**Lecture 7 - Simple parallel algorithms**

Data-parallel vs task-parallel:

* data-parallel — same operation is performed, independently, on distinct subsets of the processing data;
* task-parallel — distinct operations, that do not depend on each other, are performed in parallel

The most general approach: consider the dependency graph between computed quantities. It is a DAG (directed acyclic graph). Notes:

* parallelizable activities can be found, for instance, by splitting the graph into levels;
* the *critical path* (the longest path, or the maximum cost if distinct processing steps take different times) gives a lower limit for the execution time even on an infinite number of CPUs.
* sometimes, the critical path can be shorteden at the expense of increasing the amount of computation to be performed.
* the graph may not be known from the beginning...

**Simple data decomposition**

Processing of an array of data can often be split into independent blocks.

The easiest case is when each input produces one output — the *map* pattern. See the example for computing the sum of two vectors: [vector\_sum\_split\_work.cpp](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/vector_sum_split_work.cpp).

Simple way of computing the boundary index: beginIdx = (threadIdx \* nrElements) div nrThreads

However, beware of cache effects! Processing consecutive elements is significantly faster than processing every *k*-th element. Compare the previous program with the one at [vector\_sum\_split\_work\_bad.cpp](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/vector_sum_split_work_bad.cpp).

A more complex case arises when each output depends on a group of inputs, around the input at the same position — the *stencil* pattern. See [vector\_average\_stencil.cpp](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/vector_average_stencil.cpp).

It is preferrable to split on output than on inputs — so that each output is computed by exactly one worker (thread, task) and so no mutexes are necessary.

**Recursive decomposition**

The initial worker splits the data into two or more fragments, gives the fragments as inputs to subordinate workers, and finally it combines the results.

**Example 1:** Compute the sum of a vector. Create a binary tree of adders. The depth is *O*(log(*n*). Source code: [recursive\_decomposition\_sum.cpp](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/recursive_decomposition_sum.cpp).

**Example 2:** Merge sort. The basic (non-parallel algorith) is to divide the input vector into two parts, merge-sort each part, then merge the resulting two sorted vectors into one. For parallelizing, merge-sorting the two parts can easily be done in parallel. However, the final merge is a bit harder. It can be done as follows:

* take the middle element in the first sorted vector;
* find its position in the second sorted vector, by a binary search;
* divide the two vectors by the two positions found above;
* merge independently the two pairs of sub-vectors;
* concatenate the results (this is a no-op actually).

See the C++ implementations:

* [mergesort.cpp](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/mergesort.cpp) — serial implementation;
* [mergesort-par1.cpp](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/mergesort-par1.cpp) — parallelized, but with non-parallel merge;
* [mergesort-par2.cpp](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/mergesort-par2.cpp) — fully parallelized, including parallel merge;

**Example 3:** Compute the sequence of sums of prefixes. Given *a0, a1, ..., an-1*, compute *b0 = a0, b1 = a0+a1, b2 = a0+a1+a2,..., bn-1 = a0+a1+a2+...+an-1*.

Solution: start with a binary tree computing the sum of all numbers in the sequence. Then, compute each prefix sum from the largest parts already computed.

// First, compute the sums of 2^j consecutive numbers;

// b[i\*2^j - 1] = a[(i-1)\*2^j] + ... + a[(i-1)\*2^j + 2^j - 1]

b = a

for(size\_t k=1 ; k<n ; k = k\*2) {

for(size\_t i=2\*k-1 ; i<n ; i+=2\*k) { // in parallel

b[i] += b[i-k];

}

}

// Then, compute each partial sum as a sum of 2^j groups:

k = k/4

for( ; k>0 ; k = k/2) {

for(size\_t i=3\*k-1 ; i<n ; i+=2\*k) { // in parallel

b[i] += b[i-k];

}

}

Examples:

A diagram of a diagram

Description automatically generated

A diagram of a diagram

Description automatically generated