Composing the Dwarfs - A Performance Analysis of Coupled Mini-Apps

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 multiple 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, hydrodynamics, and use an FFT to solve Poisson's 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 individual 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 foresee when coupling applications together, motivating a stream of future work in the area.

Keywords: multi-package composition,parallel dwarfs,mini-apps,high-performance-computing

1 Introduction

A great amount of work is being devoted to understanding how scientific production applications will be ported to run on exascale supercomputers [1]. 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 [2].

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 [3] [4] [5]. 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 as 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. The majority of mini-apps contain individual algorithms and 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 Berkeley paper [6], a number of important classes of parallel algorithm were discussed and categorised 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 alluded to the necessity for those Dwarfs to be composed together and investigated as coupled applications. 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 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, and a Intel Xeon Haswell.
- 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.

3.1 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. [6], who outlined a number of Dwarfs, each of which describes a different computational class that features in modern supercomputing applications. Selecting algorithms that strongly correlate to the characteristics of an individual 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 this research is to expose some 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 3.2).
- **Sparse Linear Algebra** - *Hot*: A heat diffusion application, that uses a Conjugate Gradient linear solver (Section 3.3).
- **Structured Grid** - *Flow*: A Lagrangian-Eulerian hydrodynamics application that uses an explicit 5 point stencil (Section 3.4).

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. It is not our intention to introduce ground-breaking accuracy or methods into our applications.

3.2 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 [7] [8]. 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 Spectral class of algorithms.

We are currently relying on the optimised Intel MKL library version of the one dimensional FFT, and we then parallelise that manually using MPI. 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 [8].

Fundamental Method Following on from the discussions in Gholami et al. [9], we utilise the FFT to solve Poisson's equation for some dependent data passed to the solver.

$$\frac{d^2\phi}{dx^2} = \epsilon(x, y) \tag{1}$$

$$\text{fft}(\epsilon) = \hat{\epsilon} \tag{2}$$

$$\hat{\phi_{i,j}} = \frac{\hat{\epsilon_{i,j}} dx^2}{4 - \omega^i - \omega^{-i} - \omega^j - \omega^{-j}} \tag{3}$$

$$\text{fft}^{-1}(\hat{\phi}) = \phi \tag{4}$$

The solve phase calculates and FFT along the x and y axes of ϵ, before solving the equation in Fourier space and performing an inverse FFT to return the solution ϕ. With this current approach we are limited to periodic boundary conditions, and we plan to extend this to incorporate other boundary conditions in the future.

The arithmetic requirement of this package is low and the algorithm is generally straightforward, especially when compared to Flow, which requires a cohesion between many numerical methods. Maintaining a simple algorithm means that we will be able to extend this application more easily to 3d in the future to compare different decompositions.

Parallelisation In order to parallelise the FFT's it is necessary to perform a matrix transposition as part of each of the FFTs, which requires packing buffers, performing All-to-All communications with MPI, and then locally transposing the data. Due to the fact that we are performing an FFT before the solve, and then a subsequent inverse FFT, it is possible to reduce the number of transpositions from four to two, saving on the number of All-to-All communications.

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 the 2d implementation you are limited to a one dimensional decomposition, either row-wise or column-wise, and the parallelism is therefore limited by the problem dimensions. The reason for this restriction is that each process has data dependencies along each axis, and so tiling would greatly increase the amount of All-to-All communication required.

A 3d port of the application is imminent and will allow for greater flexibility, as the problem can be decomposed using either slabs or pencils. While the decomposition of the FFT can be easily solved and optimised for particular architecture in isolation, the 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.

3.3 Hot - Heat Conduction via a CG Solver

We have developed a simple conjugate gradient (CG) solver that implicitly solves the heat conduction equation. We do not precondition the problem, and utilise a standard iterative algorithm. The problem is memory bandwidth bound, until communication costs overwhelm during strong scaling. The problem is also well studied from a HPC optimisations perspective with a number of mini-apps solving the problem, and the HPCG benchmark now being actively used to benchmark supercomputers [10].

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

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

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

$$\begin{aligned} u_{i,j}^{n} = u_{i,j}^{n+1} &- \frac{\alpha dt}{dx^2}(u_{i,j-1}^{n+1} - 2u_{i,j}^{n+1} + u_{i,j+1}^{n+1}) \\ &- \frac{\alpha dt}{dy^2}(u_{i-1,j}^{n+1} - 2u_{i,j}^{n+1} + u_{i+1,j}^{n+1}) \end{aligned} \tag{7}$$

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. To calculate the edge centered densities from the cell centered values on the mesh, we simply use the arithmetic mean.

Parallelisation The best decomposition for this mini-app is generally a regular Cartesian tile, 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 two scalar values, α and β, are calculated, which need 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 contains some simple sparse linear algebra operation, 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.

3.4 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 [11].

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 control limits 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 [12]. 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 conduction application, the boundary conditions we chose for this application are reflective, which is particularly useful for tracking conservation to verify the results.

Parallelisation As with the heat conduction application, the algorithm does not have any inherent load imbalance, and only requires nearest neighbour communication of halo regions. 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.

4 Performance Characteristics of the Mini-Apps

Although the Dwarfs imply some high-level classification of algorithms, 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.

We have seen in previous work which of those characteristics matter to the individual application, such as *Memory Bandwidth* for performance of memory bound packages and *All-to-All* communications for scalability. Our intention is to investigate how the computational and communicational profiles of those applications interact with each other when packages are composed and make some commentary about the dominant performance characteristics.

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 can pre-emptively assess some of the likely effects that running multiple mini-apps at once lead to. 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. Table 1 lists some of the key performance characteristics of the applications that we investigate in this paper.

Some communication costs could be saved if variables could be communicated once for all applications, during the *Nearest Neighbour* or *All to All communications*, for instance. We do no expose any *Load Imbalance* as part of our suite of applications, but are currently in the process of developing *Bright*, which is a Monte Carlo neutral particle tracking application, that we expect to suffer from significant load balancing issues. As two of the applications are *Memory Bandwidth* bound, it might be possible to overlap the costly memory access delays with independent computation from other applications. It is also possible that data structures between the individual applications will require some form of transformation, which will have a major influence on performance.

5 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 have been carefully validated to ensure that they accurately solve the chosen test methods.

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) reducing the algorithms into a single solve would not capture the potential difficulties of coupling distinct packages together.

In general we do not consider the coupling approach to be necessarily valid from a scientific standpoint. Our intention is 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.

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 for temperature μ, and passes back the updated energy to start the cycle again.

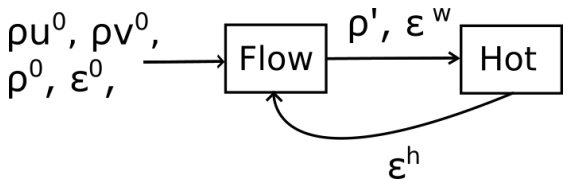

Fig. 1. The data dependency of our coupled Hot and Flow applications.

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.

6 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 on a range of different platforms to observe whether there were any unexpected architectural influences.

6.1 Experimental Setup

Each of the mini-apps has been optimised individually, drawing on the experience other mini-app optimisation exercises [13] [14]. As the applications are all brand new, we recognise that their individual performance could be improved in places, but believe the results will be relevant regardless of whether the applications require future tuning. When composing production applications it is possible that one or more of the applications will be sub-optimal, and the definition of optimal may change depending upon the nature of the composition.

	Achievable Mem BW	Peak FLOPS
Intel Xeon CPU E5-2670 v3 2.30 GHz	56 GB/s	500 Gflops
Intel Xeon Phi 7210 (64 core)	450 GB/s	2.6 Tflops
NVIDIA K40m Kepler	190 GB/s	1.4 Tflops

Table 2. The devices used in this performance analysis.

Table 2 lists the devices that we are presenting in this version of the paper. We have access to more modern devices including Intel Broadwell CPUs and NVIDIA P100 GPUs, which we are currently collecting results on.

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.

6.2 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. Both of the applications differ in their methods, as Hot is a linear solver, and Flow is a set of explicit advection routines. However, they both work on a structured mesh and solve using a five point stencil, as well as sharing nearest neighbour communications.

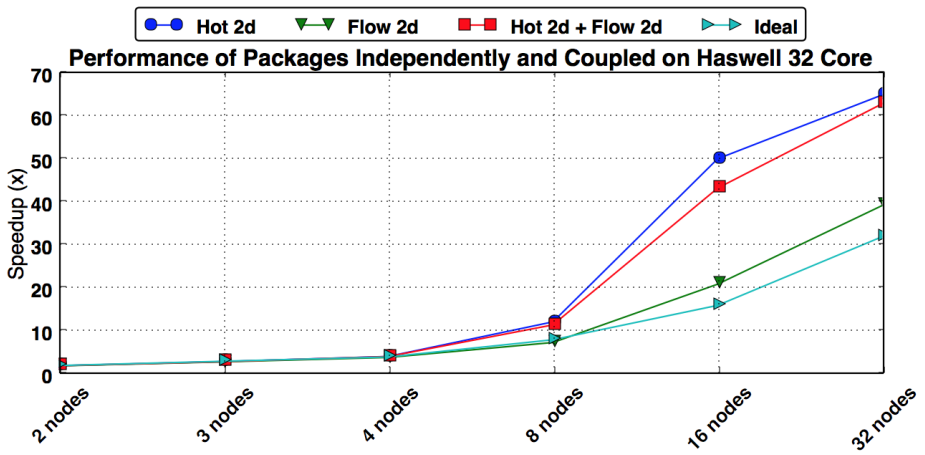

Fig. 2. Scaling Hot 2d, Flow 2d, and Hot 2d + Flow 2d on multiple nodes containing dual socket 16 core Haswell CPUs.

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

As can be seen in Figure 2, scaling the application across 32 nodes of dual socket 16 core Haswell CPUs demonstrates 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 large 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 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. Table 3 shows that, for the large problem size, device type did not influence the performance degradation, with all devices achieving similarly good results.

Device	Hot 2d	Flow 2d	Hot 2d + Flow 2d	Difference
Haswell (32 Cores)	119.3s	10.3s	129.9s	0.0%
KNL	36.8s	4.4s	41.3s	0.0%
K40m	85.0s	7.4s	92.4s	0.0%

Table 3. The performance of Hot, Flow, and Hot + Flow on multiple devices for the 5000^2 for 50 iterations. The code run on the Haswell (32 Core) was compiled with the Cray compilers, version 8.5, while the KNL was Intel 17, and the K40m was CUDA 8.0.

The results demonstrate that Hot and Flow are performance compatible, meaning they do not interfere with the performance profile of each other. Although we have not seen a performance degradation at this scale, we think it will be important work to investigate how scaling is affected as the node count is increased further, and whether the effectiveness of using shared memory to improve scalability is altered because of the compositions.

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, we observe an order of magnitudes difference in the run times of the two applications (Table 3). There are some important issues with having such a disparity in the run times. It may be possible to optimise for a single application and generally ignore the performance of the other. Also, it might be profitable to adjust the numerical method of the faster package in order to increase accuracy or improve some other beneficial metric.

6.3 Fast 2d + Hot 2d

As mentioned throughout the paper, we considered 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 [15]. 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.

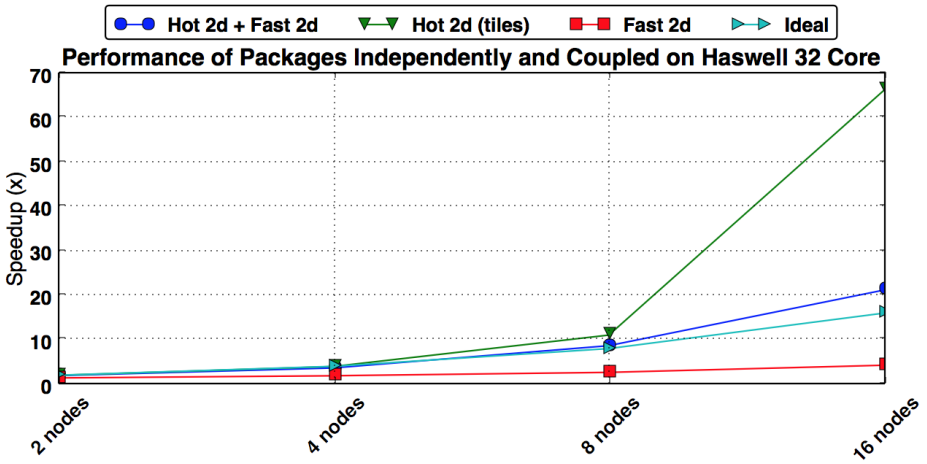

Fig. 3. Scaling Hot 2d, Fast 2d, and Hot 2d + Fast 2d on multiple nodes containing dual socket Haswell CPUs (32 cores). The problem size was 5000^2 for 10 iterations, and the applications were compiled with the Intel Compilers version 16.3.

At this problem size, the scaling of Fast was deficient, and this had a significant impact on the scalability of the composed application (3). The super-linear speedup achieved by Hot due to cache effects is negated by the overwhelming dominance of the All-to-All communications. The time to solution of Hot is an order of magnitude higher on 1 node, but by 16 nodes Hot solves the problem 2x faster than Fast.

We understand that the All-to-All communications are a significant bottleneck, and will attempt overlapping the communication with some of the compute. As there is so much time spent communicating, it might be possible to overlap some of the communication in Fast with independent computation in Hot.

Decomposition Effect Figure 3 shows the scaling of the *tiles* decomposition as it is the best performing option, however the scaling of the *rows* decomposition is significantly worse for than the *tiles* decomposition as you increase the node

count. For this particular application, the communication costs are increased as the decomposition is reduced to a wide tile with large halos.

Device	1 node	2 nodes	4 nodes	8 nodes	16 nodes
Hot (tiles)	27.82s	14.12s	7.01s	2.34s	0.42s
Hot (rows)	27.79s	14.02s	6.83s	2.39s	0.72s
Tiles vs Rows	1.00x	0.99x	0.97x	1.02x	1.72x

Table 4. The absolute runtime of the Hot application scaled across many cores, comparing the performance of tiles vs row decompositions. The problem was 5000^2 for 10 iterations, and the application was compiled with Intel 16.3.

The results in Table 4 demonstrate that when there are a small number of ranks, there is little to no difference between the decompositions. At the point that the communication becomes dominant, the performance shifts drastically and the tiles decomposition is 1.72x faster than the rows decomposition. We plan to immediately collect results that go to higher nodes counts in order to test whether the decomposition difference is maintained or becomes hidden by the communication overheads.

At 16 nodes, Fast solves the problem in 0.77s, and the runtime of Hot 2d + Fast 2d is 1.47s. Considering the results in Table 4 it makes sense given that Hot was forced to use row decomposition, but could be 1.24x faster if Hot were able to select the optimal decomposition. It is not currently clear how this will change with increasing node counts but we plan to continue the study to thousands of cores to understand the long term strong scaling implications.

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

- **Pick the best decomposition with the largest impact on a single application** - 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** - In the case of a simple two package application, consider a 'victim' that will have it's data transposed to resolve dependencies. This would lead to two transpositions, one at entry to and one at exit from the victim package. Ideally this transposition could be overlapped with independent work.

Although our 2d implementation limits us to picking the best decomposition for Fast 2d, we recognise that the 3d version will allow a choice between multiple

decompositions. Much of this discussion focusses upon the algorithms at scale rather than single node performance where we have observed good composition performance for all of the algorithms.

An important technique for improving the symptoms of poor scaling due to communication overheads is the exploitation of shared memory. We are currently in the process of measuring the effects of reducing the number of MPI ranks required using OpenMP for node-level parallelism in each application. Our expectation is that this will further delay the effects of the performance degradation caused by the decomposition choices.

7 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 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 7.1 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 and we don't yet have a solution to.
- **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 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.

7.1 Diminishing Scalability for Increasing Core Counts

Modern supercomputing has reached a scale where the core counts at the node level are increasing 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 abundant 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.

Exploiting shared memory for a single package can be challenging, depending on the structure of the algorithms it contains. Combining multiple packages using shared memory and hybridising with MPI could cause major issues from a thread placement/affinity perspective.

8 Future Compositions

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.

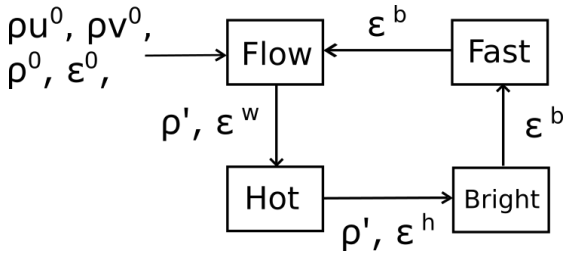

Fig. 4. The composition of a prospective multi-package application.

Including more applications into the potential compositions leads to many possible combinations. For instance, taking all four of our in-development mini-apps we could perform the coupling detailed in Figure 4.

Although the combination of applications might not appear in production applications, we expect that this combination of performance profiles would likely exist in codes ranging from Astrophysics to modelling Combustion in engines. The resulting performance profile would include significant All-to-All communication, several competing decompositions and load imbalance. We have not yet been able to determine the capacity requirements of such an application, but expect it to become an issue.

9 Future Work

We 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.

10 Related Work

Karlin et al. [16] performed a number optimisations such a loop fusion and data layout changes to improve the performance of the shock hydro mini-app LULESH. In previous work our group has used mini-apps, including TeaLeaf, RotorSim and BUDE, to investigate the performance of parallel programming models, and evaluate modern architecture [17] [5]. Bird et al. [13] performed algorithmic optimisations and developed an OpenCL port of EPOCH3D, a particle-in-cell mini-app.

Hart et al. [3] ported the NekBone mini-app to OpenMP 4.0 to conduct a performance evaluation of the programming model. Ivanov et al. [14] optimised the MPI communication strategy within Nekbone, improving its parallel performance. Reguly et al. [18] used the OPS DSL to optimise the CloverLeaf fluid dynamics code.

11 Conclusion

As architectures are becoming more complex, domain scientists will rely more upon computer scientists to assist with optimising their applications for modern supercomputers. Mini-apps are the perfect vehicle for such investigations, and are being used extensively. Discovering which aspects of real production applications are not covered by the individual applications is essential, and the purpose of this research was to motivate further investigation into those problems.

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 a fundamental rule with multi-package applications, that the complexity and computational cost of the individual package matters less to the overall performance of the composed application than the dominant conflicting performance characteristics present amongst the applications.

Changes that directly influence shared data structures are very likely to affect the performance of individual packages. In our case the mesh data could be maintained on a regular structured mesh for all of the applications, but the way that the mesh was decomposed was directly influenced by the FFT application, Fast 2d. Depending on the problem dimensions and level of scaling, this had a significant impact on the performance of the combined solve. In order to reduce the effect of this issue, the utilisation of shared memory is essential, although this will only defer the problem by some factor.

It is now clear that there are fewer incidental effects than might be expected, as the runtime of each application summed together perfectly in the majority of cases. There are many more combinations of applications that need to be investigated, and concentrating on those applications that have conflicting performance characteristics will provide the most insight.

References

1. P. Kogge and J. Shalf, "Exascale Computing Trends: Adjusting to the "New Normal" for Computer Architecture," *Computing in Science & Engineering*, vol. 15, no. 6, pp. 16–26, 2013.
2. M. Heroux, D. Doerfler *et al.*, "Improving Performance via Mini-applications," Sandia National Laboratories, Tech. Rep. SAND2009-5574, 2009.
3. A. Hart, "First Experiences Porting a Parallel Application to a Hybrid Supercomputer with OpenMP 4.0 Device Constructs," in *OpenMP: Heterogenous Execution and Data Movements: 11th International Workshop on OpenMP, IWOMP 2015, Proceedings*, 2015, pp. 73–85.

4. G. Bercea, C. Bertolli, S. Antao, A. Jacob *et al.*, "Performance Analysis of OpenMP on a GPU using a Coral Proxy Application," in *Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems*. ACM, 2015, p. 2.
5. M. Martineau, S. McIntosh-Smith, M. Boulton, and W. Gaudin, "An Evaluation of Emerging Many-Core Parallel Programming Models," in *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores*, ser. PMAM'16, 2016.
6. K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis *et al.*, "The Landscape of Parallel Computing Research: A View from Berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006.
7. P. N. Swarztrauber, "Multiprocessor ffts," *Parallel computing*, vol. 5, no. 1-2, pp. 197–210, 1987.
8. D. Takahashi, "An Implementation of Parallel 3-D FFT with 2-D Decomposition on a Massively Parallel Cluster of Multi-Core Processors," in *International Conference on Parallel Processing and Applied Mathematics*. Springer, 2009, pp. 606–614.
9. A. Gholami, D. Malhotra, H. Sundar, and G. Biros, "FFT, FMM, or Multigrid? A comparative study of state-of-the-art Poisson solvers in the unit cube," *arXiv preprint arXiv:1408.6497*, 2014.
10. J. Dongarra and M. Heroux, "Toward a New Metric for Ranking High Performance Computing Systems," Sandia National Laboratories, Tech. Rep. SAND2013-4744, 2013.
11. M. Berger, M. J. Aftosmis, and S. M. Murman, "Analysis of Slope Limiters on Irregular Grids," *AIAA paper*, vol. 490, no. 2005, pp. 1–22, 2005.
12. R. L. Bowers and J. R. Wilson, "Numerical Modeling in Applied Physics and Astrophysics," *Boston: Jones and Bartlett, c1991.*, vol. 1, 1991.
13. R. F. Bird, S. J. Pennycook, S. A. Wright, and S. A. Jarvis, "Towards a Portable and Future-Proof Particle-in-Cell Plasma Physics Code," *Proceedings of 1st International Workshop on OpenCL (IWOCL 13)*, 2013.
14. I. Ivanov, J. Gong, D. Akhmetova, I. B. Peng, S. Markidis, E. Laure, R. Machado, M. Rahn, V. Bartsch, A. Hart *et al.*, "Evaluation of Parallel Communication Models in Nekbone, a Nek5000 mini-application," in *2015 IEEE International Conference on Cluster Computing*. IEEE, 2015, pp. 760–767.
15. S. Pissis, "Parallel Fourier Transformations Using Shared Memory Nodes," *MSc in High Performance Comput ing (Univ. of Edinburgh, 2008)*, 2008.
16. I. Karlin, J. McGraw, E. Gallarado, J. Keasler, E. Leon, and B. Still, "Memory and parallelism tuning exploration using the lulesh proxy application," *2012 SC Companion: High Performance Computing, Networking Storage and Analysis (SCC 2012)*, 2012.
17. S. McIntosh-Smith, M. Boulton, D. Curran, and J. Price, "On the Performance Portability of Structured Grid Codes on Many-Core Computer Architectures," in *Supercomputing*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2014, vol. 8488, pp. 53–75.
18. I. Z. Reguly, G. R. Mudalige, M. B. Giles, D. Curran, and S. McIntosh-Smith, "The ops domain specific abstraction for multi-block structured grid computations," in *Proceedings of the Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, ser. WOLFHPC '14, 2014.