Optimizing Message-Passing on Multicore Architectures using Hardware Multi-Threading

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

***Abstract*—Shared-memory and message-passing are two opposite models to develop parallel computations. The shared-memory model, adopted by existing frameworks such as OpenMP, represents a *de-facto* standard on multi-/many-core architectures. However, message-passing deserves to be studied for its inherent properties in terms of portability and flexibility as well as for its better ease of debugging. Achieving good performance from the use of messages in shared-memory architectures requires an efficient implementation of the run-time support. This paper investigates the definition of a delegation mechanism on multi-threaded architectures able to: *(i)* overlap communications with calculation phases; *(ii)* parallelize distribution and collective operations. Our ideas have been exemplified using two parallel benchmarks on the Intel Phi, showing that in these applications our message-passing support outperforms MPI and reaches similar performance compared to standard OpenMP implementations.**

***Keywords*—message passing, shared memory, multi-core, communications-calculation overlapping, hardware multi-threading.**

I. Introduction

Traditionally, *message-passing* and *shared-memory* constitute the two fundamental concurrency models giving the basis to the modern programming models for parallel computing. The debate about the most efficient and effective way to develop parallel programs represents a very complicated and controversial issue. In general, the best solution depends on the specific application pattern and the use of communication/synchronization [2], [12]. The result is that existing frameworks adopt one of the concurrency model (notably OpenMP and MPI for shared-memory and message-passing programming respectively), while to choice of the most appropriate model for their own applications is left to programmers [22].

One of the most recognized advantage of the message-passing model is its inherent independence from the underlying architecture, i.e. *it is portable in principle*: provided that the proper run-time support is designed, a message-passing computation can be implemented on a distributed-memory cluster or on a shared-memory multiprocessor like a multi-/many-core chip multiprocessor, or (perhaps the most interesting case for very high parallelism) on a cluster of multi-/many-core nodes.

On shared-memory architectures a comparison between the performance achieved by message-passing and shared-memory implementations of the same benchmark suite is useful to understand the performance issues of the two models in a very pragmatic way. The general result is that masking the communication impact on the overall performance and providing an efficient way to accelerate distribution and collective functionalities are crucial aspects to design message-passing parallel programs able to scale acceptably with large parallelism degrees [14]. In this context, this paper investigates the design issues of message-passing supports on multi-/many-core and provides the following research contributions:

- we describe a general run-time support mechanism based on *communication threads* coupled with the functionalities of parallel programs. Although the idea of using support threads is not new, we discuss very flexible ways to assign communication threads to different sub-parts of the computation which need to be accelerated. *The result is a uniform mechanism to overlap communications with calculation and to parallelize distribution activities and collective operations*;
- we discuss the execution of communication threads on hardware contexts of multi-threaded architectures (notably the recent Intel Phi™). The importance of *Hardware Multi-Threading* (shortly HMT) for parallel programs is debatable. Instead of increasing the maximum parallelism degree of the computation, we propose to use HMT to execute support activities like the execution of point-to-point communications. *This way to use HMT by cooperative threads is essentially new in the context of message-passing programming*.

In Section V we conclude the paper with an experimental evaluation using two parallel benchmarks on the Intel Phi™. The results show that the use of communication threads, properly mapped onto HMT contexts, represents an effective way to properly mask communication latencies and to parallelize distribution activities. The general result is that the performance of our message-passing implementations is similar to corresponding shared-memory versions developed using standard tools (the difference is bounded by 10%) and outperforms classic message-passing frameworks as MPI.

II. Related Work

The analysis of message-passing libraries on multi-/many-core architectures and the performance comparison with shared-memory programming models is a debated issue. Studies like in [2], [12], [22] compare MPI and OpenMP, obtaining results in favor of one of the two paradigms depending on the application. In general they all agree that the main overhead in MPI implementations is given by the communication support.

The idea of using hardware and software supports to speedup communications in message-passing environments is quite established. Zero-copy communications are an important optimization, in which the communication support guarantees that no additional copy between the sender and the receiver buffers is performed during the transmission of a message. Over the last years, several message-passing supports have provided zero-copy communications. These include, but not only: MPICH-PM [15], VMMC-2 [6], AM [19], U-Net [18] and BIP [7].

High-performance interconnections (e.g. Infiniband and Myrinet) are equipped with the architectural support (network adapters) able to offload *partially* or *entirely* communications among nodes, making possible the overlap between communication and calculation phases. Several works analyze the idea of using cores/processors to offload communications in presence of simple interconnection networks (e.g. Gigabit Ethernet). An example is described in [13], where the authors develop a general-purpose communication engine for MPICH. Besides overlapping point-to-point communications, optimized collective operations have been the focus of several researches; an example is described in [10], where the authors discuss an optimized implementation of the primitive `MPI_BCAST`, by exploiting the multicast mechanism natively offered by the Infiniband network.

On multi-/many-core architectures several architectural facilities can be used to accelerate communications. Along this line HMT is an interesting candidate. Its original aim is to optimize the overall processor instruction throughput by masking internal and external latencies inside the CPU pipeline; how to efficiently exploit it in parallel computing is still debated. Different studies [4], [16] show that using HMT to increase the parallelism degree of applications may not offer significant advantages, and sometimes also causes performance degradations basically due to increased contention of those resources that have already limited the performance with a single thread per core.

A more interesting approach is to design *cooperative* threads which *help each other* instead of competing on resources. Some examples are speculative precomputation [3], that enables specialized prefetching activities, prepushing [17] to anticipate cache-to-cache transfers, and multipath execution [20] of different branch directions with multiple threads to reduce misprediction overhead. In this paper we propose to use HMT to accelerate message-passing *inside* shared-memory CPUs. The most similar work to ours is in [8], that presents the idea of using HMT to implement communication threads. However, this work targets distributed-memory architectures, and communication threads are made available at the programmer level (i.e. they are not inside a communication library).

III. Issues for an Efficient Message-Passing on Multicore

When developing parallel programs using message-passing, two important and interrelated aspects deserve to be carefully addressed in order to provide competitive implementations:

- the basic communication primitives, i.e. *send* and *receive*, need to be efficiently implemented in order to minimize their impact on the overall performance;
- message-passing programs make use of *collective and distribution primitives*, notably scatter (for partitioning input data), multicast (for replicating input data), gather (for collecting results), as well as distribution operations based on point-to-point communications (e.g. on-demand and round-robin scheduling). If not properly optimized, they can rapidly become the performance bottleneck when we use high parallelism.

In this paper *we focus on run-time mechanisms to develop efficient message-passing supports on multi-/many-cores.*

A. Lowering and masking communication latencies

To reduce the impact of communications on the application performance we can move in two directions:

1) since *communication latency* is a function of the message size, it can be reduced by minimizing the copies required by a pair of send/receive operations;
2) when an execution pattern characterized by a sequence of calculation and communication phases is recognized, we can *mask communications by overlapping them with calculation*. Naturally, this is possible if the application semantics allows communications to be overlapped with calculation, and if the communication form is *asynchronous* and *non-blocking*.

The two points represent different sides of the problem. The first point is meaningful also when overlapping is not possible, e.g. if the semantics does not allow it or if proper hardware/software support is not available. If it is possible to completely overlap communications with a sufficiently long calculation phase, point (2) is in principle sufficient to make the impact of communication latency null. Finally, the combination of both (1) and (2) makes it possible *to hide communication latency also with fine grain computations*.

A well-known solution to point (1) is by using *zero-copy* communications, i.e. a data transferring mode in which *exactly* one copy from the sender buffer to the receiver buffer is performed during a communication, without any additional copy (e.g. from user space to kernel space). On traditional message-passing systems like distributed-memory architectures, zero-copy communications have been implemented relying on advanced networking infrastructures and devices (e. g. InfiniBand and Myrinet) using advanced network adapters. On shared-memory architectures a zero-copy communication [11] can be performed entirely at the user-space level during the send primitive, as done in state-of-the-art frameworks like in [19].

Communication-computation overlapping is an interesting aspect when developing efficient message-passing supports. Fig. 1 shows feasible temporal situations of a sequence of calculation and communication phases where we assume that each message is the result of the last calculation phase. Fig. 1a depicts a *non-overlapping* situation in which the communication latency is entirely paid. In Fig. 1b the communication latency is *fully masked* by the next calculation phase, which can start in parallel. This situation is possible if: *(i)* the communication semantics is *non-blocking*, i.e. the send primitive returns immediately to the caller before the message has been completely copied into the destination buffer; *(ii)* proper software and architectural supports exist such that the execution of communication primitives can be delegated to proper "units" in charge of performing the copy from the sender buffer to the receiver one in parallel with the calculation. Finally, Fig. 1c shows the situation in which, though overlapping is supported, the length of calculation phase is less than the communication latency, i.e. communications are *partially overlapped* to calculation.

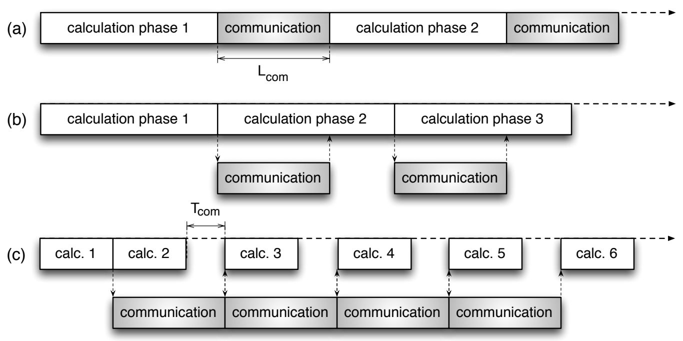


Fig. 1: Cases of communication/calculation overlapping.

As said, communication-computation overlapping requires proper software and architectural supports. More precisely:

- a run-time support mechanism to delegate the execution of communication operations;
- a communication *"unit"* able to execute communication primitives in parallel with calculation phases.

The *delegation mechanism* implements a way to assign the execution of send primitives to a different entity. This mechanism must be efficient, i.e. *with a cost much lower than the communication itself*. An efficient delegation is possible if we assume that the source entity of the parallel program (the one executing the send primitive) and the communication unit exploit a shared-memory cooperation, i.e. passing the delegation parameters (e.g. the message buffer and the receiver identifier) *by-reference* without additional copies. This is possible if they share the same virtual address space or a part of it (the one for the messages and the other communication data structures).

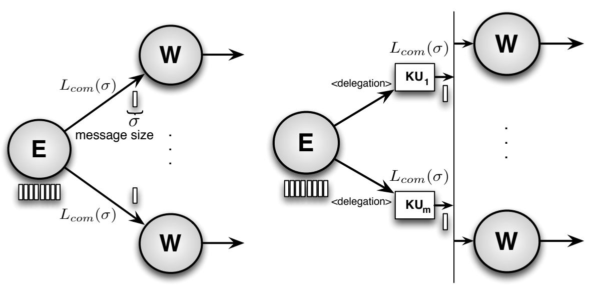
When communication inside shared-memory architectures is concerned, communication units can be concretized using the available architectural facilities:

- a communication unit can be implemented by a *core* of the architecture, dedicated to execute communication primitives delegated by different parts of the parallel computation (e.g. Emitter, Workers and Collector). In this case the core can not be used to increase the parallelism degree of the application;
- instead of using an entire core, a communication unit can be a *hardware context* in the case of HMT CPUs;
- communication primitives can be delegated to specialized *co-processors* if they are available. Some existing multicores, especially network processors, are equipped with hardware accelerators which are amenable to be used in this way. As an example, the series of `Tilera` CPUs [1] are equipped with DMA engines that can be delegated to execute the most costly part of communications, i.e. the copy of the message to the destination buffer of the receiver.

B. Point-to-point distributions and collective operations

Collectives and distributions are critical phases for the performance and scalability of message-passing parallel programs. Optimization of such operations has been the focus of many years of research. On distributed-memory architectures, high-speed collectives are based on *tree-based algorithms* with different shapes depending on the network topology [14].

Also in multi-/many-core architectures, collectives and distributions deserve special attention. As an example, let us consider a point-to-point distribution of a task-farm sketched in Fig. 2a, where an Emitter functionality (`E`) is in charge of *generating* and *scheduling* a stream of tasks to a set of Workers ({`W`}) and results are collected by a Collector functionality (`C`). This example is meaningful to understand that, *point-to-point distributions can be critical to the performance of message-passing parallel programs*. In the scheme of Fig. 2a, the *maximum throughput* B_{max} of the task-farm application is limited by the Emitter when the number of Workers is sufficiently high, i.e. $B_{max} = 1/L_{com}(\sigma)$, with L_{com} the communication latency expressed as a function of the message size σ.



(a) Centralized distribution. (b) Parallel distribution.

Fig. 2: Point-to-point distribution: centralized and parallel versions.

To alleviate this situation the Emitter functionality needs to be properly parallelized. A simple solution consists in doing the message copy in the Workers side. If the receive operation is in charge of copying the message from the sender to the receiver buffer (instead of the opposite as previously discussed), the result is a parallelization of the point-to-point distribution. However this solution increases the service time of the Worker: although the Emitter is no more a bottleneck, the Worker service time is higher since it performs the copy of the input task from the Emitter in addition to the calculation phase. Moreover, it is worth noting that this scheme is not applicable to the Collector (otherwise all the copies of the results would be performed serially by the Collector, becoming a bottleneck as the Emitter in the initial case).

A solution which achieves both parallelization of the Emitter distribution and overlapping with Workers execution consists in a more general utilization of the overlapping mechanism described in Section III-A (see Fig. 2b). In this case, if the Emitter delegates communications to $m \geq 1$ available communication units (denoted by KU in the figure), the maximum throughput of the application becomes theoretically equal to $B_{max} = m/L_{com}(\sigma)$, i.e. m times higher than in the centralized case. This represents a sort of best case improvement; how much the maximum throughput can be really improved depends on the way in which the communication units are implemented (i.e. if they are executed on dedicated cores, co-processors, or thread contexts of a HMT architecture), and on the physical communication channel exploited: on a distributed-memory system a single processor is usually enough to support the maximum network bandwidth, so multiple units only offer a slight improvement; nevertheless, *in multi-core architectures multiple KUs can be really effective to fully utilize the available shared memory bandwidth.*

The key idea of this mechanism is that multiple delegations can be issued by the Emitter with a small overhead. As we have seen in Section III-A, this is possible using a by-reference mode of passing messages to the communication units. It is worth noting that the same idea can be applied to collective primitives such as scatter and multicast. Multiple communication units are in charge of performing a part of the scatter/multicast to a sub-set of the Workers. In the best case, if the number of communication units is equal to the number of Workers, the completion time of a collective can become equal to the latency of a single point-to-point communication.

IV. A Message-Passing Support

To provide a rapid prototyping environment for the concepts and ideas introduced in Section III, we introduce the core part of our message-passing library and its runtime support for shared-memory architectures. While interesting on the long term, the extension of existing message-passing systems such as MPI requires a large effort in terms of design and development. We will address this aspect in our future work.

A. A lightweight Message-Passing library

We have implemented the minimal set of functionalities inspired by the CSP (Communicating Sequential Processes) semantics [9]. Explicit (named) *channels* are used by the functionalities of a parallel computation to exchange messages. A channel is a passive data structure (*channel_descriptor*) of the run-time support - i.e. not directly accessible by the programmer - that represents a sort of "device" to perform communications. Basic primitives are *send* and *receive* ones, operating on a specified channel (by passing its unique identifier and, in the send case, the pointer to the message buffer).

We focus on *asynchronous* point-to-point communications, i.e. send and receive operations on the same channel do not necessarily happen at the same time instant. During the channel creation, the programmer specifies the *asynchrony degree* $k \geq 0$, i.e. the maximum number of messages that can be stored at any given point in time inside the channel without blocking the sender. Synchronous communications represent a particular case, achieved setting $k = 0$ during the channel creation.

To easily guarantee the asynchrony degree, channels are univocally associated with a message *type*, defined at the channel creation. This makes it possible to statically allocate the channel data structures of our run-time support without additional overhead to manage internal buffers of dynamic size. Furthermore, the use of typed channels avoids type mismatch between send and receive pairs on the same channel.

As stated in Section III, a critical point for an efficient message-passing is to make the communication latency as low as possible. We face the problem by: *(i)* implementing zero-copy communications and, *(ii)* implementing the communication support entirely in user-space. About this point, in order to implement the sharing of run-time support data-structures (channel descriptors, delegation queues, receivers buffers) we have two possibilities:

- the functionalities of a parallel program are implemented by means of processes and POSIX `SYSV` shared memory segments to implement the (partial) sharing of memory spaces;
- all the functionalities are pthreads sharing the memory space for dynamically allocated data structures (heap).

We expect that the two solutions have different overhead (higher in the first case, to allocate and bind shared memory segments between processes). For the sake of simplicity in this paper we discuss the implementation of our message-passing library using pthreads. It is worth noting that, though we use pthreads, we require that the programs written using our library assume a local environment model, in which the unique shared data structures are the ones of our message-passing support, i.e. from a high-level viewpoint the programmer uses pthreads as processes with independent memory spaces.

Point *i)* is addressed by implementing a single copy from the address space of the sender to that one of the receiver. The channel consists of a static set of k receiving buffers used in a circular manner. The copy is performed by the send primitive, while the receive is in charge of checking the presence of a message only, returning a pointer to the correct buffer to the caller. The second point *ii)* is aimed at avoiding to perform system calls during communication (required to manage full and empty status of the circular buffer). The correctness of operations on channel descriptors and delegation queues is obtained by using user-space spin-locks mechanisms. However, as stated in the literature [5], busy-wait may have a negative effect when combined with hardware multi-threading. Solutions to partially solve this problem will be discussed in Section V, devoted to the experimental evaluation.

B. Delegation mechanism and communication threads

In this part, we briefly discuss the implementation of the delegation mechanism on our library targeting generic multicores. Since we adopt zero-copy communications, only send primitives deserve to be delegated to communication units, since receive operations are only in charge of checking the presence of a message.

In our library a communication unit is implemented as a support thread accelerating one or more functionalities of the parallel program. The programmer specifies the number of *communication threads* (hereinafter denoted by KT) during the program initialization by using a proper primitive of the library. According to the desired affinity, communication threads can be executed on dedicated cores as well as fixed on specific hardware contexts throughout the execution.

As said in Section III-A, the delegation mechanism must be as efficient as possible (faster than the communication also for short/medium message sizes) and able to support multiple delegation requests possibly in a FIFO fashion. Fig. 3 shows an abstract representation of our implementation.

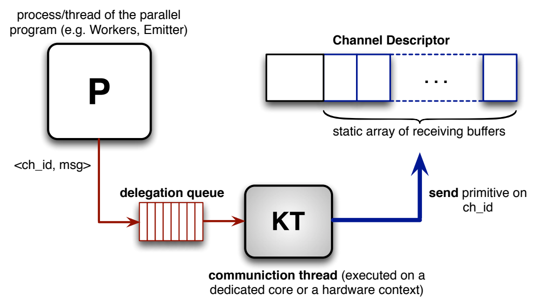

process/thread of the parallel program (e.g. Workers, Emitter)
P
<ch_id, msg>
delegation queue
KT
communiction thread (executed on a dedicated core or a hardware context)
Channel Descriptor
...
static array of receiving buffers
send primitive on ch_id

Fig. 3: Implementation of the delegation mechanism: delegation queue, channel descriptor and communication thread.

The interaction between Emitter, Worker and Collector functionalities and a communication thread is performed using a *queue* data structure of the run-time support (invisible to the programmer). Elements of the queue are *delegation descriptors* indicating the channel identifier and the message pointer. The queue is accessed following the producer-consumer paradigm protected by user-level synchronization.

From the programmer viewpoint, during the channel creation a proper set of communication threads are passed (by identifiers) to the channel creation function. This makes it possible to delegate send primitives on the channel to one or *more* communication threads, transparently selected by the run-time support according to load balancing policies. When delegated to a support thread, the send primitive implements a non-blocking semantics: it returns immediately the control to the caller with a request identifier. A *wait* primitive (similar to MPI) is provided to check the completion of the send in order to consistently re-use the message buffer.

V. Experiments

To evaluate our message-passing support we perform experiments using different parallelism patterns and comparing our library with state-of-the-art message-passing and shared-memory implementations on multicores.

Our target architecture is the many-core Intel Xeon Phi™ 5110P featuring 60 cores running at 1056 Mhz interconnected by a bi-directional ring bus. Each core is equipped with a 32 KB L1 data cache and a 512 KB L2 cache. An important component of each core is its 512 bit wide *vector processor unit* (VPU), which can be used for double and single precision SIMD instructions. The architecture is equipped with 8 GB of GDDR5 memory interfaced with the ring bus through 8 memory controllers. Instead of using the Intel Phi™ as a co-processor, we use it as a general-purpose manycore, i.e. as a stand-alone architecture on which executing our experiments. All the experiments have been compiled using the `icc` compiler ver. 13.0.1 with the `-O3` optimizations and the `-mmic` flag to produce the binary executables for the Intel Phi™ architecture.

The cores of the Intel Phi™ support a 4-way interleaved multi-threaded execution (one of the possible implementations of hardware multi-threading). Each of the four hardware threads is assigned to a buffer of two instruction bundles. Each core is capable of issuing two instructions per clock cycle so each bundle consists of two instructions. The instruction bundles of the four threads are executed in a round-robin fashion during consecutive time slots to mask internal latencies (logical dependencies) or external delays (cache line misses).

In this section, in order to evaluate the design ideas presented in Section III and IV, we study message-passing and shared-memory implementations of two parallel benchmarks.

A. Stream-parallel experiment

The first experiment is based on a *video filtering kernel*. The application works by converting a stream of 1000×1000 pixels uncompressed RGB images into the gray-scale format. This is a simple, yet widely used filter in the pre-processing phase of many complex video streaming applications [21] (e.g. feature extraction and edge detection).

The filter has been written to be easily vectorized by the Intel compiler, in order to exploit at best the VPUs of the cores. The application has been parallelized using a *task-farm* paradigm: we use several Workers, each one applying the filter to images received by a distribution entity called Emitter; output results are sent by Workers to a Collector. The Emitter distributes images using a round-robin scheduling. Given the roughly constant filtering time, this strategy balances the workload to the Workers. At the beginning of the computation the stream of images is pre-loaded in memory by the Emitter (images are read from a file passed to the program).

In a message-passing implementation images are transmitted by value. Each Worker receives the images from the Emitter, computes the filter, and sends the result to the Collector.

In a shared-memory implementation, all the program data structures can be used by any functionality of the parallel program (Emitter, Workers and Collector). In this model the Emitter can pass the images to Workers by reference. The Emitter transmits the unique identifier (e.g. the memory pointer) of the image to be processed; the filter is applied in-place and the image identifier is forwarded to the Collector at the end of the computation.

1) Using communication threads to overlap communications with calculation: we start by studying the effect of communication threads to overlap communications with calculation. In the case of the task-farm scheme, the only functionalities performing calculation are the Workers. Due to the zero-copy communications provided by our library, the transmission phase to the Collector is the only one deserving to be overlapped with the calculation of the next image.

In this experiment we compare the following implementations of the task-farm:

- **Sh-Mem**: a shared-memory implementation with POSIX pthreads in which the Emitter passes the pointers to the images to the Workers by using single-producer single-consumer shared queues protected by standard pthread spin-lock mechanisms;

- **MPI**: a message-passing implementation using the Intel MPI library ver. 4.1.0.024 and asynchronous non-blocking communications (i.e. `MPI_Isend` and `MPI_Irecv`);
- **No-KT**: the message-passing implementation using our library without communication threads;
- **KT-W**: similar to the previous point but using one communication thread for each Worker.

As stated in section IV, we use spin-locks in our support to protect channel descriptors and delegation queues. Especially for HMT architectures, the busy-waiting phase must not aggressively consume the processor resources when the synchronization involves two threads executed on the same core [5]. On Intel Phi™ we achieve this behavior by using the `delay` instruction inside the spin-loop, which avoids to fetch/issue instructions from the issuing thread for a certain number of clock cycles specified in a proper register. So doing we can slow down the busy-waiting loop and even implement advanced features such as exponential back-off.

Fig. 4 and Tab. I show the *service time* $T_S^{(n)}$ of the task-farm (inverse of the application throughput) using n Workers (referred to as *parallelism degree*) and the *scalability*:

$$S^{(n)} = \frac{T_S^{(1)}}{T_S^{(n)}}$$

Given the extensive HMT support of the Intel Phi™, it is important to use more than one thread context per core; nevertheless, the use of hardware multi-threading in parallel applications is widely known to be problematic and often with uncertain results. In particular, the affinity of threads on hardware contexts is extremely important. In the experiment of Fig. 4 two cores are dedicated to the Emitter and the Collector functionalities. Up to 58 cores, we map one Worker onto each core (plus a communication thread per core in the KT-W case). We use more Workers per core with the MPI, Sh-Mem and No-KT versions.

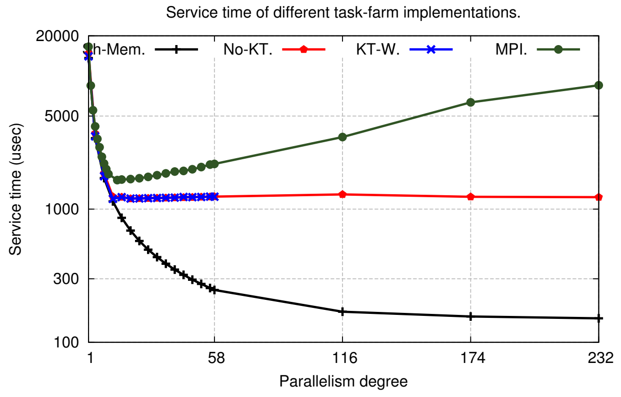


Fig. 4: Service time of the task-farm using different implementations.

From a first impact, we can note that with high parallelism degrees the shared-memory implementation is the best one. In particular, with a limited number of Workers all the message-passing implementations perform quite similarly, and their service time stops to decrease with more than 12 Workers. Intuitively, *the throughput of the message-passing versions is limited by the Emitter*, which performs point-to-point distributions to Workers. Nevertheless, a first notable result is that our library (KT-W and No-KT cases) offers better service times than MPI. The performance of KT-W version is slightly better than No-KT. In Fig. 4 this difference can not be appreciated (the two lines are very close). Tab. I gives a detailed description of the results, with the overlapping support achieving a constant improvement of $\sim 5\%$ w.r.t the No-KT version. Up to 12 Workers, when the Emitter does not limit the application throughput using message-passing, it is also interesting to note that the KT-W version obtains very similar results to the shared-memory one.

N	Sh-Mem		MPI	No-KT	KT-W		
	$T_S^{(n)}$	$S^{(n)}$	$T_S^{(n)}$	$T_S^{(n)}$	$T_S^{(n)}$	$S^{(n)}$	gain
1	13614	1	16558	14789	14022	1	5.18%
4	3407	3.99	4177	3703	3515	3.98	5.07%
8	1709	7.96	2200	1856	1764	7.94	4.97%
12	1141	11.93	1650	1241	1193	11.75	3.85%
16	860	15.83	1665	1229	1228	11.41	$\sim 0\%$
58	247	55.12	2183	1240	1239	11.31	$\sim 0\%$

TABLE I: Service time T_S and scalability S of the implementations and performance gain of using KTs; time values are reported in μs.

2) Using communication threads to parallelize the Emitter: previous results show that a sequential Emitter is able to sustain up to 12 Workers before becoming a bottleneck. To increase the overall throughput, the distribution phase can be parallelized using a proper set of communication threads denoted by KT$_E$. The number of communication threads n_{kt}^e can be empirically calculated as a function of the number N of Workers, i.e. $n_{kt}^e(N) = \lceil (N/12) \rceil$. The results of the experiments are shown in Fig. 5. Since communication threads can be used to parallelize the Emitter and to overlap Workers communications, we study two different allocation policies:

- *KT-specialized*: Emitter and Collector are fixed on two dedicated cores. A number of communication threads KT$_E$ (according to the previous function) are used for the Emitter and fixed on a number of dedicated cores. We execute an amount of communication threads for the Workers, KT$_W$. To avoid to under-utilize communication threads, we use one KT$_W$ for a group of Workers (the best configuration uses one KT$_W$ for each group of 10 Workers). It is important to observe that this solution is also useful to limit the amount of shared data structures, which is a valuable property if SYSV shared-memory segments are used to implement shared memory space between processes instead of using pthreads;
- *KT-generalized*: instead of separate KTs for the Emitter and the Workers, we use KTs in a general way. We fix at most one KT per core, which is in charge of executing send operations directed/generated to/from the Workers mapped onto that core, i.e. it executes delegated sends from the Emitter and from the Workers. It is worth noting that this version needs a unified address space for all the application, i.e. Emitter, Collector, Workers and KTs are threads of the same program.

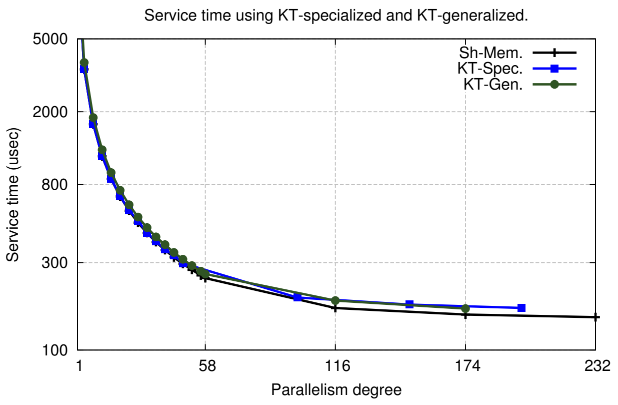


Fig. 5: Service time of the task-farm: different allocation policies and parallelization of the Emitter.

Fig. 5 shows that we obtain similar results compared to the shared-memory implementation. Numerical results are depicted in Tab. II highlighting that, when we use up to one Worker per core, Sh-Mem and the KT-specialized versions are aligned (note, however, that by using KTs we limit the number of Workers to 48). The difference increases when we use multiple thread contexts to execute more Workers per core, mainly due the presence of KTs that limit the maximum number of available contexts for Worker execution. When we use higher parallelism degrees, KT-generalized provides slightly better results compared with KT-specialized. The performance loss w.r.t Sh-Mem is bounded by 10%.

N	**Sh-Mem**		**KT Specialized**		**KT Generalized**	
	$T_S^{(n)}$	$S^{(n)}$	$T_S^{(n)}$	$S^{(n)}$	$T_S^{(n)}$	$S^{(n)}$
4	3407	3.99	3416	3.99	3711	4.01
16	860	15.8	862	15.8	930	16
48	295	46.1	298	45.7	313	47.5
58	247	55.1	-	-	260	57.2
2 per core	169.7	80.2	193.7	70.4	186.3	79.8
3 per core	156.2	87.2	177	77	**168.5**	**88.2**
4 per core	**151.3**	**90**	**169.7**	**80.3**	-	-

TABLE II: Service time T_S and scalability S of the implementations and performance gain by using KTs; time values are reported in μs.

B. Data-parallel experiment

In this section we present a second experiment based on a data-parallel *five-point stencil* computation. This application is common in finite difference approximations of differential equations [21]. Consider a two dimensional grid of points (represented as a matrix A of doubles), the value of every point at time step $t+1$ is updated by a function (usually the average) applied to the point itself and to its four neighbor points at time step t; this is repeated for a certain number of time steps. The data-parallel program consists in a set of Workers applying the computation on their partition of the matrix at each time step. We adopt a *partitioning by blocks of rows*. In this case, "vertical" communications have to be done (one row exchanged with the northern neighbor, and one with the southern neighbor). For the purpose of our analysis, we assume that each Worker starts the computation holding its partition, neglecting the scatter/gather phases on message-passing and the memory copies for the shared-memory version. This is a reasonable assumption if we consider a large number of time steps such that the initialization phase can be neglected with a negligible implication on the completion time.

We developed two shared-memory implementations. The first one, denoted by **Sh-Mem**, operates on two matrix copies, the first representing A^t (t denotes the current iteration) and the other, A^{t-1}, is overwritten with A^{t+1}. The synchronization at the end of each iteration is implemented using a pthread barrier between Workers. The second one, denoted by **OMP**, is a OpenMP version obtained by adding a `parallel for` annotation over the loop on the lines of the matrix. The OpenMP program has been compiled using the `icc` ver. 13.0.1 compiler (the only one available on Intel Phi™) with the `-openmp` flag and `granularity=fine,balanced` options.

We compare the shared-memory implementations with three message-passing versions. The first one, denoted by **No-KT**, does not use communication threads. To overlap communications with computation, the five-point stencil algorithm has been rewritten as depicted in Alg. 1. Communications of border lines (with relative indices 1 and g in Alg. 1) are overlapped with calculation of the inner region of the Worker partition (with relative indices from 2 to $g-1$). When communications with neighbors are completed, the Worker can compute the border lines. In Alg. 1 we denote by M^2 the size of the matrix and by $\mathcal{T}$ the number of time steps of the computation. We implemented a version named **KT-W**, in which for each Worker we add a communication thread, and a version denoted by **SKT** that allocates a limited number of KTs each one shared among a group of Workers.

Algorithm 1: Five-point stencil (Worker W_k).

```
for $t$=1 to $\mathcal{T}$ do
    send($A^t_{1,*}$, $W_{k-1}$);        //Communication of border elements.
    send($A^t_{g,*}$, $W_{k+1}$);
    for $i$=2 to $g-1$ do                //Internal calculation overlapped with previous sends.
        for $j$=1 to $M$ do
            $A^{t+1}_{i,j}$=$(A^t_{i,j}+A^t_{i-1,j}+A^t_{i+1,j}+A^t_{i,j-1}+A^t_{i,j+1})/5$;
        end
    end
    receive($A^t_{0,*}$, $W_{k-1}$);
    receive($A^t_{g+1,*}$, $W_{k+1}$);
    for $j$=1 to $M$ do                  //Calculation of border parts of the partition.
        $A^{t+1}_{1,j}$=$(A^t_{1,j}+A^t_{0,j}+A^t_{2,j}+A^t_{1,j-1}+A^t_{1,j+1})/5$;
        $A^{t+1}_{g,j}$=$(A^t_{g,j}+A^t_{g-1,j}+A^t_{g+1,j}+A^t_{g,j-1}+A^t_{g,j+1})/5$;
    end
end
```

1) Comparison between Shared-Memory and Message-Passing implementations: in our benchmark we test the implementations with two different matrix sizes (*i.e.* 1080×1080 and 8640×8640 pixels) and with a number of time steps fixed to 100 on the Intel Phi™. The performance metrics of interest are the *completion time* $T_C^{(n)}$ and the *scalability* $S^{(n)} = T_C^{(1)}/T_C^{(n)}$. Fig. 6 shows the results obtained with a matrix of 1080×1080 elements using up to 60 Workers for the sake of readability. Results with more Workers are summarized in Tab. III.

We can note an overall limited scalability and a slow-down when allocating multiple Workers per core, probably because we reach the available memory bandwidth. The OpenMP

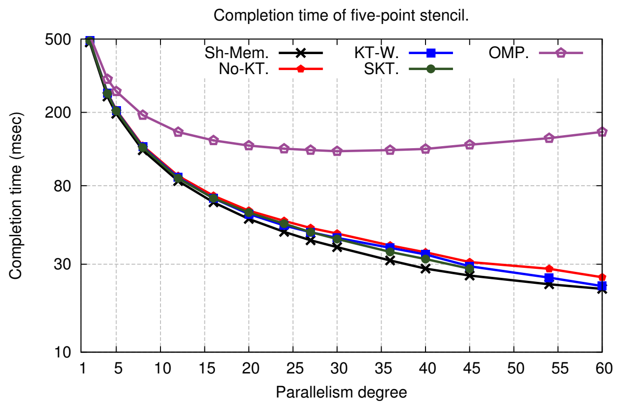


Fig. 6: Completion time with 1080×1080 matrices.

N	**Sh-Mem**		**OMP**	**No-KT**	**SKT**	**KT-W**	
	$T_C^{(n)}$	$S^{(n)}$	$T_C^{(n)}$	$T_C^{(n)}$	$T_C^{(n)}$	$T_C^{(n)}$	$S^{(n)}$
1	957	1	930	969	962	965	1
4	243	3.93	303	256	253	255	3.78
16	65.1	14.7	141	70.5	68.4	68.3	14.1
30	37.1	25.8	**123**	44.1	41.1	41.7	23.1
45	26.1	36.8	134	30.8	28.4	29.2	33
60	22.1	43.4	157	25.5	-	**22.8**	42.2
2 per core	**19.7**	48.3	197	**23.1**	**26.6**	402	2.4
3 per core	23.2	41.2	272	29.8	26.9	-	-
4 per core	24.5	39.1	351	43.7	2190	-	-

TABLE III: Numerical results of data-parallel implementations for 1080×1080 matrices. Times are reported in ms.

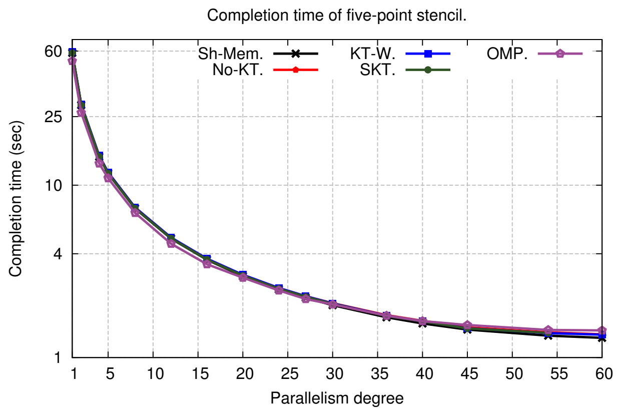


Fig. 7: Completion time with 8640×8640 matrices.

N	**Sh-Mem**		**OMP**	**No-KT**	**SKT**	**KT-W**	
	$T_C^{(n)}$	$S^{(n)}$	$T_C^{(n)}$	$T_C^{(n)}$	$T_C^{(n)}$	$T_C^{(n)}$	$S^{(n)}$
1	58.6	1	52.6	59.1	58.5	59.3	1
16	36.9	15.9	34.8	37.3	37.1	37.5	15.8
54	13.4	43.7	14.5	14.1	**13.9**	13.9	45.7
60	**13.1**	45.1	**14.4**	**13.6**	-	**13.6**	43.7
2 per core	19.5	30.1	21.7	19.1	18.2	20.1	29.4
3 per core	21.6	27.2	24.9	23.5	29.4	-	-
4 per core	22.4	26.1	26.6	27.4	31.2	-	-

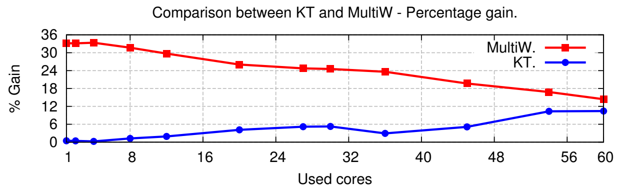
TABLE IV: Numerical results of data-parallel implementations for 8640×8640 matrices. Times are reported in s.

version is the slowest, and it is not suitable for extremely fine-grain computations, while Sh-Mem slightly outperforms all the other implementations. In this application the overlapping support works well: with a higher number of Workers, and thus smaller partitions, the gap between KT and Sh-Mem versions decreases, ending with very similar completion times. This is an expected result, because the ratio between calculation and communication decreases using higher parallelism degrees (each Worker exchanges the same number of elements with its neighbors). It is worth noting, however, that allocating two Workers and two KTs on the same core significantly lowers the performance, resulting in a very limited scalability. SKT version works better, but due to a smaller number of available contexts for Worker functionalities (some contexts are devoted to execute KTs) it is not able to reach the peak performance of the other implementations.

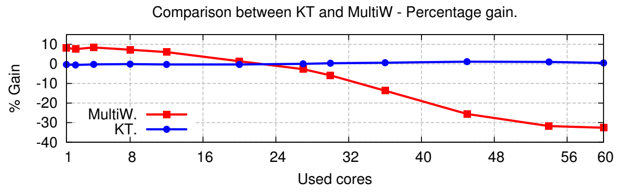
The scenario is quite different if we increase the computational grain of the problem. With 8640×8640 matrices (Fig. 7 and Tab. IV), the OMP version achieves the same performance of the other implementations. The effect of overlapping becomes negligible, i.e. No-KT, KT and SKT versions achieve very similar peak performance results.

2) Efficient exploitation of Hardware Multi-Threading: in this paper we discuss the use of hardware multi-threading to overlap communications with calculation in message-passing programs. However, even with this programming model, *the benefit of using multiple hardware contexts per core is difficult to estimate and the performance result is often unpredictable*. For instance, it is not clear if it is convenient to use multiple thread contexts to increase the parallelism degree of the program, or if they are useful to execute support functionalities such as communication threads. This interesting aspect, which still deserves further investigations, can be exemplified by the experiment shown in Fig. 8. We study the completion time of our data-parallel benchmark by using the hardware thread contexts in different ways. In the first case (denoted by **KT**) we map one Worker thread and one KT per core (we use two of the four available thread contexts); in the second case (denoted by **MultiW**) we map two Worker threads per core without any KT. The results are depicted in Fig. 8 by showing the *performance gain* w.r.t a baseline in which we do not use HMT, i.e. we use exactly one context per core by mapping one Worker thread. A positive gain represents a performance improvement (a reduction in the completion time), vice-versa a negative one is a performance loss.

The result confirms that the exploitation of HMT in parallel programs is still a sort of "dark magic": even by considering the same program with different data sizes, we obtain completely different results. In particular, we notice that for MultiW the gain is reduced when increasing the parallelism degree; here the starting point is significant: in the first case, by having an initial gain of $\sim 33\%$, we end with a good advantage even with 60 cores; in the second, starting with a low gain, we end with *a performance loss of* 30%. This suggests that, *using HMT for communication threads has the important advantage of not excessively affecting the Workers execution time on the other contexts*. In both benchmarks the application runs equally or faster; however, the performance gain is reduced, as it depends on the amount of communications that can be



(a) Matrix 1080 × 1080.



(b) Matrix 8640 × 8640.

Fig. 8: Percentage gain between KT and Multi-W compared with the baseline without HMT.

effectively overlapped.

VI. Conclusion

This paper dealt with the use of message-passing paradigm for parallel programming on multi-/many-core. Message-passing represents a valuable alternative on these architectures, not only because it is more portable in principle, but also because it is able to offer similar performance compared with shared-memory models provided that the run-time support is properly designed. We introduced a lightweight support able to outperform MPI on very fine grain computations. We provided a delegation mechanism which can be efficiently used to overlap communications with calculation and to parallelize distribution activities, effectively reducing the performance gap w.r.t shared-memory implementations at the cost of decreasing the maximum available parallelism degree.

In line with existing research works, we discussed the use of HMT to execute communication threads. Specific tests prove that this solution represents a "safer" choice in terms of performance than allocating multiple Workers on HMT contexts, ensuring the absence of performance degradation w.r.t single-context parallel programs and alleviating the impact of support threads on the maximum available parallelism degree.

References

[1] "Tilera tile64pro." [Online]. Available: http://www.tilera.com/products/processors/TILEPRO64

[2] B. Armstrong, S. Kim, and R. Eigenmann, "Quantifying differences between openmp and mpi using a large-scale application suite," in *High Performance Computing*, ser. Lecture Notes in Computer Science, M. Valero, K. Joe, M. Kitsuregawa, and H. Tanaka, Eds. Springer Berlin Heidelberg, 2000, vol. 1940, pp. 482–493.

[3] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen, "Speculative precomputation: long-range prefetching of delinquent loads," in *Proceedings of the 28th annual international symposium on Computer architecture*, ser. ISCA '01. New York, NY, USA: ACM, 2001, pp. 14–25.

[4] M. Curtis-Maury and T. Wang, "Integrating multiple forms of multithreaded execution on multi-smt systems: A study with scientific applications," in *Proceedings of the Second International Conference on the Quantitative Evaluation of Systems*, ser. QEST '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 199–.

[5] T. De Matteis, F. Luporini, G. Mencagli, and M. Vanneschi, "Evaluation of architectural supports for fine-grained synchronization mechanisms," in *Proceedings of the 11th IASTED International Conference on Parallel and Distributed Computing and Networks*, Innsbruck, Austria, 2013.

[6] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li, "Vmmc-2: Efficient support for reliable, connection-oriented communication," in *IN PROCEEDINGS OF HOT INTERCONNECTS*, 1997.

[7] P. Geoffray, L. Prylli, and B. Tourancheau, "Bip-smp : High performance message passing over a cluster of commodity smps," in *Supercomputing, ACM/IEEE 1999 Conference*, 1999, pp. 20–20.

[8] G. Goumas, N. Anastopoulos, N. Koziris, and N. Ioannou, "Overlapping computation and communication in smt clusters with commodity interconnects," in *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, 2009, pp. 1–10.

[9] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, Aug. 1978.

[10] T. Hoefler, C. Siebert, and W. Rehm, "A practically constant-time mpi broadcast algorithm for large-scale infiniband clusters with multicast," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, 2007, pp. 1–8.

[11] F. Jiao, N. Mahajan, J. Willcock, A. Chauhan, and A. Lumsdaine, "Partial globalization of partitioned address spaces for zero-copy communication with shared memory." in *HiPC*. IEEE, 2011, pp. 1–10.

[12] G. Krawezik, "Performance comparison of mpi and three openmp programming styles on shared memory multiprocessors," in *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, ser. SPAA '03. New York, NY, USA: ACM, 2003.

[13] P. Lai, P. Balaji, R. Thakur, and D. Panda, "Proone: a general-purpose protocol onload engine for multi- and many-core architectures," *Computer Science - Research and Development*, vol. 23, no. 3-4, 2009.

[14] T. Ma, G. Bosilca, A. Bouteiller, and J. J. Dongarra, "Kernel-assisted and topology-aware {MPI} collective communications on multicore/many-core platforms," *Journal of Parallel and Distributed Computing*, vol. 73, no. 7, pp. 1000 – 1010, 2013.

[15] F. O'Carroll, H. Tezuka, A. Hori, and Y. Ishikawa, "The design and implementation of zero copy mpi using commodity hardware with a high performance network," in *Proceedings of the 12th international conference on Supercomputing*, ser. ICS '98. New York, NY, USA: ACM, 1998, pp. 243–250.

[16] S. Saini, H. Jin, R. Hood, D. Barker, P. Mehrotra, and R. Biswas, "The impact of hyper-threading on processor resource utilization in production applications," in *Proceedings of the 2011 18th International Conference on High Performance Computing*, ser. HIPC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 1–10.

[17] S. Varoglu and S. Jenks, "Architectural support for thread communications in multi-core processors," *Parall. Comput.*, vol. 37, no. 1, 2011.

[18] T. von Eicken, A. Basu, V. Buch, and W. Vogels, "U-net: a user-level network interface for parallel and distributed computing," *SIGOPS Oper. Syst. Rev.*, vol. 29, no. 5, pp. 40–53, Dec. 1995.

[19] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: a mechanism for integrated communication and computation," *SIGARCH Comput. Archit. News*, vol. 20, no. 2, pp. 256–266, Apr. 1992.

[20] S. Wallace, B. Calder, and D. M. Tullsen, "Threaded multiple path execution," *SIGARCH Comput. Archit. News*, vol. 26, no. 3, Apr. 1998.

[21] B. Wilkinson and M. Allen, *Parallel programming: techniques and applications using networked workstations and parallel computers*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1999.

[22] X. Wu and V. Taylor, "Performance characteristics of hybrid mpi/openmp implementations of nas parallel benchmarks sp and bt on large-scale multicore supercomputers," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, no. 4, pp. 56–62, Mar. 2011.