Modern Trends in Parallel Computing

Peter Salhofer
FH JOANNEUM
peter.salhofer@fh-joanneum.at

1. Introduction

Parallel Computing was a central part of computer science for decades. One of the major milestones in this special field was the introduction of the ILLIAC IV supercomputer in 1971. It was the first massively parallel supercomputer and since then parallel computing was basically reserved for application areas that required enormous computing performance like weather simulations, fluid dynamics, computational biology and so forth. Whereas in the beginning, parallel computer hardware was made of expensive special hardware, some newer supercomputers consist of relatively inexpensive clusters of workstations (commodity clusters).

So parallel computing was mostly connected to super- and grid-computing, while most of today\'s standard business applications written in classical system programming languages are characterized by a sequential execution model. This wasn\'t a problem, as long as necessary performance gains could be achieved by the effect of Moore\’s law that assumes the number of transistors on a chip to approximately double every two years. This was also accompanied by a steady increase in clock rates. So programs grew faster just by running on faster hardware. Since 2012, however, improvements in the production of integrated circuits cannot keep up with Moore\’s law any longer. So instead of getting faster hardware, we get “more” hardware by an increased number cores that can be found on contemporary commodity computers. However, by introducing four cores instead of one, a sequential program doesn\'t get four-times faster, since it typically cannot utilize more than one CPU. Amdahl\’s law perfectly describes this effect and basically states the obvious, that a program can only take advantage of a parallel infrastructure if significant parts of its implementation support parallel execution.

This makes parallel programming a concern of the average software developer. On the other hand, however, a broader acceptance of parallel programming needs easy-to-use abstractions with an economical learning curve and error-prevention by design. One of these approaches might be the renaissance of functional programming resulting in the introduction of classical functional programming constructs in widespread system programming languages like Java and C#.

2. Minitrack Content

The papers in this minitrack perfectly cover the issues raised in the previous section. The first paper presented is titled “Tuning OpenCL Applications with the Periscope Tuning Framework“ and it deals with reducing the complexity of a particular aspect of parallel computation. When executing highly parallel algorithms on specific hardware it can be crucial to have the optimal settings for the target infrastructure already during compile time. This will allow the operating system on the target machine to distribute parallel tasks in an optimal way. The paper presents an algorithm that helps developers to automatically find the best combination of compiler flags so that the compiled program as perfectly suited to be executed with highest possible performance at a given infrastructure. Besides various compiler flags it also determines the optimal settings for the NDRange.

The second paper titled “OpenMP is Not as Easy as It Appears“ investigates the teaching and learning process in the context of parallel programming. As pointed out in the introduction, easy-to-use abstractions are needed to make parallel programming more common. One of these approaches is OpenMP. The paper, however, points out that it is apparently not enough to focus on such higher-level frameworks, but to make students aware of the bare basics behind parallel execution, since otherwise they are likely to run into severe errors.

We will probably see more of such studies in the future, since parallel programming is bound to become a central part of software engineering in general that no developer can afford to ignore in the near future.