Composing HPC Packages and the Influence of Dominant Performance Characteristics

Matt Martineau and Simon McIntosh-Smith

Merchant Venturers Building, Woodland Road, Bristol, BS81UB, United Kingdom
{m.martineau,cssnmis}@bristol.ac.uk

Abstract. In this paper we discuss the composition of scientific packages, and how this affects performance on highly parallel supercomputing hardware. We introduce three new mini-apps: Hot, Flow, and Fast, which solve heat diffusion and hydrodynamics, and perform parallel FFTs to solve Poissons equation. The mini-apps represent three of the seven Dwarfs of parallel programming, and have been developed with the purpose of evaluating the impact of coupling those algorithms together. Although much research has been performed that evaluates the performance profile of the seven Dwarfs, production applications rarely utilise only a single package for solving a scientific problem. We start by presenting a motivational discussion of the relevant Dwarfs, considering their invidual performance characteristics and which of those characteristics are dominant. Following this we present performance results of coupling Hot, Flow, and Fast together, using them to solve test problems on modern hardware.
The paper ends with a discussion of some of the most important performance issues that we can forsee when coupling applications together, motivating a stream of future work in the area.

Keywords: performance portability,mini-apps,high performance computing,openmp 4

1 Introduction

A great amount of work is being devoted to understanding how scientific production applications will be ported to run on modern supercomputers. Some context of the challenges currently faced by the community is presented to demonstrate the importance of this research.

1.1 The Use of Mini-Apps to Optimise for Modern Architectures

To meet power demands and continue to grow the performance capabilities of supercomputing resources, a continual diversification of architectures has been essential. This has brought significant complexities for scientific applications developers, necessitating improvements in programming environments and the

optimisation of algorithms. Due to the extent of the task, it is now essential that computer scientists are involved in the process, which has lead to a surge of co-design projects. In particular, many optimisation projects are being performed with proxy applications, or mini-apps, such that the lessons can be prototyped in collaboration between scientists and computer scientists.

A great many studies have been performed that have used mini-apps to investigate key problems such as application performance, portability, scalability, and fault tolerance. In spite of this, the research does not generally make a link between the optimisations seen for those miniaturised proxy applications and full scale production applications.

1.2 Extending the Single Mini-App Focus

This paper is a window into a set of work that is attempting to extend the success of mini-apps for evaluating modern architectures and programming environments. It is essential that the community is mindful of the purpose of mini-apps, which is to discover optimisations that have real world impact. Our aim is to begin to capture the features of production scientific codes that have been yet unexplored with mini-apps, to improve the validity of future performance investigations.

Much of the mini-app optimisation focus has been limited to individual applications which have, in general, performed well on modern hardware, using a range modern parallel programming models. Up until this point the individual mini-apps have failed to expose some important aspects of large-scale scientific applications, in that they are isolated instances of algorithms that do not encapsulate the multi-package hierarchies that scientists are truly interested in seeing results for.

In order to limit our focus to the most important algorithms that might be encountered within scientific production codes, we will follow the seven Dwarfs of parallel programming. In the 2006 View from Berkely paper [**?**], a number of important classes of parallel algorithm were discussed and categoried as parallel programming Dwarfs. Each of the Dwarfs exposes a diverse computational and communicational pattern and can encompass many of the algorithms you are likely to see in production scientific applications.

Their paper eluded to the necessity for those Dwarfs to be composed together and investigated as coupled application. Our intention is to use a new suite of mini-apps, tailored for composition, to expose the dominant performance characteristics when combining multiple packages, as would be seen in a common scientific production application. Each mini-app represents one of a subset of the Dwarfs, and we plan to consider all of the Dwarfs in future research.

We recognise that many of the issues we discuss or uncover will have been discovered by application developers or performance engineers attempting to combine multiple packages together in the past. However, it can often be challenging to generalise approaches from specific use cases, which is one of the benefits of mini-apps, where a general strategy can be employed and discussed.

Ultimately, it is essential to the relevance of results obtained by mini-apps that such features are considered alongside other performance studies.

2 Contributions

This research is primarily focussed upon discussing the implication of composing a subset of the parallel application Dwarfs, and we present a number of important contributions in order to support the discussion:

- We have developed three mini-apps that serve as proxy applications for important scientific algorithms. An FFT solver for Poisson equations (Fast), a heat conduction application (Hot), and a fluid dynamics application (Flow).
- We present results for those algorithms in isolation on some modern supercomputing architecture: an NVIDIA K40m, an Intel Xeon Phi Knights Landing (KNL) processor, an IBM POWER8 CPU, and a Intel Xeon Broadwell.
- We naively couple the Hot, Flow, and Fast mini-apps together to demonstrate their composed performance profiles.
- We offer insights into the issues and opportunities that might be experienced when composing algorithms that goes beyond the scope of our current experiments.

3 Background

In this section we will discuss the general composition of packages, and then present some basic details about each application, including a description of the numerical method and parallelisation strategy. To ensure a clear discussion, we will explain the initial development of each application prior to discussing our efforts to couple them together. This means that we will explain how each application was optimally developed, but with the caveat that this would likely have to change once coupling was required. We did not actively consider the coupling during the up-front development of each application, instead opting to choose the best strategy for each application in isolation.

4 Composition of Dwarfs

It would be possible to compose packages from any of the domains within science and engineering, but an aim of this research was to use canonical algorithms that can act as proxies for wider classes of application. In this paper we have chosen to continue on from the work discussed by Asanovic et al. [**?**], who outlined a number of Dwarfs, each of which describes a different computational class that features in modern supercomputing applications. We hope that considering applications from each of the Dwarfs will offer general insights that can be applied to the wider field.

Each of the Dwarfs has a unique set of computational and communicational characteristics that mean that they expose different requirements of modern computing hardware. While many of those characteristics have been analysed and optimised heavily on existing architectures, we intend to uncover the extent to which those classes can co-exist within an application. At this stage we will re-iterate that that the majority of production applications that solve significant scientific problems will require packages encompassing a number of the classes described by the Dwarfs.

The Dwarfs cover broad families of algorithms, each of which encapsulates a subset of possible performance characteristics, with some overlap of characteristics amongst the Dwarfs. An aim of our research is to expose dominant performance characteristics, as we hypothesise that there will be an unequal weighting of the importance of those characteristics.

The Dwarfs that we consider in this paper are:

- **Spectral Methods** - *Fast*: A Fast Fourier Transform solver for Poisson's equation (Section 4.1).
- **Sparse Linear Algebra** - *Hot*: An heat diffusion application, that uses a Conjugate Gradient linear solver (Section 4.2).
- **Structured Grid** - *Flow*: A Lagrangian-Eulerian hydrodynamics application that uses an explicit 5 point stencil (Section 4.3).

We will present the results of coupling Hot 2d with Flow 2d, and Hot 2d with Fast 2d. To support this we also discuss the potential difficulties that we anticipate will arise when composing the other Dwarfs together.

It would be possible to end up with large spaghetti applications when composing multiple packages together, which would dilute the merits of using mini-apps. We have made every effort to ensure that the applications we develop for this purpose are simple while expressing the correct performance profiles, so that the agility of the individual mini-apps will still apply when composed.

4.1 Fast - Fast Fourier Transform for Poisson's Equation

The fast fourier transform (FFT) is an important algorithm to a number of scientific and engineering domains, and there exists extensive literature regarding optimisation of parallel FFTs. As part of this research we have developed a highly simplified FFT mini-app, which solves Poisson's equation. The primary reason for choosing this particular approach is not scientific relevance, rather that it will result in verifiable outputs, while still capturing some important performance characteristics of the class of algorithms.

The mini-app is flexible and can use a hand-rolled algorithm for the one dimensonal FFT and inverse FFT steps, but also provides an option to use the optimised Intel MKL library version. Our initial assumption is that the implementation of the FFT solve would be less important to the performance than the choice of decomposition strategy, given that the FFT operation is notoriously bound by the *'all to all'* communication of data among processes.

Prior to our experimentation, we anticipated that the inclusion of an FFT package within a multi-package application would likely result in conflicts over the optimal domain decompisition. We have seen similar instances of domain decompisition conflicts with decompositions for wavefront parallelism. We have developed the application with a number of different decompositions so that it is straighforward to test the impact of those options under composition with other packages.

Fundamental Method Following on from the discussions in Gholami et al., we utilise the FFT to solve Poisson's equation for some dependent data passed to the solver. The general solve phase is as follows:

$$f = -\Delta u \tag{1}$$
$$\mathrm{fft}(f) = f' \tag{2}$$
$$(f' \circ \Delta^{-1}) = f'' \tag{3}$$
$$\mathrm{fft}^{-1}(f'') = f''' \tag{4}$$

Step (3) is the Hadamard product of the transformed f and the inverse of the laplace operator. For our custom implementation of *fft* we use an out-of-place parallel method derived from the Cooley-Tukey algorithm that can handle arbitrary problem sizes. The arithmetic requirement of this package is quite low, especially when compared to Flow, which requires a cohesion between many numerical methods. Maintaining a simple algorithm ensures that it is straighforward to adapt to the many available decompositions and capture the dominant performance characteristics exhibited by algorithms that take advantage of the FFT.

Parallelisation The parallel decomposition of the FFT operation is architecture dependent, and in particular varies depending on how many computational elements the problem will be executed on. In particular, the different data layouts include decomposition by slabs or pencils.

This issue is particularly important when composing an FFT package with other packages, as different decompositions might negatively impact on the performance of the whole application leading to a larger search space of decisions when optimising an application.

4.2 Hot - Heat Conduction via a CG Solver

We have developed a simple conjugate gradient (CG) solver that implicitly solves the heat conduction equation in order to solve the problem within a reasonable time-frame with acceptable fidelity. Our implementation uses a standard preconditioner, but otherwise adopts the simplest CG approach.

Fundamental Method Heat conduction in two dimensions can be specified as follows:

$$\frac{\partial u}{\partial t} - \alpha(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}) = 0 \tag{5}$$

$$\alpha = \frac{k}{\rho c_p} \tag{6}$$

$$u_{i,j}^{n} = u_{i,j}^{n+1} \tag{7}$$

Where u is the temperature, α is the thermal diffusivity, k is the thermal conductivity, ρ is the density, and c_p is the specific heat capacity. As you can see from equation (7), this particular form of the equation requires an implicit solve. The conjugate gradient method is an iterative method that descends towards a solution using conjugate vectors. We will not develop or explain the mathematics underpinning the CG solver, and direct interested readers to explore the wealth of existing literature [] (PUT A LOAD OF CG REFERENCES IN). The only physical feature of the application is manifested in the density calculations, where the densities are stored at the cell centers and interpolated to the edges using the arithmetic mean.

Parallelisation The best decomposition for this mini-app is essentially regular cartesian, minimising the surface area to volume of each rank to reduce communication overheads. In terms of boundary conditions we selected reflective, as this was simple, and improves the verifiability of the results.

The algorithm does not expose any load imbalance, and nearest neighbour halo exchanges are performed each timestep. At each iteration an alpha and beta value is calculated, which needs to be distributed amongst all of the ranks. The implication is that each iteration requires two calls to MPI_Allreduce, an obvious performance bottleneck at scale.

As each of the core kernels is essentially a simple linear algebra method, such as a dot product or sparse matrix-vector multiply, it was straightforward to parallelise with OpenMP and CUDA, given the extent of the data-parallelism exposed by the algorithms.

4.3 Flow - Fluid Dynamics via Lagrangian-Eulerian Flux Calculations

The Flow mini-app is a Lagrangian-Eulerian hydrodynamics solver that is staggered in space and time. Although the mini-app is significantly larger and more complicated than our other new mini-apps, the resulting code is also significantly faster, as explicit stencil methods are particularly efficient in general.

Fundamental Method The Lagrangian-Eulerian algorithm takes Euler's equations and explicitly solves them on a structured grid. The application uses simple

smoothing to introduce artificial viscosity, with shock heating accounted for as part of the mechanical work update. For the mass flux calculations, a Van-Leer flux limiter is used to maintain a monotonic profile at shock boundaries, and slope-limited second-order interpolations are performed for energy and momentum to ensure monotonic behaviour for all dependent variables.

Directional splitting is used in order to make the application two-dimensional, with the leading dimension switched each timestep, in order to maintain symmetry of the solution. Our explicit timestep controls limit the timestep based on the CFL condition, stopping sound waves travelling further than a single cell per timestep, accounting for the additional spreading that occurs due to the artificial viscous stresses. We have made the algorithm second order in time by interpolating our velocities half a timestep upstream for the advection stages.

As with the heat condution application, the boundary conditions we chose for this application are reflective. This is particularly useful for fluid dynamics as tracking of conservation is important in verifying the results.

Parallelisation As with the heat conduction application, the algorithm does not have any inherent load imbalance, and only requires nearest neighbour communication. The explicit nature of the algorithm means that multiple kernels are invoked in order, and the independent kernels are inherently data parallel, allowing them to be easily threaded for CPU and GPU.

5 Performance Characteristics of the Mini-Apps

Although the Dwarfs imply some high-level performance characteristics, we believe it is important to consider all of the characteristics of the particular algorithms we have chosen. Some of the most important performance characteristics of the mini-apps are listed in Table 1.

Property	Hot Flow	Fast
Nearest Neighbour Communciation	Yes Yes	No
Iterative	Yes No	No
Mesh-based	Yes Yes	Yes
Stencil-based	Yes Yes	No
All to All Communications	Yes No	Yes
Memory-Bandwidth Bound	Yes Yes	No
Multiple Optimal Decompostions	No No	Yes
Load Imbalance	No No	No

Table 1. Performance characteristics of the four mini-apps.

We have a good understanding from previous work which of those characteristics matter to the individual application, such as *memory bandwidth* for indiviual

application performance and *all to all* communications for scalability. Our intention is to investigate how the computational and communicational profiles of those applications interact with eachother when packages are composed.

We can pre-emptively assess some of the likely effects that running multiple mini-apps at once might observe. For instance, we would expect that there could be an impact on cache utilisation, and a generally increased memory footprint, while some state could be shared potentially improving the overall performance. It is also possible that data structures between the individual applications will require some form of transformation, which we expect to have a major influence on performance. Aside from harming the performance overall, we are hopeful that there will be opportunities to exploit overlapping in some cases, and hide the cost of expensive operations.

6 Coupling

As the mini-apps and their composition routines were developed by computer scientists interested in the evaluation of performance profiles on modern architecture, the exact physical solution to the test problems is not important. However, the individual applications happen to be both correct and highly accurate, as choosing effective numerical methods tended to present the easiest option.

We could see two different approaches to coupling the applications, the first of which involves directly coupling their constituent equations and solving the whole system at once. This approach has two important drawbacks: (1) the complexity of this approach is significant, especially considering that we plan to extend this project into a suite of seven algorithms, and (2) boiling the algorithms under a single solve wouldn't capture the potential difficulties of executing diverse routines together.

In general we do not consider the coupling approach to be necessarily valid from a scientific standpoint. Our intention is simply to model a simple data dependency between the applications, that leads to them having to rely on shared meshes and decompositions. It is likely important future work to investigate the impact of performing more detailed couplings, but would require complex domain-specific knowledge far beyond the scope of our research (THIS IS ESSENTIALLY ARSE COVERING). A simple example of how we have handled coupling can be seen in Figure 1, where the initial conditions of density (ρ), energy (ϵ), and momentum (ρu and ρv) are passed in to start the cycle. Once the Flow phase is complete, the density and energy are passed to Hot, which then solves the timestep and passes back the updated temperature.

Composing Fast 2d and Hot 2d together was more challenging than with Hot 2d and Flow 2d, as the standard decompositions were different for each applications. For the sake of simplicity we constructed an artificial dependency between the packages, where a single dependent variable is passed through each of the solvers.

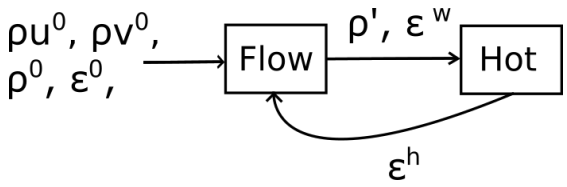

Fig. 1. The data dependency of our coupled Hot and Flow applications.

7 Performance Analysis of Compositions

Although we had some preconceptions regarding the likely effects of composing particular applications, we were objective in our assessment of the combined performance. In particular, we took time to validate that our assumptions were correct wherever possible, and have performed experimentation of a range of different platforms to observe whether there were any unexpected architectural influences.

7.1 Experimental Setup

Each of the mini-apps has been optimised using modern techniques from years of research optimising similar algorithms. We expect that the individual performance of the application is close to optimal, without sacrificing portability through the use of non-portable programming techniques. For each of the applications we have developed ports to OpenMP 3.0, OpenMP 4.0 and CUDA.

	Mem BW	**FLOPS**
Intel Xeon E5-2699 v4 @ 2.20GHz (22 core)	62 GB/s	??? G/flops
Intel Xeon Phi 7210 (64 core)	450 GB/s	??? G/flops
NVIDIA K20X Kepler	180 GB/s	??? G/flops
NVIDIA K40m Kepler	190 GB/s	??? G/flops

Table 2. The devices used in this performance analysis.

We will state the compilers and toolkit versions used as and when we present relevant data, and the devices we execute on can be found in Table 2. The devices we have chosen are present in many of the largest and most active supercomputers in the world, which we hope will strengthen the relevance of any findings.

As the heat diffusion solver is iterative, it is possible to have some variance in the number of iterations performed based on the problem dimensions and initial conditions. In order to make the results more comparable across devices and between experiments we fixed the iteration count.

7.2 Hot 2d + Flow 2d

Given the performance characteristics we have described for both Hot and Flow, we couldn't predict any significant issues that might arise when coupling them. We are presenting two applications that have been developed from scratch, and so we provide individual scaling results to demonstrate that each application performs to an acceptable level.

Scaling We provide the scaling of composing the two packages into a single application.

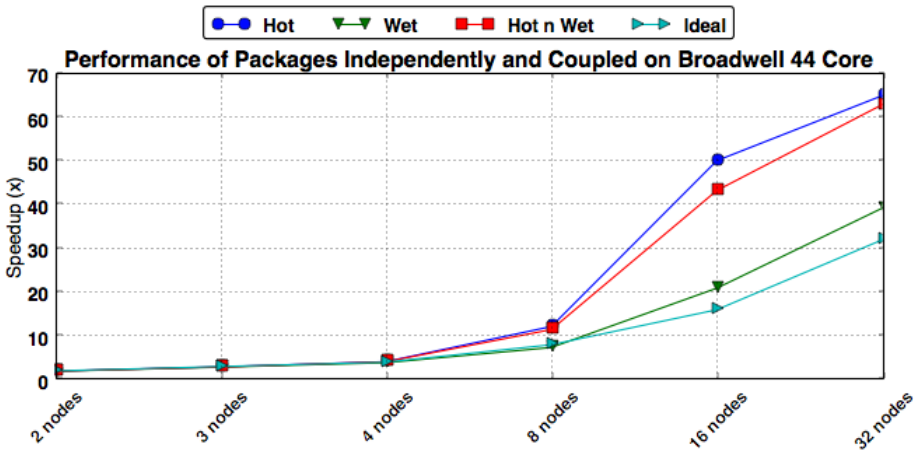

Fig. 2. Scaling Hot 2d, Flow 2d, and Hot 2d + Flow 2d on multiple nodes containing dual socket 22 core Broadwell CPUs.

As can be seen from Figure 2, the results of scaling the application across 32 nodes of dual socket 22 core Broadwell CPUs demonstrate a consistent performance profile. Each of the applications beat the ideal scaling performance over a single node due to the improved utilisation of cache that occurs as the problem is decomposed into smaller chunks. There is a noticeable gap in the scaling of the two packages at 16 nodes, that greatly reduces by the time that 32 nodes are used. This appears to be an effect of the increased data requirement that comes from executing two applications sequentially, where the overall capacity required to fit within cache is increased and so the improvements due to cache are slightly delayed.

Other Devices To ensure different effects are not seen with diverse architectures, we collected results from a range of devices.

We can see in Table 3 that, for the large problem size, device type did not influence the performance degradation, with all devices achieving similarly good

Device	Hot (s)	Flow (s)	Hot'n'Flow (s)	Difference (s)
Haswell 32 Core	119.3	10.3	129.9	0.28
Broadwell 44 Core	109.4	8.9	118.8	0.4
KNL	36.8	4.4	41.3	0.01
K20X	85.0	7.4	92.4	-0.08
Power8				

Table 3. The performance of Hot, Flow, and Hot'n'Flow on multiple devices for the 5000^2 for 50 iterations.

results. Overall this scaling profile suggests that the composition of the two application on a CPU has resulted in very little performance degradation. We wanted to continue this investigation on some other types of hardware to ensure that there weren't any other effects that we had not anticipated.

The results demonstrate that Hot and Flow are performance compatible, meaning they do not interfere with the performance profile of eachother. Although we have not seen a performance degradation at this scale, we think it will be important work to investigate how scaling is affected towards the limits of the scalability of each application.

An interesting feature of the two mini-apps is that they have vastly different performance weightings. For the 5000^2 problem size, which is a reasonable problem size for realy scientific problems, the capped inner iterations perfomed by *Hot* mean that the performance results shown in Table 3 do not truly represent the imbalance. The true disparity between the runtimes of the two packages is above an order of magnitude. We believe that this has some important implications for the optimisation of the applications when composed. Such an imbalance surely opens up the opportunity for successful overlapping of slow operations...

I HAVE LOADS OF PERFORMANCE RESULTS FOR THIS COUPLING, ON ALL DEVICES, WITH OPENMP, PROBLEM SCALING, SCALING ON MULTI-GPU etc. BUT I'M OMITTING AS THE COUPLING IS NOT INTERESTING

7.3 Fast 3d + Hot 3d

As mentioned throughout the paper, we expected that combining Fast 2d with any of the other mini-apps would lead to conflicts over domain decomposition. For Fast 2d the decomposition ends up being tightly coupled to the resulting algorithm, and in general the literature only suggested one dimensional decompositions for this particular type of problem in 2d. The underlying reason for this decision is that tiling the entire space results in increased inter-process communication. As the parallel FFT algorithm is considered bound by All-to-All communications, this option is therefore deficient.

Conflict Resolution In the event that the data structures within two packages conflict, we can theorise some potential resolutions:

- **Pick the best decomposition for the worst package** - This may or may not be easy depending on the restrictions imposed by the other packages. This is likely only suitable when you have a significant difference between the relative costs of the packages.
- **Choose some mutually beneficial layout** - This could be a complicated decision to make, and might lead to specific novel layouts for particular couplings. If the relative costs of the two packages are similar, then we expect this to be a good choice.
- **Perform a transposition of the data between packages** - Considering a simple two package application, with a victim that requires it's data transposed, this would lead to two transposition at entry to and exit from the victim package. Ideally this transposition could be overlapped with independent work.

8 Patterns Affecting Performance for Multi-Package Applications

Our research has uncovered a number of areas that we recognise present risks or opportunities to the performance of multi-package applications.

- **Diverse Computational Weightings** - Each package will expose some fixed or dynamic performance cost for particular problems. We observed with Hot and Flow that the difference in solve times differed by an order of magnitude depending on configuration.
- **Different Optimal Data Layouts** - If choosing an optimal data layout for a package results in a sub-optimal data layout for the other packages, the trade-off is going to be challenging to manage. It is also often the case that applications require different layouts for different architectures, which greatly amplifies this issue.
- **Competing Decompositions** - It is possible that different packages within an application might require different decompositions for optimal performance. We extend this discussion in Section 8.1 to predict a key bottleneck to multi-package scaling.
- **Load Imbalance** - If there is a significant load imbalance in one of the packages the usual approach would be to dynamically adapt the decomposition to re-balance, however when operating in a multi-package environment this will create a load imbalance for the other packages, which will be challenging to handle.
- **Capacity Requirements** - As you increase the number of packages that are included in an application, the capacity requirements can be inflated significantly. It will be important to overlap as much of the capacity as possible in order to minimise this effect. On devices with low capacity, such as accelerator devices, this issue might become a truly limiting factor.

We are expecting that continued research into the area of Dwarf composition will begin to uncover some of the points raised above, allowing us to begin tackling the issues.

An important feature of many algorithms that has not yet been acknowledged in this research is the structure of the computational mesh. The Unstructured Grid Dwarf describes those packages that have computational meshes formed of complex graph-based geometries. We are currently deferring the investigation of unstructured meshes, as such a mesh change is in general pervasive. Although it is possible that they exist, our group has never encountered an application that contains solvers for structured *and* unstructured meshes that are intended to be run as part of a multi-package solve.

As the unstructured characteristic greatly increases the complexity of each individual application and would have to be adopted in any applications destined for composition, we will defer this analysis for later investigation.

8.1 Diminishing Scalability for Increasing Core Counts

Modern supercomputing has reached a scale where the core counts at the node level are increasingly significantly each year, and this has major implications for writing portable and performant code. In particular, many of the largest supercomputers in the world include heterogeneous devices, such as accelerators, which greatly increase the core count, and require that significant data-parallelism is exposed within scientific application's algorithms in order to perform well.

Accelerator devices like the Intel Xeon Phi processors, have greater numbers of cores than existing Intel CPUs, for instance the Knights Landing has between 64 and 72, each with 4 hardware threads, whereas the Intel Xeon Broadwell can have up to 22 cores per socket, each core supporting up to 2 hardware threads. The IBM POWER8 CPU also exposes a large thread count, with up to 12 cores per socket and 8 hardware threads for a total of 192 threads.

This continual increase in the number of computational units available on a node has major implications for the scalability of codes, and this is amplified by the increasing number of nodes that are hosting those devices. It is well known that many algorithms reach a turnover point for strong scaling, past which communication costs begin to overwhelm the performance, and decomposing a problem will experience diminishing returns. Regardless of node-level application performance, as communication becomes the limiting factor, we are reaching a stage where the scalability of applications demands that the highly parallel applications exploit shared memory as much as possible.

We predict that it will be particularly important to understand the influence of composing multiple packages on the potential for exploiting local shared memory.

9 Future Compositions

We expect that it will be useful to extend this work to investigate the performance of diverse applications types, in particular we plan to develop new applications that cover the seven Dwarfs of parallel programming. The Bright

mini-app hosts a Monte Carlo neutral particle tracking algorithm that we are currently developing. Due to the inherent load imbalance of the problem, it is likely that the application will cause significant performance degradations in general.

Including more applications into the potential compositions leads to many possible combinations. For instance, taking our all four of our current mini-apps we could perform the following coupling:

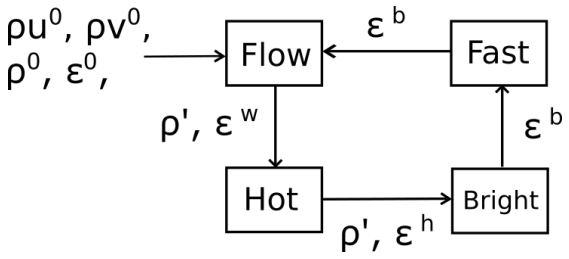

Fig. 3. The composition of a prospective multi-package application.

We would expect that this sort of package mix could be quite suitable for combustion, or astrophysics (MAKING THAT UP). The resulting performance profile would include lots of all to all communication, several competing decompositions and load imbalance.

10 Future Work

We further recognise that there are many requirements not handled by our existing applications, such as multiple materials, complex unstructured geometries, and adaptive mesh refinement. Each of these has a significant influence on the performance of individual algorithms, and likely introduces new issues for coupling packages together. As such it will be essential to consider new mini-apps that can proxy those particular features and consider their performance on new hardware.

11 Related Work

(SPEAK TO W ABOUT THIS SECTION, WHAT IS ACCEPTABLE TO BE TIED TO THIS PAPER)

12 Conclusion

Although we have only been able to compose three applications in this paper, we have already discovered and discussed some interesting problems that can arise in

the process. Importantly we have been able to observe an important fundamental rule with multi-package applications, that the complexity and computational cost of the individual package matters less to the overall performance of the application than the dominant conflicting performance characteristics present amongst the applications.

References