CHAPTER 16

Tuning TBB Algorithms: Granularity, Locality, Parallelism, and Determinism

In Chapter 2, we described the generic parallel algorithms provided by the TBB library and gave a few examples to show how they can be used. While doing so, we noted that the default behavior of the algorithms was often good enough but claimed that there were ways to tune performance if needed. In this chapter, we back up that claim by revisiting some TBB algorithms and talk about important features that can be used to change their default behaviors.

There are three concerns that will dominate our discussions. The first is granularity – the amount of work that a task does. The TBB library is efficient at scheduling tasks, but we need to think about the size of the tasks that our algorithms will create since task size can have a significant impact on performance, especially if the tasks are extremely small or extremely large. The second issue is data locality. As discussed in detail in the Preface, how an application uses caches and memory can make or break an application's performance. And the final issue is available parallelism. Our goal when using TBB is to introduce parallelism of course, but we cannot do it blindly without considering granularity and locality. Tuning an application's performance is often an exercise in balancing the trade-offs between these three concerns.

© Intel Corporation 2019
M. Voss, R. Asenjo, J. Reinders, *Pro TBB*, https://doi.org/10.1007/978-1-4842-4398-5_16

One of the key differences between the TBB algorithms and other interfaces like Parallel STL is that the TBB algorithms provide hooks and features that let us influence their behavior around these three concerns. The TBB algorithms are not just black boxes over which we have no control!

In this chapter, we will first discuss task granularity and arrive at a rule of thumb about how big is big enough when it comes to task size. We will then focus on the simple loop algorithms and how to use Ranges and Partitioners to control task granularity and data locality. We also have a brief discussion about determinism and its impact on flexibility when tuning for performance. We conclude the chapter by turning our attention to the TBB pipeline algorithm and discuss how its features affect granularity, data locality, and maximum parallelism.

Task Granularity: How Big Is Big Enough?

To let the TBB library have maximum flexibility in balancing the load across threads, we want to divide the work done by an algorithm into as many pieces as possible. At the same time, to minimize the overheads of work stealing and task scheduling, we want to create tasks that are as large as possible. Since these forces oppose each other, the best performance for an algorithm is found somewhere in the middle.

To complicate matters, the exact best task size varies by platform and application, and therefore there is no exact guideline that applies universally. Still, it is useful to have a ballpark number that we can use as a crude guideline. With these caveats in mind, we therefore offer the following rule of thumb:

RULE OF THUMB TBB tasks should be on average greater than 1 microsecond to effectively hide the overheads of work stealing. This translates to several thousand CPU cycles – if you prefer using cycles, we suggest a 10,000 cycle rule of thumb.

It’s important to keep in mind that not every task needs to be greater than 1 microsecond – in fact, that’s often not possible. In divide and conquer algorithms for example, we might use small tasks to divide up the work and then use larger tasks at the leaves. This is how the TBB `parallel_for` algorithms works. TBB tasks are used to both

split the range and to apply the body to the final subranges. The split tasks typically do very little work, while the loop body tasks are much larger. In this case, we can't make all of the tasks larger than 1 microsecond, but we can aim to make the average of the task sizes larger than 1 microsecond.

When we use algorithms like `parallel_invoke` or use TBB tasks directly, we are in complete control of the size of our tasks. For example, in Chapter 2, we implemented a parallel version of quicksort using a `parallel_invoke` and directed the recursive parallel implementation to a serial implementation once the array size (and therefore task execution time) fell below a cutoff threshold:

```
if (end - begin < cutoff) {
  serialQuicksort(begin, end);
}
```

When we use simple loop algorithms, like `parallel_for`, `parallel_reduce`, and `parallel_scan`, their range and partitioner arguments provide us with the control we need. We talk about these in more detail in the next section.

Choosing Ranges and Partitioners for Loops

As introduced in Chapter 2, a Range represents a recursively divisible set of values – typically a loop's iteration space. We use Ranges with the simple loop algorithms: `parallel_for`, `parallel_reduce`, `parallel_deterministic_reduce`, and `parallel_scan`. A TBB algorithm partitions its range and applies the algorithm's body object(s) to these subranges using TBB tasks. Combined with Partitioners, Ranges provide a simple, but powerful way to represent iterations spaces and control how they should be partitioned into tasks and assigned to worker threads. This partitioning can be used to tune task granularity and data locality.

To be a Range, a class must model the Range Concept shown in Figure 16-1. A Range can be copied, can be split using a *splitting constructor*, and may optionally provide a *proportional splitting constructor*. It also must provide methods to check if it is empty or divisible and provide a boolean constant that is true if it defines the proportional splitting constructor.

Pseudo-Signature	Semantics
R::R(const R&)	Copy constructor.
R::~R()	Destructor.
bool R::empty() const	True if range is empty.
bool R::is_divisible() const	True if range can be partitioned into two subranges.
R::R(R& r, split)	Basic splitting constructor. Splits r into two subranges.
R::R(R& r, proportional_split proportion)	**Optional**. Proportional splitting constructor. Splits r into two subranges in accordance with proportion.
static const bool R::is_splittable_in_proportion	**Optional**. If true, the proportional splitting constructor is defined for the range and may be used by parallel algorithms.

Figure 16-1. *The Range concept*

While we can define our own Range types, the TBB library provides the blocked ranges shown in Figure 16-2, which will cover most situations. For example, we can represent the iteration space of the following nested loop with a `blocked_range2d<int, int> r(i_begin, i_end, j_begin, j_end );`:

```
for (int i = i_begin; i < i_end, ++i )
  for (int j = j_begin; j < j_end; ++j )
    /* loop body */
```

Range Type	Constructor Arguments	Description
blocked_range	Value begin, Value end, [size_t grainsize]	Models a one-dimensional range.
blocked_range2d	RowValue row_begin, RowValue row_end, [size_type row_grainsize], ColValue col_begin, ColValue col_end, [size_type col_grainsize]	Models a two-dimensional range. After repeated splitting, the subranges approach the aspect ratio of the respective row and column grain sizes.
blocked_range3d	PageValue page_begin, PageValue page_end, [size_type page_grainsize], RowValue row_begin, RowValue row_end, [size_type row_grainsize], ColValue col_begin, ColValue col_end, [size_type col_grainsize]	Models a three-dimensional range. After repeated splitting, the subranges approach the aspect ratio of the respective page, row and column grain sizes.
blocked_rangeNd	const blocked_range<Value>& dim_0, ..., const blocked_range<Value>& dim_{N-1}	A preview feature that models an N-dimensional range. Requires C++11 support. After repeated splitting, the subranges approach the aspect ratio of the N blocked_range grain sizes.

Figure 16-2. *The blocked ranges provided by the TBB library*

For interested readers, we describe how to define a custom range type in the "Deep in the Weeds" section at the end of this chapter.

An Overview of Partitioners

Along with Ranges, TBB algorithms support Partitioners that specify how an algorithm should partition its Range. The different Partitioner types are shown in Figure 16-3.

Partitioner	Description	When Used with blocked_range(i,j,*g*)
`simple_partitioner`	Chunksize bounded by grain size.	$g/2 \le chunksize \le g$
`auto_partitioner` (default)	Automatic chunk size.	$g/2 \le chunksize$
`affinity_partitioner`	Automatic chunk size, cache affinity and initial uniform distribution of iterations.	
`static_partitioner`	Deterministic chunk size, cache affinity and uniform distribution of iterations without load balancing. A uniform distribution is created unless the grainsize prevents P chunks from being created.	$\max(g/3, N/P) \le chunksize$ *where:* *N is problem size* *P is number of resources*

Figure 16-3. *The partitioners provided by the TBB library*

A `simple_partitioner` is used to recursively divide a Range until its `is_divisible` method returns false. For the blocked range types, this means the range will be divided until its size is less than or equal to its grainsize. If we have highly tuned our grainsize (and we will talk about this in the next section), we want to use a `simple_partitioner` since it ensures that the final subranges respect the provided grainsizes.

An `auto_partitioner` uses a dynamic algorithm to sufficiently split a range to balance load, but it does not necessarily divide a range as finely as `is_divisible` allows. When used with the blocked range classes, the grainsize still provides a lower bound on the size of the final chunks but is much less important since the `auto_partitioner` can decide to use larger grainsizes. It is therefore commonly acceptable to use a grainsize of 1 and just let the `auto_partitioner` determine the best grainsize. In TBB 2019, the default Partitioner type used for `parallel_for`, `parallel_reduce`, and `parallel_scan` is an `auto_partitioner` with a grainsize of 1.

A `static_partitioner` distributes the range over the worker threads as uniformly as possible without the possibility for further load balancing. The work distribution and mapping to threads is deterministic and only depends on the number of iterations, the grainsize, and the number of threads. The `static_partitioner` has the lowest overhead of all partitioners, since it makes no dynamic decisions. Using a `static_partitioner` can also result in improved cache behavior since the scheduling pattern will be

repeated across executions of the same loop. A `static_partitioner` however severely restricts load balancing so it needs to be used judiciously. In section "Using a `static_partitioner`," we will highlight the strengths and weaknesses of `static_partitioner`.

The `affinity_partitioner` combines the best from `auto_partitioner` and `static_partitioner` and improves cache affinity if the same partitioner object is reused when a loop is re-executed over the same data set. The `affinity_partitioner`, like `static_partitioner`, initially creates a uniform distribution but allows for additional load balancing. It also keeps a history of which thread executes which chunk of the range and tries to recreate this execution pattern on subsequent executions. If a data set fits completely within the processors' caches, repeating the scheduling pattern can result in significant performance improvements.

Choosing a Grainsize (or Not) to Manage Task Granularity

At the beginning of this chapter, we talked about how important task granularity can be. When we use a blocked range type, we should always then highly tune our grainsize, right? Not necessarily. Selecting the right grainsize when using a blocked range can be extremely important – or almost irrelevant – it all depends on the Partitioner being used.

If we use a `simple_partitioner`, the grainsize is the sole determinant of the size of the ranges that will be passed to the body. When a `simple_partitioner` is used, the range is recursively subdivided until `is_divisible` returns false. In contrast, all of the other Partitioners have their own internal algorithms for deciding when to stop dividing ranges. Choosing a grainsize of 1 is typically sufficient for these other partitioners that use `is_divisible` as only a lower bound.

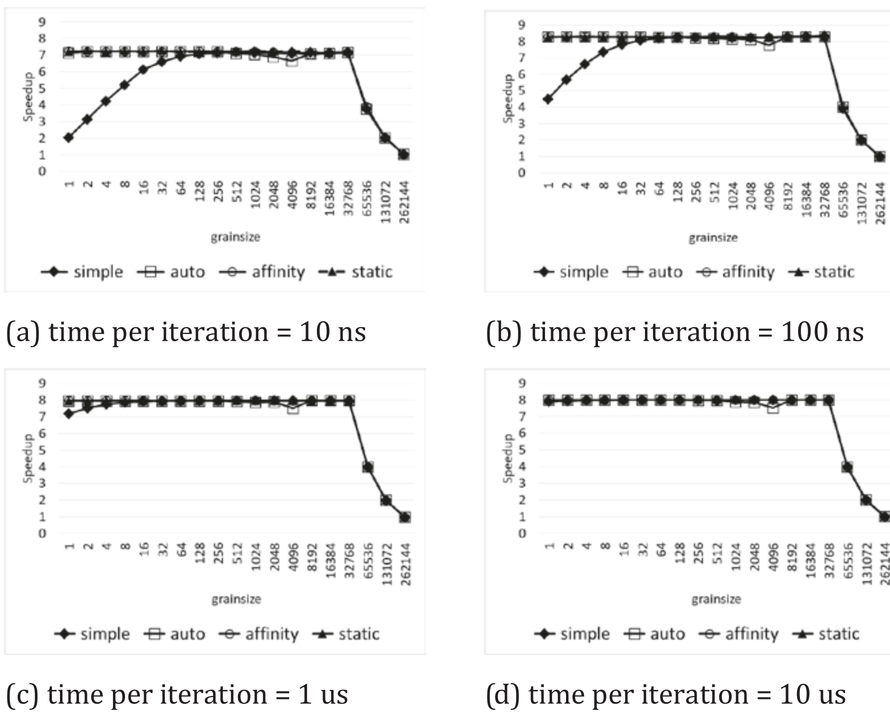
To demonstrate the impact of grainsize on the different Partitioners, we can use a simple `parallel_for` microbenchmark and vary the number of iterations in the loop (N), the grainsize, the execution time per loop iteration, and the Partitioner.

```
template< typename P >
static inline double executePfor(int num_trials, int N,
                                 int gs, P &p, double tpi) {
  tbb::tick_count t0;
  for (int t = -1; t < num_trials; ++t) {
    if (!t) t0 = tbb::tick_count::now();
    tbb::parallel_for (
      tbb::blocked_range<int>{0, N, gs},
      [tpi](const tbb::blocked_range<int> &r) {
        int e = r.end();
        for (int i = r.begin(); i < e; ++i) {
          spinWaitForAtLeast(tpi);
        }
      },
      p
    );
  }
  tbb::tick_count t1 = tbb::tick_count::now();
  return (t1 - t0).seconds()/num_trials;
}
```

Figure 16-4. *A function used to measure the time to execute a `parallel_for` with `N` iterations using a partitioner (`p`), a grainsize (`gs`), and time-per-iteration (`tpi`)*

All performance results presented in this chapter were collected on a single socket server with an Intel Xeon Processor E3-1230 with four cores supporting two hardware threads per core; the processor has a base frequency of 3.4 GHz, a shared 8 MB L3 cache, and per-core 256 KB L2 caches. The system was running SUSE Linux Enterprise Server 12. All samples were compiled using the Intel C++ Compiler 19.0 with Threading Building Blocks 2019, using the compiler flags “–std=c++11 –O2 –tbb”.

Figure 16-5 shows the results of the program in Figure 16-4 when executed for $N=2^{18}$ using each of the Partitioner types available in TBB and with a range of grainsizes. We can see that for a very small `time_per_iteration` of `10 ns`, the `simple_partitioner` approaches the other `partitioner`’s maximum performance when the grainsize is >= 128. As the time-per-iteration increases, the `simple_partitioner` approaches the maximum performance more quickly, since fewer iterations are needed to overcome scheduling overheads.



(a) time per iteration = 10 ns

(b) time per iteration = 100 ns

(c) time per iteration = 1 us

(d) time per iteration = 10 us

Figure 16-5. *Speedup for different Partitioner types and increasing grainsizes. The total number of iterations in the loop being tested is 2^{18} == 262144*

For all of the Partitioner types shown in Figure 16-5 except `simple_partitioner`, we see maximum performance from a grainsize of 1 until 4096. Our platform has 8 logical cores and therefore we need a grainsize less than or equal to $2^{18}/8$ == 32,768 to provide at least one chunk to each thread; consequently, all of the Partitioners begin to tail off after a grainsize of 32768. We might also note that at a grainsize of 4096, the `auto_partitioner` and `affinity_partitioner` show drops in performance in all of the figures. This is because picking large grainsizes limits the choices available to these algorithms, interfering with their ability to complete their automated partitioning.

This small experiment confirms that the grainsize is critically important for `simple_partitioner`. We can use a `simple_partitioner` to manually select the size of our tasks, but when we do so, we need to be more accurate in our choices.

A second take-away is that efficient execution, with a speedup close to the linear upper bound, is seen when the body size approaches 1 us (10ns x 128 = 1.28 us). This result reinforces the rule of thumb we presented earlier in the chapter! This should not be surprising since experience and experiments like these are the reason for our rule of thumb in the first place.

Ranges, Partitioners, and Data Cache Performance

Ranges and Partitioners can improve data cache performance by enabling cache-oblivious algorithms or by enabling cache affinity. Cache-oblivious algorithms are useful when a data set is too large to fit into the data caches, but reuse of data within the algorithm can be exploited if it is solved using a divide and conquer approach. In contrast, cache affinity is useful when the data set completely fits into the caches. Cache affinity is used to repeatedly schedule the same parts of a range onto the same processors – so that the data that fits in the cache can be accessed again from the same cache.

Cache-Oblivious Algorithms

A cache-oblivious algorithm is an algorithm that achieves good (or even optimal) use of data caches without depending upon knowledge of the hardware's cache parameters. The concept is similar to loop tiling or loop blocking but does not require an accurate tile or block size. Cache-oblivious algorithms often recursively divide problems into smaller and smaller subproblems. At some point, these small subproblems begin to fit into a machine's caches. The recursive subdivision might continue all the way down to the smallest possible size or there may be a cutoff point for efficiency – but this cutoff point is ***not*** related to the cache size and typically creates patterns that access data sized well below any reasonable cache size.

Because *Cache-oblivious algorithms* are not at all disinterested in cache performance, we've heard many other suggested names, such as *cache agnostic* since these algorithms optimize for whatever cache they encounter; and *cache paranoid*, since they assume there can be infinite levels of caches. But cache oblivious is the name used in the literature and it has stuck.

Here, we will use matrix transposition as an example of an algorithm that can benefit from a cache-oblivious implementation. A non-cache-oblivious serial implementation of matrix transposition is shown in Figure 16-6.

```
void fig_16_6(int N, double *a, double *b) {
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
      b[j*N+i] = a[i*N+j];
    }
  }
}
```

Figure 16-6. *A serial implementation of a matrix transposition*

For simplicity, let's assume that four elements fit in a cache line in our machine. Figure 16-7 shows the cache lines that will be accessed during the transposition of the first two rows of the N×N matrix a. If the cache is large enough, it can retain all of the cache lines accessed in b during that transposition of the first row of a and not need to reload these during that transposition of the second row of a. But if it is not large enough, these cache lines will need to be reloaded – resulting in a cache miss at each access to the matrix b. In the figure, we show a 16×16 array but imagine if it was very large.

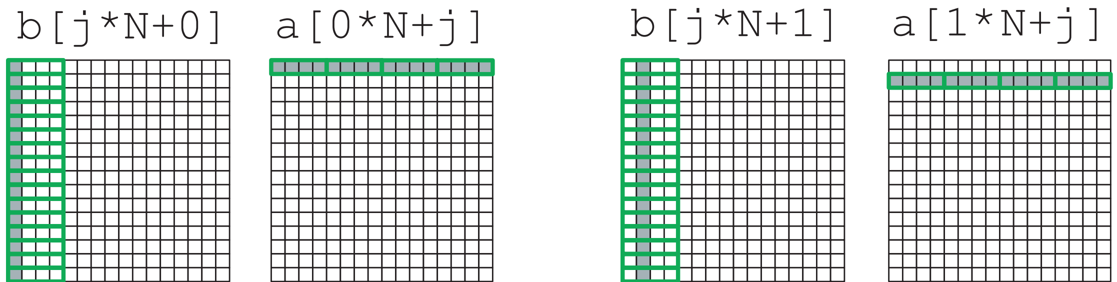


Figure 16-7. *The cache lines accessed when transposing the first two rows of the matrix* a*. For simplicity, we show four items in each cache line.*

A cache-oblivious implementation of this algorithm reduces the amount of data accessed between reuses of the same cache line or data item. As shown in Figure 16-8, if we focus on transposing only a small block of matrix a before moving on to other blocks of matrix a, we can reduce the number of cache lines that hold elements of b that need to be retained in the cache to get performance gains due to cache line reuse.

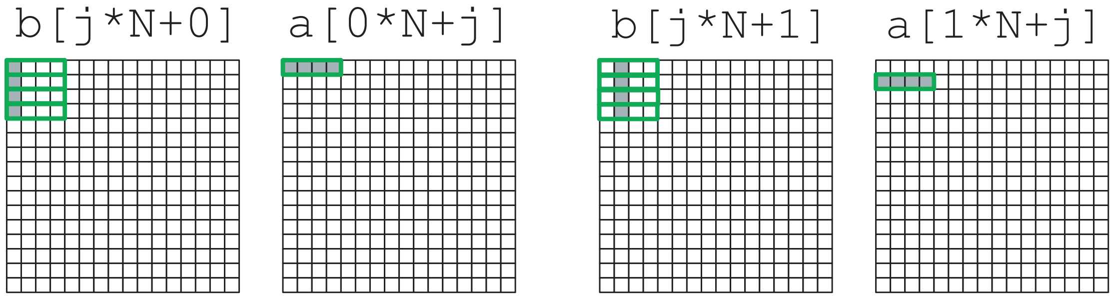


Figure 16-8. *Transposing a block at a time reduces the number of cache lines that need to be retained to benefit from reuse*

A serial implementation of a cache-oblivious implementation of matrix transposition is shown in Figure 16-9. It recursively subdivides the problem along the i and j dimensions and uses a serial for-loop when the range drops below a threshold.

```
void obliviousTranspose(int N, int ib, int ie, int jb, int je,
                        double *a, double *b, int gs) {
  int ilen = ie-ib;
  int jlen = je-jb;
  if (ilen > gs ||  jlen > gs) {
     if ( ilen > jlen ) {
       int imid = (ib+ie)/2;
       obliviousTranspose(N, ib, imid, jb, je, a, b, gs);
       obliviousTranspose(N, imid, ie, jb, je, a, b, gs);
     } else {
       int jmid = (jb+je)/2;
       obliviousTranspose(N, ib, ie, jb, jmid, a, b, gs);
       obliviousTranspose(N, ib, ie, jmid, je, a, b, gs);
     }
  } else {
    for (int i = ib; i < ie; ++i) {
      for (int j = jb; j < je; ++j) {
        b[j*N+i] = a[i*N+j];
      }
    }
  }
}
```

Figure 16-9. *A serial cache-oblivious implementation of a matrix transposition*

Because the implementation alternates between dividing in the i and j direction, the matrix a is transposed using the traversal pattern shown in Figure 16-10, first completing block 1, then 2, then 3, and so on. If gs is 4 and our cache line size is 4,

we get the reuse within each block that we showed in Figure 16-8. But if our cache line is 8 items instead of 4 (which is much more likely for real systems), we would get reuse not only within the smallest blocks but also across blocks. For example, if the data cache can retain all of the cache lines loaded during blocks 1 and 2, these will be reused when transposing blocks 3 and 4.

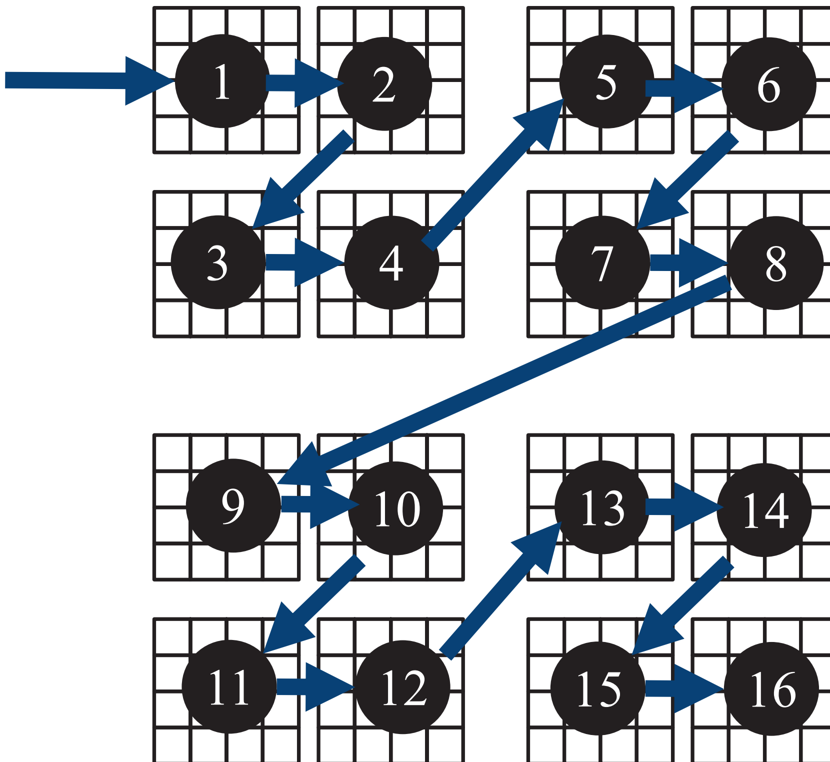


Figure 16-10. *A traversal pattern that computes the transpose for sub-blocks of a before moving on to other blocks*

This is the true power of cache-oblivious algorithms – we don't need to exactly know the sizes of the levels of the memory hierarchy. As the subproblems get smaller, they fit in progressively smaller parts of the memory hierarchy, improving reuse at each level.

The TBB loop algorithms and the TBB scheduler are designed to specifically support cache-oblivious algorithms. We can therefore quickly implement a cache-oblivious parallel implementation of matrix transposition using a `parallel_for`, a `blocked_range2d`, and a `simple_partitioner` as shown in Figure 16-11. We use a `blocked_range2d` because we want the iteration space subdivided into two-dimensional blocks.

And we use a `simple_partitioner` because we only get the benefits from reuse if the blocks are subdivided down to sizes smaller than the cache size; the other Partitioner types optimize load balancing and so may choose larger range sizes if those are sufficient to balance load.

```
double fig_16_11(int N, double *a, double *b, int gs) {
  tbb::tick_count t0 = tbb::tick_count::now();
  tbb::parallel_for(
    tbb::blocked_range2d<int,int>{0, N, gs, 0, N, gs},
    [N, a, b](const tbb::blocked_range2d<int,int> &r) {
      int ie = r.rows().end();
      int je = r.cols().end();
      for (int i = r.rows().begin(); i < ie; ++i) {
        for (int j = r.cols().begin(); j < je; ++j) {
          b[j*N+i] = a[i*N+j];
        }
      }
    }, simple_partitioner()
  );
  tbb::tick_count t1 = tbb::tick_count::now();
  return (t1-t0).seconds();
}
```

Figure 16-11. *A cache-oblivious parallel implementation of matrix transposition that uses a `simple_partitioner`, a `blocked_range2d`, and a grainsize (`gs`)*

Figure 16-12 shows that the way the TBB `parallel_for` recursively subdivides ranges creates the same blocks that we want for our cache-oblivious implementation. The depth-first work and breadth-first stealing behavior of the TBB scheduler also means that the blocks will execute in an order similar to the one shown in Figure 16-10.

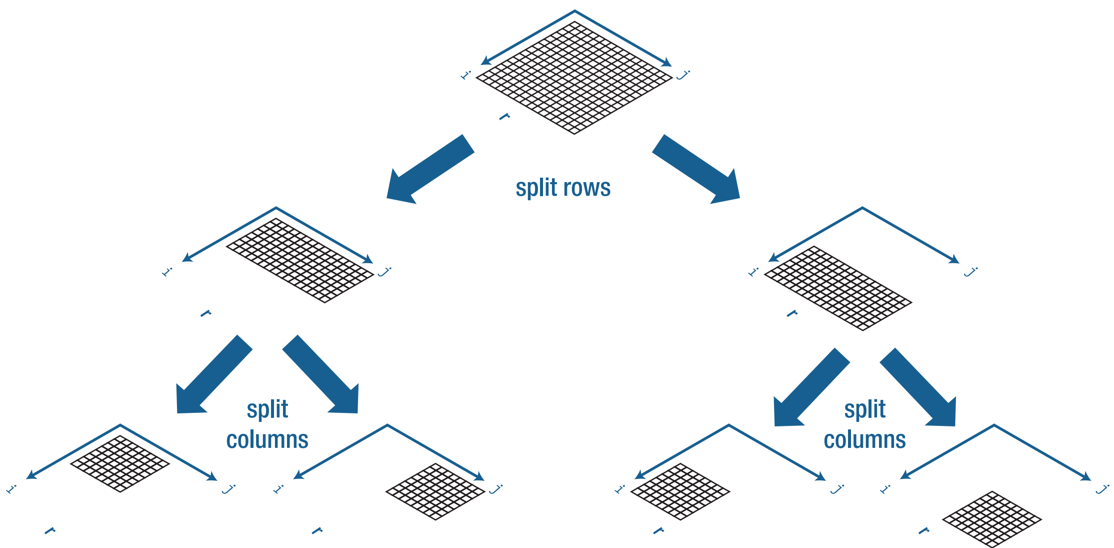


Figure 16-12. *The recursive subdivision of the blocked2d_range provides a division that matches the blocks we want for our cache-oblivious parallel implementation*

Figure 16-13 shows the performance of the serial cache-oblivious implementation in Figure 16-9, the performance of an implementation using a 1D `blocked_range`, and the performance of a `blocked_range2d` implementation similar to the one in Figure 16-11. We implemented our parallel versions so that we could change the grainsize and partitioner easily. The code for all of the versions is available in `fig_16_11.cpp`.

In Figure 16-13, we show the speedup of our implementations on an 8192×8192 matrix compared to the simple serial implementation from Figure 16-6.

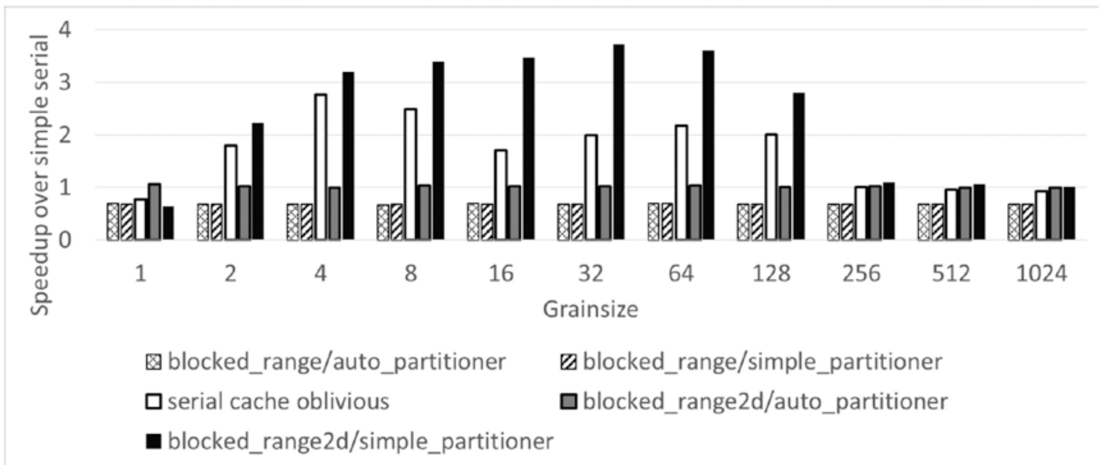


Figure 16-13. *The speedup on our test machine for N=8192 with various grainsizes and partitioners*

Matrix transposition is limited by the speed at which we can read and write data – there is no compute whatsoever. We can see from Figure 16-13 that our 1D `blocked_range` parallel implementations perform worse than our simple serial implementation, regardless of the grainsize we use. The serial implementation is already limited by the memory bandwidth – adding additional threads simply adds more pressure on the already-stressed memory subsystem and does nothing to help matters.

Our serial cache-oblivious algorithm reorders memory accesses, reducing the number of cache misses. It significantly outperforms the simple version. When we use a `blocked_range2d` in our parallel implementation, we similarly get 2D subdivisions. But as we see in Figure 16-13, only when we use a `simple_partitioner` does it fully behave like a cache-oblivious algorithm. In fact, our cache-oblivious parallel algorithm with a `blocked_range2d` and a `simple_partitioner` reduces pressure on the memory hierarchy to such a degree that now using multiple threads can improve performance over the serial cache-oblivious implementation!

Not all problems have cache-oblivious solutions, but many common problems do. It is worth the time to research problems to see if a cache-oblivious solution is possible and worthwhile. If so, the blocked range types and the `simple_partitioner` will make it very easy to implement one with TBB algorithms.

Cache Affinity

Cache-oblivious algorithms improve cache performance by breaking problems, which have data locality but do not fit into the cache, down into smaller problems that do fit into the cache. In contrast, cache affinity addresses the repeated execution of ranges across data that already fit in the cache. Since the data fits in the cache, if the same subranges are assigned to the same processors on subsequent executions, the cached data can be accessed more quickly. We can use either an `affinity_partitioner` or a `static_partitioner` to enable cache affinity for the TBB loop algorithms. Figure 16-14 shows a simple microbenchmark that adds a value to each element in a 1D array. The function receives a reference to the Partitioner – we need to receive the Partitioner as a reference to record history in the `affinity_partitioner` object.

```
template <typename Partitioner>
double fig_16_14(double v, int N, double *a, Partitioner &p) {
  tbb::parallel_for( tbb::blocked_range<int>(0, N, 1),
    [v, a](const tbb::blocked_range<int> &r) {
      int ie = r.end();
      for (int i = r.begin(); i < ie; ++i) {
        a[i] += v;
      }
    }, p
  );
}
```

Figure 16-14. *A function that uses a TBB `parallel_for` to add a value to all of the elements of a 1D array*

To see the impact of cache affinity we can execute this function repeatedly, sending in the same value for `N` and the same array `a`. When using an `auto_partitioner`, the scheduling of the subranges to threads will vary from invocation to invocation. Even if array `a` completely fits into the processors' caches, the same region of `a` may not fall on the same processor in subsequent executions:

```
for (int i = 0; i < M; ++i) {
  fig_16_14(v[i], N, a, tbb::auto_partitioner{});
}
```

If we use an `affinity_partitioner` however, the TBB library will record the task scheduling and use affinity hints to recreate it on each execution (see Chapter 13 for more information on affinity hints). Because the history is recorded in the Partitioner,

we must pass the same Partitioner object on subsequent executions, and cannot simply create a temporary object like we did with `auto_partitioner`:

```
tbb::affinity_partitioner aff_p;
for (int i = 0; i < M; ++i) {
  fig_16_14(v[i], N, a, aff_p);
}
```

Finally, we can also use a `static_partitioner` to create cache affinity. Because the scheduling is deterministic when we use a `static_partitioner`, we do not need to pass the same partitioner object for each execution:

```
for (int i = 0; i < M; ++i) {
  fig_16_14(v[i], N, a, tbb::static_partitioner{});
}
```

We executed this microbenchmark on our test machine using N=100,000 and M=10,000. Our array of doubles will be 100,000 × 8 = 800 K in size. Our test machine has four 256 K L2 data caches, one per core. When using an `affinity_partitioner`, the test completed 1.4 times faster than when using the `auto_partitioner`. When using a `static_partitioner`, the test completed 2.4 times faster than when using the `auto_partitioner`! Because the data was able to fit into the aggregate L2 cache size (4 × 256 K = 1 MB), replaying the same scheduling had a significant impact on the execution time. In the next section, we'll discuss why the `static_partitioner` outperformed the `auto_partitioner` in this case and why we shouldn't be too surprised, or excited about that. If we increase N to 1,000,000 elements, we no longer see a large difference in the execution times since array `a` is now too large to fit in the caches of our test system – in this case, re-thinking the algorithm to implement tiling/blocking to exploit cache locality is necessary.

Using a `static_partitioner`

The `static_partitioner` is the lowest overhead partitioner, and it quickly provides a uniform distribution of a blocked range across the threads in an arena. Since the partitioning is deterministic, it also can improve cache behavior when a loop or a series of loops are executed repeatedly on the same range. In the previous section, we saw that it out-performed `affinity_partitioner` significantly for our microbenchmark. However, because it creates just enough chunks to provide one to each thread in the

arena, there is no opportunity for work stealing to balance the load dynamically. In effect, the `static_partitioner` disables the TBB library's work-stealing scheduling approach.

There is a good reason though for TBB to include `static_partitioner`. As the number of cores increase, random work stealing becomes costlier; especially when transitioning from a serial part of an application to a parallel part. When the master thread first spawns new work into the arena, all of the worker threads wake up and as a *thundering herd* try to find work to do. To make matters worse, they don't know where to look and start randomly peeking into not only the master thread's deque, but each other's local deques too. Some worker thread will eventually find the work in the master and subdivide it, and another worker will eventually find this subdivided piece, subdivide it, and so on. And after a while, things will settle down and all of the workers will find something to do and will happily work from their own local deques.

But, if we already know that the workload is well balanced, that the system is not oversubscribed, and all our cores are equally powerful – do we really need all of this work-stealing overhead to just get a uniform distribution across the workers? Not if we use a `static_partitioner`! It is designed for just this case. It pushes tasks that uniformly distribute the range to the worker threads so that they don't have to steal tasks at all. When it applies, `static_partitioner` is the most efficient way to partition a loop.

But don't get too excited about `static_partitioner!` If the workload is not uniform or any of the cores are oversubscribed with additional threads, then using a `static_partitioner` can wreck performance. For example, Figure 16-15 shows the same microbenchmark configuration we used in Figure 16-5(c) to examine the impact of grainsize on performance. But Figure 16-15 shows what happens if we add a single extra thread running on one of the cores. For all but the `static_partitioner`, there is a small impact due to the extra thread. The `static_partitioner` however assumes that all of the cores are equally capable and uniformly distributes the work among them. As a result, the overloaded core becomes a bottleneck and the speedup takes a huge performance hit.

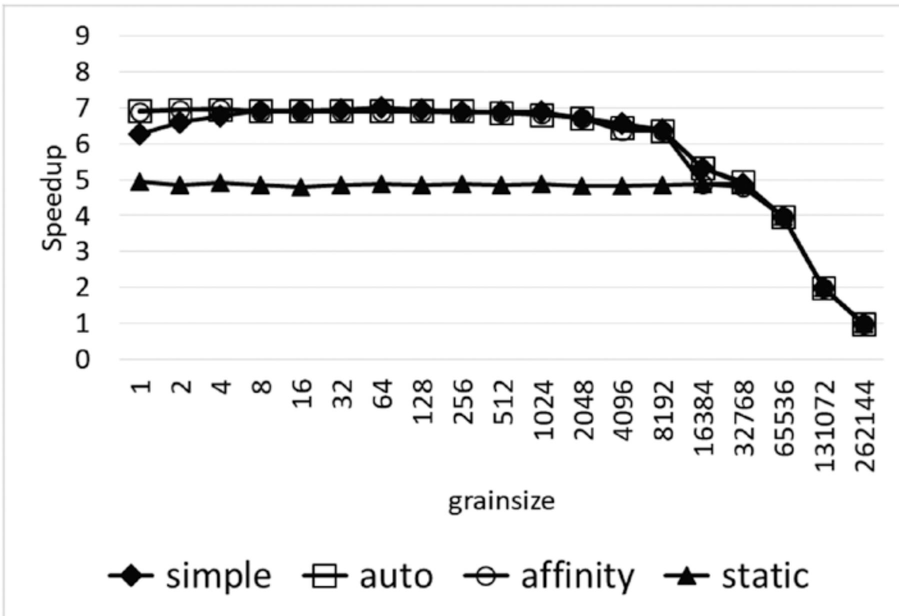


Figure 16-15. *Speedup for different Partitioner types and increasing grainsizes when an additional thread executes a spin loop in the background. The time-per-iteration is set to 1 us.*

Figure 16-16 shows a loop where the work increases with each iteration. If a `static_partitioner` is used, the thread that gets the lowest set of iterations will have much less work to do than the unlucky thread that gets the highest set of iterations.

```
template <typename Partitioner>
double fig_16_16(int N, Partitioner &p) {
  tbb::parallel_for( tbb::blocked_range<int>(0, N, 1),
    [](const tbb::blocked_range<int> &r) {
      int ie = r.end();
      for (int i = r.begin(); i < ie; ++i) {
        spinForMicroseconds(i);
      }
    }, p
  );
}
```

Figure 16-16. *A loop where the work increases in each iteration*

If we run the loop in Figure 16-16 ten times using each partitioner type with N=1000, we see the following results:

```
auto_partitioner = 0.629974 seconds
affinity_partitioner = 0.630518 seconds
static_partitioner = 1.18314 seconds
```

The `auto_partitioner` and `affinity_partitioner` are able to rebalance the load across the threads, while the `static_partitioner` is stuck with its initial uniform, but unfair distribution.

The `static_partitioner` is therefore almost exclusively useful in High Performance Computing (HPC) applications. These applications run on systems with many cores and often in batch mode, where a single application is run at a time. If the work load does not need ***any*** dynamic load balancing, then `static_partitioner` will almost always outperform the other partitioners. Unfortunately, well-balanced workloads and single-user, batch-mode systems are the exception and not the rule.

Restricting the Scheduler for Determinism

In Chapter 2, we discussed **Associativity and floating-point types**. We noted that any implementation of floating-point numbers is an approximation, and so parallelism can lead to different results when we depend on properties like associativity or commutativity – those results aren't necessarily wrong; they are just different. Still, in the case of reduction, TBB provides a `parallel_deterministic_reduce` algorithm if we want to ensure that we get the same results for each execution on the same input data when executed on the same machine.

As we might guess, `parallel_deterministic_reduce` only accepts `simple_partitioner` or `static_partitioner`, since the number of subranges is deterministic for both of these partitioner types. The `parallel_deterministic_reduce` also always executes the same set of split and join operations on a given machine no matter how many threads dynamically participate in execution and how tasks are mapped to threads – the `parallel_reduce` algorithm may not. The result is that `parallel_deterministic_reduce` will always return the same result when run on the same machine – but sacrifices some flexibility to do so.

Figure 16-17 shows the speedup for the pi calculation example from Chapter 2 when implemented using `parallel_reduce` (`r-auto`, `r-simple`, and `r-static`) and `parallel_deterministic_reduce` (`d-simple` and `d-static`). The maximum speedup is similar for both; however, the `auto_partitioner` performs very well for `parallel_reduce`, and that is simply not an option with `parallel_deterministic_reduce`. If needed, we can implement a deterministic version of our benchmark but then must deal with the complications of choosing a good grainsize.

While `parallel_deterministic_reduce` will have some additional overhead because it must perform all of the splits and joins, this overhead is typically small. The bigger limitation is that we cannot use any of the partitioners that automatically find a chunk size for us.

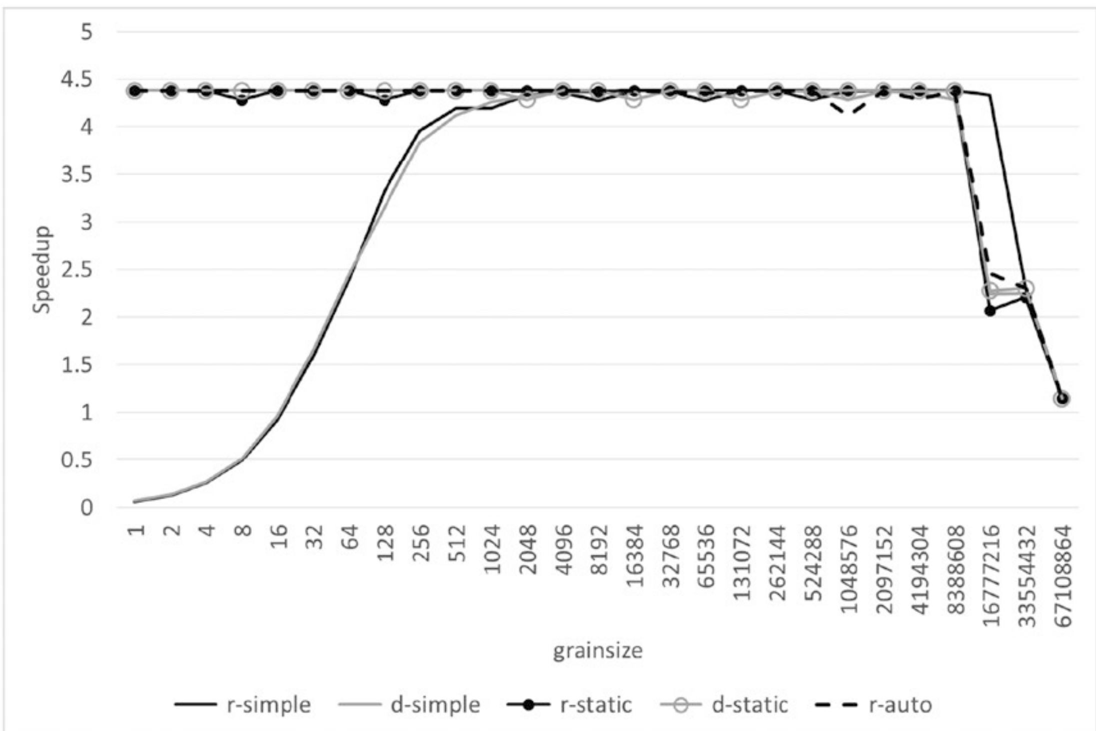


Figure 16-17. *Speedup for the pi example from Chapter 2 using `parallel_reduce` with an `auto_partitioner` (`r-auto`), a `simple_partitioner` (`r-simple`), and a `static_partitioner` (`r-static`); and `parallel_deterministic_reduce` with a `simple_partitioner` (`d-simple`) and a `static_partitioner` (`d-static`). We show results for grainsizes ranging from 1 to N.*

Tuning TBB Pipelines: Number of Filters, Modes, and Tokens

Just as with the loop algorithms, the performance of TBB pipelines is impacted by granularity, locality, and available parallelism. Unlike the loop algorithms, TBB pipelines do not support Ranges and Partitioners. Instead, the controls used to tune pipelines include the number of filters, the filter execution modes, and the number of tokens passed to the pipeline when it is run.

TBB pipeline filters are spawned as tasks and scheduled by the TBB library, and therefore, just as with the subranges created by the loop algorithms, we want the filter bodies to execute long enough to mitigate overheads but we also want ample parallelism. We balance these concerns by how we break our work into filters. The filters should also be well balanced in execution time since the slowest serial stage will be a bottleneck.

As described in Chapter 2, pipeline filters are also created with an execution mode: `serial_in_order`, `serial_out_of_order`, or `parallel`. When using `serial_in_order` mode, a filter can process at most one item at a time, and it must process them in the same order that the first filter generated them in. A `serial_out_of_order` filter is allowed to execute the items in any order. A `parallel` filter is allowed to execute on different items in parallel. We will look at how these different modes limit performance later in this section.

When run, we need to provide a `max_number_of_live_tokens` argument to a TBB pipeline, which constrains the number of items that are allowed to flow through the pipeline at any given time.

Figure 16-18 shows the structure of the microbenchmarks we will use to explore these different controls. In the figure, both pipelines are shown with eight filters – but we will vary this number in our experiments. The top pipeline has filters that use the same execution `mode,` and all have the same `spin_time` – so this represents a very well-balanced pipeline. The bottom pipeline has one filter than spins for `imbalance * spin_time` – we will vary this imbalance factor to see the impact of imbalance on speedup.

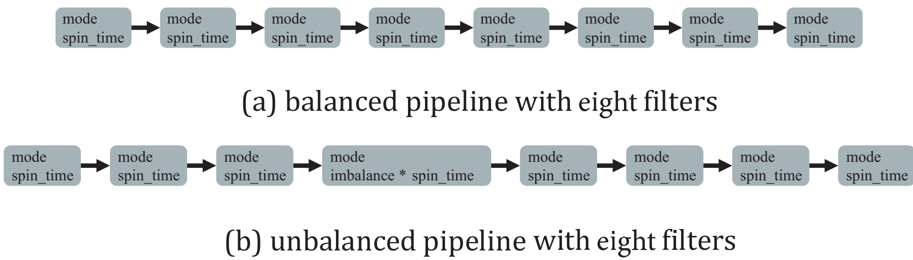


Figure 16-18. *A balanced pipeline microbenchmark and an imbalanced pipeline microbenchmark*

Understanding a Balanced Pipeline

Let's first consider how well our rule of thumb for task sizes applies to pipelines. Is a filter body of 1 microsecond sufficient to mitigate overheads? Figure 16-19 shows the speedup of our balanced pipeline microbenchmark when fed 8000 items while using only a single token. The results are shown for various filter execution times. Since there is only a single token, only a single item will be allowed to flow through the pipeline at a time. The result is a serialized execution of the pipeline (even when the filter execution mode is set to parallel).

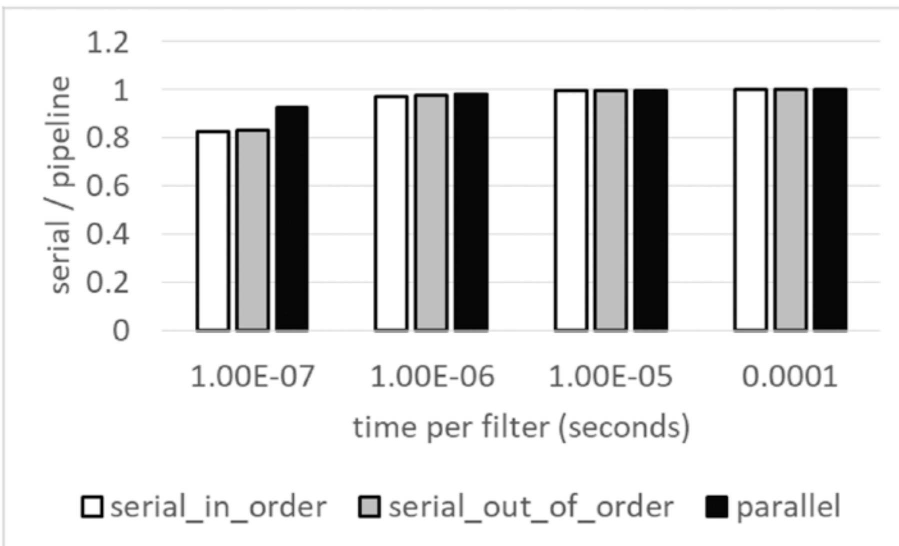


Figure 16-19. *The overhead seen by different filter execution modes when executing a balanced pipeline with eight filters, a single token, and 8000 items on our test machine*

When compared to a true serial execution, where we execute the proper number of spins in a for-loop, we see the impact of managing the work as a TBB pipeline. In Figure 16-19, we see that when the `spin_time` approaches 1 microsecond, the overhead is fairly low, and we get very close to the execution time of the true serial execution. It seems that our rule of thumb applies to a TBB pipeline too!

Now, let's look at how the number of filters affects performance. In a serial pipeline, the parallelism comes only from overlapping different filters. In a pipeline with parallel filters, parallelism is also obtained by executing the parallel filters simultaneously on different items. Our target platform supports eight threads, so we should expect at most a speedup of 8 for a parallel execution.

Figure 16-20 shows the speedup of our balanced pipeline microbenchmark when setting the number of tokens to 8. For both serial modes, the speedup increases with the number of filters. This is important to remember, since the speedup of a serial pipeline does not scale with the data set size like the TBB loop algorithms do. The balanced pipeline that contains all parallel filters however has a speedup of 8 even with only a single filter. This is because the 8000 input items can be processed in parallel in that single filter – there is no serial filter to become a bottleneck.

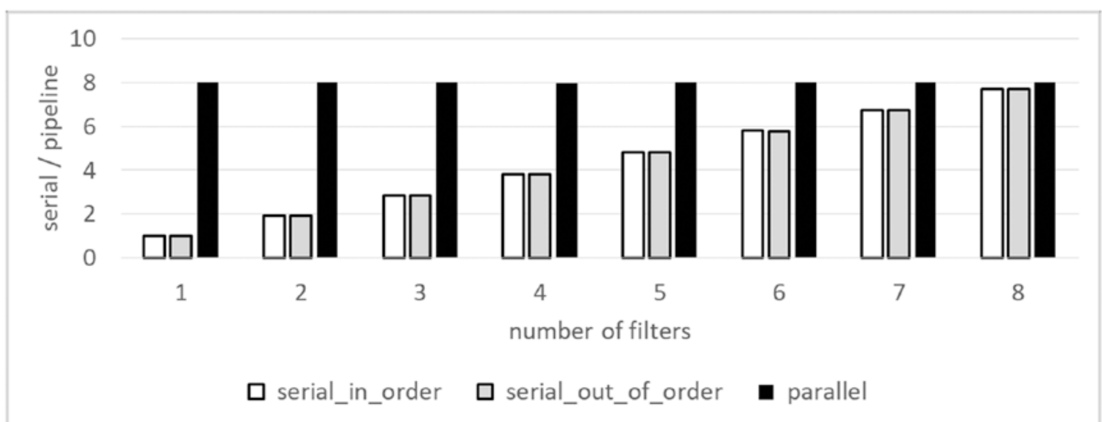


Figure 16-20. *The speedup achieved by the different filter execution modes when executing a balanced pipeline with eight tokens, 8000 items, and an increasing number of filters. The filters spin for 100 microseconds.*

In Figure 16-21, we see the speedup for our balanced pipeline when using eight filters but with varying numbers of tokens. Because our platform has eight threads, if we have fewer than eight tokens, there are not enough items in flight to keep all of

the threads busy. Once we have at least eight items in the pipeline, all threads can participate. Increasing the number of tokens past eight has little impact on performance.

Figure 16-21. *The speedup achieved by the different filter execution modes when executing a balanced pipeline with eight filters, 8000 items, and an increasing number of tokens. The filters spin for 100 microseconds.*

Understanding an Imbalanced Pipeline

Now, let's look at the performance of the imbalanced pipeline from Figure 16-18. In this microbenchmark, all of the filters spin for `spin_time` seconds except for one of the filters that spins for `spin_time * imbalance` seconds. The work required to process `N` items as they pass through our imbalanced pipeline with eight filters is therefore

$$T_1 = N * (7 * spin_time + spin_time * imbalance)$$

In the steady state, a serial pipeline is limited by the slowest serial stage. The critical path length of this same pipeline when the imbalanced filter executes with serial mode is equal to

$$T_\infty = N * \max(spin_time, spin_time * imbalance)$$

Figure 16-22 shows the results of our imbalanced pipeline when executed on our test platform with different imbalance factors. We also include the theoretical maximum speedup, labeled as "work/critical path," calculated as

$$Speedup_{max} = \frac{7 * \text{spin_time} + \text{spin_time} * \text{imbalance}}{\max(\text{spin_time},\ \text{spin_time} * \text{imbalance})}$$

Not unexpectedly, Figure 16-22 shows that serial pipelines are limited by their slowest filters – and the measured results are close to what our work/critical path length calculation predicts.

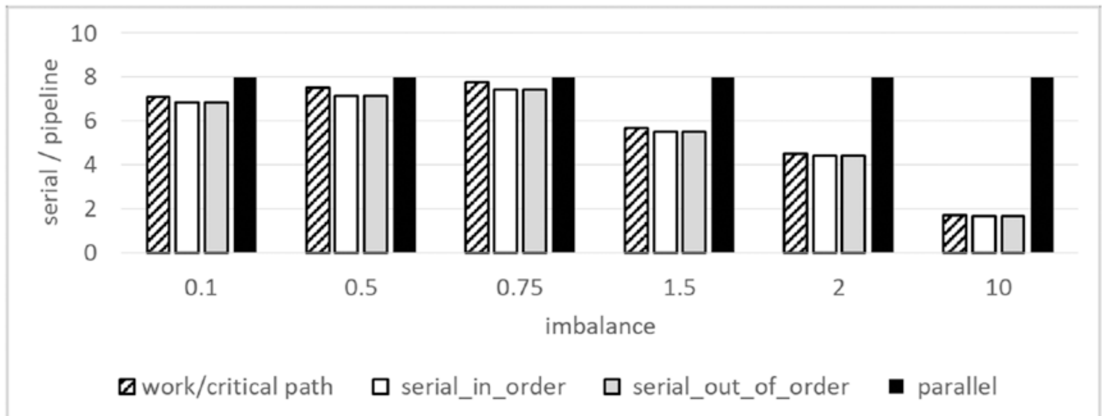


Figure 16-22. *The speedup achieved by the different filter execution modes when executing an imbalanced pipeline with eight filters, 8000 items, and different imbalance factors. Seven of the filters spin for 100 microseconds, and the other spins for* `imbalance * 100` *microseconds.*

In contrast, the parallel pipeline in Figure 16-22 is shown to not be limited by the slowest stage because the TBB scheduler can overlap the execution of the slowest filter with other invocations of that same filter. You may be wondering if increasing the number of tokens beyond eight will help, but in this case, no. Our test system has only eight threads, so we can at most overlap eight instances of the slowest filter. While there may be cases where a temporary load imbalance can be smoothed out by having more tokens than the number of threads, in our microbenchmark where the imbalance is a constant factor, we are in fact limited by the critical path length and the number of threads – and any number of additional tokens will not change that.

However, there are algorithms in which an insufficient number of tokens will hamper the automatic load balancing feature of the work-stealing TBB scheduler. This is the case when the stages are not well balanced and there are serial stages stalling the pipe. A. Navarro et al. demonstrated (see the “For More Information” section at the end of the

chapter) that a pipeline algorithm implemented in TBB can yield optimal performance if appropriately configured with the right number tokens. She devised an analytical model based on queueing theory that helps in finding this key parameter. A major take-away of the paper is that when the number of tokens is sufficiently large, the work stealing in TBB emulates a global queue that is able to feed all the threads (in queueing theory, a theoretical centralized system with a single global queue served by all the resources is known to be the ideal case). However, in reality, a global single queue would exhibit contention when it is served by a large number of threads. The fundamental advantage of the TBB implementation is that it resorts to a distributed solution with one queue per thread that behaves as a global queue thanks to the work-stealing scheduler. This is, the decentralized TBB implementation performs like the ideal centralized system but without the bottleneck of the centralized alternative.

Pipelines and Data Locality and Thread Affinity

With the TBB loop algorithms, we used the blocked range types `affinity_partitioner` and `static_partitioner` to tune cache performance. The TBB `parallel_pipeline` function and the `pipeline` class have no similar options. But all is not lost! The execution order built into TBB pipelines is designed to enhance temporal data locality without the need to do anything special.

When a TBB master or worker thread completes the execution of a TBB filter, it executes the next filter in the pipeline unless that filter cannot be executed due to execution mode constraints. For example, if a filter f_0 generates an item `i` and its output is passed to the next filter f_1, the same thread that ran f_0 will move on to execute f_1 – unless that next filter is a `serial_out_of_order` filter and it is currently processing something else, or if it is a `serial_in_order` filter and item `i` is not the next item in line. In those cases, the item is buffered in the next filter and the thread will look for other work to do. Otherwise to maximize locality, the thread will follow the data it just generated and process that item by executing the next filter.

Internally, the processing of one item in the filter f_0 is implemented as a task executed by a thread/core. When the filter is done, the task recycles itself (see task recycling in Chapter 10) to execute the next filter f_1. Essentially, the dying task f_0 reincarnates into the new f_1 task, bypassing the scheduler – the same thread/core that executed f_0 will also execute f_1. In terms of data locality and performance, this is way

better than what a regular/naive pipeline implementation would do: filter f_0 (served by one or several threads) enqueuing the item in filter f_1's queue (where f_1 is also served by one or several threads). This naive implementation wrecks locality because the item processed by filter f_0 on one core is likely to be processed on a different core by filter f_1. In TBB, if f_0 and f_1 fulfil the conditions mentioned previously, this will never happen. As a result, the TBB pipeline is biased toward finishing items that are already in-flight before injecting more items at the beginning of the pipeline; this behavior not only exploits data locality but uses less memory by reducing the size of the queues that are necessary for serial filters.

Unfortunately, TBB pipeline filters *do not* support affinity hints. There is no way to hint that we want a particular filter to execute on a particular worker thread. But, perhaps surprisingly, there is a hard affinity mechanism, `thread_bound_filter`. Using `thread_bound_filter` however requires using the more error-prone, type-unsafe `tbb::pipeline` interface, which we describe as part of the next section, "Deep in the Weeds."

Deep in the Weeds

This section covers some features that are rarely used by TBB users, but when needed, they can be extremely helpful. You might choose to skip this section and read it on demand if you ever need to create your own Range type or use a `thread_bound_filter` in a TBB pipeline. Or, if you really want to know as much as possible about TBB, read on!

Making Your Own Range Type

As mentioned earlier in this chapter, the blocked range types capture most common scenarios. Over our years of using TBB, we have personally only encountered a handful of situations in which it made sense to implement our own Range type. But if we need to, we can create our own range types by implementing classes that model the Range Concept described in Figure 16-1.

As an example of a useful but atypical range type, we can revisit the quicksort algorithm again, as shown in Figure 16-23.

```
struct SortData {
  int id;
  double value;
  SortData(int i, double v) : id(i), value(v) {}
  bool operator<(const SortData &other) const {
    return value < other.value;
  }
  bool operator==(const SortData &other) const {
    return value == other.value;
  }
};

using QSVector = std::vector<SortData>;

QSVector::iterator doShuffle(QSVector::iterator b,
                             QSVector::iterator e) {
  QSVector::iterator i = b, j = e-1;
  double pivot_value = b->value;
  while (i != j) {
    while (i != j && pivot_value < j->value) --j;
    while (i != j && i->value <= pivot_value) ++i;
    std::iter_swap(i, j);
  }
  std::iter_swap(b, i);
  return i;
}

void serialQuicksort(QSVector::iterator b,
                     QSVector::iterator e) {
  if (b >= e) return;
  QSVector::iterator i = doShuffle(b,e);
  serialQuicksort(b, i);
  serialQuicksort(i+1, e);
}
```

Figure 16-23. *The implementation of a serial quicksort*

Here, we will parallelize quicksort not as a recursive algorithm at all, but instead use a `parallel_for` and our own custom `ShuffleRange`. Our `pforQuicksort` implementation is shown in Figure 16-24.

```
class ShuffleRange {
  QSVector::iterator myBegin;
  QSVector::iterator myEnd;

public:
  static const bool is_splittable_in_proportion = false;
  static const int cutoff = 100;

  // constructors
  ShuffleRange(const QSVector::iterator b,
               const QSVector::iterator e )
    : myBegin(b), myEnd(e) { }

  ShuffleRange(const ShuffleRange &r)
    : myBegin(r.myBegin), myEnd(r.myEnd) { }

  ShuffleRange(ShuffleRange &r, tbb::split)
      : myBegin(r.myBegin), myEnd(r.myEnd) {
      QSVector::iterator b = r.myBegin;
      QSVector::iterator e = r.myEnd;
      QSVector::iterator i = doShuffle(b,e);
      r.myEnd = i;
      myBegin = i+1;
  }

  bool empty() const { return myBegin >= myEnd; }
  bool is_divisible() const { return myEnd-myBegin >= cutoff;
}
  QSVector::iterator begin() const { return myBegin; }
  QSVector::iterator end() const { return myEnd; }
};

void pforQuicksort(QSVector::iterator b, QSVector::iterator e)
{
  tbb::parallel_for(ShuffleRange(b, e),
    [](const ShuffleRange &r) {
      serialQuicksort(r.begin(), r.end());
    },
    tbb::simple_partitioner()
  );
}
```

Figure 16-24. *Implementing a parallel quicksort using a `parallel_for` and a custom `ShuffleRange` that implements a Range*

In Figure 16-24, we can see that the parallel_for body lambda expression is the base case, where we call a serialQuicksort. We also use a simple_partitioner, which means that our range will be recursively divided until it returns false from its is_divisible method. All of the shuffling magic of quicksort therefore needs to happen in the ShuffleRange class as it splits itself into subranges. The class definition of ShuffleRange is also shown in Figure 16-24.

The ShuffleRange models the Range concept, defining a copy constructor, a splitting constructor, an empty method, an is_divisible method, and an is_splittable_in_proportion member variable that is set to false. This class also holds begin and end iterators that delineate the elements of the array and a cutoff value.

Let's start with empty. The range is empty if its begin iterator is at or past its end iterator.

We use our cutoff value to determine if the range should be further divided. Remember, we are using a simple_partitioner, so the parallel_for will keep dividing the ranges until is_divisible returns false. So, the ShuffleRange is_divisible implementation is just a check against this cutoff value.

Ok, now we can look at the heart of our implementation, the ShuffleRange splitting constructor shown in Figure 16-24. It receives a reference to the original ShuffleRange r that needs to be split and a tbb::split object that is used to distinguish this constructor from the copy constructor. The body of the constructor is the basic pivot and shuffle algorithm. It updates the original range r to be the left partition and the newly constructed ShuffleRange to be the right partition.

Executing our pforQuicksort on our test platform yields performance results that are very similar to the parallel_invoke implementation from Chapter 2. But this example shows just how flexible the Range Concept is. We may think of the recursive division of the range as negligible in a parallel_for, but in our pforQuicksort implementation it is not. We rely on the splitting of the ShuffleRange to do a substantial portion of the work.

The Pipeline Class and Thread-Bound Filters

As we noted in our earlier discussions in this chapter, affinity hints are not supported by tbb::parallel_pipeline. We cannot express that we prefer that a particular filter execute on a specific thread. However, there is support for thread-bound filters if we use the older, thread-unsafe tbb::pipeline class! These thread-bound filters are not

processed at all by TBB worker threads; instead, we need to explicitly process items in these filters by calling their process_item or try_process_item functions directly.

Typically, a thread_bound_filter is not used to improve data locality, but instead it is used when a filter must be executed on a particular thread – perhaps because only that thread has the rights to access the resources required to complete the action implemented by the filter. Situations like this can arise in real applications when, for example, a communication or offload library requires that all communication happen from a particular thread.

Let's consider a contrived example that mimics this situation, where only the main thread has access to an opened file. To use a thread_bound_filter, we need to use the type unsafe class interfaces of tbb::pipeline. We cannot create a thread_bound_filter when using the tbb::parallel_pipeline function. We will soon see why it would never make sense to use a thread_bound_filter with the parallel_pipeline interface anyway.

In our example, we create three filters. Most of our filters will inherit from tbb::filter, overriding the operator() function:

```
namespace tbb {
    class filter {
    public:
        enum mode {
            parallel = implementation-defined,
            serial_in_order = implementation-defined,
            serial_out_of_order = implementation-defined
        };
        bool is_serial() const;
        bool is_ordered() const;
        virtual void* operator()( void* item ) = 0;
        virtual void finalize( void* item ) {}
        virtual ~filter();
    protected:
        explicit filter( mode );
    };
}
```

Our SourceFilter, shown in Figure 16-25, is a serial_in_order filter that inherits from tbb::filter and generates a series of numbers. The type unsafe interfaces implemented by tbb::pipeline require that we return the output of each filter as a void *. NULL is used to indicate the end of the input stream. We can easily see why the newer parallel_pipeline interface is preferred when it applies.

The second filter type we create, `MultiplyFilter`, multiplies the incoming value by 2 and returns it. It too will be a `serial_in_order` filter and inherit from `tbb::filter`.

Finally, `BadWriteFilter` implements a filter that will write the output to a file. This class also inherits from `tbb::filter` as shown in Figure 16-25.

The function `fig_16_25` puts all of these classes together – while purposely introducing an error. It creates a three-stage pipeline using our filter classes and the `tbb::pipeline` interface. It creates a pipeline object and then adds each of the filters, one after the other. To run the pipeline, it calls `void pipeline::run(size_t max_number_of_live_tokens)` passing in eight tokens.

As we should expect when we run this example, the `BadWriteFilter wf` sometimes executes on a thread other than the master, so we see the output

```
Error!
Done.
```

While this example may appear contrived, remember that we are trying to mimic real situations when execution on a specific thread is required. In this spirit, let's assume that we cannot simply make the `ofstream` accessible to all of the threads, but instead we must do the writes on the main thread.

```
class SourceFilter : public tbb::filter {
public:
  SourceFilter(size_t n);
  void *operator() (void *) override;
private:
  size_t numItems;
};

class MultiplyFilter : public tbb::filter {
public:
  MultiplyFilter();
  void *operator() (void *v) override;
};

thread_local std::ofstream output;

class BadWriteFilter : public tbb::filter {
public:
  BadWriteFilter()
    : tbb::filter(tbb::filter::serial_in_order),
      issued_error(false) { }
  void *operator() (void *v) override {
    if (output.is_open()) {
       output << reinterpret_cast<size_t>(v) << std::endl;
    } else if (!issued_error) {
      std::cerr << "Error!" << std::endl;
      issued_error = true;
    }
  }
private:
  bool issued_error;
};

void fig_16_25() {
  output.open("output.txt", std::ofstream::out);

  SourceFilter sf(100);
  MultiplyFilter mf;
  BadWriteFilter wf;
  tbb::pipeline p;
  p.add_filter(sf);
  p.add_filter(mf);
  p.add_filter(wf);
  p.run(tbb::task_scheduler_init::default_num_threads());
  std::cout << "Done." << std::endl;
}
```

Figure 16-25. *A buggy example that fails if the* `BadWriteFilter` *tries to write to* `output` *from a worker thread*

Figure 16-26 shows how we can use a thread_bound_filter to work around this limitation. To do so, we create a filter class, ThreadBoundWriteFilter, that inherits from thread_bound_filter. In fact, other than changing what the class inherits from, the implementation of the filter class is the same as BadWriteFilter.

While the classes implementations are similar, our use of the filter must change significantly as shown in function fig_16_26. We now run the pipeline from a separate thread – we need to do this, because we must keep the main thread available to service the thread-bound filter. We also add a while-loop that repeatedly calls the process_item function on our ThreadBoundWriteFilter object. It is here that the filter is executed. The while-loop continues until a call to process_item returns tbb::thread_bound_filter::end_of_stream indicating that there are no more items to process.

Running the example in Figure 16-26, we see that we have fixed our problem:

```
Done.
```

```
thread_local std::ofstream output;

class ThreadBoundWriteFilter : public tbb::thread_bound_filter {
  bool issued_error;
public:
  ThreadBoundWriteFilter()
    : tbb::thread_bound_filter(tbb::filter::serial_in_order) { }
  void *operator() (void *v) override {
    if (output.is_open()) {
       output << reinterpret_cast<size_t>(v) << std::endl;
    } else if (!issued_error) {
      std::cerr << "Error!" << std::endl;
      issued_error = true;
    }
  }
};

void fig_16_26() {
  output.open("output.txt", std::ofstream::out);

  SourceFilter sf(100);
  MultiplyFilter mf;
  ThreadBoundWriteFilter wf;

  tbb::pipeline p;
  p.add_filter(sf);
  p.add_filter(mf);
  p.add_filter(wf);

  std::thread t([&]() {
    p.run(tbb::task_scheduler_init::default_num_threads());
  });
  while (wf.process_item() !=
         tbb::thread_bound_filter::end_of_stream)
    continue;

  t.join();
  std::cout << "Done." << std::endl;
}
```

Figure 16-26. *An example that writes to `output` only from the master thread*

Summary

In this chapter, we delved deeper into the features that can be used to tune TBB algorithms. We formed our discussion around the three common concerns when tuning TBB applications: task granularity, available parallelism, and data locality.

For the loop algorithms, we focused on the blocked range types and the different Partitioner types. We found that we can use 1 microsecond as a general guide for how long tasks should execute to mitigate the overheads of task scheduling. This rough guideline holds true for both loop algorithms, like `parallel_for`, and also for the filter sizes in `parallel_pipeline`.

We discussed how the blocked range types can be used to control granularity but also to optimize for the memory hierarchy. We used `blocked_range2d` and a `simple_partitioner` to implement a cache-oblivious implementation of matrix transposition. We then showed how `affinity_partitioner` or `static_partitioner` can be used to replay the scheduling of range so that the same pieces of data are accessed repeatedly by the same threads. We showed that while `static_partitioner` is the best performing partitioner for well-balanced workloads when executing in batch mode, as soon as the load is imbalanced or the system is oversubscribed, it suffers from its inability to dynamically balance the load through work stealing. We then briefly revisited determinism, describing how `deterministic_parallel_reduce` can provide deterministic results, but only by forcing us to use a `simple_partitioner` and carefully choose a grainsize, or use a `static_partitioner` and sacrifice dynamic load balancing.

We next turned our attention to `parallel_pipeline` and how the number of filters, the execution modes, and the number of tokens impact performance. We discussed how balanced and imbalanced pipelines behave. Finally, we also noted that while TBB pipelines do not offer hooks for us to tune for cache affinity, it is designed to enable temporal locality by having threads follow items as they flow through a pipeline.

We concluded the chapter with some advanced topics, including how to create our own Range types and how to use a `thread_bound_filter`.

For More Information

For more information on cache-oblivious algorithms:

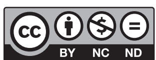
Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. 2012. Cache-Oblivious Algorithms. *ACM Trans. Algorithms* 8, 1, Article 4 (January 2012), 22 pages.

For a more in-depth discussion on pipeline parallelism:

> Angeles Navarro et al. "Analytical Modeling of Pipeline Parallelism," ACM-IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT'09). 2009.

For more information on the thundering herd problem:

> https://en.wikipedia.org/wiki/Thundering_herd_problem



Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.