Crash course introduction to parallelism: the algorithms

January 4, 2021Parallelization, PerformanceLeave a Reply

*We at **Johnny's Software Lab LLC** are experts in performance. If performance is in any way concern in your software project, feel free to contact us.*

When it comes to performance, there are two ways to go: one is to improve the usage of the existing hardware resources, the other is to use the new hardware resources. We already talked a lot about how to increase the performance of your program by better using the existing resources, for example, by decreasing number of data cache misses or avoiding branches. But there is a theoretical maximum that these techniques reach and to move forward new resources are needed. This is the story of parallelism.

Modern processors have been stuck at speeds of 3 GHz to 4 GHz for about ten years. Making them run faster has been a challenge because of heating and power consumption issues. However, the size of transistors has been decreasing and is still decreasing, which means more transistors can be added to the CPU which further translates to better CPU architecture and more processing cores. But to use more available resources we need to rethink the way we are writing our programs. So, how does parallelism comes into the picture?

The story of parallelism

The story of parallelism is the story about solving your problem in such a way so that the problem can be divided into chunks and then each chunk is solved in parallel. Some problems are trivial to parallelize, some a bit more difficult, some require that the algorithm is completely rewritten and there are some that are impossible to parallelize.

Trivially parallelizable

For the loop of a trivially parallelizable algorithm, each loop iteration is independent of the others. Examples of trivially parallelizable algorithm:

```
void increment_array(int* array, int len, int increment) {
    for (int i = 0; i < len; i++) {
        if (array[i] > 0) {
            array[i] += increment;
        }
    }
}
void add_arrays(int* res, int *a, int* b, int len) {
    for (int i = 0; i < len; i++) {
        res[i] = a[i] + b[i];
    }
}
```

Above are two examples of trivially parallelizable algorithms. The values in the current iteration do not depend on the values of other iterations. We can split the problem so CPU 1 does calculation on array

for `i` between 0 and 9999, CPU 2 does calculation on array for `i` between 10000 and 19999 etc.

Almost trivially parallelizable – reductions

Another type of problem that comes up often is finding minimum, maximum, calculating sum or calculating average on an input array. Here is an example of an algorithm calculating sum:

```
long sum(long* arr, int len) {
    long res = 0;
    for (int i = 0; i < len; i++) {
        res += arr[i];
    }
    return res;
}
```

In this case, the current value of variable `res` depends on the previous value. So there is a dependency, and we cannot split this problem into two problems.

Luckily, we can take advantage of the associativity of operator +. In addition (a + b) + c = a + (b + c). That mean we can perform the addition in any order. So we can in parallel perform the addition on CPU 1 for values of `i` between 0 and 9.999, on CPU 2 for values of `i` between 10.000 and 19.999, etc. In a final step, we *reduce* the result of each parallel execution to one final result. For addition we sum up all sums produces by each parallel execution.

Reductions are a bit more difficult to parallelize, due to the final step of reducing several values to one value. However, they are still easy.

More difficult cases

Some algorithms are more difficult, albeit not impossible to parallelize. For example, calculating histograms:

```
int calculate_histogram(int* histogram, int* val, int len) {
    for (int i = 0; i < len; i++) {
        int index = val[i];
        histogram[index]++;
    }
}
```

The above algorithm is difficult to parallelize not because it has a typical loop carried dependency, but because it has a race condition. Imagine that `val[0]` and `val[10000]` have the same value. If those two values are processed in parallel, a race condition can occur and the corresponding value in `histogram` array might get the wrong value.

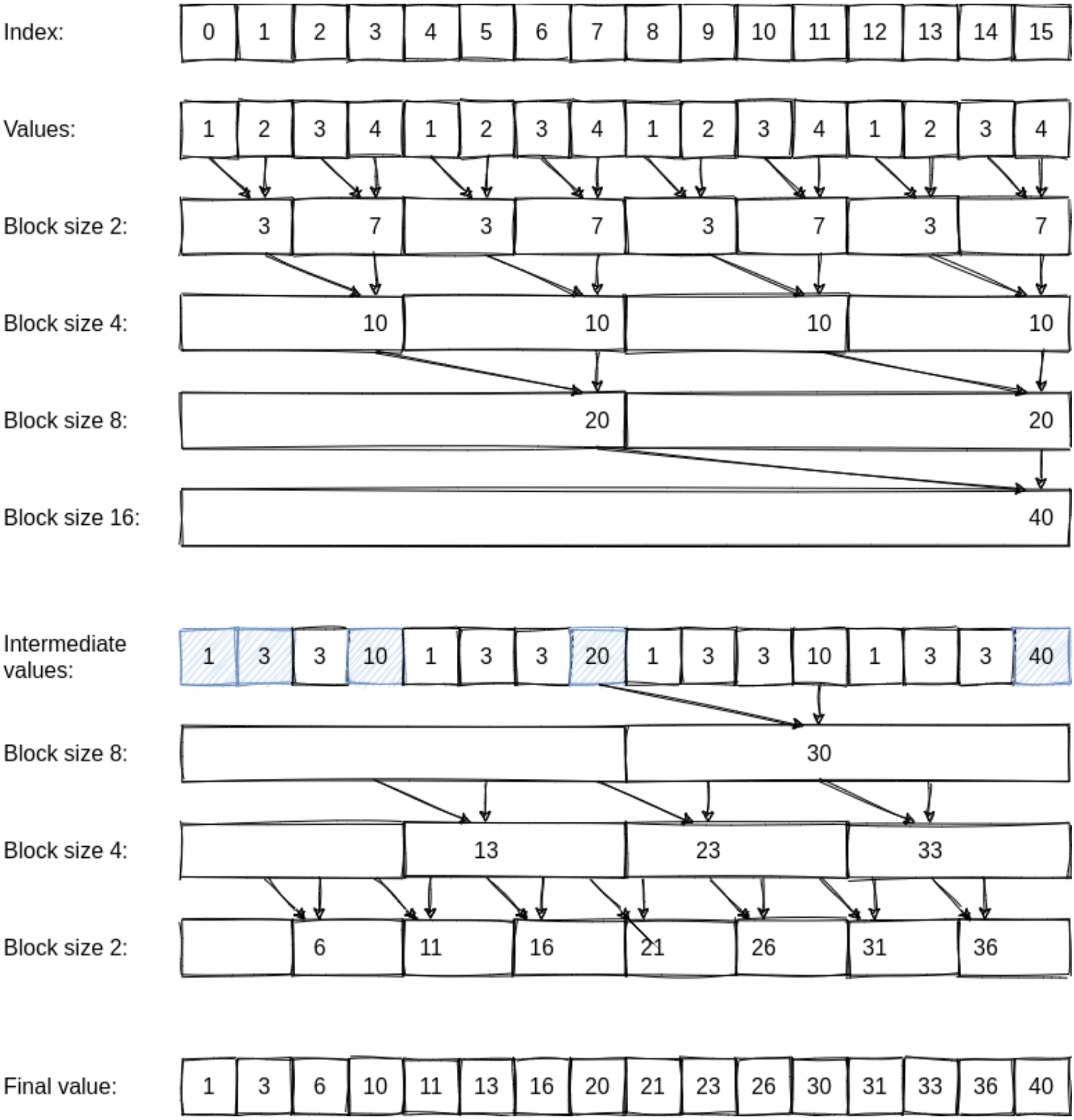
There are two solutions to this. Access histogram array using atomics (`std::atomic<int> *histogram` instead of `int* histogram`) or keep a local copy of `histogram` array for each

execution and at the end reduce them to one array. The first approach will be slower because of access to atomic variables, but it can make sense in case there is a huge number of parallel executions. The second approach is faster, but it consumes more memory and there is a reduce step at the end.

Another well known problem is calculating partial sums. Consider the following example:

```
void partial_sums(int* out, int* in, int len) {
    out[0] = in[0];
    for(int i = 1; i < len; i++) {
        out[i] = out[i - 1] + in[i];
    }
}
```

There is no simple way to parallelize this algorithm because each iteration depends on the previous. However, this problem can be parallelized using a divide-and-conquer technique like illustrated in the picture:



Parallelized version of prefix sums

The parallel prefix sum algorithm runs in two passes. In first phase we go from smallest block size to largest block size, where we calculate the intermediate values for each block size, as illustrated. In the second pass we the go the other way around, until we get to the final value.

Parallelized prefix sum actually does more additions than the serial version. If the length of the array is `n`, the serial version does `n-1` additions, whereas the parallel version does `2 * n - 2 - log(n)`.

One notable feature about the above example is that even though the algorithm is parallelizable, the bigger the block size, the smaller the parallelization potential. Even if you have unlimited hardware, when the block size becomes too big you the added hardware would be of no use.

Impossible to parallelize

There are algorithms that are sequential by nature and that are very difficult or impossible to parallelize. Those algorithms often find their application in cryptography, because they make breaking the encryption impossible by adding hardware.

An example is hash chaining, where a cryptography hash function `h` is applied successively to a string x, i.e:

$$h^n(x) = h(h(...(h(h(h(x))))))$$

A few practical consideration related to parallel programming

We saw that some algorithms cannot be parallelized easily, and others cannot. What are the other factors that prevent or limit the benefits of parallelization?

Amdahl's law

Amdahl's law is quite simple. The program's performance is limited by its sequential part. Imagine the program consists of two parts: sequential and parallel. The sequential part takes 25% percent of its execution, the parallel part takes 75% percent of the time. Now, imagine our program runs for 10 seconds, out of which 2.5 seconds is spent in the sequential part and the rest in the parallel part.

If we had an infinite amount of hardware, we could decrease the execution time of the parallel part to almost zero. However, the serial part remains, and our program cannot be faster than 2.5 seconds.

In our example of prefix sums, as the block size grows, the parallelization potential decreases. When the block size becomes large enough, we must switch to serial execution. The bottleneck will be there, and this cannot be mitigated through additional resources.

Universal Scalability Law

If you have ever parallel programmed a real word application, you have surely seen a case where the addition of more resources actually slows down the program. This is generalized as Universal Scalability Law, which basically states that the limit to parallelization does not only come from sequential parts, but from several parallel execution instances having to share resources.

We already covered the details of resource sharing in the previous article about optimizations in multithreaded environment, but its essence is as follows. Sharing resources is expensive, and the more parallel execution instances we have, the more resource sharing is going to cost us. As we add more resources we come to a point of diminishing returns, where adding resources actually slows us down.

Memory Bandwidth Limit

In modern multicore and multiprocessor systems, bandwidth to various parts of the memory subsystem (registers, L1 cache, L2 cache, main memory) is limited. All the CPU cores that are accessing the shared resource have to share the bandwidth, and when a core has to wait for its data, this effectively limits the benefits of parallelization.

As the size of your data set grows, so does your CPU has to access slower and slower memory subsystems in order to fetch the data. What also adds more stress to the memory subsystem is the data access pattern. **Accessing data one-by-one is the best, followed by accessing data with constant stride. The worse access pattern is random** (pointer chasing code like access to hash maps or binary trees).

Embedded systems and typical desktop systems are not designed with enough memory bandwidth to allow all the CPU cores to access the main memory at the same time, especially when the memory access pattern is bad.

We tested this on Intel's i5-10210U laptop with eight cores. We wrote a small program that sums 100 million elements of an array. The serial version took 539 ms, the parallel version with eight threads took 107 ms. The speedup factor is 5.03. Next, we wrote another program that performs 100 million lookups in a `std::unordered_set` (hash set). This program is much less memory friendly. The serial version took 9378 ms, the parallel version with eight threads took 3781 ms. The speedup factor is much lower in this case – only 2.48.

Final Words

We've covered the basics of parallelism as far as algorithms are concerned. Even in this short article, you've seen that parallelization is not a silver bullet, that it is not always easy nor simple nor without its limitations.

In the next article we will cover how parallelism is implemented in modern hardware, including SIMD parallelism, multithreading on several CPUs, and using graphic cards or other accelerators to achieve parallelism.

Featured image by: Parallel Algorithms (BE4M35PAG)