



A Survey of Scientific Computing in the UK

Lawrence Mitchell, David Henty, George Beckett

February 26, 2010

Abstract

In this document, we summarise the findings of a recent survey of UK HPC users and numerical analysts. We have concerned ourselves with practical aspects of algorithm and software development along with a study of probable future issues.

1 Introduction

In this short report, we summarise findings of a series of interviews carried out in October and November 2009 by phone and in person with numerical analysts and computational scientists from around the UK as part of the NAIS project. The aim has been to identify areas in which current computational and algorithmic development practices are suboptimal, and to establish how people would like these deficiencies to be addressed.

The survey has been conducted in support of the NAIS Centre's primary objective. This is to introduce and develop a new paradigm for high performance computation based on the combination of advanced annotation and machine-learning compilers, and to apply it to a range of challenging state-of-the-art numerical applications areas, such as Adaptive Finite Element Modelling, Stochastic Multiscale Modelling, Molecular Dynamics, and Optimisation. The ultimate goal is to provide new flexible, portable and efficient tools to the HPC applications community.

In our survey, almost all HPC applications we have encountered are written in standard sequential languages using MPI¹ for parallelisation through message passing. Library use, both parallel and scientific, is generally quite limited – see section 2.4.1 for details. It is widely accepted that this is not an ideal state of affairs. We explore why people program with this paradigm, and how they would like to program in an ideal world. On the numerical-analysis side, we look at how effectively collaboration occurs between algorithm developers and implementors. We consider questions such as, “Where are the sticking points in moving to newer, potentially better, algorithms?”.

This report should not be read in isolation. Over the past few years, a number of reports have appeared attempting to guide development work and thinking in numerical analysis and HPC towards exascale over the next 5 to 10 years. A series of workshops was recently held on this topic, hosted by the Oxford e-Research Centre (OeRC). The findings are summarised in Trefethen et al. (2009). As part of the PRACE project, a survey of HPC usage and applications was carried out in 2008. This was used to inform a choice of applications to be used in a benchmark suite for new supercomputers within the PRACE project; the results are detailed in Simpson et al. (2008). The weighting of different application areas in the benchmark was in terms of flops² used. Particle physics and computational chemistry codes used around 45% of available cycles on PRACE supercomputers. The PRACE report is a good place to start if one wishes to study current HPC practices. Currently, those areas with a long history in high-end computing dominate usage statistics. The studies in these fields are, however, limited to a few areas. There are potentially many more users in the tail of usage distributions who *could* make much more use of HPC systems. They either do not have the codes to do so, or else have not yet realised quite what current computational resources could allow them to do.

Finally, two reports have emerged in the last year looking at the challenges inherent in exascale computing: that is, supercomputers with around 10^{18} flops of performance and 10^6 cores. A DARPA study in 2007 looked at the current state of the art in HPC, identifying likely challenges for larger systems; details are in Kogge et al. (2008). As a result of the drive towards and challenges behind exascale, in 2009 a series of workshops was held to develop a road map for exascale computing, the International Exascale Software Project (IESP). The results of these workshops are gathered in Dongarra et al. (2009).

¹<http://www.mpi-forum.org>

²Flops is an acronym meaning floating-point operations per second.

The IESP³ report focuses specifically on defining a road map for exascale supercomputers over the next ten years. The report studies the challenges that will be faced in both hardware and software design and attempts to identify application areas that can be used to motivate the scientific case for exascale. From a purely hardware point of view, building such a large system is entirely possible, excepting practical constraints of power consumption. If not quite with today's hardware, certainly with hardware in the next five years. The issue is in programming models and software development that can exploit this hardware. The big guiding idea in the report is that of co-design. That is, to exploit exascale systems, software and hardware should be designed (to some extent) in tandem. Software and hardware will have to be designed with continuous feedback from both sides: certainly current codes will be unlikely to scale, without changes, on exascale systems.

The OeRC road map focuses purely on software and cultural challenges. It identifies that development of HPC software needs a tighter coupling between algorithm developers and software developers. It notes that although high performance computing has become a ubiquitous part of many scientists' daily life, the ability to use the machines is decreasing due to increasingly complexity. The study covers a broad range of applications, attempting to identify the common algorithmic patterns and sticking points for future progress. It proposes better integration and collaboration between different areas of computational science: informatics research on abstractions, and code generation; algorithm development; software development. Similarly to the IESP report, it promotes the idea of library and algorithm development driven by close examination of software needs.

A common theme running through these reports is that to utilise the computing potential of modern HPC systems fully, software libraries will have to take a large amount of the programming burden away from the code developer. The current programming model is for the programmer to explicitly manage almost all aspects of the computation (apart from basic linear algebra). With increasingly heterogeneous systems, especially at the high end, controlling parallelism and execution at this level does not scale for the programmer – excepting some few cases where the problem is highly regular. As a result, we need ways of encoding the problem at a higher level with library code hiding much of the complexity.

³<http://www.exascale.org/>

1.1 Format of interviews

The survey has taken the form of short ($1/2 - 1$ hour) interviews with a set of candidates (see appendix B) chosen to give good coverage of Phil Colella’s 7 dwarfs of numerical computing, see Asanovic et al. (2006) for details:

1. Dense linear algebra;
2. Sparse linear algebra;
3. Spectral methods;
4. N-body methods;
5. Structured grids;
6. Unstructured grids;
7. Monte Carlo methods.

In addition to covering these algorithmic building blocks, the interviewees were chosen to cover a range of application areas similar to those identified in Simpson et al. (2008); additionally, we have aimed for a broad range of “types” of computational scientist: end users as well as application developers and numerical analysts.

We have followed the proforma included as Appendix A, to ensure that all interviewees addressed an equivalent set of questions. Apart from the set of questions, the interviews have been reasonably unstructured, exploring areas the candidate found most interesting and relevant.

2 Findings

We have found that we can broadly split the interview candidates into three reasonably distinct groups:

1. Software developers (10)
2. Scientists who write codes (9)
3. Numerical analysts/algorithm developers (9)

The focus of these three groups is quite different. Many scientists have to write codes in the course of their research. For the most part, we have found that they are not interested in the implementation per se. Rather, computers are more like experimental apparatus, allowing the researcher to gather results. A common theme has been “I write in language X because it’s the only one I know”, rather than due to any particular technical advantages. In other words, most scientists are not, and have no desire to be, software

engineers. There is some overlap between adjacent groups, but typically numerical analysts are not also software developers (and vice versa).

Areas in which computational studies have a long history (for example molecular dynamics, lattice QCD and ab initio quantum chemistry) tend to have well-developed and supported codes. For example, Chroma (Edwards and Joó, 2005), DL_POLY (Smith and Todorov, 2006), CASTEP (Clark et al., 2005) and NWChem (Kendall et al., 2000). The software stacks are large enough that, often aided by funding through government labs, a serious amount of software engineering has gone on.

In contrast, areas where software is used as a more ad hoc tool, or where the global development community is small, codes grow rather more organically. These codes are typically much less modular, doing one thing in one particular way and not having the option of swapping out for different components. These have included codes such as Gadget (used for cosmological simulations), much of the software used by the UK Turbulence Consortium⁴, and the Hadley Centre Unified Model, used for studies of climate change. Having said that, if there is a single part of the code that dominates the run time, as in gravitational force calculations, it is often possible to just replace the single function that does the work. For example, by executing an equivalent function on a GPU or application-specific hardware such as GRAPE (Makino et al., 2003).

The implementation of new algorithms or optimisation techniques is typically quite difficult in these, less modular codes. The monolithic nature means that a change to a data structure, for example, requires global code modifications rather than local ones. The development community is also often smaller: diving in and changing things requires more initial effort. It may simply be that computational science, and software engineering practices, are not as well-developed in these areas. But whatever the reason, the codes *are* harder to maintain and extend.

On the algorithm-development side, prototyping and proof-of-concept codes are produced almost exclusively in Matlab. Ease of use and speed of development were the main reasons given. After all, you don't want to spend a long time developing something in C or Fortran, only to find it doesn't work as expected: especially when the equivalent algorithm can be coded up in an afternoon in Matlab. Two interviewees who produced such proof

⁴Although in this case, it is partly due to the simplicity of the code. Their codes are typically quite small: they consist of a short solver wrapped with some initialisation and IO routines.

of concept codes did so in Fortran and C++. Matlab not being sufficiently performant for their work, they did not want to waste time and effort on an implementation that would subsequently just be thrown away.

Computational codes are, however, not written in Matlab. Those we have spoken to have used either C, C++ or Fortran (typically f90). The distribution of languages in our interviews matched the results of Simpson et al. (2008), with about 50% of codes written in Fortran and 25% each C and C++. The sample size is, however, quite small.

Since algorithm proof-of-concept codes are typically developed in stand-alone codes, rather than in existing software packages, “responsibility” for introduction of a new algorithm into HPC software typically lies with the software developer, rather than the algorithm developer.

2.1 Hiding parallelism

The simplest mathematical formulation of an algorithm usually makes no explicit reference to any parallel computation. The need for parallelism is entirely an issue of implementation. For any given problem, the most natural way to write the algorithm is using the implicitly parallel notation of the mathematics. As a result, large codes are often designed to express the necessary parallel constructs in terms of higher-level domain-specific calls that mirror the mathematical notation. For example, users of the lattice QCD software Chroma (Edwards and Joó, 2005) never have to worry about how to get data from neighbouring lattice sites (even if that data is on a different node)⁵. Instead they write the code as a direct translation of the mathematical structure of their problem: in terms of shifts of the field. The back-end software takes care of translating this into the necessary optimised communications under the hood. This removes the burden of writing good parallel code from the end-users who just want to do science: they can just naturally express their algorithm in code. Another example comes from the programming model used in NWChem. This was developed concurrently with a toolkit for “faking” shared memory across distributed memory machines (the Global Arrays toolkit). Writes and reads to “foreign” array references can be dealt with very naturally and the programmer does not have to worry about such things.

⁵One interviewee, who had recently worked on some code using Chroma, remarked that it was the only time they’d written parallel code and never had to worry about parallelism at all.

This approach has a number of advantages over the “everything in the open” approach. Firstly, end-users of the code do not have to be expert HPC programmers to produce applications. Secondly, such an approach, if implemented correctly, vastly eases the maintenance burden. By enforcing programming to a well-designed interface, rather than having every layer of software interleaved with every other, it is much easier to make changes: if you want to change the data layout, you *know* you only have to change it in one place, rather than changing the layout somewhere, and then searching through the code to find every reference to it. For example, this approach, adopted in codes like Chroma and CASTEP, allows swapping out different communication implementations depending on which is most efficient on a particular platform, without having to make wide-scale changes.

Producing software in such a manner is no small task⁶. It requires more than just sitting down and writing code. The development team has to think about the data structures and interfaces, and has to have enough prior experience of the application area to make the “right” choices: changing the API once a version has been released is likely to upset users. Even if all this is done, it is still in general impossible to anticipate every possible generality that people will need. Indeed, too much generality was raised as a bad thing by a number of interviewees: if your library is *too* general, no-one will use it because it’s too complicated. A balance must be struck.

A further hurdle to overcome is that, typically, such efforts will not (and indeed, in the experience of interviewees, cannot) be carried out in universities. Software development work of this nature takes a significant amount of time. In universities, scientists’ time is typically split between teaching and research. There is not enough time for them to organise large-scale software development as well without eating into time from other areas. As a result, most of the large codes covered by this survey have had development driven by research labs.

Since much scientific software is initially developed by researchers at universities needing it to produce results for a paper, most codes are not highly modular nor easy to maintain. They have not yet reached the tipping point that will motivate rewriting and redesigning from the ground up. Or else, they work well enough and further software development takes a back seat to new science. For example, a few interviewees admitted that although their codes needed serious refactoring, there just wasn’t the time because new features come first. A number of interviewees bemoaned this state of affairs to

⁶When the redesign of CASTEP was started in 1999, the specification of the new code ran to around 400 pages and no new code was written until mid 2000

some degree: there is no incentive from the funding councils to put scientific results on hold for a time while major software re-engineering takes place. We should note that rewriting from scratch is not a silver bullet. At least one interviewee said the code they worked on was quite messy, but it had many years of debugging and development effort invested in it. Rewriting would effectively throw away all the debugging effort. Crucially, writing a code involves not just ensuring that it compiles and runs but also verification of results. This verification process takes a large amount of time and when rewriting, you have to start from scratch.

2.2 Development of new algorithms

In our survey we have found a number of cases in which interviewees have been involved in algorithm research “on the side”, for example the forward flux sampling method of Allen et al. (2005) and the acoustic modelling of Stefan Bilbao. That is, they have been studying in some particular area and found that no existing techniques work well enough. In effect, algorithm development occurs out of the need to address a new problem, rather than in isolation.

In most cases, we have found that a “successful” algorithm development involves implementation and use of the algorithm in further science. That is, because the algorithm is being designed to address a particular problem, a successfully developed algorithm will solve the initial scientific problem.

The numerical analysts we have interviewed have also typically been involved in development of algorithms as a result of some external collaboration. The result is either work on development of an algorithm to address a particular problem of the collaborator, or more general developments in the field that arise as a result of discussions. Typically, the implementation of these algorithms is more proof of concept than final. An implementation is provided in a high-level language (often Matlab) to demonstrate the algorithmic improvements. Some of the algorithm developers we spoke to were already implementing their ideas on parallel machines, in most cases only shared-memory architectures. Those that were not were actively moving from serial codes to parallel ones, and there seemed to be an increasing emphasis on distributed memory algorithms.

A downside of the development of new algorithms in isolation from HPC software packages is that new algorithms can take a long time to make their way into these packages. Developers have a finite amount of time to devote

to introducing new features and so a new method that shaves a bit of time off the current approach will likely fall by the wayside. If the algorithm developer wishes to get their idea into big codes, they either need to collaborate closely with the developers, or implement the new idea in the target package themselves. We should also note that in many codes the algorithm is intricately linked with the data layout and implementation. Even if the code is reasonably modular, often the constraints of data layouts mean one cannot ask “what’s the best algorithm for my problem?”, but rather, “what can we do given the current code structure?”. We mention this in relation to library usage as well (see section 2.4.1). Many codes have been designed with BLAS and LAPACK in mind, and so use them. Areas like sparse linear algebra do not have libraries that are so well established, often the library has appeared after the code has been designed. At this point, it is often too difficult to use the library, because its data layout doesn’t match that of the existing code. A few interviewees said this was a big issue. They would like libraries that are essentially data-layout agnostic so that the application can specify the best layout for the problem, rather than needing to conform to the library’s need.

2.2.1 Parallelism in the time domain

One area in which work is needed is parallelisation in the time domain. Although parallelisation across space is comparatively straightforward, codes still move in lock-step in time. To take an example, to study the probability of different conformations of a protein by molecular dynamics, the naive approach is to start in one conformation and simulate the molecule for a long time. Assuming we’ve sampled the steady state for long enough, the probability of a particular conformation is then just the time spent in the configuration divided by the total time. If we’re uninterested in the dynamical pathway between configurations, such a simulation method is very time consuming, especially if the free energy barrier between the states is large. Ideally, we’d just like to sample the transitions with the correct probability. Some techniques exist to help parallelise the “time” aspect of these processes such as parallel tempering (Earl and Deem, 2005), umbrella sampling (Torrie and Valleau, 1977) or forward flux sampling (Allen et al., 2009) in various Monte Carlo-like simulations, however, extension to other fields is in its infancy.

2.2.2 Exploiting multicore

Naively, using modern multicore chips is trivial for an existing parallel program. You just run a separate process on each core. As the amount of memory per core shrinks (see section 2.5), this approach may no longer be feasible. In fact, some interviewees mentioned that they are already having to think about approaches that view each multicore processor as a single shared memory machine which can communicate with others via MPI⁷. Sometimes just to get good performance, in other cases so that their code could run at all.

Efficiently targeting just a single CPU on today's processors is hard, in the experience of our interviewees many compilers do not do a good enough job of exploiting instruction-level parallelism, such as the short-vector SIMD instructions that can process multiple floating point numbers simultaneously. Many compilers also do not take full advantage of the large number of independent registers in modern CPUs, instead reusing the same few. This leads to significantly suboptimal performance. Things get even worse when trying to target the shared-memory architecture of multicore processors. It is very difficult not to introduce false sharing of data⁸ across different CPU caches, and in general to extract the most performance out of the chips.

Language extensions to effectively deal with the hierarchy of both memory and computational resources are currently lacking. Compiler pre-fetching to load caches and pipelines often does not work well. Targeting multicore chips is either through OpenMP⁹ (if the compute-intensive parts of the code are simple loops) or pthreads (for more complex interactions). With sufficient source code hints, compilers should be able to do the dependency analysis that would allow automatic exploitation of multicore CPUs. Further, one could imagine a variety of extra hints to give compilers knowledge of data structures that would allow better exploitation of SIMD instruction sets.

As well as multiple cores on a chip, a number of interviewees mentioned that they were in the early stages of investigating the use of accelerator cards, particularly GPUs. These provide very cheap performance boosts for codes

⁷That is, they have to treat intra-machine (core-to-core) communication differently from inter-machine communication

⁸A symptom of the coherent nature of CPU caches: memory systems consider writes to cache lines to invalidate the data across all caches. But cache lines are typically wider than individual data elements, so writing to one part of a cache line will invalidate all of it, rather than just the necessary part

⁹<http://www.openmp.org>

that have a section that needs lots of flops without too many memory accesses. A few people were worried about committing too early to a particular programming model. For example, CUDA, which is used to program Nvidia GPUs, is a proprietary language and runtime environment. They would prefer a standard way of exploiting these cards, within their existing language framework. Their worry is that by targeting CUDA specifically, they will potentially end up spending effort on a platform that has no support in 5–10 years time.

We note also that GPUs do not provide a universal solution: codes that have a small computation to communication ratio, or ones that are memory-bandwidth-bound will not benefit nearly as much as codes that essentially just need to do lots of floating point operations.

2.3 Scaling behaviour of codes

Amdahl’s law puts an upper limit on the amount of speedup one can obtain from a parallel program for a fixed problem size. Once this limit is reached, one can go no further for a fixed problem size. Instead, as the machines we run software on get bigger, we study larger problems. If the algorithmic complexity of a problem of size N is $O(N)$, then in theory we can look at a problem twice as large in the same time if we just run on twice as many processors. In practice, this latter form of parallelisation falls down at some point. Specifically, codes will only continue with this scaling behaviour if the communication is local rather than global. For large numbers of processors, we do not just have to worry about the algorithmic complexity of a problem due to its size, but also the extra run-time that execution on multiple processors entails. For example, at large core counts, the execution time of fast Fourier transforms is limited by the all-to-all communication required. As an example, the CASTEP code has an algorithmic complexity of $O(N^3)$ and has to perform 3-D FFTs with complexity $O(N^2 \log N)$. On large core counts, the performance is dominated by the communication overhead of the FFT, even though the complexity of this part of the code is lower than for other parts.

We have found, perhaps unsurprisingly, that we can divide the scaling behaviour of applications that people use into two groups. Those that only require local communication – for example lattice Boltzman methods, Quantum Monte Carlo, short-range N-body problems, QCD – scale linearly, and can happily use larger and larger machines to tackle bigger problems. Problems that require frequent global communication, for example gravitational

N-body problems or quantum chemistry codes, scale for a while. Eventually, however, performance plateaus and the addition of further processors no longer provides a speedup. In many cases, further processors cause a slowing down of the time to solution. In the experience of our interviewees, these latter codes will scale well to around 1000 processors, but after that no further gains are possible.

To increase performance, one turns to better algorithms. Reduction of algorithmic complexity certainly helps. If the complexity can be reduced from $O(N^2)$ to $O(N)$, say, then increasing the system size by an order of magnitude only requires ten times as many processors, rather than one hundred times as many. For communication-limited codes, reduction in global communication is equally important. You might be willing to take a hit in algorithmic complexity if it means you can scale to huge processor counts. A number of interviewees mentioned problems such as this, although in relation to memory bandwidth. To deal with the lack of memory bandwidth required them to rethink their algorithms globally rather than just performing local optimisations.

2.4 Writing parallel codes

MPI is very much the *lingua franca* for distributed memory applications. The typical reason given for this choice is portability. Researchers do not want to be tied to a specific platform-library combination, and hence choose the message-passing option that is most widely supported.

As discussed in section 2.1, mathematical formulations of problems are typically not explicitly parallel. Most interviewees would like to be able to write implicitly parallel codes, but are sticking with explicit message passing for portability and also because it's the only way to really control communication patterns. If the programmer doesn't know when a barrier or all-to-all communication is going to occur, they cannot "hide" the latency by doing other things. If all communication is implicit (as was the case in High Performance Fortran), it is almost impossible for the programmer to write highly-performant code, this was cited as one of the main reasons for the failure of HPF. In such cases, the programmer is unable to predict when latency will occur and has no way of hiding it when it does. Compilers can sometimes do a good job of this, but in general performance is only sufficient if the programmer has good control over communication.

2.4.1 Usage of libraries

Again for maintainability reasons, we have found use of libraries limited to a few highly portable choices: typically BLAS, LAPACK, ScaLAPACK and FFTw (although only in its serial version). There does not currently appear to be a widely-used parallel library for sparse linear algebra, although the PETSc library used by one interviewee may be filling this gap.

One comment on libraries that appeared a number of times was the problem of mismatches between the library APIs and application code. The APIs of libraries like BLAS assume access to elements is through array indexing. That is, the elements of a vector or matrix are stored as a chunk of memory in an array and BLAS routines expect to work with a pointer to this array. In some applications this is no longer the case. The data would take too much memory to store explicitly and so elements are computed on the fly: there's no pointer to pass in to the BLAS routine. Another consideration that may lead to this choice is that memory accesses are very expensive when compared to cycles. A code may choose to spend cycles computing elements, rather than pulling them out of memory. This kind of issue will likely become increasingly important over the next five to ten years due to the continued decrease in both memory capacity and memory bandwidth per core. Rather than optimising for reduced cycle counts at the expense of more memory accesses, optimisation will have to go in the other direction. This is expanded on in section 2.5.

2.5 Future directions

The IESP's stance is that in future, massively parallel machines will be even more memory starved than currently. Due to (primarily) power constraints, an exascale system will have almost no memory in comparison to current systems. Chips will probably be restricted to having memory the size of current caches, and no more. This is the only way it is currently thought that one can build exaflop machines that are not *too* power hungry. This is likely to require a significant change in programming style. In fact, some problems may not be able to take advantage of exascale systems for this reason: the specification of the problem may be too big to fit in memory. The CASTEP and CFD code developers we interviewed are worried about this issue now. Their problems are sufficiently memory intensive that they sometimes won't fit in the memory of multicore chips unless sharing data across cores.

Those interviewees whose codes currently scale to hundreds of thousands of processors are having to rethink their strategies for dealing with node failure. Most codes are currently written assuming a “long” time between node failures and deal with these by check-pointing to disk infrequently. If node failure occurs, the program exits and is restarted picking up from the most recent checkpoint. As simulations get bigger (and processor counts higher), check-pointing to disk becomes more costly. In addition, with more processors the mean time to failure decreases. Current checkpoint-restart methods will need rethinking for exascale machines.

For some applications, node failure can be dealt with on the fly. If the problem is essentially a task-farm approach – for example lots of independent Monte Carlo sampling events – the code could be written to hand out parcels of work and gather them in. This kind of strategy is being considered for the QMC code CASINO (Needs et al., 2010) to allow for better load balancing. If a parcel does not return, then one assumes that the node calculating it has failed and re-submits the task to another node. For most problems, however, node failure cannot be dealt with so easily. Software that relies on a tight coupling between the physical network layout and software communication patterns to achieve good load balance and scalability cannot just redistribute work if a node fails. For example, in a recent scaling workshop on Jugene¹⁰ one of the interviewees noticed performance drop-off at very high core counts because his assumptions of which way messages would be sent through the system were incorrect.

In addition to all-out hardware failure are more subtle issues. How should programs deal with issues like bit errors in memory? For certain classes of problem, bit errors are not an issue. For example, the iterative solvers used in many codes might just take another few iterations to converge. In other applications, we have to ask how we might spot such errors, if indeed they need spotting. An interviewee working in gravitational models said he had no way of really knowing if the model produced the “right” answer unless the error was noticeably unphysical.

The question of reproducibility is also a tricky one. Is it enough to get results that are “about the same”, or should one strive for bit-identical results from different simulations of the same set of initial conditions. Climate modellers are currently worrying about this. Is their current wish for bit-identical results possible, or even necessary, on future machines? In some ways this is a slightly philosophical issue. For many models, it is difficult to specify if an answer is “right”, only that its observables match experimental data.

¹⁰Jugene is the BlueGene/P at Jülich and currently has 294 912 cores

2.5.1 Increasing absolute performance

Realistically, most scientific computations do not display the necessary traits for scaling to even petaflop machines – only a small number of interviewees had codes that exhibited full linear scaling, the rest were limited by global communications. The only such problems that do scale are ones where communication is highly infrequent (or only local). Any use of all-to-all communication patterns cuts off scaling at some (small) finite number of processes (typically hundreds or thousands, depending on problem size). Given this, it seems prudent to focus not just on developing algorithms to allow good scaling – certain problems *need* communication which destroys scaling at some point – but also to look at the absolute performance further down the chain.

Assume that we have a well-known problem for which the algorithmic complexity is $O(N^3)$. Multiple different approaches may yield the same complexity, so for best performance we're interested in the one that provides the smallest constant factor. Once the choice of algorithm has been made, we need to implement it in the most efficient way possible. This might involve writing the code to avoid cache misses, to access data in the correct order, and so on. Anything that makes this level of optimisation easier is a win. That is, it would be nice not to have to write assembly code by hand for best performance. Currently, it is only developers of lattice QCD codes who are concerned enough about performance to do something in this area. See, for example, the assembly generation library BAGEL (Boyle, 2005). The majority of interviewees did not worry about this level of optimisation, predominantly for portability reasons. Their codes have to run on a wide number of different systems and they do not have the time to support hand-tuned sections of code for all of them. Instead, if their codes require high-performance linear algebra (say) they rely on vendor libraries to do a good job. When the manipulations are not those that have been optimised by the vendor, codes can run slowly.

This might seem a very specialised problem, in fact, it is (or will become) a general issue. With the trend to more and more cores on a chip, memory-bandwidth issues really start to affect performance. For applications that are memory-bandwidth dominated, for example finite difference CFD codes, multicore architectures are already causing performance problems. One interviewee remarked that in the recent HECToR upgrade from 2 to 4 cores per chip, performance of their software dropped by a factor of 1.7 per core purely due to the memory bandwidth reduction. They also found that compilers did not have a sufficiently detailed understanding of cache and data layout,

in one case, reordering loops to pessimise memory access patterns. To combat this, software developers need to have much finer control over memory accesses. Language extensions that can target the hierarchical architecture (multiple cores sharing one memory bus, various levels of cache, and so on) are necessary to enable this.

Another avenue that is receiving exploration is the use of software that allows specification of a problem in a high-level language (mirroring the mathematics), these are often termed domain-specific languages. At this level, knowledge of the mathematical structure has not been thrown away, and the software can do symbolic manipulations to produce the optimal code structure (for example, elimination of unnecessary temporaries or performance of loop fusion). Once this is done, the software then writes the necessary code in the target language suitable for compilation. The ab initio chemistry code NWChem uses such a system, the Tensor Contraction Engine (Baumgartner et al., 2005), since the expressions it deals with are too complicated to simplify by hand. This area of research is still in its infancy, a general compiler along these lines would likely be very popular, especially if integrated into something with a large existing support community, like GCC¹¹.

2.5.2 Load balancing

Achieving good load balance – even distribution of work between processors – is crucial for good parallel performance. If the work distribution is uneven, most processors will waste cycles idling while a few do too much. For static, highly structured problems, load balancing can be achieved by the programmer at the design stage. For many problems, where the distribution of work changes as the simulation progresses, specifying how to redistribute the work may be too difficult. An algorithmic way of specifying how to distribute work would be very useful. One interviewee mentioned work in this area by Jack Dongarra on writing linear algebra code that models parallel aspects as loosely coupled tasks with some dependency graph.

At this point, we note an issue with potential load balancing libraries, although in our experience of interviewee comments the general point holds of all library interfaces. Designing an interface that meets the needs of users is not an easy task. The tight coupling between workload, data layout, and decomposition makes it difficult to find an abstraction that can be used in

¹¹Stand-alone tools would also work, although they would present an additional barrier to adoption. If such a tool existed in a compiler, it would be easier for people to use it.

the general case. For example, the DL_POLY developers have recently introduced dynamic load balancing, but they ended up having to roll their own algorithm and implementation since existing software (and algorithms) didn't meet their needs. Perhaps this tells us that load balancing is fundamentally application-specific, or perhaps just that there is lots of room for research.

In addition to dynamic work distribution, further problems arise when load balancing. Some algorithms don't lend themselves to perfect balance. For example, multigrid methods that successively coarsen the system can lead to a situation where there is not enough work to go round. Further problems can arise if the physical distribution does not match the communication topology. For example, many N-body problems calculate short-distance forces exactly, but use approximate methods for particles that are far away. This approach works if the distribution of particles is reasonably homogeneous. If clustering occurs, to get a good distribution of work, physically close particles must be partitioned out onto network-distant processors. The work is evenly distributed, but now calculation of short range forces requires global communication. This problem is what limits the scaling behaviour of cosmological simulations.

2.5.3 Data analysis

A worry expressed by a significant fraction of interviewees is how to deal with the data they obtain from their simulations. Modern systems generate so much data that tried and tested methods of analysis and visualisation no longer really work. With a multi-terabyte or multi-petabyte dataset, spotting patterns by hand is not really feasible. Furthermore, such large datasets cannot even be analysed programmatically on desktop machines: it takes too long. Some serious development work needs to happen in parallelising not just simulation software, but also the analysis and visualisation tools. In addition, it would be useful to couple these aspects with things like automated ways of identifying "interesting" events in the data. For example, the LHC at CERN produces so much data during collisions that scientists have no possibility of saving and analysing it all. Instead, hardware triggers are used to start recording data when interesting events are observed. It is likely that similar approaches will be needed for simulation data.

3 Summary

Our survey has thrown up a number of common issues that will be increasingly important in the next few years.

1. There is a need for languages and compilers to better target the hierarchical structure of machine architecture.
2. There is a need to deal with increased memory starvation of cores.
3. There is a need for fault-tolerant approaches that scale to large numbers of processors.
4. There is a need to find ways to deal with load balancing in a semi-automatic manner.
5. There is a need for new techniques to analyse the volumes of data that modern codes can generate.
6. Development of library interfaces must be considered very carefully. One should not underestimate the amount of work involved in adapting existing codes for new algorithms, especially if data layouts need changing.

Most interviewees were concerned with how modern software should deal with the increasing number of shared-memory cores that are prevalent in modern systems. Producing good code to run on the increasingly complicated memory hierarchies is hard. It's harder still to do so in languages that provide very little in the way of support for the target architecture. In the few cases where interviewees have been concerned about absolute performance in their code, they still write assembly code. A state of affairs they bemoaned universally.

This ties in with the second point. Neither C nor Fortran compilers have the high-level understanding of code to make the simplifications that a programmer can make. They can only really optimise at the micro- rather than macro-scale. There is some promising work being done in this area for BLAS-like operations (see for example Baumgartner et al. (2005); Belter et al. (2009)), but it is not yet in the mainstream. Those interviewees that worried about this sort of thing remarked that they would only adopt a new approach if it produced *better* code than they have already. Ideally, compiler extensions like this would be provided as an extension to GCC, rather than as a completely separate research project, to leverage the existing infrastructure and large support community. We mention GCC since it is the only portable optimising compiler that is open source and thus amenable to modification by research groups. Should such features appear in commercial compilers, they would also likely be used.

The way in which people approach optimisation also needs to change. Rather than attempting to minimise instruction counts, we should be looking at ways of minimising memory accesses, even if this means spending more cycles. This will be especially important as the amount of memory available to cores drops.

The comments by interviewees about library use should be considered carefully. Development of general purpose libraries requires careful thought and collaboration with application developers. In many cases, we have heard that implementation of new techniques has not been simply a case of “what is the best algorithm/library for my problem?” but rather “what is the best option given the data layout and other constraints of the existing code?”. As such, most codes only use library interfaces that are older than the code itself. Newer, perhaps better, libraries may be available, but do not fit well with the structure of the code.

The other issues are a little less well-defined, but definitely bear thinking about. For heterogeneous, anisotropic problems, predicting load balance and communication patterns before the fact is very difficult. We should instead be looking at ways of dynamically dealing with the problem, taking the load off the end-user if possible. Similarly with fault tolerance. There is likely no universally applicable solution. However, anything that makes programming around the problem easier is going to be looked on favourably. Data analysis is likely outside of the scope of NAIS, but the challenges faced in this area are certainly worth bearing in mind.

A Interview proforma

A.1 Implementation issues

- What programming languages do you use and why?
- What are the challenges porting an existing algorithm to new architecture?
- How do you find out about new algorithms?
- What challenges do you typically face when implementing a new algorithm?
- Do you find yourself reusing similar patterns of parallelism in different codes?
- Are you aware of (or a user of) higher-level abstractions / libraries?
- How much use do you make of serial and parallel libraries?

- What is the major limiting factor to progress in your research area?
- How important is architecture-specific tuning?

A.2 Development issues

- Is parallelism potentially important?
- Do you develop algorithms with parallelism in mind?
- What would you consider a successful outcome — theory, implementation in Matlab, integration into a real application?
- What languages do you use and why?
- What motivates you to look at algorithmic development?
- How important is architecture-specific tuning?

B List of interviewees

Mark Ainsworth	Dario Alfè	Rosalind Allen
Stefan Bilbao	Peter Boyle	Ian Bush
Murray Cole	Dugald Duncan	Iain Duff
Tom Edwards	Len Freeman	Herbert Früchtl
Douglas Heggie	Chris Johnson	Roderick Johnstone
Bálint Joó	Tony Kennedy	Richard Kenway
Ben Leimkuhler	Alison Ramage	Bill Smith
Volker Springel	Kevin Stratford	Simon Tett
Arthur Trew	Paul Tulip	Andrew Turner

References

- R. J. Allen, P. B. Warren, and P. R. ten Wolde. Sampling rare switching events in biochemical networks. *Phys. Rev. Lett.*, 94:018104, 2005.
- R. J. Allen, C. Valeriani, and P. R. ten Wolde. Forward flux sampling for rare event simulations. *J. Phys.: Cond. Mat.*, 21:463102, 2009.
- K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006. URL <http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html>.

- G. Baumgartner, A. Auer, D. Bernholdt, A. Bibireata, V. Choppella, D. Ciorva, X. Gao, R. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C. Lam, Q. Lu, M. Nooijen, R. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. *Proceedings of the IEEE*, 93(2): 276–292, 2005.
- G. Belter, E. R. Jessup, I. Karlin, and J. G. Siek. Automating the generation of composed linear algebra kernels. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 2009.
- P. A. Boyle. The BAGEL assembler generation library. 2005. URL <http://www2.ph.ed.ac.uk/~paboyle/bagel/>.
- S. J. Clark, M. D. Segall, C. J. Pickard, P. J. Hasnip, M. I. J. Probert, K. Refson, and M. C. Payne. First principles methods using CASTEP. *Zeitschrift für Kristallographie*, 220:567–570, 2005.
- J. Dongarra, P. Beckman, T. Moore, J.-C. Andre, J.-Y. Berthou, T. Boku, F. Cappello, B. Chapman, X. Chi, A. Choudhary, S. Dosanjh, A. Geist, B. Gropp, R. Harrison, M. Hereld, M. Heroux, A. Hoisie, K. Hotta, Y. Ishikawa, F. Johnson, S. Kale, R. Kenway, B. Kramer, J. Labarta, B. Lucas, B. Maccabe, S. Matsuoka, P. Messina, B. Mohr, M. Mueller, W. Nagel, H. Nakashima, M. E. Papka, D. Reed, M. Sato, E. Seidel, J. Shalf, D. Skinner, T. Sterling, R. Stevens, W. Tang, J. Taylor, R. Thakur, A. Trefethen, M. Snir, A. van der Steen, F. Streitz, B. Sugar, S. Sumimoto, J. Vetter, R. Wisniewski, and K. Yelick. International exascale software project roadmap. Technical report, IESP, 2009.
- D. J. Earl and M. W. Deem. Parallel tempering: Theory, applications, and new perspectives. *Physical Chemistry Chemical Physics*, 7(23):3910–3916, 2005.
- R. G. Edwards and B. Joó. The Chroma software system for lattice QCD. *Nuclear Physics B – Proceedings Supplement*, 140:832–834, 2005.
- R. A. Kendall, E. Apra, D. E. Bernholdt, E. J. Bylaska, M. Dupuis, G. I. Fann, R. J. Harrison, J. Ju, J. A. Nichols, J. Nieplocha, T. P. Straatsma, T. L. Windus, and A. T. Wong. High performance computational chemistry: An overview of nwchem a distributed parallel application. *Computer Physics Communications*, 128:260–283, 2000.
- P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keck-

- ler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick. Exascale computing study: Technology challenges in achieving exascale systems. Technical report, DARPA IPTO, 2008.
- J. Makino, T. Fukushige, M. Koga, and K. Namura. Grape-6: The massively-parallel special-purpose computer for astrophysical particle simulation. *Publications of the Astronomical Society of Japan*, 55(6):1163–1187, 2003.
- R. J. Needs, M. D. Towler, N. D. Drummond, and P. López Ríos. Continuum variational and diffusion quantum Monte Carlo calculations. *Journal of Physics: Condensed Matter*, 22:023201, 2010.
- A. D. Simpson, M. Bull, and J. Hill. Identification and categorisation of applications and initial benchmarks suite. Technical Report D6.1, PRACE, 2008.
- W. Smith and I. T. Todorov. A short description of DL-POLY. *Molecular Simulation*, 32:935–943, 2006.
- G. M. Torrie and J. P. Valleau. Nonphysical sampling distributions in Monte Carlo free-energy estimation: Umbrella sampling. *Journal of Computational Physics*, 23(2):187–199, 1977.
- A. E. Trefethen, N. J. Higham, I. S. Duff, and P. V. Coveney. Developing a high performance computing/numerical analysis roadmap. Technical report, OeRC, 2009.