Parallel Patterns for General Purpose Many-Core

Daniele Buono, Marco Danelutto, Silvia Lametti and Massimo Torquati
Dept. Computer Science – Univ. of Pisa, Italy
{d.buono,marcod,lametti,torquati}@di.unipi.it

***Abstract*—Efficient programming of general purpose many-core accelerators poses several challenging problems. The high number of cores available, the peculiarity of the interconnection network, and the complex memory hierarchy organization, all contribute to make efficient programming of such devices difficult. We propose to use parallel design patterns, implemented using algorithmic skeletons, to abstract and hide most of the difficulties related to the efficient programming of many-core accelerators. In particular, we discuss the porting of the FastFlow framework on the Tilera Tile*Pro*64 architecture and the results obtained running synthetic benchmarks as well as true application kernels. These results demonstrate the efficiency achieved while using patterns on the Tile*Pro*64 both to program stand-alone skeleton-based parallel applications and to accelerate existing sequential code.**

***Keywords*-Structured programming, many-core, shared memory, accelerators, TilePro64.**

I. Introduction

CPU hardware advances move along two distinct axes: on the one hand, more and more powerful and integrated multi-core architectures are developed, where a relatively limited number of cores (typically 4 to 16) share a standard memory hierarchy highly optimized to support efficient cache coherency mechanisms. On the other hand, several "general purpose co-processors" are developed with a big number of cores (typically 64 to 128) interconnected using regular networks (meshes and/or rings, usually) with several distinct and very efficient mechanisms supporting accesses to core local caches as well as inter-core communications. Notable examples of this second kind of (co)processors are the Intel MIC [1] and the Tilera architectures [2]. These general purpose co-processors are aimed to compete with the more classical, GPU style co-processors in all those fields where data parallelism is not enough, and they may be seamlessly programmed using standard tools and mechanisms, such as POSIX threads. However, all vendors provide proprietary, low level libraries to support *more efficient* parallel programming of these machines. As an example, Tilera proposes two distinct environments in addition to plain Linux POSIX threads environment: a POSIX thread only environment (one thread per core, no Linux processes/scheduler behind), and a "bare metal" environment (one C/C++ program/thread per core, with access to the lower level inter core communication library only).

[0]This work has been partially supported by EU FP7 STREP No. 288570 *ParaPhrase*. Silvia Lametti is a recipient of the Google Europe Fellowship in Computer Architecture, and this research is supported in part by this Google Fellowship.

The programming scenario presented to the application programmer has therefore two different and somehow contradicting features: a) it is definitely higher level with respect to the programming scenario exposed by GPGPUs, and in general supports rapid porting/prototyping of shared memory POSIX applications, but also b) it requires a deep understanding both of the peculiarities of the hardware architecture and of the features of the proprietary libraries provided to exploit lower level hardware characteristics and therefore to be able to make a better usage of the advanced architectures.

In this work, we attack the problem of providing suitable, user (application programmer) friendly and efficient programming frameworks for general purpose many-core co-processor architectures. We adopt a structured parallel programming approach based on the usage of parallel design patterns [3] and algorithmic skeletons [4] and we demonstrate that skeletons implementing well-known parallel patterns may be used to encapsulate all the idiosyncrasies relative to the efficient implementation of parallel applications on these devices, both in case of full parallel application running on the co-processor and in case of parallelization of applications through "domain specific software accelerators" running on the co-processor. In particular our contribution may be stated as follows:

- We ported FastFlow, a shared memory structured parallel programming framework targeting shared memory multi-cores [5], on the Tilera Tile*Pro*64 architecture.
- We demonstrated that FastFlow may efficiently run synthetic kernels as well as real applications on the Tile*Pro*64 completely fulfilling the architecture potentialities both in case of stand-alone applications executed on the co-processor and of software accelerators offloaded from applications running on the host machine.
- We demonstrated different optimizations related to memory hierarchy usage and cache coherency management that may be directly encapsulated at the skeleton level, leaving the application programmer free to concentrate on the business logic of the application at hand.

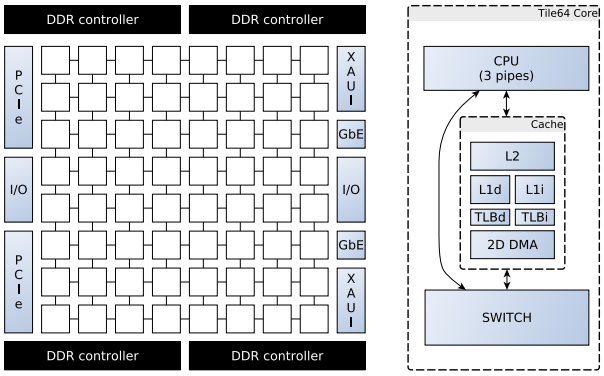

Figure 1: Tile*Pro*64 architecture (chip architecture (left) and single core architecture (right))

II. TILE*Pro*64 PROCESSOR ARCHITECTURE

The Tile*Pro*64 [2] processor features 64 identical processing cores (tiles) interconnected with Tilera's iMesh on-chip network. Each tile consists of a) a 3-way superscalar VLIW processor running at 866MHz, b) a cache subsystem composed of 16KB Level 1 instruction Cache (L1i), 8KB Level 1 data cache (L1d) and 64 KB Level 2 cache (L2), and c) a switch that implements the iMesh interconnection network. The cache subsystem also contains a Translation Lookaside Buffer (TLB) and a DMA engine that support automatic memory-to cache and cache-to-cache data transfers. In contrast to common multi-core, no automatic memory prefetching mechanisms are provided.

The iMesh [6] NoC is composed of five independent meshes, each carrying a different kind of traffic. One (MDN) is dedicated to memory transfers in such a way memory accesses are not influenced by other traffic, another one (UDN) is reserved to user-level traffic[1], The other meshes are dedicated to I/O transfers and cache coherence protocol.

To sustain the memory bandwidth requirements relative to the 64 cores, the Tile*Pro*64 provides four on-chip memory controllers that are placed at the four edges of the mesh. Therefore, the memory latency from a given tile depends on the tile position in the mesh and on the memory controller selected.

Figure 1 shows the architecture diagram of the Tile*Pro*64 processor. The Tile*Pro*64 design includes an innovative, distributed and scalable approach handling virtual memory and cache coherence on chip. Virtual memory pages are "striped" among the physical controllers in 8KB chunks. The memory requests are therefore automatically spread across the different memory controllers. By using a different operating mode, the programmers may directly require to allocate entire VM pages on particular memory controllers, as usual in NUMA architectures.

[1]supporting explicit data transfers among tiles under application programmer orchestration

Tilera also provides advanced cache management mechanisms and policies. As an example, in order to guarantee cache coherence, the Tile*Pro*64 implements Dynamic Distributed Cache (DDC)[7]. This is a kind of distributed directory mechanism implemented on top of L2 caches. The basic idea is that for each cache line a tile is elected to be the cache line *Home Node*. This Home Node is responsible for handling the coherency relative to the line, by always maintaining an updated version and sending proper invalidations when needed. This "homing" mechanism is handled by the L2 cache of the tile, so that any local L2 space is contended by the core on the tile and the DDC.

The home node is commonly determined by an hash function that enforces a uniform distribution among the 64 tiles. This represents a substantial improvement when using a few cores: by using DDC we are given the abstraction of a virtual distributed inclusive L3 cache, composed by the sum of all tiles' L2 caches. However, when a lot of cores are used in a computation, for every cache line in the program working set there must be an home node containing and managing the line updated copy. On average, the tile using a line will not be its home, and therefore two copies of the line will be allocated in L2, *de facto* halving the L2 capacity. For this reason DDC is specified on a virtual-memory-page basis and allows different homing strategies. Aside from the hashing, a programmer can define a specific tile as home node for the entire memory page. If properly selected, the home node will match the core that is extensively using that page, reducing the L2 contention effect. Or, coherency can be completely disabled by defining no home node for the page. This turns out to be quite useful when using read-only memory pages. In all the other cases, disabling coherency means that the (application) programmer must explicitly flush data from caches to ensure data coherence.

Using programmer-specific DDC settings is facilitated by the Tilera proprietary library TMC[7]. Instead of using the heap space, however, TMC entries add and remove virtual memory pages, with the required DDC settings. The following code shows common usage of these functions.

```
1  // Variable containing allocation settings
2  tmc_alloc_t alloc = TMC_ALLOC_INIT;
3  // Setting the Home Node on the worker node
4  tmc_alloc_set_home(&alloc,id_worker);
5  // Disabling the automatic Cache Coherence mechanisms...
6  tmc_alloc_set_home(&alloc,TMC_ALLOC_HOME_INCOHERENT);
7  // Allocating a set of virtual memory pages with the specified settings
8  t = (task_t*) tmc_alloc_map(&alloc,sizeof(task_t));
9  // Flushing modified data in case of incoherent memory
10 tmc_mem_flush(t,sizeof(task_t));
11 // And dellocating a set of virtual memory pages
12 tmc_alloc_unmap(t, sizeof (task_t));
```

The Tile*Pro*64 processor architecture defines a relaxed memory consistency model: both load and store can be reordered (similar to POWER architectures and in contrast to x86 architectures where store order is guaranteed). A memory fence instruction is provided to force ordering, if needed. Moreover, according to a pure RISC approach,

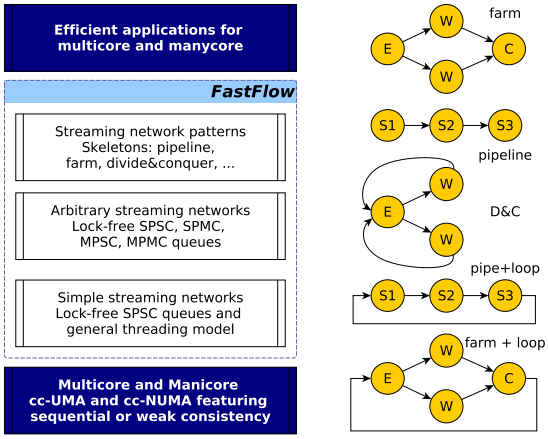

Figure 2: FastFlow layered architecture and basic patterns.

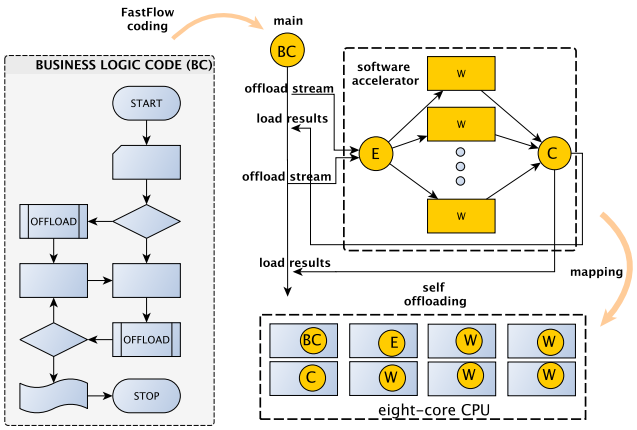

Figure 3: Self-offloading on the FastFlow software accelerator.

a single atomic Test-And-Set has been included in the instruction set.

Some of the 64 cores are not available to user programs, as they are used to run O.S. specific code to drive external interfaces (i.e. networks and PCIe). Indeed the programmer may disable or move to specific tiles most of these drivers.

In addition to normal inter-tile data exchange through shared memory, the Tile*Pro64* architecture provides mechanisms to support inter-tile message passing through the iMesh NoC. This mechanisms ensure ultra low latency communications, while providing several limitations on the amount of data exchanged in a single communication.

III. The FastFlow programming framework

FastFlow[2] is a structured parallel programming environment implemented in C++ on top of POSIX threads. It provides programmers with predefined and customizable stream parallel design patterns such as task farms and pipelines. It has been initially designed and implemented to be efficient in the execution of fine grain parallel applications on general purpose multi-core architectures [8]. The FastFlow design is layered (see Fig. 2). The lower layer implements a lock-free and wait-free Single-Producer, Single-Consumer queue [9]. On top of this mechanism, the second layer provides Single-Producer Multiple-Consumers and Multiple-Producers Single-Consumer queues using arbiter threads. This abstraction is designed in such a way that arbitrary networks of activities can be expressed while maintaining the high efficiency of the synchronizations. Eventually, the third layer provides several parallel patterns as standard C++ classes [10] (Fig. 2, right).

In FastFlow, the concept of concurrent activity (i.e. of a flow of control that may be run in parallel with other flows of control) is abstracted by the `ff_node` class. An `ff_node` is used to encapsulate sequential portions of code implementing functions as well as higher level parallel patterns such as pipelines and farms. Each `ff_node` will be used to run a concurrent activity in a thread, and it has associated two (shared memory) message queues: one used to receive pointers to input data to be processed, and one to deliver the pointers to computed results.

The FastFlow predefined patterns may be customized in different ways, e.g. by nesting pipelines with farm stages and vice versa. Using the customization features, different patterns may be implemented in terms of the pipeline and farm building blocks, including divide&conquer, map and reduce patterns.

Although not detailed in this work, FastFlow is being currently extended to support the offloading of data parallel computation to GPUs, and to target homogeneous multi-core COW/NOW.

A. FastFlow *accelerator interface*

The original design of FastFlow actually provides the application programmer with two distinct operation modes. The first one (*standalone parallel program* mode) basically provides the possibility to write full parallel applications as (compositions of) FastFlow parallel design patterns. The application code, in this case, consists in a C++ `main` where the pattern (nesting) structure is declared and executed using proper calls to the methods provided by the `ff_node` class.

The second mode (*accelerator* mode) instead, supports the self-offloading of parallel computations from within standard C++ sequential code to a software accelerator programmed as a parallel design pattern composition. In this case, the programmer creates the accelerator with mechanisms similar to those needed to create a standalone parallel FastFlow program and then *offloads* tasks to be computed to the accelerator that asynchronously and in parallel computes

[2]Project site http://sourceforge.net/projects/mc-fastflow/

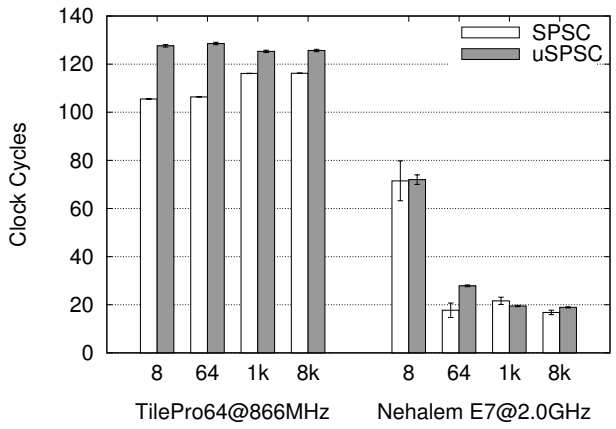

Figure 4: Average latency times for the FastFlow queues on Tile*Pro*64 and Intel processors varying the buffer size.

them, delivering the results to the main program upon request (Fig. 3, right). The accelerator is run on the unused cores available on the target architecture, hence the *self*-offloading term. In [5] more information about the FastFlow accelerator can be found.

IV. Porting FastFlow on Tile*Pro*64

Our original idea was to use structured parallel programming techniques to exploit general purpose many-core. Therefore we decided to port FastFlow on Tile*Pro64*. FastFlow is actually provided as a set of `.hpp` files written in C++ standard. In principle, recompiling is the only action required to port the framework on the a different architecture. However, two particular issues should be taken into account. First, the Single-Producer Single-Consumer (SPSC) lock free and wait free FastFlow queue requires a memory fence instruction, which usually changes with the instruction set/architecture. Second, some of the synchronization mechanisms of the POSIX threads are known not to be scalable (e.g. barriers), and less general, proprietary mechanisms and libraries implementing the same kind of synchronization should be used instead, if available. Indeed, solving these issues are the only significant steps we had to perform to port FastFlow onto the Tile*Pro64*.

While considering how to deal with these issues, we also considered deeper modifications in the FastFlow runtime support, to better exploit this architecture. In particular: a) we developed a way to exploit per-virtual-page DDC policies in a programmer-transparent way, so that every aspect is handled by the FastFlow support, and b) we redefined the FastFlow accelerator mechanism to support offloading of computations from the main CPU cores to a Tile*Pro*64 co-processor.

A. SPSC queue & synchronization mechanisms

The first part of the porting focused on the architecture dependent instructions in the FastFlow runtime support. In particular, the lock free and wait free SPSC queue implementation requires to use a Write Memory Barrier instruction (WMB)[3] to enforce memory write operations ordering. FastFlow was engineered in such a way it turned out to be quite easy to define the behavior of the WMB for each supported architecture, as shown in the following code.

```
1 #ifdef __x86_64__ // x86 32/64−bit: no memory fence is needed.
2 #define WMB() __asm__ __volatile__ ("": : :"memory")
3 #endif
4 #ifdef __tile__ // Tilera: using a compiler intrinsic for memory fence.
5 #define WMB() __insn_mf();
6 #endif
```

Indeed, the usage of this fence instruction comes at a cost: Fig. 4 outlines the differences in between the average queue latency on the Tile*Pro*64 and an x86 processor for bounded (SPSC) and unbounded (uSPSC) FastFlow queues.

All critical synchronizations in FastFlow are implemented on top of the SPSC queue, in a lock-free fashion. However, some portions of code exist, used in non-critical path, where atomic operations and pthread-based synchronization mechanisms are used. Such mechanisms, implemented in kernel space, are inefficient on the Tile*Pro*64 with a high number of threads, therefore we substituted them with equivalent Tilera's TMC spin-based user-level routines, which are more scalable and predictable.

B. Memory Allocation Policies

As explained in Section III, the third layer of FastFlow provides parallel patterns. The pattern structure of the application can be used to derive important static information on the flow of data and tasks among threads. In order to provide optimized memory management policies based on this knowledge, we decided to exploit the flexible cache coherence mechanisms by defining enhanced cache coherence settings for virtual pages containing task data. We implemented three "Memory Allocation Policies", that affect the selection of the Home Node for the data structures:

- **Hash Home Node (HHN)**. This is the default mode defined by the architecture: an hash function is used to uniformly distribute Home Nodes among all the caches. HHN guarantees automatic cache coherence and uniform usage of all the caches, although it may increase the NoC traffic and reduce the effective amount of cache usable per tile with high parallelism degrees.
- **No Home Node - NHN** With this policy the homing mechanism is disabled, resulting in incoherent memory pages, which can affect the correctness of the application. However, the stream-parallel paradigm data-flow semantics guarantees that each task is managed by only one concurrent entity at a time. As a consequence

[3]In many works, the WMB instruction is also referred to as store-fence.

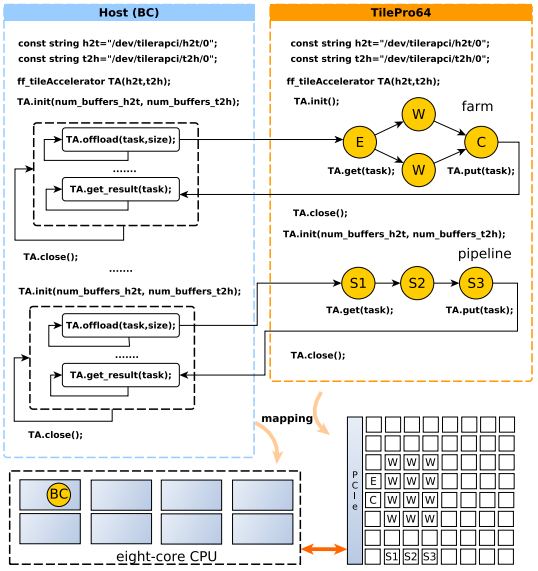

Figure 5: Task offloading to the Tile*Pro*64 accelerator.

coherency need to be ensured *only* when a task is passed to a different concurrent entity. This property allows us to let the application programmer work freely with incoherent memory pages both in read and write modes. When the work on the local task is finished and before sending the task to another concurrent entity, the FastFlow runtime can *automatically* and *transparently* add memory flush operations to enforce cache coherence.

- **Fixed Home Node (FHN)** We specifically select, for each task, a tile that becomes its Home Node. This proves to be effective when we succeed identifing the thread in the parallel pattern (composition) performing most of the work relative to a specific task. In this case, we keep the automatic cache coherence, but we remove most of the performance overhead of the DDC mechanism. This characterization is actually possible for the farm paradigm, where each task is entirely processed by a single thread. Although theoretically very promisingly, the main problem relative to this policy arises considering that the destination thread for a specific task is usually defined late at runtime. This requires a delayed allocation of data for the tasks. In turn, this may require the storage of task data in temporary buffers up to the point where the destination tile is eventually decided. More generally, to fully exploit the FHN features, the programmer should define his own task scheduling policy by overwriting the proper FastFlow class methods.

The usage of these policies requires the adoption of the proprietary memory allocation functions described in section II. Keeping transparency in mind we decided to extend the FastFlow specific memory allocator to "hide" the policy-dependend operations, in such a way the application programmer is just required to call redefined malloc and free operations. This integration also allows useful performance-related optimizations to be implemented. Knowing that the overhead of allocating and deallocating virtual memory pages is actually bigger than with common malloc/free ops, because of the interaction with TLBs, for example, the allocator implements a caching mechanism that recycles unused memory pages for later requests in such a way this problem is solved.

C. Accelerator

The data exchange between the host and the accelerator takes place via the PCI-Express link using a proprietary zero-copy communication Tilera library. On the host side the `main` program is executed, which may also be a concurrent FastFlow program, whilst on the Tile*Pro*64 a complete skeleton composition is instantiated and run through another, dedicated `main` program. As exemplified in Fig. 5, all the functions required on the host side are encapsulated in a `ff_tileAccelerator` object, using two device files to communicate with the skeleton. `ff_tileAccelerator` provides `init` and `close` functions to start and stop accelerator operations, in addition to the `offload` and `get_result` functions supporting asynchronous task offloading to and result retrieving from the accelerator, respectively. The number of input and output buffer slots for each communication channel can also be specified in the initialization phase. The buffer slots determine the maximum number of 64Kbyte messages that can be stored in the channel without being blocked in the offloading operation. On the Tile*Pro*64 side, after the `ff_tileAccelerator init` has been called and before the `close` is called, the complete "accelerator" skeleton is defined and run, similarly to what happens when a stand-alone FastFlow skeleton program has to be run. The coordinated execution of the offloading functions on the host and on the Tile*Pro*64 directly and seamlessly implements a flow of tasks from the host to the co-processor and of results from the co-processor to the host.

Conceptually, the offloading operation directly connects the host side `offload` with the Emitter of a farm skeleton or the first stage of a pipeline skeleton on the co-processor, while the delivering a result operation on the co-processor connects the Collector of a farm skeleton or the last stage of a pipeline skeleton with the host side `get_result`. Furthermore, the asynchronous offloading allows the host to execute a different part of the application, including another FastFlow skeleton, in parallel with the offloaded work.

V. Experiments

Experiments were performed on a TILEncore card [2], equipped with a 866MHz Tile*Pro*64 and 8GB of RAM. The

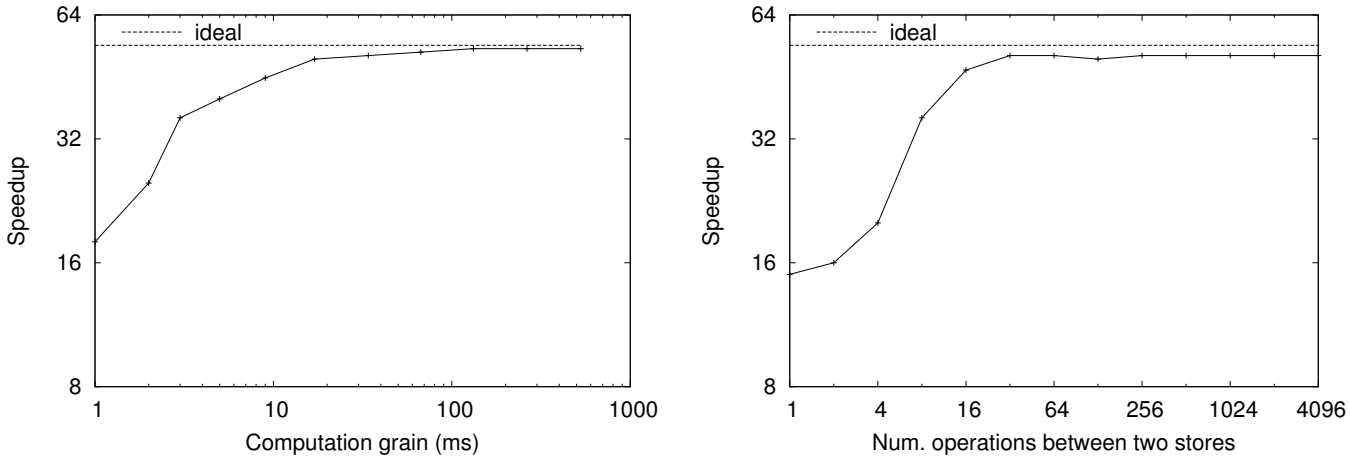

Figure 6: MAP-like benchmark. Left: Speedup obtained varying per worker computation grain. Right: Speedup obtained varying the computation grain w.r.t. the number of memory operations.

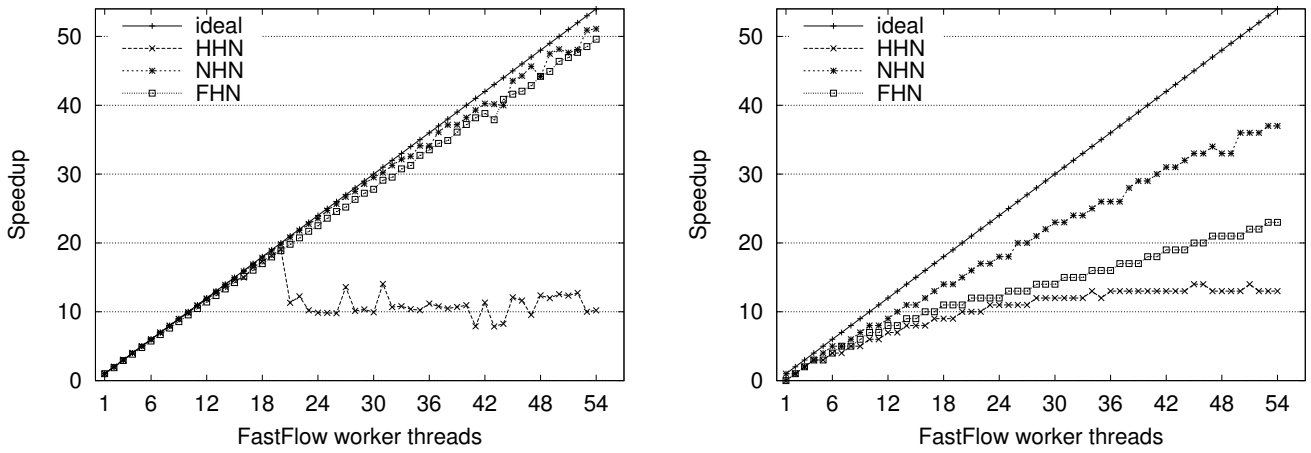

Figure 7: Stream matrix multiplication ($A_i \times B_i$) using different memory task allocation policies: Hash Home Node (HHN), No Home Node (NHN) and Fixed Home Node (FHN). Left: 64×64 integer matrices. Right: 128×128 integer matrices.

board is installed on a 24-core AMD Opteron server. Only 56 out of 64 cores were available to run user code. Two additional concurrent activities are needed when running FastFlow task farms (emitter and collector–E and C in the left side of Fig. 2), and therefore the maximum parallelism degree in our experiments was 54 (W in Fig. 2).

A. Synthetic Benchmarks

The first experiment was aimed at characterizing the computational grain required to fully exploit the Tile*Pro*64 architecture by using the FastFlow programming model, that is to compensate the (small) overhead introduced by FastFlow. We wrote a map-like application, where the same operation is applied to a fixed set of elements. We removed memory access operations from the code to avoid memory-related problems, in such a way each thread (map worker) was just performing a number of multiplications without memory access nor synchronizations. By varying the number of operations, we can easily change the computation grain and analyze the speedup obtained by the parallel FastFlow application. Figure 6 (left) shows that when using the FastFlow run time on the Tile*Pro*64, good speedups may be obtained if tasks last more than 10ms, and in case of relatively long computations (around 100ms) maximum speedup may also be expected.

Current multi-core architectures employs a limited number of cores, and often incur memory-bandwidth problems when using high parallelism. On the Tile*Pro*64 the problem is exacerbated by the limited amount of caches and the high number of cores. To study this problem, we used the same map-like benchmark as before, but in this case to each

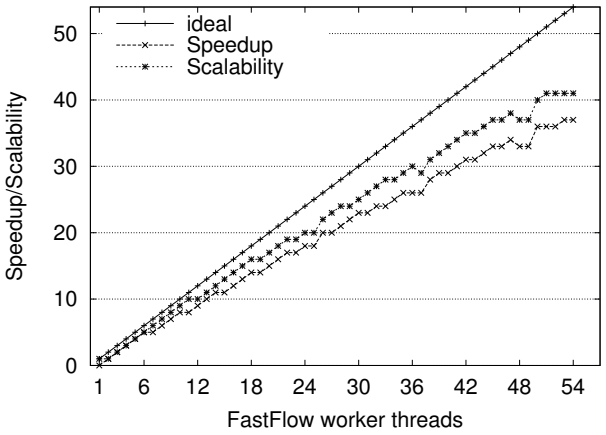

Figure 8: Speedup vs Scalability for the stream matrix multiplication benchmark using 128×128 integer matrices.

worker is given a partition of an input data structure where a given number of integer multiplications is executed on each element. The result is stored in an output data structure. In the test we vary the number of multiplication per element, so that we are able to change the amount of time between two memory operations. To guarantee that the computation grain is enough to maintain the expected speedup, when reducing the number of operations per elements we increase the number of elements, so that the overall number of multiplications in the benchmark is kept fixed.

The results in Fig. 6 (right) confirm that the Tile*Pro*64 is a well balanced architecture, despite the high number of cores. With a very limited number of instructions per element (i.e. from 1 to 16) the memory subsystem becomes the bottleneck. However, as long as algorithms exploit locality we usually fall in the right part of the graph, where the memory subsystem is able to efficiently support parallel activities on 54 cores.

B. Kernel-based Benchmarks

To obtain a better idea on real world applications we executed a matrix multiplication written in FastFlow exploiting the farm skeleton on a stream of 3200 matrices. The matrix multiplication problem is interesting w.r.t. cache hierarchies because performs $O(N^3)$ operations over $O(N^2)$ data, conceptually resulting in very good data reuse; however, because of the algorithm, this actually happen only if matrices are small enough to completely stay in cache. In this case the number of memory-cache transfers matches the amount of data (i.e. we transfer an element from the memory only the first time we access it); on the other hand, if matrices are larger than the cache, we lose data reuse (i.e. by the time we access a previously used element, it has been removed from the cache because of cache trashing), and the number of memory-cache transfers grows to $O(N^3)$.

Given the previous results we expect that in this case the memory bandwidth and the coherence system becomes the bottleneck. We therefore tested the different cache coherence allocation modes discussed in section IV, to check if and how much a specific cache-coherence policy affects the performance of an application.

Figure 7 shows the results for the two test cases: one using 64×64 and the other using 128×128 matrices of integers. For each one we tested the three DDC policies supported in the farm paradigm. With high parallelism degrees we can actually see very different results depending on the DDC policy used. Considering smaller matrices, we have that each matrix takes 16KB of space, so that the entire working set of each worker for each task is 48KB. This means that in this case the working set is small enough to fit in the L2 cache of one tile and therefore the number of memory transfers are minimized. We obtained a very good scalability with the FHN and NHN policies. Instead, the standard HHN policy, works very well up to $\sim$ 20 nodes, then it suddenly stops working. This is because of the L2 halving effects described in section II: with a large parallelism degree, the L2 cache available for each tile is less than the required 48KB.

When using 128×128 matrices, the working set is larger than the cache sizes independently of the DDC policy adopted. In this case the incoherent policy (NHN) still works better than the other two policies. In this test is possible to observe the benefits of using the DDC as a big virtual L3 cache when using the HHN policy: when running sequential programs or parallel ones with small parallelism degree, we may have that the sum of all L2 caches is large enough to contain the working set of the application, so that the performance can be much better than the other two allocation policies. This also means that the speedup (calculated w.r.t. a standard sequential version) is indeed an unfavorably metric for the other two modes. This is shown in Fig. 8, where we compare the speedup and the scalability of using the NHN policy for the case 128×128: using 54 workers thread the application is 42 times faster w.r.t one worker, although the speedup is $\sim 37X$. This is because the reference sequential version uses the standard HHN policy, and is 11% faster than the corresponding one using the NHN policy.

C. TilePro64 as an accelerator

Finally, we tested the many-core in a more realistic environment, where a sequential application uses the Tile*Pro*64 to speed-up just a specific part of the computation, by exploiting the FastFlow accelerator interface. The accelerated part consists of a set of matrices $A_1, A_2, ..., A_N$ that are multiplied by a fixed matrix B; the resulting matrices are then gathered back to the sequential program. This pattern of computation is used in many different numerical and graphical applications.

The results are sketched in Fig. 9 for different matrix sizes considering the HHN (left) and the NHN (right) policies.

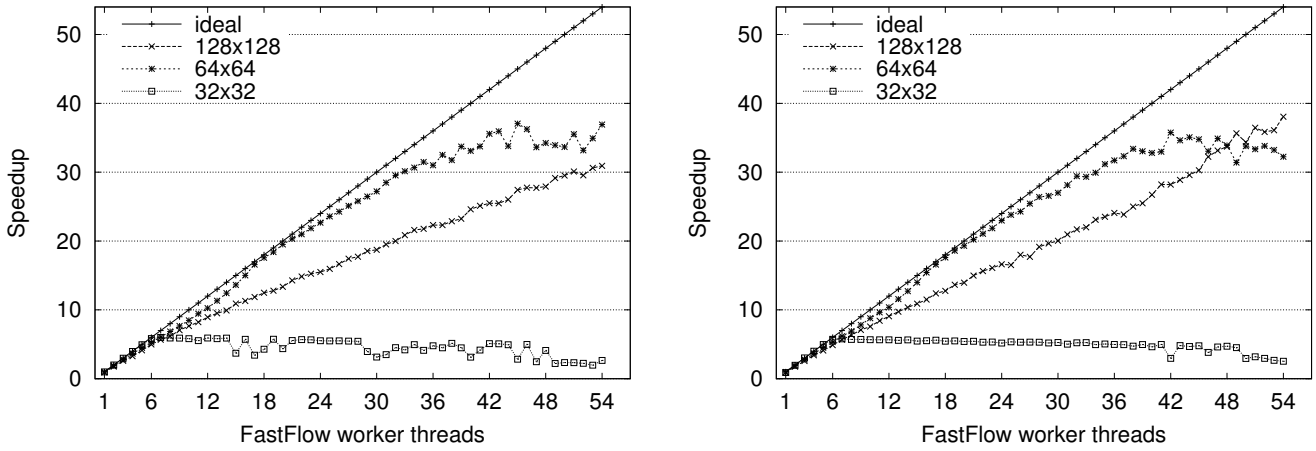

Figure 9: Stream matrix multiplication ($A_i \times B$) using the accelerator considering three different task granularities. Left: Hash Home Node (HHN) policy set. Right: No Home Node (NHN) policy set.

As expected, with very small computations (i.e. matrices of 32×32 integer elements) the FastFlow runtime plus the communication overheads make offloading inconvenient. However, with bigger tasks, we reach the same performance results obtained by the stand-alone configuration, meaning that we are able to completely mask the communication overhead. In this benchmark the HHN policy behaves quite well mainly because the farm workers receive only one matrix (A_i), thus the DDC traffic due to cache contention is reduced. Furthermore, for the 64×64 case, the working set fits in the L2 cache of a single tile, thus no significant performance differences result when using either the NHN or the HHN policy.

VI. Related work

There have been few research efforts evaluating high level parallel programming models for general-purpose many-core architectures, none of them focused on Tile*Pro*64 processor, to the best of our knowledge.

In [11] the authors discuss the porting of the UPC parallel compiler on Tile64, evaluating two alternative implementations for the communication systems: one based on shared memory mechanisms, and one using an MPI layer exploiting the iMesh NoC. The study reveals various optimization techniques based on specific features of the Tile64. The Remote Store Programming model (RSP) has been presented and evaluated on the Tile*Pro*64 in [12]. The authors demonstrated the performance advantages of using the RSP model with respect to the classical cache-coherent programming model using several benchmarks. Our study confirms their results, although by using a completely different approach. In fact, by carefully using low level architectural features of the Tile*Pro*64 processor, which allows to minimize memory load latency by increasing cache locality, we obtained better performance in almost all tests. In [13], the authors presented an adaptive task management scheme for the Tile*Pro*64, also evaluating the performance of some well-known multi-queue work-stealing scheduling strategies. The work mainly focuses on designing a smart adaptive task management code able to dynamically reconfigure itself at runtime, with no focus on the features of the Tile*Pro*64. Other works are mainly focused on the performance evaluation of specific applications optimized for Tile64 and Tile*Pro*64 [14], [15], [16].

The algorithmic skeleton community has proposed various programming frameworks aimed at providing the application programmer with very high level abstractions completely encapsulating parallelism through patterns [4], [17]. Programming frameworks based on the algorithmic skeleton concept exist targeting both clusters/networks of workstations and shared memory multi-core. Some of them also target GPGPUs [18], [19]. To the best of our knowledge, our work is the first aimed at porting a skeleton based programming framework on general purpose many-core architectures, either as stand-alone programming environment or as accelerator.

VII. Lessons Learned and Conclusions

In this paper we presented the porting of the FastFlow parallel programming framework on Tile*Pro*64. The availability of general purpose cores on the Tilera co-processor allowed us to port the FastFlow runtime basically unmodified. We rewrote a very limited part of the original FastFlow code using Tilera specific libraries instead of generic Pthread ones in order to provide performance-effective solutions, especially as far as memory allocation policies are concerned.

Most of our work focused on efficient exploitations of the highly configurable cache coherence mechanism peculiar

of the Tile*Pro*64. We demonstrated that the overhead introduced by the default memory allocation mechanism when using high percentages of available cores may significantly affect the scalability of the parallel application. This means that in order to reach the ideal speedup we need specific application driven solutions. We proved that proper management of software cache coherence mechanisms may be a promising and effective way of programming general purpose many-core architectures with high parallelism degrees, as long as a parallel patterns are used to support the programmer activities. The usage of parallel patterns gives enough knowledge relative to the parallel structure of the program to support the definition of per-pattern memory allocation policies and, equally important, to mask all the architecture-related mechanism management in the FastFlow runtime support, making those techniques effective and completely transparent to the application programmer.

We also tested the possibility to use the Tile*Pro*64 as a general-purpose accelerator. With sufficiently coarse grain tasks we were able to efficiently mask the communication overhead, reaching performance results as good as in the stand-alone configuration. From a software engineering viewpoint, the FastFlow code used to offload computations on this co-processor turns out to be very similar to the code used for self-offloading computations on spare cores of a multi-core architecture. Eventually, the possibility to use a POSIX-based OS and the availability of a stable, well documented programming environment, made our work much easier and faster w.r.t to the effort required to program non general purpose accelerators (GPU). This is even more important in an application programmer perspective: a FastFlow program written for an x86 architecture will run (efficiently) on a Tile*Pro*64 machine with minor modifications to the original code.

REFERENCES

[1] L. Koesterke, J. Boisseau, J. Cazes, K. Milfeld, and D. Stanzione, "Early experiences with the intel many integrated cores accelerated computing technology," in *Proc. of the 2011 TeraGrid Conference: Extreme Digital Discovery*. New York, NY, USA: ACM, 2011, pp. 21:1–21:8.

[2] Tilera Corporation, "TilePro Processor Family," 2012, http://www.tilera.com/products/processors/TILEPro_Family.

[3] T. Mattson, B. Sanders, and B. Massingill, *Patterns for parallel programming*. Addison-Wesley Professional, 2004.

[4] M. Cole, "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming," *Parallel Computing*, vol. 30, no. 3, pp. 389–406, 2004.

[5] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati, "Accelerating code on multi-cores with fastflow," in *Proc. of 17th Intl. Euro-Par 2011 Parallel Processing*, ser. LNCS, vol. 6853. Springer, 2011, pp. 170–181.

[6] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. Brown, and A. Agarwal, "On-chip interconnection architecture of the tile processor," *Micro, IEEE*, vol. 27, no. 5, pp. 15 –31, sept.-oct. 2007.

[7] Tilera Corporation, *Tile Processor User Architecture Manual*, 2011.

[8] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, "Fastflow: high-level and efficient streaming on multi-core," in *Programming Multi-core and Many-core Computing Systems*, ser. Parallel and Distributed Computing. Wiley, 2012, ch. 13.

[9] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati, "An efficient unbounded lock-free queue for multi-core systems," in *Proc. of 18th Intl. Euro-Par 2012 Parallel Processing*, 2012, pp. 662–673.

[10] M. Aldinucci, M. Danelutto, and M. Torquati, "Fastflow tutorial," Università di Pisa, Dipartimento di Informatica, Italy, Tech. Rep. TR-12-04, 2012.

[11] O. Serres, A. Anbar, S. Merchant, and T. El-Ghazawi, "Experiences with upc on tile-64 processor," in *Proc. of the 2011 IEEE Aerospace Conference*, ser. AERO '11, Washington, DC, USA, 2011, pp. 1–9.

[12] H. Hoffmann, D. Wentzlaff, and A. Agarwal, "Remote store programming," in *High Performance Embedded Architectures and Compilers*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, vol. 5952, pp. 3–17.

[13] C. T. D. Waddington and K. Sivaramakrishnan, "Scalable lightweight task management for mimd processor," in *Systems for Future Multicore Architectures, EuroSys workshop, Salzburg, Austria*, April 2011, pp. 1–6.

[14] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele, "Power and performance evaluation of memcached on the tilepro64 architecture," *Sustainable Computing: Informatics and Systems*, vol. 2, no. 2, 2012.

[15] C. Villalpando, A. Johnson, R. Some, J. Oberlin, and S. Goldberg, "Investigation of the tilera processor for real time hazard detection and avoidance on the altair lunar lander," in *Aerospace Conference, 2010 IEEE*, march 2010, pp. 1 –9.

[16] C. Yan, F. Dai, and Y. Zhang, "Parallel deblocking filter for h.264/avc on the tilera many-core systems," in *Proc. of the 17th Inter. Conf. on Advances in multimedia modeling*, ser. MMM'11, Berlin, 2011.

[17] H. González-Vélez and M. Leyton, "A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers," *Software: Practice and Experience*, vol. 40, no. 12, pp. 1135–1160, 2010.

[18] J. Enmyren and C. W. Kessler, "Skepu: a multi-backend skeleton programming library for multi-gpu systems," in *Proc. of the 4th Inter. workshop on High-level parallel programming and applications*, ser. HLPP '10, New York, NY, USA, 2010.

[19] S. Ernsting and H. Kuchen, "Algorithmic skeletons for multi-core, multi-gpu systems and clusters," *Int. J. High Perform. Comput. Netw.*, vol. 7, no. 2, pp. 129–138, Apr. 2012.