by a parallel implementation of the same algorithm on p threads. In such a case, for a perfect parallelisation we would have:

$$T_p = T_s/p$$

Amdahl's Law introduces α as a measure of the fraction of parallelised code in the parallel implementation, and states that:

$$T_p = (1-\alpha)T_s + \alpha * \frac{T_s}{no\ of\ threads}$$

Thus, the best time for a parallel implementation is restricted by the fraction $(1-\alpha)$ of unparallelised code. Let us rearrange equation (1) to give:

$$T_p = \frac{T_s}{no\ of\ threads} + \frac{no\ of\ threads - 1}{no\ of\ threads} * (1-\alpha)T_s$$

The first term is the time for an ideal parallel implementation. The second term can be considered as an overhead or degradation of the performance. In this case it is an overhead due to unparallelised code. However, this model is too simplistic in that it takes no account of any of the various factors affecting performance, such as how well the parallelised code has been implemented. Let us therefore consider equation (2) to be a specific form:

$$T_p = \frac{T_s}{no\ of\ threads} + \sum_i O_i$$

where each O_i is an overhead.

We are only interested in temporal overheads like how does the measured parallel time differ from the ideal parallel time. Work on spatial overheads like how do the memory requirements of the parallel implementation differ from those of the sequential implementation can be considered as an extension of the current work.

Hence, we define the overhead as:

$$Overhead = T_p - \frac{T_s}{no\ of\ threads}.$$

We use a the previous defined two array sum program to measure the runtime overhead. The parallel execution time is calculated with only the parallel directive timed.

```
start = omp_get_wtime();
    #pragma omp parallel for num_threads(no_threads)
    for(int i=0;i<N;i++){
        c[i] = a[i] + b[i];
    }
    end = omp_get_wtime();
```

Similarly for TBB:

```
tick_count t_start = tick_count::now();
task_scheduler_init init(no_threads);
parallel_for(blocked_range<int>(0, N),TwoArraySum(a,b,c));
tick_count t_end = tick_count::now();
```

We setup few experiments to compare the overheads that include thread creation, synchronization and deletion.

6.1 Creation and Deletion Overhead

We make comparisons based on the parallel loops doing no work at all to measure just the creation and deletion overheads for thread counts of 32, 64 and 128. We notice from figure 18 that the overhead of

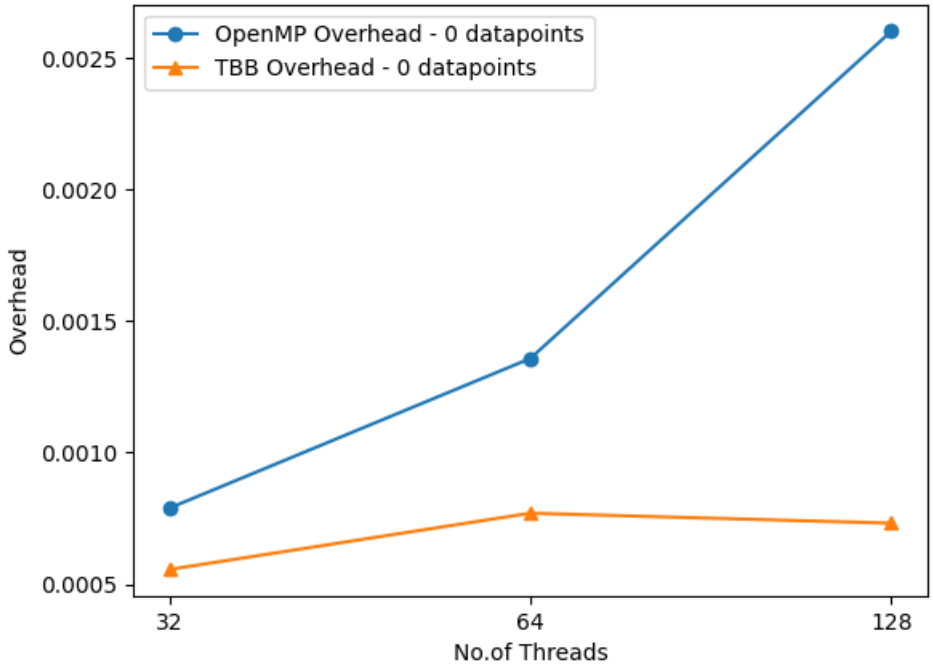

Figure 18: Overhead - OpenMP vs TBB: Creation and Deletion

creation and deletion increases for OpenMP while it almost remains constant for TBB as the number of threads increases. This is because the task scheduler in TBB initializes several logical threads in comparison to physical threads by OpenMP as discussed in section 5.2.

6.2 OpenMP Scheduling Overhead

We run experiments to compare various scheduling policies provided by OpenMP, such as static and dynamic scheduling, to observe the overheads caused by these strategies.

6.2.1 Dynamic vs static overheads

We measure the overhead with static and dynamic scheduling in OpenMP using the implementation of Two Sum Array. The algorithm has equal work per iteration. The overhead is defined as the difference in execution time of dynamic scheduling w.r.t to static scheduling. Referring to the figure 19, we come to a conclusion that the dynamic scheduling has more overhead compared to the static

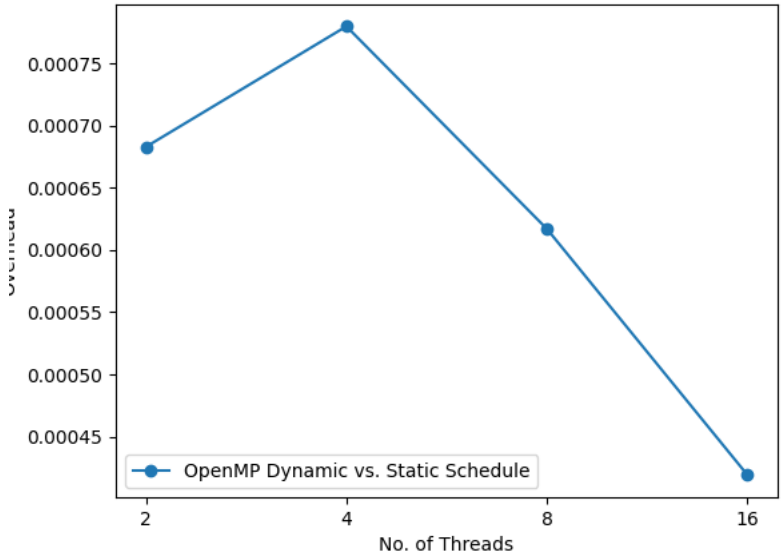

Figure 19: Execution times OpenMP: Static vs Dynamic Scheduling

scheduling. We also notice the a drop in the relative overhead between dynamic and static scheduling decreases with increase in number of threads which isn't significant but still shows an overhead between the two policies.

Dynamic scheduling is expensive as there is some communication between the threads after each iteration of the loop. It is appropriate when the iterations require different computational costs. This scheduling type has higher overhead then the static scheduling type because it dynamically distributes the iterations during the runtime unlike static at compile time.

These overheads could also be due to the wrong scheduling policy for an algorithm.

6.3 TBB Scheduling Overhead

We run experiments to measure scheduling overheads for TBB by varying grain sizes.

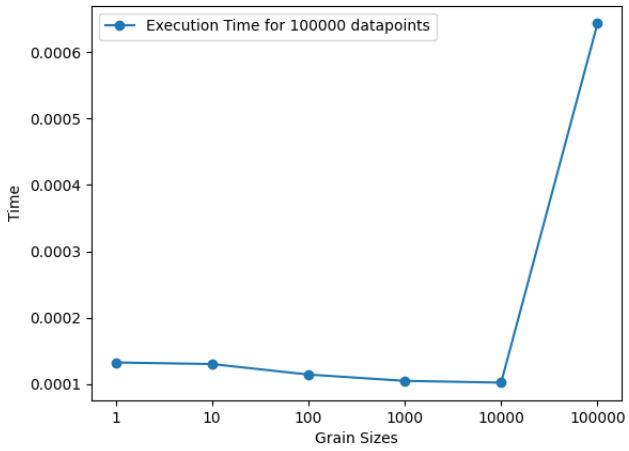

Figure 20: Execution times TBB: Grain Sizes

Referring to the figure 20, we notice that the scale is similar to logarithmic. The downward slope on the left side indicates that with a grainsize of one, most of the overhead is parallel scheduling overhead, not useful work. An increase in grainsize brings a proportional decrease in parallel overhead. Then the curve flattens out because the parallel overhead becomes insignificant for a sufficiently large grainsize. At the end on the right, the curve turns up because the chunks are so large that there are fewer chunks than available hardware threads.

6.4 Comparing overheads with respect to chunk sizes

We compare the execution times of OpenMp with static scheduling varying chunk sizes and TBB with varying grain sizes to show how chunk sizes can affect the parallel models.

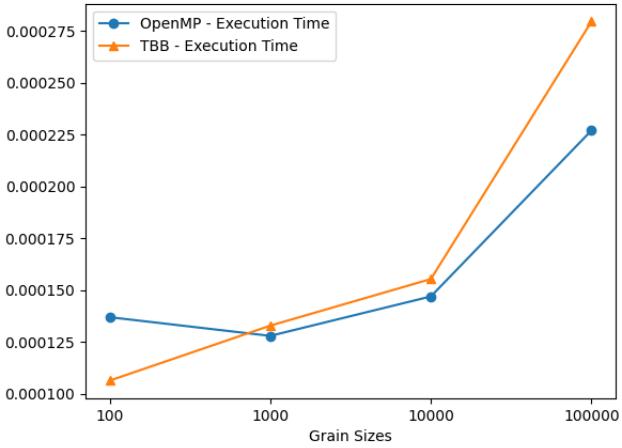

Figure 21: Execution times OpenMP vs TBB: Chunk Sizes

Referring to the figure 21, we notice that the execution time increase with the increase in chunk sizes for both the programming models. This is shown to keep in mind the overheads that could occur due to wring selection of chunk or grain sizes. In TBB smaller chunk sizes result in more parallel scheduling overhead rather than actual work and larger chunk sizes result in idle threads. OpenMP also gives rise to similar issues, although we deal with synchronization and thread communication issues rather than scheduling in TBB.

7 Programmers Control

The parallel constructs of TBB and OpenMP offer a comparable level of abstraction. In neither case we have to work with threads directly. TBB's parallel for template resembles the semantics of OpenMP's parallel loop construct. OpenMP's parallel loop construct can be configured by specifying a scheduling strategy and a block size value, whereas TBB relies solely on its task-scheduling mechanism. However, OpenMP's parallel loop construct is easier to use, because the schedule clause may be omitted.

7.1 Threads

TBB revolves around the concept of a task. A task is a unit of independent work. The programmer expresses problems as tasks. At what time the tasks are executed is decided by the run-time system. In TBB this behavior is realized through the task scheduler. By focusing on specifying tasks instead of managing threads the programmer is relieved of issues such as load balancing. Upon instantiation, the task scheduler initializes several logical threads. Note the distinction between a logical thread, specified with a threading API, and a physical thread executing on a processor. These logical threads are placed in a global thread pool. By utilizing a thread pool, the scheduler removes the overhead of

thread allocation and destruction between each parallel region. TBB parallel templates are optimized for tasks that are non-blocking and can be executed out of order. For blocking or dependent tasks the programmer can control the task scheduler directly.

OpenMP allows a section of code to be executed by several cooperating threads. This is done using the "Fork and Join" programming model. A master thread forks and creates several worker threads. These threads then performs a series of instructions in parallel which make up a parallel region. At the end of the parallel region the worker threads waits until all threads have finished and are then killed off. After this, sequential execution is continued by the master thread. The astute reader may notice that the only high-level difference from a thread pool approach is the termination of worker threads after the parallel region. All OpenMP C/C++ directives start with #pragma omp, followed by one or more clauses. The initialization and destruction of the worker threads is handled by the compiler. This high-level specification can turn a sequential program into one that utilizes multiple processors fairly easily. It also includes several helper functions. These functions let the programmer manage threads at run-time. The most useful of which allows querying a thread for its id and to get the total number of available threads.

7.2 Private/Shared Variable

A variable in an OpenMP parallel region can be either shared or private. If a variable is shared, then there exists one instance of this variable which is shared among all threads. If a variable is private, then each thread in a team of threads has its own local copy of the private variable. Why we need private and shared variables in OpenMP? Because the compiler cannot know our algorithm and what we really want. Therefore, we need to say the compiler what variables are local (private) for each iteration of the loop and what variables share their states between different loop iterations. However, we do not need such control in TBB. Why? Because the similar idea can be expressed with language constructions. Any variable passed to the body/lambda by reference(&) can be considered as "shared" because it is not copied and any its change can be visible in other iterations. However, if the programmer needs some local variables they can declare them inside the body/lambda. It should be noted that variables passed to the body/lambda by value(=) cannot be modified inside the loop in TBB.

7.3 Data Structures

The TBB provides various parallel data-structures such as Concurrent queue, Concurrent vector, and Concurrent hash, thus standing strong on "reduce the developer burden of managing complex and error-prone parallel data structure". OpenMP does not provide such data structures to it's users, hence, in order to not rely on shared data structures at all (because one will have to rely on locks at some point, which will slow things down), each thread would have to write to its own structure, and coalesce the results once the parallel section has completed.

7.4 Mutual Exclusion & Locking

Unlike OpenMP, TBB provides several mutex implementations that differ in properties like scalability, fairness, being re-entrant, or how threads are prevented from entering a critical section. Scalable mutexes do not perform worse than a serial execution, even under heavy contention. Fairness prevents threads from starvation and, for short waits, spinning is faster than sending waiting threads to sleep.

OpenMP, being designed to work both C/C++ as well as Fortran, do not offer the same level of abstraction as TBB does. When using locks, the programmer first has to manually initialize the locks by calling the omp_init_lock function. TBB on the other hand, implicitly initialize the locks through a default constructor. Also, OpenMP locks have to be manually acquired and released. TBB, being a C++ library, can make full use of the features offered by the language. An example of this is the scoped_lock, which makes use of a C++ feature known as Resource Acquisition Is Instantiation. This means that the lock is acquired at the time of declaration. When the code declaring the lock goes out of scope, the lock is automatically released through the lock objects destructor. This simplifies the code and removes the risk of introducing bugs by forgetting to release a lock. This could be implemented for OpenMP locks as well by wrapping the acquisition and releasing in a class, but is not provided by default.

7.5 Reduction

TBB offers more flexible reductions since it is invoked as a standard method. The OpenMP reduction clause only supports primitive types as reduction values and only primitive reduction operations when combining them. The behavior of the TBB reduction template can somewhat be mimicked in OpenMP. Instead of using the reduction clause, thread private copies of complex objects can be declared inside a parallel region. At the end of this region, the objects can then be joined in a critical section. This would however not offer the same performance as only a single thread could execute the critical section at a time.

Lastly, as mentioned before, unlike OpenMP, which provides multiple scheduling strategies, TBB always uses a fixed scheduling strategy that is affected by the selected grain size.

8 Conclusion

As mentioned several times earlier, OpenMP and TBB shows promise in different situations. Using OpenMP, we can parallelize existing sequential program with little or no program redesign. TBB, on the other hand, fosters a structured, object-oriented programming style, which comes along with the cost of re-writing parts of the program.

For basic loop parallelizations, OpenMP seems to have the edge over TBB because of its minimal syntax and intuitive semantics. This ease in expressing basic parallelism does however come at the cost of flexibility. Past a certain complexity threshold, expressing parallelism with OpenMP becomes awkward. This point has not been shown to a great extent in the parallel algorithms presented here. OpenMP has however, come along way with the introduction of the tasking model in version 3.0. For the next version of the specification, several interesting features have been proposed. These include better error handling, better interoperability with other threading packages, support for transactional memory and improvements to the tasking model. These features would go along way in bridging the gap between the two approaches. Additional features is not necessarily positive. By introducing more and more advanced constructs, the OpenMP specification does run the risk of loosing its focus on the areas where it shines.

TBB, being a library and consisting only of pure C++ code, makes it a potentially better fit in enterprise settings. In large projects spanning several years the ability to understand how different components of a system communicate becomes greater. The abstractions provided by TBB allows this to be expressed more clearly and at a higher level. Also, the building blocks provided by TBB makes it possible to express complex forms of parallelism. These advanced constructs have not been needed for the algorithms presented in this project, and as such any examples of these constructs would be contrived. The ability to express advanced forms of parallelism has increased further in recent months with the addition of a message passing system for shared memory systems to the library.

Performance wise, TBB showed a slight edge overall in the experiments conducted for this project, but not enough to conclude that it is superior to OpenMP.

To conclude this it might be useful to conduct experiments with OpenMP implementations in C, since that would likely result in improved performance. Overall, the choice of which approach to choose depends entirely in the problem at hand. Hopefully, this project has been able to give the reader an intuition to aid in this choice.

9 References

1. Assessing Modern Parallel Programming Languages
2. Productivity analysis of the UPC language
3. Intel Threading Building Blocks Outfitting C++ for Multi-Core Processor Parallelism
4. Parallel Programming Models and Paradigms: OpenMP Analysis
5. Measuring Syncronisation and Scheduling Overheads in OpenMP
6. Performance Comp i on with OpenMP Parallelization for Multi-core Systems
7. A comparative analysis between parallel models in C/C++ and C#/Java