Run-time Mechanisms for Fine-Grained Parallelism on Network Processors: the TILEPro64 Experience

Daniele Buono and Gabriele Mencagli
Department of Computer Science, University of Pisa
Largo B. Pontecorvo, 3, I-56127, Pisa, Italy
Email: {d.buono, mencagli}@di.unipi.it

***Abstract*—The efficient parallelization of very fine-grained computations is an old problem still challenging also on modern shared memory architectures. Scalable parallelizations are possible if the base mechanisms provided by the run-time support (for inter-thread/inter-process synchronization/communication) are carefully designed and developed on top of parallel architectures. This requires a deep knowledge of the hardware behavior and the interaction patterns used by the parallelism paradigms. In this paper we present our experience in developing efficient inter-thread interaction mechanisms on the Tilera TILEPro64 network processor. Although it is a domain-specific parallel architecture, the TILEPro64 represents a notable example of how advanced architectural structures, such as user-accessible on-chip interconnection networks and configurable cache coherence protocols, are of great importance to design lightweight cooperation mechanisms enabling efficient parallel implementations of fine-grained problems. The paper presents our ideas and an experimental evaluation that compares our proposals with other existing run-time supports.**

***Keywords*—Network Processors, Parallel Computing, Fine-grained Parallelism, Run-time Supports**

I. Introduction

An increasing number of emerging applications, such as network traffic and sensor data processing, e-business transactions monitoring, and real-time analysis of data streams from social media, are characterized by the presence of fine-grained computations that need to be efficiently parallelized to meet strong throughput and latency requirements.

The design of efficient, scalable fine-grained parallelizations is a challenging issue on modern multi-/many-core architectures. In these applications the execution is communication bound: the frequency of synchronization/communication is very high and each individual task is small in size and execution time. In this context the overhead of setting up the run-time system and executing core-to-core synchronization can easily exceed the parallelization benefit, with the risk of reducing the performance instead of improving it. For these reasons, fine-grained parallelism requires support both from the hardware architecture and the run-time system. Fast communication and synchronization mechanisms can not rely on standard techniques [4], [6], [5]. As an example pthread mutexes and condition variables are not effective, since they can easily take thousands of clock cycles that may represent a significant portion of the execution time.

Over the last years a large research effort has been devoted to studying efficient solutions on multi-core architectures. *Lock-free* queues represent a de-facto standard for low latency inter-thread interaction. Several research works [4], [6] have investigated the design of efficient queues based on lock-free shared data structures and their porting on today's multicores. These works are based on very low-level optimizations aimed at hiding latencies of the cache coherence subsystem and improving the efficacy of the hardware prefetcher.

In this work we investigate the design of lightweight mechanisms on network processors. Network processors are application-specific microprocessors targeting network domain problems such as pattern matching and classification, route lookup, data manipulation, intrusion detection, and queue management. The interesting aspect of network processors is the availability of very specific architectural structures that can be exploited to speed up inter-core cooperation. Notably, configurable cache coherence protocols and the presence of user-accessible on-chip core-to-core interconnection networks enlarge the space of possible optimizations, leading to very efficient ways to conceive run-time mechanisms on these architectures.

This paper describes our experience on the TILEPro64 network processor [12]. Inspired by the classic mechanisms used in networking applications to exchange packet descriptors, we define efficient mechanisms (send and receive operations) that rely on the architectural facilities available on this architecture. We compare our solutions with standard techniques based on pthread synchronization and the lock-free queues made available by the `FastFlow` parallel programming framework [4], [1]. The results confirm that the architectural facilities available on network processors provide a further degree of freedom left to the run-time support designer to develop synchronization mechanisms that outperform standard solutions and support scalable parallelizations of very fine-grained problems.

The organization of this paper is the following. Sect. II presents a brief overview of similar works. Sect. III introduces the features of network processors and the TILEPro64 architecture. Sect. IV outlines the design of our run-time mechanisms. Sect. V provides and experimental evaluation of our work by studying micro-benchmarks and a complete parallel application on the TILEPro64. Finally Sect. VI concludes this paper.

II. Related Work

In the literature a common approach to low latency synchronization on multicores consists in *lock-free* concurrent data structures used to exchange references to shared data between cores. In [10] the authors propose a single-producer

single-consumer lock-free queue targeting data traffic monitoring in high-speed networks. The rationale is to reduce memory access overhead by improving the cache locality of accessing the control variables of the queue. Another similar approach has been described in [7], which is based on cache-aware optimizations and lazy updates to decrease the communication overhead. In [6] the FastForward framework for pipeline parallel applications has been proposed by relying on fast queue/dequeue operations properly optimized for the cache coherence protocols of modern multicores. Similar ideas have been adopted by the low-level communication support of `FastFlow`, a parallel programming framework based on algorithmic skeletons [4], [1].

In this paper we focus on the space of optimizations for fine-grained parallelism offered by network processors [13] with special attention on the TILEPro64 architecture. As described in [13], programming models of networking applications exploit lightweight cooperation mechanisms among cores and specialized co-processors based on the exchange of packet descriptors through special on-chip interconnection networks. Similar mechanisms can be adopted to design run-time supports for general-purpose parallel programs, not limited to the network domain only. Although the porting of some parallel programming frameworks on TILEPro64 has been discussed in the past [5], their mechanisms do not exploit all the features of the underlying architecture in terms of cache coherence protocols and on-chip interconnection networks. Further optimizations are possible, that lay the groundwork for lightweight mechanisms supporting very fine-grained parallelism.

III. Network Processors: Technological Trend and Programming Models

The technological roadmap of network processors started from general-purpose CPUs performing classical routing protocols and switching functionalities [13], [8]. To improve the scalability with high bandwidth traffics, the second generation of network processors exploited specialized hardware (ASICs chips - *Application Specific Integrated Circuits*) to interpret networking functionalities directly at the hardware level. Despite their initial success, ASIC-based network processors were hard to design and lacked the flexibility and extensibility to incorporate additional features and to cope with the dynamicity of modern communication protocols and internet traffics [8].

Over the recent years, network processors have evolved to combine high-speed co-processors, specialized for common networking tasks, with programmable cores, in an attempt to achieve the trade-off between high performance and flexibility [13]. Examples are the IBM PowerEN, Intel IXP and more recently Broadcom XLP and the Tilera architectures. The trend is to have heterogeneous multicores with the following distinguishable features:

- a high number of simple, often in-order cores with support to hardware multi-threading;
- the presence of a set of integrated co-processors responsible to accelerate specific tasks, e.g. checksum, compression/decompression, encryption and authentication, regular expressions, XML parsing and so on;
- a multi-level cache hierarchy composed of private only or private and shared levels of cache with configurable coherence mechanisms;
- a uniform and low latency cooperation mechanism among cores/co-processors which exploits dedicated on-chip interconnection networks.

The heterogeneity of network processors and the need of high bandwidth and low latency pose serious problems to software design and development. The design of networking applications has followed two basic parallelism paradigms. The first model, called *run-to-completion* [3], is based on the *task-farm* paradigm in which incoming packets are dispatched to a pool of identical processing engines executed on general-purpose cores. The engines execute the same functions on different packets possibly by interacting with co-processors for the acceleration of specific tasks. The second model derives from the *pipeline* parallelism paradigm, sketched in Fig. 1. Each engine executes a part of the data-plane processing on each packet by possibly interacting with specialized co-processors. Once a packet has been processed by an engine, it is passed to the next stage of the pipeline. The two models have different features in terms of load balancing (easier on task farming) and to reduce the contention on co-processors (better in pipelining).

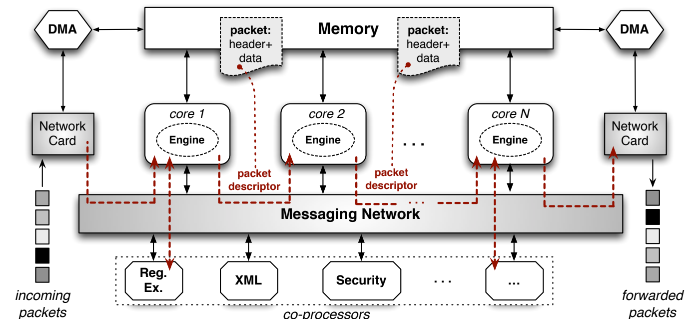


Fig. 1: Pipeline model for packet processing on network processors and the use of on-chip messaging networks.

On both the models a certain degree of coordination is required to distribute packets and to offload part of the computation on the co-processors. Furthermore, the functions applied on packets usually have a fine computational grain. Instead of using costly mechanisms such as spin-locks, mutexes and semaphores, last-generation network processors offer alternative ways to perform the cooperation between cores/co-processors. An interesting approach is represented by on-chip interconnection networks (referred to as *Messaging Networks*) used to exchange messages containing packet descriptors, i.e. the packet headers and the initial memory address of the packet in memory (see Fig. 1). The transmission of packet descriptors is performed over the messaging network by skipping all the shared memory and cache hierarchy levels, thus exploiting the on-chip interconnection to limit the memory contention by sharply reducing the communication latency. Bus, Ring or Mesh networks are available on modern network processors like Broadcom XLP (formerly Netlogic), Cavium and Tilera. In the next part of this section we will provide a detailed description of the Tilera TILEPro64 architecture.

A. The TILEPro64 Network Processor

The TILEPro64 [12] is equipped with 64 identical processing cores (called *tiles*) interconnected by an on-chip network named Tilera *iMesh*. Each link consists of two 32-bit-wide unidirectional physical links carrying the traffic in both the directions. Each tile is composed of: *(a)* a 3-way VLIW in-order processor running at 866MHz with a single thread context, *(b)* a private cache subsystem composed of 16KB L1i, 8KB L1d and 64KB L2, and *(c)* a switch for the interconnection with the iMesh network. To sustain the memory bandwidth requirements of the 64 tiles, four DDR2 memory controllers are placed at the edges of the chip, as shown in Fig. 2. The TILEPro64 is mounted on a PCI express card of a host machine and it is equipped with on-chip PCIe and network controllers.

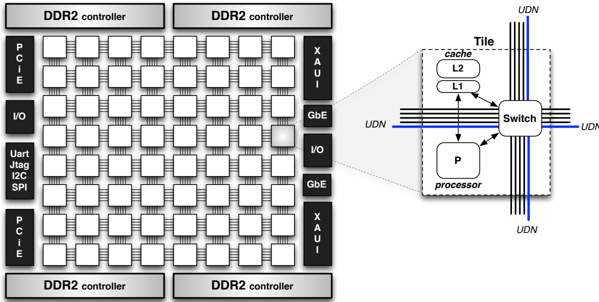


Fig. 2: TILEPro64 architecture: interconnection networks and the structure of a generic tile.

The TILEPro64 is a peculiar architecture. The rationale of its design is to sacrifice the number of co-processors to increase the chip area available to general-purpose cores. However, the TILEPro64 shares interesting features with other network processors, mainly the presence of on-chip messaging networks and configurable cache coherence policies.

The iMesh network is composed of five independent 2D meshes each one carrying a different kind of traffic. The *Memory Dynamic Network* (MDN) is dedicated to transfers between tiles and the four memory controllers. Other two meshes (IDN and CDN) are dedicated to I/O transfers and the cache coherence protocol. The *User Dynamic Network* (UDN) supports the explicit transfers of small messages (up to 128 32-bit words) among tiles under application programmer control. Each tile has five UDN hardware queues connected directly to the processor registers. Special assembler instructions are provided to perform the enqueue/dequeue and the transmission over UDN. The UDN serves user-land processes or threads, providing a flexible and low latency cooperation mechanism. This is a departure from the standard multi-core design, in which the user-land interaction between threads is possible only through shared memory [12].

The TILEPro64 provides a flexible cache subsystem named *Dynamic Distributed Cache* (DDC) [12]. A tile T_i is called the *home node* for a cache line x (64 bytes) if when a miss occurs on that tile, the line is directly loaded from the memory. A miss from a different tile T_j is served by the home tile T_i that transmits the line x from its L2 cache to the T_j's L2 cache. Therefore, for that cache line the L2 cache of the home tile acts as a virtual L3 cache for all the other tiles. The home tile is responsible to handle the coherence of that line and to maintain the updated copy in its L2 cache, i.e. if T_j writes the cache line x, the modified words are forwarded to the home tile T_i that updates its copy and transmits the invalidation messages to the other tiles (if they exist) that hold a copy in their private L1/L2 caches. The TILEPro64 supports a distributed hash-based homing, in which cache lines are homed on different tiles by hashing their base addresses. However, the programmer is also able to manually set the home tile for all the cache lines belonging to the same virtual memory page.

IV. Lightweight Run-time Mechanisms for Fine-Grained Parallelism

In this section we will exploit the architectural features of the TILEPro64 to design efficient run-time support mechanisms for general-purpose fine-grained parallel computations.

A clear and well-defined structuring of run-time supports is possible if parallel programs instantiate well-known parallelism patterns with a precise semantics in terms of communication/synchronization between parallel entities and their role in the overall computation. Notable examples are task-farm, pipeline and data-parallel patterns like map, reduce and stencils. According to this approach, known in the literature as *Structured Parallel Programming* (SPP) [9], the run-time support consists of a limited set of base mechanisms for point-to-point and asymmetric distributions, non-determinism and collective operations like multicast, scatter, gather, and so on.

We start with a very basic mechanism, that we call *communication interface*, used by cores to synchronize and exchange messages. The mechanism provides a point-to-point communication between two partners with a buffer of one position. Fig. 3 describes the `send` and the `receive` primitives.

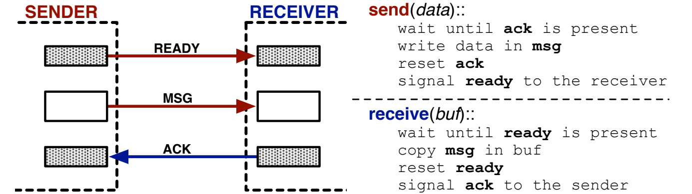


Fig. 3: Abstract definition of the `send` and `receive` primitives on the communication interface.

The pseudo-code uses two boolean events: *(i)* the *ready* event specifies the presence of a new message, *(ii)* the *ack* represents the reception of the last transmitted message. The signal operation sets the event to true, the reset one does the opposite. For a correct behavior, the ready and the ack events are initialized to *false* and *true* respectively. By assuming that the functionalities of a parallel program share part of their address space (e.g. they are implemented as threads), the message *msg* can be a memory pointer to a shared data structure. As discussed in previous works [4], [6], this approach is general and makes it possible to implement work distribution, result collection and other interactions by exchanging data structures *by reference*. Throughout this paper we assume that the message transmitted on the communication interface is a single machine word representing a memory pointer.

In the next parts of this section we will study two implementations of this base mechanism on TILEPro64: the first is

based on shared memory variables and a proper configuration of the cache coherence directives to minimize the communication latency; the second exploits the UDN facility to exchange messages using a separated interconnection network.

A. Communication interface on Shared Memory

The first implementation consists in a symmetric communication mechanism based on shared memory variables. The ready and the ack events are implemented by two boolean flags (initialized to 0 and 1 respectively) and the waiting of an event is implemented by a while-loop on the corresponding flag. The shared variable *msg* is the message word (a memory pointer) transmitted from the sender to the receiver. The code of the `send` and `receive` operations is shown in Fig. 4.

```
send(cm_int, data)::
1  while(cm_int->ack == 0);
2  cm_int->msg = data;
3  cm_int->ack = 0;
4  cm_int->ready = 1;
```

```
receive(cm_int, buf)::
1  while(cm_int->ready == 0);
2  buf = cm_int->msg;
3  cm_int->ready = 0;
4  cm_int->ack = 1;
```

Fig. 4: Shared memory implementation of the symmetric communication mechanism.

Let us consider an abstract multi-processor architecture $\mathcal{M}$ respecting the Sequential Consistency memory model [2]. Accordingly, load/store instructions of the same processor are executed in the program order and they can be interleaved with instructions of different processors in any sequential order. On this abstract architecture the following result holds:

Proposition 1. *(Correctness) The* `send` *and* `receive` *algorithms executed on $\mathcal{M}$ implement a lock free single-producer single-consumer shared buffer of one position.*

Proof sketch: Initially $(ready, ack) = (0, 1)$, i.e. the sender can proceed by executing the `send` while the receiver is eventually waiting on line 1 of the `receive`. The sender writes the data in the msg field and sets the flags such that $(0, 1) \rightarrow (1, 0)$. Now the receiver is the only one of the two partners that can execute the communication primitive. It reads the message and copies it in a private variable buf, and sets the flags such that $(1, 0) \rightarrow (0, 1)$ going back to the initial condition. It is worth noting that row 2 in the `send` must be executed after ack is equal to 1 (otherwise the new message can overwrite a previous and possible unreceived message), and row 4 after row 2 (the $ready$ must be set to 1 after the store of the message in msg is visible to the receiver). Similarly, row 2 in the `receive` must be executed if and only if $ready$ is equal to 1, and row 4 after row 2 (saving the message in the private variable before it is overwritten by the sender). ■

The TILEPro64 architecture adopts a weak memory consistency model [12]. Loads and stores to different addresses issued by the same core can become visible to the other cores in a different order w.r.t the program one. To execute correctly the communication primitives on this architecture, we have to force the ordering of the instructions and their visibility to the other cores. Both in the `send` and in the `receive` algorithm we have to execute a *memory fence* between rows 3 and 4. It is worth noting that the store in the msg variable and the load of msg at row 2 in the `send` and in the `receive` algorithms can never overpass the load of ack and ready flag at row 1 respectively, since each core of the TILEPro64 processes the instructions in-order.

One of the most important features of TILEPro64 is its configurable cache coherence mechanism (see Sect. III-A). The default strategy, referred to as *hash-for-home* [11] (see Fig. 5), assigns the home nodes for the cache lines by hashing their base addresses: i.e. cache lines are distributed across many tiles by load balancing memory accesses across several caches. Accordingly, the cache line of the communication interface is homed at a generic tile T_k the can be different from the sender's tile T_i and the receiver's one T_j. During the execution of a communication primitive (e.g. a `send` by T_i) the cache line is modified causing a write-through of the written words to the home tile T_k and the invalidation of the cache line copy in the T_j's cache. The successive read by T_j (e.g. during the spinning on the ready flag) causes a request to the home tile and the transmission of the entire cache line back to the T_j's cache. This interaction pattern is represented in Fig. 5. A similar behavior can be described for the execution of the `receive` operation performed by T_j.

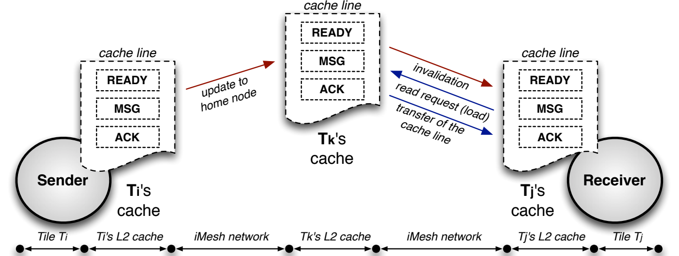


Fig. 5: Cache lines and cache coherence traffic with the hash-for-home strategy: execution of the `send` primitive.

With the hash-for-home strategy we have one invalidation and the transfer of the cache line at the end of the execution of each communication primitive. This behavior gives the clue for an important optimization to reduce the communication latency: the modified words of the communication interface should be transmitted directly to the L2 cache of the tile that needs to read these values. To do that, we adopt a *fixed-homing* strategy in which the home tile of a cache line is defined by software using a specific API provided by the *Tilera Multicore Library* (TMC) [11]. For the ready flag and the message word the ideal home node is the receiver's tile. However the opposite holds for the ack flag. Tab. I shows the activities performed on the fields of the communication interface.

	Ready	Ack	Msg
Sender	Write Only	Read and Write	Write Only
Receiver	Read and Write	Write Only	Read Only

TABLE I: Reading and writing activities on the fields of the communication interface.

Since each cache line can have exactly one home tile, a solution consists in partitioning the fields of the communication interface in two different cache lines: the first one contains only the ack flag; the second cache line contains the ready flag and the message word. A clever allocation consists in homing the cache line of the ack flag on the sender's tile while the other cache line is homed on the receiver's L2 cache. The new

interaction pattern is depicted in Fig. 6. During a `send`, the ready flag and the message word are transmitted directly from the sender's tile to the receiver's one, which is the home node of that cache line (it owns the updated copy). The opposite behavior occurs during the execution of the `receive`, when the new value of the ack flag is transmitted directly from the receiver to the sender's L2 cache.

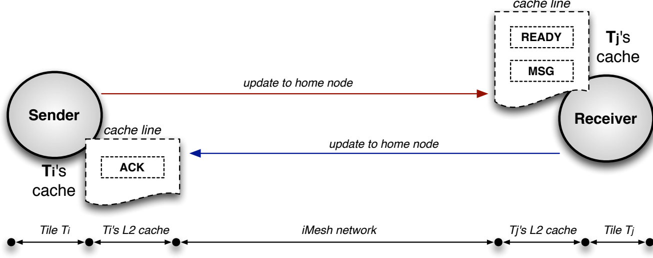
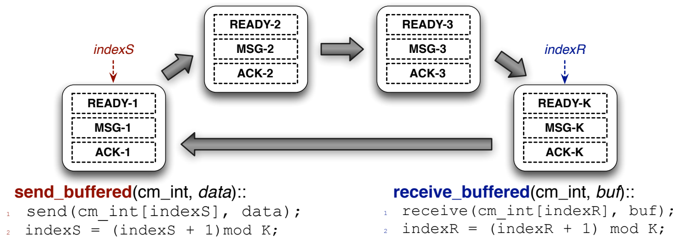


Fig. 6: Cache lines and cache coherence traffic with the fixed-homing strategy.

Furthermore, this allocation makes it possible to avoid cache invalidation messages. The store instructions on the write-only fields of the communication interface, i.e. the ack cache line for the receiver and the ready+msg cache line for the sender, can be performed with the `no-allocate-on-write` flag available in the Tilera's instruction set. Store instructions with this flag enabled do not bring the cache line into the L2 cache in the case of a write miss. In this way, the sender and the receiver do not need to transfer their write-only part of the communication interface into their L2 cache, with a further reduction in the traffic on the iMesh network.

1) Generalizations of the mechanism: the basic symmetric communication interface on shared memory can be used to emulate several possible generalizations of the mechanism, notably *(a)* communications with more than one buffer position and, *(b)* asymmetric many-to-one communications.

We denote by $K \geq 1$ the *asynchrony degree*, i.e. the maximum number of messages that a sender can send without waiting for the first sent message being received. In the basic implementation of our communication interface K is equal to 1. A higher asynchrony degree can be obtained by using K symmetric communication interfaces used in a round-robin fashion by the sender and the receiver, as depicted in Fig. 7. The sender and the receiver have two private variables `indexS`



send_buffered(cm_int, *data*)::

```
1 send(cm_int[indexS], data);
2 indexS = (indexS + 1)mod K;
```

receive_buffered(cm_int, *buf*)::

```
1 receive(cm_int[indexR], buf);
2 indexR = (indexR + 1) mod K;
```

Fig. 7: Implementation of a K-buffered symmetric communication with K communication interfaces used in a round-robin fashion.

and `indexR` (both initialized to zero) which denote the next communication interface to use. Each time the sender wants to transmit a new message, the interface with index `indexS` is selected and the `send` primitive is executed on it (according to the same pseudo-code of Fig. 4) and `indexS` is incremented by 1 modulo K. Symmetric actions are performed on the receiver's side using the private variable `indexR`.

The second generalization is a form of *non-determinism* consisting in an asymmetric *many-to-one* communication interface with many senders and one receiver. By executing a `receive` operation it is possible to receive a message from any source selected by a strategy invisible to the programmer (it is implemented in the run-time support).

Our communication interface can be easily extended to cover the asymmetric behavior. An asymmetric interface serving N senders can be implemented by N single-producer single-consumer interfaces. The `receive` primitive must be modified in order to implement the non-deterministic selection (see Fig. 8). Instead of a while-loop on a single ready flag, the asymmetric version scans all the flags waiting for at least one ready becoming equal to 1. Once selected one of the ready flag, the implementation proceeds exactly as in the symmetric case, as shown in Fig. 8. The wait-until statement can be implemented as a while-loop on the set of ready flags possibly with a fair selection policy. To exploit the cache optimizations discussed before, the pairs $\{\langle ready_i, msg_i \rangle | i = 1, \ldots, N\}$ can be grouped in a set of cache lines homed in the receiver's tile. Similarly, the ack flags $\{\langle ack_i \rangle | i = 1, \ldots, N\}$ are allocated in separated cache lines each one homed in the corresponding sender's tile.

receive_asym(cm_int, *buf*)::

```
1 i ← wait until at least one Ready == 1;
  //Let i the selected sender
2 buf = cm_int->msg[i];
3 cm_int->ready[i] = 0;
4 cm_int->ack[i] = 1;
```

Fig. 8: Pseudo-code of the `receive` primitive on the asymmetric communication interface (many-to-one).

B. Exploiting the User Dynamic Network

The hardware facilities of the TILEPro64 can be exploited to provide an alternative implementation of the communication interface without using shared memory variables for the synchronization and the message transmission. This implementation relies on the UDN on-chip network (see Sect. III-A). Every tile can transmit a message composed of one header word and the payload by specifying: *(i)* the x- and y-coordinates of the destination tile; *(ii)* a *tag* associated with the message. When a new message arrives at the destination tile, the switch unit inspects the tag and forwards the message to the corresponding UDN hardware queue. If a tag miss occurs, the message is dispatched to the special *catch-all* queue.

The UDN implementation is based on using two UDN queues, one on the sender's side and one on the receiver's side, named respectively *ack_queue* and *msg_queue*. The ack event is implemented by the reception of a special word (`ack`) from the ack_queue. The reception of a message from the msg_queue by the receiver corresponds to both the ready event and the copy of the message in the temporary variable *buf* of the receiver. This idea is sketched in Fig. 9.

During the initialization of the communication interface (e.g. by calling a special `initialize` function), the receiver sends a first `ack` to the sender. When the sender executes the `send` primitive, it must wait for the reception of the ack, and then it can transmit the message to the destination tile. The receiver waits for the reception of a new message from its input msg_queue. When a new message is received, the receiver notifies the ack to the sender's ack_queue. In the pseudo-code of Fig. 9 `push` and `pop` are two macros for the corresponding assembler instructions implementing the transmission and the reception from the corresponding UDN queue. The two instructions have a blocking semantics: i.e. the `push` returns if and only if the destination queue has enough space to enqueue the new message; similarly the `pop` blocks the calling thread as long as the target input queue is empty.

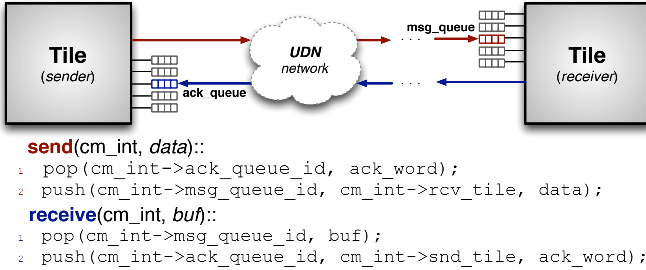


Fig. 9: UDN implementation of the symmetric communication interface.

Although the synchronization and the message transmission exploit the UDN network only, some shared data between the sender and the receiver are still necessary, such as the identifiers of the msg_queue and the ack_queue, and the coordinates of the sender's and the receiver's tiles. However, such data are defined during the initialization of the communication interface, and for the successive `send` and `receive` invocations they are read-only data present in both the L1d and L2 private caches of the sender and the receiver. In conclusion, no memory or coherence traffic is generated by the UDN communication support.

1) Generalizations of the mechanism: an asynchrony degree greater than one can be obtained by using the buffering capability of the UDN queues and by slightly changing the initialization of the communication interface. Instead of generating a single ack message during the initialization, the receiver can initialize the communication interface by sending K `ack` words to the sender's ack_queue. Each ack corresponds to the possibility to transmit a new message by the sender. Since the buffering space of a UDN queue is of 128 32-bit words, the maximum asynchrony degree is $K = 128$.

Support to many-to-one communications can be achieved through the hardware de-multiplexing of the UDN network. The asymmetric communication interface with N senders and one receiver uses N ack_queues, one for each sender, and one msg_queue on the receiver as shown in Fig. 10. The receiver at the application-level must be able to distinguish the sender of each message, in order to transmit the `ack` to the corresponding ack_queue. Therefore, in the asymmetric implementation each message is composed of a pair of words, the first is the sender identifier, the second is the real message word (a pointer to a data structure shared between the sender and the receiver). During the initialization, the receiver transmits one or more `ack` words to each source, in order to give a proper asynchrony degree to the senders. By assuming the same asynchrony degree per sender, the maximum value of K is $128/(2\,N)$.

Despite its flexibility, there are some constraints imposed by the UDN implementation. The number of communication interfaces that can be used per tile is limited by the number of UDN queues (we have four queues per tile plus the catch-all queue). To remove this constraint, generalizations of the mechanisms that use both shared memory and UDN are possible. For instance, a set of *logical* communication interfaces can be implemented on a smaller number of *physical* UDN queues. Each message now contains also the identifier of the communication interface. The reception of a message is treated as an interrupt: a run-time support handler is called each time a new message is present in one of the UDN queues; the message is inspected to determine the corresponding communication interface, and the message word is copied in a local buffer in memory associated with the communication interface. In this paper we do not discuss such implementation, which will be left to our future work.

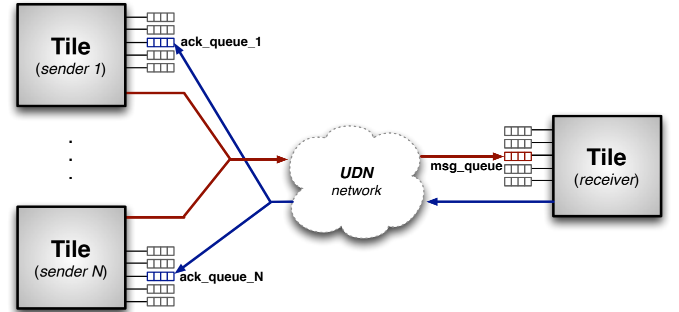


Fig. 10: Asymmetric (many-to-one) communication interface on UDN.

V. Experiments

In this section we propose two classes of experiments. The first one consists in a set of micro-benchmarks in which we study the communication latency of different implementations of the base mechanisms using the shared memory and the UDN supports. The second experiment is a fine-grained parallel application. All the experiments have been compiled with the `tile-gcc` cross compiler using the `−O3` optimization flag.

A. Micro-benchmarks of the Communication Latency

The latency benchmarks are carried out using a ping-pong scheme. A sender thread transmits a one-word message to a receiver thread executed on a different tile and waits for a reply from the receiver. The receiver receives the message, and sends back a reply to the sender. The benchmark consists of many iterations $\mathcal{I}$. The execution time of a pair of `send/receive` primitives (named $T_{exchange}$) is measured as the completion time of the benchmark T_C divided by the number of iterations, i.e. $T_{exchange} = T_C/\mathcal{I}$. The communication latency to execute a single communication primitive can be estimated by $L_{com} \simeq T_{exchange}/2$.

Fig. 11a provides a graphical view of the results for the symmetric communication interface. We compare the latency achieved by three different run-time supports:

- the shared memory support without cache optimizations (i.e. the hash-for-home strategy shown in Fig. 5). We refer to this implementation as ch_sym_sm;
- the shared memory support (ch_sym_sm_cache) with a clever homing of the cache lines of the communication interface (see Fig. 6);
- the run-time support using the UDN network (denoted by ch_sym_udn).

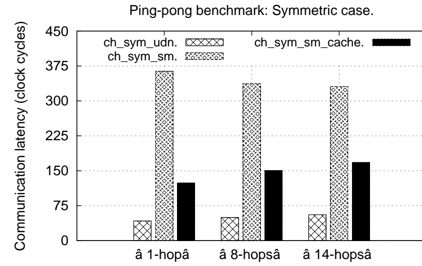
Each experiment consists in $\mathcal{I} = 10^5$ iterations of the ping-pong benchmark executed by two threads. We measure the average communication latency (in clock cycles τ and in μsec), the standard deviation and the maximum measured latency. Since the communication latency depends on the distance in the iMesh network between the sender and the receiver, we take into account three cases: the minimum one (1 hop), the worst case (14 hops) and the average one ($\sqrt{64} = 8$ hops). The numerical results are summarized in Tab. II.

Run-time support	*Hops*	L_{com}			
		Avg		*Std Dev*	*Max* (τ)
		τ	μsec		
ch_sym_udn	1	42.13	0.0486	0.091	42.39
	8	49.63	0.0573	0.087	49.88
	14	55.64	0.0642	0.080	55.84
ch_sym_sm	1	363.55	0.4198	0.2039	363.83
	8	337.01	0.3892	0.1621	337.14
	14	331.06	0.3823	0.1769	331.22
ch_sym_sm_cache	1	124.27	0.1435	0.0848	124.38
	8	150.31	0.1736	0.1121	150.42
	14	168.33	0.1934	0.1081	168.42

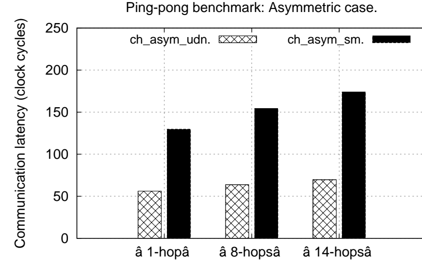
TABLE II: Detailed results of the ping-pong micro-benchmark: symmetric communication.

In general the experiments confirm our initial intuition. First of all the cache optimizations discussed in Sect. IV-A improve the latency of the shared memory support. This is a meaningful result: when a *producer-consumer* pattern is recognized, the homing of the cache lines can be properly set in order to reduce the latency to update the modified cache lines in the home node. For write-only cache lines is more convenient to allocate the home node remotely. In contrast, for read-write data accessible by one single consumer thread, a local homing strategy in the L2 cache of the consumer is the best solution to reduce the coherence traffic. This optimization leads to a $50\% \div 65\%$ improvement w.r.t the basic ch_sym_sm variant. This result shows that the flexibility and configurability of the cache subsystem of the TILEPro64 makes it possible powerful optimizations, especially when the interaction pattern between threads is regular and well-identified.

A meaningful analysis is the comparison between the implementation using shared variables and the one with the UDN support. As we can observe, the UDN implementation considerably outperforms the best shared memory variant. With the UDN support all the phases of the inter-thread cooperation are performed over UDN, without cache lines transfers between tiles. Small messages of few words can be exchanged with the minimum overhead as possible, and they become promptly available to the receiver tile that can use them directly from its internal registers without additional copies. The result is that the communication latency is one third of the one of the best shared memory implementation.



(a) Symmetric communication.



(b) Asymmetric communication.

Fig. 11: Communication latency with the ping-pong micro-benchmark.

As we can expect the communication latency is influenced by the distance between the sender's and the receiver's tiles. The longer the distance (in hops) the greater the latency. This behavior is confirmed by our benchmarks. Apparently, the ch_sym_sm variant without cache optimizations behaves differently: the latency does not increase (even decreases) with a longer distance between the sender and the receiver. The reason is that the home node of the communication interface cache line is a third tile w.r.t the sender's and the receiver's ones (see Fig. 5). Thus, a better latency can be achieved even if the sender and the receiver are farther. In fact, it is the distance between sender and the home node and between the home node and the receiver that plays a decisive role in the latency of this implementation. Since in the hash-for-home policy the home node is assigned by hashing the starting memory address of the communication interface data structure, no control can be applied by the run-time support designer.

To evaluate the asymmetric communication mechanism we consider two experiments. The first one consists in the ping-pong micro-benchmark with one sender and one receiver communicating through an asymmetric communication interface. The idea of this benchmark is to show the additional overhead of the asymmetric mechanism compared to the symmetric one when the communication involves only two threads (one sender and one receiver). The results are shown in Fig. 11b and Tab. III. We consider two different implementations: ch_asym_udn over the UDN network, and the shared memory version ch_asym_sm with the cache homing optimizations discussed in Sect. IV-A1.

In the average case the gain of using the UDN support compared with the shared memory version is more than 50%. Tab. III shows an important metric denoted by Δ, i.e. the

Run-time support	*Hops*	L_{com}				
		Avg		*Std Dev*	*Max* (τ)	Δ
		τ	μsec			
ch_asym_udn	1	56.11	0.0648	0.0387	56.18	24.9%
	8	63.64	0.0735	0.0552	63.74	22.0%
	14	69.63	0.0804	0.0747	69.78	20.1%
ch_asym_sm	1	129.75	0.1498	0.0972	129.88	4.22%
	8	154.23	0.1781	0.0968	154.41	2.54%
	14	173.87	0.2008	0.4851	174.61	3.19%

TABLE III: Detailed results of the ping-pong micro-benchmark: asymmetric communication.

overhead of the asymmetric mechanism w.r.t the corresponding symmetric implementation. It is worth noting that by using the shared memory support, the asymmetric mechanism and the symmetric one provide very similar results (Δ is less than 5%). In contrast, the difference is more remarkable with the UDN support. In this case the asymmetric mechanism features a penalty of 20% in communication latency. The reason is that in the asymmetric communication interface each UDN message is composed of two 32-bit words, the first identifies the sender's tile, the second is the memory pointer passed by the sender to the receiver. In the symmetric interface the first word is not necessary, since the communication interface allows the communication between a static pair of tiles.

The second experiment (Fig. 12) focuses on the efficiency of the asymmetric communication mechanism by variating the number of senders. During the execution only one sender transmits messages to the receiver following the ping-pong scheme, while all the other senders are idle. The goal is to show the overhead of the non-deterministic selection by changing the cardinality of the senders set. In the experiment the distance between the active sender and the receiver is fixed to 1 hop.

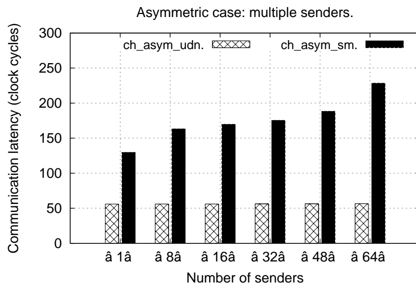


Fig. 12: Overhead of the non-deterministic selection.

As we can note, the UDN support is very efficient. Although there are multiple senders, the messages belonging to the same communication interface are natively demultiplexed on the same hardware queue by the UDN network without additional overhead. Furthermore, the queue is directly accessible by the destination tile through special registers, without reading from the memory. The results show an ideal situation: the latency offered by the asymmetric communication interface does not depend on the number of senders, i.e. no overhead is paid by the presence of multiple idle senders. In contrast, in the shared memory support we need to loop on the ready flags (see Fig. 8) with a cost proportional to the number of senders.

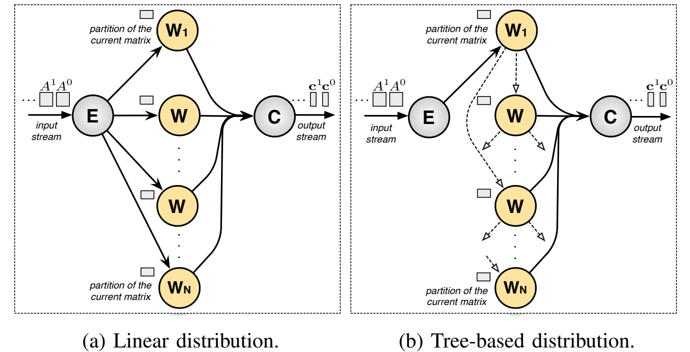
B. A fine-grained parallel application

The concluding part of this paper shows a complete application executed on the TILEPro64. The application operates on an input stream of elements and generates an output stream of results. Each input element is a square matrix of integers: we denote by $A^k \in \mathbb{N}^{M \times M}$ the k-th element of the input stream where M^2 is the size of the matrix. For each matrix the computation applies the *matrix-vector multiplication*, i.e. A^k is multiplied by a constant vector $\mathbf{b} \in \mathbb{N}^M$ and $\mathbf{c}^k \in \mathbb{N}^M$ is the result vector calculated as follows:

$$c_i^k = \sum_{j=1}^{M} a_{ij}^k \cdot b_j$$

The output stream is composed of the sequence of result vectors $\{\mathbf{c}^k\}_{k=1}^N$ where N is the stream length.

We consider a data-parallel parallelization of the problem based on the *map* paradigm. The parallelization scheme consists of a set of threads: an *Emitter* (`E`) and a *Collector* (`C`), respectively responsible to interface the computation with the reception of input elements and the transmission of output results, and a set of *Workers* ({`W`}). Each Worker applies the computation on a partition of the current input matrix. To have independent Workers (as required by the map pattern), each Worker operates on a subset of the rows of the current matrix and calculates the corresponding elements of the result vector by reading the constant vector $\mathbf{b}$ (shared among the Workers). The Emitter is responsible to multicast the initial memory address of the matrix to the Workers (needed to address their partitions). This can be done according to two strategies: *(i)* a *linear distribution* (see Fig. 13a), in which the Emitter transmits the address through a sequence of N communications where N is the cardinality of the Workers set (i.e. the *parallelism degree*); *(ii)* a *tree-based distribution* (see Fig. 13b) with a logarithmic latency, implemented directly by the set of Workers by mapping a tree on the linear array of Workers. Finally, Workers transmit the pointers to their partitions of the result vectors to the Collector using an asymmetric communication interface.



(a) Linear distribution. (b) Tree-based distribution.

Fig. 13: Map parallelization with different distribution strategies.

As said before, in our UDN support the number of communication interfaces per tile is limited by the available number of UDN hardware queues. By using a binary-tree distribution we can use this kind of support for our application. According to the *depth-first* mapping strategy, the first Worker is the root of the tree at level $log_2 N$ (it receives directly from the Emitter), and each Worker j at level i ($0 < i \leq log_2 N$) sends the memory pointer to Workers $j+1$ and $j+2^{i-1}$. In this way each Worker has one input communication interface and at most

three output communication interfaces (one with the Collector and at most two communications for the multicast).

We analyze the application performance with different kinds of run-time supports. We denote by `udn_tree` the implementation using our UDN support and by `sm_tree` the version based on shared memory communications. In both the cases the distribution is performed using the binary-tree topology mapped onto Workers. To make our analysis stronger, we compare our supports with two state-of-the-art solutions. The first one is based on the `FastFlow` lock-free queues [4], [1]: single-producer single-consumer queues between Emitter and Workers, and a multiple-producers single-consumer queue between Workers and the Collector. The second implementation is based on lock-based queues protected by standard POSIX mutexes and condition variables for thread synchronization. Also in these cases we use the binary-tree distribution for the multicast. We are interested in three metrics:

- the *service time* T_S, i.e. the average time between the completion of two successive input matrices;
- the maximum *throughput* B_W, i.e. the highest input data rate that the parallelization can sustain without being a bottleneck. It is calculated as the average number of matrices served per time unit (the inverse of T_S) multiplied by the size of each matrix;
- the *scalability* $\mathcal{S}^{(N)}$ with N Workers is a measure of the relative speed of the parallel computation. It is the ratio between the service time with one Worker and the service time with a parallelism degree equal to N: i.e. $\mathcal{S}^{(N)} = T_S^{(1)}/T_S^{(N)}$.

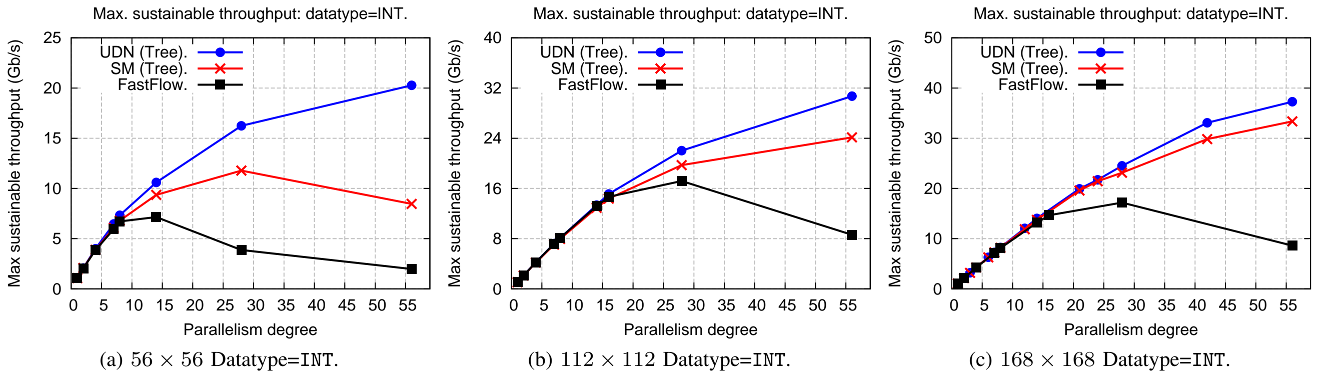
Fig. 14 shows the maximum sustainable throughput with three problem sizes $M = \{56, 112, 168\}$. With the highest parallelism degree $N = 56$ (2 tiles execute the Emitter and the Collector threads and 4 tiles are reserved to the TILEPro64's operating system) these sizes allow to partition each matrix in an equal number of rows per Worker. The results are very interesting. The UDN on-chip network provides a way to exploit additional bandwidth of the iMesh network, by separating the memory and cache coherence traffic due to the execution of matrix-vector multiplications with the inter-thread cooperation traffic. In this way, the efficiency of cooperation mechanisms is not influenced by the under-load memory access latency of the program. The result is that the finer the computation (smaller partitions to Workers) the more the gain of using the UDN support. We achieve an improvement of 58%, 21%, 10% with $M = \{56, 112, 168\}$ compared with our support based on shared memory communication interfaces. Notably, the experiments show that `FastFlow` lock-free queues achieve a lower throughput, and the difference is more remarkable with smaller matrices. The reason is that `FastFlow`, though it has been efficiently ported on the TILEPro64 [5], makes only a partial use of the available hardware features. Firstly, the fixed-homing policy has been applied only to the application level data structures of particular parallelism patterns (actually only the task-farm), and has not been applied to the run-time support mechanisms as proposed in this paper. Furthermore, the UDN network has not been exploited yet in this framework. `FastFlow` can be extended to incorporate these optimizations, which is a future direction of our work.

Tab. IV summarizes the best results achieved by the different implementations. The experiments show the importance of the tree-based distribution. We denote by `sm_linear` the implementation using our shared memory support and the linear distribution strategy. In that case, the delay of the multicast grows linearly with the parallelism degree and the Emitter limits the performance of the overall application. Finally, the implementation based on lock-based queues is completely uneffective with very fine-grained computations.

	56×56			112×112			168×168		
	T_S	B_W	$\mathcal{S}$	T_S	B_W	$\mathcal{S}$	T_S	B_W	$\mathcal{S}$
`udn_tree`	4.9	20.3	19.0	13.07	30.7	28.6	24.2	37.3	34.8
`sm_tree`	11.9	8.46	7.95	16.6	24.1	22.5	27.1	33.4	31.2
`sm_linear`	41.3	2.43	2.28	46.6	8.61	8.03	47.3	19.1	17.8
`FastFlow`	14.0	7.15	6.72	28.2	14.7	14.6	39.3	22.9	21.5
`lck_queue`	43.4	2.31	2.17	79.9	13.2	5.80	115.3	7.83	7.31

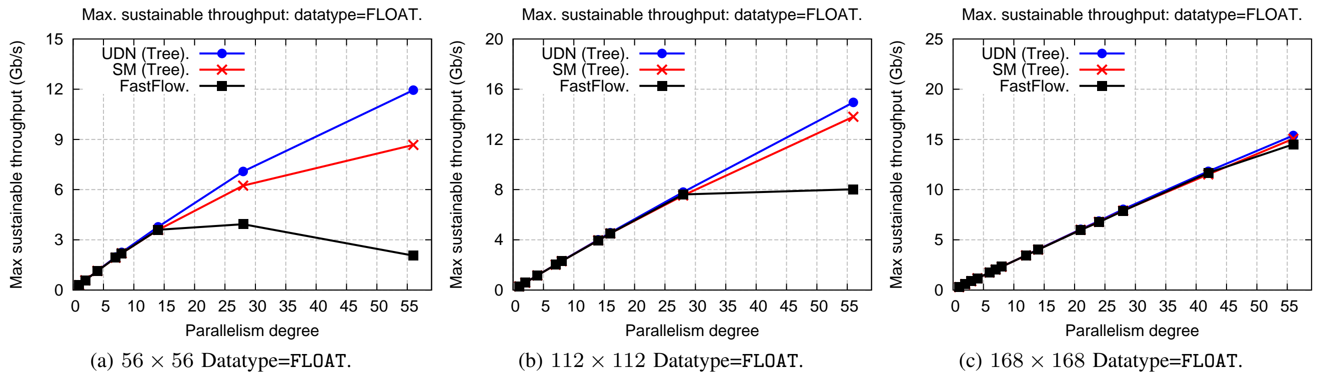
TABLE IV: Best results of different implementations: Datatype=`INT`, T_S and B_W expressed in μsec and Gb/s.

As we can note the scalability increases with bigger problem sizes, when the communication overhead is a smaller portion of the overall execution time. However, it is still far from the ideal one. The reason is that the efficiency is limited by the available memory bandwidth. In fact, though each input matrix is stored in memory by interleaving the accesses among the four on-chip memory controllers (Fig. 2), their aggregate memory bandwidth is not sufficient to sustain a high number of working tiles. To demonstrate this fact we repeated the



(a) 56×56 Datatype=`INT`. (b) 112×112 Datatype=`INT`. (c) 168×168 Datatype=`INT`.

Fig. 14: Maximum sustainable throughput in Gb/s by the map parallelization: matrices of size $M = \{56, 112, 168\}$ and Datatype=`INT`.



(a) 56×56 Datatype=FLOAT.

(b) 112×112 Datatype=FLOAT.

(c) 168×168 Datatype=FLOAT.

Fig. 15: Maximum sustainable throughput in Gb/s by the map parallelization: matrices of size $M = \{56, 112, 168\}$ and Datatype=FLOAT.

experiments with a different data type. Instead of integers, each input matrix is now composed of single precision decimal numbers. This case is meaningful: the matrix size is the same (4 bytes per scalar), but TILEPro64 does not have floating-point units and decimal operations are emulated by software. The result is that we have a coarser grain computation with the same number of cache misses. Thus, the memory controllers are subjected to a lower pressure from the tiles and the parallelization achieves near optimal scalability, see Tab. V.

	56×56			112×112			168×168		
	T_S	B_W	S	T_S	B_W	S	T_S	B_W	S
`udn_tree`	8.40	11.9	40.9	26.9	14.9	51.1	58.7	15.4	52.7
`sm_tree`	11.5	8.67	29.7	29.1	13.8	47.3	59.9	15.1	51.6
`sm_linear`	41.4	2.42	8.29	39.5	10.2	34.8	58.4	15.5	52.9
`FastFlow`	25.5	3.94	13.5	50.1	8.02	27.5	62.3	14.5	49.7
`lck_queue`	67.3	1.49	5.10	126.8	3.17	10.8	159.4	5.67	19.4

TABLE V: Best results of different implementations: Datatype=FLOAT, T_S and B_W expressed in μsec and Gb/s.

Fig. 15 shows the throughput with Datatype=FLOAT. The gain of UDN is remarkable also with small matrix sizes and decreases with larger problem sizes up to achieving the same performance of the shared memory and the `FastFlow` support. In conclusion these experiments show that the UDN is a very interesting architectural feature, suitable to achieve scalable parallelizations of very fine-grained problems.

VI. CONCLUSIONS

Network processors provide architectural facilities not available on traditional multicores, that can be exploited by efficient cooperation mechanisms that enable fine-grained parallelism. This work explains the design principles of low-latency cooperation mechanisms between threads on the TILEPro64, and compares them with other available run-time supports. The results confirm the effectiveness of our mechanisms which are capable of achieving significant improvements in throughput and scalability when dealing with the parallelization of very fine-grained computations. In the future we plan to apply our experience to provide a full parallel programming framework on the TilePro64.

ACKNOWLEDGMENT

We thank Prof. M. Vanneschi for the original ideas inspiring this work, and F. Mariti (B.S. student) for his effort in developing parts of the experiments.

REFERENCES

[1] "The fastflow (ff) parallel programming framework," 2014, http://mc-fastflow.sourceforge.net/.

[2] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *Computer*, vol. 29, no. 12, pp. 66–76, Dec. 1996.

[3] C. Albrecht, R. Hagenau, E. Maehle, A. C. Dring, and A. Herkersdorf, "A comparison of parallel programming models of network processors." in *ARCS Workshops*, ser. LNI, U. Brinkschulte, J. Becker, D. Fey, K.-E. Gropietsch, C. Hochberger, E. Maehle, and T. A. Runkler, Eds., vol. 41. GI, 2004, pp. 390–399.

[4] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati, "An efficient unbounded lock-free queue for multi-core systems," in *Proceedings of the 18th International Conference on Parallel Processing*, ser. Euro-Par'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 662–673.

[5] D. Buono, M. Danelutto, S. Lametti, and M. Torquati, "Parallel patterns for general purpose many-core," in *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, Feb 2013, pp. 131–139.

[6] J. Giacomoni, T. Moseley, and M. Vachharajani, "Fastforward for efficient pipeline parallelism: A cache-optimized concurrent lock-free queue," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '08. New York, NY, USA: ACM, 2008, pp. 43–52.

[7] A. Gidenstam, H. Sundell, and P. Tsigas, "Cache-aware lock-free queues for multiple producers/consumers and weak memory consistency," in *Proceedings of the 14th International Conference on Principles of Distributed Systems*, ser. OPODIS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 302–317.

[8] R. Giladi, *Network Processors: Architecture, Programming, and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

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

[10] P. P. C. Lee, T. Bu, and G. Chandranmenon, "A lock-free, cache-efficient shared ring buffer for multi-core architectures," in *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '09. New York, NY, USA: ACM, 2009, pp. 78–79.

[11] Tilera Corporation, "UG101 - Tile Processor User Architecture Manual," 2011, http://http://www.tilera.com/scm/docs/UG101-User-Architecture-Reference.pdf.

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

[13] K. Yi and J.-L. Gaudiot, "Features of future network processor architectures," in *Modern Computing, 2006. JVA '06. IEEE John Vincent Atanasoff 2006 International Symposium on*, 2006, pp. 69–76.