Unit 4 – Multi-core applications

Threading Building Blocks

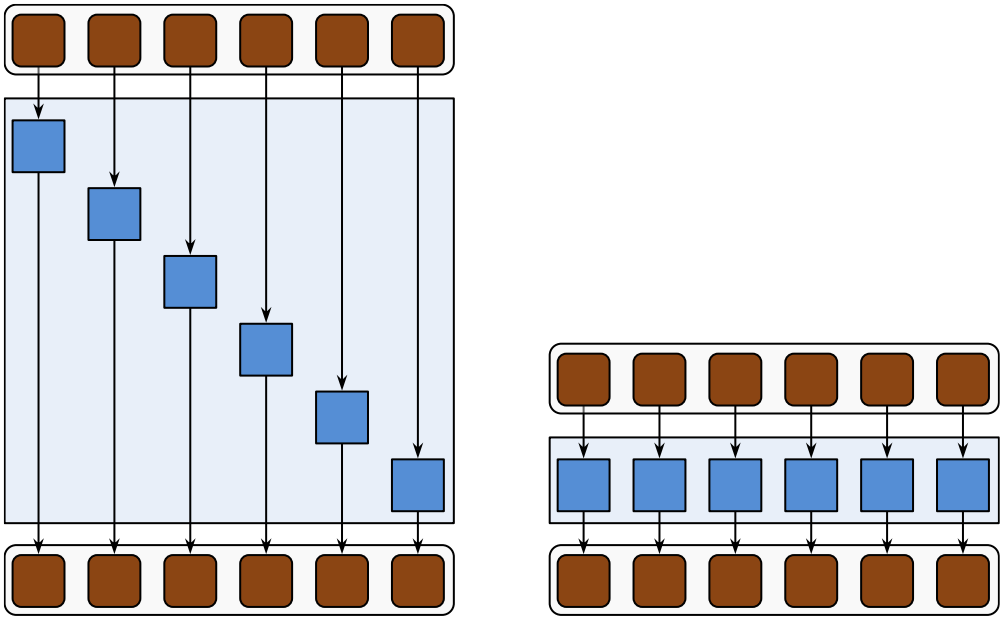
- C++ template library for parallel programming on multi-core processors. It helps to leverage **multi-core** performance.
- Computation broken down into *tasks* that can run in parallel. It manages and schedules threads to execute these tasks.
- Intel TBB® enables you to specify logical parallelism instead of threads. TBB run-time library automatically maps logical parallelism onto threads in a way that makes efficient use of resources.
- Used for shared-memory parallel programming and heterogeneous computing (intra-node distributed memory programming). Many parallel patterns can be implemented with TBB, e.g.: map, reduce, pipeline.
- The set of generic parallel algorithms available in TBB is shown in Table I. Template functions covered here: *parallel_for, parallel_invoke, parallel_reduce, parallel_pipeline*
- References:
  - ✓ Intel® Threading Building Blocks – Handbook. Intel® TBB is now called Intel oneAPI TBB (oneTBB).
  - ✓ M. McCool, A. Robison, J. Reinders, "Structured Parallel Programming: Patterns for Efficient Computation"
  - ✓ M. Voss, R. Asenjo, J. Reindeers, "Pro TBB: C++ Parallel Programming with Thread Building Blocks".

TABLE I. GENERIC ALGORITHMS IN THE TBB LIBRARY

Category	Generic Algorithm	Brief Description
Functional parallelism	`parallel_invoke`	Evaluates several functions in parallel
Simple loops	`parallel_for`	Map pattern over a range of values
	`parallel_for_each`	Map pattern over an iterator (parallel_do w/o work feeder)
	`parallel_reduce`	Reduction pattern over a range of values
	`parallel_deterministic_reduce`	Reduction pattern over a range of value with deterministic split/join behavior
	`parallel_scan`	Scan pattern (partial reductions) over a range of values
Complex loops	`parallel_do`	Workpile pattern: loop where the iteration space is unknown in advance and more iterations can be added before the loop exits.
Sorting	`parallel_sort`	Parallel sort of elements of a sequence
Pipeline	`pipeline`	Implementation of software pipeline
	`parallel_pipeline`	Strongly typed functions for pipelined execution

PARALLEL_FOR

- This template function allows us to implement a **map** pattern. Fig. 1(a) depicts the serial execution of a loop. Though there are no dependencies between loop iterations, we still need to run the iterations sequentially. Fig. 1(b) depicts the map pattern, where a function is applied to all elements of a collection, usually producing a new collection with the same shape as the input. Here, we can execute all iterations in parallel given enough processors.



(a) (b)

Figure 1. (a) serial loop execution. (b) map pattern: a function is applied to all elements of a collection. Rounded rectangles represent data. Rectangles represent tasks.

- The map pattern replicates a function over every element of an index set. The function must have no side-effects in order for the map to be implementable in parallel while achieving deterministic results. Also, it must not modify global data that other instances of the function depend on. The map pattern can replace a serial loop where:
  - ✓ Every iteration is independent.
  - ✓ The number of iterations is known in advance.
  - ✓ Every computation depends only on the iteration count and data read using the iteration count as an index into a collection.

TBB SYNTAX

- For example: We want to apply a function `Fun` to every element of an array. We can either:
  - ✓ Update `a[i]` itself. Example: `Fun(a[i])`: $a[i] \leftarrow a[i] \times (a[i]+1)$
  - ✓ Define another array (e.g. `b[i]`) onto which we place the results.

- Sequential implementation (the iteration space is of type `size_t` and goes from `0` to `n-1`)

```
void SerialApplyFun( float a[], size_t n ) {
    for( size_t i=0; i!=n; ++i )
        a[i] = a[i]*(a[i]+1); // Fun(a[i])
}
```

  - ✓ **body**': For a function, this is the code enclosed within the curly brackets: `{ … }`.

- Concurrent Implementation: The template function *parallel_for* breaks the iteration space (`0` to `n-1`) into chunks and launches the operations for each chunk on a separate thread. Put it another way, *parallel_for* divides up the iterations into tasks and provides them to the Task Scheduler for parallel execution.

- To specify the iteration space and the chunks, we use the `blocked_range` template class provided by the library. It is a one-dimensional iteration space over the specified type.
  - ✓ `blocked_range <type> (i,j, grain_size)`: Half-open range: `[i,j)`. `type`: *size_t, int, etc*. `grain_size = 1` by default. For example:
    - `blocked_range<int>(0,5)`: Range `[0,5)` with grain size of 1 ≡ `[0 1 2 3 4]`
    - `blocked_range<int>(5,14,2)`: Range `[5,14)` with grain size of 2 ≡ `[ 5 7 9 13]`

METHOD WITH A CLASS

- Here, we use a class to define the operation applied to every element. Then, we call *parallel_for*.
  - ✓ Class:

```
class ApplyFun {
    float *const my_a; // 'private' access (default access level)
public:
    void operator()( const blocked_range<size_t> &r ) const {
        float *a = my_a;
        for( size_t i=r.begin(); i!=r.end(); ++i )
           a[i] = a[i]*(a[i]+ 1) // Fun(a[i]);
    }
    ApplyFun( float a[] ) : my_a(a) {}
};
```

    - The body of the `SerialApplyFun` function was converted into a form (`ApplyFun`) that operates on a chunk. This form is a STL (standard template library)-style function object, called the ***body object*** (or ***body***), in which `operator()` processes a chunk.
    - **Function applied to every element**: $f(a[i]) = a[i] * (a[i]+1)$.
    - Range argument to loop template: `const blocked_range<type> &r, const blocked_range<type> r`. This defines `&r` and `r` as a blocked range of 'type'. The actual range boundaries are not defined here.
    - `operator()` loads `my_a` into local variable `a`. Though not necessary, we do this because:
      - Style: it makes the loop body look more like the original.
      - Performance: The compiler may optimize better if we put frequently accessed values into local variables.

  - ✓ Using *parallel_for*: Here, we embed *parallel_for* in a function, but this is optional.

```
Using namespace tbb;
void ParallelApplyFun( float a[], size_t n ) {
    parallel_for(blocked_range<size_t>(0,n), ApplyFun(a) );
}
```

    - Iteration space: `blocked_range<size_t>(0,n)`. This is a half-open range `[0,n) = [0,n-1]`.

- TBB *parallel_for* recursively (range splits into two subranges, each of which can split into two subranges, and so on) splits the range `[0,n)` into subranges and makes copies of the body (`ApplyFun`) for each of these subranges (argument `r` in `operator()`). For each such body/subrange pair, it invokes `operator()`: each subrange `r` is processed by the sequential

loop. The range and subranges are implemented as `blocked_range` objects. When worker threads are available, *parallel_for* may execute iterations in non-deterministic order. Fig. 2 shows an example of execution.

- An instance of `ApplyFun` needs member fields that remember all the local variables that were defined outside the original (large) loop but used inside it. These fields will be usually initialized by the constructor for the body object.

- TBB *parallel_for* requires the body object to have a copy constructor, which is invoked to create a separate copy (or copies) for each worker thread. It also invokes a destructor to destroy these copies. In most cases, the implicitly generated copy constructor and destructor work correctly.
  - ✓ Since the body object may be copied, the `operator()` should not modify the body, otherwise the modification may or may not become visible to the original thread, depending upon whether `operator()` is acting on the original thread or a copy. Hence, the body object's `operator()` must be declared `const` (so it can't modify the object on which it is called)

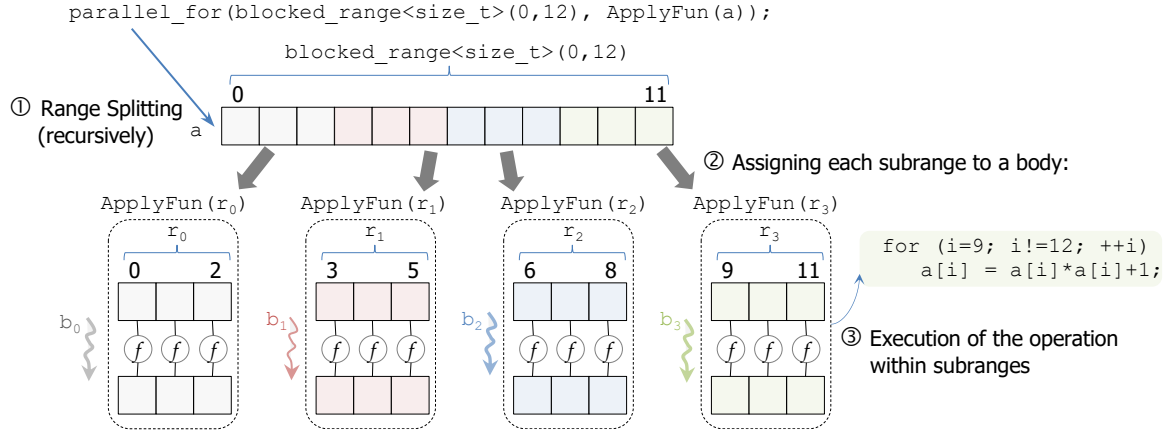


Figure 2. *parallel_for* execution example for `parallel_for(blocked_range<size_t>(0,n), ApplyFun(a))` for n=12. Here, the iteration space is partitioned into 4 chunks (subranges) to be executed concurrently. Each chunk is processed by a body (copies are made as needed for each chunk). Within each chunk, the processing is sequential.

LAMBDA EXPRESSIONS

- Available in the Version 11.0 of the Intel® C++ Compiler. They are very useful when using libraries like TBB to specify the user code to execute a task. They are used to create anonymous function objects.
- Basic syntax: `[capture-list] (params) -> ret { body}`
  - ✓ `capture-list`: comma-separated list used to make the variables outside the lambda expression accessible inside the lambda expression, via copy or reference.
    - We capture a variable by value by listing the variable name in the capture-list.
    - We capture a value by reference by prefixing with &. And we can use this to capture the current object by reference.
    - There are also defaults:
      - `[=]`: captures all automatic variables used in the body by value and the current object by reference.
      - `[&]`: captures all automatic variables used in the body and the current object by reference
      - `[]`: captures nothing. Allowed in some circumstances.
  - ✓ `params`: list of function parameters (optional), just like for a named function. If no function parameters use `()` or omit.
  - ✓ `ret`: return type. If `->ret` is not specified, it is inferred from the return statements
  - ✓ `body`: function body

- Examples:
  - ✓ `[i, &j] (int k0, int &l0) -> int { j=2*j; k0 = 2*k0; l0 = 2*l0; return i+j+k0+l0; };`
    - It captures `i` by value, `j` by reference. It has a parameter `k0`, and another parameter `l0` that is received by reference.
    - We can think of a lambda expression as an instance of a function object, but the compiler creates the class definition for us. The lambda expression is analogous to an instance of this class:

```
class Functor {
   int my_i;
   int &my_jRef;
public:
   Functor (int i, int &j): my_i {i}, my_jRef{j} {}
   int operator () (int k0, int &l0) {
      myjRef = 2 * my_jRef; k0 = 2 * k0; l0 = 2*l0;
      return my_i + my_jRef + k0 + l0;
   }
};
```

  - ✓ `[&] (float x) -> float { return x++; }`
  - ✓ `[] () { cout << "This is a lambda expression" << endl; }`
  - ✓ `[] { funct (parameters) }`
    - We can invoke function `funct` and specify its parameters. Note that this lambda expression captures nothing, has no parameters (`()` could have been used but was omitted) nor return type.

- Whenever we use a C++ lambda expression, we can substitute it with an instance of a function object. C++ lambda expressions simplify the use of TBB by eliminating the need of defining a class for each use of a TBB algorithm.

- This makes *parallel_for* much easier to use as it lets the compiler do the tedious work of creating the function object.

  ✓ Normal lambda expression: It replaces both the declaration and construction of the function object `ApplyFun` in the previous example: only one call to *parallel_for* is required.

```
parallel_for( blocked_range<size_t>(0,n), [&](const blocked_range<size_t> r) {
   for(int i=r.begin(); i!=r.end(); ++i) // 0 <= i < n
      a[i] = a[i]*(a[i]+1); // Fun (a[i])
  } );
```

  - The lambda expression creates a function object very similar to `ApplyFun`.

  ✓ Compact lambda expression: TBB has a form of *parallel_for* expressly for parallel looping over a consecutive range of integers (*parallel_for (first, last, step, f)* ≡ *for (i=first; i< last; i+=step) f(i)*). The *step* parameter is optional.

```
parallel_for(size_t(0), size_t(n), [&] (size_t i) { // 0 <= i < n
   a[i] = a[i]*(a[i]+1); // Fun(a[i]);
 } );
```

RACE CONDITIONS

- *parallel_for* assumes that the body of the loop is thread-safe, i.e., it does not have race conditions. It is then important to ensure that variables inside loop only depend on the index of the loop. Otherwise the threads might interact with each other updating variables at the wrong time.

- Here, we show how to do create thread-safe implementations when every iteration in the loop is independent and every computation depends on the iteration index and data read using that index. Otherwise, you need to use advanced synchronization mechanisms (e.g: atomic operations, mutual exclusions).

- For example, we want to apply the following operation to 100-element vector $\vec{v}$ of type `int`. The result $\overrightarrow{vo}$ should also be of type `int`.

$$vo[i] = round\left(\left(\frac{v[i]}{256}\right)^{0.7}\right) \times 256$$

- This is a straightforward operation; the following is a typical sequential implementation:

```
...
double tmp, aux;
int *vi, *vo;
vi = (int *) calloc (100,sizeof(int));
vo = (int *) calloc (100,sizeof(int));
...
for (i = 0; i < 100; i++) {
   tmp = ( (double) vi[i]) /256;
   aux = pow(tmp,0.7)*256;
   vo[i] = (int) (aux + 0.5); // Rounding + Saturation
}
...
```

  ✓ In this simple operation, we use temporal variables `tmp` and `aux` in order to make the code more readable.

- Using TBB, we can replace the for loop with *parallel_for* (compact lambda expression):

```
tbb::parallel_for (int(0), int(100), [&] (int i) { // 0 <= i < 100
          tmp = ( (double) vi[i]) /256;
          aux = pow(tmp,0.7)*256;
          vo[i] = (int) (aux + 0.5);
        } );
```

  ✓ The code inside the loop is not thread-safe. The threads interact causing `tmp` and `aux` to be updated by other threads that are not associated with the corresponding thread. This might cause race conditions.

    - Note that the race condition can be a rare occurrence. In this example, we found race conditions for large vector sizes (> 10,000) and it only affected a few data points. These race conditions can be very difficult to spot.

Thread-safe implementations

- First approach: we declare `tmp` and `aux` as vectors that depend on the iteration index. This way, every thread will only access its respective `tmp[i]` and `aux[i]`.

```
double *tmp, *aux;
tmp = (int *) calloc (100,sizeof(int));
aux = (int *) calloc (100,sizeof(int));

tbb::parallel_for (int(0), int(100), [&] (int i) { // 0 <= i < 100
          tmp[i] = ( (double) vi[i]) /256;
          aux[i] = pow(tmp[i],0.7)*256;
          vo[i] = (int) (aux[i] + 0.5);
        } );
```

  - ✓ While this approach works, it is inefficient if the operation inside the loop is more complex (like requiring extra loops and conditions).

- Second approach (using functions): This is the recommended approach, where we encapsulate the function `Fun[i]` (applied to every element of the array) in a function.

```
int Fun (int *di, int k) {
  double tmp, aux, result;
    tmp = ( (double) di[i]) /256;
    aux = pow(tmp,0.7)*256;
    result = (int) (aux + 0.5);
    return result;
}
...
tbb::parallel_for (int(0), int(100), [&] (int i) { // 0 <= i < 100
          vo[i] = Fun(vi, i);
       } );
```

  - ✓ Note that every iteration must be independent of each other. Every computation must depend only on the iteration index and data read using the iteration index as an index into the collection.

PARALLEL_INVOKE

- Perhaps the simplest algorithm provided by the TBB library. This template function allows us to implement a **map** pattern.
- *parallel_invoke* executes a list of (2 to 10) tasks in parallel and waits for all tasks to complete. This is different than *parallel_for*.
- To execute functors $f_0, f_1, f_2, \ldots, f_9$, the syntax is:

```
parallel_invoke(const Func0& f0, const Func1&f1, ..., const Func9& f9);
```

  - ✓ Each argument must have a type for which the `operator()` is defined.
  - ✓ Note that the arguments are usually function objects (functors), though they can also be pointers to functions or lambda expressions.

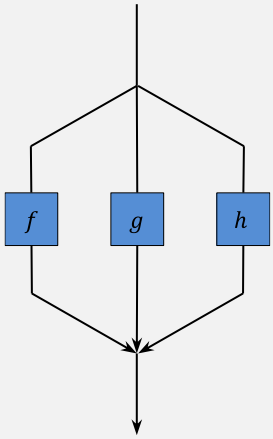
- Basic example with lambda expressions:

```
int main () {
   parallel_invoke (
           [] () { cout << "Hello " << endl;},
           [] () { cout << "TBB! " << endl;}
   );
   return 0;
}
```

  - ✓ Lambda expressions avoid creating function objects (functors) when using *parallel_invoke*.
    - We use lambda expressions to specify the functions: this can include expressions and calls to functions.
  - ✓ Note that the resulting output may contain either `Hello` or `TBB!` First. There might not even be newline character between the two strings and two consecutive headlines at the end of the output.

- Example with functors, function pointers, and lambda expressions:

```
void bar (int a) {
  int t;
  t = a*a*a;
  cout << "(bar) a^3 = "<< t << "\n";
}

class MyFunctor {
   int arg;
public:
   MyFunctor(int a): arg(a) {}
   void operator() () const { bar(arg);}
};

void f () {
  cout << "(function ) executed!\n";
}

int main () {
   MyFunctor g(2);
   MyFunctor h(3);

   // f,g,h evaluated in parallel
   parallel_invoke(f,g,h); // f: pointer to function. g,h: functors

   // f and bar(1) evaluated in parallel
   parallel_invoke(f, []{ bar(1); }); // lambda expression (no need to create function object)
   return 0;
}
```



  - ✓ `parallel_invoke(f,g,h)`: If the three function invocations execute for roughly the same amount of time and there are no resource constraints, this parallel implementation can be completed in a third of the time it takes to sequentially invoke the functions one after the other.
  - ✓ `parallel_invoke(f, []{ bar(1); })`: We can use lambda expressions to avoid creating function objects. We could also do: `parallel_invoke (f,g,h, []{ bar(1); }).`

  - ✓ Recall that it is the responsibility of the developer to invoke functions in parallel only when they can be safely executed in parallel. TBB will not automatically identify dependencies and apply synchronization and other parallelization strategies to make the code safe.

PARALLEL_REDUCE

- This template function allows us to implement a **reduction** pattern. A reduction combines every element in a collection into a single element using an associative combiner function.
- A reduction can be implemented as a serial loop, where there is data dependency, as depicted in Fig. 3(a). However, Fig. 3(b) shows how a reduction can be parallelized using a tree structure. Note that the tree parallelization of the reduction is implemented using the same number of operations as the serial version. A very common example of a reduction is the accumulation of all elements in a collection.
- Reductions can use operations other than accumulation, such as maximum, minimum, multiplication, Boolean operations.

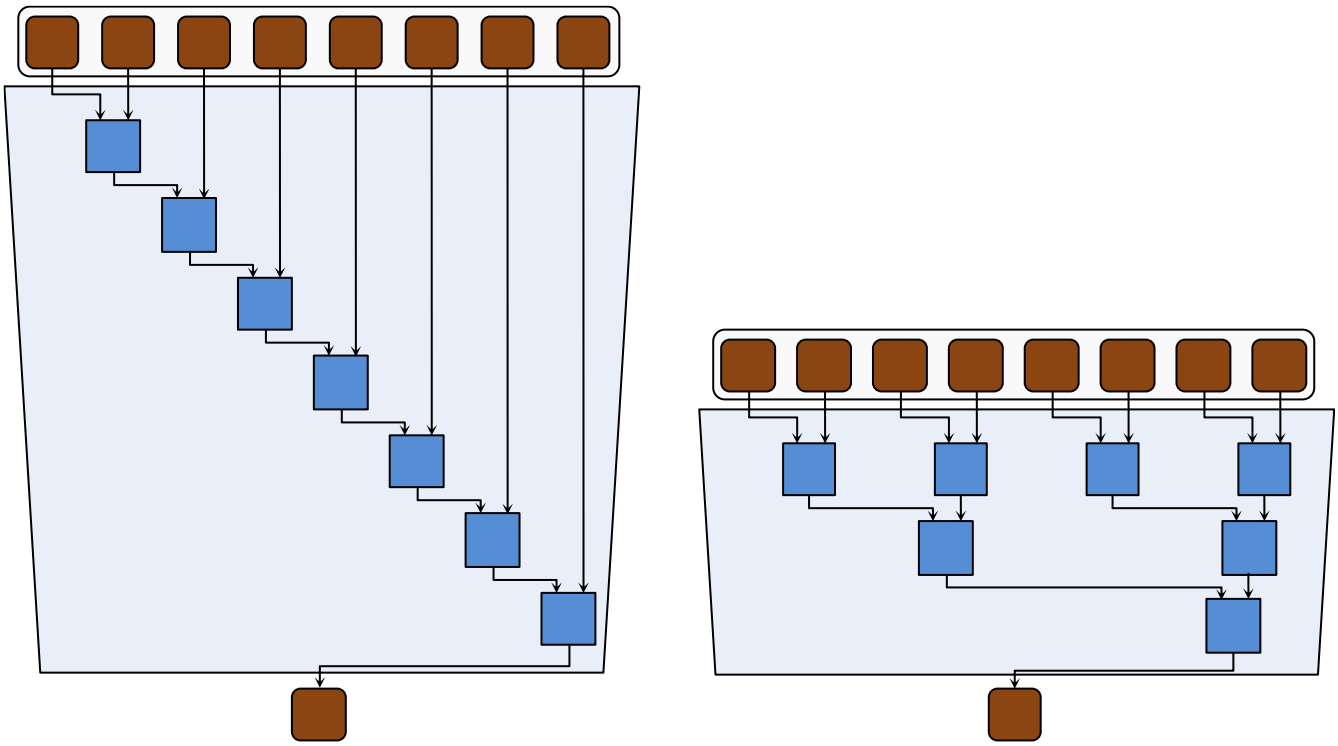


Figure 3. Reduction. (a) serial implementation. (b) parallel implementation.

TBB SYNTAX

- Example: Applying the reduction operation (accumulation of cubes). $\sum_{i=0}^{n-1} f(a[i])$, $f(a[i]) = a[i] \times a[i] \times a[i]$.
  - ✓ Using *parallel_reduce*, where we indicate the iteration space, as well as the object:

```
float ParallelSumFun( float a[], size_t n ) {
    SumFun sf(a); // Object 'sf' created with argument a
    parallel_reduce(blocked_range<size_t>(0,n), sf );
    return sf.my_sum; }
```

    - The reduction is applied to range `[0,n)` for object `sf`.

  - ✓ The class `SumFun` specifies the details of the reduction (e.g.: how to accumulate subsums and combine them):

```
class SumFun {
    float * my_a; // 'private' access (default access level)
public:
    float my_sum;

    void operator()( const blocked_range<size_t> &r ) {
        float *a = my_a;
        float sum = my_sum;

        for ( size_t i=r.begin(); i!=r.end(); ++i )
           sum += a[i]*a[i]*a[i]; // Associative combiner function
        my_sum = sum;
    }
    SumFun (SumFun &x, split): my_a(x.my_a), my_sum(0) {} // my_a = x.my_a, my_sum = 0
    void join (const SumFun &y) { my_sum += y.my_sum; }
    SumFun (float a[]): my_a(a), my_sum(0) {} // my_a = a, my_sum = 0
};
```

    - `SumFun`: the `operator()` is not const. This is because we need to be able to update `my_sum`.

- The class `SumFun` has a splitting constructor and a join method that must be present for *parallel_reduce* to work.
  - Splitting constructor: `SumFun (SumFun &x, split): my_a(x.my_a), my_sum(0) {}`. This represents the splitting of body `x` into `x` a newly constructed body `y`: `SumFun y (x, split)`. We fork a body (function object) to allow concurrent execution. Arguments: a reference to the original object (`x`), and an argument of type `split` (defined by the library to distinguish the splitting constructor from a copy constructor).
  - Join method: `void join (const SumFun &y) { my_sum += y.my_sum; }`. This represents the merging of the results of bodies `x` and `y`: `x.join(y)`. Body name `y`: declared here, it is implicitly used in the splitting constructor.
- **Function applied to every element**: $f(a[i]) = a[i] \times a[i] \times a[i]$.
- **Associative Combiner function**: Scalar addition

TBB parallel_reduce operation

- TBB *parallel_reduce* first recursively splits the range `[0,n)` into subranges.
- Then, it assigns each subrange to a body: it recursively attempts to split a body.
  - ✓ If worker threads are available, *parallel_reduce* invokes the splitting constructor for a body. For each such split of the body, it invokes the `join` method to merge the result of the bodies.
    - Fig. 4(a) depicts the split of a body `x` into `x` and a new body `y` (iteration range is first split into two subranges), where each body performs the reduction of a subrange, and then `join` is used to merge the results of the two bodies.
  - ✓ If worker threads are not available, *parallel_reduce* does not invoke the splitting constructor for a body. For a range divided into two subranges, the second subrange is reduced using the same body that reduced the first subrange.
    - Fig. 4(b) depicts this case, where the same body performs the reduction of the two subranges.

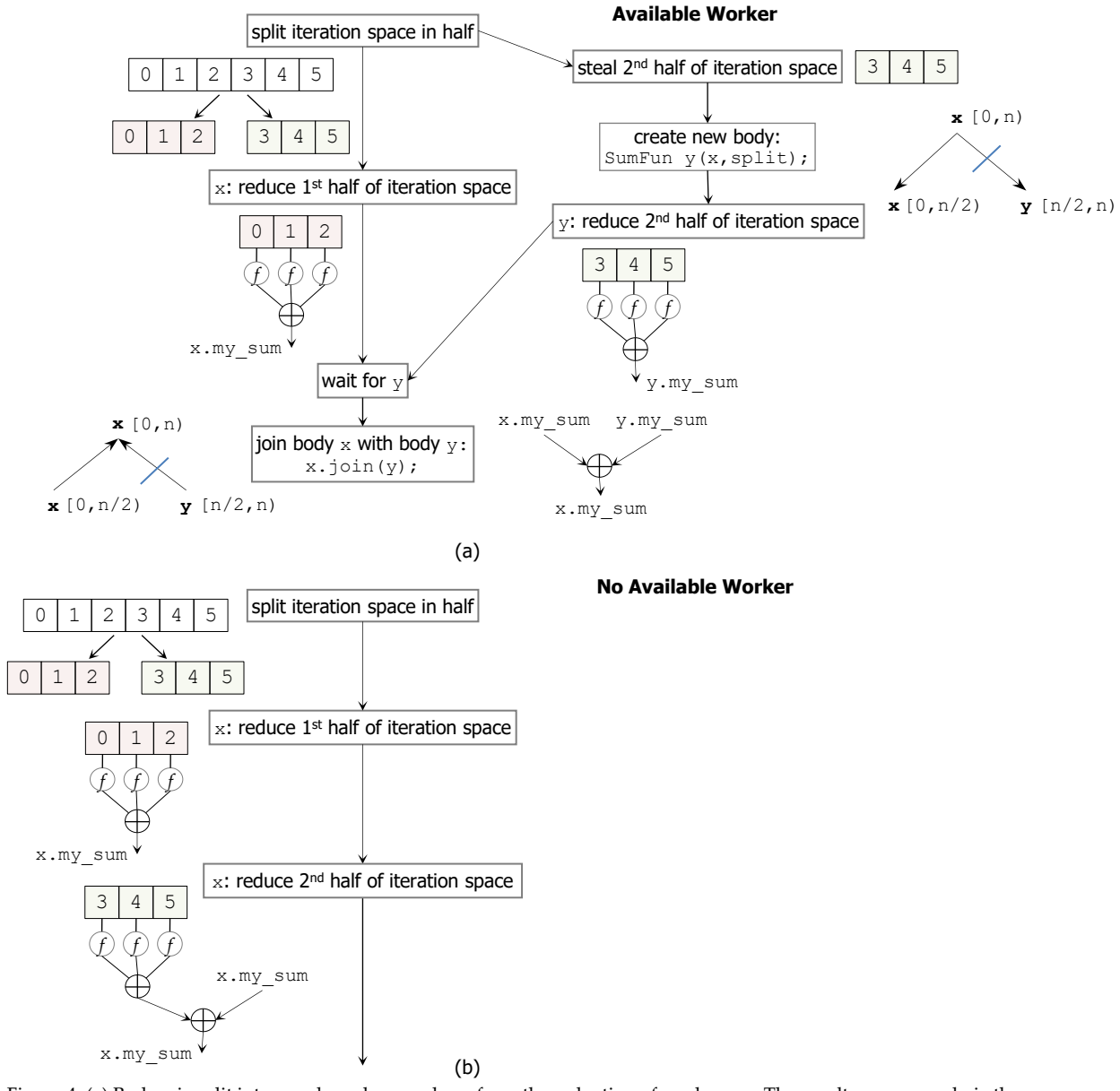


Figure 4. (a) Body x is split into x and y, where each perform the reduction of a subrange. The results are merged via the merge method. (b) Body x is not split: it processes both subranges; thus the operator() in the body cannot discard earlier accumulations.

- With the subranges assigned to bodies (not necessarily all different bodies), the reduction for each subrange is computed. Finally, the `join` method (if applicable) is used recursively until the final result is computed.
  - ✓ If each subrange is assigned to a different body, then all reductions are executed concurrently.
  - ✓ For a range that is divided recursively into many subranges, when assigning each subrange to a body, we may have the following cases along the way:
    - A body splits. Each of the resulting bodies may or may not split.
    - A body did not split (the same body must process two subranges). Each time the body operates on a subrange, it may split (or not) into two bodies.
  - ✓ It is possible that each range is assigned to the same body. Here, *parallel_reduce* runs sequentially from left to right. Sequential execution never invokes the splitting constructor or method join.

- Note that *parallel_reduce* may copy a body while the body's `operator()` or method `join` runs concurrently.

- Fig. 5 depicts a *parallel_reduce* operation example. The range is recursively split at each level into two subranges, until we are left with four subranges (r_0, r_1, r_2, r_3). Then, we assign each subrange to a body. Here, the Task Scheduler lets us have a different body (b_0, b_1, b_2, b_3) for each subrange. Then, all bodies concurrently execute their reduction. The `join` method is used recursively to merge the results of two bodies: first, we merge two sets of bodies (b_0 and b_1 into b_0, b_2 and b_3 into b_2), then we merge the result of two bodies (b_0 and b_2 into b_0).

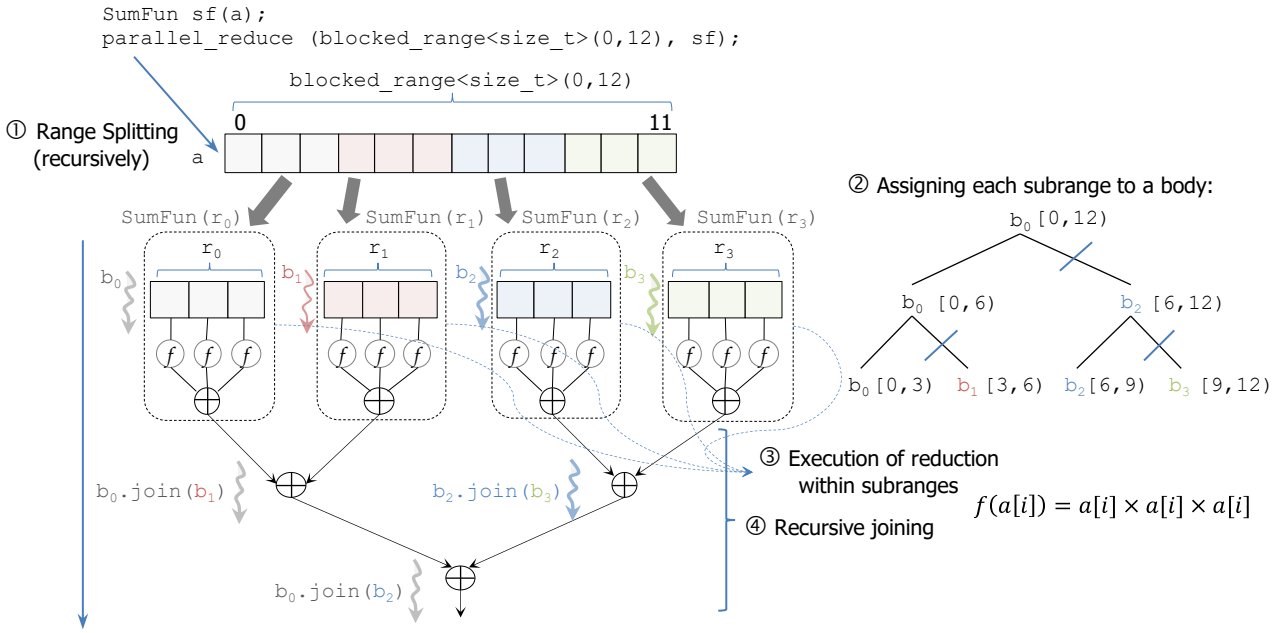


Figure 5. *parallel_reduce* execution example for range [0,12). The iteration space is partitioned into 4 chunks. Each chunk is processed by a different body. Within each chunk, the processing is sequential. Partial results (from different bodies) are merged recursively.

- Fig. 6 depicts a *parallel_reduce* operation example, where not all subranges can be assigned to a different body. Here, the Task Scheduler lets us have a different body (b_0, b_1, b_2) for subranges r_0, r_1, r_2, but b_2 is assigned to r_3 as well. Then, only b_0, b_1, b_2 can execute concurrently. The `join` method is used recursively to merge the results of two bodies: first, we merge b_0 and b_1 into b_0, then we merge b_0 and b_2 into b_0. Here, b_2 reduces two subranges (r_2, r_3) sequentially from left to right: b_2 reduces r_2 first, then b_2 reduces r_3.
  - ✓ Because the same body (b_2) is used to merge multiple subranges, the `operator()` cannot discard earlier accumulations.
    - Initializing `sum = 0` in `operator()` is incorrect, as the body would return a partial sum only for the last subrange.
  - ✓ Instead, `my_sum = sum` ensures that the b_2 returns the accumulated sum of all the subranges that b_2 reduced. This is illustrated in Fig. 6, where when processing subrange r_3, our initial sum is the partial sum of subrange r_2.

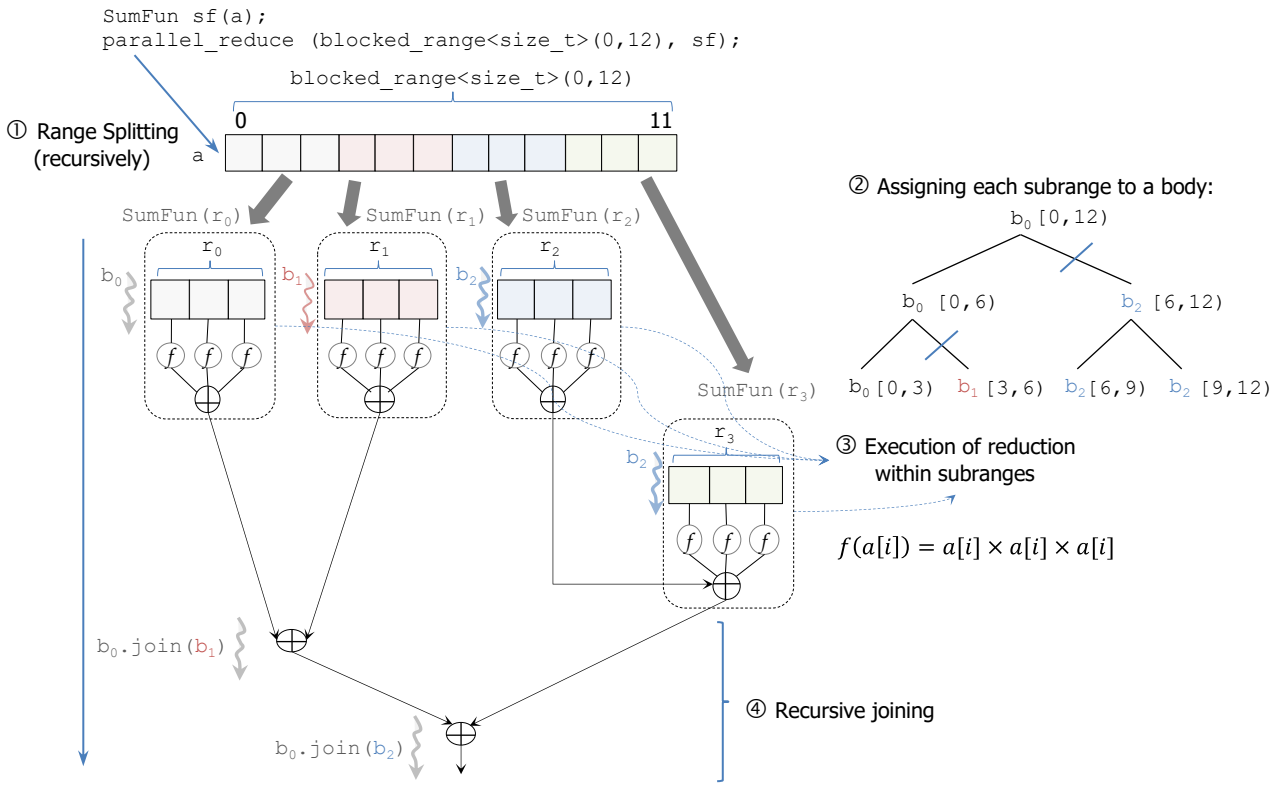


Figure 6. *parallel_reduce* execution example for range [0,12). Here, the iteration space is partitioned into 4 chunks. Only three chunks are processed by different bodies. Body `b2` executes the reduction of `r2` and `r3` sequentially, just accumulating the partial results.

TBB parallel reduce as map and reduce

- In our application example $\sum_{i=0}^{n-1} f(a[i])$, $f(a[i]) = a[i] \times a[i] \times a[i]$, a function f is applied to every element $a[i]$, and then the resulting $f(a[i])$ are added up. This operation could be implemented as:
  - ✓ Using *parallel_for* (map pattern) to compute $f(a[i])$ for $i = 0, \dots n-1$.
  - ✓ Using *parallel_reduce* (reduction pattern) to compute $\sum_{i=0}^{n-1} f(a[i])$.
- In our implementation, we only used *parallel_reduce* where we effectively computed each $f(a[i])$ and $\sum_{i=0}^{n-1} f(a[i])$. This effectively implements the map and reduction pattern: when the subranges are computed concurrently, the computation of the $f(a[i])$ in different subranges is also concurrent. Note that within a subrange, the computation of $f(a[i])$ is sequential, as it would be the case in a subrange created by *parallel_for*.
- Either approach would work similarly, but note that launching only *parallel_reduce* may be more optimal (time, resources) than launching both *parallel_for* and *parallel_reduce*.

Example: body splitting for a different range:

- Fig. 7 shows different assigning of subranges to bodies when applying *parallel_reduce* over `blocked_range<int>(0,20,5)`. The smallest non-divisible subrange is a 5-element subrange. After range splitting, we ended up with four subranges. When assigning a subrange to a body, more bodies are created (split) depending on the availability of worker threads (the '/' mark denotes where copies of a body were created by the splitting constructor):
  - ✓ Fig. 7(a): Three bodies. `b0` splits into `b0` and `b2`. Then `b0` splits again into `b0` and `b0`. `b2` does not split. `b0` processes subrange `[0,5)`, `b1` processes `[5,10)`. Body `b2` processes subranges `[10,15)` and `[15,20)` (in that order: left to right). On the way back up the tree, *parallel_reduce* invokes `b0.join(b1)` and `b0.join(b2)` to merge the results of the leaves.
  - ✓ Fig. 7(b): Four bodies. This is similar to the case of Fig. 7(a), but `b2` does split into `b2` and `b3`.
  - ✓ Fig. 7(c): Two bodies. `b0` splits into `b0` and `b1`. `b0` processes the subranges `[0,5)`, `[10,15)`, `[15,20)`, while `b0` processes the subrange `[5,10)`. On the way back up the tree, `b0` merges with `b1`: `b0.join(b1)`. As `b0` processes the subranges (including merging with the result of `b1`), it accumulates the partial sums into a total sum.
  - ✓ Fig. 7(d): One body (no worker threads available). Here, *parallel_reduce* executes sequentially from left to right. There is no splitting or `join` calls. Here, b_0 evaluates each of the four subranges in left to right order.

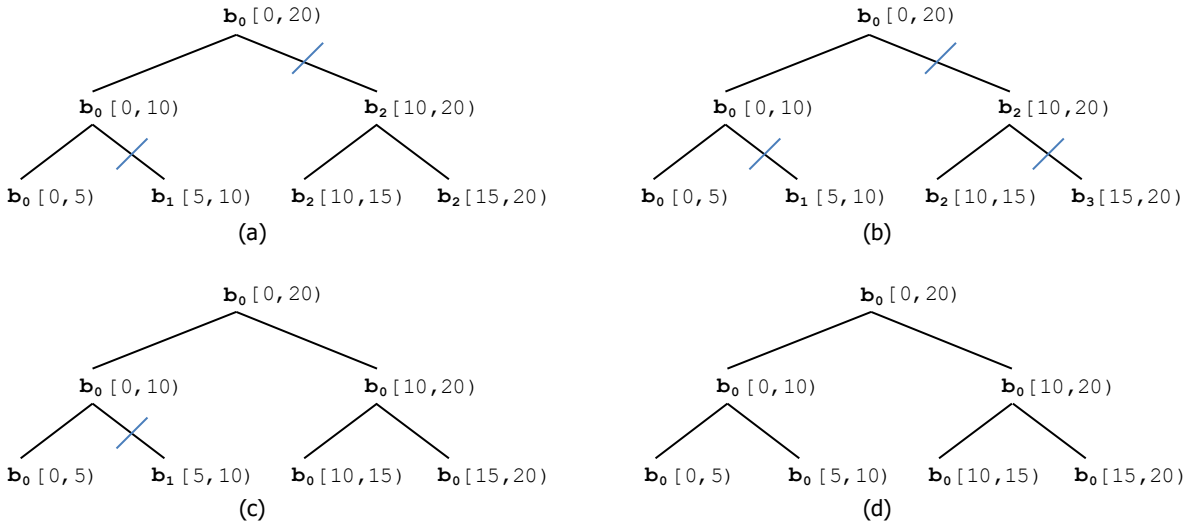


Figure 7. Sample executions of *parallel_reduce*. (a) 3 bodies: b_2 sequentially processes two subranges (b) 4 bodies: each subrange is processed concurrently. (c) 2 bodies. (d) 1 body: fully sequential operation over the four subranges.

PARALLEL_SCAN

- This template function implements a **scan** pattern in parallel. A scan operation (reduction with intermediate values) generates all partial reductions of an input sequence, resulting in a new output sequence.
- Table II shows the mathematical definition of the scan operation: let ⊗ be an associative operation with a constant element id. Input sequence: $z_0, z_1, \ldots z_{n-1}$. Output sequence: $y_0, y_1, \ldots, y_{n-1}$.

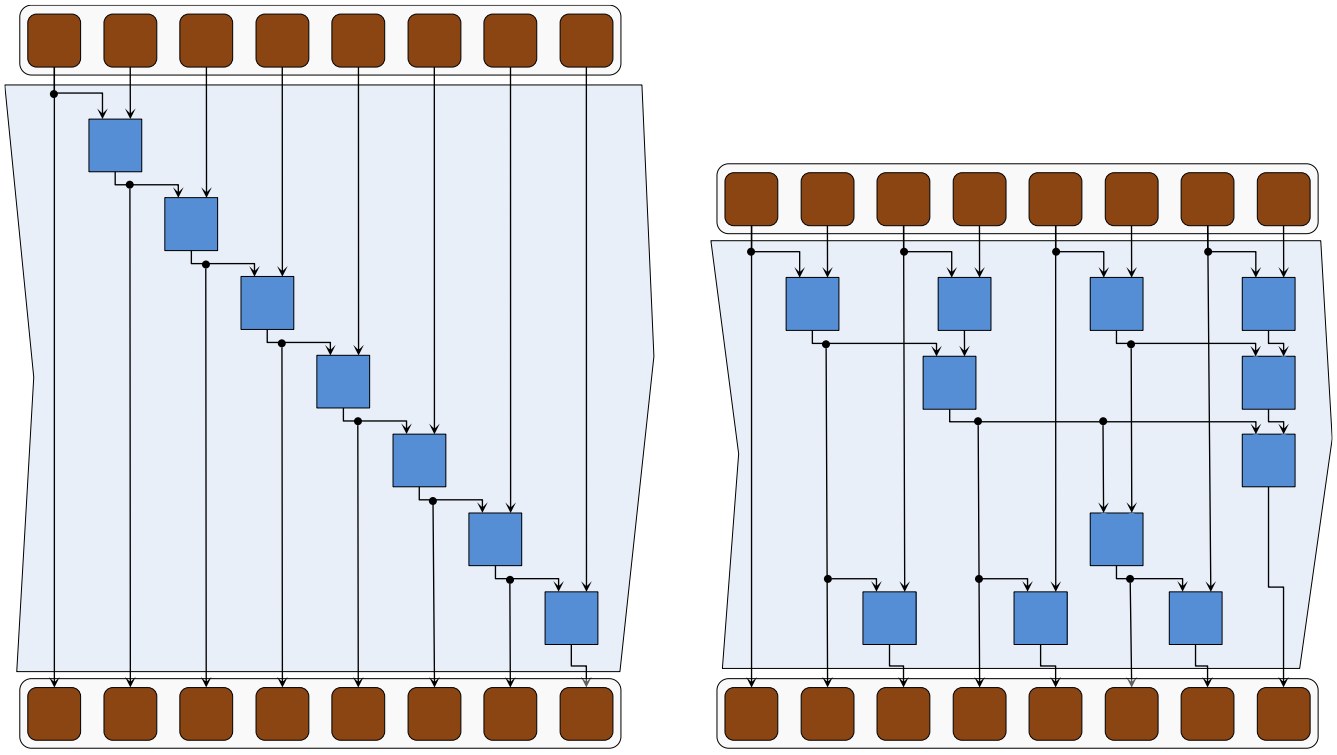
TABLE II. MATHEMATICAL DEFINITION OF SCAN OPERATION AND ITS SERIAL IMPLEMENTATION

Generic Scan Operation	Serial implementation

$y_0 = \text{id} \otimes z_0$
$y_1 = y_0 \otimes z_1$
…
$y_i = y_{i-1} \otimes z_i$
…
$y_{n-1} = y_{n-2} \otimes z_{n-1}$
	`size_t tmp; // or double tmp`
	
	`tmp = id;`
	`for i = 1:n`
	`  tmp = tmp ⊗ z[i]`
	`  y[i] = tmp;`
	`end`

- Despite the loop-carried dependency, the scan operation can be parallelized. Like reduction, we can take advantage of the associativity of the combiner function to reorder operations.
  - ✓ However, unlike reduction, parallelizing scan comes at the cost of redundant computations.

- Parallel scan is performed by reassociating the application of ⊗ and using two passes (it may invoke ⊗ up to twice as many times as the serial algorithm). Fig. 8(a) depicts the serial implementation, while Fig. 8(b) depicts one possible parallel implementation of the scan pattern.



(a) (b)

Figure 8. Scan pattern (a) serial implementation. (b) parallel implementation

TBB PARALLEL_SCAN

- The range is divided by the TBB library into chunks and TBB tasks are created to apply the body (scan) to these chunks.
  - ✓ *Prefixes*: intermediate results for each element in the range, i.e., `y[i]`.
- However, the scan body may be executed more than once on the same chunk of iterations: first in a *pre-scan* mode and then later in a *final-scan* mode. So, *TBB parallel_scan* involves two passes, of which the *pre-scan* pass is not always executed.
  - ✓ *pre-scan* mode: the body is passed a 'starting' (partial) prefix value for the element that precedes its subrange. It returns a partial (not yet final) prefix for the last element in its subrange. This is the result (also called *summary*) of the reduction. The prefixes `y[i]` are not updated.
  - ✓ *final-scan* mode: the body is passed an accurate (final) prefix value for the element that precedes its subrange. It returns the (final) prefixes for each iteration in is subrange (including the one for the last element, i.e., the results of the reduction). Scan results are computed and returned (i.e., the prefixes `y[i]` are updated)

- Example: Scan operation applied to summation, i.e., operator $\otimes \equiv +$.
  - ✓ The *parallel_scan* template indicates the iteration space, as well as the object:

```
void main () {
   size_t z[16] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
   size_t y[16], id = 0;
   int n = sizeof(z)/sizeof(z[0]);

   SumScan sf(y,z,id); // Object 'sf' created with arguments y, z, and id
   parallel_scan(blocked_range<size_t>(0,n), sf);
   printf ("Result: %ld\n", sf.sum); for (int i=0; i < n; i++) printf ("y[%d] = %ld\n",i,y[i]);
}
```

  - ✓ The class `SumScan` specifies the details of the *parallel_scan* (this is called the imperative form):

```
class SumScan {
    size_t id;
    size_t* y;
    const size_t* z;
public:
    size_t sum;
    SumScan(size_t y_[], const size_t z_[], size_t id_) : sum(id_), z(z_), y(y_), id(id_) {}

    template <typename Tag>
    void operator()( const blocked_range<size_t> &r, Tag) { // accumulate summary for range r
       size_t temp = sum;
       for( int i = r.begin(); i < r.end(); ++i ) {
           temp = temp + z[i];
           if( Tag::is_final_scan() ) // bool is_final_scan(): true for a final_scan_tag, else false
                y[i] = temp; // scan result
       }
       sum = temp; // summary: from final_scan or pre_scan
    }
    SumScan( SumScan& b, split ) : z(b.z), y(b.y), sum(id) {} // split constructor
    void reverse_join(SumScan& a) { sum = a.sum + sum; }
    void assign( SumScan& b ) { sum = b.sum; }
};
```

  - ✓ Fig. 9 depicts a possible execution of *parallel_scan*. Range z[0:15]: Split into 4 subranges; a body operates on a subrange.
    - First body b_0 is created and assigned the 1st subrange.
    - b_0 is split: b_0 and b_2 (assigned the 3rd subrange).
    - b_0 is split (again): b_0 and b_1 (assigned the 2nd subrange).
    - Body b_0 executes in *final_scan* mode. Result: y[0:3], b_0.sum=10.
    - Body b_1 executes in *pre_scan* mode. y[3]: inaccurate. Result: b_1.sum=26.
    - Body b_2 executes in *pre_scan* mode. y[7]: inaccurate. Result: b_2.sum=42.
    - Reverse join of b_1 and b_0 into b_1: add summaries of b_1 and b_0 $\Rightarrow$ b_1.sum=26+10=36.
    - Reverse join of b_2 and b_1 into b_2: add summaries of b_2 and b_1 $\Rightarrow$ b_2.sum=36+42=78.
    - b_0 assigned the 2nd subrange and executes in *final_scan* mode. y[3] (b_0.sum=10) is accurate. Result: y[4:7], sum=10+(5+6+7+8)=36.
    - b_1 assigned the 3rd subrange and executes in *final_scan* mode. y[7] (b_1.sum=36) is accurate. Result: y[8:11], sum=36+(9+10+11+12)=78.
    - b_2 assigned the 4th subrange and executes in *final_scan* mode. y[11] (b_2.sum=78) is accurate. Result: y[12:15], sum=78+ (13+14+15+16)=136.
    - The summary of b_2 is assigned to the summary of b_0 $\Rightarrow$ b_0.sum = 136. b_2 and b_1 are destroyed.

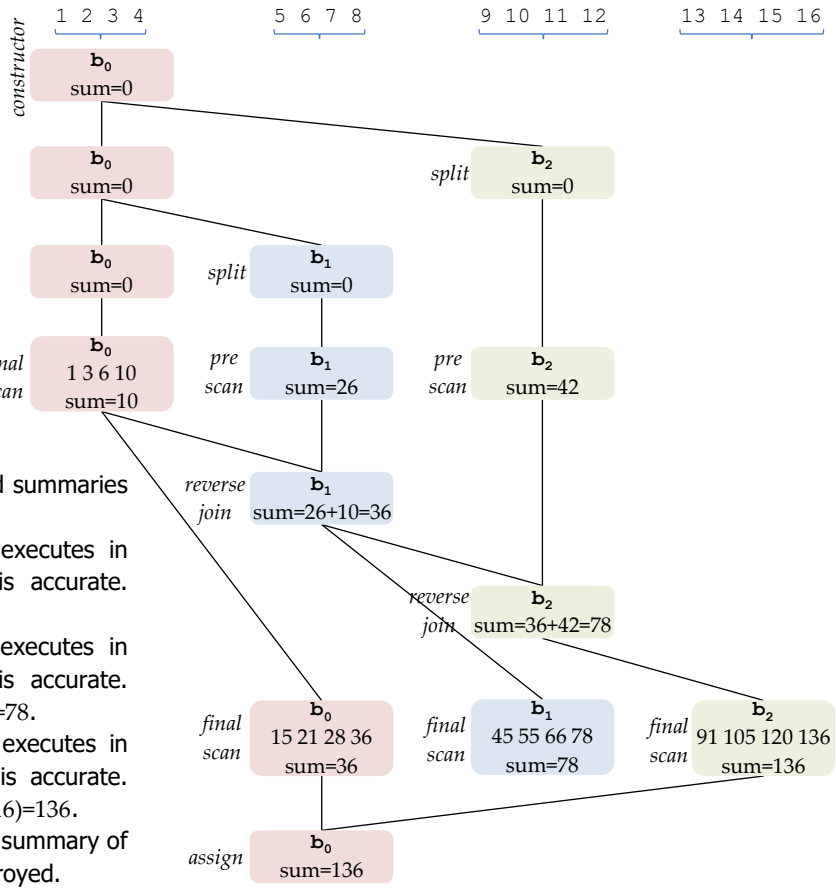


Figure 9. *parallel_scan* sample execution.

- ✓ Subranges are processed from left to right.
- ✓ *parallel_scan* is in charge of distributing the workload, when to create and execute a body, and when to whether use *final_scan* of *pre-scan* mode.
- ✓ `is_final_scan()`: It enables differentiation between *pre_scan* mode and *final_scan* mode.

- ✓ Body split and reverse join:
```
SumScan( SumScan& b, split ) : z(b.z), y(b.y), sum(id) {}
void reverse_join(SumScan& a) { sum = a.sum + sum; }
```
  - Split constructor: It specifies that body `b` is split into body `b` and a new body `a` (this name `a` is declared in the reverse join method). The new body has the same input data (`z, y, id`). The new body is assigned a different subrange.
  - Reverse join method: the results of bodies `b` and `a` are merged into body `a` (this is the reverse of `join` in *parallel_reduce*). It is only the results that are merged, not the bodies.

- ✓ Assign summary of `b` to the current object: `void assign( SumScan& b ) { sum = b.sum; }`
  - The summary of the last subrange (being acted upon by a body `b`) is assigned to the current object (usually the first one that was created).

- ✓ A lambda expression also exists. However, lambda expressions for *parallel_scan* only run starting from TBB Update 1 (2018). You need to update TBB in order for this lambda expression to work, otherwise it would not recognize this *parallel_scan* with 4 arguments.

```
void main () {
  size_t z[16] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
  size_t y[16], id = 0;
  int n = sizeof(z)/sizeof(z[0]);

  size_t sum = parallel_scan (blocked_range<size_t>(0,n), id,
               // Compute summary of range r
               [&z,&y] (const blocked_range<size_t> &r, size_t sum, bool is_final_scan) -> size_t {
                     size_t temp = sum;
                     for (size_t j = r.begin(); j < r.end(); j++) {
                        temp = temp + z[j];
                        if (is_final_scan)  y[j] = temp;
                     }
                     return temp;
               },
               // Combine body
               [] (size_t left, size_t right) -> size_t {
                      return left + right;
                   }
               );

   printf ("Result: %ld\n", sum); for (int i=0; i < n; i++) printf ("y[%d] = %ld\n",i,y[i]);
}
```

PARALLEL_FOR_EACH

- This template function applies a function object `f` to each element in a sequence [first, last), possibly in parallel:
  `void parallel_for_each (InputIterator first, InputIterator last, const Func &f)`

- This template function implements a **workpile** pattern. For some loops, the end of the iteration space is not known in advance, or the loop body may add more iterations to do before the loop exits. This is a generalization of the map pattern, where we add more to the "pile" of work to be done.
- A linked list is an example of an iteration space that is not known in advance. Moreover, accessing items in a linked list is inherently serial.

TBB PARALLEL_FOR_EACH

- Unlike the other TBB directives, *parallel_for_each* explicitly requires us to work with sequential containers in C++ (e.g.: vectors, lists).
  - ✓ Lists: stores elements at non-contiguous memory locations (internally use a doubly linked list).
  - ✓ Vectors: store elements at contiguous memory location (like an array)
  - ✓ Insertion and deletion of elements is more efficient in lists than in vectors.
  - ✓ A list does not allow for random access, whereas a vector allow for random access.

- *parallel_for_each* accesses the elements of the sequential containers via iterators. An invocation of *parallel_for_each* never causes two threads to act on an input iterator concurrently
  - ✓ iterator: object that points to an element in a range of elements and defines operator that can iterate through elements in a range.

- Example: applying square root to each element of an array:
  - ✓ The class `Applysqrt` specifies the details of the *parallel_for_each* (this is called the imperative form):

```
class Applysqrt {
public:
   void operator() (double &v) const {
       v = sqrt(v);
   }
};
```

  - ✓ Using a 'vector':

```
using namespace std;
using namespace tbb;

int main () {
   int a[10] = {2,3,4,5,6,7,8,9,10,11};
   int i;
   vector <double> myarray; // declaration of an array that can change in size

   for (int i = 0; i < 10; i++) {
      myarray.push_back(a[i]); // push_back: adds elements at the end of vector
   }

   // Imperative form of parallel_for_each:
   parallel_for_each (myarray.begin(), myarray.end(), Applysqrt());

   // Lambda expression form:
   parallel_for_each (myarran.begin(), myarray.end(),
                      [=] (double &elem) { elem = sqrt(elem); } );

   for (i = 0; i < 10; i++) printf ("myarray[%d] = %6.4f\n",i, myarray[i]);
   return 0;
}
```

    - `myarray.begin()`: returns iterator pointing to first element of `myarray`.
    - `myarray.end()`: returns iterator pointing to last element of `myarray`.
    - We can access the individual elements of `myarray` directly (i.e., they provide random access).

  - ✓ Example: (lists):

```
using namespace std;
using namespace tbb;

int main () {
   list <double> mylist = {3,4,5,6,7,8,9,10};
   mylist.push_front(2);
   mylist.push_back(11);

   // Printing the elements of the list:
```

```
    // create iterator 'it'. Note that we access the list via it (we print it).
    for (auto it mylist.begin(); it != mylist.end(); ++it) // cannot use it < mylist.end()
        cout << *it << endl;

    // Alternative printing method:
    for (double x: mylist) // variable x is used to iterate over the list elements
        cout << x << endl;

    // Imperative form of parallel_for_each:
    parallel_for_each (mylist.begin(), mylist.end(), Applysqrt());

    // Lambda expression form:
    parallel_for_each (myarran.begin(), myarray.end(),
                       [=] (double &elem) { elem = sqrt(elem); } );

    for (double x: mylist)
        cout << x << endl;

    return 0;
}
```

  - The list can only be accessed sequentially.

- Note that in both examples, we know the size of the vector/list. However, these sequential containers allow for the insertion/deletion of more elements in a dynamic fashion. This is where *parallel_for_each* is useful: while it is being executed, it is conceivable that the list is still adding elements.

PIPELINING

- This is a common parallel pattern that mimics a traditional manufacturing assembly line. The following is a helpful explanation (source: https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/pipelining/index.html):
  "A useful method of demonstrating this is the laundry analogy. Let's say that there are four loads of dirty laundry that need to be washed, dried, and folded. We could put the first load in the washer for 30 minutes, dry it for 40 minutes, and then take 20 minutes to fold the clothes. Then pick up the second load and wash, dry, and fold, and repeat for the third and fourth loads. Supposing we started at 6 PM and worked as efficiently as possible, we would still be doing laundry until midnight.
  However, a smarter approach to the problem would be to put the second load of dirty laundry into the washer after the first was already clean and whirling happily in the dryer. Then, while the first load was being folded, the second load would dry, and a third load could be added to the pipeline of laundry. Using this method, the laundry would be finished by 9:30."

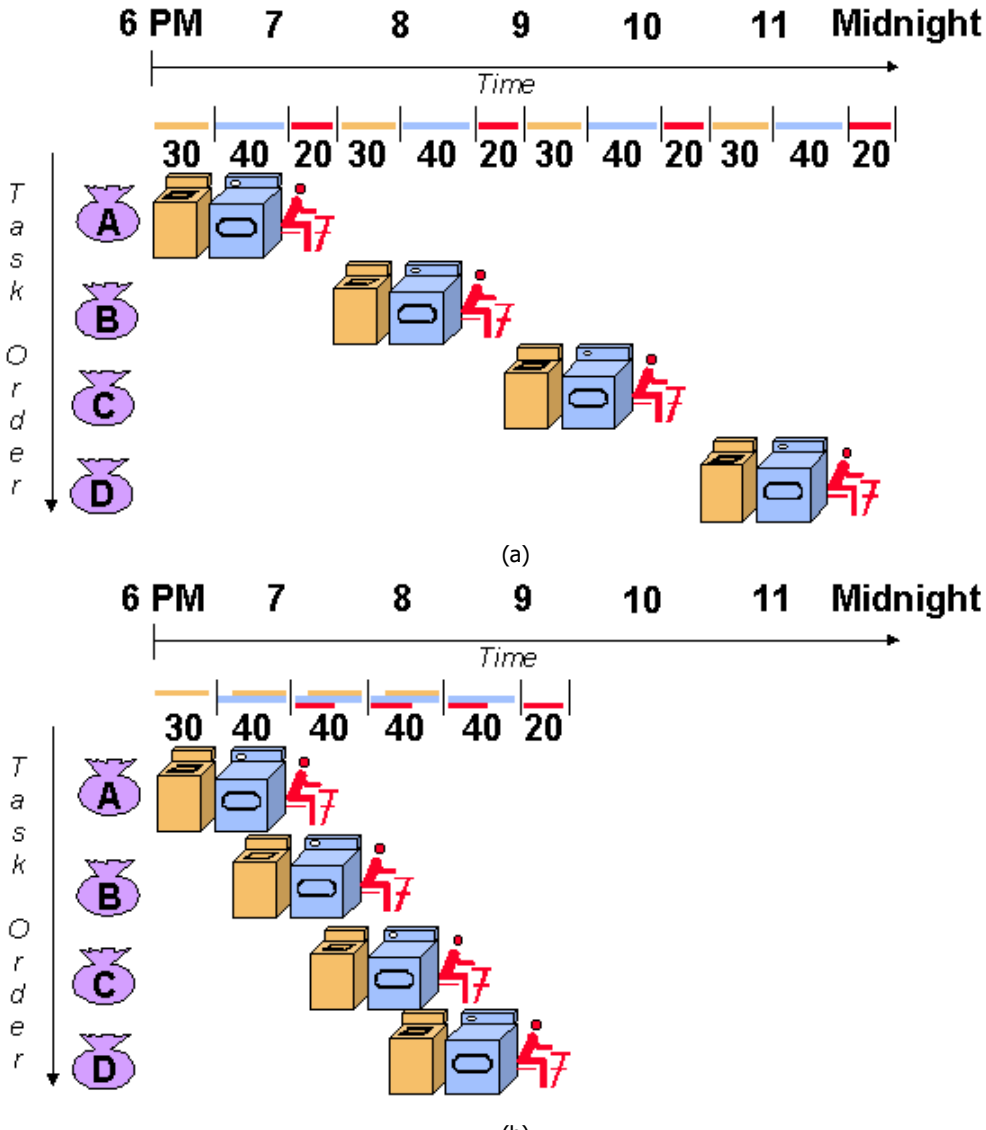


Figure 10. Pipeline explanation. (a) normal sequential operation. (b) pipeline approach.
Source: *https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/pipelining/index.html*

- Pipelines are found in:
  - ✓ Instruction pipelines: The processor breaks the execution of an instruction into stages. Results of one stage are fed onto the next stage. This allows multiple instructions to be in different stages of processing at the same time.
  - ✓ Hardware pipelines: A digital circuit is divided into stages, results of one stage are fed into the inputs of the next stage.
  - ✓ Software pipelines: A software routine can be thought of as a sequence of computing processes with the output stream of one process being fed as the input stream of the next one. Two parallel execution choices:
    - processor (or thread) assigned to execute the task of a single stage.
    - processor (or thread) executes the entire pipeline. Data usually arrives sequentially. When the 1st data arrives, processor 1 starts computation. When the 2nd data arrives, processor 2 starts computation, and so on. When the number of processors is exhausted, we wait until processor 1 finishes its pipeline so it can start a new one.

PIPELINE MODEL FOR SOFTWARE

- Pipeline: linear sequence of stages. Data flows through the pipeline, from the first stage to the last stage.
  - ✓ Stages of the pipeline can often be generated by using functional decomposition of tasks in an application.
  - ✓ Data is partitioned into pieces (also called items or data units).
  - ✓ Each stage performs a transform on the data (this transformation is called a task).
  - ✓ A stage's transformation of items maybe one-to-one or more complicated.
  - ✓ Stages in a pipeline can be balanced (uniform processing time) or non-balanced (non-uniform).
  - ✓ Type of pipeline stages:
    - Serial stage: It processes one item at a time, though different stages can run in parallel.
    - Parallel stage: It processes multiple items at once and can deliver output items out of order.

- Pipelines can be classified depending on the type of stages they contain:
  - ✓ Serial Pipeline: Pipeline with only serial stages. The throughput of the pipeline is limited to the throughput of the slowest serial stage, because every item must pass through that stage at a time.
  - ✓ Parallel Pipeline: This pipeline includes parallel stages (it might include serial stages as well) to make it more scalable.

SERIAL PIPELINE

- Fig. 11 shows a pipeline with 4 stages. Data is fed to the pipeline in terms of data units (or items). For example, for data unit 'a', Stage 1 applies a transform like S1(a), while Stage 2 applies a transform like S2(S1(a)), and so on. We call this a serial pipeline, where each stage can only process one data unit at a time.

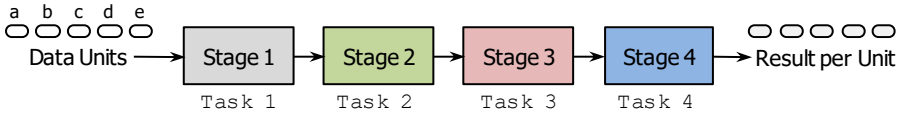


Figure 11 4-stage serial pipeline. Each task is performed by a separate stage. The example shows 5 data units that go through the pipeline along with the final result per data unit.

Pipeline with Uniform Stages

- Here, each stage has a uniform processing time of T cycles. Fig. 12 depicts an example with 5 data units and 4 stages.
  - ✓ Sequential pipeline execution: This naïve approach is depicted in Fig. 12(a). We feed the first data unit 'a' and wait until the final result from Stage 4 is computed. Then, we feed data unit 'b' and wait until we get the result from Stage 4. This repeats until feed the last data unit ('e') and get the corresponding final result from Stage 4. The total computation time is given by $(5 \times 4) \times T$ cycles.
  - ✓ Concurrent pipeline execution: This is depicted in Fig. 12(b). If we continuously feed a new data unit right after Stage 1 has processed a previous data unit, we can expose parallelism (all stages will be busy after a little while). The total computation time is given by $(4 + 5 - 1) \times T = 8T$ cycles. This large reduction in computation time is an advantageous feature of pipelining.

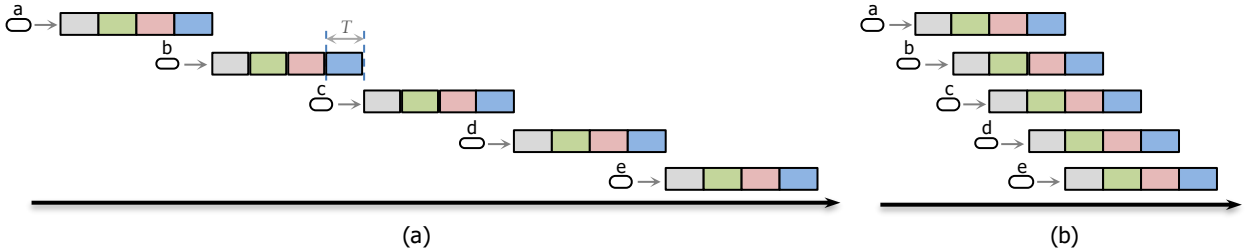


Figure 12. (a) Sequential pipeline execution for 5 data units: it takes 20×T cycles. (b) Concurrent parallel execution for 5 data unit: it takes 8×T cycles. Note how all stages are busy after some initial delay.

- For a pipeline with q stages (each with a processing time T) that is continuously fed n data units, we have that:
  - ✓ Latency (total time for one item to go through the whole system): $q \times T$. This is also called initial latency (number of cycles it takes to process the first data unit).
  - ✓ Total Processing Time: $(q + n - 1) \times T$ cycles.
  - ✓ Throughput (rate at which items are processed, in terms of data units per cycle): $\frac{n}{(q+n-1)\times T} = \frac{1}{\left(\frac{q-1}{n}+1\right)\times T}$
    - In practice, n can be very large and thus the throughput is given by: $\frac{1}{\left(\frac{q-1}{n}+1\right)\times T}\Big|_{n\to\infty} = \frac{1}{T}$ data units per cycle. This can be interpreted as the rate at which new items are processed after the first one (i.e., after the initial latency).

Pipeline with Non-Uniform Stages

- When the processing times of the stages are non-uniform, the slowest stage limits the throughput. Unlike the case with uniform stages, here we cannot guarantee that all stages will be necessarily operating at the same time.

- ✓ Fig. 13(a) depicts the case where Stage 3 takes $1.5T$ cycles, while the other stages take T cycles each. The latency is $4.5T$ cycles. The pipeline must wait until Stage 3 computes its result before feeding a new data to Stage 3. Thus, the processing time is given by $(4.5-1)\times T + (n-1)\times 1.5T + T = 4.5\times T + (n-1)\times 1.5T$. The throughput is given by: $\frac{n}{(4,5+(n-1)\times 1.5)\times T} = \frac{1}{\left(\frac{4.5-1.5}{n}+1.5\right)\times T}$. When $n\to\infty$, the throughput results in $\frac{1}{1.5T}$.
- ✓ Fig. 13(b) depicts the case where Stage 2 takes $2T$ cycles, while the other stages take T cycles each. The latency is $5T$ cycles. The pipeline must wait until Stage 2 computes its result before feeding a new data to Stage 2. Thus, the total processing time is given by $(5-2)\times T + (n-1)\times 2T + 2T = 5\times T + (n-1)\times 2T$. The throughput is given by: $\frac{n}{(5+(n-1)\times 2)\times T} = \frac{1}{\left(\frac{5-2}{n}+2\right)\times T}$. When $n\to\infty$, the throughput results in $\frac{1}{2T}$.

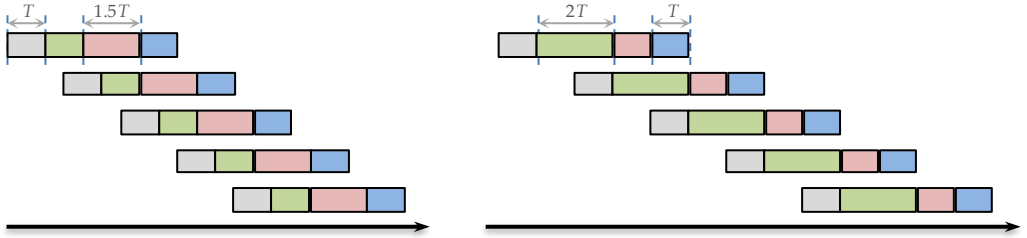


Figure 13. Pipelining for non-uniform stages. (a) Largest stage takes 1.5T cycles. Here, all the stages are busy at one point. (b) Largest stage takes 2T cycles. Here, at most only 3 stages are busy at a time.

- In general (for q stages and n data items), the total processing time is given by $L\times T + (n-1)\times f\times T$ cycles, where $f\times T$ is the processing time of the largest stage ($f>1$) and $L\times T$ is the latency. Note that this formula holds even if the other stages are unbalanced. Also, a balanced pipeline (T cycles per stage) is a special case where $L=q$ and $f=1$.
  - ✓ Throughput: $\frac{n}{(L+(n-1)\times f)\times T} = \frac{1}{\left(\frac{L-f}{n}+f\right)\times T}$ data units per cycle. When $n\to\infty$, the throughput results in $\frac{1}{fT}$, and as such it is determined by the slowest stage.

TABLE III. SERIAL PIPELINE: PROCESSING TIMES. n: NUMBER OF ITEMS. T: NUMBER OF CYCLES OF THE SMALLEST STAGE

Serial Pipeline	Processing Time (cycles)	Throughput (data units per cycle)	Comments
Uniform (each stage takes T cycles)	$(q+n-1)\times T$	$\frac{n}{(q+n-1)\times T} = \frac{1}{T}, if\ n\to\infty$	q: Number of pipeline stages
Non-Uniform (at least one stage takes more than T cycles)	$L\times T + (n-1)\times f\times T$	$\frac{n}{(L+(n-1)\times f)\times T} = \frac{1}{fT}, if\ n\to\infty$	L: factor of the pipeline latency ($L\times T$) f: factor of the largest stage ($f>1$)

PARALLEL PIPELINE

- This is a pipeline where at least one parallel stage is included. Fig. 14 depicts a 4-stage parallel pipeline, where Stage 2 is a parallel stage.
- While a parallel stage can process multiple items at once, note that serial stages are usually present in a parallel pipeline. As such, multiple items usually arrive to the parallel stage at different times. Nevertheless, if the processing time of the parallel stage is larger than the serial stages, the processing of multiple items can be overlapped, thereby reducing the overall processing time.
- The introduction of parallel stages introduces a complication to serial stages. In a serial pipeline, each stage receives items in the same order. In a parallel pipeline, when a parallel stage intervenes between two serial stages, the later serial stage can receive items in a different order than the earlier stage.
- Some applications require consistency in the order of items flowing through the serial stages, and usually the requirement is that the final output order be consistent with the initial input order.

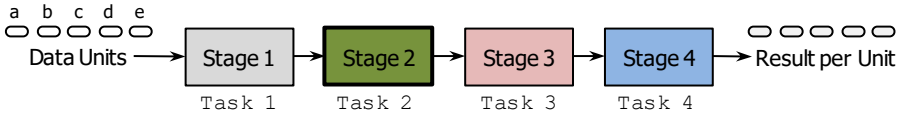


Figure 14. 4-stage parallel pipeline. Stage 2 is a parallel stage that can process multiple items concurrently.

Parallel pipeline with Uniform Serial Stages

- Fig. 15 illustrates the difference between a serial stage and a parallel stage in a pipeline. Fig. 15(a) shows a serial pipeline with a serial Stage 2, while Fig. 15(b) shows a parallel pipeline with serial stages and a parallel Stage 2. Stage 2 (whether serial or parallel) processing time is $4T$ cycles, while the other stages take T cycles.
  - ✓ Serial pipeline: We must wait until Stage 2 computes is result before feeding a new data to it. The total processing time is given by $7T + (n-1)\times 4T$ cycles, where $4T$ is the processing time of the largest stage and $7T$ is the latency.
  - ✓ Parallel pipeline: Note how we can overlap the execution of up to 4 items in Stage 2. The total processing time is greatly reduced to $7T + (n-1)\times T$ cycles.

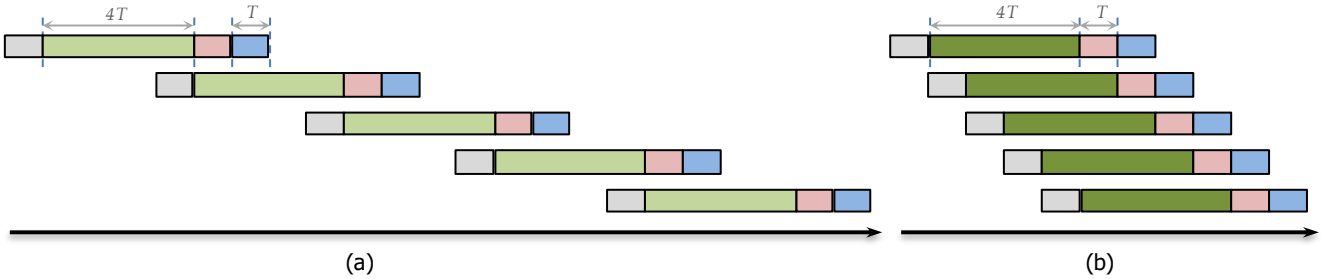


Figure 15. (a) Non-uniform serial pipeline. Stage 2: largest serial stage that takes 4T cycles. (b) Parallel pipeline with uniform serial stages. Stage 2: parallel stage with a processing time of 4T cycles.

- In general, a parallel stage takes $p \times T$ cycles ($p \geq 1$). The processing time of a parallel pipeline with uniform serial stages is given by $L \times T + (n-1) \times T$ cycles, with an asymptotic throughput ($n \to \infty$) of $\frac{1}{T}$. If the parallel stage were a serial stage instead (i.e., a non-uniform serial pipeline), the processing time would be: $L \times T + (n-1) \times p \times T$ cycles, with an asymptotic throughput ($n \to \infty$) of $\frac{1}{pT}$. Thus, there is a speed-up of p, which is the result of overlapping execution of items.
  - ✓ These formulas are valid even when there are other parallel stages, where p is the factor of the largest parallel stage.

- Fig. 16 shows more examples with a parallel Stage 2:
  - ✓ Fig. 16(a): $5T$ cycles for Stage 2. We can overlap the processing of 5 items. Processing time: $8T + (n-1) \times T$ cycles.
  - ✓ Fig. 16(b): $2T$ cycles for Stage 2. We can overlap the processing of 2 items. Processing time: $5T + (n-1) \times T$ cycles.
  - ✓ Fig. 16(c): <u>All</u> stages are uniform. Parallel Stage 2 (T cycles) has no advantage. Processing time: $4T + (n-1) \times T$ cycles.

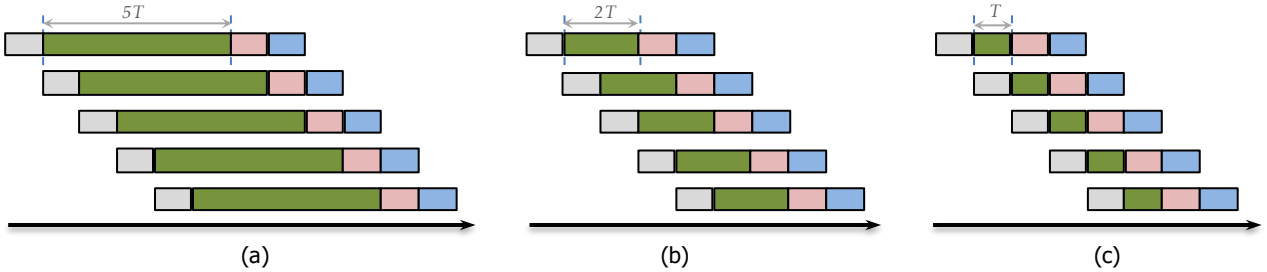


Figure 16. Parallel pipeline execution. Stage 2 is parallel. (a) Stage 2 takes 5T cycles. (b) Stage 2 takes 2T cycles. (c) Stage 2 takes 1 cycle; here, the parallel nature of Stage 2 is not exploited.

Parallel pipeline with Non-Uniform Serial Stages

- If the other serial stages are unbalanced, the processing time is given by $L \times T + (n-1) \times f \times T$ cycles with an asymptotic throughput ($n \to \infty$) of $\frac{1}{fT}$., where f is the factor ($f > 1$) of the largest serial stage. Fig. 17 shows two cases.

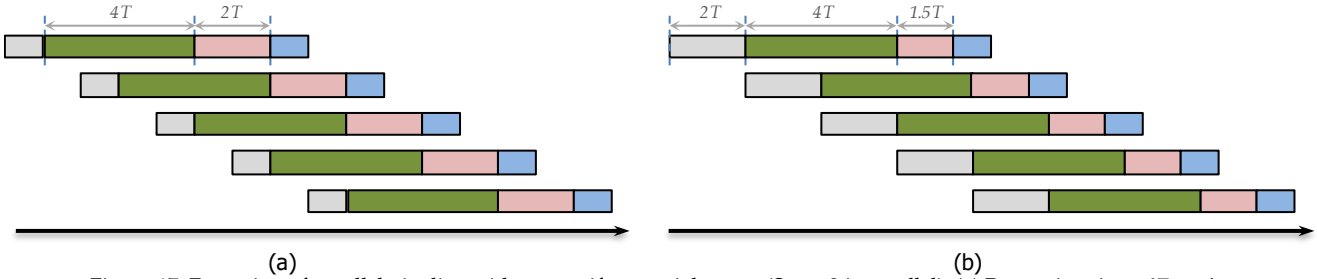


Figure 17. Execution of parallel pipeline with non-uniform serial stages (Stage 2 is parallel). (a) Processing time: $6T + (n-1) \times 2T$ cycles. (b) Processing time: $8.5T + (n-1) \times 2T$ cycles.

- The processing time of the parallel stage(s) get(s) absorbed into $L \times T$ factor. What limits the pipeline is the largest serial stage. The processing times of the parallel stages do not limit the pipeline:
  - ✓ For $p > f$: If the parallel stage were a serial stage, the processing time would be $L \times T + (n-1) \times p \times T$ cycles, with an asymptotic throughput ($n \to \infty$) of $\frac{1}{pT}$. Thus, there is a speed-up of p/f (the result of overlapping execution of items).
  - ✓ For $p \leq f$: If the parallel stage were a serial stage, the processing time would be $L \times T + (n-1) \times f \times T$. Here, there is no speed-up (compared with a serial pipeline), i.e., the benefits of parallel stages are nonexistent.

TABLE IV. PARALLEL PIPELINE: PROCESSING TIMES. n: NUMBER OF ITEMS. T: NUMBER OF CYCLES OF THE SMALLEST STAGE

Parallel Pipeline	Processing Time (cycles)	Throughput (data units per cycle)	Comments
Uniform serial stages (T cycles each)	$L \times T + (n-1) \times T$	$\frac{n}{(L+n-1) \times T} = \frac{1}{T}, if\ n \to \infty$	L: factor of the pipeline latency
Non-uniform serial stages (at least one takes more than T cycles)	$L \times T + (n-1) \times f \times T$	$\frac{n}{(L+(n-1) \times f) \times T} = \frac{1}{fT}, if\ n \to \infty$	f: factor of the largest serial stage ($f > 1$)

PARALLEL_PIPELINE (TBB 3.0)

- Since a parallel stage might cause an ordering issue with the serial stages, Intel TBB defines three kinds of stages:
  - ✓ `parallel`: processes incoming items (even when arriving out of order) in parallel.
  - ✓ `serial_out_of_order`: Processes items one at a time, in arbitrary order.
  - ✓ `serial_in_order`: Processes items one at a time, in the same order as the other `serial_in_order` stages in the pipeline.
- The difference in the two kinds of serial stages has no impact on asymptotic speedup. The throughput of the pipeline is still limited by the throughput of the slowest stage.

- A common representation of a serial pipeline is depicted in Fig. 18(a), while a parallel pipeline is depicted in Fig. 18(b). A serial stage includes a feedback loop that represents updating its state, while a parallel stage does not include a feedback loop. In these diagrams, the processing stages handle a sequence of data items (not just a single item).

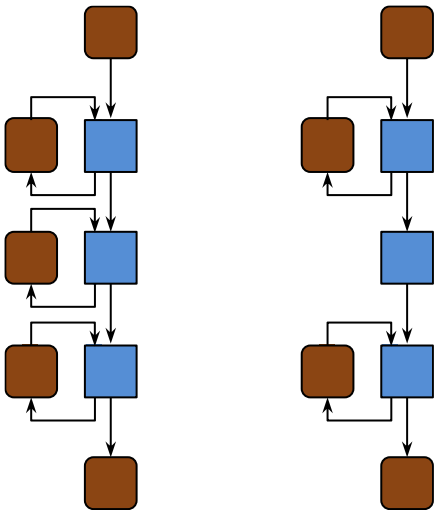


Figure 18. (a) Serial pipeline. Each stage can maintain its state so that later outputs can depend on earlier ones. (b) Parallel pipeline. The parallel stage is stateless; thus, multiple invocations of it can run in parallel.

- Two basic strategies for implementing a pipeline:
  - ✓ Stage-bound workers: Serial stages have one worker, while parallel stages may have multiple workers. A worker processes items as they arrive: it takes a waiting item, performs work, then passes items to the next stage. It is a simple strategy (essentially the same as map), but no data locality for each item.
  - ✓ Item-bound workers: Each worker handles an item at a time and carries the item through the pipeline. On finishing the last stage, it loops back to the beginning for the next item. This is a more complex strategy, but it has much better data locality for items (each item has a better chance of remaining in cache throughout pipeline). However, workers can be stuck waiting at serial stages.

- The difference can be viewed as whether items flow past stages or stages flow past items. The two approaches have different locality behavior. The bound-to-stage approach has good locality for internal state of a stage, but poor locality for the item. Hence, it is better if the internal state is large and item state is small. The bound-to-item approach is the other way around.

- **Hybrid approach**: Based on the two basic strategies, the current implementation of TBB's *parallel_pipeline* uses a modified bind-to-item approach. Workers begin as item-bound.
  - ✓ A worker picks up an available item and carries it through as many stages as possible.
  - ✓ When entering a stage, the worker checks whether it is ready to process the item. If so, the worker continues into the stage. Otherwise, it parks the item, leaving it for another worker to pick it up when the stage is ready to accept it, and starts over.
  - ✓ When leaving a serial stage (a worker finishes applying a serial stage to an item), the worker checks if there is a parked item. If so, it spawns a new worker that unparks that item and continues carrying it through the pipeline.
  - ✓ The approach retains good data locality without requiring workers to block at serial stages.

TBB SYNTAX

- Simplest common sequence of stages for a parallel pipeline (TBB: we use the keyboard *parallel_pipeline*) is serial-parallel-serial, where serial stages are in order.
- Naming conventions: Data units (items) are called **tokens**. Stages are called filters.
- A stage is required to map one input item to one output item. The steps to build a pipeline in TBB are:
  - ✓ A filter (stage) is built with `filter_t<X,Y>`. Type `X` is the input type; type `Y` is the output type.
    - Exceptions: first stage (`filter_T<void, …>`) and the last stage (`filter_t<…,void>`).

- ✓ Glue stages together with operator `&`. The output type of a stage must match the input type of the next stage.
  - From a system perspective, the result acts like a big stage. The top-level glued result is a `filter_t<void,void>`.
- ✓ Invoke *parallel_pipeline* on the `filter_t<void,void>`. The call must also provide an upper bound on the number of items in flight (usually called *ntokens*).

- The *parallel_pipeline* function is a strongly typed lambda-friendly interface for building and running pipeline. To build and run a pipeline from functors $g_0, g_1, g_2, \ldots, g_n$, the syntax is:

```
parallel_pipeline( ntoken,  // maximum number of live tokens
                   make_filter<void,I1>(mode0,g0) &
                   make_filter<I1,I2>(mode1,g1) &
                   make_filter<I2,I3>(mode2,g2) &
                   ...
                   make_filter<In,void>(moden,gn) );
```

- In general, functor g_i should define its `operator()` to map objects of type I_i to objects of type I_{i+1}.
  - ✓ Functor g_0: special case, because it notifies the pipeline when the end of the input stream is reached. *parallel_pipeline* passes a `flow_control` object `fc` to the input functor of a filter.
    - g_0 must be defined such that the expression `g0(fc)` either returns the next value in the input stream, or
    - If the input functor reaches the end of the input stream, it invokes `fc.stop()` and returns a dummy value. This indicates there are no more items, and the currently returned item should be ignored.

- `ntoken` (maximum number of live tokens) parameter to *parallel_pipeline*:
  - ✓ Sets a cap on the number of items that can be in processing at once.
  - ✓ Keeps parked items from accumulating to where they eat up too much memory.
  - ✓ Space is now bound by `ntoken` times the space used by serial execution.

Generic Layout

- This 3-stage parallel pipeline (serial-parallel-serial) is a helpful example. The functors are: g_0, g_1, g_2.

```
parallel_pipeline (ntoken, // 'filter_mode' instead of 'filter' in latest tbb
                   make_filter<void,T>(filter_mode::serial_in_order,
                                       [&](flow_control& fc ) -> T {
                                             T item = g0(); // g0 returns an item of type T
                                             if( !item ) { fc.stop(); return NULL; }
                                             return item;
                                       } ) &
                   make_filter<T,U>   (filter_mode::parallel, g1) &
                   make_filter<U,void>(filter_mode::serial_in_order, g2) );
```

  - ✓ Mode of each functor: `filter_mode::serial_in_order` for g_0 and g_2, `filter_mode::parallel` for g_1.
  - ✓ `T, U` : generic types. The output of each stage matches the input type of the next stage.
  - ✓ Functors g_1 and g_2: They can be defined as i) Classes elsewhere. They may or may not have arguments, and ii) lambda expressions inside *parallel_pipeline*.
  - ✓ Functor g_0 is defined with a lambda expression:
    - The `flow_control` object `fc` is specified as `[&](flow_control& fc) -> T`, where `T` is output type of functor g_0.
    - Functor description: g_0 returns successive items of type `T` when called. If there are no more items (`if (!item)`), it invokes `fc.stop()` and returns a dummy value (NULL).
  - ✓ Stage 1: functor g_0 maps items from `void` to `T`. It uses other variables in the code to input items.
  - ✓ Stage 2: functor g_1 maps input items of type `T` to output items of type `U`. Items can be processed in parallel.
  - ✓ Stage 3: functor g_2 maps items from `U` to `void` and it uses other variables in the code to output items.

Example (with lambda expressions)

- 3-stage parallel pipeline (serial-parallel-serial) that returns the sum of squares of a sequence defined by `[first,last)`. In the C++ code, the three functors are specified as lambda expressions.
  - ✓ Stage 1 (it feeds input data items into the pipeline): Each time it is invoked, it returns an item (from input array `first`) or indicates that there are no more items. Its functor (g_0) returns successive items (pointers of type `float*`) when called, eventually returning NULL when done.
  - ✓ Stage 2: Parallel stage that maps an item of type `float*` to an item of type `float`. Its functor (g_1) returns the square of the value pointed by a `float*` variable. The input to this stage is specified in g_1 (`float *p`), and its type must match the output type of the previous stage. The stage is parallel since we expect this operation to take the longest per item.
  - ✓ Stage 3 (pipeline end point): It receives items (in order) of type `float` and accumulates them. Its functor (g_2) specifies the input to the stage (`float x`). Syntax-wise, the stage has output; however, the generated data is placed in a variable `sum` that is returned by the main `SumSquare` function.
- The C++ code is available below. `ntoken = 16`.

```
float SumSquare( float* first, float* last ) {
   float sum = 0;
   parallel_pipeline (16, // ntoken = 16
                      make_filter<void,float*>(filter_mode::serial_in_order,
                                               [&](flow_control& fc)-> float* { //functor g0: λ exprsn
                                                      if( first < last ) {
                                                          return first++;
                                                      } else {
                                                          fc.stop();
                                                          return NULL; }
                                               } ) &
                      make_filter<float*,float>(filter_mode::parallel,
                                                 [](float* p) { return (*p)*(*p); } ) &
                      make_filter<float,void>   (filter_mode::serial_in_order,
                                                 [&](float x) { sum += x; })  );
   return sum;
}

int main( ) {
   int i;
   float fi[101], *fo, ff;

   for (i = 0; i < 100; i++) fi[i] = i;
   fo = &fi[100]; // fi[100] will not be considered

   ff = SumSquare (fi, fo);
   cout << ff << "\n"; // sum of the squares of 0 to 99: 328350
   return 0;
}
```

- ✓ Note that we can read any input data in Stage 1, and we can modify input data and other variables in Stage 3.
- ✓ In general, for more complex operations, the functors are defined in classes.

- Fig. 19 depicts the pipeline and the operations at each stage. Note the execution of the pipeline with overlapping of operations for Stage 2 (assigned a latency of $3T$ for example). Data is processed in batches of `ntoken=16` items at most.

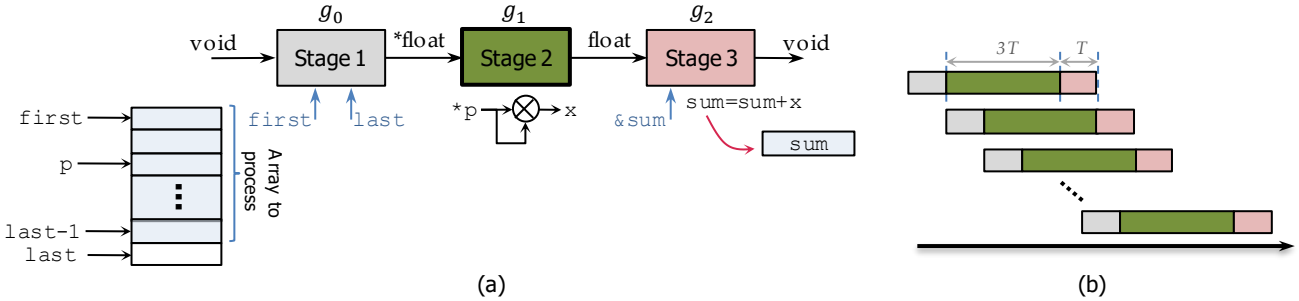


Figure 19. Serial-parallel-serial pipeline. Table V includes more details. (a) Variables in blue are functor's parameters fed to its parameterized constructor. Stage 1 feeds input items into the pipeline. Parallel Stage 2 performs the squaring of an item (items can be processed in parallel). Stage 3 accumulates the result one item at a time. (b) Sample execution with Stage 2 taking $3T$ cycles.

TABLE V. STAGES OF THE PIPELINE IN FIG. 19. STAGE INPUTS ARE FUNCTORS' OPERATOR() PARAMETERS. STAGE PARAMETERS (IN BLUE) ARE FUNCTOR'S INPUT PARAMETERS FED TO ITS PARAMETERIZED CONSTRUCTOR (USUALLY DATA MEMBERS).

Stage	input		output		Functor: input parameters	Comments
	syntax	*type*	*syntax*	*type*		
Stage 1		void	return first++/NULL	*float	g_0: first, last	special: input to stage is a *flow_control* object
Stage 2	float *p	float*	return (*p)*(*p)	float	g_1: none	g_1: no data members, but the stage has input
Stage 3	float x	float		void	g_2: &sum	Data member implied in this λ expression

- Fig. 19 illustrates the advantages of pipelining compared to sequential execution. This example is intended to demonstrate syntax mechanics, as it is not optimal to implement the calculation: parallel overhead would offset any execution time gains.

Throughput of the Pipeline

- This is the rate at which tokens flow through it and is limited by two constraints:
  - ✓ First, if a pipeline is run with N tokens, then obviously there cannot be more than N operations running in parallel. Selecting the right value of N may involve some experimentation. Too low a value limits parallelism; too high a value may demand too many resources (for example, more buffers).
  - ✓ Second, the throughput of a pipeline is limited by the throughput of the slowest sequential filter. This is true even for a pipeline with no parallel filters. No matter how fast the other filters are, the slowest sequential filter is the bottleneck. So, in general you should try to keep the sequential filters fast, and when possible, shift work to the parallel filters.
- To really benefit from a pipeline, the parallel filters need to be doing some heavy lifting compared to the serial filters.