SWIFT: Using task-based parallelism, fully asynchronous communication, and graph partition-based domain decomposition for strong scaling on more than 100 000 cores.

Matthieu Schaller
Institute for Computational Cosmology (ICC)
Department of Physics
Durham University
Durham DH1 3LE, UK
matthieu.schaller@durham.ac.uk

Pedro Gonnet
School of Engineering and Computing Sciences
Durham University
Durham DH1 3LE, UK

Google Switzerland GmbH
8002 Zürich, Switzerland

Aidan B. G. Chalk
School of Engineering and Computing Sciences
Durham University
Durham DH1 3LE, UK

Peter W. Draper
Institute for Computational Cosmology (ICC)
Department of Physics
Durham University
Durham DH1 3LE, UK

ABSTRACT

We present a new open-source cosmological code, called SWIFT, designed to solve the equations of hydrodynamics using a particle-based approach (Smooth Particle Hydrodynamics) on hybrid shared / distributed-memory architectures. SWIFT was designed from the bottom up to provide excellent *strong scaling* on both commodity clusters (Tier-2 systems) and Top100-supercomputers (Tier-0 systems), without relying on architecture-specific features or specialized accelerator hardware. This performance is due to three main computational approaches:

- **Task-based parallelism** for shared-memory parallelism, which provides fine-grained load balancing and thus strong scaling on large numbers of cores.
- **Graph-based domain decomposition**, which uses the task graph to decompose the simulation domain such that the *work*, as opposed to just the *data*, as is the case with most partitioning schemes, is equally distributed across all nodes.
- **Fully dynamic and asynchronous communication**, in which communication is modelled as just another task in the task-based scheme, sending data whenever it is ready and deferring on tasks that rely on data from other nodes until it arrives.

In order to use these approaches, the code had to be re-written from scratch, and the algorithms therein adapted to the task-based paradigm. As a result, we can show upwards of 60% parallel efficiency for moderate-sized problems when increasing the number of cores 512-fold, on both x86-based and Power8-based architectures.

Keywords

Physics; Cosmology; Fluid dynamics; Smooth Particle Hydrodynamics; Task-based parallelism; Asynchronous data transfer; Extreme scaling

1. INTRODUCTION

For the past decade physical limitations have kept the speed of individual processor cores constrained, so instead of getting *faster*, computers are getting *more parallel*. Systems containing up to 64 general-purpose cores are becoming commonplace, and the number of cores can be expected to continue growing exponentially, e.g. following Moore's law, much in the same way processor speeds were up until a few years ago.

As a consequence, in order to get faster, computer programs need to rely on better exploitation of this massive parallelism, i.e. they need to exhibit *strong scaling*, the ability to run roughly N times faster when executed on N times as many processors. Without strong scaling, massive parallelism can still be used to tackle ever *larger* problems, but not make fixed-size problems *faster*.

Although this switch from growth in speed to growth in parallelism has been anticipated and observed for quite some time, very little has changed in terms of how we design and implement parallel computations. Branch-and-bound synchronous parallelism using OpenMP[10] and MPI[28], as well as domain decompositions based on geometry or space-filling curves [34] are still commonplace, despite both the architectures and problem scales having changed dramatically since their introduction.

PASC '16 June 8–10, 2016, Lausanne, Switzerland

ACM ISBN 978-1-4503-2138-9.

DOI: 10.1145/1235

The design and implementation of SWIFT[1] [19, 30, 17], a large-scale cosmological simulation code built from scratch, provided the perfect opportunity to test some newer approaches, i.e. task-based parallelism, fully asynchronous communication, and graph partition-based domain decompositions.

This paper describes these techniques, which are not exclusive to cosmological simulations or any specific architecture, as well as the results obtained with them. While [19, 17] already describes the underlying algorithms in more detail, in this paper we focus on the parallelization strategy and the results obtained on larger Tier-0 systems.

2. PARTICLE-BASED HYDRODYNAMICS

Smoothed Particle Hydrodynamics [14, 25] (SPH) uses particles to represent fluids. Each particle p_i has a position $\mathbf{x}_i$, velocity $\mathbf{v}_i$, internal energy u_i, mass m_i, and a smoothing length h_i. The particles are used to interpolate any quantity Q at any point in space as a weighted sum over the particles:

$$Q(\mathbf{r}) = \sum_i m_i \frac{Q_i}{\rho_i} W(\|\mathbf{r} - \mathbf{r}_i\|, h) \tag{1}$$

where Q_i is the quantity at the ith particle, h is the *smoothing length*, i.e. the radius of the sphere within which data will be considered for the interpolation, and $W(r, h)$ is the *smoothing kernel* or *smoothing function*. Several different forms for $W(r, h)$ exist, each with their own specific benefits and drawbacks.

The particle density ρ_i used in (1) is itself computed similarly:

$$\rho_i = \sum_{j,\ r_{ij} < h_i} m_j W(r_{ij}, h_i) \tag{2}$$

where $r_{ij} = \|\mathbf{r_i} - \mathbf{r_j}\|$ is the Euclidean distance between particles p_i and p_j. In compressible simulations, the smoothing length h_i of each particle is chosen such that the number of neighbours with which it interacts is kept more or less constant, and can result in smoothing lengths spanning several orders of magnitudes within the same simulation.

Once the densities ρ_i have been computed, the time derivatives of the velocity and internal energy, which require ρ_i, are computed as follows:

$$\frac{dv_i}{dt} = -\sum_{j,\ r_{ij} < \hat{h}_{ij}} m_j \left[\frac{P_i}{\Omega_i \rho_i^2} \nabla_r W(r_{ij}, h_i) + \frac{P_j}{\Omega_j \rho_j^2} \nabla_r W(r_{ij}, h_j) \right], \tag{3}$$

$$\frac{du_i}{dt} = \frac{P_i}{\Omega_i \rho_i^2} \sum_{j,\ r_{ij} < h_i} m_j (\mathbf{v}_i - \mathbf{v}_j) \cdot \nabla_r W(r_{ij}, h_i), \tag{4}$$

where $\hat{h}_{ij} = \max\{h_i, h_j\}$, and the particle pressure $P_i = \rho_i u_i (\gamma - 1)$ and correction term $\Omega_i = 1 + \frac{h_i}{3\rho_i} \frac{\partial \rho}{\partial h}$ are computed on the fly.

The computations in (2), (3), and (4) involve finding all pairs of particles within range of each other. Any particle p_j is *within range* of a particle p_i if the distance between p_i and p_j is smaller or equal to the smoothing distance h_i of p_i, e.g. as is done in (2). Note that since particle smoothing lengths may vary between particles, this association is not symmetric, i.e. p_j may be in range of p_i, but p_i not in range of p_j. If $r_{ij} < \max\{h_i, h_j\}$, as is required in (3), then particles p_i and p_j are within range *of each other*.

[1] SWIFT is an open-source software project and the latest version of the source code, along with all the data needed to run the test cased presented in this paper, can be downloaded at www.swiftsim.com.

The computation thus proceeds in two distinct stages that are evaluated separately:

1. *Density* computation: For each particle p_i, loop over all particles p_j within range of p_i and evaluate (2).

2. *Force* computation: For each particle p_i, loop over all particles p_j within range of each other and evaluate (3) and (4).

Finding the interacting neighbours for each particle constitutes the bulk of the computation. Many codes, e.g. in Astrophysics simulations [14], rely on spatial *trees* for neighbour finding [14, 20, 29, 33], i.e. k-d trees [7] or octrees [24] are used to decompose the simulation space. In Astrophysics in particular, spatial trees are also a somewhat natural choice as they are used to compute long-range gravitational interactions via a Barnes-Hut [6] or Fast Multipole [9] method. The particle interactions are then computed by traversing the list of particles and searching for their neighbours in the tree. In current state-of-the-art simulations (e.g. [27]), the gravity calculation corresponds to roughly one quarter of the calculation time whilst the hydrodynamics scheme takes approximately half of the total time. The remainder is spent in the astrophysics modules, which contain interactions of the same kind as the SPH sums presented here. Gravity could, however, be vastly sped-up compared to commonly used software by employing more modern algorithms such as the Fast Multipole Method [9].

Although such tree traversals are trivial to parallelize, they have several disadvantages, e.g. with regards to computational efficiency, cache efficiency, and exploiting symmetries in the computation (see [17] for a more detailed analysis).

3. PARALLELIZATION STRATEGY

One of the main concerns when developing SWIFT was to break with the branch-and-bound type parallelism inherent to parallel codes using OpenMP and MPI, and the constant synchronization between computational steps it results in.

If *synchronization* is the main problem, then *asynchronicity* is the obvious solution. We therefore opted for a *task-based* approach for maximum single-node, or shared-memory, performance. This approach not only provides excellent load-balancing on a single node, it also provides a powerful model of the computation that can be used to distribute the work equitably over a set of distributed-memory nodes using general-purpose graph partitioning algorithms. Finally, the necessary communication between nodes can itself be modelled in a task-based way, interleaving communication seamlessly with the rest of the computation.

3.1 Task-based parallelism

Task-based parallelism is a shared-memory parallel programming paradigm in which a computation is broken-down in to a set of *tasks* which can be executed concurrently. In order to ensure that the tasks are executed in the right order, e.g. that data needed by one task is only used once it has been produced by another task, *dependencies* between tasks are specified and strictly enforced by a task scheduler. Additionally, if two tasks require exclusive access to the same resource, yet in no particular order, they are treated as *conflicts* and the scheduler ensures that they are not executed concurrently. Computations described in this way then parallelize trivially: each processor repeatedly grabs a task for which all dependencies have been satisfied and executes it until there are no tasks left.

The main advantages of using a task-based approach are

- The order in which the tasks are processed, and how they are

assigned to each processor is completely dynamic and adapts automatically to load imbalances.

- If the dependencies and conflicts are specified correctly, there is no need for expensive explicit locking, synchronization, or atomic operations to deal with most concurrency problems.
- Each task has exclusive access to the data it is working on, thus improving cache locality and efficiency.

Task-based parallelism is not a particularly new concept and therefore several implementations thereof exist, e.g. Cilk [8], QUARK [35], StarPU [4], SMP Superscalar [1], OpenMP 3.0 [13], Intel's TBB [26], and, to some extent, Charm++ [21].

Since none of these existing taks-based libraries provided the flexibility required to experiment with different scheduling and communication techniques, (SWIFT is an interdisciplinary effort between Computer Science and Astrophysics to study not only cosmological phenomena, but also novel simulation algorithms and parallel computing techniques) we chose to implement our own task scheduler in SWIFT, which has since been back-ported as the general-purpose QUICKSCHED task scheduler [18]. In QUICKSCHED and SWIFT, task dependencies are specified explicitly, as opposed to being implicitly derived from data dependencies, allowing us to more easily build complex task hierarchies. This also allowed us to extend the scheduler with the concept of task conflicts and integrate the asynchronous communication scheme described further on.

Despite its advantages, and the variety of implementations, task-based parallelism is rarely used in practice (notable exceptions include the PLASMA project [2] which uses QUARK/StarPU, and the `deal.II` project [5] which uses Intel's TBB). The main problem is that to effectively use task-based parallelism, most computations need to be completely redesigned to fit the paradigm, which is usually not an option for large and complex codebases.

Since we were re-implementing SWIFT from scratch, this was not an issue. The tree-based neighbour-finding described above was replaced with a more task-friendly approach as described in [17], in which the domain is first decomposed into a grid of *cells* of edge length larger or equal to the largest particle radius. An initial set of interaction tasks is then defined over all cells and pairs of neighbouring cells, such that if two particles are close enough to interact, they are either in the same cell or they span a pair of neighbouring cells. These initial interaction tasks are then refined by recursively splitting cells that contain more than a certain number of particles and replacing tasks that span a pair of split cells with tasks spanning the neighboring sub-cells. The resulting refined set of tasks contains all the cells and pairs of cells over which particle interactions must be computed.

The dependencies between the tasks are set following equations (2), (3), and (4), i.e. such that for any cell, all the tasks computing the particle densities therein must have completed before the particle forces can be computed, and all the force computations must have completed before the particle velocities may be updated. The task hierarchy is shown in Fig. 1, where the particles in each cell are first sorted (round tasks) before the particle densities are computed (first layer of square tasks). Ghost tasks (triangles) are used to ensure that all density computations on a cell of particles have completed before the force evaluation tasks (second layer of square tasks) execute. Once all the force tasks on a cell of particles have completed, the integrator tasks (inverted triangles) update the particle positions and velocities. The decomposition was computed such that each cell contains ~ 100 particles, which leads to tasks of up to a few milliseconds each.

Due to the cache-friendly nature of the task-based computations, and their ability to exploit symmetries in the particle interactions,

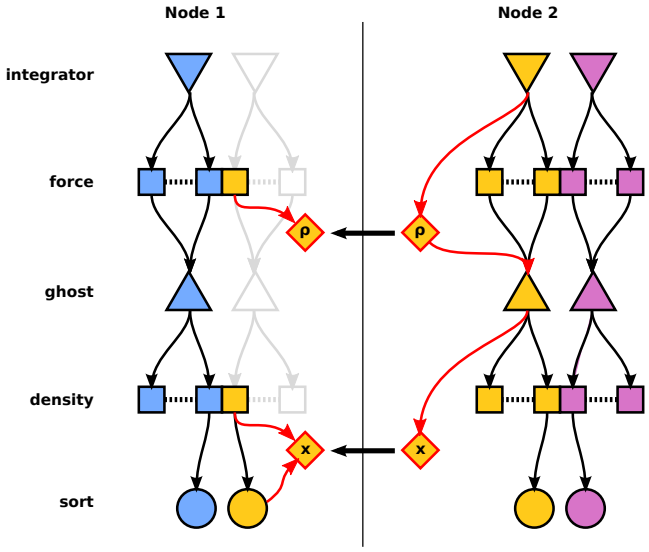

Figure 1: Task hierarchy for the SPH computations in SWIFT, including communication tasks. Arrows indicate dependencies, i.e. an arrow from task A to task B indicates that A depends on B. The task color corresponds to the cell or cells it operates on, e.g. the density and force tasks work on individual cells or pairs of cells. The task hierarchy is shown for three cells: blue, yellow, and purple. Although the data for the yellow cell resides on Node 2, it is required for some tasks on Node 1, and thus needs to be copied over during the computation using `send`/`recv` tasks (diamond-shaped).
Figure adapted from [17].

the task-based approach is already more efficient than the tree-based neighbour search on a single core, and scales efficiently to all cores of a shared-memory machine [17].

3.2 Task-based domain decomposition

Given a task-based description of a computation, partitioning it over a fixed number of *ranks* (using the MPI terminology) is relatively straight-forward: we create a *cell hypergraph* in which:

- Each *node* represents a single cell of particles, and,
- each *edge* represents the tasks, connecting the cells.

Since in the particular case of SWIFT each task references at most two cells, the cell hypergraph is just a regular *cell graph*.

Any partition of the cell graph represents a partition of the computation, i.e. the nodes belonging to each partition each belong to a rank, and the data belonging to each cell resides on the partition/rank to which it has been assigned. Any task spanning cells that belong to the same partition needs only to be evaluated on that rank/partition, and tasks spanning more than one partition need to be evaluated on both ranks/partitions.

If we then weight each edge with the computational cost associated with the tasks, then finding a *good* cell distribution reduces to finding a partition of the cell graph such that the maximum sum of the weight of all edges within and spanning in a partition is minimal (see Fig. 2). Since the sum of the weights is directly proportional to the amount of computation per rank/partition, minimizing the maximum sum corresponds to minimizing the time spent on the slowest rank. Computing such a partition is a standard graph prob-

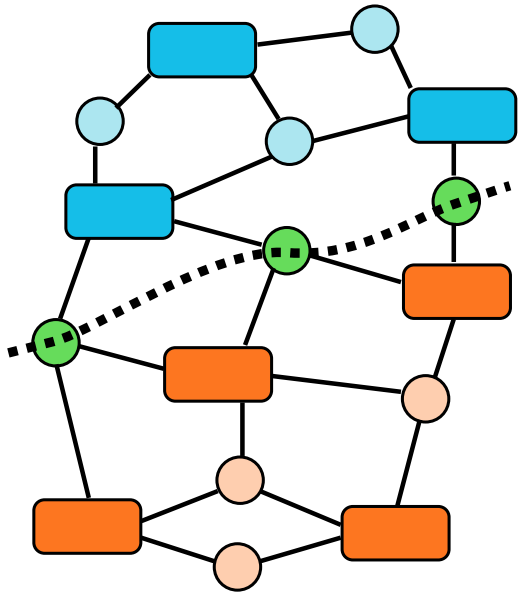

Figure 2: Illustration of the task-based domain decomposition in which the tasks (circles) are *hyperedges* that connect one or more resources (rectangles). The resources are partitioned along the thick dotted line. The blue and orange tasks are executed on the respective partitions, whereas the green tasks/hyperedges along the cut line are executed on both. The cost of this partition is the sum of the green tasks, which are computed twice, as well as the cost imbalance of the tasks executed in each partition.

lem and several software libraries which provide good solutions[2], e.g. METIS [22] and Zoltan [11], exist.

In SWIFT, the graph partitioning is computed using the METIS library. The cost of each task is initially approximated via the asymptotic cost of the task type and the number of particles involved. After a task has been executed, it's effective computational cost is computed and used.

Note that this approach does not explicitly consider any geometric constraints, or strive to partition the *amount* of data equitably. The only criteria is the computational cost of each partition, for which the task decomposition provides a convenient model. We are therefore partitioning the *computation*, as opposed to just the *data*.

Note also that the proposed partitioning scheme takes neither the task hierarchy, nor the size of the data that needs to be exchanged between partitions/ranks into account. This approach is therefore only reasonable in situations in which the task hierarchy is wider than flat, i.e. the length of the critical path in the task graph is much smaller than the sum of all tasks, and in which communication latencies are negligible.

3.3 Asynchronous communications

Although each particle cell resides on a specific rank, the particle data will still need to be sent to any neighbouring ranks that have tasks that depend on this data, e.g. the the green tasks in Fig. 2. This communication must happen twice at each time-step: once to send the particle positions for the density computation, and then again once the densities have been aggregated locally for the force computation.

Most distributed-memory codes based on MPI [28] separate computation and communication into distinct steps, i.e. all the ranks first exchange data, and only when the data exchange is complete does computation start. Further data exchanges only happen once computation has finished, and so on. This approach, although conceptually simple and easy to implement, has three major drawbacks:

- The frequent synchronization points between communication and computation exacerbate load imbalances,
- the communication phase consists mainly of waiting on latencies, during which the node's CPUs usually run idle, and
- during the computation phase, the communication network is left completely unused, whereas during the communication phase, all ranks attempt to use it at the same time.

It is for these reasons that in SWIFT we opted for a fully *dynamic and asynchronous* communication model in which local data is sent whenever it is ready, data from other ranks is only acted on once it has arrived, and there is no separation into communication and computational phases. In practice this means that no rank will sit idle waiting on communication if there is any computation that can be done.

This fits in quite naturally within the task-based framework by modelling communication as just another task type, i.e. adding tasks that send and receive particle data between ranks. For every task that uses data that resides on a different rank, `send` and `recv` tasks are generated automatically on the source and destination ranks respectively. At the destination, the task is made dependent of the `recv` task, i.e. the task can only execute once the data has actually been received. This is illustrated in Fig. 1, where data is exchanged across two ranks for the density and force computations and the extra dependencies are shown in red.

The communication itself is implemented using the non-blocking `MPI_Isend` and `MPI_Irecv` primitives to initiate communication, and `MPI_Test` to check if the communication was successful and resolve the communication task's dependencies. In the task-based scheme, strictly local tasks which do not rely on communication tasks are executed first. As data from other ranks arrive, the corresponding non-local tasks are unlocked and are executed whenever a thread picks them up.

One direct consequence of this approach is that instead of a single `send`/`recv` call between each pair of neighbouring ranks, one such pair is generated for each particle cell. This type of communication, i.e. several small messages instead of one large message, is usually strongly discouraged since the sum of the latencies for the small messages is usually much larger than the latency of the single large message. This, however, is of no concern in SWIFT since nobody is actively waiting to receive the messages in order, and the communication latencies are covered by local computations. A nice side-effect of this approach is that communication no longer happens in bursts involving all the ranks at the same time, but is more or less evenly spread over the entire computation, and is therefore less demanding of the communication infrastructure.

4. SCALING TESTS

In this section we present some strong scaling tests of the SWIFT code on different architectures for a representative cosmology problem.

4.1 Simulation setup

[2]Computing the optimal partition for more than two nodes is considered NP-hard.

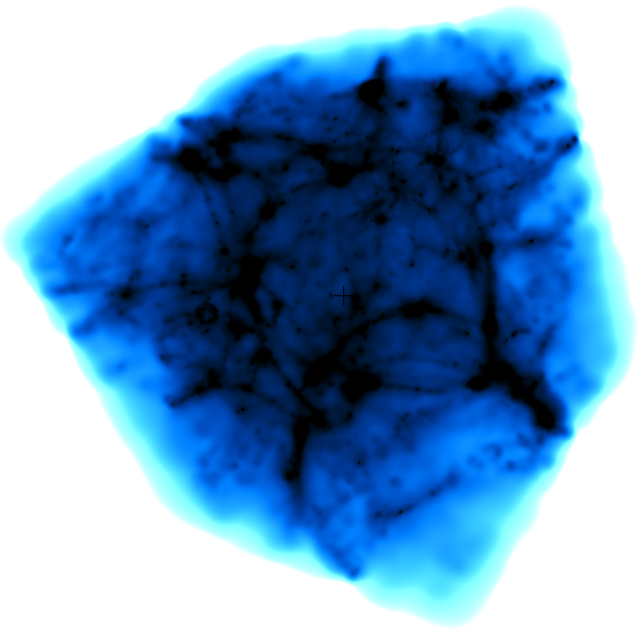

Figure 3: The initial density field computed from the initial particle distribution used for our tests. The density ρ_i of the particles spans 8 orders of magnitude, requiring smoothing lengths h_i changing by a factor of almost 1000 across the simulation volume.

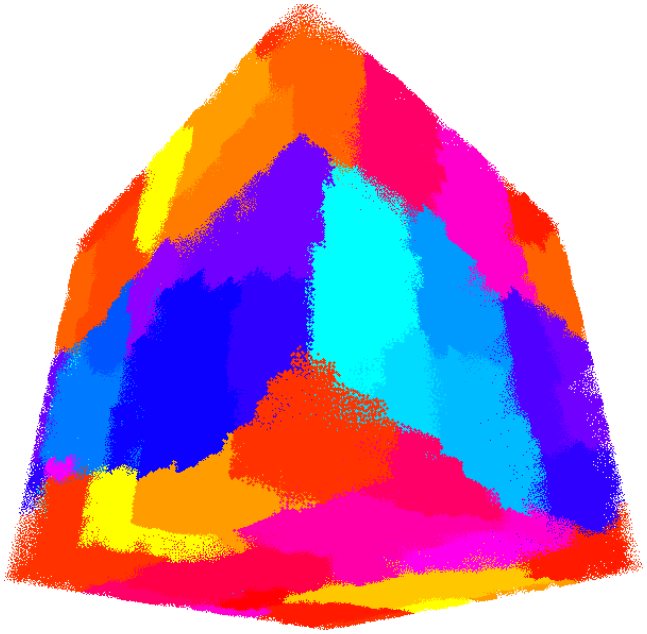

Figure 4: The particles for the initial conditions shown on Fig. 3 colored according to the node they belong to after a load-balancing call on 32 nodes. As can be seen, the domain decomposition follows the cells in the mesh but is not made of regular cuts. Domains have different shapes and sizes.

In order to provide a realistic setup, the initial particle distributions used in our tests were extracted by resampling low-redshift outputs of the EAGLE project [27], a large suite of state-of-the-art cosmological simulations. By selecting outputs at late times (redshift $z = 0.5$), we constructed a simulation setup which is representative of the most expensive part of these simulations, i.e. when the particles are highly-clustered and no longer uniformly distributed. This distribution of particles is shown on Fig. 3. In order to fit our simulation setup into the limited memory of some of the systems tested, we have randomly down-sampled the particle count of the output to $800^3 = 5.12 \times 10^8$, $600^3 = 2.16 \times 10^8$ and $376^3 = 5.1 \times 10^7$ particles with periodic boundary conditions respectively. Scripts to generate these initial conditions are provided with the source code. We then run the SWIFT code for 100 time-steps and average the wall clock time of these time-steps after having removed the first and last ones, where disk I/O occurs.

The initial load balancing between nodes is computed in the first steps and re-computed every 100 time steps, and is therefore not included in the timings. Particles were exchanged between nodes whenever they strayed too far beyond the cells in which they originally resided.

On all the machines, the code was compiled out of the box, without any tuning, explicit vectorization, or exploiting any other specific features of the underlying hardware.

4.2 x86 architecture: COSMA-5

For our first test, we ran SWIFT on the COSMA-5 DiRAC2 Data Centric System[3] located at the University of Durham. The system consists of 420 nodes with 2 Intel Sandy Bridge-EP Xeon E5-2670[4] CPUs clocked at 2.6 GHz with each 128 GByte of RAM. The 16-core nodes are connected using a Mellanox FDR10 Infiniband 2:1 blocking configuration.

This system is similar to many Tier-2 systems available in most universities or computing facilities. Demonstrating strong scaling on such a machine is essential to show that the code can be efficiently used even on the type of commodity hardware available to most researchers.

The code was compiled with the Intel compiler version $2016.0.1$ and linked to the Intel MPI library version $5.1.2.150$ and METIS library version $5.1.0$.

The simulation setup with 376^3 particles was run on that system using 1 to 256 threads on 1 to 16 nodes, the results of which are shown on Fig. 5. For this strong scaling test, we used one MPI rank per node and 16 threads per node (i.e. one thread per physical core). We also ran on one single node using up to 32 threads, i.e. up to one thread per physical and virtual core. Fig. 4 shows the domain decomposition obtained via the task-graph decomposition algorithm described above for the 16 node run.

4.3 x86 architecture: SuperMUC

For our next test, we ran SWIFT on the SuperMUC x86 phase 1 thin nodes [5] located at the Leibniz Supercomputing Center in Garching near Munich, currently ranked 23rd in the Top500 list[6]. This system consists of 9 216 nodes with 2 Intel Sandy Bridge-EP Xeon E5-2680 8C[7] at 2.7 GHz CPUS. Each 16-core node has 32 GByte of RAM. The nodes are split in 18 "islands" of 512 nodes within

[3] icc.dur.ac.uk/index.php?content=Computing/Cosma

[4] http://ark.intel.com/products/64595/Intel-Xeon-Processor-E5-2670-20M-Cache-2_60-GHz-8_00-GTs-Intel-QPI

[5] https://www.lrz.de/services/compute/supermuc/systemdescription/

[6] http://www.top500.org/list/2015/11/

[7] http://ark.intel.com/products/64583/Intel-Xeon-Processor-E5-2680-(20M-Cache-2_70-GHz-8_00-GTs-Intel-QPI)

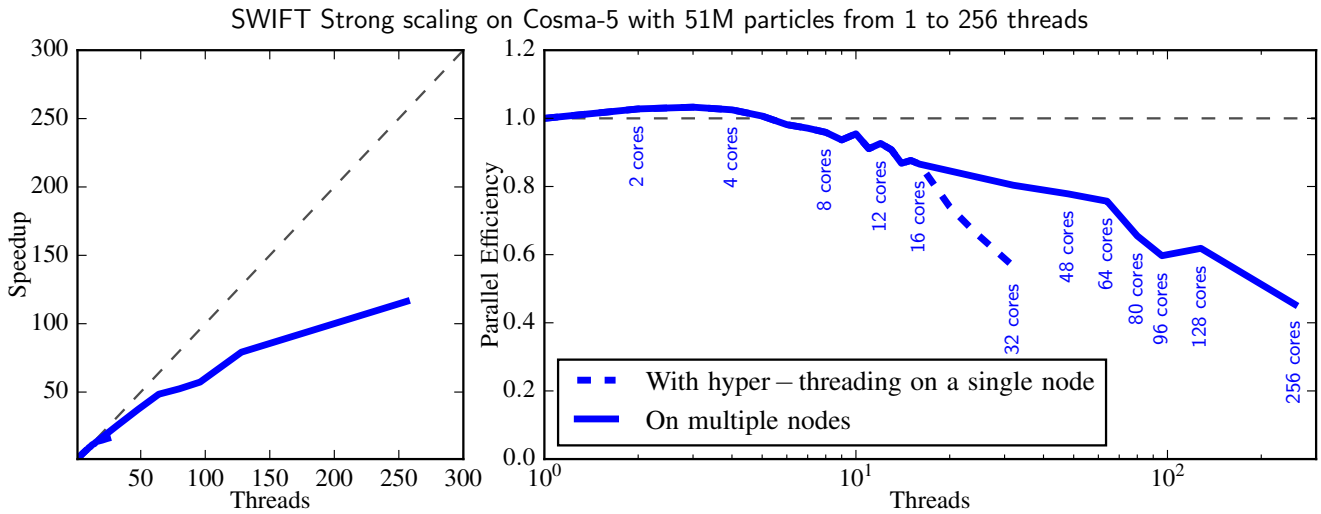

Figure 5: Strong scaling test on the COSMA-5 machine (see text for hardware description). *Left panel:* Code Speed-up. *Right panel:* Corresponding parallel efficiency. Using 16 threads per node (no use of hyper-threading) with one MPI rank per node, a good parallel efficiency (60%) is achieved when increasing the thread count from 1 (1 node) to 128 (8 nodes) even on this relatively small test case. The dashed line indicates the efficiency when running on one single node but using all the physical and virtual cores (hyper-threading). As these CPUs only have one FPU per core, the benefit from hyper-threading is limited to a 20% improvement when going from 16 cores to 32 hyperthreads.

which communications are handled via an Infiniband FDR10 non-blocking Tree. These islands are then connected using a 4:1 Pruned Tree.

This system is similar in nature to the COSMA-5 system used in the previous set of tests but is much larger, allowing us to test scalability at a much larger scale.

The code was compiled with the Intel compiler version 2015.5.223 and linked to the Intel MPI library version 5.1.2.150 and METIS library version 5.0.2.

The simulation setup with 800^3 particles was run using 16 to 2048 nodes (4 islands) and the results of this strong scaling test are shown in Fig. 6. For this test, we used one MPI rank per node and 16 threads per node, i.e. one thread per physical core.

4.4 Blue Gene architecture: JUQUEEN

For our last set of tests, we ran SWIFT on the JUQUEEN IBM Blue Gene/Q system[8] located at the Jülich Supercomputing Center, currently ranked 11th in the Top500 list. This system consists of 28 672 nodes with an IBM PowerPC A2 processor running at 1.6 GHz and 16 GByte of RAM each. Of notable interest is the presence of two floating units per compute core. The system is composed of 28 racks containing each 1 024 nodes. The network uses a 5D torus to link all the racks.

This system is larger than the SuperMUC supercomputer described above and uses a completely different processor and instruction set. We use it here to demonstrate that our results are not dependant on the hardware being used.

The code was compiled with the IBM XL compiler version 30.73.0.13 and linked to the corresponding MPI and METIS library versions 4.0.2.

The simulation setup with 600^3 particles was first run using 512 nodes with one MPI rank per node and varying only the number of threads per node. The results of this test are shown in Fig. 7.

[8]http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUQUEEN/Configuration/Configuration_node.html

We later repeated the test, this time varying the number of nodes from 32 to 8192 (8 racks). For this test, we used one MPI rank per node and 32 threads per node, i.e. two threads per physical core. The results of this strong scaling test are shown in Fig. 8.

5. DISCUSSION & CONCLUSIONS

The strong scaling results presented in the previous sections on three different machines demonstrate the ability of our framework to scale on both small commodity machines, thanks to the use of task-based parallelism at the node level, and on the largest machines (Tier-0 systems) currently available, thanks to the task-based domain distribution and asynchronous communication schemes. We would like to emphasize that these results were obtained for a realistic test case without any micro-level optimization or explicit vectorization.

Excellent strong scaling is also achieved when increasing the number of threads per node (i.e. per MPI rank, see fig. 7), demonstrating that the description of MPI (asynchronous) communications as tasks within our framework is not a bottleneck. One common conception in HPC is that the number of MPI communications between nodes should be kept to a minimum to optimize the efficiency of the calculation. Our approach does exactly the opposite with large number of point-to-point communications between pairs of nodes occurring over the course of a time-step. For example, on the SuperMUC machine with 32 nodes (512 cores), each MPI rank contains approximately 1.6×10^7 particles in 2.5×10^5 cells. SWIFT will generate around 58 000 point-to-point asynchronous MPI communications (a pair of `send` and `recv` tasks) *per node* and *per timestep*. Each one of these communications involves, on average, no more than 6 kB of data. Such an insane number of small messages is discouraged by most practitioners, but seems to works well in practice. Dispatching communications over the course of the calculation and not in short bursts, as is commonly done, may also help lower the load on the network.

One time-step on 8 192 nodes of the JUQUEEN machine takes

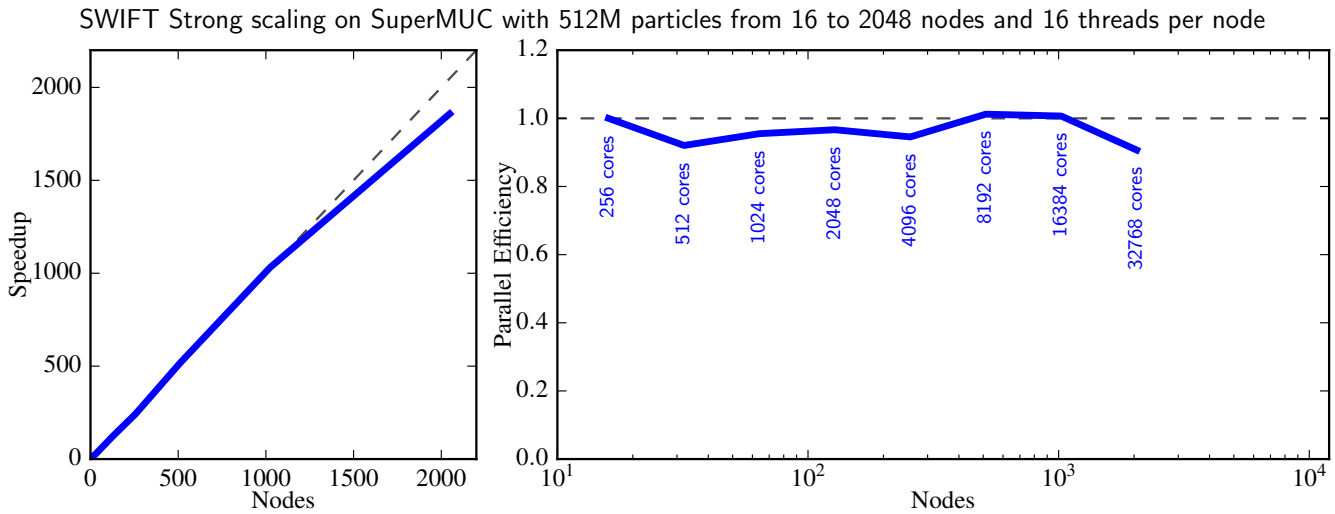

Figure 6: Strong scaling test on the SuperMUC phase 1 machine (see text for hardware description). *Left panel:* Code Speed-up. *Right panel:* Corresponding parallel efficiency. Using 16 threads per node (no use of hyper-threading) with one MPI rank per node, an almost perfect parallel efficiency is achieved when increasing the node count from 16 (256 cores) to 2 048 (32 768 cores).

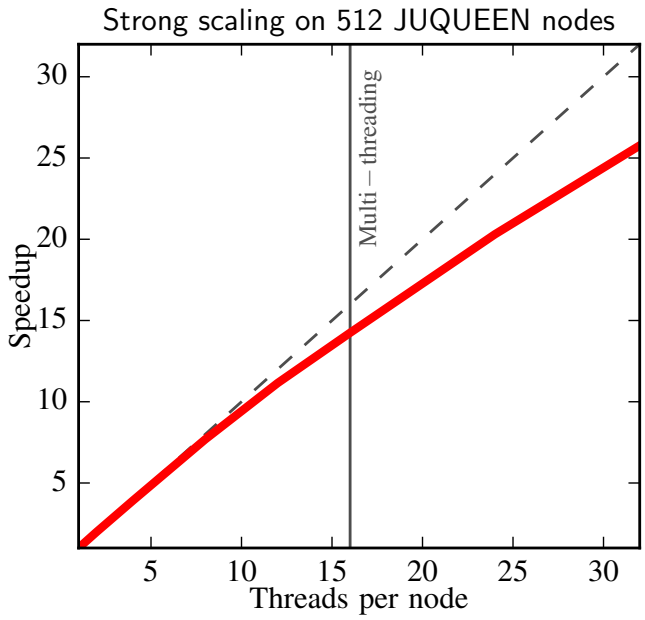

Figure 7: Strong scaling test of the hybrid component of the code. The same calculation is performed on 512 node of the JUQUEEN Blue Gene supercomputer (see text for hardware description) using a single MPI rank per node and varying only the number of threads per node. The code displays excellent scaling even when all the cores and hardware multi-threads are in use.

63 ms of wall-clock time. All the loading of the tasks, communications and running of the tasks takes place in that short amount of time. Our framework can therefore load-balance a calculation over 2.6×10^5 threads with remarkable efficiency.

We emphasize, as was previously demonstrated in [17], that SWIFT is also much faster than the GADGET-2 code [29], the *de-facto* standard in the field of particle-based cosmological simulations. The simulation setup that was run on the COSMA-5 system takes 2.9 s of wall-clock time per time-step on 256 cores using SWIFT whilst the default GADGET-2 code on exactly the same setup with the same number of cores requires 32 s. The excellent scaling performance of SWIFT allows us to push this number further by simply increasing the number of cores, whilst GADGET-2 reaches its peak speed (for this problem) at around 300 cores and stops scaling beyond that. This unprecedented scaling ability combined with future work on vectorization of the calculations within each task will hopefully make SWIFT an important tool for future simulations in cosmology and help push the entire field to a new level.

These results, which are in no way restricted to astrophysical simulations, provide a compelling argument for moving away from the traditional branch-and-bound paradigm of both shared and distributed memory programming using synchronous MPI and OpenMP. Although fully asynchronous methods, due to their somewhat anarchic nature, may seem more difficult to control, they are conceptually simple and easy to implement[9]. The real cost of using a task-based approach comes from having to rethink the entire computation to fit the task-based setting. This may, as in our specific case, lead to completely rethinking the underlying algorithms and reimplementing a code from scratch. Given our results to date, however, we do not believe we will regret this investment.

SWIFT, its documentation, and the test cases presented in this paper are all available at the address www.swiftsim.com.

6. ACKNOWLEDGEMENTS

This work would not have been possible without Lydia Heck's

[9]The SWIFT source code consists of less than 21 K lines of code, of which roughly only one tenth are needed to implement the task-based parallelism and asynchronous communication.

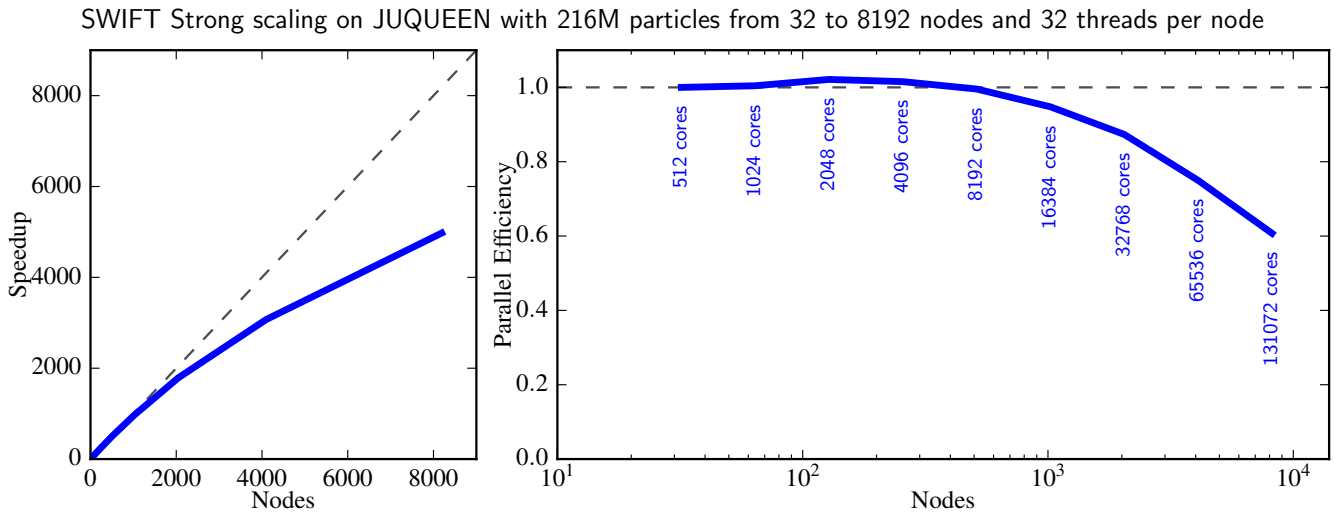

Figure 8: Strong scaling test on the JUQUEEN Blue Gene machine (see text for hardware description). *Left panel:* Code Speed-up. *Right panel:* Corresponding parallel efficiency. Using 32 threads per node (2 per physical core) with one MPI rank per node, a parallel efficiency of more than 60% is achieved when increasing the node count from 32 (512 cores) to 8 192 (131 072 cores). On 8 192 nodes there are fewer than 27 000 particles per node and only a few hundred tasks, making the whole problem extremely hard to load-balance effectively.

help and expertise. We acknowledge the help of Tom Theuns, James Willis, Bert Vandenbroucke, Angus Lepper and other contributors to the SWIFT code.
We thank Heinrich Bockhorst and Stephen Blair-Chappell from INTEL as well as Dirk Brommel from the Jülich Computing Center and Nikolay J. Hammer from the Leibniz Rechnenzentrum for their help at various stages of this project.
This work used the DiRAC Data Centric system at Durham University, operated by the Institute for Computational Cosmology on behalf of the STFC DiRAC HPC Facility (www.dirac.ac.uk). This equipment was funded by BIS National E-infrastructure capital grant ST/K00042X/1, STFC capital grant ST/H008519/1, and STFC DiRAC Operations grant ST/K003267/1 and Durham University. DiRAC is part of the National E-Infrastructure. This work was supported by the Science and Technology Facilities Council ST/F001166/1 and the European Research Council under the European Union's ERC Grant agreements 267291 "Cosmiway", and by INTEL through establishment of the ICC as an INTEL parallel computing centre (IPCC).

7. REFERENCES

[1] *SMP Superscalar (SMPSs) User's Manual, Barcelona Supercomputing Center*, 2008.

[2] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. In *Journal of Physics: Conference Series*, volume 180, page 012037. IOP Publishing, 2009.

[3] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, 2011.

[4] W. Bangerth, R. Hartmann, and G. Kanschat. deal.II – a general purpose object oriented finite element library. *ACM Trans. Math. Softw.*, 33(4):24/1–24/27, 2007.

[5] J. Barnes and P. Hut. A hierarchical o (n log n) force-calculation algorithm. *Nature*, 1986.

[6] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[7] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. *Cilk: An efficient multithreaded runtime system*, volume 30. ACM, 1995.

[8] J. Carrier, L. Greengard, and V. Rokhlin. A fast adaptive multipole algorithm for particle simulations. *SIAM Journal on Scientific and Statistical Computing*, 9(4):669–686, 1988.

[9] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[10] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan. Zoltan data management services for parallel dynamic applications. *Computing in Science & Engineering*, 4(2):90–96, 2002.

[11] A. Duran, R. Ferrer, E. Ayguadé, R. M. Badia, and J. Labarta. A proposal to extend the OpenMP tasking model with dependent tasks. *International Journal of Parallel Programming*, 37(3):292–305, 2009.

[12] R. A. Gingold and J. J. Monaghan. Smoothed particle hydrodynamics-theory and application to non-spherical stars. *Monthly notices of the royal astronomical society*, 181:375–389, 1977.

[13] P. Gonnet. Efficient and scalable algorithms for smoothed particle hydrodynamics on hybrid shared/distributed-memory architectures. *SIAM J. Scientific Computing*, 37(1), 2015.

[14] P. Gonnet, A. B. G. Chalk, and M. Schaller. QuickSched: Task-based parallelism with dependencies and conflicts. *ArXiv e-prints*, Jan. 2016.

[15] P. Gonnet, M. Schaller, T. Theuns, and A. B. Chalk. Swift: Fast algorithms for multi-resolution sph on multi-core architectures. In *8th International SPHERIC Workshop*. Trondheim, Norway, 2013.

[16] L. Hernquist and N. Katz. TREESPH-A unification of SPH with the hierarchical tree method. *The Astrophysical Journal Supplement Series*, 70:419–446, 1989.

[17] L. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[18] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[19] D. Meagher. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, 19(2):129–147, 1982.

[20] D. J. Price. Smoothed particle hydrodynamics and magnetohydrodynamics. *Journal of Computational Physics*, 231:759–794, Feb. 2012.

[21] J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Incorporated, 2007.

[22] J. Schaye, R. A. Crain, R. G. Bower, M. Furlong, M. Schaller, T. Theuns, C. Dalla Vecchia, C. S. Frenk, I. G. McCarthy, J. C. Helly, A. Jenkins, Y. M. Rosas-Guevara, S. D. M. White, M. Baes, C. M. Booth, P. Camps, J. F. Navarro, Y. Qu, A. Rahmati, T. Sawala, P. A. Thomas, and J. Trayford. The EAGLE project: simulating the evolution and assembly of galaxies and their environments. *MNRAS*, 446:521–554, Jan. 2015.

[23] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference (Vol. 1): Volume 1-The MPI Core*, volume 1. MIT press, 1998.

[24] V. Springel. The cosmological simulation code GADGET-2. *MNRAS*, 364:1105–1134, Dec. 2005.

[25] T. Theuns, A. Chalk, M. Schaller, and P. Gonnet. Swift: task-based hydrodynamics and gravity for cosmological simulations. In *Proceedings of the 3rd International Conference on Exascale Applications and Software*, pages 98–102. University of Edinburgh, 2015.

[26] J. Wadsley, J. Stadel, and T. Quinn. Gasoline: a flexible, parallel implementation of TreeSPH. *New Astronomy*, 9(2):137–158, 2004.

[27] M. S. Warren and J. K. Salmon. A parallel hashed oct-tree n-body algorithm. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 12–21. ACM, 1993.

[28] A. YarKhan, J. Kurzak, and J. Dongarra. *QUARK Users' Guide*. Electrical Engineering and Computer Science, Innovative Computing Laboratory, University of Tennessee, April 2011.