ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

ARCHIVIO ISTITUZIONALE
DELLA RICERCA

Alma Mater Studiorum Università di Bologna
Archivio istituzionale della ricerca

Controlling NUMA effects in embedded manycore applications with lightweight nested parallelism support

This is the final peer-reviewed author's accepted manuscript (postprint) of the following publication:

Published Version:
Marongiu, A., Capotondi, A., Benini, L. (2016). Controlling NUMA effects in embedded manycore applications with lightweight nested parallelism support. PARALLEL COMPUTING, 59(Special issue), 24-42 [10.1016/j.parco.2016.02.002].

Availability:
This version is available at: https://hdl.handle.net/11585/575138 since: 2020-06-01

Published:
DOI: http://doi.org/10.1016/j.parco.2016.02.002

Terms of use:

Some rights reserved. The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (https://cris.unibo.it/).
When citing, please refer to the published version.

(Article begins on next page)

12 October 2024

This is the post peer-review accepted manuscript of:

Marongiu, Andrea, Alessandro Capotondi, and Luca Benini. "Controlling NUMA effects in embedded manycore applications with lightweight nested parallelism support." Parallel Computing 59 (2016): 24-42.

DOI: http://dx.doi.org/10.1016/j.parco.2016.02.002

The published version is available online at:
https://www.sciencedirect.com/science/article/pii/S016781911600044

IN COPYRIGHT - NON-COMMERCIAL USE PERMITTED

This Item is protected by copyright and/or related rights. You are free to use this Item in any way that is permitted by the copyright and related rights legislation that applies to your use. In addition, no permission is required from the rights-holder(s) for non-commercial uses. For other uses you need to obtain permission from the rights-holder(s).

Controlling NUMA effects in embedded manycore applications with lightweight nested parallelism support

Andrea Marongiu[a,b,*], Alessandro Capotondi[b], Luca Benini[a,b]

[a] *D-ITET, Swiss Federal Institute of Technology in Zurich (ETHZ). Gloriastrasse 35, 8092 Zurich, Switzerland.*
[b] *DEI, University of Bologna. Viale Risorgimento 2, 40136 Bologna, Italy.*

Abstract

Embedded manycore architectures are often organized as fabrics of tightly-coupled shared memory clusters. A hierarchical interconnection system is used with a crossbar-like medium inside each cluster and a network-on-chip (NoC) at the global level which make memory operations nonuniform (NUMA). Due to NUMA, regular applications typically employed in the embedded domain (e.g., image processing, computer vision, etc.) ultimately behave as irregular workloads if a flat memory system is assumed at the program level. Nested parallelism represents a powerful programming abstraction for these architectures, provided that i) streamlined middleware support is available, whose overhead does not dominate the run-time of fine-grained applications; ii) a mechanism to control thread binding at the cluster-level is supported. We present a lightweight runtime layer for nested parallelism on cluster-based embedded manycores, integrating our primitives in the OpenMP runtime system, and implementing a new directive to control NUMA-aware nested parallelism mapping. We explore on a set of real application use cases how NUMA makes regular parallel workloads behave as irregular, and how our approach allows to control such effects and achieve up to 28× speedup versus flat parallelism.

Keywords: Manycores, nested parallelism, OpenMP

*Corresponding author
Email address: a.marongiu@iis.ee.ethz.ch (Andrea Marongiu)

1. Introduction

The multi-core paradigm has allowed system-on-chip (SoC) designers to successfully tackle many technology walls in the past decade [1] [2] and has now entered the manycore era, where hundreds of simple processing units (PU) are integrated on a single chip. To overcome the scalability bottlenecks encountered when interconnecting such a large amount of PUs, several recent embedded manycore accelerators leverage tightly-coupled *clusters* as building blocks. Representative examples include NVIDIA X1 [3], Kalray's MPPA 256 [4], PEZY-SC [5], ST Microelectronics STHORM [6]. These products leverage a hierarchical design, which groups PUs into small-medium sized subsystems (clusters) with shared L1 memory and high-performance local interconnection. Scalability to larger system sizes employs cluster replication and a scalable interconnection medium like a network-on-chip (NoC). A shared memory model is often assumed, where each cluster can access local or remote (i.e., belonging to another cluster) L1 storage, as well as L2 or L3 memories. However, due to the hierarchical nature of the interconnection system, memory operations are subject to non-uniform accesses (NUMA), depending of the physical path that corresponding transactions traverse.

NUMA has been historically a key architectural feature of large-scale high performance computing (HPC) systems. Regular applications parallelized with a flat memory system in mind ultimately behave as highly irregular workloads in a NUMA system. Indeed regular workload parallelization assumes that nominally identical shares of computation and memory will be assigned to threads. If such threads are mapped to processors which feature a different access time (latency/bandwidth) to the target memory, such threads will experience very different execution times. Table 1 shows the execution time (in 100K cycles) of several applications running on the multi-cluster accelerator considered in this work[1]. The first row refers to a *high-locality* configuration, where the applica-

[1] for more details on the benchmarks and the platform see Sections 4 and 2.

	Color Tracking	**FAST**	**Mahalanobis**	**Strassen**	**NCC**	**SHOT**
High-locality	5	49	25	201	47	4
Poor-locality	136	223	102	638	245	16
Variance	*22×*	*5×*	*4×*	*3×*	*5×*	*4×*

Table 1: Irregular behavior induced by NUMA in regular workloads [×100KCycles].

tions are executed on a single cluster and the data is accessed from the same cluster's L1 memory. The second row refers to a *poor-locality* configuration, where the applications are executed on a single cluster and the data is accessed from a remote cluster's L1 memory. Even if the applications have completely regular access pattern, NUMA effects lead to up to 22× variance between clusters, if data is not distributed in an architecture-aware manner. The barrier semantics implied at the end of a fork/join contruct will force fast clusters to sit idle waiting for the slow clusters to complete.

Well consolidated programming practices have been established in the HPC domain for the control of NUMA, but such practices need to be revisited for adoption in the embedded manycore domain, due to some key differences between the latter and HPC systems. First, large-scale HPC systems rely on the composition of several SMP nodes, where inter-node communication leverages orders-of-magnitude slower channels than the coherent multi-level cache hierarchy within each node (intra-node memory hierarchies are in fact transparent to the program). In embedded manycores L1 and L2 memories are typically implemented as scratchpads (SPM), which are explicitly managed by the program via DMA transfers. Inter-cluster communication is much costlier than local memory access, yet it is way faster compared to inter-node communication in HPC systems, as it leverages on-chip interconnection.

For these reasons, in HPC systems it is common to use a combination of message passing (MPI), for inter-node communication, and *fork/join thread parallelism* (e.g., OpenMP [7]) within a node. Direct access to a remote note from within parallel threads is typically disallowed. The locality of memory operations within a node is managed transparently by caches. Intra-node NUMA

effects in multi-socket systems are mitigated by pinning threads to specific cores (*thread binding*). In embedded manycores remote cluster access is sometimes allowed (e.g., if data produced in a remote cluster needs to be accessed only once or has in general poor reuse), thus while MPI could still be used for intra-cluster communication [8], there is in general wider consensus towards simpler and unified programming interfaces such as OpenMP.

Another important difference between HPC and embedded manycore systems is found at the level of applications and software stacks. Applications in HPC typically leverage coarse-grained parallel tasks, capable of tolerating large overheads implied by underlying runtime systems running on top of legacy operating system (OS), libraries, etc. Applications in the embedded domain leverage fine-grained parallelism and run on top of native hardware-abstraction-layers (HAL), while a full-fledged OS is typically lacking. Support for programming models such as OpenMP is usually designed as a streamlined SW layer on top of bare metal [9] [10] [11].

Nested (or *multi-level*) parallelism represents a powerful programming abstraction for cluster-based embedded manycores, addressing the issues of efficient exploitation of i) a large number of processors and ii) a NUMA memory hierarchy. Nested parallelism has been traditionally used to increase the efficiency of parallel applications in large systems. Exploiting a single level of parallelism means that there is a single thread (master) that produces work for other threads (slaves). Additional parallelism possibly encountered within the unique parallel region is ignored by the execution environment. When the number of processors in the system is very large, this approach may incur low performance returns, since there may not be enough coarse-grained parallelism in an application to keep all the processors busy. Nested parallelism implies the generation of work from different simultaneously executing threads. Opportunities for parallel work creation from within a running parallel region result in the generation of additional work for a set of processors, thus enabling better resource exploitation.

In this paper we explore the applicability of nested parallelism plus thread-

binding capabilities as an afficient means of controlling NUMA effects in cluster-based embedded manycore. Matching the key requirements of embedded applications, the focus is on two key aspects: i) enabling fine-grained parallelism via streamlined support of nesting; ii) leveraging the ability of clustering threads hierarchically, where outer levels of coarse-grained (task) parallelism could be distributed among clusters, and data (e.g., loop) parallelism could be used to distribute work within a cluster.

Our work is based on the STMicroelectronics STHORM [6] manycore, plus a cycle-accurate SystemC simulator of an architectural template modeled after STHORM or similar cluster-based platforms (e.g., Kalray MPPA [4]). The simulator is used to explore hardware extensions to accelerate critical (cost-wise) operations for nested fork/join thread management.

More in details, we make the following contributions:

1. We present a lightweight runtime layer for nested parallelism on cluster-based embedded manycores, identifying the most critical operations to fork and join nested parallelism, and proposing SW-only and HW-accelerated solutions for their efficient implementation.
2. We integrate our fork/join primitives in the OpenMP runtime system, and implement an extension to expose an abstract notion of clusters at the programming interface level, so as to make nested parallelism mapping NUMA-aware.
3. We show with several real application use cases how a regular workload partitioning scheme that considers flat memory translates into irregular thread behavior due to NUMA effects, and how this ultimately impacts the performance. We show that our solution allows to achieve very high speedups even for very fine-grained workloads.

The rest of the paper is organized as follows. In Section 2 we present the target architectural template. Our optimized implementation of the support for nested parallelism is described in Section 3, and the experimental evaluation and results are discussed in Section 4. Section 5 discusses previous work on

nested parallelism support, while Section 6 contains conclusive remarks.

2. Architectural Template

In this section we describe the *cluster*-based manycore architecture targeted in this paper. *Clusters* are the central building block of several recent manycores [4] [6] [3] [5] [12]. These products consider a hierarchical design, where simple processing units (PU) are grouped into small-medium sized subsystems (the *clusters*) sharing high-performance local interconnection and memory. Scaling to larger system sizes is enabled by replicating *clusters* and interconnecting them with a scalable medium like a NoC.

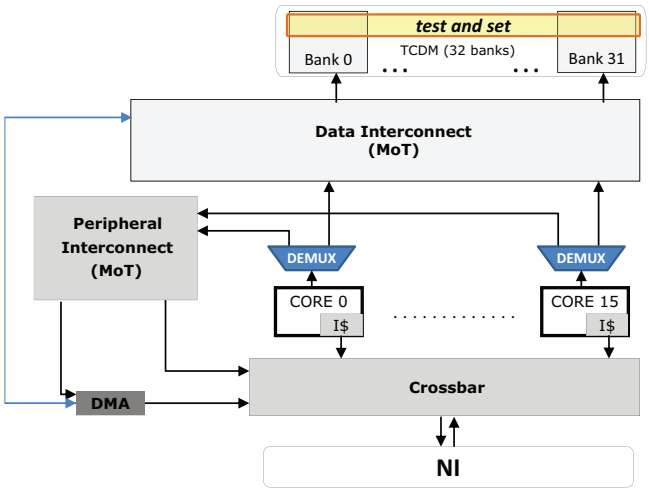

Figure 1: On-chip shared memory cluster

The simplified block diagram of the target *cluster* is shown in Figure 1. It contains up to sixteen RISC32 processor cores, each featuring a private instruction cache. Processors communicate through a multi-banked, multi-ported Tightly-Coupled Data Memory (TCDM). This shared L1 TCDM is implemented as explicitly managed SRAM banks (i.e., scratchpad memory), to which processors are interconnected through a low-latency, high-bandwidth data interconnect. This network is based on a logarithmic interconnection design which

allows 2-cycle L1 accesses (one for request, one for response). This is compatible with pipeline depth for load/store for most processors, hence it can be executed in TCDM without stalls – in absence of conflicts. Note that the interconnection supports up to 16 concurrent processor-to-memory transactions within a single clock cycle, given that the target addresses belong to different banks (one port per bank). Multiple concurrent reads at the same address happen in the same clock cycle (broadcast). A real conflict takes place only when multiple processors try to access different addresses within the same bank. In this case the requests are sequentialized on the single bank port. To minimize the probability of conflicts i) the interconnection implements address interleaving at the word-level; ii) the number of banks is M times the number of cores (M=2 by default).

Processors can synchronize by means of standard read/write operations to an area of the TCDM which provides *test-and-set* semantics (a single atomic operation returns the content of the target memory location and updates it).

Since the L1 TCDM has a small size (256KB) it is impossible to permanently host all data therein or to host large data chunks. The software must thus explicitly orchestrate data transfers from main memory to L1, to ensure that the most frequently referenced data at any time are kept close to the processors. To allow for performance- and energy- efficient transfers, the cluster is equipped with a DMA engine.

The overall manycore platform is composed of a number of *clusters*, interconnected via a NoC as shown in Figure 2. The topology we consider in our experiments is a simple 2×2 mesh, with one cluster at each node, plus a memory controller to the off-chip main memory.

Overall, the memory system is organized as a partitioned global address space. Each processor in the system can explicitly address every memory segment: local TCDM, remote TCDMs and main memory. Clearly, transactions that traverse the boundaries of a cluster are subject to NUMA effects: higher latency and smaller bandwidth.

This architectural template captures the key traits of existing cluster-based

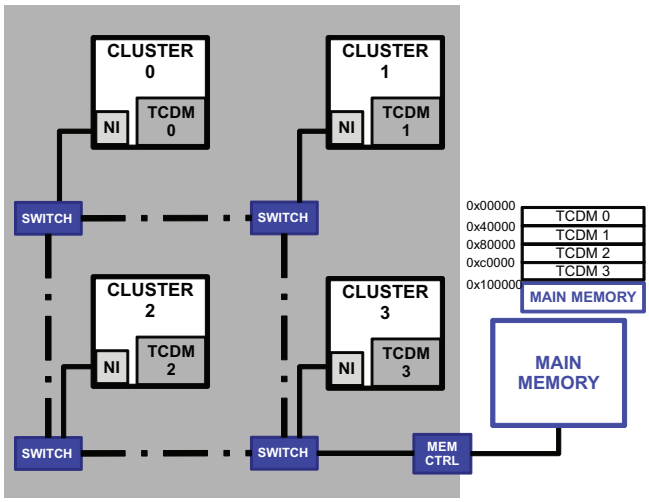

Figure 2: Multi-cluster architecture

manycores such as STMicroelectronics STHORM [6] or Kalray MPPA [4] in terms of core organization, number of clusters, interconnection system and memory hierarchy. As a concrete instance of this template we built a cycle-accurate SystemC simulator, based on the *VirtualSoC* virtual platform [13]. *VirtualSoC* is a prototyping framework developed at University of Bologna, targeting the full-system simulation of massively parallel heterogeneous SoCs. It allows to easily instantiate several manycore templates, as the number of cores and clusters, the interconnect type and the memories are fully parameterizable. The platform also comes with tools and libraries for software developments, on top of which we built our runtime system for lightweight nested parallelism support, plus accurate counters for performance measurement and execution traces, which we use to evaluate the effectiveness of our techniques. The *VirtualSoC* simulator can also be easily extended thanks to a fully modular design. We exploit this feature to explore the benefits of adding custom HW blocks to the platform to accelerate the execution of critical parts of fork/join mechanisms (see Section 3.2.1).

The *VirtualSoC* simulator, the HW extensions and the programming model described in this paper can be downloaded (currently as beta version) by

contacting the authors through the group website (`http://www-micrel.deis.unibo.it/virtualsoc/`).

3. Nested Parallelism Support

In this section we present our lightweight support for nested parallelism, targeting the STHORM and the VirtualSoC platforms. Similar to most embedded parallel platforms, the presented runtime system sits on top of bare metal, as an OS is lacking. More specifically, we build upon native hardware abstraction layer (HAL) support for basic services such as core identification, memory allocation and lock (test and set memory) reservation. In the following we first introduce the basic design choices to enable compact support data structures and low-cost fork/join primitives (Section 3.1). Then, we identify critical operations for scalable nested parallelism support, discussing several optimizations to their implementation and deal with NUMA memory effects (Section 3.2). This section also describes a NUMA-aware core binding extension to OpenMP for data locality. Performance characterization and experimental results throughout this section focus on highlighting the benefits of **NUMA-aware thread management** only, enabled by our proposal. The additional benefits of **NUMA-aware core binding** on data access behavior in real applications are evaluated in the next Section *Benchmarks* (Section 4).

3.1. Key Design Choices for Streamlined Nested Parallelism Support

A central design choice for our lightweight nested parallelism support is the adoption of a *fixed thread pool* (FTP) approach. At boot time we create as many threads as processors, providing them with a private stack and a unique ID (matching the hosting processor ID). We call these threads *persistent*, because they will never be destroyed, but will rather be re-assigned to parallel teams as needed. Persistent threads are non-preemptive. We promote the thread with the lowest ID as the *global master thread*. This thread will be running all the time, and will thus be in charge of generating the topmost level of parallelism.

The rest of the threads are docked on the global pool, waiting for a master thread to provide work. At startup, all the persistent threads other than the global master (hereafter called the *global slaves*) execute a microkernel code where they first notify their availability on a private location of a global array (Notify-Flags, or *NFLAGS*), then they wait for work to do on a private flag of another global array (Release-Flags, or *RFLAGS*). To minimize the probability of banking conflicts on the TCDM when multiple processors are accessing these data structures, we allocate them in such a way that consecutive elements of the arrays are mapped on contiguous memory banks. In this way each processor insists on a different TCDM bank. The status of global slaves on the thread pool (idle/busy) is annotated in a third global array, the *global pool descriptor*. When a master thread encounters a request for parallelism creation, it fetches threads from the pool and points them to a work descriptor.

Besides the global data structures above described, each thread *team* has an associated *team descriptor*. This data structure relies on a simple bitmask to describe the composition of the nested teams. The mask has as many bits as the number of persistent threads. Bits corresponding to the IDs of the threads belonging to the team are set to 1. This allows multiple coexisting teams by masking only the fields of the global data structures that are of interest for the current team, as shown in Fig. 3. Furthermore, the use of bitmasks allows to

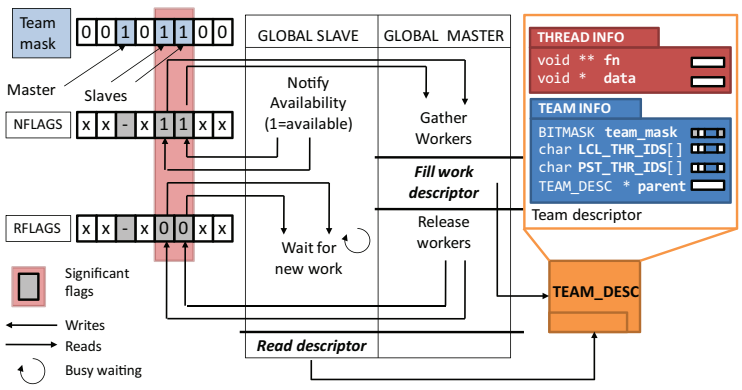

Figure 3: Thread docking, synchronization and team descriptor

quickly inspect the status of individual threads and update team composition through fast bitwise logic operations.

A more detailed description of the *team descriptor* and its data structures is provided in the following.

3.1.1. Forking threads

Nested parallelism allows multiple threads to concurrently act as masters and create new thread teams. The first information required by a master to create a parallel team is the status of the global slaves in the pool. As explained, this information in stored in the *global pool descriptor* array. Since several threads may want to concurrently create a new team, accesses to this structure must be locked.

Let us consider the example shown in Fig. 4. Here we show the task graph

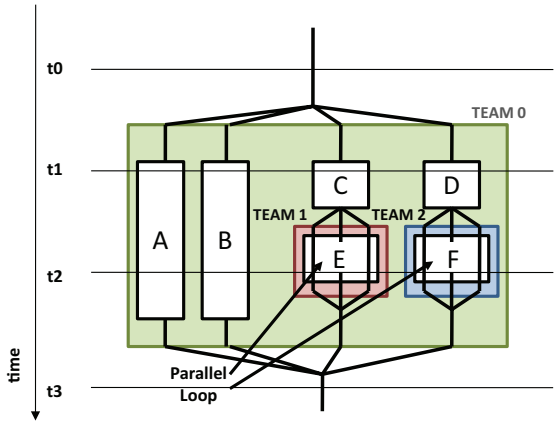

Figure 4: Application with nested parallelism

of an application which uses nested parallelism. At instant *t0* only the global master thread is active, as reflected by the pool descriptor in Figure 5. Then parallel *TEAM 0* is created, where tasks A, B, C and D are assigned to threads 0 to 3. The global pool descriptor is updated accordingly (instant *t1*). After completing execution of tasks C and D, threads 2 and 3 are assigned tasks E and F, which contain parallel loops. Thus threads 2 and 3 become masters of *TEAM 1* and *TEAM 2*. Threads are assigned to the new teams as shown in

Fig. 5 at instant *t2*. Note that the number of slaves actually assigned to a team may be less than what requested by the user, depending on their availability.

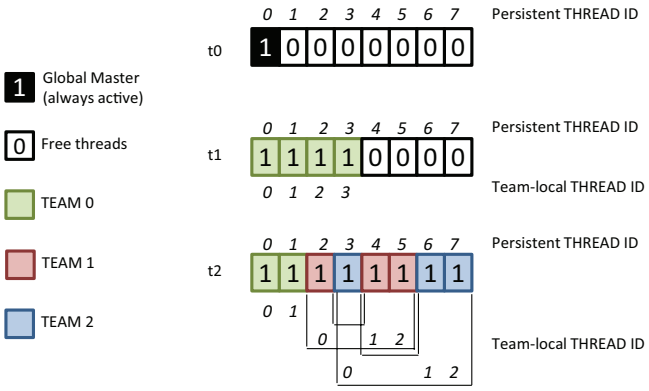

Figure 5: Global pool descriptor

Besides fetching threads from the global pool, creating a new parallel team involves the creation of a *team descriptor* (Fig. 3), which holds information about the work to be executed by the participating threads. This descriptor contains two main blocks:

1. *Thread Information*: A pointer to the code of the parallel function, and its arguments.
2. *Team Information*: when participating in a team, each thread is assigned a team-local ID. The ID space associated to a team as seen by an application is expressed in the range 0,..,N-1 (N being the number of threads in the team).

To quickly remap local thread IDs into the original persistent thread IDs and vice versa, our data structure maintains two arrays. The *LCL_THR_IDS* array is indexed with persistent thread IDs and holds corresponding local thread IDs. The *PST_THR_IDS* is used for services that involve the whole team (e.g., joining threads, updating the status of the pool descriptor), and keeps the dual information: it is indexed with local thread IDs and returns a persistent thread ID. Moreover, to account for region nesting each descriptor holds a pointer to the parent region descriptor. This enables fast context switch at region end.

The memory footprint for this descriptor grows with the number N of cores with the following formula:

$$F(N)_{bytes} = ceil[\frac{N}{8}] + 2N + 12$$

For the 64-core system implementation considered in this paper a team descriptor occupies 148 Bytes. Once the team master has filled all its fields, the descriptor is made visible to team slaves by storing its address in a global *TEAM_DESC_PTR* array (one location per thread). Fig. 6 shows a snapshot of the *TEAM_DESC_PTR* array and the tree of team descriptors at instant *t2* from our previous example.

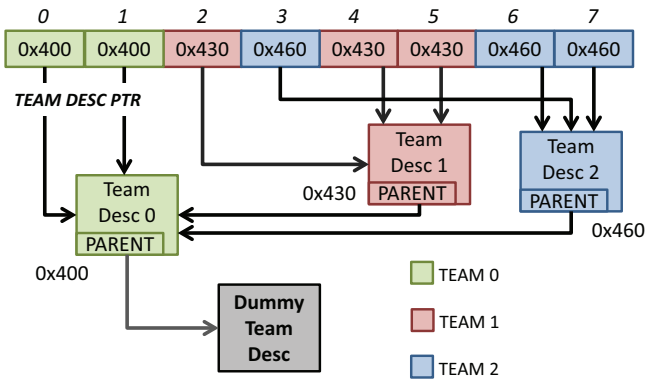

Figure 6: Tree of team descriptors to track nesting

3.1.2. Joining Threads

Joining threads at the end of parallel work involves global (barrier) synchronization. Supporting nested parallelism implies the ability of independently synchronizing different thread teams (i.e., processor groups). To this aim, we leverage the mechanism described previously to dock threads, which behaves as a standard *Master-Slave* (MS) barrier algorithm, extended to selectively synchronize only the threads belonging to a particular team. The MS barrier is a two-step algorithm. In the *Gather* phase, the master waits for each slave to notify its arrival on the barrier on a private status flag (our NFLAGS array). After arrival notification, slaves check for barrier termination on a separate private

location (our RFLAGS array). The termination signal is sent by the master in these private locations during the *Release* phase of the barrier. Fig. 3 shows how threads belonging to *TEAM 1* (instant *t2* of our example) synchronize through these data structures.

An implementation for a single-cluster architecture of this basic support infrastructure for nested parallelism has been presented in our earlier work [11]. For more details interested readers are referred to this paper. In the following sections we describe how the basic concepts illustrated here need to be extended when multi-cluster architectures with NUMA memory hierarchy are concerned.

3.2. Nested Parallelism on Multi-Clusters

The most straightforward solution to extend the described nested parallelism support to a multi-cluster manycore is that of enlarging data structures (*RFLAGS*, *NFLAGS*, *global pool* and *team* descriptors) to accommodate information for a very large number of cores, while maintaining an identical, non-hierarchical mechanism for thread docking and recruiting. This naive extension leverages centralized data structures and centralized control, and is subject to two main sources of inefficiencies. First, many operations which depend on the number of involved slaves are sequentialized on a single (master) thread. Second, when we cross the physical boundary of a cluster, NUMA memory effects impact the cost for team creation and close.

Using nested parallelism provides a natural solution to the first issue. Here, the *global master* should be able to create a first (OUTER) team composed of as many threads as clusters, and to map each of these threads on the first core of each cluster. These slaves would then become *local masters* of a nested (INNER) team on each cluster. This parallelizes the creation of teams spanning multiple clusters over multiple cluster controllers (local masters).

To deal with the second issue we need to design a mechanism that creates local team descriptors for the inner regions, confining the accesses to the data structures within a cluster and preventing NUMA effects. The first modification in this direction is the distribution of all the runtime support structures. To

guarantee locality of bookkeeping operations when inner regions are created, all these structures must be reorganized per-cluster.

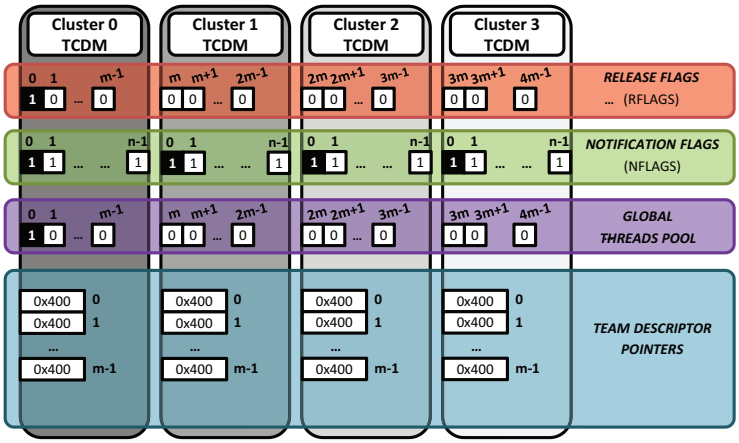

Figure 7: Distributed runtime support data structures

Figure 7 shows how this is done. RFLAGS for all threads on a given cluster are allocated in the same TCDM. The way "virtual" (team-specific) thread IDs are calculated is also made cluster-aware. Given M (the number of threads on a cluster) - and CL_{id} (the cluster ID), RFLAGS on a TCDM are indexed in the range $[CL_{id} \times M \ ; (CL_{id} + 1) \times M - 1]$.

The *global thread pool* and per-thread team descriptor pointers are distributed in the same manner.

NFLAGS must be organized differently, since they are used by team masters to synchronize with slaves during the join phase. Thus, to ensure that polling on these flags is always performed on local memory, we replicate the whole NFLAGS array (one flag per each core in the system) over every TCDM.

Another key feature that we need to support is fetching threads in a cluster-aware manner during the fork phase. To this end, we modify the team fetch algorithm to selectively allow scanning the *global thread pool* with a *stride* M, starting from the current master thread ID.

Figure 8 shows the breakdown of fork and join execution time on *VirtualSoC* as the total number of threads is increased. Here the OUTER thread team is

composed of 4 threads (one per cluster), while the INNER teams have 1 to 16 threads each. We show in the plot as many bars as the number of local masters. The total time is broken down in three main contributions for both outer and inner regions: INIT (memory allocation and data structure initialization); FETCH (thread recruitment) and RELEASE (thread start). All these contributions increase linearly with the number of involved threads, and it is where we will focus our optimization effort in the next section. The Y-axis reports execution cycles, but along this direction the plot can be read as a timing diagram. It is possible to notice that the start time of different INNER masters is not aligned, since creating the OUTER team is done on a single master, which starts new threads in sequence. This clearly affects the overall duration of the fork operation and, eventually, of the parallel computation synchronization.

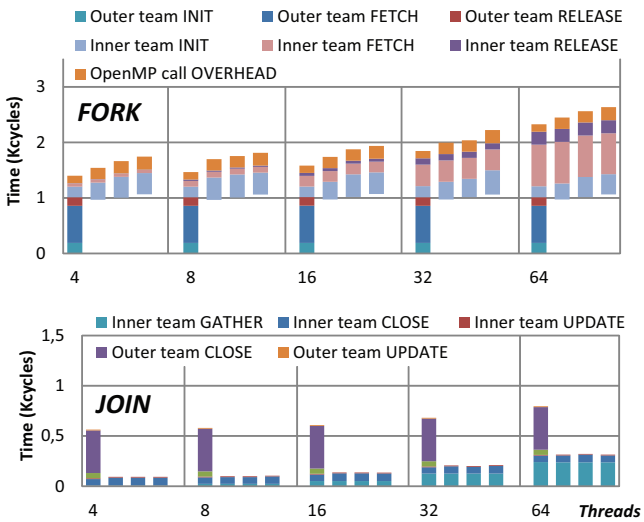

Figure 8: Execution cycles scaling for explicit nesting. One bar per cluster (local master).

Overall, the time to fork a 64-thread team is ≈2700 cycles. This is 33% faster compared to the naive centralized approach.

For the join operation we measure the contribution for three main phases: GATHER (verify that all threads have joined), CLOSE (dispose of allocated memory and data structures) and UPDATE (point global data structures to

current parallel team). In this case GATHER increases linearly with the number of threads, which is what we try to minimize in the following.

Overall, joining 64 threads has a cost of ≈800 cycles, which is 20% faster compared to the centralized approach.

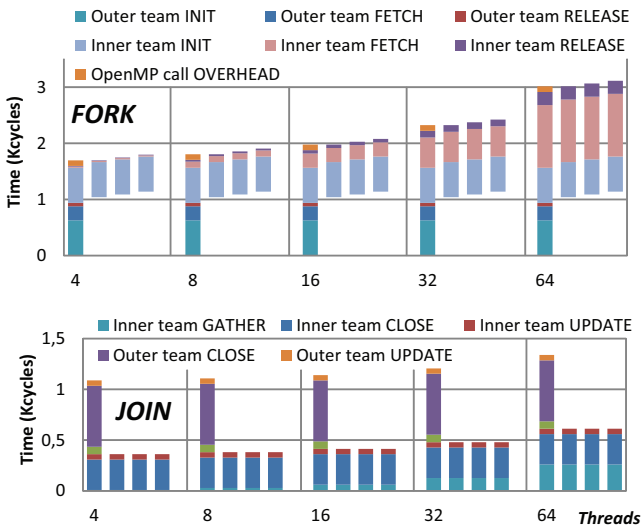

Figure 9: Execution cycles scaling for explicit nesting on STHORM. One bar per cluster (local master).

Results for this same experiment running on the STMicroElectronics STHORM development board are shown in Figure 9. Overall, it is possible to see that the fork cost for 64 threads increases to ≈3000 cycles (10% higher than *VirtualSoC*). This is mainly due to the higher cost for the memory allocation primitives provided with the official STHORM SDK, which we did not optimize. This effect is even more evident for the join operation, where the free primitives issued by multiple threads are sequentialized on a single cluster controller processor. By optimizing the memory management libraries on the STHORM SDK we could clearly achieve nearly identical results to those obtained on the *VirtualSoC* simulator.

3.2.1. Hardware-accelerated nested parallelism

From Figure 8 we see that during the concurrent creation of the INNER (nested) parallel teams, there are basically three sections of the algorithm that require linearly increasing time with the number of slaves, and which deserve more attention. Thread *fetch* and *release* for the fork phase, and thread *gather* for the join phase, as shown in the leftmost plot in Figure 10.

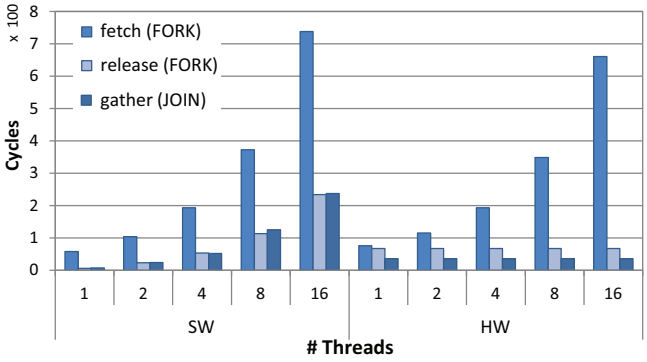

Figure 10: Execution cycles scaling of *fork* and *join*.

It has to be observed that:

1. during *release* the team master sequentially writes into RFLAGS (one write per slave). This could be made a constant-time operation having the ability to broadcast this information to all the slaves at the same time.
2. during *gather* the team master sequentially checks that all slaves have written into NFLAGS. This could be made a constant-time operation having the ability to put the team master in sleep and notify it when all slaves have joined.
3. during *fetch* the team master i) sequentially selects slaves to recruit by inspecting their status, then ii) points them to the team descriptor by writing the address into each slave's field of the TEAM_DESC_PTR array. ii) could also be made a constant-time operation if the broadcast mechanism mentioned above allowed 32-bit word broadcast.

To this aim, we enhance our simulation infrastructure with a *hardware syn-*

chronizer (HWS) block that implements the discussed features. The HWS is implemented as a functional SystemC module annotated with timing information extracted from a HW implementation based on our previous work [14]. Each cluster in the system integrates a HWS block, which can be configured via memory-mapped registers to broadcast signals (or one 32-bit word) to a set of processor in a cluster, identified by a bitmask. Hierarchically interconnected HWS blocks allow inter-cluster synchronization.

The rightmost plot in Figure 10 shows the execution time scaling for the most critical parts of *fork* and *join* using the hardware-accelerated primitives. The HWS allows to make release and gather constant-time operations, comparable to the cost of SW primitives for 4 threads. The word-broadcast feature allows to speed up thread fetch by ≈13% on the fast on-cluster interconnection considered in this work. This value would significantly increase if a slower interconnection medium was considered (e.g., a NoC).

We obtain the results shown in Figure 11 for the HW-accelerated nested parallelism support. Comparing to Figure 8, the HWS allows a net reduction of ≈10% and ≈28% of the *fork* and *join* time, respectively. Moreover, the HWS allows perfectly aligned start time of the nested teams on various clusters, which has a significant impact on overall parallel region duration.

Function call overhead (time spent invoking primitives for fork and join) and *inner team init* (time spent to allocate memory for the inner team descriptor and populate it) are two important contributors to overall fork/join cost. In the common case where the goal is to spawn a parallel region that involves all the cores in the system, we can avoid those costs. In fact, all the threads/cores need to be pointed to a unique team descriptor and destroyed jointly. To that aim we provide dedicated functions to transparently synchronize threads and clusters in a hierarchical manner, without the need for explicit calls to outer level and inner level parallelism creation functions. Figure 12 shows the fork/join cost when these functions are used. Overall, a net reduction of ≈37% and ≈36% of the *fork* and *join* time, respectively, is achieved compared to SW.

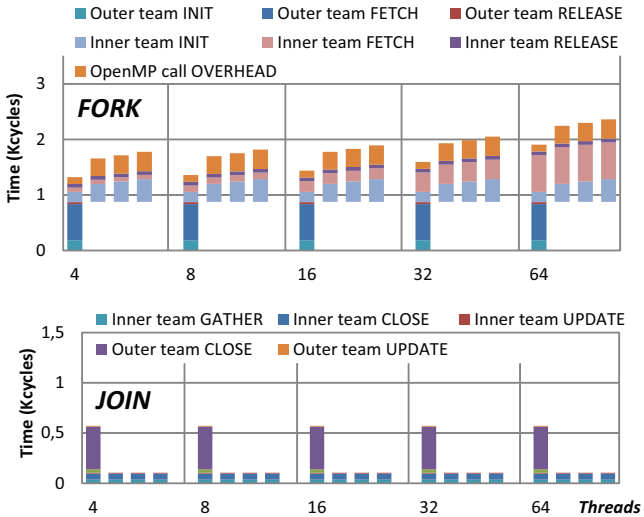

Figure 11: Execution cycles scaling for explicit nesting with HW support

3.2.2. NUMA-aware nested parallelism in OpenMP

Due to the relevance of affinity control in the context of ccNUMA machines, the OpenMP architecture review board has included in the recent specification v4.0 the definition of a new `proc_bind` construct, to be coupled to the `parallel` directive.

```
proc_bind ( master | close | spread )
```

The `master` policy assigns every thread in the team to the same place as the master thread. The `close` policy assigns the threads to places close to the place of the parent's thread. The master thread executes on the parent's place and the remaining threads in the team execute on places from the place list consecutive from the parent's position in the list, with wrap around with respect to the place list. The `spread` policy creates a sparse distribution for a team of T threads among the P places of the parent's place partition. It accomplishes this by first subdividing the parent partition into T subpartitions if T is less than or equal to P, or P subpartitions if T is greater than P. Then it assigns 1 ($T \leq P$) or a set of threads ($T > P$) to each subpartition. The subpartitioning is not only a

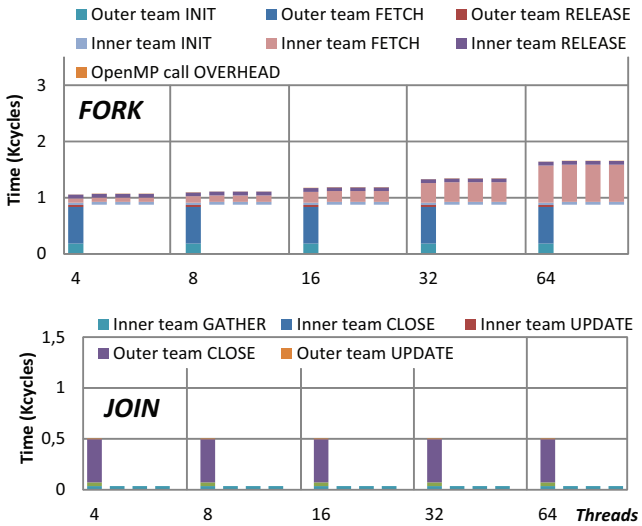

Figure 12: Execution cycles scaling for implicit nesting with HW support

mechanism for achieving a sparse distribution, it is also a subset of places for a thread to use when creating a nested parallel region.

We believe that such an extension could also be very useful to mitigate NUMA effects within a cluster-based manycore SoC with explicitly managed memory hierarchy. Thus, we implemented the proposed extension and integrated our framework for lightweight nested fork/join in the OpenMP runtime library. In Section 4 we describe how this simple extension allows to control nested thread management so as to achieve regular data behavior and to extract high degrees of fine-grained parallelism.

3.2.3. Multi-level nesting

In the previous sections we have discussed the optimization of the support for two-level nested parallelism, which is the most common case for deploying computation with high data locality in out target system. However, our framework is capable of supporting multiple levels of parallelism nesting. In this section we use the EEPC benchmarks [15] to characterize the cost of nesting up to 5 OpenMP parallel regions. The original methodology has been extended

to account for nested parallel regions as described in [16]. This methodology basically computes runtime overheads by subtracting the execution time of the parallel microbenchmark from the execution time of its reference sequential implementation. The parallel benchmark is constructed in such a way that it would have the same duration of the reference in absence of overheads.

In Fig. 13 we show the task graph representation of the microbenchmarks used to assess the cost of nested parallelism with depth 1 and 2, as an example. The computational kernel (indicated as W in the plots) is composed uniquely

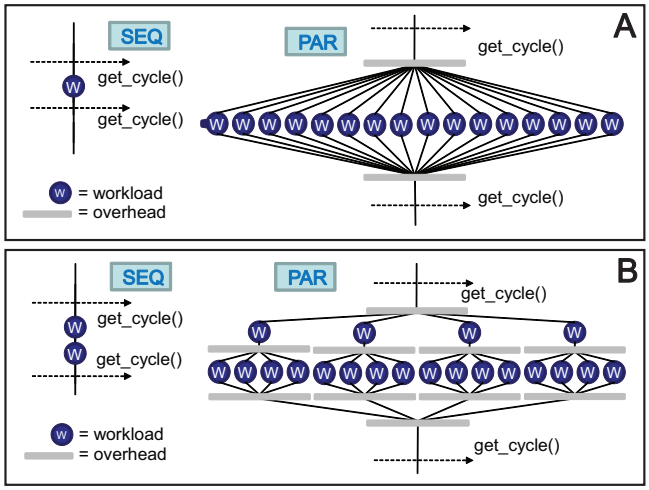

Figure 13: EEPC microbenchmark for nested parallelism overhead assessment. A) 1 level, B) 2 levels

of ALU instructions, to prevent memory effects from altering the measure. We consider a simple pattern where a parallel region is opened, then the block W is executed. This pattern is nested up to 5 times. The thick gray lines in our plots represent the sources of overhead that we intend to measure.

The difference between the parallel and sequential versions of the microbenchmark represents the total overhead for opening and closing as many parallel regions as the nesting depth indicates.

Figure 14 shows this overhead for varying granularities of the work unit (W). The upper plot refers to VirtualSoC, the bottom plot to STHORM. There are

as many curves as the considered levels of nesting.

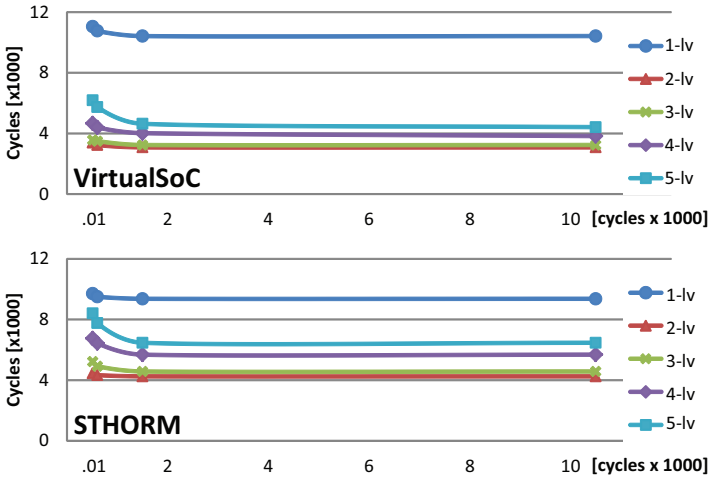

Figure 14: Cost of multi-level nested parallelism

The total number of threads created for each experiment is always 64 (all the processors in the system are involved in parallel computation). For example, the curve marked as **1-lv** refers to the experiment where we create a single parallel region composed of 64 threads. The **2-lv** experiment considers two nested parallel regions with 4 *spread* threads on the first level and 16 *close* threads on the second. The **5-lv** experiment considers an outermost parallel regions with 4 *spread* threads and 4 nested parallel regions composed of 2 thread each.

Using NUMA-aware nested parallelism is always faster than single-level parallelism in cluster-based architectures. As we already discussed in Section 3.2, this is expected, since single-level parallelism creation beyond a single cluster involves a significant number of remote NUMA memory transactions. When the granularity of the parallel workload is very small (tens to few hundreds of cycles) the cost for nested parallelism creation has a slightly higher overhead, mostly due to contention for shared data structures (the accesses to these structures from multiple masters trying to concurrently create additional parallelism are sequentialized). However, for workload granularities in the order of thousand

cycles and above these overheads are fully amortized.

With respect to VirtualSoC, the prototype STHORM implementation has slightly higher cost for multi-level nested parallelism support. As already mentioned previously, this is largely due to the lack of optimization for on-chip memory allocation primitives. The STHORM SDK provides centralized memory allocation services (i.e., requests for memory allocation from multiple masters are diverted to a single cluster controller, which services the requests in a FIFO manner). This implies that most of the initialization phases in our nested parallelism support library have bigger fixed (i.e., independent of the size of the parallel region) costs on STHORM. These costs become relevant when the size of the thread team being created is small.

3.3. Nested parallelism support cost scaling

Table 2 summarizes how the cost for a fork/join operation for different approaches scales with the number of cores involved. Flat parallelism scales linearly with the number of cores in the platform; for 64 cores fork/join cost reaches 8.5KCycles. Nested fork/join shows better scalability considering that a part of the computational cost to recruit/park threads is parallelized among different clusters. The cost thus increases linearly with the number of clusters plus the number of cores per cluster, rather than with the total number of cores.

Pattern	**Scaling**	**Number of cores**					
		2	**4**	**8**	**16**	**32**	**64**
Flat	$O(Cores)$	0.4	0.5	0.9	1.7	3.6	8.5
Nested	$O(Clusters + \frac{Cores}{Clusters})$	1.6	2.3	2.3	2.5	2.9	3.4
Nested HW	$O(Clusters + \frac{Cores}{Clusters})$	1.7	2.3	2.3	2.4	2.6	2.9
Implicit Nested HW	$O(Clusters + \frac{Cores}{Clusters})$	1.1	1.5	1.6	1.7	1.8	2.1

Table 2: Fork/Join cost [KCycles] scaling increasing the number of cores using different parallel patterns.

4. Benchmarks

In this section we validate our nested parallelism support runtime for NUMA embedded manycores using six benchmarks (summarized in Table 4) from the computer vision, image processing and linear algebra domain, typically targeted by the manycore accelerators considered in this work. Such applications employ a regular computation and memory access structure, but deploying the parallel workload on all the available cores with no awareness of the clustered platform organization (referred to as "*flat*" parallelization) leads to varying execution times for nominally identical threads. This irregular behavior is consistently observed for every benchmark, due to the OpenMP memory model and lack of NUMA-awareness in the *flat* parallelization scheme.

FAST	Corner detector
CT	Object tracking based on a specific color
Mahalanobis	Mahalanobis distance between two point clouds
Strassen	Matrix multiplication using Strassen decomposition
NCC	Normalized cross-correlation algorithm
SHOT	3D descriptor for surface matching. Two main kernels: 1) local reference frame radius; 2) histogram interpolation

Table 3: Benchmarks

In the following, we first provide details about the various parallelization schemes used in the evaluation, using the Color Tracking application as an example. Second, we show the speedup achieved by all the benchmarks when various approaches are adopted to deploy parallelism over the whole manycore platform.

4.1. Parallelization Patterns

To parallelize the six target benchmarks we have used a couple of patterns, enabled by the availability of NUMA-aware nested parallelism support. As an

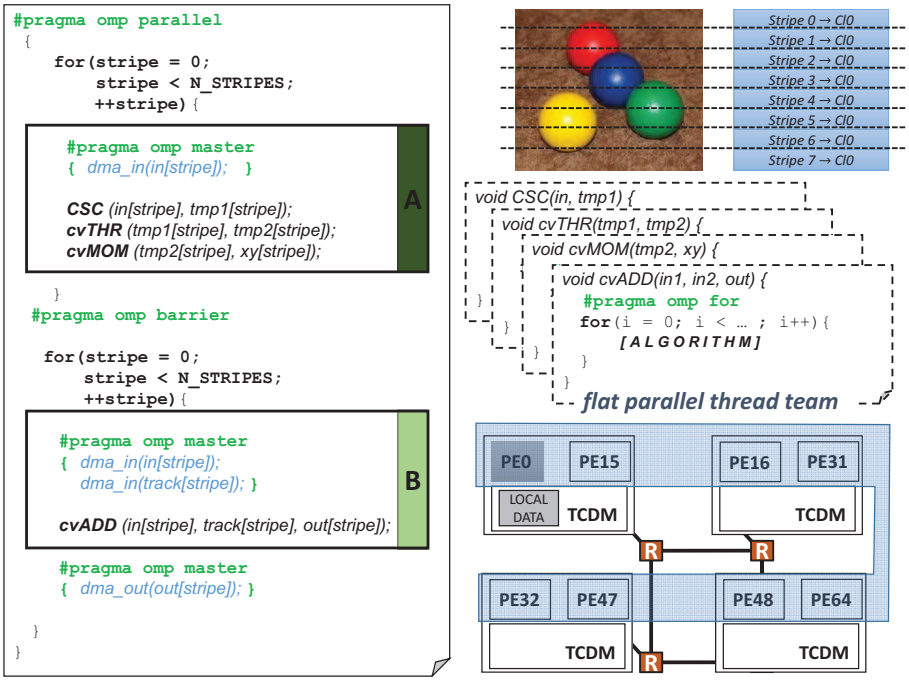

Figure 15: Flat parallelization scheme.

example, we illustrate in the following how we have partitioned and parallelized Color Tracking with the various schemes.

Color-based tracking consists of a cascade of four functional kernels. Color space conversion (`CSC`), threshold-based color filter (`cvTHR`), motion vector calculation (`cvMOM`) and motion vector to reference frame addition (`cvADD`).

Input and output frames are stored in the main memory, as well as the temporary output buffers for every kernel. To improve locality of computation, data must be moved to TCDMs using the DMA engine. To achieve efficient data transfers we use standard double buffering techniques. The input image is split in several stripes; while one stripe is being processed the next one can be pre-loaded to the TCDM. The same mechanism is used for output data. The size of stripes is an important parameter to achieve efficiency, and strictly depends of the parallelization strategy.

4.1.1. Flat Parallelization

In the *flat* parallelization scheme only one single level of parallelism is created, i.e., only one parallel thread team. Logically, we are abstracting the platform as a flat (i.e., assumed homogeneous computing and memory resources) team of 64 threads, headed by the *master* thread mapped on PE 0 within cluster 0. As the code snippet in Figure 15 shows, the *master* thread is responsible for bringing in and out data from the main memory into the local TCDM (DMA primitives are enclosed within a **#pragma omp master** directive). However, since the parallel team spans multiple clusters, threads belonging to clusters 1, 2 and 3 will experience longer memory access (the corresponding transactions are transported through the NoC).

4.1.2. Nested data parallelization

The second recurrent parallelization pattern in our application kernels distributes single-program, multiple-data computation all over the available cores in the system. Figure 16 shows the pseudo code for the data-parallelization pattern. A first level of parallelism creates as many threads as clusters. Associating the `proc_bind` clause to this parallel region ensures that the four threads are mapped on different clusters (local masters). Data parallelism is implemented at the stripe level within each cluster by exploiting a second level of parallelism. To improve the computation to communication ratio (CCR) we merge the `CSC`, `cvThresh` and `cvMOM` kernels into a single kernel. As already explained, `cvADD` can not be merged with the previous kernels because it requires as an input the motion vectors for the whole image. A barrier is required between the two nested parallel regions, since the barrier implied at their end would only synchronize threads within each cluster independently (no inter-cluster synchronization).

Again, if the `proc_bind` clauses were not used, the composition of the nested teams would still span multiple clusters and NUMA effects would still be present.

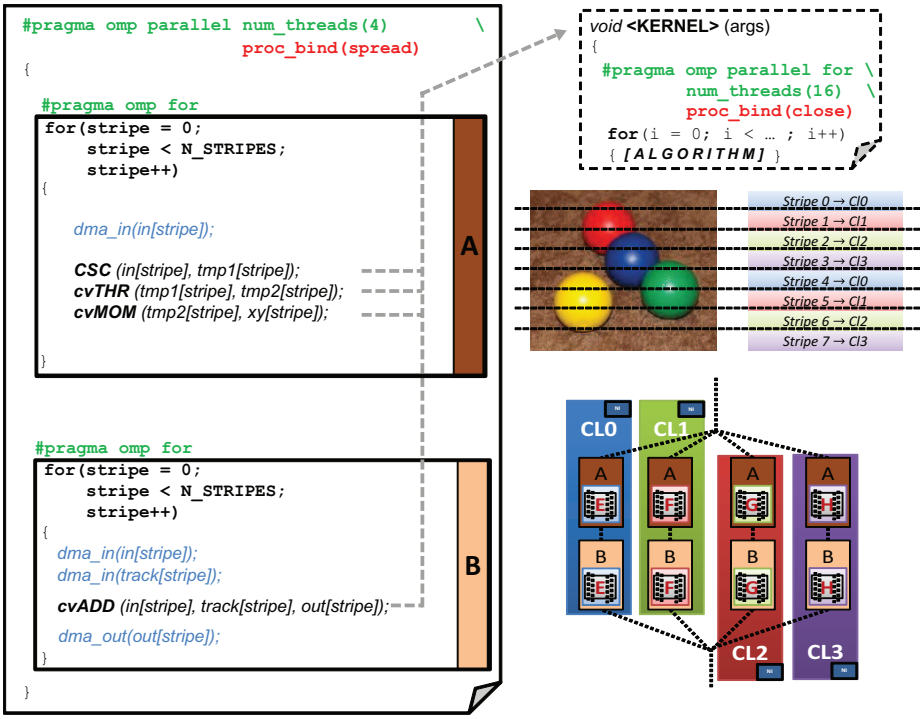

Figure 16: Nested data parallel color tracking.

4.2. Evaluation of Nested Parallelism Support

In this Section we evaluate the effectiveness of our nested parallelism support, comparing the performance of the various presented policies to spawn parallelism throughout the whole platform:

1. **Flat** - A single parallel region of 64 threads is created (no nesting);
2. **Nested (non-NUMA)** - Two nested parallel regions are created, but with no use of the `proc_bind` clause (no NUMA awareness);
3. **Nested (NUMA)** - Two nested parallel regions are created, using the `proc_bind` clause (NUMA-aware nesting);
4. **Nested HW (NUMA)** - Same as before, with HW-accelerated nesting support.

Note that all the policies are evaluated on the same lightweight implementation presented in Section 3, so the focus here is on the effect of NUMA-aware core

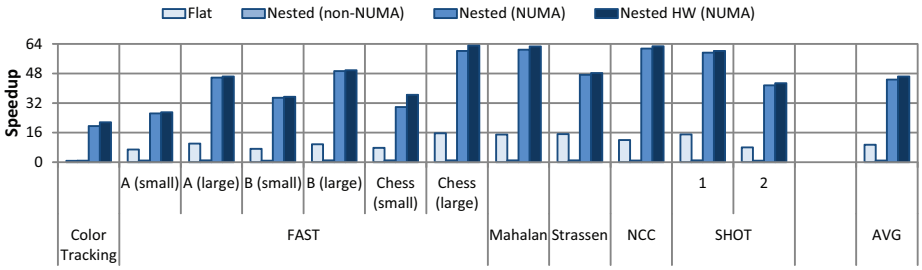

Figure 17: Comparison of various approaches to nested parallelism support.

binding[2]. As a main metric of performance we consider speedup of the parallel application versus the sequential.

Results for this experiment are shown in Figure 17. The flat parallelization scheme, as expected, severely limits the maximum achievable speedup, due to irregular memory behavior among nominally identical threads. It is interesting to note that NUMA-unaware nesting can exhacerbate this irregularity and achieve poorer locality that the flat scheme. Indeed, besides poor data locality, in this case we are sistematically enforcing costly inter-cluster communication due to thread management (i.e., implied by fork/join of parallel regions spanning multiple clusters). This confirms that the ability of creating nested parallelism alone is not sufficient to achieve good performance, if it is not augmented with NUMA-awareness. When nesting is made NUMA-aware we can achieve up to 63× speedup (46× on average). This solution can get up to 28× faster than flat parallelism (for Color Tracking, 7× on average). HW-accelerated nesting improves SW-only nesting by ≈20% for very fine-grained and short-running workloads (FAST, small images).

Some benchmarks leverage very fine-grained parallelization, for which the overhead introduced by runtime support for nested parallelism has a higher impact. This is the case of FAST [17]. FAST is a corner detection algorithm, which operates by comparing the intensity value of a target image point px with all the surrounding pixels in a circular area. px is classified as a corner if there

[2]For the effect of NUMA-aware thread management see the previous Section 3.2.

exists a set of contiguous pixels within the circle that are all brighter (minimum) or darker (maximum) of px (within a tolerance threshold). The parallelization pattern adopted here is the same already shown in Figure 16, but in this case only one parallel region is required. The granularity of the workload distributed to parallel threads in FAST depends of two parameters: i) overall duration of the computation and ii) corner density (number of corners detected). To allow studying the impact of these factors on the overall speedup we perform experiments on two types of images. The first is a chess pattern, which we use as a sort of synthetic use-case, useful to understand the scalability of the algorithm when increasing the size of the input image. We consider the following image sizes: 32×32, 64×64, 128×128, 256×256 and 512×512 pixels. For this type of image the corner density is 15%. The ratio between the number of corners and the total number of pixels remains constant when scaling the image, but the amount of processed pixels increases, which has an effect on the granularity of the parallel work, and – consequently – on the parallelization overhead. The second type of image is a real urban traffic scene, representative of what could be captured by a camera on a driver assistance system, showing the road and cars and buildings on the background. Typically, these real-life images have much lower corner density. We consider two real images with corner density 1,5% and 6%, respectively, in two sizes: small (320×240) and large (640×480).

In Figure 18 we show the execution time and speedup for the experiment with the synthetic image pattern when increasing the input image size. We show normalized execution cycles (bars, left Y-axis) and speedup (lines, right Y-axis) for HW-accelerated nested parallelism versus sequential execution. For image sizes around 256×256 the speedup gets closer to the ideal one (≈60×).

Figure 19 shows the results for the two real images. Image A (≈ 1.5% corner density) reaches up to 27× and 46× speedups for small and large images, respectively. Image B (≈ 6% corner density) reaches 35× and 50× speedups.

Since the computation time varies depending on whether the current pixel is a keypoint or not, and being the keypoints clustered in specific regions of the image, some load imbalance between parallel threads is present. This is shown

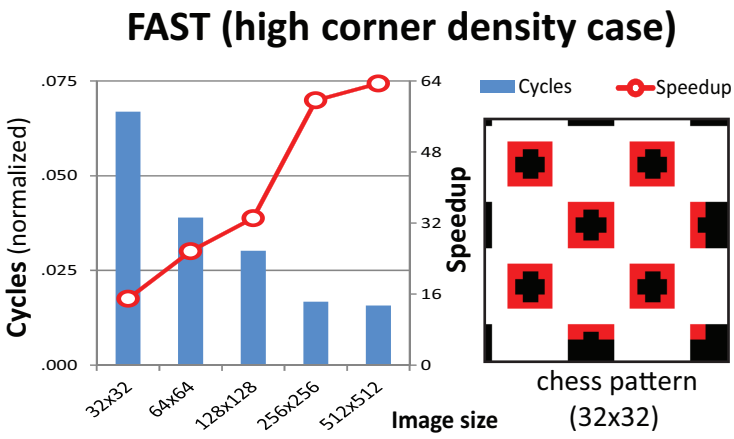

Figure 18: FAST performance for Chess pattern images.

in the bottom part of Figure 19, where we indicate the variance in execution time among threads.

Overall, the results demonstrate that our nested parallelism support layer is capable of extracting high degrees of parallelism even for very fine-grained workloads.

4.3. Impact of fork/join on application scalability

In Table 2, at the end of Section 3, we have shown the fork/join cost scaling for our nested parallelism support. The way this impacts the speedup of real applications depends of the granularity of the parallel workload. As a last experiment, we have measured the duration of parallel regions in each benchmark for increasing core count and matched the numbers to those reported in Table 2. Figure 20 shows the impact of the fork/join overhead on real workloads for various core counts when using HW-accelerated nested parallelism. In general, reducing the number of cores i) reduces fork/join cost; ii) increases the granularity of parallel regions (the same amount of work is shared between less workers). Thus, the results for parallelization over 64 cores (what we have shown previously in this section) represent the worst case in terms of overhead impact. Even in this case, it is possible to see that fork/join overhead is always

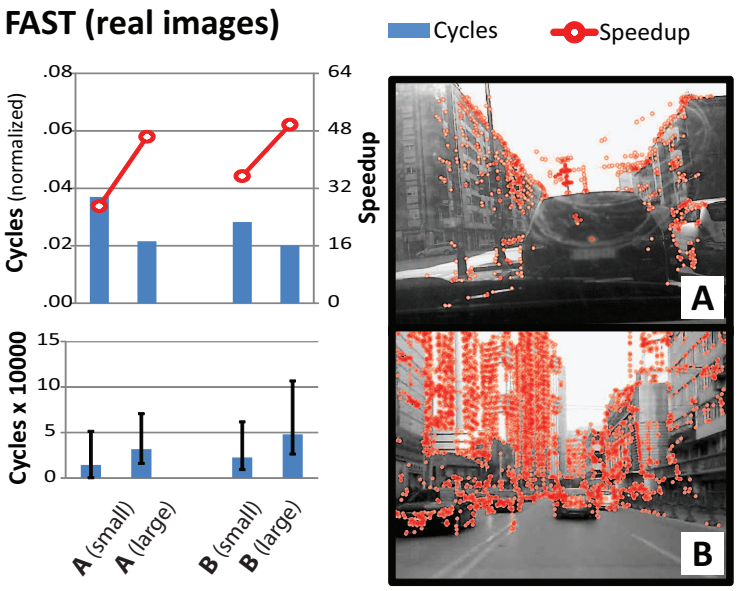

Figure 19: FAST performance for real images.

negligible.

In conclusion, in terms of application scalability for all the benchmarks considered in this work fork/join does not impact at all the scalability of the employed parallelization scheme.

5. Related Work

There are two main research areas related to the work presented in this paper: support for scalable thread fork/join in large systems considering multi-level parallelism, and management of fork/join parallelism in NUMA systems. We describe related work in the two areas in separate sections.

5.1. Nested Parallelism Support

Nested parallelism can be implemented in different ways [23] [24] [25] [26] [27]. In the literature many techniques exist, which can be categorized into two main approaches:

	core count					
	2	4	8	16	32	64
Color Tracking	0,02%	0,06%	0,12%	0,26%	0,61%	1,43%
FAST	0,00%	0,00%	0,00%	0,01%	0,02%	0,05%
Mahalanobis	0,00%	0,01%	0,01%	0,03%	0,06%	0,14%
Strassen	0,00%	0,00%	0,00%	0,01%	0,01%	0,04%
NCC	0,00%	0,00%	0,01%	0,02%	0,04%	0,10%
SHOT1	0,00%	0,01%	0,02%	0,03%	0,08%	0,18%
SHOT2	0,01%	0,02%	0,04%	0,10%	0,22%	0,52%

Figure 20: Impact of fork/join overhead for real benchmark as the number of cores is scaled.

Dynamic thread creation (**DTC**): whenever the application asks for additional parallelism, it is mapped on a lightweight thread from some standard package (e.g., *pthreads*). This approach allows very flexible creation of parallelism as needed, but it is very expensive [18] [20] [21]. Table 4 shows the cost (in cycles) to fork and join a thread team for different approaches. The last column of the table shows normalized cycles for 16 cores, to allow for a direct comparison to our own solution. The first four rows show several implementations that realy on DTC. On average this approach has ≈32× higher overheads compared to us (and up to ≈113×).

Fixed thread pool (**FTP**): A fixed number of lightweight threads (typically as many as the number of processors) is created at system startup and constitute a fixed pool of workers. When a program requests the creation of parallelism, threads are fetched from the pool [28] [22] [19]. If the number of logical threads created at an outermost parallel construct is less than the number of threads in the pool, some of them will be left unutilized and available for nested parallelism. The last four rows of Table 4 show fork/join cost for FTP solutions published in literature. While being much faster than DTC, state-of-the-art FTP solutions have on average ≈6× higher overheads compared to us (and up to 14×).

Ref.	Architecture	Kind	#Cores	Fork/Join Cost (KCycles)	Projected 64-core Cost (KCycles)
[18]	Intel Xeon Phi	DTP	240	≈1700	**453.6**
[19]	Intel Xeon X5650 GCC	DTP	8	≈5.7	**45.6**
	Intel Xeon X5650 ICC		8	≈4.3	**34.4**
	Samsung Exynos 4412		4	≈2.4	**38.4**
	TI ARMA15 CorePac		4	≈3.4	**54.4**
[20]	Intel Xeon X5355	DTP	4	≈9.3	**148.8**
[21]	IBM Cyclops-64	FTP	160	≈30	**12**
[22]	STM STHORM	FTP	16	≈1.5	**6**
[19]	TI c66x DSP	FTP	8	≈7.1	**56.8**
	This work	FTP	64	≈4	**4**

Table 4: Fork/Join cost and projected cost for 64 cores for several implementations [KCycles].

There also are many hybrid approaches, which combine in some ways DTC and FTP. Some techniques start with a FTP approach, and dynamically create new threads when there are no idle workers on the pool [21]. Other solutions leverage thread creation at the outermost level of parallelism, where the computation is assumed to be coarse enough to amortize the overhead, and a simple work descriptor shared by threads at the innermost level of parallelism [23] [29].

The work from Tanaka et al. [27] relies on a fixed thread pool, but allows multiple logical threads to be mapped on a single physical thread and maintains a work queue from which threads which become idle can fetch (or steal) work. The latter approach is based on the widely adopted abstraction of a work queue [30][31], and is an orthogonal technique to nesting. OpenMP itself, since specification v3.0, provides tasks or dynamic loop scheduling, also based on the notion of a work queue, which allow to specify work units at a finer granularity than threads. In these programming models, once a thread team has been defined,

to extract more parallelism it is not necessary to create additional threads: the more lightweight abstraction of the work queue allows existing threads to push and fetch work from there. This offers in many situations a more flexible means to creating parallelism than that offered by nesting alone.

However, while work queues allow very flexible parallelism creation, they do not support the logical clustering of threads in the multilevel structure, which is key to achieving data locality and balancing of static workload partitioning. When considering the cluster-based design of our target architecture, the capability of confining a thread team within the boundaries of a cluster is key to achieve locality and balancing. We thus believe that a lightweight support for the creation of nested thread teams is fundamental to enabling fine-grained parallelism. In this paper we present our streamlined and optimized implementation of nested parallelism. Work queue-based parallelism can orthogonally be provided within our support.

5.2. Thread Affinity Control for NUMA Systems

Thread binding and affinity are major concerns on NUMA architectures, and in literature different approaches and programming model extensions exist to deal with this issue. OpenMP is a powerful and easy-to-use programming model for shared memory multiprocessors, but it has no awareness of the underlying memory system organization. Early solutions to this problem were offered inside specific software development environments. All these solutions use core identifiers and environment variables to specify the binding between cores and threads. GNU and Intel compilers provide environment variables (`GOMP_CPU_AFFINITY` and `KMP_AFFINITY`) to specify a list of CPUs to which to bind threads. These variables enumerate a set or a range of core IDs where the threads are allowed to be placed. The Intel compiler also provides two specifiers: `scatter` and `compact`, which define how the threads must be allocated to cores. This is similar to the OpenMP extension that we consider in this paper, but it works well only for a single level of parallelism, because the thread binding policy cannot be changed at runtime. Moreover, thread to processor

binding ultimately relies on costly operating system primitives such as linux `sched_setaffinity`, which can not be used on the manycore systems targeted in this paper, for two reasons. First, the lack of full-fledged operating systems. Second, the necessity of supporting very fine-grained parallel workloads, which can not tolerate high-overheads for parallelism creation. The PGI compiler [32] enables thread binding via the `MP_BIND` variable. The user specifies on a second variable (`MP_BLIST`) the core list where the threads can be allocated.

Extensions to the Intel compiler (the *subscatter* and *subcompact* policies) have been proposed to manage thread binding for nested parallel regions [33]. However, the bind mechanism is still based on environment variables, which makes it difficult to use and to change at runtime.

A more generic approach extends the standard processor GROUP to represent complex hierarchical memory architectures and allows the programmer to assign work to these groups [34]. The main limitation of this solution is that it puts on the programmer the burden of in-depth hardware knowledge and exploitation.

ForestGOMP[35] introduces a different notion of thread groups, called *bubbles*. These bubbles can have a hierarchical structure to describe a nesting relation. A scheduler (BUBBLESCHED) assigns the threads to specific cores of the system taking NUMA concerns into account, then a thread stealing mechanism allows to change the mapping and migrate threads as necessary. A disadvantage of this approach is that it is hard for the programmer to understand what the scheduler does, and thus to optimize the code.

A recent work from Eichenberger et al. [36] tries to put together previous approaches in a more generic, portable and flexible way. Two basic concepts are defined: *places* and *affinity*. The first describes the platform topology and memory hierarchy, defining a set of places where the threads can be allocated; the second allows to implement different allocation patterns throughout the places: *spread* maximizes the distance between places and *compact* puts all threads in a single place.

We implement the affinity control directives proposed in OpenMP v4.0, and

integrate our lightweight support to nested parallelism in the runtime library.

6. Conclusion

To scale to the manycore paradigm several recent embedded MPSoCs have been architected as fabrics of tightly-coupled, shared memory clusters. Key to extract the massive peak parallelism offered by these systems is the availability of an easy-to-use yet powerful programming model and associated runtime layer. When considering the computing systems at hand, two main concerns arise. First, since the target platform is typically meant to run very fine-grained parallel workloads, it is fundamental to provide very lightweight primitives to create and manage parallelism over a very large number of cores. Second, since cluster-based manycores feature NUMA memory architectures, the runtime system and the programming model should be made aware of this hardware peculiarity to prevent scalability bottlenecks and performance blockers.

Nested parallelism provides an intuitive conceptual framework to address the second point, provided that i) an efficient implementation of the first is available and ii) the capability of binding thread teams to specific cores and clusters is provided. In this paper, we have presented an efficient runtime layer for nested parallelism on cluster-based embedded manycores, identifying the most critical operations to fork and join nested parallelism, and proposing SW-only and HW-accelerated solutions for their implementation. Our fork/join primitives have been integrated in the OpenMP programming model, and the associated compiler implements an extension to expose an abstract notion of clusters at the programming interface level, which makes nested parallelism mapping NUMA-aware.

This extended OpenMP interface allowed us to explore on a set of real application use cases how NUMA affects the performance of flat parallelism, and how our approach provides control over such effects and achieves up to 28× speedup versus flat parallelism. In terms of fork/join cost, our solution scales better than the original flat approach, as it is a function of the number of clus-

ters (plus the number of cores in a single cluster) rather than the total number of cores. In terms of application scalability, for all the benchmarks considered in this work the impact of fork/join is always negligible and does not affect at all the scalability of the employed parallelization scheme.

Acknowledgements

This work has been supported by EU FP7 project P-SOCRATES (contract number: 611016).

References

[1] A. Munir, S. Ranka, A. Gordon-Ross, High-performance energy-efficient multicore embedded computing, Parallel and Distributed Systems, IEEE Transactions on 23 (4) (2012) 684–700.

[2] J. Diaz, C. Munoz-Caro, A. Nino, A survey of parallel programming models and tools in the multi and many-core era, Parallel and Distributed Systems, IEEE Transactions on 23 (8) (2012) 1369–1386.

[3] Nvidia Inc., NVIDIA Tegra X1 - NVIDIA'S New Mobile Superchip.

[4] Kalray, MPPA 256 - Programmable Manycore Processor, `www.kalray.eu/products/mppa-manycore/mppa-256/`.

[5] PEZY Computing, PEZY-SC Many Core Processor, http://www.pezy.co.jp/en/products/pezy-sc.html.

[6] D. Melpignano, L. Benini, E. Flamand, B. Jego, T. Lepley, G. Haugou, F. Clermidy, D. Dutoit, Platform 2012, a many-core computing accelerator for embedded SoCs: performance evaluation of visual analytics applications, in: Proceedings of the 49th Annual Design Automation Conference, ACM, 2012, pp. 1137–1142.

[7] OpenMP, ARB, OpenMP application program interface, v. 4.0, no. July, 2013.

[8] J. Joven, A. Marongiu, F. Angiolini, L. Benini, G. De Micheli, An integrated, programming model-driven framework for NoC–QoS support in cluster-based embedded many-cores, Parallel Computing 39 (10) (2013) 549–566.

[9] G. Mitra, E. Stotzer, A. Jayaraj, A. P. Rendell, Implementation and optimization of the OpenMP accelerator model for the TI Keystone II architecture, in: Using and Improving OpenMP for Devices, Tasks, and More, Springer, 2014, pp. 202–214.

[10] B. Chapman, L. Huang, E. Biscondi, E. Stotzer, A. Shrivastava, A. Gatherer, Implementing OpenMP on a high performance embedded multicore MPSoC, in: Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on, IEEE, 2009, pp. 1–8.

[11] A. Marongiu, P. Burgio, L. Benini, Fast and lightweight support for nested parallelism on cluster-based embedded many-cores, in: Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012, IEEE, 2012, pp. 105–110.

[12] H. Xu, J. Tanabe, H. Usui, S. Hosoda, T. Sano, K. Yamamoto, T. Kodaka, N. Nonogaki, N. Ozaki, T. Miyamori, A low power many-core SoC with two 32-core clusters connected by tree based NoC for multimedia applications, in: VLSI Circuits (VLSIC), 2012 Symposium on, 2012, pp. 150–151.

[13] D. Bortolotti, C. Pinto, A. Marongiu, M. Ruggiero, L. Benini, VirtualSoC: A Full-System Simulation Environment for Massively Parallel Heterogeneous System-on-Chip, in: IPDPS Workshops, 2013, pp. 2182–2187.

[14] J. L. Abellán, J. Fernández, M. E. Acacio, D. Bertozzi, D. Bortolotti, A. Marongiu, L. Benini, Design of a collective communication infrastructure for barrier synchronization in cluster-based nanoscale MPSoCs, in: Proceedings of the Conference on Design, Automation and Test in Europe, ACM, 2012, pp. 491–496.

[15] J. M. Bull, D. O'Neill, A microbenchmark suite for OpenMP 2.0, ACM SIGARCH Computer Architecture News 29 (5) (2001) 41–48.

[16] V. V. Dimakopoulos, P. E. Hadjidoukas, G. C. Philos, A microbenchmark study of OpenMP overheads under nested parallelism, in: OpenMP in a New Era of Parallelism, Springer, 2008, pp. 1–12.

[17] E. Rosten, R. Porter, T. Drummond, Faster and better: A machine learning approach to corner detection, Pattern Analysis and Machine Intelligence, IEEE Transactions on 32 (1) (2010) 105–119.

[18] D. Schmidl, T. Cramer, S. Wienke, C. Terboven, M. S. Müller, Assessing the performance of OpenMP programs on the Intel Xeon Phi, in: Euro-Par 2013 Parallel Processing, Springer, 2013, pp. 547–558.

[19] E. Stotzer, A. Jayaraj, M. Ali, A. Friedmann, G. Mitra, A. P. Rendell, I. Lintault, Openmp on the low-power ti keystone ii arm/dsp system-on-chip, in: OpenMP in the Era of Low Power Devices and Accelerators, Springer, 2013, pp. 114–127.

[20] P. E. Hadjidoukas, G. C. Philos, V. Dimakopoulos, Exploiting fine-grain thread parallelism on multicore architectures, Scientific Programming 17 (4) (2009) 309–323.

[21] J. del Cuvillo, W. Zhu, G. Gao, Landing openMP on Cyclops-64: An efficient mapping of openmp to a many-core system-on-a-chip, in: Proceedings of the 3rd conference on Computing frontiers, ACM, 2006, pp. 41–50.

[22] Y. Lhuillier, M. Ojail, A. Guerre, J.-M. Philippe, K. B. Chehida, F. Thabet, C. Andriamisaina, C. Jaber, R. David, Hars: a hardware-assisted runtime software for embedded many-core architectures, ACM Transactions on Embedded Computing Systems (TECS) 13 (3s) (2014) 102.

[23] E. Ayguade, X. Martorell, J. Labarta, M. Gonzalez, N. Navarro, Exploiting multiple levels of parallelism in OpenMP: a case study, in: Parallel

Processing, 1999. Proceedings. 1999 International Conference on, 1999, pp. 172 –180.

[24] S. Karlsson, A portable and efficient thread library for OpenMP, in: In Proc. 6th European Workshop on OpenMP, KTH Royal Institute of Technology, John Wiley, 2004, pp. 43–47.

[25] X. Martorell, E. Ayguadé, N. Navarro, J. Corbalán, M. González, J. Labarta, Thread fork/join techniques for multi-level parallelism exploitation in NUMA multiprocessors, in: Proceedings of the 13th international conference on Supercomputing, ACM, 1999, pp. 294–301.

[26] P. E. Hadjidoukas, V. V. Dimakopoulos, Nested parallelism in the ompi openp/c compiler, in: Euro-Par 2007 Parallel Processing, Springer, 2007, pp. 662–671.

[27] Y. Tanaka, K. Taura, M. Sato, A. Yonezawa, Performance evaluation of openmp applications with nested parallelism, in: Languages, Compilers, and Run-Time Systems for Scalable Computers, Springer, 2000, pp. 100–112.

[28] S. N. Agathos, V. V. Dimakopoulos, A. Mourelis, A. Papadogiannakis, Deploying OpenMP on an embedded multicore accelerator, in: Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 International Conference on, IEEE, 2013, pp. 180–187.

[29] M. Gonzalez, J. Oliver, X. Martorell, E. Ayguade, J. Labarta, N. Navarro, Openmp extensions for thread groups and their run-time support, in: Languages and Compilers for Parallel Computing, Springer, 2001, pp. 324–338.

[30] N. Brookwood, AMD fusion family of APUS: enabling a superior, immersive pc experience, Insight 64 (1) (2010) 1–8.

[31] V. Nahavandipoor, Concurrent Programming in Mac OS X and iOS: Unleash Multicore Performance with Grand Central Dispatch, " O'Reilly Media, Inc.", 2011.

[32] The Portland Group, PGI Compiler User Guide, http://www.pgroup.com/doc/pgiug.pdf.

[33] D. Schmidl, C. Terboven, D. an Mey, M. Bücker, Binding nested OpenMP programs on hierarchical memory architectures, in: Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More, Springer, 2010, pp. 29–42.

[34] G. Zhang, Extending the openmp standard for thread mapping and grouping, in: OpenMP Shared Memory Parallel Programming, Springer, 2008, pp. 435–446.

[35] F. Broquedis, N. Furmento, B. Goglin, P.-A. Wacrenier, R. Namyst, Forest-GOMP: An Efficient OpenMP Environment for NUMA Architectures, International Journal of Parallel Programming 38 (5-6) (2010) 418–439.

[36] A. E. Eichenberger, C. Terboven, M. Wong, D. an Mey, The design of OpenMP thread affinity, in: Proceedings of the 8th international conference on OpenMP in a Heterogeneous World, IWOMP'12, Springer-Verlag, 2012, pp. 15–28.