Distributed Job Allocation for Large-Scale Manycores *

Subramanian Ramachandran, Frank Mueller

North Carolina State University

Abstract. Contemporary operating systems heavily rely on single system images with shared memory constructs that may not scale well to large core counts. We consider the challenge of distributed job allocation, where each job is comprised of a set of tasks to be mapped to disjoint cores. A naive solution performing fragmented allocations may quickly escalate to deadlocks, where jobs hold and wait for cores in circular dependencies. To tackle these challenges, we propose a deadlock free distributed job allocation protocol. We have devised two policies for avoiding deadlocks, namely *active cancellation* and *sequencer-based atomic broadcast*. The protocol and the two policies have been implemented and evaluated on a Tilera TilePro64 processor with 64 cores on a single socket. Results show sparse job allocations to incur lower overhead for *active cancellation* while *sequencer-based atomic broadcast* has less overhead for denser allocations.

1 Introduction

While Moore's law has held for a considerable time in microprocessor design, it has reached its limits and may not keep pace with the ever increasing processing demand. Nonetheless, multicore/manycore processors have the potential to enjoy continued performance increases to meet future processing needs while reducing/constraining power consumption. Current trends in the industry indicate that the number of cores that fit on a single chip is rapidly increasing. With current single microprocessor chips packing 64+ cores on a die [1–3] and specialized computing devices, e.g., graphic processing units (GPUs), already support over 1000 core today.

Current multicores fall short of their scalability potential. One reason for this stems from reusing conventional Single System Image (SSI) OS designs for multicore architectures. With SSI, resources are aggregated to present a single view of the OS environment while data access and communication are realized via shared memory over traditional bidirectional buses. This approach delivers some performance increases in the natural evolution from single core up to 16 cores, but it deteriorates rapidly when the number of cores increases further [4]. Recent work [5, 6, 4] shows that coherent shared memory may not scale well to large core counts. They instead promote the usage of scalable message passing for OS communication in large-scale manycores. Intel's Knights Landing (KNL) next-generation Xeon Phi provides L2 cache coherence via SSI, which may lead to contention at the mesh interconnect, similar to the overheads of shared memory shown to exceed those of simple message passing over the network-on-chip (NoC) for 16 cores or more in previous Tilera research [4, 7]. Based on these observations, we conjecture that future large-scale NoCs may support shared memory partitions only for partitions of 8-16 cores complemented by message passing across partitions. This would depart from an SSI design and necessitate a distributed paradigm, which

* This work was supported in part by NSF grants 0905181 and 1239246.

then requires a distributed job allocation approach for parallel codes to be executed, the focus of this work. An alternative solution would be hierarchical locks for non-uniform memory access (NUMA) systems [8], but this would still require a centralized job allocation strategy with limitations on scalability whereas a distributed approach is more general We believe that research on distributed designs for NoCs is essential for now as the implications of scalability limitations persist.

In this work, we propose a novel protocol to tackle the challenges of job allocation in a distributed system. Allocating jobs of tasks on a partitioned multi-resource system is known to be NP-hard, even for prioritized jobs [9].The problem is further complicated in a distributed system due to the distributed nature of job generation. A naive approach allowing fragmented allocations could quickly lead to deadlocks in the job allocation algorithm. Our distributed job allocation protocol with two policies, *active cancellation* and *sequencer-based atomic broadcast*, takes a well disciplined approach in solving these issues. First, we avoid deadlocks by enforcing a globally unique order to resolve conflicting job allocations. Second, we split the job allocation problem into two subproblems: 1) query and reserve available resources; 2) find a good task-to-core mapping. Such a split enables effective heuristics [10, 11] to tackle NP-hard task-to-core mapping while our distributed job allocation protocol reserves cores for the job.

While our distributed job allocation protocol is generic in scheduling any application, we use Message Passing Interface (MPI) [12] applications as our standard workload in this work for the following reasons: All ranks (tasks) of an MPI program need to start execution at the same time. Such a workload demands guaranteed availability of cores to start execution or waits until they are available. This allows us to model the job wait time as the overhead of the distributed job allocation protocol. And enables more flexible execution models where tasks are dynamically created in a distributed manner, e.g., using fine-grained task graphs to track dependencies.

In summary, this paper, makes the following contributions: (1) We propose the Pico-kernel Adaptive and Scalable Operating System (PICASO) to address the scalability challenges of future manycore processors. (2) We analyze the distributed job allocation problem and present a protocol with two policies, *active cancellation* and *sequencer-based atomic broadcast*. (3) We evaluate the solutions on the Tilera TilePro64 through a set of benchmarks to analyze the performance and scalability.

2 PICASO

PICASO features a distributed message passing system comprised of pico-kernels per core. Pico-kernels are worker cores that execute a job's user tasks. A set of pico-kernels are managed by a micro-kernel. Micro-kernels are dedicated cores for control purposes, e.g., to manage a set of pico-kernels and schedule jobs in coordination with other micro-kernels. Let a *micro-kernel domain* be the set of pico-kernels governed by this micro-kernel. Micro-kernels are typically topographically centered within its domain.

A pico-kernel reports only to its parent micro-kernel. A micro-kernel, on the other hand, apart from controlling its set of pico-kernels, also co-ordinates with other micro-kernels. An advantage of such a system is the decentralization of control, where each micro-kernel may engage in fast and autonomous decisions on managing its set of pico-kernels. Since pico-kernels are just worker cores, we use the terms pico-kernels and cores interchangeably in this work. Fig. 1 shows how a PICASO system with micro-

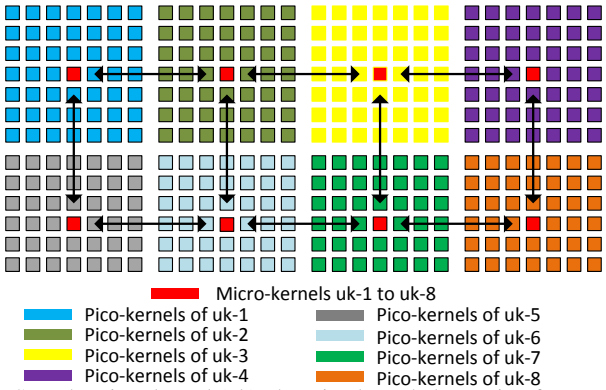


Fig. 1. Sample micro kernel (uk) plus pico kernel abstraction for manycores

and pico-kernel abstraction can be organized in a large-scale manycore system. In contrast to other manycores, PICASO partitions the available cores into different domains represented by different colors. Each domain has a topologically centered core chosen to be the micro-kernel. The chosen micro-kernels (in red) manage their set of pico-kernels. All external interactions occur only between micro-kernels.

3 Distributed Job Allocation

We use the following terminology in our discussion: (1) A *task* is the basic unit of execution. (2) A *job* consists of a collection of tasks. (3) The *home micro-kernel* of a particular job is the micro-kernel where the job submission was initiated.

Assumptions: In this work, we consider jobs that require to be co-scheduled, i.e., these jobs consist of inter-dependent tasks that need to be concurrently executed on different nodes/cores. An example would be jobs of MPI programs, where all associated tasks need to start execution at the same time. Such a job allocation process can be divided into two steps: (1) Query available idle cores and reserve them for this job. (2) Devise the best possible task-to-core mapping from the available cores. Our focus in this paper is on the former part. Once enough cores are reserved for a job, methods and results from prior work [10, 11] can be applied to find the best task-to-core mapping for a given job. However, the problem becomes more complicated when extended to a distributed system due to the nature of job generation.

Conventional solutions involve a centralized resource manager that handles all job allocations. All cores continuously report their availability to this entity. Such an approach does not scale to a large number of cores due to (1) contention at the centralized entity (because of the incessant status updates) and (2) a single point of failure. More importantly, it allows for only a single job submission portal. These restrictions are undesirable for large core counts where jobs generate allocations queued up at different cores throughout the system.

Our proposed pico-/micro kernel distributed system abstraction partitions the available cores between different micro-kernels. This domain-specific delegation of scheduling capabilities to micro-kernels enables jobs that can be locally satisfied within a single micro-kernel domain to be handled by fast and autonomous decisions.

For jobs requiring more cores than can be locally satisfied, the *home micro-kernel*, where the particular job is submitted, co-ordinates with other micro-kernels to devise

the allocation of cores to this job. Multiple job requests submitted at different micro-kernels could compete with each other for resources. Hence, we need a co-ordination protocol to resolve these conflicts and to choose the next job to execute loosely based on a globally unique order. This global unique order could be based on user-defined priority or a First Come First Serve (FCFS) policy. Such an ordering guarantees fairness and avoids starvation. Adhering to loose ordering rather than strict allows non-conflicting job allocations to proceed in parallel, thereby increasing the system utilization.

But a lack of such co-ordination protocols may lead to deadlocks. Deadlocks can happen when multiple jobs submitted at different micro-kernels hold different subsets of cores and wait for more cores to become available. Yet, none are able to proceed because all cores have been allocated to jobs without meeting the full allocation request of any single job in full. Fig. 2 shows a deadlock condition with two micro-kernel domains. Each domain has initially 8 pico-kernels (worker cores) available. In step 1, two job submissions require 12 and 16 cores, respectively. In step 2, each job first holds on to available local cores and sends out a request for more cores. In step 3, each micro-kernel is blocked waiting indefinitely for responses to their requests. Since none of the job requests are fully satisfied, the system remains deadlocked.

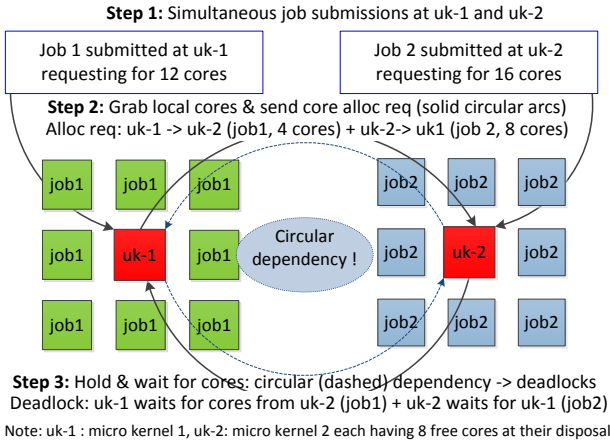


Fig. 2. Deadlock: 2 simultaneous jobs submissions (uk = micro-kernel)

Random back-off schemes could be used to recover in case of potential deadlocks. In such a method, different micro-kernels yield their cores and retry their job allocations after waiting for a randomly chosen back-off time. This probabilistically avoids a deadlock again, but fails to guarantee a bound on completion time for the allocation algorithm. A more serious issue is potential starvation of jobs that require large allocations as they might never be satisfied. Therefore, a job allocation algorithm that avoids starvation with an upper bound on completion time is required.

4 Deadlock-free Job Allocation

We have devised a distributed job allocation protocol for large-scale manycores. Two policies for deadlock avoidance are proposed, namely (1) active cancellation and (2) sequencer-based atomic broadcast. Both of these policies require that a globally unique order be established. For example, we could use timestamps of the job submission time along with the micro-kernel identifier to devise a globally unique job identifier, or we

could use user-defined priorities in conjunction with a method to break ties for matching priorities. For the discussion in this work, we will refer to job priority based on a globally unique job ordering rather than a user-defined priority. In the following sections, we examine the two different approaches, compare their capabilities and finally conclude with a detailed performance evaluation.

4.1 The Main Scheduling Loop

Algorithm 1 shows the main scheduling loop. It performs two main functions: (1) Process any incoming message, and (2) in the absence of an incoming message, schedule pending job requests submitted at this micro-kernel.

Algorithm 1 Scheduling loop at each micro-kernel

```
while TRUE do
  post nonblocking receive for fixed size header
  repeat
    if policy == active cancellation then
      schedule job via active cancellation // (2)
    else if policy == atomic broadcast then
      schedule jobs via atomic broadcast // (2)
    end if
  until fixed size header is received
  receive entire message body (blocking) // (1)
  call respective message handler routine
end while
```

The scheduling loop uses message passing as the only means of communication between micro-kernels, and between a micro-kernel and its set of pico-kernels. There can also be architecture-specific optimizations for micro- to pico-kernel communication, not shown here.

The significant message types of the distributed job allocation protocol are as follows: *Core Allocation Request:* Sent by the *home micro-kernel* of the job. The request is propagated to all micro-kernels via an efficient request propagation scheme. *Core Allocation Response:* Sent by a micro-kernel when it commits certain cores to a particular job. *Job Spawn Request:* Sent by the *home micro-kernel* when it devises the best task allocation for the given job. This request follows the same propagation path earlier traversed by the *Core Allocation Request*. Micro-kernels that are not part of an allocation, release their reservations for this job when they receive this request. *Job Cancel Request:* When the *active cancellation* policy is used, this message is sent by the *home micro-kernel* if it determines that there is an higher priority job to be satisfied first (see Subsection 4.2). *Submit Job to Sequencer:* Under the *sequencer-based atomic broadcast* policy, all micro-kernels use this message to submit their job requests to the fixed sequencer (see Subsection 4.3).

4.2 Active Cancellation

The periodic *active cancellation* procedure works as follows: Any micro-kernel that launches a job requiring more than the locally satisfiable cores sends a core allocation request to all its neighbors. This request is propagated to all other micro-kernels via an efficient request propagation scheme (see Section 4.4). A greedy policy is employed, i.e., the request to *each* micro-kernel always asks for the total number of cores required

for the job, even if other micro-kernels have already simultaneously allocated a subset of cores for this job. This policy frequently allocates more cores than needed for a job, but guarantees a successful allocation (of cores committed to this job) and facilitates termination (unlike non-greedy approaches).

The algorithm handles the arrival of a higher priority core allocation request as follows: Each micro-kernel maintains a wait queue based on the globally unique order consisting of both the job requests it has sent out and the job requests it has received. All incoming job requests are inserted in the wait queue as per the globally unique ordering. If the new request is the head of the wait queue, it first checks if this request has a higher priority than any job request it has sent out earlier. If so, it engages in active cancellation of the lower priority job changing it to the BLOCKED state pending a renewed request. This frees up resources otherwise allocated to unsuccessful lower priority job requests. Finally, the micro-kernel commits as many cores as it can afford for this job request by responding with the committed cores to the *home micro-kernel* of this particular job request. The micro-kernel contributes new cores to this commitment whenever its resources become free. This scheme satisfies multiple job requests *loosely* based on the global ordering but also offers a relaxation to this hard criteria by allowing a lower priority request to proceed if its allocation is satisfied quickly enough before a higher priority job overrides it in the wait queue. This relaxation is allowed under the assumption that any job using a successful allocation will *eventually* complete, after which time the resources it was given becomes available for the next high priority job in the wait queue (bounded by the longest job).

4.3 Sequencer Based Atomic Broadcast

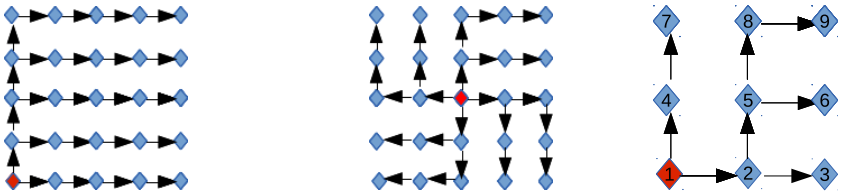
This method is inspired by the sequencer based atomic broadcast as explained in Défago et al. [13]. In this method, a micro-kernel is elected to be the single sequencer of the system. All job requests, even if submitted at different micro-kernels, are in turn forwarded to the sequencer to ensure globally unique ordering . The sequences sends the request to all the micro-kernels only once it has determined which job to execute next. Our approach differs here. Instead of broadcasting the request, we use a custom built request propagation scheme as explained in Section 4.4. This ensures that the job allocations happen in order without any collisions. Less conflicts directly translate to fewer messages compared to *active cancellation*. But since each micro-kernel has to send requests to the sequencer, it leads to contention at the sequencer and additional delays even for small allocation requests, which could have been solved with just a few neighboring micro-kernels. As we show in Section 7, this additional overhead translates into real performance benefits only in case of dense and large job allocations.

4.4 Pattern-Based Message Propagation

An efficient method for propagating request messages, such as core allocation and job spawn requests from any given source to all other micro-kernels in a 2D mesh topology, is required. Multi-casting messages from a given source to all micro-kernels is inefficient as this involves sending individual messages to each micro-kernel, unless hardware support for multi-casting exists [14]. Therefore, we have designed and implemented two alternatives: 1) a fixed pattern-based propagation scheme and 2) an adaptive pattern-based propagation scheme. We use the term *nodes* when introducing these schemes, as they not only apply to micro-kernels but any set of nodes in a 2D mesh

topology. The adaptive pattern-based propagation scheme has the advantage that it does not expect nodes to be arranged in a 2D mesh topology.

Fixed pattern-based propagation When a message needs to be sent to all nodes in a 2D mesh processor NoC, the source sends the message only to its neighboring nodes. Each neighbor in turn propagates the request to its next set of unvisited neighbors following a predefined pattern over a minimum spanning tree. The pattern depends on the placement of the initial source of the message. Consider Fig. 3(a). The source initially sends the request to all its neighbors with an embedded information to propagate the request toward the East (and North if in column 1). Each node receiving this message propagates the request as per the embedded information. Similarly, if the source is located at bottom-right, the propagation will be toward West(+North) etc. Fig. 3(b) shows the pattern when the source is located at the center, in which case each arm takes the responsibility of propagating the request in all four directions. Following such a predefined pattern avoids duplicate requests, which waste link resources and increase processing time at the nodes.



(a) Fixed pattern, source: lower left (b) Fixed pattern, source at center (c) Adaptive pattern

Fig. 3. Pattern-based request propagation schemes

Adaptive pattern-based propagation This scheme involves an initialization phase responsible for forming the adaptive pattern. In this phase, an empty message is forwarded from the given source to all its neighbors. Each neighbor in turn broadcasts the message to all of its next set of neighbors until all the nodes have been visited. At this point, each node has received the given message from multiple sources. It chooses one among these sources as a preferred source and informs it. The preferred source remembers this decision and forwards all messages it receives to this node. The criteria to choose the preferred source can be based on various policies, e.g., the first received request or shortest distance from the source to this node, to name a few. At the end of the first phase, every node has identified its preference from which source it wishes to receive a request in the future; or, alternatively, each node has remembered a list of neighbors to forward a message to that was received from a particular source. This forms an adaptive pattern (spanning tree) ensuring each node receives a message only once. An advantage of such adaptive patterns compared to fixed patterns is that the patterns could be adaptively rearranged in case of link failures. The initialization phase needs to be run only once during the system startup or when recovering from faults, hence reducing the overhead by amortizing the costs.

As an example, consider the pattern shown in Fig. 3(c) for a 3×3 tile with numbered nodes. This pattern is formed with 1 as the source node and forwarding paths from nodes 1 to 2 & 4, 2 to 3 & 5, 4 to 7, 5 to 8 & 6 and 8 to 9.

5 Implementation

The distributed job allocation protocols are applicable to any system of inter-networked cores, even heterogeneous cores [1, 14]. But for the purpose of implementation and experimentation alone, the job allocator has been optimized for a 2D-mesh architecture, such as the Tilera TilePro64 [1, 2]. The Tilera TilePro64 processor has 64 tiles interconnected with a 2D-Mesh NoC interconnect. Each tile has a processor engine running at 700 MHz, a switch engine for routing on the NoC over five different network interconnects and a cache engine. The User Dynamic Network (UDN) interconnect is the only one available for user-generated messages. We use the services of the NoCMsg [4] library. NoCMsg provides a deadlock free, scalable and efficient low-level message passing layer over UDN with an MPI like interface. This motivated our design choice and, hence, our scheduling loop. The protocols and the messages were designed entirely around these MPI like interfaces. This, in itself, makes our design generic enough to be ported to other message passing libraries as well.

For our experiments, we use an ordering based on a FCFS policy. Each tile on the TilePro64 has synchronized clocks. Hence, we use the time-stamp of the job submission along with the unique micro-kernel identifier of the job's home micro-kernel as a tie breaker for job submissions.

6 Evaluation Framework

We use the TilePro64 processor [1] for our evaluation. While the TilePro64 supports 64 tiles, at least two tiles are reserved exclusively by Tilera's hypervisor for administrative tasks and Input/Output operations. The maximum square tile size that can be reserved for user tasks is 7×7. We choose a square tile size so as to eliminate possibilities of discrepancies due to other asymmetric tile sizes. Overall, the Tilera platform limits our evaluation to 49 cores. Tilera supports a subset of Linux (but not a fully compatible Linux design) for system calls that go through the hypervisor. Job allocation, however, becomes the responsibility of the user to pin tasks to specific cores. We lift this burden via our distributed job allocation design, which is agnostic of Tilera's Linux layer and generalizes to any distributed OS design. We support two different experimental

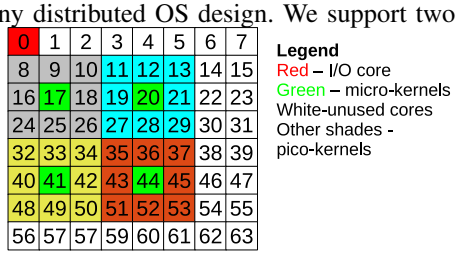


Fig. 4. PICASO system, 6×6 tile on TilePro64

frameworks for testing the performance of the job allocator, (1) a *real task* mode, and (2) a *partial simulation* mode. The *real task* mode supports execution of jobs that are MPI programs from the NAS Parallel benchmarks (NPB). Fig. 4 shows the *real task* mode on the Tilera TilePro64 processor. This small PICASO system on a 6×6 tile has been divided into four regions. Each region has a topologically centered micro-kernel managing a set of 8 pico-kernels. Thus, a combination of NPB of power of two sizes (1,2,4,8,16 and 32) can be executed. This platform is primarily used to assess the schedulability of real user tasks.

The limited number of usable cores on the TilePro64 constraints our scalability tests on the *real task* mode. To overcome this, we have developed a *partial simulation* framework, where we consider all cores in the reserved tile as micro-kernels without pico-kernels. Task execution is simulated by timers triggering a job completion message after a certain user-defined execution time. This simulation platform is justified by the fact that the distributed job allocation protocol requires only micro-kernel interaction. Our results could be directly translated to the *real task* mode combining them with the pico-kernel management overheads obtained in the *real task* mode. *Partial simulation* assesses our protocol with up to 49 micro-kernels on a 7×7 tile.

The following sections detail the experiments/results under the *real task* mode and the *partial simulation* mode for different job allocation mixtures.

7 Experimental Results

The distributed job allocator and user programs are compiled as applications with O3 optimizations using Tilera's C/C++/Fortran compilers of the Multicore Development Environment (MDE) 3.03.

7.1 Performance Analysis

We first analyze the performance of both proposed schemes under *partial simulation*. We execute a set of job loads.For each job, we measure the job allocation overhead as the wait time of the job from the time of submission to the time it receives all the resources to execute. This wait time includes both the overhead of the distributed job allocation protocol and the time spent waiting for the earlier job allocations to terminate and to release its cores. Our focus is to measure the overhead of the distributed job allocation protocol in isolation. Hence, for performance tests, we use an initial state where no jobs are active. We then trigger simultaneous job submissions from different micro-kernels as they have the highest probability to result in fragmented allocations. This creates a workload for our protocol triggering its deadlock avoidance subsystem. Note that all our experiments cover cases where the job allocations require large numbers of cores that need more than one micro-kernel domain to be fully satisfied. Recall that job allocations, which could be satisfied within a single micro-kernel domain, have a constant overhead.

For all our experiments, the reported job wait times are averaged over 15 runs. The maximum relative standard deviation observed in these experiments was less than 20%, except for the experiment in Fig. 5(b) with a relative standard deviations of up to 41%. We discuss this exception and other significant experimental details in the following relevant sections.

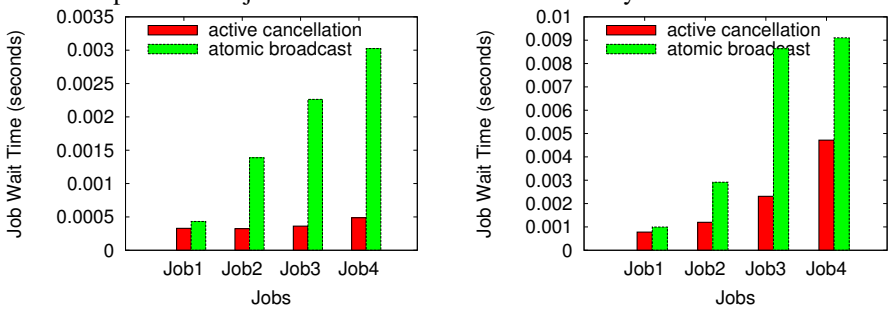
In our experiments, we compare both our proposed polices, *active cancellation* and *sequencer-based atomic broadcast*, against one another. When reporting the relative performance improvement or degradation, we always follow the convention of comparing *active cancellation* against *sequencer-based atomic broadcast* as follows: Let the overhead of *active cancellation* be denoted as O_{ac} and the overhead of *sequencer-based atomic broadcast* be denoted as O_{ab}. Then the relative performance change of *active cancellation* is given by: $\frac{(O_{ab}-O_{ac})}{O_{ab}} \times 100\%$

7.2 Overhead for Sparse Job Allocations

This experiment uses the *partial simulation* mode. Job allocation requests are generated simultaneously from the four extreme corners of a 7×7 tile. These requests can be

satisfied with just a few nearby micro-kernels even before the conflicting job requests arrive from the other corners. Hence, in most of these cases, cancellation of the lower priority job request may not even be required as all the simultaneously submitted jobs are satisfied without the need for global ordering. Conversely, with *sequencer-based atomic broadcast*, all requests have to still go to the single sequencer, which can only serve one request at a time so that serialization delays impact these small job allocations. This experiment proves that *active cancellation* provides best performance in scenarios where sparse job submissions can proceed in parallel.

In the following set of experiments, we consider two scenarios: Jobs that can be execute in parallel and jobs that need to be executed serially one after another.



(a) 3 micro-kernels (b) 12 micro-kernels

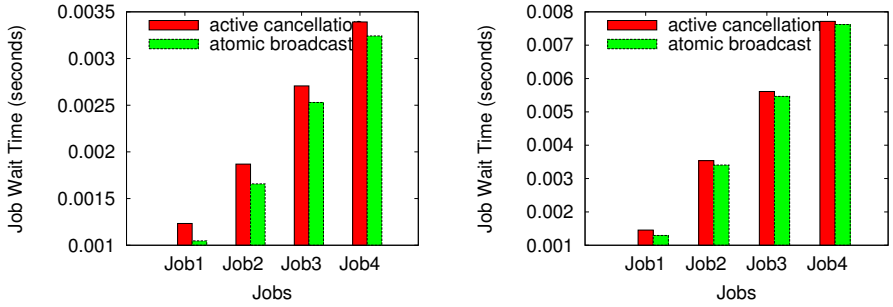
Fig. 5. Overhead for Parallel Allocations

Jobs executing in parallel Figures 5(a) and 5(b) depict the scenario where each job can proceed in parallel. For the four jobs (x-axis), their corresponding job wait times are depicted (y-axis). The job wait time does not include execution times of prior jobs as all these jobs execute in parallel. Hence, the measured job wait time can be considered as the exclusive protocol overhead. We observe a relative decrease in the job wait times for *active cancellation* when compared to *sequencer-based atomic broadcast*.

In the 1st experiment (Fig. 5(a)), each job requires a number of pico-kernels (cores) that is satisfied with available cores from 3 out of a total of 49 micro-kernel domains. We observed a relative performance improvement for *active cancellation* over *sequencer-based atomic broadcast* of 23% for the 1st job, 76% for the 2nd job and 83% for the 3rd and 4th jobs. The serialization at the sequencer results in "backpressure" that aggregates latency (compared to resolving requests in parallel).

In the 2nd experiment (Fig. 5(b)), each job requires a number of pico-kernels (cores) that is satisfied with available cores from 12 out of the total 49 micro-kernel domains. The relative performance improvement of *active cancellation* over *sequencer-based atomic broadcast* for the four jobs were: 21% for the first job, 58% for the second job, 73% for the third job and 48% for the fourth job. For *active cancellation*, we observe a maximum relative standard deviation of 41% in this experiment, which is explained as follows: The wait time of each job depends on how many cancellations are required after the first job has been successfully allocated. In some runs, we observe that a lower priority job request propagated fast enough to succeed in its allocation before a higher priority job triggers cancellation. In these cases, the job wait times for the lower priority jobs are reduced. They are otherwise above average if more cancellations occur.

Jobs executing serially When jobs execute serially, job wait times depend largely on execution times of preceding jobs. When prior jobs take a long time, this becomes the main contributor to the job wait time. Conversely, when the execution time is lower than the minimum job allocation overhead, then the overhead of the distributed job allocation protocol is the main contributor to the job wait time. Hence, for the next two experiments, we consider both short and long running jobs. Short running jobs help assess the actual overhead of the two polices. Long running jobs demonstrate that for serially executing jobs, this performance improvement is not entirely carried over as a reduction in the job wait times.



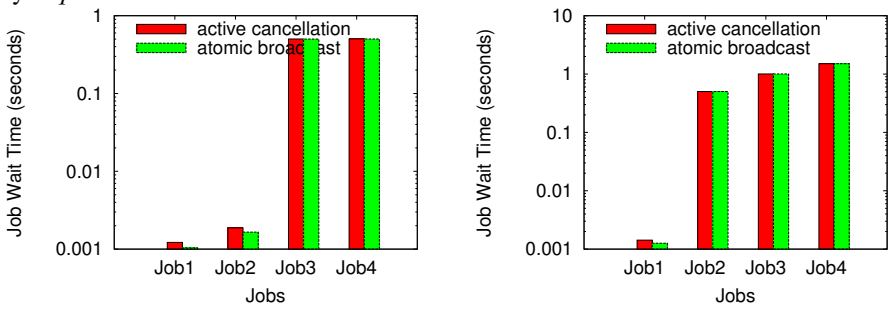
(a) 24 micro-kernels (short runs) (b) 49 micro-kernels (short runs)

Fig. 6. Overhead for Short Jobs

Short Running Jobs: We set the job execution times to 0.001 seconds, which is below the minimum overhead observed. Fig. 6(a) depicts a case where each job requires a number of pico-kernels (cores) that is satisfied by exactly 24 out of the total 49 micro-kernel domains. Hence, two out of the four jobs can run in parallel. As not all jobs can run in parallel, allocations of the lower priority jobs require cancellation so that the allocation of higher priority jobs is satisfied. This results in an additional overhead for *active cancellation* compared to *sequencer-based atomic broadcast* of $\approx$ 17% and $\approx$ 12%, respectively, for the first two jobs, but considerably less for the next two jobs (7% and 4%, respectively). Fig. 6(b) depicts the case where all jobs require a number of pico-kernels (cores) that is satisfied by exactly all available 49 micro-kernel domains and, hence, execute serially one after another. Here, *active cancellation* incurs additional overhead as lower priority job allocations need to be canceled to enforce the globally unique order. The overhead for *active cancellation* is $\approx$ 12% for the 1st job and reduces considerably to 4% for the 2nd job, and then to $\approx$ 1% for the 3rd/4th jobs.

Long Running Jobs: For these experiments, we set the job execution times to 0.5 seconds, which is much higher than the overhead of the distributed job allocation protocol. Hence, in these cases, the execution time is the main contributor to the job wait time. During the initial execution delay for the spawned jobs, the job allocation protocol reorders the job wait queue. Therefore, subsequent jobs are spawned as soon as the earlier jobs complete with a minimal overhead. Fig. 7(a) depicts a case where each job requires a number of pico-kernels (cores) that is satisfied by exactly 24 out of the total 49 micro-kernel domains. Hence, two out of the four jobs can run in parallel. Job wait times are depicted on the y-axis on a *logarithmic* scale. Here, *active cancellation* incurs an additional overhead of 17% and 13%, for the first two jobs, respectively. The additional overhead for the next two jobs is very minimal (0.03% to 0.05%). Fig. 7(b)

depicts the case where all jobs require a number of pico-kernels (cores) that is satisfied by exactly all available 49 micro-kernel domains and, hence, execute serially one after another. Job wait times are depicted on the y-axis on a *logarithmic* scale again. Here, *active cancellation* incurs an additional overhead of $\approx 12\%$ for the first job but only $0.01 - 0.04\%$ for subsequent jobs. Hence, the above experiments show that for long running jobs, which execute serially one after another, the performance gain achieved by *sequencer-based atomic broadcast* is minimal.



(a) 24 micro-kernels (long runs) (b) 49 micro-kernels (long runs)

Fig. 7. Overhead for Long Jobs

7.3 Job Allocation Overhead for Increasing Tile Sizes

In this experiment, we scale the tile size ($n \times n$) from 2×2 to the maximum supported size of 7×7. Per tile size, we generate n simultaneous job requests, each requiring pico-kernels (cores) satisfied by exactly n micro-kernel domains. E.g., in a tile size of 2×2, there will be 2 simultaneous job requests requiring pico-kernels (cores) satisfied by 2 micro-kernels each, and in a tile size of 7×7, there will be 7 simultaneous job requests requiring pico-kernels (cores) satisfied by 7 micro-kernels each. This experiment shows the additional overhead for jobs that can ideally execute in parallel.

The results depicted in Fig. 8 compare the job wait times of the first and last jobs for *active cancellation* and *atomic broadcast*. Here, the job wait times are depicted on the y-axis for different tile sizes on the x-axis. We observe that the wait time for the first among the n jobs is consistently lower for *active cancellation* as it does not incur the overhead of submitting all job requests at the sequencer. We observe a reduction in the job wait time of the first job from 6% for a tile size of 2×2 to up to 60% for a tile size of 7×7. For the sake of analysis, let us assume that the highest priority job overrides all other jobs in their home micro-kernels before any of the lower priority jobs gets a chance to execute. In this case, there will be one initial request sent for the highest priority job. For all other lower priority jobs, there will be $n-1$ initial requests plus $n-1$ cancel and finally $n-1$ repeat requests sent in total. Thus, all subsequent jobs incur this additional overhead. Notice that significant performance gains in spawning the first job compensates for this additional overhead for subsequent jobs to a large extent. Compared to *sequencer-based atomic broadcast*, we observe a slight increase in the job wait times for *active cancellation* ($1 - 12\%$ for smaller tile sizes, i.e., 2×2 and 3×3). But for larger tile sizes, we observe a more significant reduction in the overhead for *active cancellation* (up to 15%). This experiment reinforces our earlier finding that as long as multiple simultaneous job submissions can execute in parallel, *active cancellation* has a lower overhead compared to *sequencer-based atomic broadcast*.

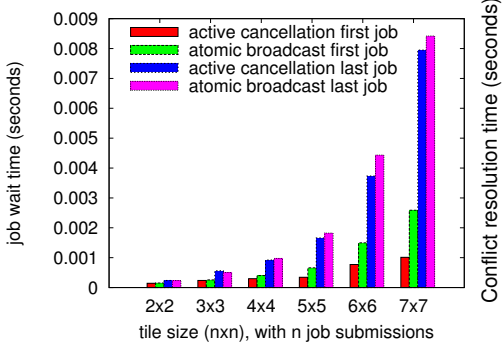


Fig. 8. Job Allocations as tile size increases

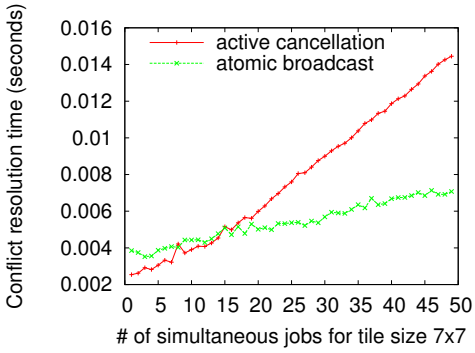


Fig. 9. Worst case for n simultaneous jobs

7.4 Worst-case Conflict Resolution for n Simultaneous Jobs

In this experiment with n simultaneous job submissions, we measure the conflict resolution time for the first job to execute. We use a fixed tile size of 7×7 in the *partial simulation* mode. As all the cores are considered to be micro-kernels in this mode, a maximum of 49 micro-kernels are available. All job submissions require a large number of pico-kernels (cores) that can only be satisfied by the cores available in all the 49 micro-kernel domains. In this worst-case scenario, the *sequencer-based atomic broadcast* scheme provides the best performance. The *sequencer-based atomic broadcast* scheme just has to wait for the job allocation request with the highest priority to arrive. It can then send out core allocation requests one after another. The maximum overhead occurs when the highest priority job request is the one that reaches the sequencer last. Compare this to the considerable overhead in *active cancellation*. Here, in the worst-case, the $n-1$ lower priority job requests together could have reserved all available cores in all micro-kernels. But none would have reserved enough to proceed executing. Hence, for the highest priority job request to execute, it has to override each of the lower priority job request in all other micro-kernels by sending job cancel requests. In the worst-case, $n-1$ cancellation requests need to be sent before the first job can get enough cores for its allocation to be satisfied. We see this reflected in Fig. 9. The wait time for the first job is shown on the y-axis and x-axis depicts n, the number of simultaneous job submissions. We observe that the worst-case performance is better for the *sequencer-based atomic broadcast* scheme once the number of micro-kernels simultaneously requesting allocations exceeds $1/4^{\text{th}}$ of the total number of micro-kernels.

7.5 Experiments with NPB Codes in Real Task Mode

The *real task* mode on the TilePro64, introduced in Section 6, consists of 4 micro-kernels, each managing a set of 8 pico-kernels. We can execute jobs that require a maximum of 32 cores in this mode. To confirm the pattern observed under the *partial simulation* mode, we conduct similar, yet scaled down experiments in *real task* mode.

Job Allocations executing in parallel Here, two jobs (NPB FT Class=S size=16) run in parallel in two different micro-kernel domains with inputs chosen to be L2 resident (to ensure that experiments are not dominated by DRAM memory latencies). Each job requires 16 cores, which can be satisfied in parallel. We measure the average job wait time. This wait time is exclusively due to the protocol overhead as it does not include

any resource wait time. This experiment is an approximation of the sparse job allocations explained in the context of the *partial simulation* mode. We observe results following the same pattern: Under *active cancellation*, less overhead is incurred compared to *sequencer-based atomic broadcast*. These results are shown in Fig. 10. The y-axis depicts job wait time for the two jobs executing in parallel (x-axis).

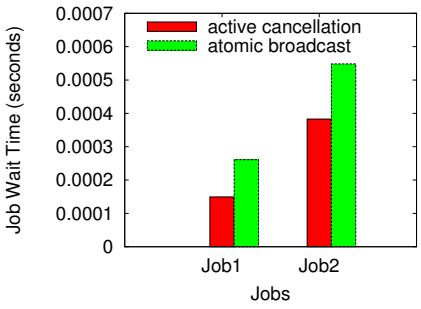


Fig. 10. Real task mode: Parallel Job Alloc.

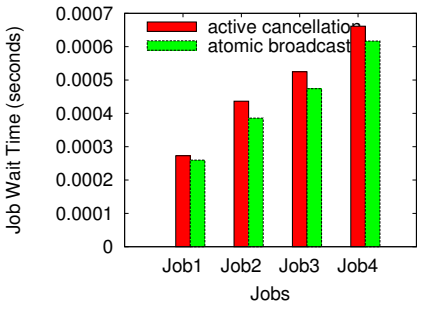


Fig. 11. Real task mode: Serial Job Alloc.

Job allocations executing serially In this experiment, four jobs (NPB FT Class=S size=32) requiring all the 32 cores available from all of the four micro-kernels are submitted simultaneously. These job submissions compete for all resources and are eventually serialized to execute one after another. Thus, this experiment is similar to the *partial simulation* mode experiment in Section 7.4, which measured the worst-case conflict resolution time for n simultaneous job submissions. We obtain similar results, where *sequencer-based atomic broadcast* performs much better than *active cancellation*. Fig. 11 shows these results with the exclusive job wait time on the y-axis for the four jobs on the x-axis. Exclusive job wait time is calculated here as the actual job wait time minus execution times of all prior jobs. This metric provides the job allocation overhead in isolation. A purely centralized approach should perform inferior. But our sequencer approach uses a centralized approach enhanced by contention-free communication over a spanning tree of micro-kernels (Fig. 3), which scales further.

7.6 Performance of Pattern-Based Propagation

To evaluate the impact of scalability of pattern-based message propagation, a simple experiment was devised. A request is broadcasted to all nodes (cores) in a 7×7 tile (max. 49 nodes). The time to broadcast this request and receive a reply from all endpoints in the reverse path of broadcast is measured. The results in Fig. 12 compare the time taken on the y-axis against the number of nodes to which the message is broadcast on the x-axis. Four different schemes are compared: (1) A naive broadcast scheme (The source sends m individual messages to m recipients.); (2) distributed flooding (The source sends the message to all its neighbors who multi-cast the message to their neighbors until all nodes have received the message.); (3) fixed pattern-based propagation (see Section 4.4); and (4) adaptive pattern-based propagation (see Section 4.4). We resort to analysis to determine scalability in number of cores on our single chip platform.

For our analysis, let us assume a tile size of $n \times n$. One sender needs to broadcast the message to the remaining $n^2 - 1$ recipients. Among the different schemes, the naive broadcast scheme tends to be the most time consuming. In this scheme, a single source node sends the message to all the recipients and waits for replies from each of them.

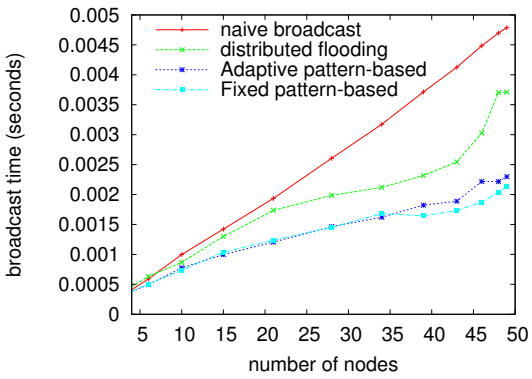


Fig. 12. Different request propagation schemes

This increases the load on the single source. The number of individual end-to-end messages on the NoC equals the number of recipients of the broadcast, i.e., $n^2 - 1$. But it is important to note that, on a 2D mesh topology with X-Y dimension ordered routing, the messages are sent over the same link multiple times resulting in unnecessary link utilization. We can easily observe that as the same X-Y path is traversed multiple times, there is heavy contention on a few links that become the bottleneck.

Distributed flooding performs slightly better. In this method, the load on the single source node is reduced as all nodes contribute to forwarding the message. Also, the message is sent exactly once over each link. But the number of individual messages on the NoC is comparatively larger than that of the naive broadcast scheme. For a tile size of $n \times n$, the total number of messages equals the total number of links on the NoC, i.e., $2n(n-1)$. Hence, after a threshold point, the cost of distributed flooding tends to increase and is as costly as the naive broadcast scheme. This trend was observed in Fig. 12, when the number of nodes is greater than 43.

Fixed pattern-based propagation, where messages propagate in a predefined pattern, uses the least number of individual messages, namely $n^2 - 1$. The fixed pattern reduces the number of links used to $n^2 - 1$ and the message is sent exactly once on each link. Also, the load on the single source node is considerably reduced as each recipient forwards the message further. Hence, pattern-based propagation consumes the least amount of time (see Fig. 12).

In the adaptive pattern-based propagation scheme, the number of individual messages is $n^2 - 1$, which is the same as in the fixed pattern-based scheme. Also, the scheme ensures that the message is sent only once per link. Even the additional cost in setting up the adaptive pattern is amortized over multiple runs. Hence, the adaptive pattern-based scheme performs as good as the fixed pattern-based scheme (see Fig. 12). The adaptive pattern-based scheme is only slightly costlier than the fixed pattern-based scheme. This is explained as follows: Depending on the adaptive pattern formed, certain nodes may need to forward the message to more than one recipient (unlike the fixed pattern-based scheme). E.g., nodes 2 and 5 incur this additional processing time in Fig. 3(c).

8 Related work

Manycores have sparked many OS redesigns [15–18, 6, 5, 19, 20]. Our micro-kernel and pico-kernel abstraction is design for larger number of cores and was inspired by FOS

and Barrelfish [6, 15], where application and OS services run on physically separate cores. In contrast to FOS, we benefit more from spatial locality as pico-kernels (cores) only need to communicate with their parent micro-kernel. We follow the core of the design principles postulated by Peter et al. [21] for designing multi-core schedulers. We even go one level further and take a purely distributed message passing approach as the primary means of communication via adoption of NoCMsg [4] for low-level messaging. Boyd-Wickizer et al. [20] analyze and fix scalability issues in the Linux kernel for several system applications and show that good scalability up to 48 cores could be achieved by modest changes. However, their workload consisted of embarrassingly parallel codes, such as independent Apache threads and parallelized "make" commands.

The compute chip of BlueGene/Q [22, 23] has 16 cores for executing application tasks, one core dedicated to OS services and one (disabled) to increase manufacturing yield. This is similar to our approach of dedicated micro-kernels for OS services and applications. Our design differs as we propose multiple dedicated micro-kernels managing the cores in a manycore chip rather than across nodes. Kobbe et al. [24] provide agent-based allocation on a multicore for malleable applications, where cores of a job are governed by a single agent. ADAM [25] also uses an agent-based approach but requires a global (centralized) agent coordinating smaller agents per cluster of cores. Our approach is much finer grained with multiple micro-kernels coordinating the allocation of a job in a distributed manner. It is also lighter weight than agent-based allocation in Grid/Cloud computing, which use complex allocation schemes with high latency unsuitable for on-chip allocation [26, 27].

Job schedulers for HPC clusters, such as the TORQUE resource manager [28], SLURM [29] and the Maui scheduler [30], use similar algorithms for resource allocation and employ backfilling algorithms to increase utilization. These cluster schedulers are centralized while Mesos [31] only allocates a subset of requested resources and Omega [32] allows parallel schedulers to access shared state in a lock free manner. All of them have scalability limitations due to shared/centralized state (covered by our sequences-based approach in experiments), while our advanced design follows a distributed/message passing design and scales. Omega employs an optimistic concurrency control and has parallel scheduling capabilities. But atomic updates to the shared state serialize scheduling decisions. Instead, we allow individual micro-kernels to be scheduled in parallel and resolve conflicts only when needed. Our techniques and algorithms also have been tailored and optimized to benefit from the on-chip communication of NoC processors. Job co-scheduling for High-end computing (HEC) systems often use a single job submission portal [33, 34], which requires a centralized resource manager that does not scale. Tang et al. [35] propose a distributed job co-scheduler for HEC systems. They propose to resolve deadlocks by yielding the resources after a predefined wait time subject to deadlock (see Section 3). Our approach differs as we avoid deadlocks in job allocation and guarantee a definite completion time for the distributed job allocator. NoC architectures like the Kalray MPPA-256 [14] have specialized support for multi-casting, which can vastly improve the performance of our distributed job allocation protocol as job requests propagate fast resulting in fewer cancellations for *active cancellation*. Most NoC architectures ([1, 3]) lack hardware support for multi-casting, while our efficient pattern-based request propagation schemes can be applied to them.

9 Conclusion

We introduce PICASO, a distributed message passing system, to meet the scalability challenges of future manycore processors and demonstrate the ease and usability of such a system in managing large numbers of cores on a single chip. We study the distributed job allocation problem and propose a protocol with two policies, *active cancellation* and *sequencer-based atomic broadcast*. Both policies avoid fragmented allocations (that would otherwise lead to deadlocks) and guarantee allocations loosely following a global order. Experimental TilePro64 results indicate that for sparse job allocations the *active cancellation* scheme provides lower overhead while for denser job allocations the *sequencer-based atomic broadcast* scheme provides lower overhead.

References

1. : Tilera tile64 processor family https://en.wikipedia.org/wiki/TILE64.
2. Wentzlaff, D., Griffin, P., Hoffmann, H., Bao, L., Edwards, B., Ramey, C., Mattina, M., Miao, C.C., III, J.F.B., Agarwal, A.: On-chip interconnection architecture of the tile processor. IEEE Micro **27** (2007) 15–31
3. Howard, J.e.a.: A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In: Solid-State Circuits Conference Digest of Technical Papers (ISSCC). (2010) 108–109
4. Zimmer, C., Mueller, F.: Nocmsg: Scalable noc-based message passing. In: International Symposium on Cluster Computing and the Grid (CCGRID). (2014) (accepted)
5. Baumann, A., Barham, P., Dagand, P.E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., Singhania, A.: The multikernel: a new os architecture for scalable multicore systems. In: Symposium on Operating Systems Principles. (2009) 29–44
6. Wentzlaff, D., Agarwal, A.: Factored operating systems (fos): the case for a scalable operating system for multicores. SIGOPS Oper. Syst. Rev. **43** (April 2009) 76–85
7. Zimmer, C., Mueller, F.: Nocmsg: A scalable message passing abstraction for network-on-chips. ACM Transactions on Architecture and Code Optimization **12**(1) (February 2015)
8. Chabbi, M., Fagan, M., Mellor-Crummey, J.: High performance locks for multi-level numa systems. In: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. (2015) 215–226
9. Davis, R.I., Burns, A.: A survey of hard real-time scheduling for multiprocessor systems. ACM Computing Surveys (CSUR) **43**(4) (2011) 35
10. Zimmer, C., Mueller, F.: Low contention mapping of real-time tasks onto tilepro 64 core processors. In: IEEE Real-Time Embedded Technology and Applications Symposium. (2012) 131–140
11. Agarwal, T., Sharma, A., Kale, K.: Topology-aware task mapping for reducing communication contention on large parallel machines. In: International Parallel and Distributed Processing Symposium. (April 2006)
12. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: MPI: The Complete Reference. 2 edn. Volume 1. MIT Press (1998)
13. Défago, X., Schiper, A., Urbán, P.: Total order broadcast and multicast algorithms: Taxonomy and survey. ACM Computing Surveys (CSUR) **36**(4) (2004) 372–421
14. de Dinechin, B.D., de Massas, P.G., Lager, G., Lger, C., Orgogozo, B., Reybert, J., Strudel, T.: A distributed run-time environment for the kalray mppa-256 integrated manycore processor. Procedia Computer Science **18**(0) (2013) 1654 – 1663
15. Baumann, A., Peter, S., Schüpbach, A., Singhania, A., Roscoe, T., Barham, P., Isaacs, R.: Your computer is already a distributed system. why isn't your os? In: HotOS. (2009)
16. Gamsa, B., Krieger, O., Appavoo, J., Stumm, M.: Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In: OSDI. (1999) 87–100

17. Boyd-Wickizer, S., Chen, H., Chen, R., Mao, Y., Kaashoek, M.F., Morris, R., Pesterev, A., Stein, L., Wu, M., Dai, Y.h., et al.: Corey: An operating system for many cores. In: OSDI. (2008) 43–57
18. Nightingale, E.B., Hodson, O., McIlroy, R., Hawblitzel, C., Hunt, G.: Helios: heterogeneous multiprocessing with satellite kernels. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, ACM (2009) 221–234
19. Liu, R., Klues, K., Bird, S., Hofmeyr, S., Asanovic, K., Kubiatowicz, J.: Tessellation: Space-time partitioning in a manycore client os. HotPar09, Berkeley, CA **3** (2009) 2009
20. Boyd-Wickizer, S., Clements, A.T., Mao, Y., Pesterev, A., Kaashoek, M.F., Morris, R., Zeldovich, N.: An analysis of linux scalability to many cores. (2010)
21. Peter, S., Schüpbach, A., Barham, P., Baumann, A., Isaacs, R., Harris, T., Roscoe, T.: Design principles for end-to-end multicore schedulers. In: 2nd Workshop on Hot Topics in Parallelism, Berkeley, CA, USA. (2010)
22. Haring, R.A., Ohmacht, M., Fox, T.W., Gschwind, M.K., Satterfield, D.L., Sugavanam, K., Coteus, P.W., Heidelberger, P., Blumrich, M.A., Wisniewski, R.W., et al.: The ibm blue gene/q compute chip. Micro, IEEE **32**(2) (2012) 48–60
23. Boyle, P.: The bluegene/q supercomputer. PoS LATTICE2012 **20** (2012)
24. Kobbe, S., Bauer, L., Lohmann, D., Schröder-Preikschat, W., Henkel, J.: Distrm: Distributed resource management for on-chip many-core systems. In: Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis. (2011) 119–128
25. Al Faruque, M.A., Krist, R., Henkel, J.: Adam: Run-time agent-based distributed application mapping for on-chip communication. In: Design Automation Conference. 760–765
26. Cao, J., Jarvis, S.A., Saini, S., Kerbyson, D.J., Nudd, G.R.: Arms: An agent-based resource management system for grid computing. Sci. Program. **10**(2) (April 2002) 135–148
27. Berman, F., Wolski, R., Casanova, H., Cirne, W., Dail, H., Faerman, M., Figueira, S., Hayes, J., Obertelli, G., Schopf, J., Shao, G., Smallen, S., Spring, N., Su, A., Zagorodnov, D.: Adaptive computing on the grid using apples. IEEE Transactions on Parallel and Distributed Systems **14**(4) (April 2003) 369–382
28. Staples, G.: Torque resource manager. In: Supercomputing. (2006) 8
29. Yoo, A.B., Jette, M.A., Grondona, M.: Slurm: Simple linux utility for resource management. In: Job Scheduling Strategies for Parallel Processing, Springer (2003) 44–60
30. Jackson, D., Snell, Q., Clement, M.: Core algorithms of the maui scheduler. In: Job Scheduling Strategies for Parallel Processing, Springer (2001) 87–102
31. Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A.D., Katz, R., Shenker, S., Stoica, I.: Mesos: A platform for fine-grained resource sharing in the data center. In: USENIX Conference on Networked Systems Design and Implementation. (2011) 295–308
32. Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M., Wilkes, J.: Omega: Flexible, scalable schedulers for large compute clusters. In: European Conference on Computer Systems. (2013) 351–364
33. Huedo, E., Montero, R.S., Llorente, I.M.: A framework for adaptive execution in grids. Software: Practice and Experience **34**(7) (2004) 631–651
34. Kannan, S., Roberts, M., Mayes, P., Brelsford, D., Skovira, J.F.: Workload management with loadleveler. IBM Redbooks **2** (2001) 2
35. Tang, W., Desai, N., Vishwanath, V., Buettner, D., Lan, Z.: Job coscheduling on coupled high-end computing systems. In: ICPP Workshops. (2011) 317–326