Fast and Lightweight Support for Nested Parallelism on Cluster-Based Embedded Many-Cores

Andrea Marongiu, Paolo Burgio, Luca Benini
DEIS – University of Bologna
Viale Risorgimento 2, 40136 Bologna – Italy
Email: {a.marongiu, paolo.burgio, luca.benini}@unibo.it

***Abstract*—Several recent many-core accelerators have been architected 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 non-uniform (NUMA). Nested parallelism represents a powerful programming abstraction for these architectures, where a first level of parallelism can be used to distribute coarse-grained tasks to clusters, and additional levels of fine-grained parallelism can be distributed to processors within a cluster. This paper presents a lightweight and highly optimized support for nested parallelism on cluster-based embedded many-cores. We assess the costs to enable multi-level parallelization and demonstrate that our techniques allow to extract high degrees of parallelism.**

I. Introduction

Recently, several many-core architectures have been proposed that leverage tightly-coupled *clusters* as a building block. Examples include the HyperCore Architecture Line (HAL) processors from Plurality [12], ST Microelectronics Platform 2012 [11], or even GPGPUs like NVIDIA Fermi [13]. In these architectures each cluster is composed of a small-medium number (typically up to 16) of cores, interconnected through a high-bandwidth, low-latency communication and memory system. Inter-cluster communication is achieved through a scalable interconnection medium, such as a NoC. These systems often leverage a shared memory model, 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 on the physical path that corresponding transactions traverse.

Similar to traditional NUMA systems, nested (or multi-level) parallelism represents a powerful programming abstraction for these architectures. Exploiting a single level of parallelism means that there is a single thread (master) that produces work for other processors (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 be not enough coarse-grained parallelism in an application to keep all the processors busy. Nested parallelism is used to increase the efficiency of parallel applications in large systems, and 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 all or a restricted set of processors, thus enabling better resource exploitation. In our cluster-based architecture nested parallelism is extremely beneficial, where a first level of parallelism can be used to distribute coarse-grained tasks to clusters, and one or more inner levels of fine-grained (e.g., loop-level) parallelism can be distributed to processors within a cluster.

978-3-9810801-8-6/DATE12/

Nested parallelism can be implemented by using a mix of programming models. For example, a message passing layer such as MPI could be used to express outer levels of parallelism among clusters, and data parallelism *à la* OpenMP could be used to distribute inner levels within a cluster. However, the use of two different programming models makes application development cumbersome, as the programmer is required to manually create threads and orchestrate their communication and synchronization using different paradigms. Moreover, this approach makes it difficult, if not impossible, the application of global policies (for instance, to perform load balancing or improve data locality) that cross the boundary of each layer. A more appealing solution is one where the programmer is allowed to create nested parallel regions from within a unique programming model, such as OpenMP. In this paper we will consider this approach.

Supporting nested parallelism on a resource-constrained system such as our clusters is a challenging task. Relying on solutions where new threads are created on the fly whenever more parallelism is needed is not feasible, since this approach would shortly run out of memory, and would impose too large time overheads to enable fine-grained parallelism. In this paper we present a lightweight and highly optimized data structure and support for nested parallelism on cluster-based embedded many-cores. We provide a detailed analysis of the necessary costs to create additional parallelism at an arbitrary nesting level, and demonstrate that our techniques allow to extract high degrees of parallelism on real applications.

The rest of the paper is organized as follows. In Section II we discuss previous work on nested parallelism. The target architectural template is presented in Section III. Our optimized implementation of the support for nested parallelism is described in Section IV. We provide the results of our experiments in Section V, and summarize conclusive remarks and future research directions in Section VI.

II. Related Work

Nested parallelism can be implemented in different ways [1] [2] [3] [4] [5]. In literature many techniques exist, which can be categorized into two main approaches:

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,

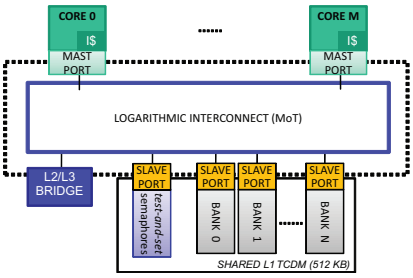


Fig. 1. On-chip shared memory cluster template

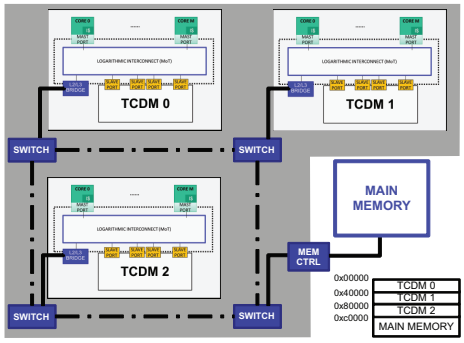


Fig. 2. Multi-cluster architecture and global address space

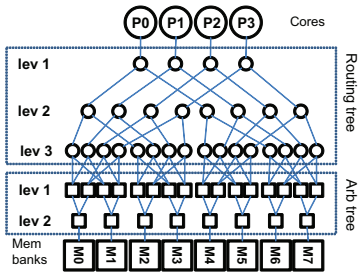


Fig. 3. Mesh of trees 4x8

but has a major drawback: thread creation is expensive both in terms of space (memory footprint) and time [10], [16]. In a resource-constrained platform such as ours this approach would quickly run out of memory, and the resulting time overheads would disallow fine-grained parallelism.

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 idle workers. When a program requests the creation of parallelism, physical threads are fetched from the pool. 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.

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 [6]. 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 [1] [7]. The work in [5] relies on a fixed thread pool, but allows multiple logical threads to be mapped on a single physical thread and maintais 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* [8] [9], and is in fact an orthogonal technique to nesting. OpenMP itself, since its latest specification [17], provide *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 the enclosure of 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 the following we describe our streamlined and optimized implementation of nested parallelism at the cluster level. Work queue-based parallelism can orthogonally be provided within our support.

III. Architectural Template

The architectural template that we consider in this work is a many-core programmable accelerator which leverages tightly-coupled *clusters* as a building block. A simplified block diagram of the target cluster is shown in Fig. 1. In this template, scaling to large system sizes is enabled by replicating clusters and interconnecting them with a scalable medium like a NoC (See Fig. 2). In this work we focus on the cluster subsystem, and describe an optimized implementation of the support for nested parallelism for this hardware. A cluster consists of a configurable number (up to 16) of processors with private instruction caches. Processors are interconnected through a low-latency, high bandwidth logarithmic interconnect similar to the one proposed by Plurality [12] or [14], and communicate through a fast multi-banked, multi-ported tightly-coupled data memory (TCDM). The number of memory ports in the TCDM is equal to the number of banks to allow concurrent accesses to different banks. Conflict-free TCDM accesses have two-cycles latency. The logarithmic interconnect is built as a parametric, fully combinational mesh-of-trees (MoT) interconnection network (see Fig. 3). Data routing is based on address decoding: a first-stage checks if the requested address falls within the TCDM address range or has to be directed off-cluster. The interconnect provides fine-grained address interleaving on the memory banks to reduce banking conflicts in case of multiple accesses to logically contiguous data structures. The crossing latency is one clock cycle. If no bank conflicts arise, data routing is done in parallel for each core. In case of conflicting requests, a round-robin based scheduler coordinates accesses to memory banks in a fair manner. Banking conflicts result in higher latency, depending on the number of conflicting requests. Multiple concurrent reads on a same address are satisfied through read broadcast, which completes in one cycle.

Processors can synchronize by means of standard read/write operations at a memory bank providing *test-and-set* semantics (hardware semaphores). Addresses belonging to L2/L3 memories are also mapped in the global address space (see Fig. 2), but corresponding transactions are transported off-cluster through a memory controller.

IV. Lightweight Support for Nested Parallelism

As discussed in the previous section, the FTP approach is the one which provides the simplest requirements for supporting nested parallelism, thus it represents the natural choice for our architecture. At boot time we create as many threads as processor, 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. Here

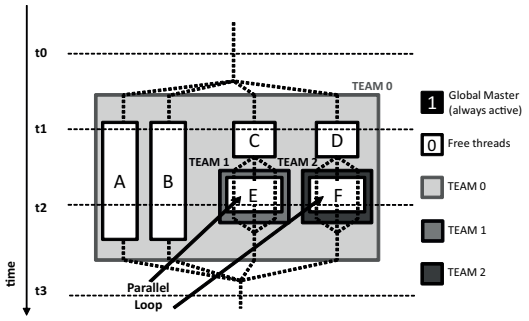


Fig. 4. Application with nested parallelism

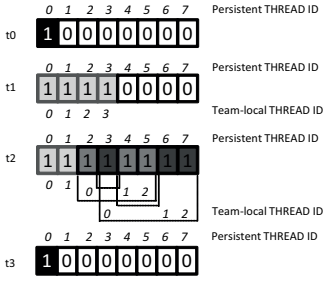


Fig. 5. Global pool descriptor

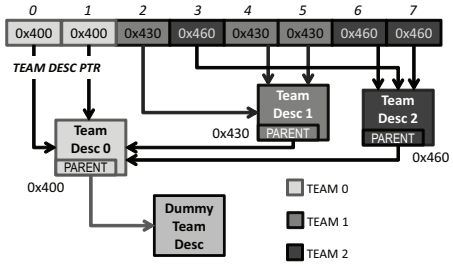


Fig. 6. Tree of team descriptors to track nesting

it is important to point out that 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 them with 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*). 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. A detailed description of the various data structures is provided in the following.

A. Forking threads

The first piece of 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 of an application which uses nested parallelism. At instant *t0* only the global master thread is active, as mirrored by the pool descriptor depicted 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. Besides fetching threads from the global pool, creating a new parallel team involves the creation of a *team descriptor* (see Fig. 7), which holds information about the work to be executed by the participating threads. This descriptor contains two main blocks:

a) Thread Information: A pointer to the code of the parallel function, and its arguments.

b) 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, with N being the number of threads composing 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.

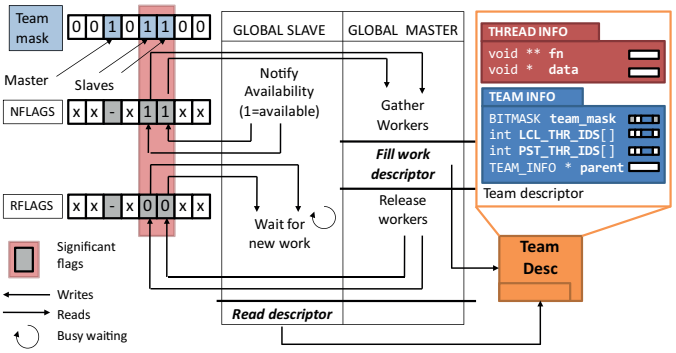


Fig. 7. Thread docking, synchronization and team descriptor

This team descriptor has a memory footprint of only 48 Bytes. Once the team master has filled all its fields, the descriptor it 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.

B. Joining Threads

Joining threads at the end of parallel work typically involves global (barrier) synchronization. Supporting nested parallelism implies the ability of independently synchronizing different thread teams (i.e., processor groups). To this aim we can leverage the mechanism described previously to dock threads, which behaves as a standard *Master-Slave* 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. 7 shows how threads belonging to *TEAM 1* (instant *t2* of our example) synchronize through these data structures.

V. Experimental Results

In this section we validate our nested parallelism support design. The architectural details of our target platform are summarized in Table I.

TABLE I
Architectural parameters

ARM v6 cores	(up to) 16	TCDM banks	16
$I\$_i$ size	1 KB	TCDM size	512 KB
$I\$_i$ line	4 words	L3 latency	50 cycles
t_{hit}	= 1 cycle	L3 size	256 MB

As a first exploration, we characterize the cost for opening and closing parallel teams, providing a breakdown of the various sources of overhead. We compare two different implementations of thread docking, namely one which busy-waits for available work to do, and one that puts cores to sleep when idling (idle/wake in the plots). For the *busy-wait* implementation we allocate polling flags for global slaves on different banks of the TCDM to reduce the conflicts. Fig. 8 and Fig. 9 show the cost in (hundred) clock cycles for opening and closing a team, respectively, at the outermost level of parallelism. In this experiment the master thread requests the maximum number of available threads, and we consider increasing sizes for the thread pool. The breakdown plot shows the cost for each of the three main steps taken upon creation of a new team:

1) Allocate and populate the team descriptor.
2) Fetch the slave threads from the global thread pool.
3) Release the slaves from global synchronization structures.

The first component does not depend on the number of threads requested. However, the busy-waiting implementation is subject to the effect of memory bank conflicts. The fact that it is almost insensitive to the polling activity of the slave threads idling on the pool confirms the importance of distributing poll flags on separate memory banks, which eventually make its performance very close to the sleep/wake implementation. On the contrary, the time spent for fetching and releasing slave threads is dependent on their number, since these operations take place from within a loop iterating for as many times as the number of requested slaves.

Overall, it is possible to see that opening a new team composed of 16 threads takes $\approx$ 690 cycles for the busy-wait implementation, and $\approx$ 600 cycles for the sleep/wake implementation. The breakdown for the team closing shows two components: the time to collect the team threads on the synchronization structure, and the time to tear down the team descriptor and restore the execution context of the parent team by updating global data structures. Collecting threads on the dock is done iterating over the team participants, so the execution time of this section increases with the number of threads in the team. Updating data structures with the information about the parent team context, on the contrary, is independent of the number of threads. It is important to recall here that opening and closing a team implies the use of critical sections to protect updates to global data structures. As such, if more than one attempt to create/destroy a new team at the same time takes place the execution of (parts) of the procedure gets serialized on the concurrent calling threads, which we study next.

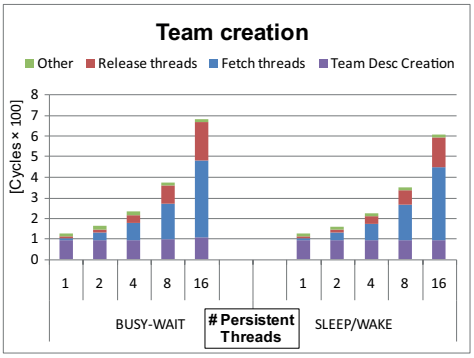


Fig. 8. Cost of creating a new team

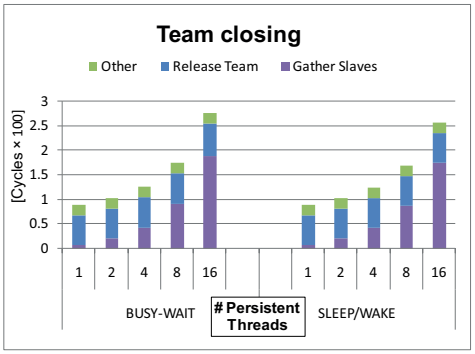


Fig. 9. Cost of closing a team

Besides characterizing the cost of our basic constructs to create and destroy parallel teams, we study the effect of nested parallelism creation from within OpenMP. The library functions invoked by the compiler when a `#pragma omp parallel` construct is encountered have been rewritten as a wrapper around our primitives for parallelism creation. A programming model such as OpenMP exposes a simple and intuitive interface for nested parallelism, however it introduces additional function call overhead to interact with the runtime environment. To measure the OpenMP runtime overhead we use the EEPC microbenchmarks [15], and extend the methodology 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. 10 we show the task graph representation of the microbenchmarks used to assess the cost of nested parallelism with depth 1, 2 and 4 respectively. The computational kernel (indicated as W in the plots) is composed uniquely 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 4 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. We thus divide the gross overhead by the nesting depth to have an average cost for parallel region opening and close. Figure 11 shows this cost for varying granularities of

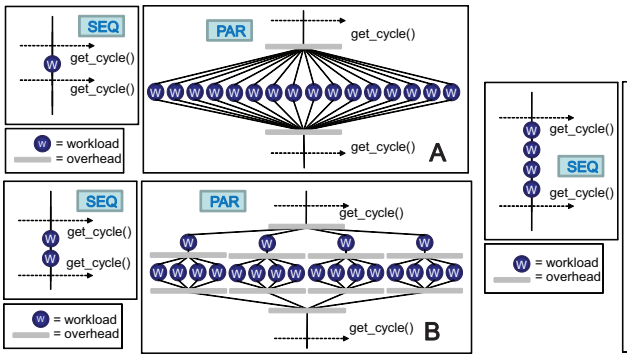
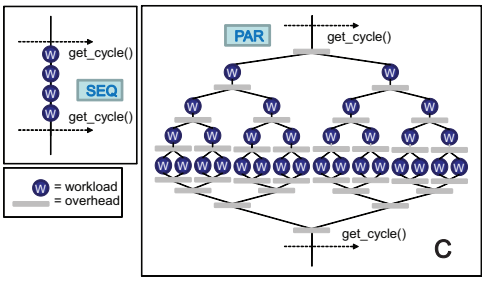


Fig. 10. Microbenchmark for nested parallelism overhead. A) 1 level, B) 2 levels, C) 4 levels



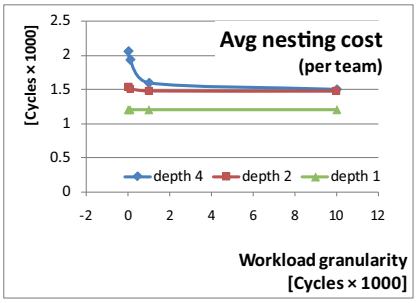


Fig. 11. Cost for different level of parallelism

the work unit (W). The first two things to notice are that i) the cost for single-level parallelism creation from OpenMP is, as expected, slightly higher than the sum of the costs for opening and closing a team that we described earlier, but not much so (roughly 15%), and ii) inner parallelism is slightly costlier to create than the outermost level. The latter is a consequence of the fact that when two or more threads try to concurrently open a new team, the execution of the opening sequence gets serialized due lock-protected updates to global data structures. For this reason, when W contains very small amounts of work this effect is dominant, and the cost for parallelism creation increases with the depth of nesting.

A. *Strassen matrix multiplication*

In this section we evaluate the effectiveness of our nesting support on a real application kernel, and compare it against other techniques to extract multi-level parallelism. As outlined in Section IV, one efficient abstraction that can be orthogonally applied to nesting is the *work queue*. OpenMP supports this type of parallelism through dynamic loops (or, in the latest specification, *tasks*). We refer to this kind of parallelism as *tasking* in the following, and compare it against nesting.

The target application for our experiment is the *Strassen* algorithm for matrix multiplication. It is a good candidate for our exploration, since it naturally exposes high degrees of parallelism, both at the task- and data-level, thus being easily parallelized with both the proposed approaches. The algorithm is shown in the leftmost part of Figure 12.

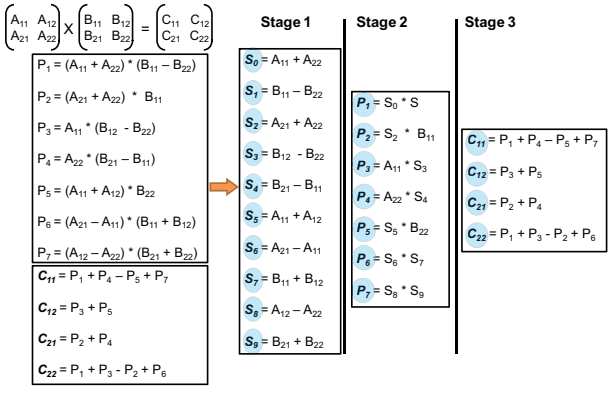


Fig. 12. Strassen algorithm for matrix multiplication and its basic kernels

The input matrices A and B are decomposed in four sub-matrices, which can be processed in parallel. Sub-matrices undergo a number of sums/subtractions and multiplications. Each of these operations is fully data-parallel. The algorithm is naturally structured in three stages: in stage one ten sums are computed, which we identify as S_0 ... S_9. These sums can be mapped to parallel tasks, or be data-parallelized. In stage two, seven multiplications are computed (P_1 ... P_7), which similarly exhibit both data and task parallelism. Finally, in the third stage four sets (C_{11} ... C_{22}) of sums and subtractions lead to the final result. Our strategy to parallelize the application with *tasking* is the following. We create a single level of parallelism using all the threads in the pool. We build a large parallel region containing all the operations from the three stages in sequence. All of the operations are data parallelized, namely, all the threads can dynamically fetch work from all of the loops. Ideally, this scheme can extract the maximum degree of parallelism, and has a theoretical speedup of 16×.

To parallelize the application with the *nesting* approach we follow the natural task partition of the application. Figure 13 shows a pictorial representation of this parallelization scheme.

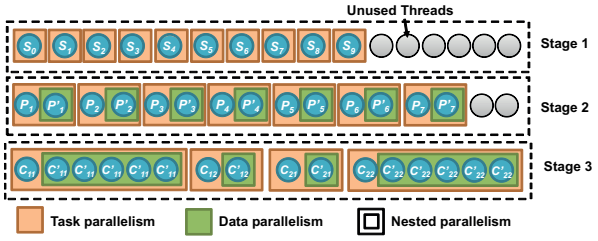


Fig. 13. Strassen algorithm parallelized with nesting support

At stage 1, we assign all the ten sums (S_0, ... , S_9) to as many threads. We do not create additional data parallelism on the remaining six threads, because this would lead to unbalanced execution. These left out threads remain idle in this stage, thus ihnerently limiting the parallelization speedup to 10×. In the second stage, the seven multiplications (P_1, ... , P_7) are initially assigned to seven threads. Each of these threads generates a nested region and exploits an additional thread thus leveraging data parallelism as well. The remaining two threads in the global pool are left idle, thus the maximum achievable speedup is limited to 14x. In the third stage, four parallel threads are assigned the final sums (C_{11}, ... , C_{22}). The workload contained in these tasks is unbalanced by a factor of 3:1 for tasks C_{11} and C_{22} with respect to the other two (three sums instead of one). By creating nested data-parallel regions with different number of threads (C_{11} and C_{22} will run on six threads, while C_{12} and C_{21} will run on two) we are

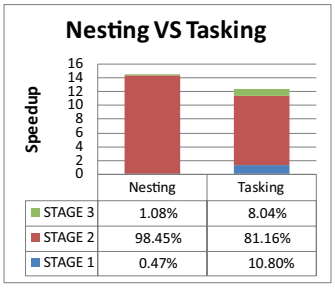


Fig. 14. Nesting and tasking speedup

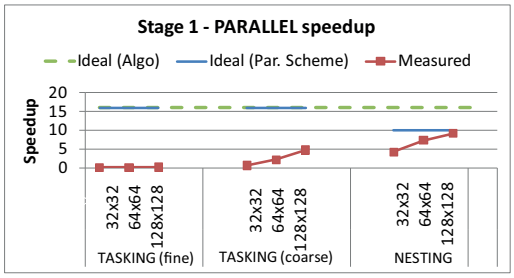


Fig. 15. Effect of task granularity (Stage 1)

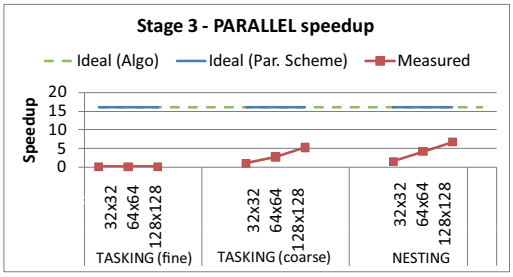


Fig. 16. Effect of task granularity (Stage 3)

capable of balancing the workload and exploiting all available threads. This stage fully exploits the computational resources of the system, with a theoretical speedup of 16x.

As a first experiment, we consider an instance of the algorithm using 64x64 matrices (four 32x32 submatrices). Results for this experiment are shown in Figure 14, where we report the speedup achieved by the two parallelization strategies. Overall, the theoretical speedup for the *nesting* approach is $\approx 14\times$ ($16\times$ for *tasking*). It is possible to see that, notwithstanding the threads left idle at times, the *nesting* approach matches its theoretical speedup. The *tasking* approach, on the contrary, is far from it. The numbers on the table below the figure show the percentages of time spent in the three stages. It is possible to notice two things. First, the multiplication kernels unsurprisingly dominate execution time. Second, the first and third stages take non negligible time with the tasking approach as compared to nesting. This is attributable to the overhead for distributing workload from the work-queue at a too fine granularity.

In the following experiment we "zoom-in" these phases to have better insight. Figure 15 shows how the execution time of the first stage is affected by the size of the input submatrices, which are set to 32x32, 64x64 and 128x128. This plot confirms that for fine-grained workload (leftmost plot) the *tasking* approach cannot achieve any speedups. To see how this phenomenon can be mitigated by considering coarser work units we increased the chunk size for the parallel loop to its maximum (the number of iterations is evenly divided among participating threads). Even in this case (plot in the middle), if the matrix size is too small the overhead for the work queue is not amortized. With bigger matrix sizes (128x128) we achieved a $5\times$ speedup. All of those results are far from the theoretical speedup achievable with the *tasking* parallelism because of the implementation overheads. The rightmost plot shows how the *nesting* approach achieves much better results. For matrix sizes of 128x128 this parallelization scheme achieves its theoretical speedup peak.

Similar plots are provided in Fig. 16 for the third stage.

VI. Conclusion

In this paper we have presented a highly optimized implementation of nested parallelism for cluster-based embedded MPSoCs. Our results confirm that an extremely lightweight support for nesting enables the extraction of high degrees of parallelism from applications. Work-queue based approaches can be coupled to this support, but it is extremely important that their implementation is also highly optimized to prevent high overheads from inhibiting the potential for parallelism exploitation, which we plan as future work.

Acknowledgment

This work was supported by projects FP7 VIRTICAL (288574) and JTI SMECY (ARTEMIS-2009-1-100230), funded by the European Community.

References

[1] E. Ayguadé, X. Martorell, J. Labarta, M. Gonzalez, and N. Navarro, "Exploiting multiple levels of parallelism in OpenMP: A case study," in *Proceedings of the 1999 International Conference on Parallel Processing*, ser. ICPP '99, 1999, pp. 172–.

[2] 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.

[3] X. Martorell, E. Ayguadé, N. Navarro, J. Corbaln, M. Gonzlez, and J. Labarta, "Thread fork/join techniques for multi-level parallelism exploitation in numa multiprocessors," in *in NUMA Multiprocessors. In 13th Int. Conference on Supercomputing ICS'99, Rhodes*, 1999, pp. 294–301.

[4] P. Hadjidoukas and V. Dimakopoulos, "Nested parallelism in the OMPI OpenMP/C compiler," in *Euro-Par 2007 Parallel Processing*, ser. Lecture Notes in Computer Science, A.-M. Kermarrec, L. Boug, and T. Priol, Eds. Springer Berlin / Heidelberg, vol. 4641, pp. 662–671.

[5] Y. Tanaka, K. Taura, M. Sato, and A. Yonezawa, "Performance evaluation of OpenMP applications with nested parallelism," in *Selected Papers from the 5th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, ser. LCR '00, 2000, pp. 100–112.

[6] D. Novillo, "OpenMP and automatic parallelization in GCC," in *Proc. of the 2006 GCC Summit*, Ottawa, Canada (June 2006)

[7] M. González, J. Oliver, X. Martorell, E. Ayguadé, J. Labarta, and N. Navarro, "OpenMP extensions for thread groups and their runtime support," in *Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing-Revised Papers*, ser. LCPC '00, 2001, pp. 324–338.

[8] AMD, "The AMD Fusion Family of APUs."

[9] Apple, Inc, "The Grand Central Dispatch ."

[10] G. J. Narlikar and G. E. Blelloch, "Space-efficient scheduling of nested parallelism," *ACM Transactions on Programming Languages and Systems*, vol. 21, 1999.

[11] ST Microelectronics and CEA. Platform2012: A Many-core programmable accelerator for Ultra-Efficient Embedded Computing in Nanometer Technology.

[12] Plurality Ltd. HyperCore Processor. *www.plurality.com/hypercore.html*.

[13] NVIDIA. Next Generation CUDA Compute Architecture: Fermi - WhitePaper. *www.nvidia.com/object/fermi_architecture.html*, 2010.

[14] A. Rahimi, I. Loi, M. R. Kakoee, and L. Benini, "A fully-synthesizable single-cycle interconnection network for shared-L1 processor clusters," in *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE) 2011*, 2011, pp. 1 – 6.

[15] University of Edinburgh, "OpenMP Microbenchmarks V2.0." *www2.epcc.ed.ac.uk/computing/research_activities/openmpbench/openmp_index.html*

[16] V. V. Dimakopoulos, P. E. Hadjidoukas, and G. C. Philos, "A microbenchmark study of OpenMP overheads under nested parallelism," in *Proceedings of the 4th international conference on OpenMP in a new era of parallelism*, ser. IWOMP'08, 2008, pp. 1–12.

[17] www.openmp.org. OpenMP Application Program Interface v.3.0. *www.openmp.org/mp-documents/spec30.pdf*