A Comparative Analysis of Parallel Programming Models for C++

Arno Leist Andrew Gilman
Institute of Natural and Mathematical Sciences
Massey University
Auckland, New Zealand
{a.leist, a.gilman}@massey.ac.nz

***Abstract*—The parallel programming model used in a software development project can significantly affect the way concurrency is expressed in the code. It also comes with certain trade-offs with regard to ease of development, code readability, functionality, runtime overheads, and scalability. Especially the performance aspects can vary with the type of parallelism suited to the problem at hand. We investigate how well three popular multi-tasking frameworks for C++ — Threading Building Blocks, Cilk Plus, and OpenMP 4.0 — cope with three of the most common parallel scenarios: recursive divide-and-conquer algorithms; embarrassingly parallel loops; and loops that update shared variables. We implement merge sort, matrix multiplication, and dot product as test cases for the respective scenario in each of the programming models. We then go one step further and also apply the vectorisation support offered by Cilk Plus and OpenMP 4.0 to the data-parallel aspects of the loop-based algorithms. Our results demonstrate that certain configurations expose significant differences in the task creation and scheduling overheads among the tested frameworks. We also highlight the importance of testing how well an algorithm scales with the number of hardware threads available to the application.**

Keywords*—*parallel programming models; performance; TBB; Cilk Plus; OpenMP 4.0

I. INTRODUCTION

With the widespread availability of parallel processors, and the focus on even more parallel execution units in upcoming chips from all major manufacturers, the need for simple but efficient programming models specifically designed for parallel computing is becoming increasingly apparent.

For software developers, this means a rethinking of the way code is written. We can no longer expect that our applications will automatically run faster with every new processor generation, unless they are able to dynamically scale to large numbers of threads that can be assigned to processing units by a runtime scheduler in such a way that a good load balance is achieved. And all of this should be done with as little overhead as possible, both from the perspective of the programmer and from a performance perspective. While some overheads are algorithm specific, such as a need for thread local storage, others are inherent to the programming model and parallel runtime system. It is important to choose a parallel programming model that not only performs well on the current generation of widely deployed quad to hexa-core processors, but also on new generations of many-core processors.

We investigate how well three of the most popular task-based parallel programming models for C++ — Intel Threading Building Blocks (TBB) [1], Intel Cilk Plus [2] and OpenMP 4.0 [3] — perform when faced with three different but equally common parallel scenarios: divide-and-conquer style algorithms (merge sort), for-loops with no data contention (matrix multiplication) and loops performing a reduction operation (dot product). We want to determine whether the way concurrent tasks are spawned has a significant effect on the runtime schedulers. Secondly, we are also interested in the support for parallel reduction operations offered by the frameworks, as these are difficult to do in parallel while at the same time retaining good scalability to the large numbers of execution units found in many-core devices. Finally, we also look at the support for vectorisation offered by Cilk Plus and OpenMP 4.0, and how it affects the results of the loop-based algorithms.

While there are already a number of existing publications comparing some subset or superset of TBB, Cilk Plus and OpenMP, they tend to focus on a particular parallelisation strategy, such as subdividing the range of a loop or a recursive algorithm. Most of these articles also do not yet utilise the constructs for expressing opportunities for vectorisation to the compiler, which have been added in Cilk Plus and OpenMP 4.0. These extensions are becoming increasingly important, given the trend towards wider vector units in general purpose CPU cores, and the tendency towards heterogeneous architectures that tightly integrate data-parallel graphics processing units with the CPU to use them as compute accelerators [4]. The ability to easily and efficiently combine task and data-parallelism will be essential for good performance scaling on upcoming hardware.

The following section puts our work into context with existing publications. Section I-B then provides a brief introduction to each of the chosen programming models, which is followed by the implementation of the selected algorithms in Section II. Section III gives the results of the tests, comparing the performance of each of the parallel implementations with respect to a serial reference implementation. In Section IV, we discuss the results in more depth, before we draw some conclusions from the findings in Section V.

A. Related Work

Articles that focus on only one type of parallel problem are able to go into a significant level of detail with regard to implementation decisions, but do not demonstrate how well the programming models deal with a variety of use cases, all of which can easily occur in a single application. For example, matrix operations, in particular matrix multiplication, are a common example for the parallelisation of loop-constructs, as demonstrated in [5] for TBB, Cilk++ (the predecessor to Cilk Plus, developed by Cilk Arts before their acquisition by Intel), OpenMP, the Open Message Passing Interface (Open MPI) [6] and POSIX (Portable Operating System Interface) threads (Pthreads) [7]; or in [8] with TBB, Cilk++, OpenMP, Pthreads, Intel Array Building Blocks (ArBB) [9] and a number of less well known programming models.

The authors of [10], on the other hand, chose the well known Mandelbrot algorithm to compare TBB, OpenMP and Pthreads.

 ISBN: 978-1-61208-346-9

This is similar to matrix multiplication, in that the algorithm's outermost loop is straight forward to parallelise. While the authors briefly discuss an implementation that uses SSE vector intrinsics to further speed-up the computation, none of the chosen programming models offered the necessary support on its own at the time of publication.

Jarp et al. [11] compare the performance of an algorithm for high energy physics data analysis when implemented in TBB, OpenMP and Cilk Plus. Where possible, the code is vectorised using either Cilk Plus array notation or the auto-vectorisation capabilities of the Intel C++ compiler. While more complex than in the articles mentioned so far, the parallelisation is also based on executing loop iterations concurrently, with no interdependence between the iterations of the first few loops and the need for a reproducible reduction operation to sum the results of the final loop.

In [12], the authors compare the performance of irregular graph algorithms implemented using TBB, Cilk Plus, and OpenMP 3.0 on Intel's MIC architecture based "Knights Ferry" prototype, the predecessor of the first officially released generation of Intel Xeon Phi coprocessors. The algorithms implemented are the memory bandwidth intensive graph colouring and breadth-first search algorithms, as well as a synthetic algorithm that artificially increases computation over the same data. Again, loops over vertex and adjacency sets are used to expose parallelism to the programming models.

In [13], Krieger et al. provide a good discussion of asynchronous parallelism represented using task graphs. They compare implementations of a sparse matrix solver and a molecular dynamics benchmark using TBB, which offers explicit support for task graphs, to OpenMP and Cilk Plus. Neither of the latter two models supports the concept of task graphs out of the box. Instead, the authors map the concepts of loop-based and task-spawning based parallelism to task graphs.

Podobas et al. [14] focus on recursive algorithms and specifically the overhead of task creation, spawn and join operations when using Cilk++, several different implementations of OpenMP 3.0, as well as a multi-tasking library developed by one of the authors of the paper.

Although this is not an exhaustive overview of the literature, it illustrates how our article approaches the problem from a more general perspective than what has been done before.

B. The Programming Models

This section gives a brief introduction to the programming models. Specific features that are relevant to the article are discussed in more detail in the implementation section.

1) Threading Building Blocks (TBB): is a portable C++ template library for the development of concurrent software using task parallelism. A runtime scheduler is responsible for maintaining a thread pool and efficiently mapping tasks to worker threads for execution. In doing so, it abstracts the programmer from the platform-specific threading libraries, and also from the details of the hardware, such as the number of physical cores available to the application.

TBB offers a variety of constructs to express concurrency, ranging from simple parallel loops to flexible flow graphs. It also provides a scalable memory allocator and a number of concurrent data structures, such as a thread-safe vector, queue, and hash map implementation.

One of the advantages of TBB compared to Cilk Plus and OpenMP is that it does not require any special compiler support. Any reasonably modern C++ compiler is able to build a program that uses TBB.

2) Cilk Plus: introduces three new keywords to C/C++: `_Cilk_for`, `_Cilk_spawn`, and `_Cilk_sync`. In usage and throughout much of the official documentation, it is common to include the header `cilk/cilk.h`, which defines the alternative and more streamlined keywords `cilk_for`, `cilk_spawn`, and `cilk_sync` as synonyms for the language keywords. As the names suggest, the keywords are used to express concurrent code sections that can be executed in parallel by the runtime scheduler. Code sections that are bounded by these parallel control statements, or the start and end of the application, but that do not contain any such constructs, are called "strands" in Cilk terminology.

Cilk Plus also adds array notations, which provide a natural way to express data-parallelism for array sections. For example: $c[0 : n] = a[0 : n] + b[0 : n]$ performs an element by element addition of arrays a and b and writes the results to c. Furthermore, `#pragma simd` can be used to enforce loop vectorisation.

Cilk has to be explicitly supported by the compiler. At the time of writing, this includes stable support in the Intel icc/icpc compiler, and a `cilkplus` development branch for GCC 4.9. We would have liked to test the performance of the Cilk extensions in GCC, but the use of any SIMD functionality—array notations or `#pragma simd`—caused internal compiler errors when using the current development version of GCC 4.9 at the time of writing this article (January 2014).

3) OpenMP: consists of a number of compiler directives, introduced into the code using the `#pragma` keyword, runtime library routines, and environment variables. Concurrency is generally expressed with parallel regions that are denoted by a `#pragma omp parallel` directive. The master thread is forked at the beginning of each such region into a number of worker threads. At the end of each parallel region, the worker threads are joined back in after implicit synchronisation. Inside the regions, OpenMP provides an option to use a selection of work-sharing constructs that distribute the work amongst the worker threads.

II. Implementing the Algorithms

We choose three algorithms with very different parallel characteristics to compare how each of the programming models and associated runtime libraries handles the given challenges.

A. Merge Sort

The first is merge sort, representing the widely applicable category of recursive divide-and-conquer algorithms. The implementation can spawn recursive tasks during both the sorting and merge phases to increase the amount of work that can be done in parallel. This ensures that even in the last steps of the algorithm, at the top of the merge sort hierarchy, where the longest, sorted subsets of the input data are merged, plenty of concurrent work is available to keep all worker threads busy.

Algorithm 1 gives the pseudo-code for function `sort`. The implementation recursively subdivides the input range until the threshold of length ≤ 32 is reached. At this point, it switches to the insertion sort algorithm, which has a higher worst case computational complexity of $\mathcal{O}(n^2)$, but only a small constant overhead, which makes it more efficient for very short arrays.

As can be seen, the recursive calls on lines 8 and 9 operate on non-overlapping sections of the data array, [low, mid]

 ISBN: 978-1-61208-346-9

Algorithm 1 The implementation of the merge sort algorithm subdivides the input range until the threshold of length ≤ 32 is reached. It then switches to the insertion sort algorithm, which offers better performance for small inputs.

```
function sort(data, aux, low, high)
 1: if low < high then
 2:     length ← high + low + 1
 3:     if length > 32 then
 4:         mid ← (low + high)/2
 5:         //Note: This is where the parallel implementations
 6:         //check if (mid−lo) > cutoff_sort before they spawn
 7:         //the following calls to sort() as concurrent tasks.
 8:         call sort(data, aux, low, mid)
 9:         call sort(data, aux, mid + 1, hi)
10:         call merge(data, aux, low, mid, high)
11:     else
12:         call insertion_sort(data, low, high)
13:     end if
14: end if
```

and [mid+1, hi] respectively. Therefore, they can be safely executed in parallel. Depending on the programming model and implementation, either both calls can be spawned as child tasks while the parent task is suspended until they have completed their work, or only the first call gets spawned and the parent processes the second recursive call itself. The second version intuitively introduces a smaller overhead, as less tasks are spawned. In either case, the spawned tasks and their parent must be synchronised before `merge` is called, which combines the two independently sorted sub-ranges.

The serial implementation uses a straight forward merge function that expects the two ranges to be consecutive in memory. It copies the first half into an auxiliary data array before it combines the two ranges in the original data array. This is quite efficient, because memory is accessed sequentially, but it is not obvious how to parallelise this operation. Therefore, the recursive merge algorithm from [15] is implemented and used for all parallel implementations. The pseudo-code in Algorithm 2 illustrates the procedure.

The merge operation picks the median index of the first range, idx_1, and performs a binary search for the value found at this index in the second range. Assuming an ascending sort order, the search returns an index, idx_2, according to these rules:

- If the search range is empty, then it returns low_2
- If $\text{in}[\text{idx}_1] \leq \text{in}[\text{low}_2]$, then it returns low_2
- If $\text{in}[\text{idx}_1] > \text{in}[\text{low}_2]$, then it returns the *largest* index in the range $[\text{low}_2, \text{high}_2+1]$ such that $\text{in}[\text{idx}_2-1] < \text{in}[\text{idx}_1]$

Note that our implementation uses C++ templates to pass a function pointer or functor to the procedures. The operation defined by this function is used for all comparisons, and, therefore, defines how data of a given type is sorted.

On line 10, the algorithm copies the value from the median index to the output array, and then recursively calls itself twice. The first call passes the lower parts of the two input ranges to `merge_recursive`, while the second call passes the upper parts. Just like before, the recursive calls operate on non-overlapping data regions and can be safely performed in parallel. However, since this algorithm does not traverse memory sequentially and introduces computational overhead, it runs slower than the simple serial merge implementation. Therefore, once sufficient tasks have been spawned, it is beneficial to switch to a more efficient serial merge operation for the shorter data ranges on the lower levels of the hierarchy. A cutoff is defined (line 6) that causes the implementation to do exactly

Algorithm 2 The implementation of the recursive merge function is based on the algorithm described in [15]. It merges two sub-ranges of array `in`, $[\text{low}_1, \text{high}_1]$ and $[\text{low}_2, \text{high}_2]$, and writes the result to array `out` beginning at index low_{out}.

```
function merge_recursive(in, low_1, high_1, low_2, high_2,
                         out, low_out)
 1: length_1 ← high_1−low_1 + 1
 2: length_2 ← high_2−low_2 + 1
 3: if length_1 ≤ length_2 then
 4:     swap low_1 ↔ low_2, high_1 ↔ high_2, length_1 ↔ length_2
 5: end if
 6: if length_1 + length_2 > cutoff_merge then
 7:     idx_1 ← (low_1 + high_1)/2 //pick the median index
 8:     idx_2 ← call binary_search(in[idx_1], in, low_2, high_2)
 9:     idx_3 ← low_out + (idx_1 − low_1) + (idx_2 − low_2)
10:     out[idx_3] ← in[idx_1]
11:     call merge_recursive(in, low_1, idx_1 − 1, low_2, idx_2 − 1,
                             out, low_out)
12:     call merge_recursive(in, idx_1 + 1, high_1, idx_2, high_2,
                             out, idx_3 + 1)
13: else if length_1 > 0 then
14:     call merge(in, low_1, high_1, low_2, high_2, out, low_out)
15: end if
```

that. This third merge function works like the first one, except that the two input ranges do not have to be consecutive, as this is the case when switching from the recursive merge operation part way down the hierarchy.

Now that the merge sort algorithms have been described, we can have a look at the different ways used to spawn the recursive calls in the programming models.

TBB offers a number of ways to spawn concurrent tasks that call functions, two of which are well suited for this algorithm: manual task management and the template function `tbb::parallel_invoke`. The former, more flexible but also more complex option, is used in function `sort`. It defines two types derived from `tbb::task`, one to perform the recursive subdivision of the sort routine, the other to initiate the recursive merge operation. This code is quite verbose and is therefore not included here, but examples of the concept can be found in the reference documentation of the task scheduler [1] under continuation-passing style.

The implementation of `recursive_merge`, on the other hand, uses `tbb::parallel_invoke` with the C++11 lambda function syntax to spawn the recursive calls and block until they complete. The resulting code to implement lines 11 and 12 of Algorithm 2 is concise, as the following listing shows (with template arguments omitted for readability):

```
tbb::parallel_invoke(
    [&]{ merge_recursive(in, low1,  idx1-1,
         low2, idx2-1, out, lowOut); },
    [&]{ merge_recursive(in, idx1+1, high1,
         idx2, high2,  out, idx3+1); } );
```

The **Cilk Plus** code for spawning functions is even simpler, as the following listing demonstrates (with both template and function arguments omitted for brevity):

```
cilk_spawn merge_recursive(...);
merge_recursive(...);
cilk_sync;
```

The code essentially behaves like one would expect: the first call to `merge_recursive` is spawned as a separate task, whereas the second call is executed by the current thread, until the `cilk_sync` statement synchronises both execution paths. The actual implementation of Cilk Plus works slightly

 ISBN: 978-1-61208-346-9

differently, in that the worker thread that performs the spawn immediately executes the spawned task, while the so called *continuation* of the function, that is the statements following the spawn, can be stolen by an idle worker. The assumption is that, most of the time, other threads are busy with their own work, and the continuation is not stolen but simply executed by the initial worker once it has completed the spawned function call. This is more efficient, as it takes advantage of data locality in the processor caches.

In the **OpenMP** implementation, we utilise the tasking facility introduced in specification version 3.0. An explicit task can be created using the `task` directive, followed by a structured block. When this directive is encountered by a thread, the task may be executed right away, or placed into a pool from which all worker threads in the current team can take tasks to execute. The following extract from the `merge_recursive()` subroutine demonstrates the syntax simplicity:

```
#pragma omp task
    merge_recursive(...);
#pragma omp task
    merge_recursive(...);
#pragma omp taskwait
```

Two tasks at the next level of recursion will be created and the execution of the current task will be halted at the `taskwait` directive, until all tasks created at the current level of recursion have been completed.

B. Matrix Multiplication

The second algorithm in our comparison is matrix multiplication. Given an $n \times m$ matrix A and an $m \times p$ matrix B, the algorithm computes matrix $C = AB$ of size $n \times p$. The most straight forward implementation uses three nested loops, iterating over the n rows of matrix A, the p columns of matrix B, and, on the innermost level, over the m columns of A while multiplying the values with the corresponding cells from B.

A simple but significant optimisation is to swap the two inner loops, which allows the data for matrix B to be read in row-major order and, thus, from sequential memory addresses. This has a big impact on performance. The pseudo-code for this implementation is given in Algorithm 3. While there are even more efficient serial algorithms for matrix multiplication, this implementation is straight forward to parallelise, because it does not have any data races across the rows of matrix A, given the precondition that the memory pointed to by C does not overlap with A or B. As such, it is used to represent the category of algorithms that can be parallelised by executing independent loop iterations.

The concurrent implementations simply subdivide the iteration space of the outermost loop into chunks that can be executed in parallel. The concurrency is limited to at most n chunks of one iteration each. If we do not swap the inner two loops, then the outer two loops can be collapsed into one loop of length np, but the improved memory access pattern of the optimised code easily makes up for the more limited concurrency, as even the serial version of this implementation runs faster than the multi-threaded code for the basic approach.

To further optimise memory access, each row of the matrix arrays is padded such that the memory address of the first element is aligned to a 64-byte boundary. This means that the data in a row is *memory aligned* for vector instructions of up to 512-bit width. It also coincides with the common cache line size on x86 processors. This is relevant, because the algorithm

Algorithm 3 The algorithm for matrix multiplication.

```
1: for row_a ← 0 to n − 1 do
2:     for col_b ← 0 to p − 1 do
3:         C[row_a][col_b] ← 0 //initialise row_a in C
4:     end for
5:     for col_a ← 0 to m − 1 do
6:         for col_b ← 0 to p − 1 do
7:             C[row_a][col_b] ← C[row_a][col_b] +
                                 A[row_a][col_a] × B[col_a][col_b]
8:         end for
9:     end for
10: end for
```

lends itself to vectorisation of the innermost loop. The padding, along with the compiler specific alignment guarantee given by `__assume_aligned`, in theory enables the compiler to use the more efficient vector instructions for aligned memory access. In theory, because Intel's compiler refuses to do so for unknown reasons during our tests when targeting AVX, where it uses the unaligned instructions instead, even though it complies and uses aligned instructions when targeting SSE.

The programming model specific changes to the code that allow the iteration range of the loop on line 1 of Algorithm 3 to be processed in parallel, as well as the changes to vectorise the loop on line 6 where applicable, are explained below.

TBB offers the convenient `tbb::parallel_for` function, which uses a partitioner to recursively split the specified range into smaller chunks until a certain condition is fulfilled. While different partitioning strategies can be implemented, the default `tbb::auto_partitioner`, which attempts to perform sufficient splitting to balance load, generally performs quite well. This partitioner is used here. The following code listing shows the TBB implementation.

```
tbb::parallel_for(0,n,[&](const size_t rowA) {
    // loop body
} );
```

The vectorisation of the innermost loop uses the Cilk Plus constructs as explained below.

The **Cilk Plus** keyword `cilk_for` is the obvious choice to parallelise the for-loop when using this programming model. The compiler converts the loop body into a function that is called recursively using a divide-and-conquer strategy, thus turning it into a directed acyclic graph of strands that each execute a chunk of up to *grain size* consecutive iterations. The grain size can be defined with the following pragma placed just before the loop: `#pragma cilk grainsize = 10`, but just like discussed for TBB, there are advantages to leaving it up to the runtime to decide what the grain size should be, especially since the pragma is a compile time construct, where the number of worker threads is usually not yet known. The signature of the `cilk_for` loop used in our example is:

```
cilk_for (size_t rowA = 0; rowA < n; ++rowA)
```

Once again, the change to the serial implementation is minimal. In fact, it is sufficient to change the definition of `cilk_for` from `_Cilk_for` to `for` to turn it into a regular for-loop, which can be very useful when debugging a concurrent program.

The second aspect we want to parallelise is the innermost loop on lines 6 to 8 using the Cilk Plus array notation:

```
cRowPtr[0:p] += aRowPtr[aCol] * bRowPtr[0:p];
```

A scalar value from A is multiplied with every element from an entire row of B, and the resulting vector is added to the

matching elements in the current row of C. It should be noted that the second argument in the array notation does not specify the last index in the range, but rather the length of the range.

The **OpenMP** loop work-sharing construct allows for simple parallelisation of the serial matrix multiplication code. Adding the `#pragma omp for` directive immediately before the outer loop indicates to the compiler that each loop iteration can be executed concurrently. The iteration range will be subdivided into equal chunks and executed concurrently by the threads in the current team.

Vectorisation of the inner loop can be achieved with the `simd` directive, introduced in specification version 4.0, by inserting `#pragma omp simd aligned(aRowPtr,bRowPtr:64)` directly before the inner loop. The aligned clause indicates to the compiler that addresses pointed to by `aRowPtr` and `bRowPtr` are aligned to 64-byte boundaries.

C. Dot Product

The dot product of two vectors a and b is defined as $a \cdot b = \sum_{i=1}^{n} a_i b_i$. The difficulty in parallelising this simple algorithm lies in the sum operation, which updates a shared variable across concurrent loop iterations, introducing data races unless specific precautions are taken. This is a very common problem in concurrent programming.

A common approach is to use some form of thread local storage to accumulate the values of the shared variable across all iterations of the loop executed by a given worker thread, before a reduction strategy is used to safely merge—or reduce—these intermediate results into the final value. The three frameworks used here all offer support for reduction operations, which are discussed in the following paragraphs.

TBB provides the function `parallel_reduce`, which works similar to `parallel_for`. The form we use here expects the following arguments: `(range, identity, func, reduction)`. This shows that, instead of passing the first and last indices of the iteration range directly to the loop construct, as we did in the matrix multiplication code, `parallel_reduce` expects an object that models TBB's range concept as its first argument. This is more flexible and also an alternative option for `parallel_for`. Most commonly, the predefined `tbb::blocked_range` is used, which takes the begin and end values for the range as its arguments, as well as an optional grain size.

The second argument is the left identity value used for the reduction. This is followed by a functor (function object) that implements the body of the loop, taking a reference to the range type and an initialisation value for the reduction variable as its arguments. The last argument is another functor that is called whenever two intermediate values from different tasks need to be joined. This functor accepts two references of the type of the reduction variable, and it returns the merged value, which, in this example, is the sum of the arguments. Conveniently, the C++ standard template library defines the binary functor type `std::plus` in header `functional`, which does exactly what is needed for the join operation.

The following code listing shows the implementation of the reducer in TBB, where `T` is the type of the reduction variable:

```
const T total = tbb::parallel_reduce(
  tbb::blocked_range<size_t>(0, length),
  T(), /* the left identity value */
  [=](const tbb::blocked_range<size_t> &range,
      T init)
  {
    for (size_t i = range.begin();
         i < range.end(); ++i)
    {
      init += vector1[i] * vector2[i];
    }
    return init;
  }, std::plus<T>() /* the join operation */
);
```

Cilk Plus reducers are based on the concept of so called hyperobjects [16], which return a view of the given type when they are dereferenced. If the strand that is doing the dereferencing was stolen (i.e., it is being executed by a different worker thread than its parent), then the hyperobject returns a new instance of the view, otherwise it can safely return the same instance as before. This minimises the overhead of creating view instances. When a spawned strand finishes and merges back into its parent, the reduction operation is invoked to reduce the values of the two view instances, leaving the result in the surviving strand's view. At the end of the concurrent section, all strands have been merged back into the leftmost view, which now contains the final value. The following listing illustrates how this works in practice.

```
cilk::reducer< cilk::op_add<T> > sum{ T() };
cilk_for (size_t i = 0; i < length; ++i) {
    *sum += vector1[i] * vector2[i];
}
const T total = sum.get_value();
```

The type of the reduction operation is specified as a C++ template argument (the C syntax for reducers depends on macros and typedefs instead) to the `cilk::reducer` type, which, when dereferenced, returns a reference to a value of type `T` that is guaranteed not to be in use by another worker. The reduction operation must be associative, but the order of the operands is the same as in the serial computation.

OpenMP provides a facility for the reduction operation through an additional `reduction` clause attached to the loop work-sharing construct. The following listing demonstrates the ease of expressing the concurrent loop for the dot product operation using OpenMP constructs, including reduction on variable `sum`:

```
#pragma omp parallel for reduction(+:sum)
for (size_t i = 0; i < length; ++i) {
    sum += vector1[i] * vector2[i];
}
```

III. Performance Results

The test system runs Ubuntu 13.10 on a 3.4 GHz quad-core Intel Core i7-3770 and 16 GBytes of PC-1600 system memory. The processor supports 2-way simultaneous multithreading (SMT), and is, therefore, seen as having eight logical cores by the system and applications. We use the Intel C++ compiler version 14.0.1, as this is the only compiler that offers stable support for all three programming models at the time of writing.

All tests are repeated 30 times and the fastest result is selected for each, as this most accurately reflects the actual time taken by the test case, with the least interruptions from system processes that are running in the background. Non-essential processes, such as the window manager, were stopped for the test runs.

We first test the merge sort algorithm, which uses two cutoff values to decide when to stop spawning concurrent tasks as

 ISBN: 978-1-61208-346-9

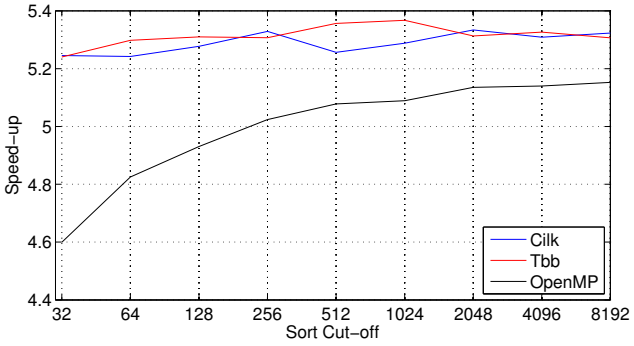

Fig. 1. Scaling the parallel sort cutoff value from 32 to 8192.

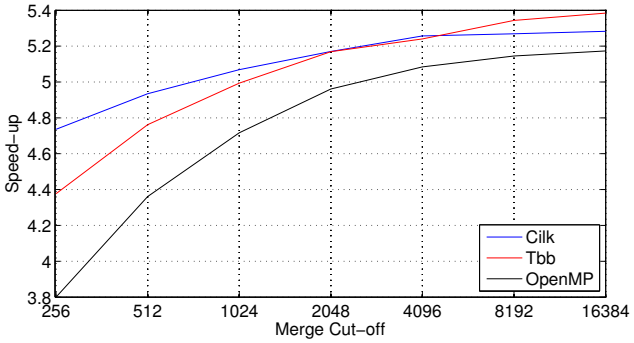

Fig. 2. Scaling the parallel merge cutoff value from 256 to 16384.

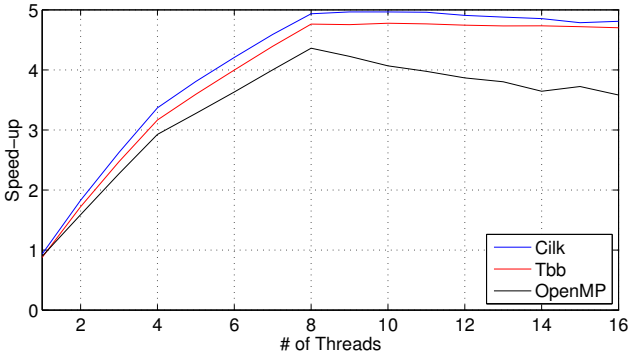

Fig. 3. The merge sort algorithm running with 1–16 worker threads. The parallel cutoffs are set to 512 and 8192 for sorting and merging respectively.

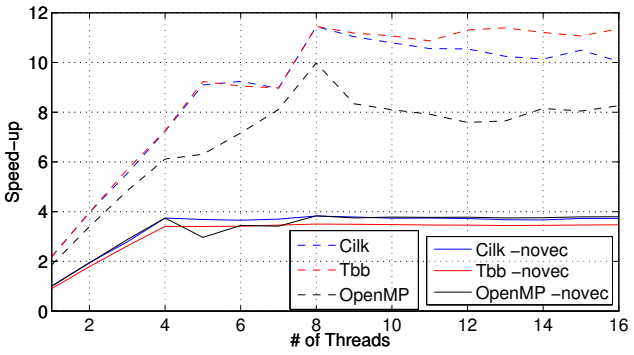

Fig. 4. Matrix multiplication with 1–16 worker threads.

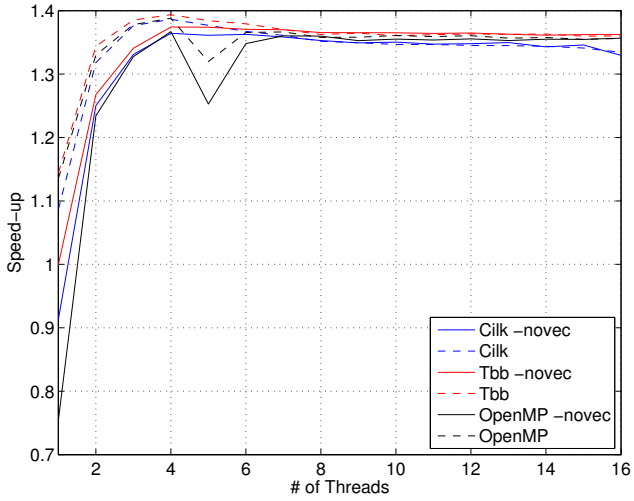

Fig. 5. Dot product with 1–16 worker threads.

indicated by $\texttt{cutoff}_{\texttt{sort}}$ on line 5 of Algorithm 1 and by $\texttt{cutoff}_{\texttt{merge}}$ on line 6 of Algorithm 2. Figure 1 gives the results for $\texttt{cutoff}_{\texttt{sort}}$ values from 32 to 8192, with the merge cutoff set to a constant 8192; and Figure 2 gives the results for $\texttt{cutoff}_{\texttt{merge}}$ values from 256 to 16384, this time with the sort cutoff set to a constant 64 for Cilk Plus, 512 for TBB, and 4096 for OpenMP. These numbers for the sort cutoff were chosen as they are the smallest values that, on average, approach the best performance achieved with the respective framework. It is important to set the cutoffs as low as possible without causing too much overhead, as they limit the amount of concurrency available at runtime.

Now that the effects of the cutoffs are established, we look at how the number of worker threads available to the process affects the performance. Figure 3 shows the results for merge sort. The data array to be sorted contains 50×10^6 integers. This test is also performed for the matrix multiplication and dot product, with the results given in Figures 4 and 5, respectively. The tests for matrix multiplication use two integer matrices of size 2500×2500 each, and the dot product implementations are tested with two vectors of 10^9 integers as input. These latter two plots also differentiate between code compiled with and without the `-novec` flag, which disables vectorisation when set.

IV. Discussion

The results show that all three programming models discussed here can perform well in a wide range of common parallel scenarios, but that there are also some caveats to be considered. The OpenMP implementation in the Intel compiler tends to be a little slower than TBB and Cilk Plus. It is also more susceptible to a loss of performance than the other frameworks in four different ways.

Firstly, the results for the scaling of parallel cutoff values in the merge sort implementation (Figures 1 and 2) show that the task creation and scheduling overheads introduced by a very large number of small tasks affect OpenMP more severely than they affect Cilk Plus or TBB.

Secondly, Figure 3 clearly illustrates that the OpenMP performance suffers significantly from oversubscription of the processor when more than eight threads (i.e., the number of

 ISBN: 978-1-61208-346-9

logical cores) are used in the divide-and-conquer algorithm. Interestingly, the same can be seen for matrix multiplication with vectorisation enabled, but not without vectorisation. Cilk Plus also takes a slight dip in these situations, whereas TBB remains unperturbed. Only for the dot product, where performance is clearly limited by the memory bandwidth, does TBB also take a small performance hit once the thread count exceeds four (i.e., the number of physical cores).

Thirdly, the static division of the problem space for parallel for-loops only works well when all cores of the processor are fully dedicated to the current task, the number of worker threads is equal to the number of logical cores, and each loop-iteration consists of the exact same set of instructions, that is, the amount of work to be done in every iteration is the same. If the operating system, or indeed a concurrent part of the application itself, schedules some other work on one of the cores, then the static work division can unnecessarily delay the completion of the loop, potentially letting some cores go idle, as the remaining threads are not able to take some of the load from the busy core and distribute it among the idle cores. The effects of load balancing issues are evident in the significant dip of the OpenMP performance in Figures 4 and 5 with five threads, and to a lesser extent six and seven threads, only that in our tests it is caused by a number of worker threads that is not a multiple of the core count. Static work load balancing with its minimal scheduling overhead certainly has a place in the repertoire of a parallel framework, but a dynamic scheduling algorithm would appear to be a more robust default choice. It should be noted that OpenMP does offer dynamic scheduling policies as options.

The final aspect we would like to highlight with regard to OpenMP is the lower SIMD performance when using the respective pragma for matrix multiplication (Figure 4). However, it is unclear why it is slower than the other implementations or auto-vectorisation, and we intend to investigate this further by comparing the machine instructions generated by the compiler for the different implementations.

The performance of TBB is impressive given that it does not have the advantage of being integrated into the language, which means that certain optimisations that can only be done when the compiler is aware of the parallel constructs are not open to it. The drawback of TBB is that it is somewhat more verbose than the other approaches, which makes the code more difficult to read. The introduction of lambda functions into the C++11 standard has helped tremendously in this regard, allowing for much more streamlined constructs than before, but TBB still does not reach the level of integration of Cilk Plus or OpenMP. Especially the former integrates beautifully into the language, to the point where it almost becomes difficult to spot the parallel sections. However, this minimalistic feel in Cilk Plus is also partly due to the smaller number of configuration options, which may be seen as a drawback when one needs more control. OpenMP pragmas are plentiful and concise, allowing for many optional parameters. OpenMP 4.0 even adds support for a number of additional modes of operation, such as off-loading of parallel sections to an accelerator, that are not covered by either of the other frameworks.

While TBB does not include support for vectorisation, it does not hinder it being added to the algorithm by other means either, as demonstrated by the use of Cilk Plus array notation in the vectorised TBB results. These array notations are another example of Cilk's seamless integration into the language, as the resulting code is both concise and easy to understand.

V. Conclusion

To conclude, we would like to emphasise several important characteristics of parallel programming models: ease of development, code readability, functionality, runtime overheads, and scalability. The last point is significant because of the trend towards integrating ever higher numbers of parallel execution units into new processor architectures.

In the future, we would like to compare the performance of Cilk Plus and OpenMP with their respective implementations in GCC, and potentially other compilers as well, to determine how consistent the results given here are across different implementations. The performance of TBB is expected to offer less surprises between compilers, as it is a template library rather than a language extension.

We would also like to run the same tests on the Intel Xeon Phi, Intel's many-core architecture, to find out how well the frameworks scale to the significantly larger number of cores. This would give a good indication of how future-proof the parallel programming models are.

References

[1] Intel® Corporation, "Threading Building Blocks 4.2 update 2," https://www.threadingbuildingblocks.org/, December 2013, retrieved: April, 2014.

[2] ——, "Cilk™ Plus," https://www.cilkplus.org/, retrieved: April, 2014.

[3] OpenMP Architecture Review Board, "OpenMP 4.0," http://openmp.org/, July 2013, retrieved: April, 2014.

[4] A. Leist, D. P. Playne, and K. A. Hawick, "Exploiting Graphical Processing Units for Data-Parallel Scientific Applications," *Concurrency and Computation: Practice and Experience*, vol. 21, no. 18, pp. 2400–2437, December 2009.

[5] E. Ajkunic, H. Fatkic, E. Omerovic, K. Talic, and N. Nosovic, "A Comparison of Five Parallel Programming Models for C++," in *MIPRO, 2012: Proceedings of the 35th International Convention*, P. Biljanovic, Z. Butkovic, K. Skala, S. Golubic, N. Bogunovic, S. Ribaric, M. Cicin-Sain, D. Cisic, Z. Hutinski, M. Baranovic, M. Mauher, and J. Ulemek, Eds., Opatija, Croatia, May 2012, pp. 1780–1784.

[6] The Open MPI Project, "A High Performance Message Passing Library," http://www.open-mpi.org/, retrieved: April, 2014.

[7] *IEEE Standard 1003.1c-1995: Threads Extension*, IEEE, 1995.

[8] P. Michailidis and K. Margaritis, "Computational Comparison of Some Multi-core Programming Tools for Basic Matrix Computations," in *Proceedings of the 14th IEEE International Conference on High Performance Computing and Communications*, G. Min, L. Lefevre, J. Hu, L. Liu, L. T. Yang, and S. Seelam, Eds., Liverpool, UK, June 25-27 2012, pp. 143–150.

[9] Intel® Corporation, "Array Building Blocks," http://software.intel.com/en-us/articles/intel-array-building-blocks, retrieved: April, 2014.

[10] W. Tristram and K. Bradshaw, "Investigating the Performance and Code Characteristics of Three Parallel Programming Models for C++," in *Southern Africa Telecommunication Networks and Applications Conference (SATNAC)*, Stellenbosch, South Africa, September 2010.

[11] S. Jarp, A. Lazzaro, A. Nowak, and L. Valsan, "Comparison of Software Technologies for Vectorization and Parallelization," CERN openlab, Tech. Rep., September 2012.

[12] E. Saule and U. Catalyurek, "An Early Evaluation of the Scalability of Graph Algorithms on the Intel MIC Architecture," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, Shanghai, May 2012, pp. 1629–1639.

[13] C. Krieger, M. Strout, J. Roelofs, and A. Bajwa, "Executing Optimized Irregular Applications Using Task Graphs within Existing Parallel Models," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion*, Salt Lake City, Utah, US, November 2012, pp. 261–268.

[14] A. Podobas, M. Brorsson, and K.-F. Faxn, "A Comparison of some recent Task-based Parallel Programming Models," in *Proceedings of the 3rd Workshop on Programmability Issues for Multi-Core Computers*, Pisa, Italy, January 2010.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction To Algorithms*, 3rd ed. MIT Press, 2009.

[16] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin, "Reducers and Other Cilk++ Hyperobjects," in *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*. Calgary, AB, Canada: ACM, 2009, pp. 79–90.

 ISBN: 978-1-61208-346-9