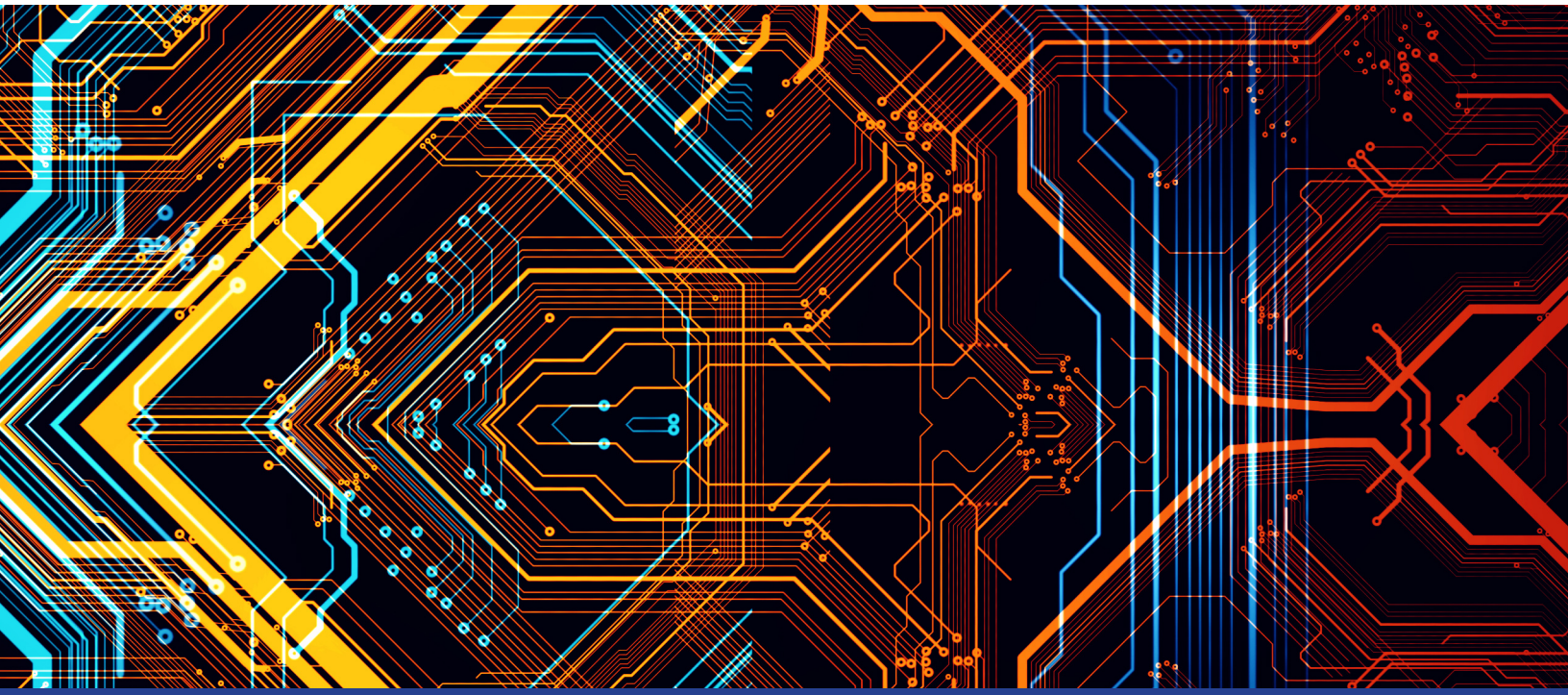

THE PERFORMANCE OPTIMISATION AND PRODUCTIVITY (POP) PROJECT

Pursuing the Never-Ending Quest for Performance

Mike Croucher, Developer Advocate, Numerical Algorithms Group (NAG)

For a long time now, the route to increased performance has been via parallelization. Vectorization, threads, MPI*, OpenMP*, GPUs, FPGAs, and dozens more hardware and software technologies promise to give you the performance you and your users crave. So you choose a set of technologies, embark on your code optimisation journey, and realize some fantastic speedups that your users eagerly consume. The success stories roll in and you sit back, content that the community is now using your product to solve bigger and more advanced problems than anyone ever considered feasible. All is going well.

Sign up for future issues

But the quest for improved performance is never over, and soon your users want you to perform the speedup trick once again. The models they're building are bigger and more complex than ever. And the hardware they're running them on has new vectorization tricks—and much higher core counts—than you ever considered before. Your code base is huge, your budget limited, and all the low-hanging fruit has been picked and devoured.

Where do you start applying your development efforts?

The PoP Project

The **Performance Optimisation and Productivity (PoP) project** is a European Union-funded, international group of partners working to improve parallel software via several complementary routes including:

- **Developing a general methodology** that can be used to understand parallel performance
- **Developing open source tools** that can be used to apply the PoP methodology
- **Creating a set of detailed case studies** where PoP experts demonstrate these developments by auditing and refactoring the code of academic and industrial clients (available for free for clients within the EU).

The PoP methodology can be applied to a range of parallelization schemes and programming languages. OpenMP and MPI in Fortran*, C, and C++ are the most popular, but PoP has also worked on applications written in MATLAB*, Python*, and Perl*, among others.

The PoP Methodology

Traditionally, there are several things we can try to gather intelligence about our application, such as scaling experiments, profiling, and tracing using products like **Intel® VTune™ Amplifier** or the open-source tools developed by some PoP partners. These can result in a huge amount of data to sift through, containing everything from instruction counters to cache misses. It can be difficult to move from this sea of information to the kind of insights that would really help a code developer determine the most appropriate direction to follow to improve the code.

The PoP methodology distills this sea of data into a **small hierarchy of metrics** that measure the relative impact of the different factors inherent in parallelization. Each metric is a measure of efficiency between 0 and 1, where higher numbers are better. As a rule of thumb, PoP considers anything below 0.8 as worthy of further attention.

A case study can help us understand these metrics.

The PoP Metrics and zCFD*

One of the PoP partners, **The Numerical Algorithms Group (NAG)**, recently worked on the commercial computational fluid dynamics solver **zCFD*, developed by Zenotech**. By generating the PoP metrics from Intel VTune Amplifier data and collaborating with the original developers, NAG helped improve the runtime of one particular simulation by 3x.

The first step in the audit was to limit the collection of Intel VTune Amplifier data to only the region of interest (RoI). zCFD uses a Python package (**zCFD-driver***) that calls computational kernels written in C++. As such, the team used the **NERSC Python VTune Instrumentation and Tracing Technology (ITT) API bindings** to disable tracing outside the RoI.

Once the Intel VTune Amplifier data was collected for simulation runs on varying numbers of cores, the first set of PoP metrics could be computed (**Table 1**). (How to compute the PoP metrics from Intel VTune Amplifier data is outside the scope of this article. For details, see the **PoP webinar on this case study**. An alternative method is described in the article **Automatic Calculation of PoP Metrics Using Scalasca.**)

Table 1. PoP metrics

Threads	1	2	4	6	8	10	12
Global Efficiency	0.97	0.71	0.66	0.52	0.55	0.49	0.33
Parallel Efficiency	0.97	0.80	0.77	0.64	0.67	0.60	0.50
Computational Efficiency	1.00	0.89	0.85	0.82	0.82	0.82	0.66

The headline figure is global efficiency, which is the product of the parallel and computational efficiencies. Parallel efficiency measures the effect that parallelizing the code has on the runtime. This includes the impact of factors such as:

- **How well-balanced** the computational load is between threads
- **How much time** is lost to parallel overheads

It's calculated as the ratio between the average amount of time that threads spend in useful computation and the total runtime of the application.

Computational efficiency describes how well the computational load of the application scales with the number of threads. It's the ratio between the total time across all threads that the code spends in useful computation and the time the serial code spends in useful computation.

We observe that there's a general decline in global efficiency as the number of threads increases. This is largely driven by a corresponding decline in parallel efficiency. The computational efficiency doesn't decline as much, except on 12 threads.

Taken together, these efficiencies suggest that the prime opportunity for improvement lies in the way work is divided among threads rather than the computations each thread performs. For example, on 10 threads, the computational efficiency of 0.82 denotes that there's the potential to improve runtime by 18% if issues associated with computation are addressed—compared with a potential 44% improvement from addressing parallelization issues that the parallel efficiency of 0.56 suggests. With that said, there's something very strange going on with computational efficiency at 12 cores.

Parallel Efficiency

Now that we understand that focusing on parallel efficiency should give us the most gains, we can dive deeper to try to understand why it's so poor. A straightforward metric we can obtain from Intel VTune Amplifier is the percentage of runtime spent in serial sections of code (**Table 2**).

Table 2. Runtime in serial code

Threads	1	2	4	6	8	10	12
Percentage of Runtime in Serial	—	88.6	84.6	75.0	74.2	70.1	66.6

By the time we reach 12 cores, 33% of our runtime is spent in serial code sections. Further investigation determines there was a region the developers had attempted to parallelize, but that was actually still running sequentially. Some refactoring corrected this.

Load balance efficiency (**Table 3**) shows that work is spread unevenly across threads.

Table 3. Load balance efficiency

Threads	1	2	4	6	8	10	12
Load Balance Efficiency	1.00	0.88	0.89	0.85	0.89	0.86	0.85

Further investigation shows that the main load imbalance occurred in a region of code that called the `pow()` function. This was hitting a **slow code path**. Because both the base and the exponent were close to 1, `pow()` was computing the result to high accuracy, which took a lot of time. But this level of accuracy was not needed by the computation. This was resolved by scaling the base, raising it to the power, and then undoing the scaling[1]:

```
double pow(double a, double b) {
    a *= 5.0;
    tmp = pow(a,b);
    return tmp/pow(5.0,b);
}
```

The two calls to `pow()` can be computed at the same time using vectorization, so this change only incurred the cost of a single extra divide.

Computational Efficiency

Although the metrics showed us that computational efficiency isn't as important as parallel efficiency for this particular problem, there's something very strange going on when we move from 10 to 12 cores that warrants a closer look. We might hope it's something straightforward that we can easily fix. Happily, this is the case.

There are three submetrics that make up computational efficiency (**Table 4**):

1. Instructions per cycle (IPC) efficiency
2. Instructions efficiency
3. CPU frequency efficiency

Instruction efficiency is the ratio of the total number of useful instructions for a reference case (e.g., one processor) compared to values when increasing the numbers of processes. A decrease in instruction efficiency corresponds to an increase in the total number of instructions required to solve a computational problem.

IPC efficiency compares IPC to the reference, where lower values indicate that the rate of computation has slowed. Typical causes for this include decreasing cache hit rate and exhaustion of memory bandwidth, which can leave processes stalled and waiting for data.

CPU frequency efficiency looks at how clock speed changes as the number of threads increases.

Table 4. Submetrics that make up computational efficiency

Threads	1	2	4	6	8	10	12
IPC Efficiency	1.00	0.94	0.93	0.92	0.91	0.90	0.91
Instructions Efficiency	1.00	1.00	1.00	1.00	1.00	1.00	1.00
CPU Frequency Efficiency	1.00	0.94	0.91	0.89	0.90	0.91	0.72

There's nothing much of interest going on with the IPC and instructions efficiencies, but the CPU frequency drops sharply going from 10 to 12 cores.

Zenotech determined that the CPU frequency governor was set to on-demand by default on the machine used for the audit, and that this was responsible for the drop in operating frequency. Adding `--cpu-freq=performance` to the Slurm* commands resolved the issue by instructing the CPU to run at its base frequency even when fully populated with threads.

Results

Guided by these metrics, the developers of zCFD made the changes to the code and compute environment described above (along with a few more that we don't have the space to describe here). Recalculating the metrics on the new code resulted in the efficiencies shown in **Table 5.**

Table 5. Efficiencies

Threads	1	2	6	12
Global Efficiency	1.00	0.89	0.73	0.56
Parallel Efficiency	1.00	0.98	0.89	0.76
Computational Efficiency	1.00	0.91	0.82	0.74

We see across-the-board improvements comparing **Table 5** to **Table 1**. And when Zenotech ran the new code on a much larger problem, they observed speedups of up to 3x compared to the original code. Even with this success, the metrics suggest there might be yet more room for improvement. The quest continues.

Applying for a PoP Code Audit

The PoP Project provides performance optimization and productivity services for academic and industrial code in all domains. They offer a portfolio of services designed to help users optimize parallel software and understand performance issues. The services are free of charge to academic, research, or commercial organisations in the EU. You're invited to apply for PoP time **via the website**.