The Past, Present and Future of High Performance Computing

Ruud van der Pas[1]

Sun Microsystems, Technical Developer Tools
16 Network Circle, Mailstop MK16-319, Menlo Park, CA 94025, USA
ruud.vanderpas@sun.com

Abstract. In this overview paper we start by looking at the birth of what is called "High Performance Computing" today. It all began over 30 years ago when the Cray 1 and CDC Cyber 205 "supercomputers" were introduced. This had a huge impact on scientific computing. A very turbulent time at both the hardware and software level was to follow. Eventually the situation stabilized, but not for long.
Today, there are two different trends in hardware architectures and have created a bifurcation in the market. On one hand the GPGPU quickly found a place in the marketplace, but is still the domain of the expert. In contrast to this, multicore processors make hardware parallelism available to the masses. Each have their own set of issues to deal with.
In the last section we make an attempt to look into the future, but this is of course a highly personal opinion.

Keywords. High Performance Computing, HPC, MPI, OpenMP, GPGPU, Multicore

1 Introduction

In this paper we review the past and present of "High Performance Computing", (HPC), followed by an outlook into the future.

The next section fairly extensively explores the past of HPC. This is not only because it allows us to better understand the current situation, but also anticipate what the future might bring.

In section 3 we summarize the present situation, but make a distinction in time here. Until not so long ago, the HPC landscape was fairly stable. Recent changes caused quite a disruption, giving rise to two different and diverging trends.

In the last section we try to project how these trends might evolve and what the future could bring.

2 The Past

High Performance Computing (commonly abbreviated as "HPC") started in the mid 1970's. Those days, certain computer systems were specifically designed

Dagstuhl Seminar Proceedings 09061
Combinatorial Scientific Computing
http://drops.dagstuhl.de/opus/volltexte/2009/2083

for scientific computing and their performance was so impressive that the term "supercomputer" was coined and stuck for a relatively long time.

The two most well-known initial representatives were the CDC Cyber 205 and Cray 1 [1] systems, very expensive, but extremely powerful, mainframes. Their peak performance was 200 Mflop/s ("Million Floating Point Operations per Second") and 160 Mflop/s respectively. This is very low compared to what is available today, but a revolution a little over 30 years ago.

These designs were optimized for operations on vectors, but there were vast architectural differences. In both systems, vector processing was tightly integrated to perform basic operations on entire vectors very fast and efficiently. The vector units could, for example, add two vectors or perform an update on a vector using one or two instructions. An instruction that could not be handled by the vector hardware was executed in the more conventional and much less powerful scalar unit. Due to the need to quickly move the data in and out, the memory subsystem had a very high bandwidth.

Given vector based algorithms are rather dominant in scientific computing, the choice to optimize the design for these kind of operations should not come as a surprise.

This design choice was not only a strength, but also a weakness. In order to get good performance out of these systems, a very significant portion of the execution time had to be spent in the vector hardware.

This was relatively easy to find in certain classes of algorithms, in particular numerical linear algebra methods. Even in these favorable cases, achieving near peak performance could however still be hard and often required non-portable extensions to be used.

In a certain way, life was easy. For a while, there were two main architectures to choose from. The programming language was Fortran, but in some cases assembly coding was needed to get the full performance benefit. Certainly compared to more modern processors, these vector processors were fairly simple and straightforward. The performance was often even predictable by just analyzing the assembly code.

There was also another side of this coin though. Tapping the full performance potential was very hard in many cases. In order to obtain a substantial percentage of the peak performance, a significant portion of the execution time had to be spent in the vector units, not the scalar part of the system. The common phrase for this was "the application had to be well vectorized". Unfortunately, Amdahl's Law is hard to beat. This law states that the fraction of the execution time that is not significantly optimized, limits the performance gains. For example, if 20% of the total time is not vectorized, the program can never be more than 5 times faster, regardless how fast the vector units are.

The software environment was very bare bones compared to what is available today. Compiler technology was in its infancy, forcing users to resort to all sorts of low level and often non-portable tricks. Batch queues had to be used to submit jobs, sometimes even with an architecturally different front-end system, making

it very hard to develop and test the application prior to running the production job(s).

A few years later the landscape started to change. Several Japanese vector systems became more widely commercially available. Not only did they deliver impressive performance, the software had also improved. Meanwhile companies like CDC and Cray also continued to innovate and improve their designs, as well as improve the development environment.

At the same time, attached co-processors gained popularity in certain markets. Although much lower in cost, it was not easy to program these systems.

Then something interesting happened. The first "mini supercomputers" appeared. Often they were architecturally very similar to the Cray design and that is why they were also sometimes referred to as "Baby Crays".

Although not as powerful as their bigger counterpart, they offered an attractive alternative to many. Not only were they much cheaper, they also used an interactive (Unix) Operating System. This was augmented by easier to use software for application development and performance tuning. It is therefore no surprise these systems were quite successful, although they never entirely replaced the more powerful vector systems.

At about the same time, the first parallel systems appeared. The vector processors from Cray, but also from mini supercomputer vendor Convex for example, were turned into a shared memory architecture with 2-8 processors.

An interesting architecture was developed and marketed by Alliant. It did not use vector technology, but had a shared memory architecture based on Motorola processors. Transparent to the user, the system had both interactive, as well as execution processors.

Also distributed memory parallel systems became commercially available. The most well-known examples of the this type of computer are probably the Intel Hypercube and the Connection Machine series ("CM") manufactured and sold by Thinking Machines.

Market acceptance of parallelism was however, and unfortunately, low to poor. In a way the hardware technology was very much ahead of its time. To make matters worse, the parallel programming models were all different and proprietary, greatly inhibiting application portability.

By the mid 80's a very disruptive change took place. The "killer micro" appeared on the supercomputing stage and never left again.

Initially, the commodity microprocessors of those days were no match for the dedicated and highly tuned vector systems in particular.

Soon however, this situation started to change. New and improved designs came to market in rapid succession. The vector processors could not keep up with this pace. Although still substantially faster, the performance gap slowly, but steadily, reduced. The volume element also started to play a role. For the same amount of money, one could purchase significantly more microprocessors.

On top of that, these processors were used in small shared memory systems. With the MIPS R8000 processor and Power Challenge system, Silicon Graphics was a clear leader in this area.

Thanks to the relatively low price, it was attractive to cluster microprocessor based systems to form a parallel computer.

The emerging architectures were sufficiently different to warrant a new description of this part of the market. The more general, but still specific, "High Performance Computing" (HPC) term was increasingly used and is common terminology to the present day.

Although by this time substantial improvements in compiler technology had been made, especially regarding serial performance and vectorization, parallelization was still lagging behind.

Automatically parallelizing compilers for shared memory systems were available, but their use and capabilities were limited. In addition to this, compilers supported proprietary extensions to explicitly express and control parallelization both for shared memory, as well as distributed memory programming.

Although there were many common features, the differences in functionality and syntax were substantial enough to seriously hinder portability.

3 The Present

For a while, HPC has been going through a stabilizing period. Until very recently, the landscape converged to a fairly standard set of components, both at the hardware, as well as software level. Then things started to change in a rather disruptive way. This is why we would like to distinguish between two notions of "the present" in the next two subsections.

3.1 The Previous Present

Over time, market forces and the realization that parallel programming models should be standardized, created a more stable situation.

At the hardware level, the majority of HPC systems were eventually based on microprocessors, no longer on vector technology. As one might expect, the main features of the latter crept into commodity hardware, although substantial differences remained. Especially regarding the memory subsystem.

A significant feature all HPC systems have in common is that they all consist of clusters of systems. The size and architecture of the "nodes" may be vastly different, but the choice regarding the interconnect is limited.

Aside from the microprocessor taking over, the biggest revolution took place at the software level. In addition to Fortran and C, C++ became a supported and more commonly used programming language in HPC for example.

The real breakthrough was in the area of parallel programming, however.

The first big step forward was made through the "Parallel Virtual Machine" (PVM) library. For the first time, one could use a portable library to write and execute a parallel program on a distributed memory system. Many platforms were supported through PVM and it quickly became very popular as the programming model of choice for a cluster of relatively small sized nodes, like a cluster of PCs.

Despite this success, PVM has its limitations and there were still competing programming models around.

The "Message Passing Interfacing" (MPI) [2] project addressed this. Several groups and individuals joined forces to define a complete, powerful and portable standard for distributed memory parallel computing. The first specification was released in 1994, three years later followed by MPI-2. MPI has been extremely successful and remains the distributed memory programming model of choice in HPC.

Luckily, very good progress was made on the shared memory side as well.

In particular, OpenMP [3] has established itself as the main programming model for shared memory parallel systems. Similar to the situation for distributed memory systems, acceptance of shared memory parallel programming was seriously hindered by portability issues, as well as significant differences in functionality.

From the first specification in 1997, OpenMP provided a small, yet powerful and portable, programming model for shared memory systems. The tasking concept introduced with the 3.0 specification released in 2008 makes it possible to more conveniently parallelize a wider range of applications.

3.2 The Current Present

The reasonably stable situation sketched in the previous section did not last very long however. Two trends in hardware development create a bifurcation in today's trend in HPC.

The increasingly powerful graphics co-processor in systems attracted attention to be used as an additional compute device. In just a few years, interest in this technology increased dramatically. These kind of processors are usually referred to as "GPGPU" (General Purpose Graphics Processing Unit) or "GPU" for short. They have several restrictions, but a huge performance potential. Encouraging results are realized. As getting the data in and out is relatively slow, they are currently more suitable for algorithms and applications that are more CPU than I/O dominated.

This trend does not stop with a single accelerator card. These processors can be used as a building block in a massively parallel system with a primarily distributed memory architecture. For performance reasons there could also be a relatively small portion of shared memory. This can, for example, be used for fast data exchange across the parallel system.

Several such products are available on the market today and have found their own place in the HPC market.

The programming model is still an issue. Although the recent OpenCL [4] standard may help, programming for such systems is still specific to the target architecture and the application can not be used on a general purpose system. Tapping the potential also requires quite a lot of low level detail at the application and architecture level for the developer to consider. Therefore, this is still the domain of a small group of expert HPC developers.

At about the same time, the opposite trend also occurred. Thanks to advances in process technology, chip designers were able to put multiple cores onto one processor.

Simply said, a core is a piece of hardware that can execute an application. There are vast differences across the various designs, but for our purpose it is sufficient that a multicore processor can be seen as a (small) parallel system. Under the hood a fairly complex cache subsystem can be found, consisting of caches exclusive to the core, as well as shared caches accessible to all cores. Thanks to cache coherence, this is transparent to the developer, turning a multicore processor into a shared memory parallel system.

This has huge implications, however. Multicore even makes a laptop or PC, a small parallel computer. After two decades, parallelism has reached the masses.

What is equally important is that the existing software environment can be used to program such systems. A native threading model, OpenMP, or MPI, are all available and at the program development level there is no need for changes. This means that previous parallelization efforts are preserved.

There is, however, one major difference compared to a single core system: parallel programming requires a new set of skills and most developers lack this background. It will take some time before a large number of developers "think parallel", but the arrow points in one direction only. An increasing number of applications will take advantage of the parallelism made available through multicore technology.

4 The Future

In the final section of this paper we try to look into the future, but obviously this outlook is limited by the personal view of the author.

For some time, it can be expected that the gap in the bifurcation continues to widen.

The GPUs will increase in computational power and lift restrictions over time, although the relatively slow connection with the main processor will remain to be a bottleneck. As more and more developers are attracted to this platform, there will be more of an incentive to ease the application development cycle. Already the first encouraging signs of an easier to use development environment appear, but this is still in its infancy.

Meanwhile, hardware alternatives are explored, for example by combining a very fast special purpose compute device with a multicore architecture. This provides a much tighter hardware integration with the associated performance benefits. There are several serious hurdles to be taken still though. In particular there are challenges at the software level, but these are not insurmountable and might provide a very viable alternative to the current GPGPU based parallel systems.

Regardless of this, and recalling history, it is probably inevitable that the current situation will evolve to a much more integrated architecture. This will

lower cost, improve performance and accelerate wider adoption. It is not hard to see the benefits of these features and therefore this is likely to happen.

The multicore trend forces developers to think about parallelization, but parallel programming remains hard. Admittedly, tool support has improved, but identifying the parallelism, getting the correct results and realizing scalable performance are challenges not to be underestimated.

Interestingly, these are the same problems the GPGPU developer is faced with. The key difference is that this person often has some, or substantially, more experience parallelizing an application. Eventually, the less experienced developer for multicore will get there too though.

As has always been the case, the leapfrog model ensures that architectures and software converge. Tighter hardware integration will address the bottlenecks of today and disjoint programming models converge to a unified standard. This is when the two trends will start to grow towards each other, eliminating the current bifurcation.

In the mean time, parallel programming has never been as exciting as it is today. By the time the bifurcation is closed again, there are undoubtedly new and very complicated challenges to address, because that is probably the best definition of "High Performance Computing".

References

1. *The Supermen: The Story of Seymour Cray and the Technical Wizards Behind the Supercomputer*, John Wiley and Sons, ISBN: 978-0-471-04885-5, 1997.
2. http://www.mcs.anl.gov/mpi
3. http://www.openmp.org
4. http://www.khronos.org/opencl