easily give rise to the misguided idea that excellent software is free. The open-source software movement further reinforces the idea that all software should be free. There are excellent examples of open-source software projects — such as those supported by the Linux, Apache and Eclipse foundations — and these are typically of high quality, fully documented and supported. Similarly, GitHub now has over three million users and supports over 16 million repositories, most of which contain open-source code. However, much of open-source scientific research software is often of poor quality, inefficient, undocumented and unmaintained. Producing high-quality software that is well documented, debugged and easy to install and use is expensive. Merely making scientific software open source is not a guarantee of quality.

Many developers of scientific software receive their training from other scientists rather than from a traditional computer-science software-engineering course. There is still a mismatch between some standard software-engineering environments and methodologies and the practice of working computational scientists developing parallel codes for high-performance computing systems[5]. However, as can be seen from the example of the ATLAS LHC experiment, significant parts of the physics research community certainly use many of the modern tools of software engineers to develop and manage their large code base.

Part of the problem is that the long-tail science researchers are often not well equipped to design, write, test, debug, install and maintain software — having only very basic training in programming. Enter 'software carpentry'. This movement was started by Greg Wilson in 1998 and has now evolved into a worldwide volunteer effort to raise standards in scientific computing. Wilson runs software-carpentry boot camps that focus on four aspects of computational competence[6]. A typical course introduces a handful of basic Unix shell commands, teaches students how to build simple Python programs step by step, emphasizes the benefits of version-control systems and extols the virtues of structured data. As Wilson notes, teaching such basic skills is not very interesting for computer scientists.

Support from above

A key problem in supporting research software development is that funding agencies in many countries do not view software development as an intellectual exercise worthy of a research grant. Instead, in their proposals, scientists have to focus on the research aspects of a particular scientific application and gloss over the fact that software will need to be developed to enable the science. However, both in the USA and the UK, the major research funding agencies have recently changed their policies on funding research software development.

The Advanced Cyberinfrastructure Division of the National Science Foundation (NSF) in the USA has produced a vision and strategy for supporting research software[7]. The plan is to address research software sustainability issues though a tiered approach. First, the NSF will form small groups to design software elements tailored for particular advances in science and engineering. Second, larger teams will be enlisted to build software infrastructure that is accessible to diverse scientific communities. The NSF intends to create hubs of excellence in software infrastructure and technologies. And finally, it plans to provide incentives for individuals and communities to build on existing software frameworks in their software development.

In the UK, the Research Councils have also recently changed their policy on software-development costs. In addition to providing support for the UK Software Sustainability Institute, the Engineering and Physical Sciences Research Council (EPSRC) now issues regular calls for proposals that are focused purely on either developing new and innovative software — adding novel functionality to existing software, or simply making widely used software packages more efficient and/or robust (http://go.nature.com/XFCWXy). The EPSRC also now offers personal fellowships specifically for individuals who specialize in software development. These are encouraging signs, but there is a long way to go before skilled developers of research software achieve parity of recognition with their partner research scientists. ❐

Tony Hey is at the eScience Institute at the University of Washington, Seattle, Washington 98195, USA. Mike C. Payne is in the Cavendish Laboratory at the University of Cambridge, Cambridge CB3 0HE, UK. e-mail: tony.hey@live.com; mcp1@cam.ac.uk

References

1. Nowakowski, P. *et al. Procedia Comput. Sci.* **4,** 608–617 (2011).
2. Koop, D. *et al. Procedia Comput. Sci.* **4,** 648–657 (2011).
3. Bell, C. G. *Comput. Sci.* **1,** 4–6 (1987).
4. Stodden, V. *et al. Setting the Default to Reproducible: Reproducibility in Computational and Experimental Mathematics* (ICERM, 2012); http://go.nature.com/nacwjm
5. Basili, V. R. *et al. IEEE Software* **25,** 29–36 (2008); http://go.nature.com/xTSILw
6. Wilson, G. *F1000Research* **3,** 62 (2014); http://go.nature.com/8V6ui9
7. *A Vision and Strategy for Software for Science, Engineering and Education* (NSF, 2012); http://go.nature.com/M29xjk

Programming revisited

Thomas C. Schulthess

Writing efficient scientific software that makes best use of the increasing complexity of computer architectures requires bringing together modelling, applied mathematics and computer engineering. Physics may help unite these approaches.

For half a century, the performance of computers has been improving exponentially. These impressive developments in scientific computing — which most scientists now take for granted — are documented by the Top500 project (www.top500.org). Since 1993 this website has been monitoring how the fastest computing systems solve dense linear equations with the high-performance LINPACK (HPL) benchmark and the sustained performance for HPL-type problems has been increasing roughly by three orders of magnitude per decade.

This improvement in performance is not limited to high-end supercomputers, but permeates all information technology infrastructures available to scientists today. For example, the current HPL performance of typical smartphones is comparable to that of Cray X-MP supercomputers in the late 1980s. Hard computational problems that originally required unique computing infrastructures become solvable within years on regular computers, and typically a decade or two later can be handled by commodity devices most people around the world carry in their pocket.

© 2015 Macmillan Publishers Limited. All rights reserved

المنارة للاستشارات
www.manaraa.com

Whereas this long-term sustained exponential growth had profound impact on the productivity of scientists and opened many new avenues in physics research, not all types of problems in scientific computing have seen the same performance improvements. For example, the sustained performance of climate codes, as documented by the European Centre for Medium-Range Weather Forecasts (ECMWF) over approximately the same period as the Top500 project, has improved only by a factor of 100 per decade (Peter Bauer, manuscript in preparation). This is still an exponential growth, but it demonstrates the significant decrease in efficiency for software applications in some fields. This is more important, as meteorological and climate simulations have been around since the dawn of modern computing[1]. They rely on complex, but typically well-engineered computer codes that have been designed to run on the top supercomputing systems. If experts use computers inefficiently, what does this say about the applications developed by regular researchers?

In this Commentary, I discuss state-of-the-art programming and parallelization in physics today. I try to analyse the challenges in writing efficient scientific software and examine possible ways in which physicists can deal with the rapidly increasing complexity of computer architectures. To do so it is important to first recall the main uses of computing in physics.

Predictions and data analysis

Long before the advent of modern computing, modelling and simulation were used in physics in two ways. The first and best known (which we call the traditional way) is the use of computers to solve challenging theoretical problems that have no known analytical solution. In this case, the theory is well understood and the governing equations are solved numerically with elaborate computational methods to make quantitative and verifiable predictions. Sometimes the numerical solution of a theoretical problem may lead to new insights in its own right, as was the case with the discovery of the fluctuation theorem[2]. This was an argument for defining computer simulations as a third, independent pillar of science, complementing theory and experiment[3]. For our purpose, this distinction is not necessary, as from a computational point of view we are still solving known equations. The simulations are carefully planned — that is, the mathematical analysis and algorithms are well known and the elaborate computer codes, as in the case of climate simulations, have been developed and optimized. Scientists, and physicists in particular, will not shy away from great efforts in using cutting-edge technologies to solve such problems, and they will use imperative programming languages such as C or FORTRAN with machine-level codes to squeeze every last bit of performance out of a computing system.

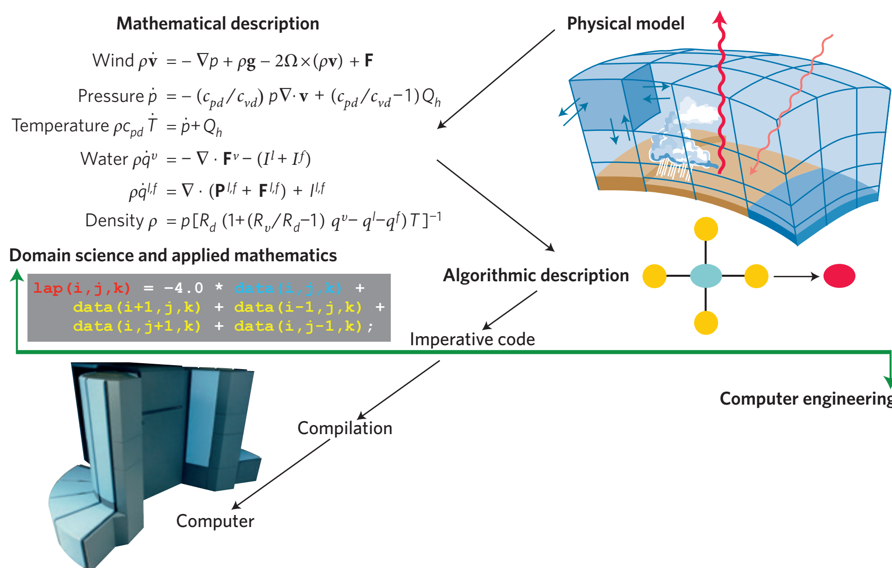

Figure 1 | Traditional computational science workflow. A physical model (of the atmosphere, for instance) is first translated into mathematical equations (here, non-hydrostatic compressible Euler equations), which in turn are solved with algorithms (such as finite differences on a structured grid), implemented in a program (for example, stencil code), and subsequently compiled into machine code that executes on a canonical computer architecture. The green line marks the separation of work. The physical model image is adapted from ref. 9, NPG. The supercomputer image © British Crown Copyright, The Met Office / Science Photo Library.

The second, and profoundly different, use is the analysis of experimental data with the help of modelling and simulations before the theory and governing equations are known. This is essentially what Johannes Kepler did when he analysed Tycho Brahe's planetary orbit data with heliocentric elliptical models to discover the three famous laws that now carry his name — Newton's theory of gravitation, which explains Kepler's laws, came later. Scientists today use computers to rapidly prototype models, thereby assimilating in a matter of seconds or minutes many orders of magnitude more data than Kepler did in months of laborious manual computations. Along with the development of electronic computing came large experimental facilities, which significantly increased the importance of systematic exploratory tools for data analysis. This lead to a substantial improvement of mathematical algorithms over the past few decades, which, together with the emergence of social media on the World Wide Web, have made this exploratory use relevant to areas outside of natural sciences, for instance in economics and social sciences. These have, in turn, led to the argument that a fundamentally new, fourth paradigm of science is emerging: 'data science'[3]. For our present purpose, however, this distinction is again not necessarily important. But, for this second exploratory use of modelling and simulation scientists use more descriptive programming languages like Python or Ruby, and they rely on existing libraries even if they are not optimized.

Scientists and computer engineers

Programming serves two complementary purposes: one is to specify the computation and the other is to manage computer resources. Most scientists are familiar with the former, whereas the latter is considered to be primarily the concern of computer engineers. The distinction is important as it allows a clear separation of concerns: scientists only need to know about the complexity of models and mathematics, and system engineers only need to focus on the complexity of the computer.

In this ideal case, the programming environment allows scientists to specify the computational tasks in terms of human-readable equations — descriptive programming — that are independent of the underlying system, which is portable across many platforms. The Python programming language, with its many associated libraries and tools, provides such an environment, but at the cost of performance. When the computation is big and has to be scaled, performance does matter. In this case scientists have the choice of algorithms

© 2015 Macmillan Publishers Limited. All rights reserved
المنارة للاستشارات
www.manaraa.com

and may want to specify many more details by using imperative programming languages such as FORTRAN or C/C++. This imperative approach can be productive and for many years FORTRAN has been the *lingua franca* in computational physics because it provided a good compromise between programmer productivity and performance. It has become embedded in a well-established workflow (Fig. 1). But a more subtle issue relates to the portability of codes.

The portability of imperative codes is not an issue as long as all computing platforms adhere to the same abstraction and suitable compilers are available to translate the user code into efficient machine code. This was certainly the case in the 1950s and 1960s, when imperative algorithm description languages were first developed and the von Neumann model still provided a good abstraction of all computer systems of the time. Even in the 1970s, when vector architectures became available, scientists could still create portable code without too much trouble. In theory, compilers would vectorize the code, but in practice the programmer had to pay attention to the layout of the data and choose the corresponding loop order to help the compilers discover sections of the code that could be vectorized. Nevertheless, as long as only one of the architectures deviates slightly from the canonical von Neumann model, both code portability and good performance can be achieved.

The break of traditional order

Things began to fall apart in the late 1980s with the introduction of distributed memory systems, which later became known as massively parallel processor (MPP) arrays — computers with, by the late 1990s, thousands of processors. Scientists faced two challenges: parallelism and the distributed memory architecture.

Initially, parallelism was the biggest concern as there is a theoretical limit to its benefit to performance called Amdahl's law — the speed-up of a parallel implementation cannot be better than $1/s$, where s is the serials fraction of the work. However, this turned out to be a minor problem in many physics applications, in which sufficient parallelism can be identified and the serial fraction is small enough to allow scaling of computations to thousands or more processes. Certainly, parallel algorithms have to be developed, but this is not the most significant problem from a portability point of view.

The harder problem — which still eludes an elegant solution — turned out to be the memory architecture. Pragmatic scientists would use libraries to adapt a code to the memory architecture of an MPP system and portable libraries emerged in the 1990s. Currently the most popular is the Message Passing Interface (MPI)[4] that initially implemented a two-sided model, where processes called MPI tasks would exchange information through messages with send and receive methods. Another notable effort was the Symmetric Hierarchical Memory (SHMEM) access library that implemented a one-sided communication model. SHMEM was industry-lead and is still in use today, despite the fact that it was not standardized in its early stages. The broader supercomputing community initiated MPI and released its first standard in 1994.

MPI is so dominant in parallel computing today that the science community often mistakes it for a parallel programming model. But MPI is a low-level library to handle data movements between distributed memory nodes and should rather be considered part of the runtime system of a programming environment. By using MPI in a program, the scientific programmers cross the line from specifying what needs to be computed to managing resources of the computing system. This can introduce portability problems if not done carefully. MPI is often thought to be difficult to use — which is clearly not the case — because scientists are not aware that they have crossed the line that formerly separated them from computer engineers.

Increasing diversity

MPP was just the first step towards more complex architectures that would no longer be hidden from the user. The next step concerned the memory wall, that is, the many orders of magnitude discrepancy between cycle time (on the order of nanoseconds) and memory access latency, which is measured in tens or hundreds of cycles. To hide memory latency as much as possible, processors now have a deep memory hierarchy (memory cashes) and sophisticated runtime systems are built to exploit temporal and spatial locality of the application codes. This is, in principle, transparent to the user. However, for all practical purposes, algorithms have to be specially designed and tuned with memory hierarchy and cash sizes in mind to maximize computational performance on a particular processor.

Just short of a decade ago, the number of processing cores in microprocessors began to increase exponentially as a consequence of the end of Dennard scaling, which has allowed transistors to shrink and improve performance while using less power. Furthermore, to accommodate higher transistor densities at constant energy density, the clock frequency has been steadily decreasing. Thus, individual cores in new processors are slower than in previous generations. All applications now have to exploit parallelism or lose performance on newer generation processors.

Multi-core processors are programmed with a shared-memory threading model, of which a number of portable ones exist today. The best known is OpenMP, but the new C++11 thread library is also rapidly gaining popularity. Besides supporting multiple threads, almost all microprocessors used

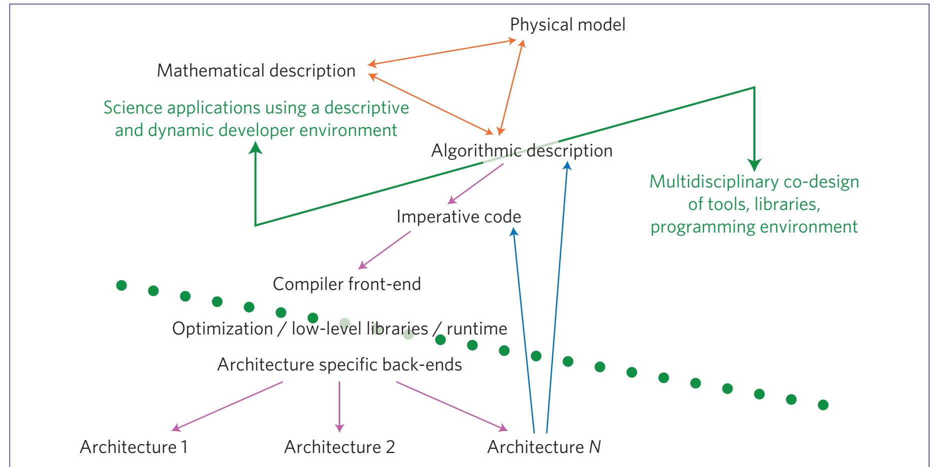

Figure 2 | A new programming workflow. The workflow that takes into account the present reality with multiple, diverging computer architectures: models, their mathematical description, as well as algorithms are developed iteratively in a descriptive environment; algorithms and their imperative implementations are specialized for specific architectures. The green dotted line marks the separation between parts of the software stack developed specifically for scientific computing and the more generic parts used in other markets as well.

© 2015 Macmillan Publishers Limited. All rights reserved
www.manaraa.com

in scientific computing today have vector registers. Like in the old days of vector machines, programmers have to rely on the compiler to vectorize codes, and try to help through proper choices of data layout and loop annotation, or processor specific vector instructions. Writing programs that vectorize well remains a difficult task and the resulting computer codes are not necessarily performance portable either. An interesting development that has recently gained significant popularity in physics is NVIDIA's Compute Unified Device Architecture (CUDA), a parallel programming model based on C/C++ language and libraries to program graphics processing units (GPU). In the CUDA model, rather than relying on the compiler to detect vectorizable parts in otherwise serial code, programmers write code from the view of individual, fine-grained threads. The serials code generated by the CUDA compiler is bundled by the hardware into small groups of threads executing in lockstep — with a single instruction, hence the term single instruction multiple threads (SIMT) — and control flow divergence is managed transparently to the programmer. This bypasses the vectorization challenge, and in practice, algorithms that have been redesigned for the CUDA model perform much better on vector processors as well. As open source compiler projects such as GCC and LLVM begin to support SIMT programming, it will be interesting to see whether a new programming environment can emerge that also provides better portability to vector processors.

Today's supercomputers, and thus most of the future scientific computing systems at all levels, are combinations of distributed memory clusters with massively multithreaded nodes that have complex and heterogeneous memory architectures. Thus, whether the node-level threading model X is OpenMP, C++11 threads, CUDA or something else, the pragmatic programmer will resort to MPI+X on an MPP system. But, this is still an uneven mixture of a proper programming model X and the low-level library MPI. A large number of projects have encountered this problem, and partitioned global address space (PGAS) languages like X10, Chapel, and UPC have been designed to support distributed multi-dimensional arrays, for instance. These new languages have not had much traction and have not yet been adopted by larger software development projects. However, many large software projects and experienced developers that rely on MPI as a low-level library will develop a front end for their particular domain. Over time these solutions develop into domain specific libraries or languages (DSLs) that may even exchange MPI with some other low-level library. The DSLs provide a much higher-level interface, where their maintainers, along with vendors, assure performance portability across platforms. An early example of domain specific library development is the NWChem quantum chemistry code[5] that motivated the Global Arrays library[6], which is now a commonly used PGAS solution for distributed arrays.

Where to go from here

With technology reaching the end of Moore's law scaling around the conclusion of the decade, architectural diversity and complexity will likely continue to increase. On-node parallelism will grow for a few more processor generations and may reach around 10^5 or more threads per node. More importantly, there will be even greater heterogeneity in memory than today, and the need to focus on data locality, as well as minimize data movement in algorithms, will persist. This is a profound change compared with computing in the von Neumann paradigm, where the programmer simply had to acknowledge that memory exists because the problem had to fit to the available resources. Today and increasingly so in the future, programmers will have to design algorithms that properly map data on the distributed memory, avoid communication as much as possible, and make optimal use of the memory hierarchy. The mathematical formalism to express locality in algorithms still needs to be developed though, as present methods to analyse complexity focus only on computational costs.

With these developments the canonical workflow shown in Fig. 1 would be unsustainable even for the traditional use of scientific computing, where simulations dominate, resulting in fewer programmers being able to take advantage of new computing technology. Several well-established simulation domains deal with complex, continually evolving models (like climate and earth sciences) or require rapid prototyping of new algorithms to manage computational complexity (like quantum chemistry or condensed matter physics). Thus, it seems more practical to look at a new workflow, from the point of view of the exploratory use of computing systems. This is depicted in Fig. 2, where we acknowledge that in science, the models, their mathematical analysis and algorithmic implementation evolve iteratively. Algorithms still have to be implemented with imperative code and compiled on a specific machine, but the choice of algorithms and their implementation will depend on the architecture used. If this consequence of diverging architectures is accepted, one ends up with a different separation of work.

The separation between descriptive and imperative programming models is both natural and important. Most exploratory use of computing systems should be able to rely entirely on descriptive developer environments like IPython[7]. These environments can be engineered as efficient, high-performance computing tools, provided that common algorithmic motifs are well implemented and available. The guiding example for Python should be numpy, a library that implements multi-dimensional arrays: a dense matrix multiplication implemented in Python will perform poorly, but with numpy arrays and appropriate back-end methods the matrix multiplication can be as fast as the best blocking implementation available that makes modern supercomputers perform well with respect to the HPL benchmark.

This should be repeated for other common algorithmic motifs, most of which are already quite well understood in terms of existing DSL implementations. Similar to linear algebra implementations that at some point were unified into what we now know as the basic linear algebra subroutines (BLAS) and LAPACK[8] — which one could consider to be the first successful DSL — systems should be implemented and designed for other motifs. Eventually DSLs other than BLAS will begin to dominate. They will provide new benchmarks that, like HPL today, will drive the development of efficient computing systems for the areas that have been falling behind.

A second line probably needs to be drawn between the part of the system that is designed specifically for scientific computing and some of its parts, such as processors, memory, and, in the near future, even networking components that are designed and produced primarily for high-volume markets, bearing much of the development costs. Relying on such high-volume parts can cut costs and leave more resources to be invested in the packaging of scientific computing systems, including the development of software tools.

Such a reorganization of the concerns in how computing systems are developed and built would raise scientific productivity because of both higher performance and more efficient programmability in a descriptive sense. However, it will require the various subdomains of the scientific computing community, the domain sciences, applied mathematics, and computer science, to grow together in a much more multidisciplinary way. Scientists will have to give up on the idea that legacy codes must run forever. Computer scientists and engineers will have to make some low-level libraries and parts of the runtime systems

© 2015 Macmillan Publishers Limited. All rights reserved

المنارة للاستشارات www.manaraa.com

more accessible, just as they did with MPI. And mathematicians should provide accessible descriptions for data centric analysis, which is not a trivial task. Perhaps physicists can take the lead in gluing these pieces together, as they did for the MANIAC computer in the 1950s. ❐

Thomas C. Schulthess is at Institut fur Theoretische Physik, ETH Zurich, Zurich 8093, Switzerland. He also directs the Swiss National Supercomputing Centre in Switzerland and holds a visiting distinguished scientist appointment at Oak Ridge National Laboratory in the USA.
e-mail: schultho@ethz.ch

References

1. Lynch, P. *The Emergence of Numerical Weather Prediction: Richardson's Dream* (Cambridge Univ. Press, 2006).
2. Evans, D. J., Cohen, E. G. D. & Morriss, G. P. *Phys. Rev. Lett.* **71,** 2401–2404 (1993).
3. Hey, T., Tansley, S. & Tolle, K. (eds) *The Fourth Paradigm: Data-Intensive Scientific Discovery* (Microsoft Research, 2009).
4. Message Passing Interface Forum. MPI: A Message-Passing Interface standard. *Int. J. Supercomput. Appl. High Perfor. Comput.* **8,** 165–416 (1994).
5. Valiev, M. *et al.*, *Comput. Phys. Commun.* **181,** 1477–1489 (2010).
6. Nieplocha, J. & Harrison, R. *J. Supercomput.* **11,** 119–136 (1997).
7. Pérez, F. & Granger, B. E. *Comput. Sci. Eng.* **9,** 21–29 (2007).
8. Anderson, E. *et al. LAPACK Users' Guide (Software, Environments and Tools)* (SIAM, 1987).
9. Kutzbach, J. E. & Ruddiman, W. F. *Sci. Am.* **264,** 66–75 (1991).

Acknowledgements

The author thanks Thomas Lippert for insightful discussion on the respective roles of simulations and data analysis, as well as Peter Messmer, Oliver Fuhrer and Torsten Hoefler for important comments on parts of the manuscript. Generous support from the NCCR-MARVEL project of the Swiss National Fund and Oak Ridge National Laboratory is acknowledged.

Look to the clouds and beyond

Sergey Panitkin

Research in high-energy physics produces masses of data, demanding extensive computational resources. The scientists responsible for managing these resources are now turning to cloud and high-performance computing.

The ATLAS experiment[1] at the Large Hadron Collider (LHC) is designed to explore fundamental properties of matter at the highest energy ever achieved in a laboratory. Since the LHC became operational six years ago, the experiment has produced and distributed more than 150 petabytes of data worldwide. Thousands of ATLAS physicists are engaged in daily analysis of this data, and their work has led to the publication of more than 400 papers on various aspects of LHC physics. So how does one go about managing the computational resources required for such a formidable programme of research?

Modern research in high-energy physics is practically impossible without massive computing infrastructure. ATLAS currently uses more than 100,000 CPU cores arranged in a grid spanning well over 100 computing centres. But the next LHC run, which started this month, will require more resources than this grid can possibly provide to sustain the pace of the proposed research. The grid infrastructure will be sufficient for the planned analysis and data processing, but it will fall short of the requirements for the large-scale Monte Carlo simulations that accompany the experiments — as well as any extra activities. Additional computing and storage resources are therefore required. To meet these challenges, ATLAS is engaged in an ambitious programme to expand the current computing model to incorporate additional resources, including supercomputers and high-performance computing clusters as well as commercial and academic clouds.

Grids and clouds

The ATLAS computing model[2] is based on a grid paradigm[3], with multilevel, hierarchically distributed computing and storage resources. The grid model was conceived in the late 1990s and has served ATLAS and the entire LHC physics community well. However, new cloud computing technologies offer attractive features that can help to improve the operation and elasticity of scientific distributed computing.

ATLAS now treats grid and cloud computing as complementary technologies that can coexist at different levels of resource abstraction. The flexibility of the cloud technology allows the representation of distributed resources in a variety of ways — ranging from small stand-alone private or public clouds to a large-scale heterogeneous federation of clouds located on several continents.

Cloud resources, especially those sold commercially by the likes of Amazon and Google, can temporarily augment grid resources at times of escalating demand for computing power. Such spikes in demand exceed the level of available ATLAS computing resources by an order of magnitude at times, and are expected to grow in amplitude and frequency during the next LHC run. Some of the existing cloud technologies, like OpenStack (http://www.openstack.org) for example, can be used as a new way to manage computing resources at different ATLAS grid sites, thus helping to improve operational efficiency and resource utilization flexibility.

A cloud computing system offers virtualization, which shields applications from disruptive changes in hardware and system software. It also reduces the need for resource-provider personnel to have an intimate knowledge of the user's application — meaning that resource centres need not have in-house expertise in grid computing or ATLAS software to contribute to ATLAS distributed computing. Cloud technologies can also provide a convenient way of dynamically managing resource allocation between multiple projects within a single centre. And they can help to aggregate heterogeneously distributed resources into a unified framework.

To manage the grid, ATLAS makes use of a workload management system designed specifically for distributed data processing and analysis. Typically though, a cloud system requires an additional management layer. This layer may be independent of the way that the computing workload is managed, or it can interface with the management system directly to control the number of virtual machines running. To implement this layer, ATLAS explored several options — and even went as far as creating an R&D group in 2011 to investigate cloud computing technologies and their utility for the experiment. Since then, the cloud R&D project has been transformed into the cloud operations project, which uses a variety of virtualized resources and cloud platforms that are integrated with the distributed computing system.

One of the longest running cloud resources in the ATLAS experiment is a heterogeneous cloud computing system that

www.manaraa.com

© 2015 Macmillan Publishers Limited. All rights reserved

Copyright of Nature Physics is the property of Nature Publishing Group and its content may not be copied or emailed to multiple sites or posted to a listserv without the copyright holder's express written permission. However, users may print, download, or email articles for individual use.

المنارة للاستشارات

www.manaraa.com