Parallel patterns: What organizations can learn from supercomputers

Lorenz Graf-Vlachy

University of Passau
Chair of Technology, Innovation, and Entrepreneurship
94030 Passau
lorenz.graf-vlachy@uni-passau.de

The author would like to acknowledge helpful comments from Georg Wittenburg.

This manuscript was published in *Management+Innovation*, 2014, Nr. 1, pp. 34-39.

Parallel patterns: What organizations can learn from supercomputers

In today's high-speed world, leaders face the need to make their organizations faster and faster. Engineers building and programming supercomputers have historically been facing similar challenges. To address these challenges, they developed some ingenious techniques to increase the speed of computations – and now is the time for managers to see what can be learned from supercomputer engineers.

Performance pressure on organizations is high these days. Companies need to become quicker providers of solutions to customers' problems, administrations need to respond to citizens' requests faster, intelligence agencies need to increase the speed of their information processing [1].

A key step to addressing these needs is to hire the right people and train them well. But, of course, this is only half of the equation. The other half is making sure that an organization's individual members interact in the most efficient way possible.

Similar to organizations, today's supercomputers consist of large numbers of individual processing units that need to interact. Hence, the engineers that program supercomputers face challenges akin to those faced by organizational leaders. In fact, supercomputer programming can point us to several different approaches for effectively harnessing the power of discrete elements.

The ascent of parallel computers

Ever since the creation of the first functional freely programmable computer "Z3" by Konrad Zuse in 1941, engineers raced to build better and faster computers. And in fact, over the years, performance of computers improved dramatically. While computers could initially only perform comparably simple operations, today they are used to simulate highly complex phenomena such as nuclear explosions, the earth's climate, or the effects of a new drug on organisms.

A large part of the race was about who would build the fastest processor. The processor is the unit of the computer that performs the actual computations, i.e. the actual "electron brain." At least since the 1970s, processor speed would double roughly every two years. This development was dubbed "Moore's law," and was built on the observation and prediction that the number of transistors on integrated circuits, and hence computing power, increased at this pace [7]. Computer processors got faster and faster. And they keep improving to this day—Moore's law still holds.

However, when we take a look at the fastest computers of the world, we see that another aspect has become more and more relevant over the last decades. In the TOP500 list, a ranking of the world's fastest supercomputers, the last computer that obtained its power from a single processor appeared in 1996 [12]. What do the other, faster supercomputers consist of? Well, quite simply, *multiple* processor. In fact, all contemporary supercomputers are actually many computing devices linked together.

Linking many comparably cheap processors together to form one single supercomputer has led to dramatic—exponential, really—performance increases. The fastest supercomputer in 2014 is about 600,000 times faster than the top supercomputer of a decade ago and it can compute a staggering 33.8 PetaFLOPs per second. This means the world's fastest supercomputer can compute 33,900 millions of millions operations per second. That's a figure with 15 zeros.

This exceptional computing device, which is located in Guangzhou, China, goes by the name of "Milky Way 2." It consists of over three million individual processors. Since all these processors are operating in parallel, supercomputers of this type are also called "parallel computers."

Programming problems

Usually, such supercomputers are not commissioned to solve many small problems, so that each processor gets its own to solve. Instead, they are usually built to solve one big problem. And this can be a challenge.

Most algorithms, the blueprints of software, are designed for one single processor only. Hence, most computer software is written to run on a single processor. It takes its input, processes it in isolation from the outside world, and then spits out the result. In the realm of supercomputers, however, software programs need to make use of the many processors available if they want to be worthy of running on such exquisite hardware. Problem solving activities need to be performed in parallel—and this usually requires re-thinking the way software is written.

Because it is often not obvious how to identify and exploit what is called "concurrency" in a problem and appropriately structure a computer program to make use of multiple processors, many software engineers across the world invested a lot of time and thought into this matter. They came up with a few basic principles on how to write programs for supercomputers. In the language of software engineers, they developed "design patterns" for parallel computing [3].

Organizing work with parallel patterns

There exist at least three basic organization principles for how to structure a problem, and hence a computer program for parallel computing [6]. These patterns of assigning work to processors are by *tasks*, by *data decomposition*, and by *data flow* (see Figure 1).

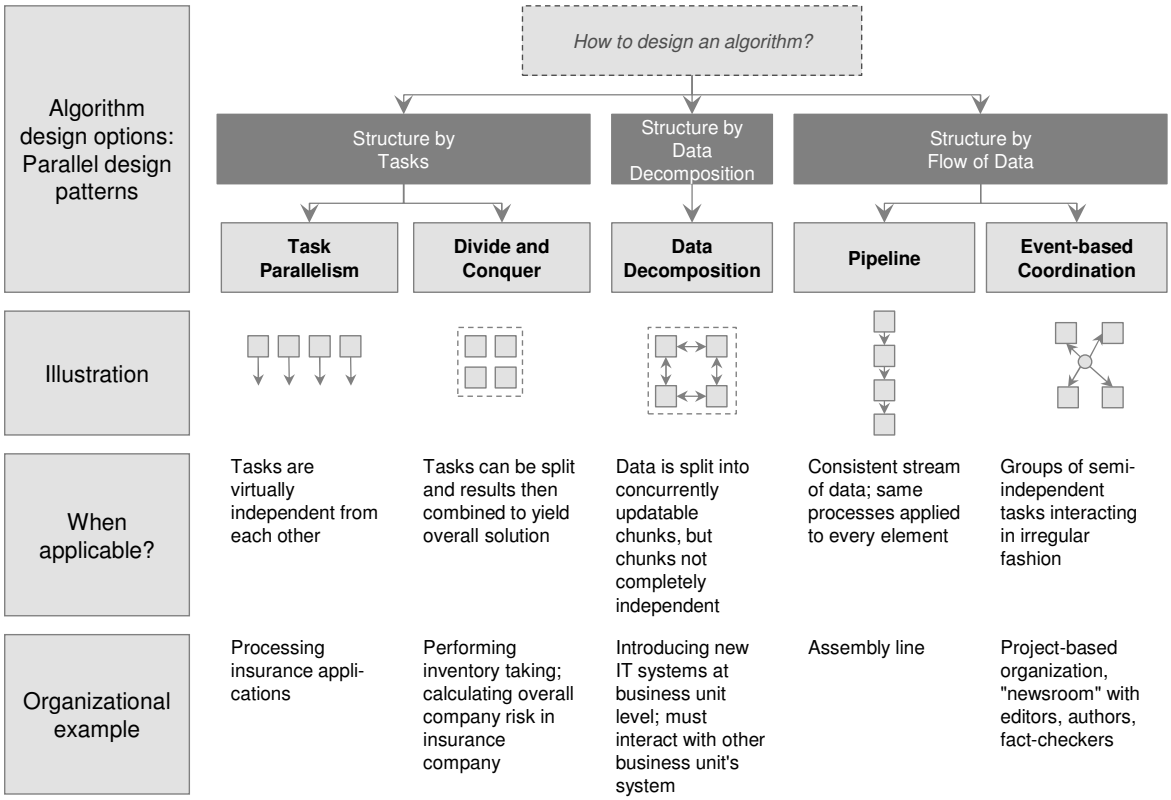

Figure 1: Parallel design patterns

First, in many cases, computer programs are best understood as a collection of *tasks* that need to be performed. Structuring by tasks means directly asking the question: What to do? Two sub-patterns can help answer this question.

One way of structuring by task is *task parallelism*. Basically, this can be used whenever the overall problem is essentially just a set of individual tasks and the tasks are virtually independent from each other. In this case, each processor can work on one task. An example would be the problem of taking the square root of eight different numbers. Eight different processors could simply take the square root of one number each. This is so simple that the technical term for problems like this is “embarrassingly parallel.”

Another task-based strategy is *divide and conquer*. It can be used when the overall problem can be split into smaller, independent subtasks and when the results of these subtasks can be recombined to yield the full solution. An example could be the multiplication of eight numbers. In the first step, four processors each multiply two numbers, then two processors each multiply two of the results, and finally, one processor multiplies the last two results.

Second, sometimes, programs are best understood as a sequence of operations on data chunks resulting from the *decomposition of the data* into concurrently updatable pieces.[1] This design pattern takes the perspective of data, as opposed to that of tasks and therefore asks: Of what structure is the data that is to be manipulated?

The pattern is useful when the data can be split into concurrently updatable chunks, but the chunks are not completely independent (in the case of complete independence, the simpler *task parallelism* should be used). The basic idea is that each processor accesses mostly "its own" chunk of data exclusively, and that only after each "update cycle," the processors working on "neighboring data" need to briefly interact with each other. An example is the blurring of digital images. To blur an image, every pixel's color has to be changed depending on the color of the surrounding pixels. Using the decomposition design pattern would imply that the entire image is split up into smaller sections and that each processor can process one section. Largely, these tasks can be conducted independently, and only at the edges of the section does every processor need to access the information from the neighboring sections.

Third, in some occasions, the *flow of data* is the best organizing principle for parallel programs. The fundamental question here is: When does data have to be processed? Again, there are two sub-patterns.

[1] Technically, there is a second pattern related to data composition, namely the "recursive data" pattern. The pattern is appropriate when the problem to be solved has the structure of following links through a recursive data structure, e.g., a binary tree. Since this pattern does not seem to have any relevance to real-world organizational problems, it is not further treated here.

One is called *pipeline*. It is useful when there is a consistent, regular stream of incoming data flowing through multiple stages and the same manipulation processes is applied to every element. Naturally, every processor can take on one stage. If, for example, an incoming stream of Fahrenheit temperature values must be converted to Celsius, one processor could deduct 32 from the Fahrenheit temperature, and another processor could multiply the result by 5/9 to yield the Celsius temperature.

The other flow-based pattern is *event-based coordination*. It applies when groups of semi-independent tasks are interacting in an irregular fashion. This means there is an irregular information inflow, the tasks are asynchronous and they react to events and cause events. Consider, for example, a computer system that approximates complex mathematical problems "on demand." Whenever a new problem arrives, one processor reacts to this event and begins approximating a solution. After a predefined time span, the processor signals that its time is up and presents its current best approximation. This event causes another processor to take up the solution and evaluate it against certain quality criteria, e.g. minimum precision. If the quality is sufficient, the processor signals that the calculation is finished. If the quality is not sufficient, the processor announces this, and another processor reacts to this event by further refining the approximation.

Organizations as computers

There exists a long tradition of looking at computers to understand other phenomena. Several scholars have, for example, studied computers to better understand individuals and their brains, and developed ideas like information processing theory from the comparison [4, 8, 9]. Such comparisons can be helpful and inspiring even when one considers that computers are really profoundly different from human brains [10].

What if organizations can be understood as supercomputers built out of human minds instead of silicon-based processors? Maybe there is something to learn from parallel design patterns if one wants to optimize organizational processing speed. In fact, organizations are similar to computers

at least in the way that both can be understood as highly complex information processing devices [2, 11].

Computers take input data, perform complex manipulations contingent on this data, and finally present an output. Organizations do just the same. They collect informational input from the environment (e.g., market research information indicating a new opportunity or a customer complaint), process this data (e.g., set up an R&D project or understand the reason for the complaint), and create an output (e.g., introduce a new product or produce a response to the customer complaint).

Hence, the wide set of parallel design patterns presented above can inform organizational decision making about structures and processes with regard to three different aspects.

First, they are a reminder that thinking hard about concurrency in tough problems might pay off even when it is not immediately obvious, and they provide a variety of patterns to look for. Is there really not an intelligent cut through that project behind schedule which allows using a *data decomposition* approach? Maybe a second project manager could take over some of the workload and both project managers would only need to align on a very limited set of issues every once in a while?

Second, the different design patterns make it easier to identify possible alternative approaches to parallelization. Take the example of building an IT system that impacts multiple departments. Is it necessary to *divide and conquer*, i.e. build the modules for the departments individually, and later connect all the partial solutions simultaneously with a "Big Bang" launch? Or might it be preferable to take a *data decomposition* approach, i.e. build department-specific systems and simultaneously create interfaces only between the individual departments that really need them?

Third, organizations should consider whether a number of patterns can be fruitfully combined or nested. An example could be insurance companies' claims processing departments, which today often operate in *task parallelism* logic: Every incoming letter is assigned to a clerk for processing.

Instead, incoming mail might well be opened, digitized, and preprocessed in a *pipeline* fashion first, reaping benefits of specialization. Only in a final step, every claim could be assigned to an individual clerk for processing, thereby introducing some *task parallelism* logic.

Speculating on parallelism

Beyond programming patters, however, there is at least one additional technique that parallel computers leverage to increase computation speed.

When computers with multiple processors execute complex programs, they will invariably have some of the processors run idle at times. The more processors there are, the greater the chance that a few of them will not have anything to do at a given point in time. Software and hardware engineers have long understood this, and successfully capitalize on it. The technique they use is called “speculative execution” [5].

The basic idea is that the processor tries to make a guess about which part of a computer program gets executed next and to run it before it is known whether it actually needs to be run. In case it turns out that the program part needed to be executed, the computer wins time. In case it turns out the program part did not need to be executed, the computer simply disposes of the calculated results.

Imagine a computer program running on a computer with two processors. Assume further that the program uses one processor to perform a calculation and that the further execution of the program depends on the outcome of that calculation. If speed is paramount, it might be a good idea for the second processor to make an educated guess about the first processor’s result and “speculate” on it. The second processor can attempt to continue the program execution without knowing the actual result that the first processor will produce. In case the second processor speculates correctly, there is a significant upside: The result of the calculation that it computed in advance can be used immediately. In case the speculation does not work out and the calculation result

cannot be used, little is lost: Had the processor not speculated, it would simply have been idle. The only cost of a lost speculation shows up on the electricity bill.

Similarly, if there are resources that have to wait for others before they can begin to work in an organization, it might be fruitful to speculate on future needs and act accordingly. Consider, for example, the case in which you need to wait for a CEO decision on which of two product ideas to pursue further. If time is critical, it might be worth beginning to map out business plans for both of them before the decision is taken.

Naturally, employees might not be happy about discarding the results of their work in case the speculating manager loses the bet. However, explicit recognition of their work and adequate celebration of work output, even if it is ultimately unneeded, might attenuate any hard feelings and turn intelligent speculation into a competitive advantage.

Harnessing parallelism in organizations

By considering the three programming patterns described, as well as the idea of speculative execution, organizations can increase their speed through identifying concurrency in their tasks and choosing appropriate organization structures and processes. Alternatively, if speed is sufficient already, improving structure and processes might potentially allow organizations to use less sophisticated individual processors, i.e. to reduce personnel cost.

Where do you see opportunities to apply parallel patterns to program your organization?

References

[1] D'Aveni, R./Canger, J. M./Doyle, J. J. (1995): Coping with hypercompetition: Utilizing the new 7S's framework, in: Academy of Management Executive, Vol. 9 (1995), No.3, pp. 45–60.

[2] Galbraith, J. R. (1974): Organization design: An information processing view, in: Interfaces, Vol. 4 (1974), No. 3, pp. 28–36.

[3] Gamma, E./Helm, R./Johnson, R./Vlissides, J. (1994): Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley.

[4] Johnson-Laird, P. N. (1988): The Computer and the Mind: An Introduction to Cognitive Science. Cambridge, MA: Harvard University Press.

[5] Kaeli, D./Yew, P.-C. (2005): Speculative Execution in High Performance Computer Architectures. Boca Raton, FL: CRC Press.

[6] Mattson, T. G./Sanders, B. A./Massingill, B. L. (2004): Patterns for Parallel Programming, Addison-Wesley Professional.

[7] Moore, G. E. (1965): Cramming more components onto integrated circuits, in: Electronics, Vol. 38 (1965), No. 8, pp. 114–117.

[8] Newell, A./Shaw, J. C./Simon, H. A. (1958): Elements of a theory of human problem solving, in: Psychological Review, Vol. 65 (1958), No. 3, pp. 151–166.

[9] Newell, A./H. A. Simon (1972) Human Problem Solving. Englewood Cliffs, NJ: Prentice-Hall.

[10] Searle, J. R. (1980): Minds, brains, and programs, in: Behavioral and brain sciences, Vol. 3 (1980), No. 3, pp. 417–457.

[11] Tushman, M. L./Nadler, D. A. (1978): Information processing as an integrating concept in organizational design, in: Academy of Management Review, Vol. 3 (1978), No. 3, pp. 613–624.

[12] TOP500 (2014): TOP500 Supercomputer Sites, online: http://www.top500.org, as of 2014-07-01.