Condensing the cloud: running CIEL on many-core

Malte Schwarzkopf Derek G. Murray Steven Hand
University of Cambridge Computer Laboratory
15 JJ Thomson Avenue
Cambridge, United Kingdom
{firstname.lastname}@cl.cam.ac.uk

1. INTRODUCTION

Distributed execution engines have revolutionised data processing by making parallel programming simple. Systems such as MapReduce [10], Dryad [13] and Hadoop [1] can achieve massive throughput when running on thousands of commodity servers, yet generally only require the programmer to provide sequential code. These systems were designed to scale out across many worker machines, each of which had at most a handful of processors. As such, *intra-server parallelism* is either left entirely to the developer [13, 17], or managed centrally by partitioning machines into a small number of coarse-grained "slots" [1].

We are interested in how best to marry distributed execution engines with future multi-core systems. In particular, we ask *(a)* Are execution engines suitable for coordinating execution on a single many-core machine? and *(b)* What is the appropriate treatment of multi-core workers in a distributed cluster? Baumann *et al.* have pointed out that modern many-core architectures already share many characteristics with distributed systems [6]. Indeed, as core counts grow, it becomes inefficient to maintain cache-coherent shared memory; like in the Barrelfish multikernel architecture, distributed execution engines are "distributed by default". Our ultimate aim is to devise a system that achieves both inter- and intra-server parallelism: a *multi-scale* execution engine for executing parallel programs across any combination of machines and cores.

In this paper we make a preliminary investigation into the first of the above questions. Using our recently-developed CIEL distributed execution engine [17], we compare the performance of simple benchmarks on three quite different 48-core platforms: an AMD "Magny-Cours" ccNUMA server, an experimental Intel Single-Chip Cloud computer, and an Amazon EC2 cluster of 48 uniprocessor VMs (§3). Our results demonstrate task creation/coordination overhead to be a problem when using fine-grained tasks, and we demonstrate a few simple improvements that mitigate these issues. We also outline a set of further challenges and opportunities (§4), before discussing related work and concluding.

2. ARCHITECTURE COMPARISON

At present, the most commonly used platform for highly parallel processing is a cluster of machines (either physical or virtual). These machines typically have only a handful of cores; in the virtual machine case, a single virtual core is typically considered best practice. In both physical and virtual machine clusters, however, the large number of machines necessitates data transfer over the network between dependent tasks, although data-local task scheduling can mitigate this problem somewhat. This raises the question of whether we can do better by using many-core machines with access to local data, especially if we are going to see a continued increase in the number of cores per machine.

In this section, we compare the technical specifications and processor designs of Intel's experimental Single-Chip Cloud and the AMD "Magny-Cours" architecture. We briefly touch upon the key differences between parallel execution interfaces in many-core machines and we also discuss the potential for "hybrid" clusters of multiple many-core machines.

2.1 Intel Single-Chip Cloud (SCC)

The Intel Single-Chip Cloud Computer experimental processor is a 48-core "concept vehicle", created as a platform for many-core software research. It makes extensive use of a network-on-chip for message passing between the cores, each of which is based on the P54C design used by early Pentium processors. There are a few differences: the L1 cache is larger, a new instruction for explicit cache invalidation and a special memory type for message passing have been added [11]. The cores are 32-bit only, run at between 100 and 800 MHz, and do not support out-of-order execution. A pair of cores shares a *tile*, which also holds two L2 caches, a share of the message passing buffer (MPB) and a router that connects it to other tiles using a wormhole-switched 2D mesh network; these on-chip network routers are clocked at either 800 MHz or 1.6 GHz. The message latency between adjacent tiles is only 4 clock cycles in the no-load case.

Unlike the Intel Terascale Research Prototype chip [15, 21], the SCC cores are full-featured general-purpose processors, and so we can easily run an individual instance of Linux on each core. The cores communicate via either the on-die shared-memory MPB, with each core having access to an 8 KB share of the total 384 KB; or via shared memory. Since the cores use 32-bit virtual addressing, they can only access up to 4 GB of memory; larger memory sizes (up to 64 GB) can by partitioned between the cores, with each core by default receiving 1 GB [4].

The message passing hardware enables software-managed cache coherency, replacing the familiar bus snooping of traditional SMP configurations. Instead, RCCE, a purpose-built message passing library that uses the SCC's MPB, is provided and can be used to achieve the required coherence [20].

Hence when running Ciel on the SCC, each worker runs within the fault domain of a separate OS instance. Thus—even though all cores are, of course, similarly affected by global events such as power failures—OS-level failures only affect a single worker, as in the distributed case. This contrasts with the more traditional system described next.

2.2 AMD Opteron Magny-Cours (AMD-MC)

AMD's "Magny-Cours" processor takes a more conventional approach to building a many-core system: each die holds up to six cores, with two dies per socket, enabling up to 48 cores with four sockets [8]. Communication between the cores, both within a socket and between different sockets, is realised using AMD's HyperTransport bus protocol [12]. Each socket has two DDR memory channels; the system used in this paper is configured with 64 GB of RAM (16 GB per socket).

As the system is 64-bit, every core can access all DRAM (unlike on the SCC), although of course with non-uniform access latencies. In addition to 64 KB L1 instruction and data caches, each core has a private L2 cache (512 KB). There is also a 6 MB L3 cache which is shared between all six cores on a die; cache coherency within and across sockets is maintained using a directory protocol called "HyperTransport Assist" (HT Assist). This makes use of a "Probe Filter", which determines whether the memory controller generates a broadcast probe, a directed probe, or no probe at all. Coherency control is managed totally in hardware, although a NUMA-aware operating system can optimize the way in which it performs memory allocation (e.g. via Linux's `libnuma` or the `numactl` utility).

2.3 Hybrid clusters

A single many-core machine may not suffice to run all required cluster workloads, either because the number of workers available is limited by the number of cores in the machine, or because eventually I/O bandwidth becomes a limiting factor. For these reasons, we believe that in the future we will see *hybrid clusters* that comprise several many-core machines—or even a heterogeneous mix of multi- and many-core machines.

In the following, we investigate the operation of an existing execution engine, Ciel [17], on different many-core architectures. From the results, we distill a set of challenges and propose potential approaches to improve performance of execution engines in a multi-scale setting such as the clusters of multi- and many-core machines described here.

3. CIEL ON MANY-CORE

The primary aim of this paper is to investigate the performance of Ciel, a system designed for clusters of commodity machines, when running on the architectures described in the previous section. In this section, we first describe the configuration of Ciel on a many-core machine (§3.1), and then, using a series of microbenchmarks, analyse the overhead of fine-grained task creation (§3.2) and message-passing latency (§3.3). Finally, we demonstrate that an existing application developed for distributed clusters can achieve good performance on a many-core machine (§3.4).

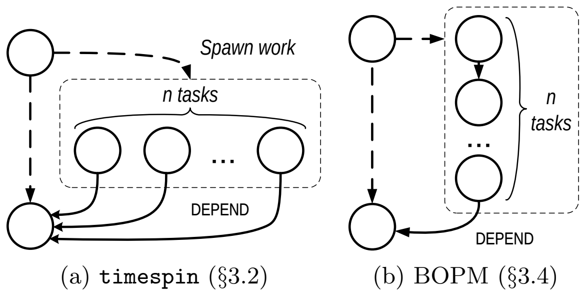


Figure 1: Ciel job topologies for the benchmarks in §3.

3.1 CIEL architecture

Ciel uses a master-worker architecture similar to that used by previous systems (such as MapReduce and Dryad). In a Ciel cluster, the *master* is responsible for coordinating and dispatching *tasks* (the atomic unit of computation) to *workers*. Workers execute tasks, and also store *objects*, which may be the inputs to or outputs from tasks. Data is exchanged directly between workers, by making remote requests to other workers' object stores.

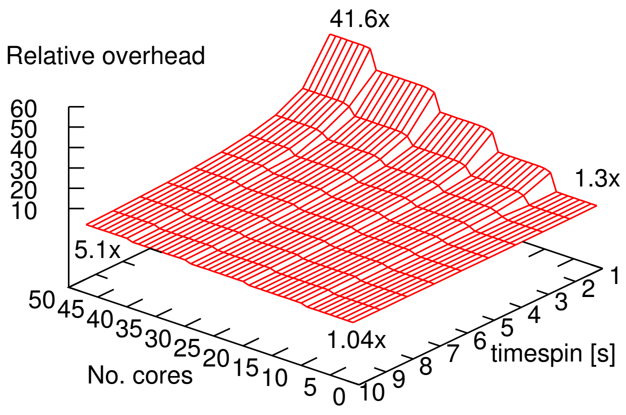
A Ciel job is represented by a *dynamic task graph*. As in Dryad, the task graph is a DAG of tasks, where edges comprise a happens-before relation on tasks, and also imply data-flow between tasks. However, in Ciel, the graph is *dynamic*, which means that a task may *spawn* child tasks, and *delegate* the production of its outputs to its children. This enables a Ciel job to contain data-dependent control flow, which allows it to express iterative and recursive algorithms. To simplify the construction of a dynamic dependency graph, the Skywriting scripting language allows programmers to specify a job using familiar imperative or functional control-flow constructs [16].

For the following experiments, the Ciel master runs on a single core (although it may use multiple threads for I/O concurrency), and the workers—and hence tasks—execute on the remaining cores.

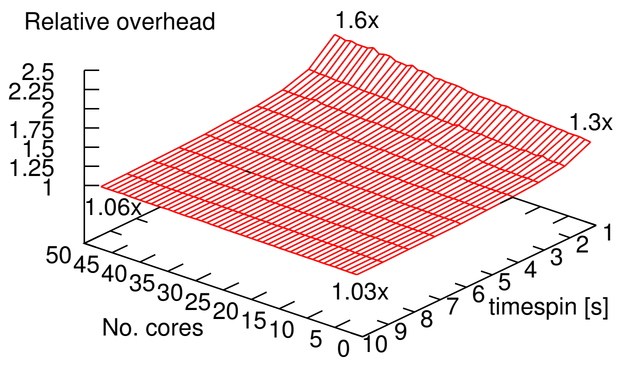
3.2 Task granularity

Distributed execution engines are usually evaluated on coarse-grained workloads, where the amount of work performed by individual task is sufficiently great to dominate any overheads of task creation and synchronisation. For example, MapReduce uses a default task input size of 64 MB, which corresponds to a single block in the underlying distributed file system [10]. However, while the communication overhead in a cluster encourages the use of relatively large tasks, the tightly-coupled environment of a many-core machine might allow the use of finer grained tasks. In this experiment, we evaluate the performance of Ciel on a synthetic benchmark we have built which allows us to control the task granularity. At the finest granularity, the overheads imposed by the execution engine are exposed, and we can identify opportunities for improvement specifically for a many-core environment.

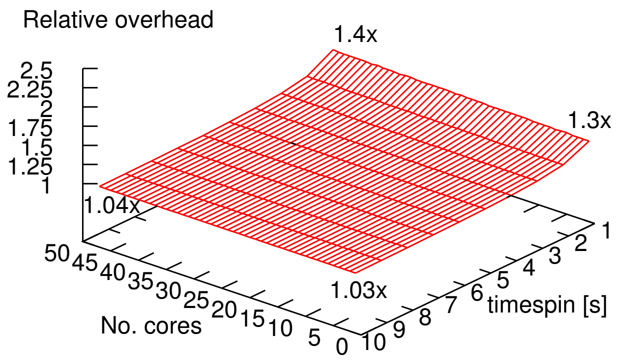
Our `timespin` benchmark spawns one task per core, each of which spins for a controllable period of time t, then combines the task results in a single synchronization task (Fig-



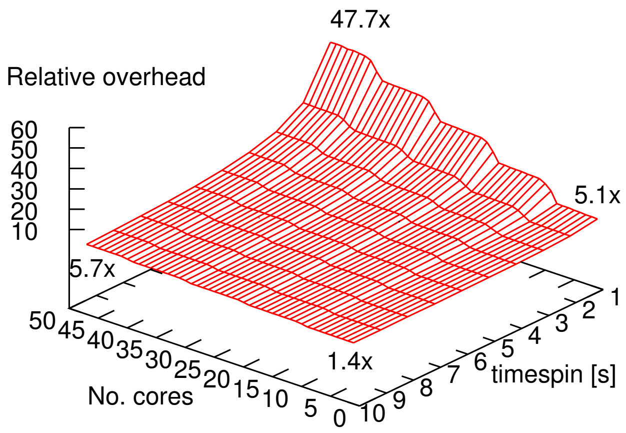
(a) AMD-MC, using separate object stores.



(b) AMD-MC, shared object store.



(c) AMD-MC, using local IPC mode.



(d) On the SCC.

Figure 2: Relative overhead of `timespin`. All values are medians of five runs.

ure 1(a)). The makespan of a `timespin` run will therefore be $\simeq t + \epsilon$, where ϵ represents the overhead introduced by CIEL. In the following, we use the term *relative overhead* to refer to the quantity $(t+\epsilon)/t$: ideally this will be close to 1.

In the first experiment, we configure the AMD-MC machine as a cluster comprising a single master and 47 independent workers, running on top of a 64-bit installation of Debian 6.0. Figure 2(a) shows the effect of reducing the granularity of tasks (reducing spin duration) and increasing the parallelism (number of cores). In the worst case—spinning for one second on 47 cores—the relative overhead is 42×, which is almost as slow as executing the tasks serially!

A major source of overhead is the fact that the workers must remotely read their arguments, which are stored in CIEL's object store, from the initial (root task) worker. The current version of CIEL uses a multithreaded HTTP server to serve the arguments to each task. Therefore there is overhead from serialising and transmitting the arguments over a loopback TCP/IP connection, and from contention on the HTTP server; this server only admits ten connections at once, resulting in noticeable "steps" being present in Figure 2(a).

The first optimisation that we considered was sharing the object store between all workers on a single host. Figure 2(b) shows the performance of the `timespin` benchmark when the object store is shared. The worst-case overhead is drastically reduced from 42× to 1.6×.

We next focused on the task dispatch protocol, which still uses HTTP over loopback TCP and so presented another potential optimisation. We developed a version of CIEL, which uses local IPC, and hence incorporates the master and worker logic in a single application. This application has one process per core, and these processes communicate using OS-level pipes and semaphores. Figure 2(c) shows the performance of the `timespin` benchmark on this optimised version of CIEL. The worst-case overhead is reduced further from 1.6× to 1.4×, a relative further improvement of around 13%. In Figure 3 we focus on the most fine-grained case. The result shows that CIEL in local IPC mode scales relatively well as the number of cores increases, suggesting that the next promising area to target will be the fixed cost overheads introduced even on a single core.

3.3 Message passing performance

We repeated the `timespin` benchmark on the Intel SCC, running 48 instances of Linux (Figure 2(d)). However, since the SCC cores run completely independent instances of Intel's SCC Linux, the performance is roughly as bad as when using independent object stores on the AMD-MC machine (Figure 2(a)). A major cause of the overhead is the fact that the Linux instances on the SCC communicate by making HTTP requests to each other using the special `rckmb`

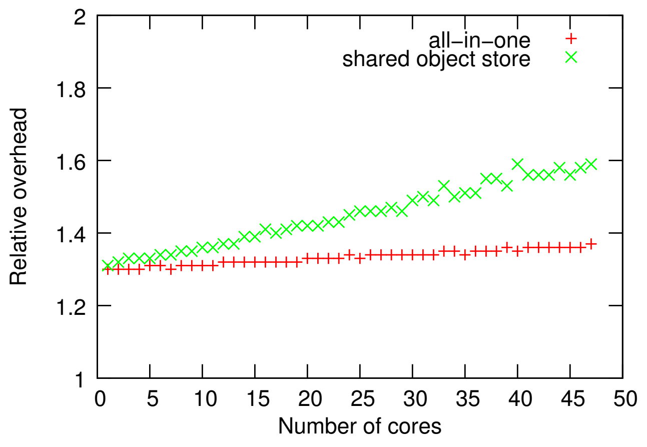


Figure 3: Detailed comparison of the 1s case of `timespin` using a shared object store and all-in-one mode.

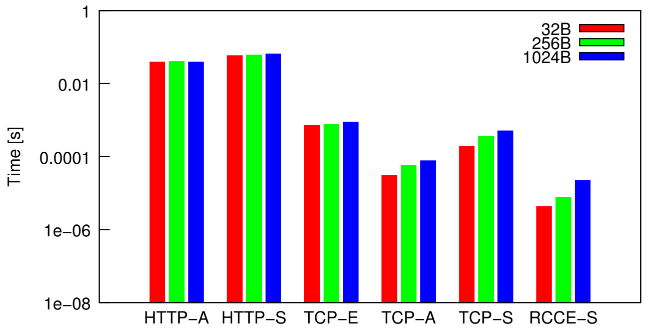


Figure 4: Message latency for various protocols (HTTP, TCP and RCCE) and platforms (**A**MD-MC, **S**CC, **E**C2). Note the log-scale y-axis.

network driver that implements TCP/IP over the SCC's on-chip message-passing buffers. To further investigate this, we compared the performance of different message-passing transports to identify opportunities for improving the overhead in fine-grained CIEL jobs. We measured the time to transmit small (32 bytes), medium (256 bytes) and large (1 KB) messages over HTTP and raw TCP sockets, as well as using RCCE on the SCC.

Figure 4 shows the cost of various message-passing techniques on the many-core machines as well as in the distributed cluster. The worst case on both the SCC and the AMD-MC machine is HTTP access to the pre-existing CIEL object store, which uses the abovementioned multithreaded HTTP server (written in Python) to provide remote access[1]. For small messages—such as a CIEL task descriptor, which is usually around 256 bytes—raw TCP achieves almost two orders of magnitude improvement, and using native RCCE message passing on the SCC ("non-gory" API) improves by another order of magnitude. At larger message sizes, the cost of using RCCE approaches that of loopback TCP on the AMD machine, while on the SCC—where TCP is implemented over the `rckmb` network driver—a performance gap remains. We believe this is since RCCE uses the small

[1]The performance of HTTP requests between machines in the distributed cluster on EC2 was comparable to the data series shown and is omitted owing to a lack of space.

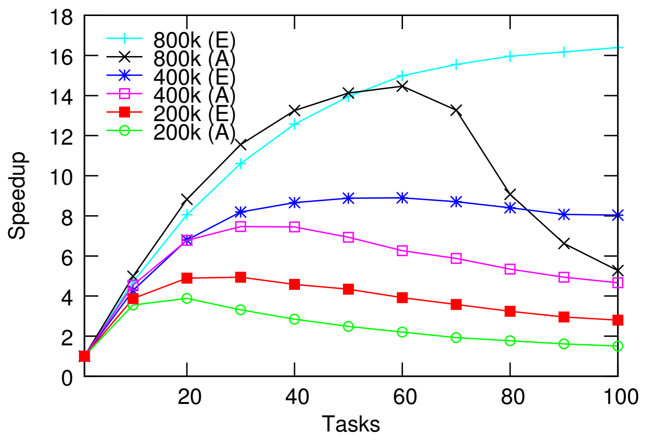


Figure 5: Speedup comparison of the BOPM benchmark running distributed on **E**C2 and on the **A**MD-MC machine.

8 KB per-die message passing buffers on the SCC, whereas TCP can make use of larger buffers on the AMD-MC machine (and in the distributed case).

3.4 Binomial options pricing

We now consider a compute-intensive application that can be parallelized using CIEL. The binomial options pricing model (BOPM) is a dynamic programming algorithm for numerically computing the expected value of a stock option, based on a model of market volatility, long-range trends, and the strike and current spot prices [9]. BOPM is a CPU-intensive algorithm that has $O(n^2)$ running time, where n is the number of time steps (*i.e.* the resolution). Figure 1(b) shows the task dependency graph. In the dynamic programming matrix, the element at position (i, j) depends on elements at $(i + 1, j)$ and $(i + 1, j + 1)$, which means that it exhibits pipelined parallelism.

We compared the performance that CIEL achieves when running BOPM on the 48-core AMD-MC machine, and a cluster of 48 EC2 `m1.small` virtual machines running Ubuntu 10.04, for various resolutions. Due to the faster clock speed, the AMD-MC machine achieves better absolute performance in all configurations. As a more meaningful comparison, Figure 5 compares the parallel speedup over a serial execution. In both cases, the speedup initially increases with the number of tasks, because the maximum degree of parallelism becomes greater. However, increasing the number of tasks achieves diminishing returns, and eventually the overhead of task creation begins to dominate and reduces the parallel speedup.

4. CHALLENGES AND OPPORTUNITIES

The results of the previous section point to several challenges—and opportunities for optimisation—when running CIEL on a many-core architecture. In this section, we discuss these challenges, and draw examples from the literature that could improve performance of CIEL.

4.1 Reducing contention

When tasks become very fine-grained, the overhead from contention begins to dominate. This is particularly noticeable in Figures 2(a) and 2(d), where the server that serves task inputs is heavily contended, leading to a pronounced step in the graph where the number of cores exceeds the number of server threads. A simple approach would be to replace the server component with an event-based server that uses asynchronous I/O [22].

However, as tasks grow even more fine grained, individual data structures in the master will become contended. For example, the master maintains a task graph, and at present all accesses to the task graph are serialised in a single thread. This strategy may be overly conservative: since many updates to the task graph affect disjoint regions, we could reduce contention by implementing fine-grained locking, and/or moving to an optimistic update scheme using transactions.

4.2 Efficient message-passing

The results in §3.3 highlight the opportunities for exploiting efficient message-passing mechanisms on the SCC processor. Replacing the heavyweight HTTP-based messaging in CIEL with messages based on RCCE would greatly reduce the minimum task execution time, and hence allow finer-grained tasks to run efficiently.

As the granularity decreases further, having all workers communicate with a central master—whether running on-chip, in the same machine, or remotely—will become a source of contention. Previous work on Barrelfish has shown that some collective operations can be made more efficient by exploiting the interconnect structure and building a multicast tree [5]. Given that CIEL is aware of the task dependency structure, we could exploit similar communication patterns in CIEL on many-core. A possible approach supporting this is to have each worker report to a local "coordinator" process (or core), which acts as a proxy between the local workers and the central master. However, on the SCC, RCCE allows only strictly blocking communication [20]—but in order to allow the coordinator to receive from any other core on the chip, a non-blocking receive functionality is required. The iRCCE [7] extension to RCCE provides non-blocking abstractions such as message queues and request handlers. We have started implementing the coordinator/worker model using iRCCE for the SCC, and using a local worker thread pool on the AMD-MC architecture. One challenge is that the current implementations of RCCE and iRCCE busy-wait for incoming messages. We are developing an interrupt-driven alternative alleviating this performance bottleneck.

4.3 Exploiting sharing

Figures 2(a) and 2(b) show how *sharing* the object store between workers can lead to a performance improvement. The benefits arise from the file system layer, since in the shared case, each object maps to a single file, which can often be read directly from the buffer cache. However, the buffer cache is a rather crude mechanism for performance optimisation, and we believe we should be able to do better by explicitly managing the objects ourselves.

For example, the BOPM application (§3.4) uses files to *stream* data between tasks. This approach is suited to distributed clusters, because it avoids deadlock when there are more runnable tasks than available workers. On a single many-core system, however, this streaming could be safely achieved by direct message passing between tasks, or even via a shared-memory ring buffer. More generally, instead of sending large messages (e.g. the bulk data passed between tasks), we could simply pass pointers to shared memory between tasks. This would, however, be especially challenging on the SCC, as the shared memory accessible from all cores is limited to 64 MB at present, although means to extend it have been suggested [14]. An ideal system would adapt the means used to share and communicate objects depending on the relative locations of the tasks involved, and the communication channels available.

4.4 Multiplexing input/output

All of the benchmarks considered in this paper are CPU-intensive, but transfer relatively little data. However, CIEL and other distributed execution engines are primarily designed for high-throughput processing of large amounts of data. Assuming for now that the workers use hard disks for storage, the most efficient data access pattern is long sequential reads. Naïvely running multiple I/O-bound tasks on the same machine will harm performance, because contention in the disk scheduler would lead to undesirable disk access patterns.

The opportunity for improvement comes from the fact that modern servers, in addition to multiple cores, have multiple disks. A current holder of the "Indy" sort benchmark record, TritonSort, achieves high throughput by explicitly partitioning disks into input disks and output disks [19]. However, TritonSort is a specialised application for sorting, and more general scheduling policies are required in order to get equivalent performance in an execution engine. For example, each many-core machine could support slots in multiple scheduling classes: d "I/O" slots (where d is proportional to the number of disks in the machine) and $n - d$ "compute" slots for CPU-bound tasks (where n is the number of cores available in the machine).

4.5 Intelligent fault tolerance

CIEL is currently rather conservative with respect to fault tolerance: all task outputs are written to disk to enable recovery after a software fault. However when tasks are co-located on a single machine, a larger degree of fate sharing occurs. As such it may be more sensible to avoid these writes, and perhaps move instead to a lightweight in-memory store. More generally, we believe that a multi-scale CIEL will need to take fault domains into account when assigning tasks to workers. For example, when proactively scheduling a back-up task, it will be prudent to assign this to a worker in a disjoint fault domain.

5. RELATED WORK

The problem of achieving high throughput on multiple processors has been extensively studied before, most notably in the High-Performance Computing (HPC) community. The *de facto* standard parallel programming models are OpenMP (for shared memory) and MPI (for distributed memory). OpenMP is mainly geared towards parallelising independent loop bodies, though recent versions have included task-parallel constructs [3]. However, the shared memory assumption ultimately limits the scalability of an OpenMP program. MPI is a lower-level programming model, in which processes explicitly send messages to one another. Because the message endpoints must be named explicitly, MPI does not easily support dynamic cluster membership, which is a typical requirement for a distributed execution engine (since workers may fail at any time).

Phoenix [18, 23] reimplements the MapReduce programming model for many-core machines. It makes extensive use of shared memory communication for moving data and threading to schedule tasks. However, Phoenix only works within on a single many-core machine and has no support for participation in a larger cluster. Notably, it lacks inter-machine message passing primitives, and furthermore, it offers no fault tolerance as all threads are within the same fault domain.

The idea of using an execution engine to simplify parallel programming on many-core machines is similar to the idea of *deterministic multiprocessing* (DMP). For example, Determinator is an operating system that enforces deterministic parallelism by restricting processes' ability to write to shared memory, and hence it avoids the problem of data races in parallel code [2]. Ciel also supports deterministic parallelism, but offers a richer programming model than Determinator because it supports *futures* in addition to strict fork-join parallelism. Furthermore, although it includes a distributed execution mode, Determinator does not provide Ciel's fault tolerance guarantees.

6. CONCLUSIONS

In this paper, we have investigated the performance of the Ciel distributed execution engine on a variety of many-core platforms. Our results show that an unmodified version of Ciel can run on a wide range of platforms, and that we can exploit platform-specific features to enhance performance. We are actively exploring some of the challenges outlined above, and hope to move on to develop a multi-scale version of Ciel which can coordinate execution across a hybrid cluster of cores and machines.

7. ACKNOWLEDGMENTS

We would like to thank Intel for giving us access to an SCC system, and the anonymous reviewers for their comments, which substantially enhanced this paper.

References

[1] Apache. Apache Hadoop `http://hadoop.apache.org`.

[2] Aviram, A., Weng, S.-C., Hu, S., and Ford, B. Efficient system-enforced deterministic parallelism. In *Proceedings of OSDI* (2010).

[3] Ayguadé, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Teruel, X., Unnikrishnan, P., and Zhang, G. The Design of OpenMP Tasks. *IEEE Trans. Parallel Distrib. Syst. 20*, 3 (2009), 404–418.

[4] Baron, M. The Single-Chip Cloud Computer. *Microprocessor Report 24*, 4 (2010).

[5] Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., and Singhania, A. The Multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of SOSP* (2009).

[6] Baumann, A., Peter, S., Schüpbach, A., Singhania, A., Roscoe, T., Barham, P., and Isaacs, R. Your computer is already a distributed system. why isn't your OS? In *Proceedings of HotOS* (2009).

[7] Clauss, C., Lankes, S., Galowicz, J., and Bemmerl, T. iRCCE: A Non-blocking Communication Extension to the RCCE Communication Library for the Intel Single-Chip Cloud Computer – User Manual. Tech. rep., Chair for Operating Systems, RWTH Aachen University, December 2010. Users' Guide and API Manual.

[8] Conway, P., Kalyanasundharam, N., Donley, G., Lepak, K., and Hughes, B. Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor. *IEEE Micro 30*, 2 (Mar. 2010), 16–29.

[9] Cox, J. C., Ross, S. A., and Rubinstein, M. Option pricing: A simplified approach. *Journal of Financial Economics 7*, 3 (1979), 229–263.

[10] Dean, J., and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of OSDI* (2004).

[11] Howard, J., Dighe, S., Hoskote, Y., Wijngaart, R. V. D., Mattson, T., et al. A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. In *Proceedings of ISSCC 2010* (2010), vol. 9, pp. 58–59.

[12] Hypertransport Consortium. HyperTransport 3 Specification, `http://www.hypertransport.org/default.cfm?page=HyperTransportSpecifications`, 2008.

[13] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of EuroSys* (2007), ACM, pp. 59–72.

[14] Kubaska, T. Shared Memory on Rock Creek, `http://communities.intel.com/docs/DOC-5644`.

[15] Mattson, T., Van der Wijngaart, R., and Frumkin, M. Programming the Intel 80-core network-on-a-chip terascale processor. In *Proceedings of Supercomputing* (2008), IEEE Press.

[16] Murray, D. G., and Hand, S. Scripting the cloud with Skywriting. In *Proceedings of HotCloud* (2010), no. 3.

[17] Murray, D. G., Schwarzkopf, M., Smowton, C., Smith, S., Madhavapeddy, A., and Hand, S. CIEL: a universal execution engine for distributed data-flow computing. In *Proceedings of NSDI* (2011).

[18] Ranger, C., Raghuraman, R., and Penmetsa, A. Evaluating MapReduce for Multi-Core and Multiprocessor Systems. In *Proceedings of HPCA* (2007).

[19] Rasmussen, A., Porter, G., Conley, M., Madhyastha, H. V., Mysore, R. N., Pucher, A., and Vahdat, A. TritonSort: A Balanced Large-Scale Sorting System. In *Proceedings of NSDI* (2011).

[20] van der Wijngaart, R., Mattson, T., and Haas, W. Light-weight communications on Intel's single-chip cloud computer processor. *ACM SIGOPS Operating Systems Review 45*, 1 (2011), 73–83.

[21] Vangal, S. R., Howard, J., Ruhl, G., Dighe, S., Wilson, H., Tschanz, J., Finan, D., Singh, A., Jacob, T., Jain, S., Erraguntla, V., Roberts, C., Hoskote, Y., Borkar, N., and Borkar, S. An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS. *IEEE Journal of Solid-State Circuits 43*, 1 (Jan. 2008), 29–41.

[22] Welsh, M., Culler, D., and Brewer, E. SEDA: an architecture for well-conditioned, scalable internet services. In *Proceedings SOSP* (2001).

[23] Yoo, R., and Romano, A. Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)* (2009).