Programmation Parallèle (PRPA)

E. Carlinet {edwin.carlinet@lrde.epita.fr}

2021

EPITA Research & Development Laboratory (LRDE)

Agenda

Loop-oriented Parallel Design Patterns (with TBB)

Agglomeration

Element-wise

Odd-even communication

Wavefront

Reduction

Flow and Graph-oriented Design Patterns

Pipeline

Agenda

Agenda

1. CM1 - Introduction to parallelism
2. CM1 - Instruction and data-level parallelism
3. CM2 - Thread level parallism
4. CM2 - Parallel Design Patterns (with TBB)
5. CM3 - C++ Memory model (Atomics)
6. CM4 - Data structure for concurrent programming

Loop-oriented Parallel Design Patterns (with TBB)

Copyright

Inspired from:

Intel's TBB Developer guide

About ranges

- Important concept in TBB that controls **looping**
- Predefined ranges (`blocked_range<int>`, `blocked_range2d<int>`, `blocked_range3d<int>`
- Can be defined from STL ranges (`blocked_range<STLIterator>`)
- Can be customized

About ranges

- Splittable
- Has a grain size

```
class R {
  // True if range is empty
  bool empty() const;
  // True if range can be split into non-empty subranges
  bool is_divisible() const;
  // Splits r into subranges r and *this
  R( R& r, split );
  // Splits r into subranges r and *this in proportion p
  R( R& r, proportional_split p );
  // Allows usage of proportional splitting constructor
  static const bool is_splittable_in_proportion = true;
  // ...
};
```

Agglomeration

Agglomeration

Context

- An algorithm permits fine-grained parallelism
- Each item computation is too fast to compensate thread sync.

Example

- Parallelizing the conversion of a vector of strings to int

Solution

- Group the computations into blocks. Evaluate computations within a block serially.
- Block size:
  - large enough to amortize parallel overhead
  - too large a block size may limit parallelism or load balancing (number of blocks to small)
- Loop is "small" ($<$ 10,000 cycles) may be impractical to parallelize at all

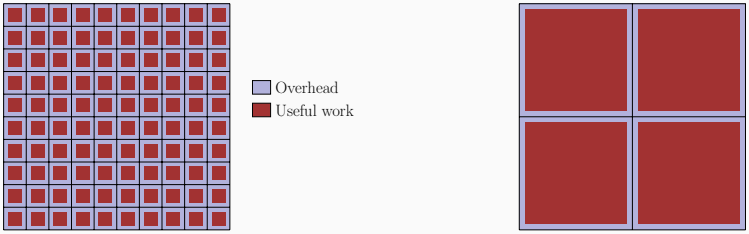
Chunking strategies: the grain size

Chunking is controlled by a **partitioner** and a **grainsize**. To gain the most control over chunking, you specify both.

```
parallel_for(blocked_range<size_t>(0,n,G),
             ApplyFoo(a),
             simple_partitioner());
```

- The **grainsize** sets a minimum threshold for parallelization.
- Using `simple_partitioner` guarantees that $\lceil G/2 \rceil \leq \mathit{chunksize} \leq G$.

Chunking strategies: the grain size



- Left: small grainsize leads to a relatively high proportion of overhead
- Right: large grainsize reduces overhead, reduces potentially parallelism

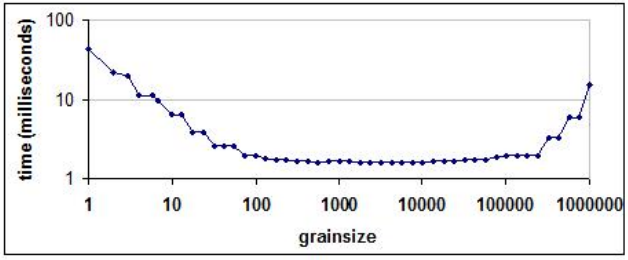


Figure 1: Time for `a[i] = b[i] * c` on 1M float items (4 physical cores / 8 logical cores)

Chunking strategies: the partitioners

- N is the size of the range
- G in the grain size
- T is the number of available worker (ressources)

Partitioner	Description	Chunk size
simple	Split until the grain size is reached.	$G/2 \leq$ chunksize $\leq G$
auto	Minimize work splitting while maximizing work stealing.	$G/2 \leq$ chunksize
static	Distributes range iterations among worker threads as uniformly as possible	$\max(G/3, N/T) \leq$ chunk_size
affinity	Same as auto, but better cache affinity (e.g. loop is re-executed over the same data set)	$G/2 \leq$ chunksize

	Simple	Auto	Static	Affinity
Workload Balancing	✓	✓		✓
Cache affinity			✓	✓
Upper bound	✓			

Example with *parallel_reduce*:

`Body::Body( Body&, split)` Splitting constructor. May run concurrently with operator() and join().
`void Body::operator()(const Range&)` Accumulate result for subrange.
`void Body::join( Body& rhs)` Join results. The result in rhs should be merged into the result of this.

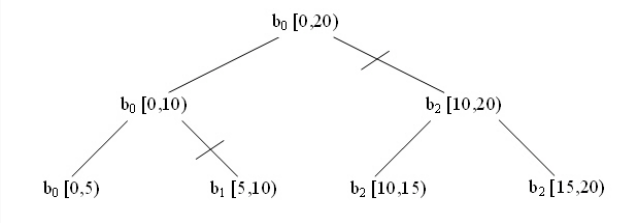


Figure 2: Possible execution of simple partitioning over the `[0,20]` with grain size 5

	1	2	3	4	5	6
Thread 0	b0.split()	b0.split()	b0([0,5])	...	b0.join(b1)	b0.join(b2)
Thread 1			b1([5, 10])	...	X	
Thread 2		b2([10, 15])	...	b2([15, 20])	...	X

Element-wise

Element-wise

Context

- Sweep over a set of items and do independent computations on each item.
- No information is carried or merged between the computations.

Solution

- Number of items known: `parallel_for`
- Number of items unknown: `parallel_do`
- Use *agglomeration* if individual computations are small
- If followed by a local reduction, consider doing the element-wise operation as part of the reduction.

Example

Normalization between [0,1] (Map)

$$output(k) = \frac{input(k) - vmin}{vmax - vmin}$$

Sliding window smoothing (Stencil)

$$output(k) = \frac{1}{5} * \sum_{i=-2}^{2} input(k+i)$$

Example

Normalization between [0,1] (Map)

$$output(k) = \frac{input(k) - vmin}{vmax - vmin}$$

```
tbb::parallel_for(0, len, [=](int k) {
    out[k] = (input[k] - vmin) / (vmax - vmin);
});
```

What about inplace processing (`out = in`) ?

Sliding window smoothing (Stencil)

$$output(k) = \frac{1}{5} * \sum_{i=-2}^{2} input(k+i)$$

```
tbb::parallel_for(2, len-2, [=](int k) {
    int sum = 0;
    for (int i = -2; i <= 2; ++i)
        sum += input[k + i];
    output[k] /= 5;
});
```

Example

Normalization between [0,1] (Map)

$$output(k) = \frac{input(k) - vmin}{vmax - vmin}$$

```
tbb::parallel_for(0, len, [=](int k) {
    out[k] = (input[k] - vmin) / (vmax - vmin);
});
```

Sliding window smoothing (Stencil)

$$output(k) = \frac{1}{5} * \sum_{i=-2}^{2} input(k+i)$$

```
tbb::parallel_for(2, len-2, [=](int k) {
    int sum = 0;
    for (int i = -2; i <= 2; ++i)
        sum += input[k + i];
    output[k] /= 5;
});
```

What about inplace processing (`out = in`) ?

OK for *map*, harder for the *stencil* because of dependancies.

Odd-even communication

Odd-even communication

Context

Operations on data cannot be done entirely independently, but data can be partitioned into two subsets such that all operations on a subset can run in parallel.

Example: Isotropic diffusion inplace

$$u^{k+1}(x,y) = u^k(x,y) + \lambda(\Delta_N^k(x,y) + \Delta_S^k(x,y) + \Delta_W^k(x,y) + \Delta_S^k(x,y))$$

where:

- $\Delta_N^k(x,y) = u^k(x,y-1) - u^k(x,y)$
- $\Delta_S^k(x,y) = u^k(x,y+1) - u^k(x,y)$
- $\Delta_E^k(x,y) = u^k(x+1,y) - u^k(x,y)$
- $\Delta_W^k(x,y) = u^k(x-1,y) - u^k(x,y)$

Isotropic diffusion - 10 iterations ($\lambda = 0.25$)

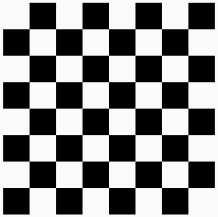
```
for (int k = 0; k < K; ++k)
{
  for (int y = 0; y < height; ++y)
    for (int x = 0; x < width; ++x)
      g(x,y) = f(x,y+1) + f(x,y-1) + f(x+1,y) + f(x-1,y);
  swap(f, g);
}
```

Parallelization

- If *not* inplace -> no problem
- If *inplace* (input = output) -> data race

Solution

- Alternate between updating one subset and then the other subset.
- Apply the elementwise pattern to each subset



```
for (int k = 0; k < K; ++k)
{
  for (int y = 0; y < height; ++y)
    for (int x = x%2; x < width; x+=2)
      g(x,y) = f(x,y+1) + f(x,y-1) + f(x+1,y) + f(x-1,y);

  for (int y = 0; y < height; ++y)
    for (int x = x%2+1; x < width; x+=2)
      g(x,y) = f(x,y+1) + f(x,y-1) + f(x+1,y) + f(x-1,y);
  swap(f, g);
}
```

Wavefront

Wavefront

Context

Perform computations on items in a data set, where the computation on an item uses results from computations on predecessor items.

Example (LCS)

The longest common substring between \(x_{1..i}\) and \(y_{1..j}\) is:

$$LCA(x_{1..i}, y_{1..j}) = \begin{cases} LCA(x_{1..i-1}, y_{1..j-1}) + 1 & \text{if } x_i = y_j \\ \max(LCA(x_{1..i-1}, y_{1..j}), LCA(x_{1..i}, y_{1..j-1})) & \textit{otherwise} \end{cases}$$

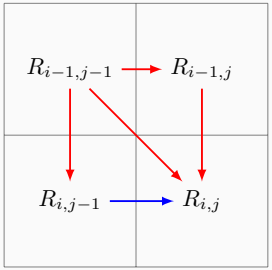


Figure 3: Direct and indirect data dependencies

Solution

```
template<typename InputIterator, typename Body>
void parallel_do( InputIterator first, InputIterator last, Body body);
```

with body:

```
Body::operator()(T item, parallel_do_feeder<T>& feeder) const
```

- Parallel variant of topological sort with `tbb::parallel_do`
- Associate an **atomic** counter to each item
- When an item is processed:
  - Decrement successors counter
  - If counter drops to 0, add it to the work list
- Can be combine with agglomeration pattern (for efficiency)

```
using P = std::pair<int, int>;
std::atomic<int> cnt[n][m] = 2; // Pseudo-code
P origin = {0, 0};
parallel_do(&origin, &origin + 1,
    [](P cell, parallel_do_feeder<P>& feeder)
    {
      auto [i, j] = cell;
      // Should check bounds: HERE
      LCA[i][j] = (x[i] == y[j]) ? (LCA[i-1][j-1]) : max(LCA[i-1][j], LCA[i][j-1])
      if (j < m-1 && --cnt[i][j+1] == 0)
          feeder.add({i, j+1})
      if (i < n-1 && --cnt[i+1][j] == 0)
          feeder.add({i+1, j})
    });
}
```

Note: this is inefficient, do block processing instead

Reduction

Reduction

Context

- Many serial algorithms sweep over a set of items to collect summary information
- Perform an *associative* reduction operation across a data set.

Example

- Parallelizing the sum of a vector of strings

Solution

- `tbb::parallel_reduce` if fully associative
- `tbb::parallel_deterministic_reduce` if almost associative (e.g summing floats)

```
long sum_string_tbb(const char* strings[], std::size_t n, int nthread)
{
  return
    tbb::parallel_reduce(
      tbb::blocked_range<const char**>(strings, strings + n),          // Range
      0L,                                                             // Id
      [](const tbb::blocked_range<const char**>& rng, long init) {    // Map
        return init + local_sum(rng.begin(), rng.size());
      },
      std::plus<long>(),                                              // Reduce
      tbb::static_partitioner()
      );
}
```

Flow and Graph-oriented Design Patterns

Devide and conquer

Context

- Problem can be transformed into subproblems that can be solved independently
- Splitting problem or merging solutions is relatively cheap compared to cost of solving the subproblems.

Devide and conquer

Context

- Problem can be transformed into subproblems that can be solved independently
- Splitting problem or merging solutions is relatively cheap compared to cost of solving the subproblems.

Example

- Quicksort / Merge sort

Solution

- `tbb::parallel_invoke` if the number of subproblems is always the same
- `tbb::task_group` if the number of subproblems varies

Quicksort

```
void quicksort(T* begin, T* end)
{
    if (end - begin <= 1)
        return;
    T pivot = *begin;
    T* mid = std::partition(begin + 1, end, [=](T x) { return x < pivot; });
    std::swap(*begin, *(mid-1));
    tbb::parallel_invoke(
        [=]() { Quicksort(begin, mid-1); },
        [=]() { Quicksort(mid, end); });
}
```

Parallel computation of the depth of a tree

```
struct Node
{
    int   n_child;
    Node* child;
};

int walk(const Node* node)
{
    if (node == NULL)
        return -1;

    std::vector<int> depths(node->n_child);
    tbb::task_group jobs;
    for (int i = 0; i < node->n_child; ++i)
        jobs.run( [=,&depths]() { depths[i] = walk(node->child[i]); });
    jobs.wait(); // wait for completion

    return *(std::min_element(depths.begin(), depths.end())) + 1;
}
```

Note: this is an example (you should limit recursion !)

Pipeline

Pipeline

Context

- Pipelining is a common parallel pattern that mimics a traditional manufacturing assembly line.
- Data flows through a series of pipeline filters and each filter processes the data in some way

Example

In video processing:

- some operations on frame does not depend on previous frame (pre-proc. e.g. denoising)
- some operations depend on previous frame (e.g. tracking)

Online word correction

Online word correction

1. Read the next token in input file
2. Search for the word and its traduction (can take time)
3. Write the next token in output file

```
tbb::parallel_pipeline(
    nrequest, // maximal number of requests that can be handled in parallel
    tbb::make_filter<void, std::string>(tbb::filter::serial_in_order, Reader()),
    tbb::make_filter<std::string, std::string>(tbb::filter::parallel, Traducer()),
    tbb::make_filter<std::string,void>(tbb::filter::serial_in_order, Writer()))
```

Filter	Description
`parallel`	Process multiple items in parallel and in no particular order.
`serial_out_of_order`	Process items one at a time, and in no particular order.
`serial_in_order`	Process items one at a time, and with order.