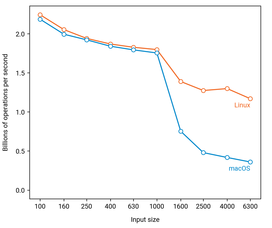
Programming Parallel Computers

Chapter 2: Case study

Version 1: Linear reading

Let us now try to identify the bottleneck that we have in **version 0**. In our computer we have got two main components that we use heavily: the CPU and the main memory. In the innermost loop, we read data from the main memory to the CPU registers and then do arithmetic operations in the CPU. Which part is the bottleneck — getting data from the memory to the CPU, or doing arithmetic operations in the CPU?

Let us see how the performance varies depending on the input size. We also try out the same code on two different computers, one running Linux and another running macOS — both of these have exactly the same Intel Core i5-6500 processor:



Something clearly happens when we increase the input size from n = 1000 to n = 1600. For n = 1000, the input consists of 1 million single-precision floating point numbers, each of which takes 4 bytes, for the total input size is **4 megabytes**, while for n = 1600, there are about 2.5 million elements and the total input size is about **10 megabytes**.

If we check the **specifications** of the processor that we use here as the running example, we can see that the size of the **last-level cache is 6 megabytes**. Clearly we have got a memory bottleneck: everything is fine as long as the entire input fits in the cache memory, but the performance drops once we run out of cache memory. So for n = 4000, the worst bottleneck in our implementation is getting input data from the main memory to the CPU.

What is the source of the problem?

Let us look at the following fragment around the innermost loop:

```
for (int j = 0; j < n; ++j) {
    // ...
    for (int k = 0; k < n; ++k) {
        // ...
        float y = d[n*k + j];
        // ...
    }
    // ...
}
```

For example, for `n = 10`, we will access `d` in the following order:

```
d[0], d[10], d[20], ..., d[90],
d[1], d[11], d[21], ..., d[91],
...
d[9], d[19], d[29], ..., d[99],
d[0], d[10], d[20], ..., d[90],
d[1], d[11], d[21], ..., d[91],
...
d[9], d[19], d[29], ..., d[99],
...
```

That is, we will scan through the entire array `d` repeatedly, and in a somewhat non-linear order. If all of `d` fits in the cache memory, there is nothing particularly bad about this, but as soon as we run out of cache memory, this starts to hurt us.

For example, when we first read `d[0]`, it gets automatically stored in the cache memory, in case it might be needed again soon. However, we will not refer to it again until we have read all other elements — and by the time we access `d[0]` again, it has already had to make room for other elements in the cache. We will have to fetch it again from the main memory — and in modern computers, this is expensive.

But why was macOS slower? [advanced]

Memory addresses that we use in a computer program do not refer directly to the physical addresses; they are **virtual memory** addresses, the operating system maintains **page table** that tells how to map virtual memory addresses to physical memory addresses, and the CPU has to do this translation whenever we read or write anything that is stored in the memory.

This would be painfully slow, unless the CPU had a separate cache, so-called **TLB**, that keeps track of recent translations between virtual memory addresses and physical memory addresses.

The virtual memory is organized in **pages**. A full page of virtual memory is always mapped to a full page of main memory. Hence, if we do a lot of memory lookups to consecutive addresses, they typically fall within the same memory pages, and address translation is cheap.

The main difference between macOS and Linux is in the default size of the memory page. Modern Linux kernels provide a so-called **Transparent Hugepage Support**, and in the system we used for benchmarking this was enabled by default. In brief, whenever our program allocates a large block of memory, our Linux system typically returns virtual memory that is organized in pages of **2 MB**. However, macOS defaults to the more traditional page size of **4 KB**.

When we increase `k` by one, the memory address pointed by `d[n*k + j]` increases by 16 KB (assuming n = 4000). Hence, in the case of Linux, we are typically still within the same 2 MB memory page, while in the case of macOS, we hit another memory page. There is not enough space to store information on all of these pages in the TLB, and hence we will pay the additional penalty of looking up page tables.

The same behavior can be reproduced on Linux by switching off Transparent Hugepage Support.

How to fix it?

A very good rule of thumb for modern computers is that **reading memory in a linear order is a good idea**. For example, instead of reading elements `d[0]`, `d[n]`, `d[2*n]`, etc, it would be much better to read `d[0]`, `d[1]`, `d[2]`, etc, in this order.

There are two main reasons for favoring linear reading:

- Memory is always transmitted in full **cache lines** between the main memory and various levels of cache in the CPU. A cache line is 64 bytes. Hence, when you read `d[0]` (which takes only 4 bytes), you are simultaneously transmitting also e.g. `d[1]`, `d[2]`, ..., `d[15]` from the main memory to the caches near the CPU. This would of course be convenient, if these were exactly the elements that your code would need next.
- Linear reading is **what the hardware expects** and what it is optimized for. When the CPU notices that a program seems to be reading consecutive elements of memory, **hardware prefetching** will automatically kick in: the CPU will guess that the program will keep requesting consecutive words of memory, and it will start to fetch data from the main memory to the cache memory already before the program has asked for it.

So what we will do is to modify the code so that we are mostly doing just linear reading of the input. The basic idea is this:

- Original: we scan matrix `d` by rows (linear reading) and the same matrix `d` by columns (non-linear reading).
- Improved: we first construct matrix `t` which is the transpose of `d`, and then we can scan `d` by rows (linear reading) and `t` by rows (also linear reading).

Here is a straightforward implementation of this idea:

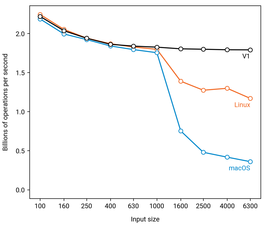
```
void step(float* r, const float* d, int n) {
    std::vector<float> t(n*n);
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            t[n*j + i] = d[n*i + j];
        }
    }

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            float v = std::numeric_limits<float>::infinity();
            for (int k = 0; k < n; ++k) {
                float x = d[n*i + k];
                float y = t[n*j + k];
                float z = x + y;
                v = std::min(v, z);
            }
            r[n*i + j] = v;
        }
    }
}
```

And now we already know how to parallelize it with OpenMP:

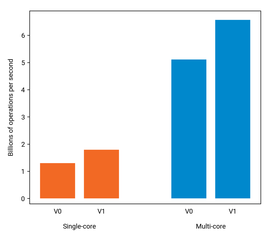
```
void step(float* r, const float* d, int n) {
    std::vector<float> t(n*n);
    #pragma omp parallel for
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            t[n*j + i] = d[n*i + j];
        }
    }

    #pragma omp parallel for
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            float v = std::numeric_limits<float>::infinity();
            for (int k = 0; k < n; ++k) {
                float x = d[n*i + k];
                float y = t[n*j + k];
                float z = x + y;
                v = std::min(v, z);
            }
            r[n*i + j] = v;
        }
    }
}
```

Results

The modification was clearly worth the effort. Now the code runs roughly as fast on small and large inputs; the memory is no more the limiting factor:



The gains on the Linux computer for n = 4000 are rather modest, though:



We will next figure out what is the performance bottleneck that prevents us from getting more than 1.8 billion operations per second (0.50 operations per clock cycle) with a single core, or roughly 4 times that on 4 cores.