**INTRODUCTION**

*Shared-memory* and *message-passing* are the two fundamental concurrency models giving the basis to the modern programming models for parallel computing.
**INTRODUCTION**

**Shared-memory** and **message-passing** are the two fundamental concurrency models giving the basis to the modern programming models for parallel computing.

**Long-standing questions:**
- which model to use?
- which approach is more efficient?
Shared-memory and message-passing are the two fundamental concurrency models giving the basis to the modern programming models for parallel computing.

**Long-standing questions:**
- which model to use?
- which approach is more efficient?

**Results:**
- existing frameworks adopt one of the concurrency models;
- programmers choose the most appropriate one for their own applications.
Message passing is characterized by inherent properties of portability, flexibility and ease of debugging.
INTRODUCTION

Message passing is characterized by inherent properties of portability, flexibility and ease of debugging.

Good performance message-passing model in shared memory architectures requires an efficient implementation of the run-time support in order to:

- masking communication latencies;
- provide an efficient way to accelerate distribution and collective functionalities.
Our work investigates the design issues of message-passing on multi-/many-core:
Our work, investigates the design issues of message-passing on multy-/many-core:

- describing a general run-time support mechanism based on communication threads coupled with the functionalities of parallel programs;
Our work, investigates the design issues of message-passing on multy-/many-core:

- describing a general run-time support mechanism based on *communication threads* coupled with the functionalities of parallel programs;
- discussing the execution of communication threads on hardware contexts of multi-threaded architectures.
DESIGNING AN EFFICIENT MESSAGE-PASSING ON MULTICORE
EFFICIENT MESSAGE-PASSING

Two important issues to be addressed:
Two important issues to be addressed:

- efficient implementation of basic communication primitives (i.e. send and receive);
EFFICIENT MESSAGE-PASSING

Two important issues to be addressed:

- efficient implementation of basic communication primitives (i.e. send and receive);
- optimization of collective and distribution primitives (scatter, multicast, or distributions based on point-to-point communications);
We can move in two directions:
BASIC COMMUNICATION PRIMITIVES

We can move in two directions:

- lowering the communication latency by minimizing the copies required by a pair of send/receive operations;
We can move in two directions:

- lowering the communication latency by minimizing the copies required by a pair of send/receive operations;

A well known solution relies in implementing zero-copy communications.
BASIC COMMUNICATION PRIMITIVES

We can move in two directions:

- lowering the communication latency by minimizing the copies required by a pair of send/receive operations;
- mask communications by overlapping them with calculation.
We can move in two directions:

- lowering the communication latency by minimizing the copies required by a pair of send/receive operations;
- mask communications by overlapping them with calculation
We can move in two directions:

- lowering the communication latency by minimizing the copies required by a pair of send/receive operations;
- mask communications by overlapping them with calculation
We can move in two directions:

- lowering the communication latency by minimizing the copies required by a pair of send/receive operations;
- mask communications by overlapping them with calculation
We can move in two directions:

- lowering the communication latency by minimizing the copies required by a pair of send/receive operations;
- mask communications by overlapping them with calculation
We can move in two directions:

- lowering the communication latency by minimizing the copies required by a pair of send/receive operations;
- mask communications by overlapping them with calculation
We can move in two directions:

- lowering the communication latency by minimizing the copies required by a pair of send/receive operations;
- mask communications by overlapping them with calculation
We can move in two directions:

- lowering the communication latency by minimizing the copies required by a pair of send/receive operations;
- mask communications by overlapping them with calculation

It requires proper software and architectural supports for delegating communication primitives to proper units.
We need:

1. an efficient run-time support mechanism to delegate the execution of communication primitives;
2. a communication unit able to execute communication primitives in parallel with calculation phases. In shared-memory architectures can be concretized using:
   - a dedicated core of the processor;
   - a hardware context in the case of HMT CPUs;
   - a specialized co-processor if available (e.g. in Tilera CPUs).
We need:

1. an efficient run-time support mechanism to delegate the execution of communication primitives;
We need:

1. an efficient run-time support mechanism to delegate the execution of communication primitives;
2. a communication unit able to execute communication primitives in parallel with calculation phases. In shared-memory architecture can be concretized using:
COMMUNICATION OVERLAPPING

We need:

1. an efficient run-time support mechanism to delegate the execution of communication primitives;
2. a communication unit able to execute communication primitives in parallel with calculation phases. In shared-memory architecture can be concretized using:
   a dedicated core of the processor;
We need:

1. an efficient run-time support mechanism to delegate the execution of communication primitives;
2. a communication unit able to execute communication primitives in parallel with calculation phases. In shared-memory architecture can be concretized using:
   - a dedicated core of the processor;
   - a hardware context in the case of HMT CPUs;
We need:

1. an efficient run-time support mechanism to delegate the execution of communication primitives;
2. a communication unit able to execute communication primitives in parallel with calculation phases. In shared-memory architecture can be concretized using:
   - a dedicated core of the processor;
   - a hardware context in the case of HMT CPUs;
   - a specialized co-processor if available (e.g. in Tilera CPUs).
We need:

1. an efficient run-time support mechanism to delegate the execution of communication primitives;
2. a communication unit able to execute communication primitives in parallel with calculation phases. In shared-memory architecture can be concretized using:
   - a dedicated *core* of the processor;
   - *a hardware context in the case of HMT CPUs*;
   - a specialized *co-processor* if available (e.g. in Tilera CPUs).
Collectives and distributions are critical for the performance and scalability of message-passing parallel programs.
Collectives and distributions are critical for the performance and scalability of message-passing parallel programs.

In multi-/many-core architectures deserve special attention. For example: *point-to-point distribution of a task farm*
Collectives and distributions are critical for the performance and scalability of message-passing parallel programs.

In multi-/many-core architectures deserve special attention. For example: *point-to-point distribution of a task farm*

The maximum throughput is

\[ B_{\text{max}} = \frac{1}{L_{\text{com}}(\sigma)} \]

where \( L_{\text{com}} \) is the communication latency expressed as a function of the message size \( \sigma \).
Collectives and distributions are critical for the performance and scalability of message-passing parallel programs.

In multi-/many-core architectures deserve special attention. For example: *point-to-point distribution of a task farm*

The maximum throughput is

\[ B_{max} = \frac{1}{L_{com}(\sigma)} \]

where \( L_{com} \) is the communication latency expressed as a function of the message size \( \sigma \).

Emitter should be parallelized
Collectives and distributions are critical for the performance and scalability of message-passing parallel programs.

In multi-/many-core architectures deserve special attention. For example: *point-to-point distribution of a task farm*

Emitter delegate communications to $m \geq 1$ comm. units (KUs).
Collectives and distributions are critical for the performance and scalability of message-passing parallel programs.

In multi-/many-core architectures deserve special attention. For example: *point-to-point distribution of a task farm*

Emitter delegate communications to $m \geq 1$ comm. units (KUs).

The maximum throughput is

$$B_{max} = \frac{m}{L_{com}(\sigma)}$$

i.e. $m$ times higher.

(theoretically; practically it depends on how KUs are implemented)
IMPLEMENTATION OF THE MESSAGE-PASSING SUPPORT
IMPLEMENTATION

We have implemented the minimal set of functionalities inspired by the CSP semantics:

- **typed channels** are used by the functionalities of a parallel computation to exchange messages;
IMPLEMENTATION

We have implemented the minimal set of functionalities inspired by the CSP semantics:

- **typed 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
We have implemented the minimal set of functionalities inspired by the CSP semantics:

- **typed 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;
- the basic communication primitives are **send** and **receive** operating on a specified channel.
We have implemented the minimal set of functionalities inspired by the CSP semantics:

- **typed 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
- the basic communication primitives are **send** and **receive** operating on a specified channel
- **asynchronous** point-to-point communications: send and receive operations on the same channel do not necessarily happen at the same time instant
IMPLEMENTATION

Features:
Features:

- zero-copy communications:
Features:

- zero-copy communications:
  - all the functionalities of the parallel program are implemented by threads ⇒ the run-time support shared structures are allocated in the heap;
IMPLEMENTATION

Features:

○ zero-copy communications:
  ○ all the functionalities of the parallel program are implemented by threads ⇒ the run-time support shared structures are allocated in the heap;
  ○ alternatively, use POSIX SYSV Shared Memory Segment;
Features:

- zero-copy communications:
  - all the functionalities of the parallel program are implemented by threads ⟷ the run-time support shared structures are allocated in the heap;
  - alternatively, use *POSIX SYSV Shared Memory Segment*;
  - single copy from the sender to the receiver. The channel consists of a static set of $k$ receiving buffers. The copy is performed by the send primitive, while the receive is only in charge of checking the presence of a message, returning a pointer to the buffer.
Features:

- zero-copy communications:
  - all the functionalities of the parallel program are implemented by threads ⇒ the run-time support shared structures are allocated in the heap;
  - alternatively, use *POSIX SYSV Shared Memory Segment*;
  - single copy from the sender to the receiver. The channel consists of a static set of $k$ receiving buffers. The copy is performed by the send primitive, while the receive is only in charge of checking the presence of a message, returning a pointer to the buffer.

- communication support entirely in user-space, by means of busy-waiting techniques (properly optimized) for guaranteeing the correctness.
DELEGATION MECHANISM

Thanks to zero-copy communication, only send primitives deserve to be delegated to KUs.
DELEGATION MECHANISM

Thanks to zero-copy communication, only send primitives deserve to be delegated to KUs.

KU implemented as a support thread (communication thread or KT), that can be executed on dedicated cores or fixed on specific hardware thread contexts.
Thanks to zero-copy communication, only send primitives deserve to be delegated to KUs.

KU implemented as a support thread (*communication thread* or *KT*), that can be executed on dedicated cores or fixed on specific hardware thread contexts.
Thanks to zero-copy communication, only send primitives deserve to be delegated to KUs.

KU implemented as a support thread (*communication thread* or *KT*), that can be executed on dedicated cores or fixed on specific hardware thread contexts.
Thanks to zero-copy communication, only send primitives deserve to be delegated to KUs.

KU implemented as a support thread (communication thread or KT), that can be executed on dedicated cores or fixed on specific hardware thread contexts.
EXPERIMENTS
Our target architecture is the many core Intel Xeon Phi™ 5110P:
Our target architecture is the many core Intel Xeon Phi™ 5110P:

- 60 cores@1056MHz in a bi-directional ring, each one equipped with:
  - 32KB L1/512KB L2 cache
  - 512 bit wide VPU
  - 4-way interleaved multi-threaded execution

Synchronization mechanisms tailored to this architecture

We will use it as a stand-alone architecture on which executing our experiments.
TARGET ARCHITECTURE

Our target architecture is the many core Intel Xeon Phi™ 5110P:

- 60 cores@1056MHz in a bi-directional ring, each one equipped with:
  - 32KB L1/512KB L2 cache
  - 512 bit wide VPU
  - 4-way interleaved multi-threaded execution
- 8 GB of GDDR5 memory interfaced through 8 memory controllers
TARGET ARCHITECTURE

Our target architecture is the many core Intel Xeon Phi™ 5110P:

- 60 cores@1056MHz in a bi-directional ring, each one equipped with:
  - 32KB L1/512KB L2 cache
  - 512 bit wide VPU
  - 4-way interleaved multi-threaded execution

- 8 GB of GDDR5 memory interfaced through 8 memory controllers

Synchronization mechanisms tailored to this architecture
Our target architecture is the many core Intel Xeon Phi™ 5110P:

- 60 cores@1056MHz in a bi-directional ring, each one equipped with:
  - 32KB L1/512KB L2 cache
  - 512 bit wide VPU
  - 4-way interleaved multi-threaded execution
- 8 GB of GDDR5 memory interfaced through 8 memory controllers

Synchronization mechanisms tailored to this architecture

We will use it as a stand-alone architecture on which executing our experiments
STREAM-PARALLEL EXPERIMENT

A video filtering kernel: convert a stream of $1000 \times 1000$ pixels images into the gray-scale format.
A video filtering kernel: convert a stream of $1000 \times 1000$ pixels images into the gray-scale format.

Parallelized using a task-farm paradigm:

- each Worker applies 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 scheduler;
STREAM-PARALLEL EXPERIMENT

A video filtering kernel: convert a stream of $1000 \times 1000$ pixels images into the gray-scale format.

Parallelized using a task-farm paradigm:

- each Worker applies 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 scheduler;

In a message-passing implementation images are transmitted by value
A video filtering kernel: convert a stream of $1000 \times 1000$ pixels images into the gray-scale format.

Parallelized using a task-farm paradigm:

- each Worker applies 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 scheduler;

In a message-passing implementation images are transmitted by value.

In a shared-memory implementation, the Emitter can pass the images to Workers by reference.
### STREAM-PARALLEL EXPERIMENT

Overlap communications Workers → Collector

<table>
<thead>
<tr>
<th>Parallelism degree</th>
<th>Service time (usec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>58</td>
</tr>
<tr>
<td>5</td>
<td>116</td>
</tr>
<tr>
<td>10</td>
<td>174</td>
</tr>
<tr>
<td>50</td>
<td>232</td>
</tr>
</tbody>
</table>

- **SH-Mem**: shared memory implementation (Best one)
- **MPI**: message passing using Intel MPI library (Worst one)
- **No-KT**: m.p. using our library w/o KTs
- **KT-W**: m.p. using our library with KTs

Better than MPI but still limited by the Emitter.
STREAM-PARALLEL EXPERIMENT

Overlap communications Workers → Collector

Service time of different task-farm implementations.

- **SH-Mem**: shared memory implementation
- **MPI**: message passing using Intel MPI library
- **No-KT**: m.p. using our library w/o KTs;
- **KT-W**: m.p. using our library with KTs.
STREAM-PARALLEL EXPERIMENT

Overlap communications Workers → Collector

- **SH-Mem**: shared memory implementation ⇒ **Best one**
- **MPI**: message passing using Intel MPI library
- **No-KT**: m.p. using our library w/o KT;
- **KT-W**: m.p. using our library with KT.

Service time of different task-farm implementations.

![Graph showing service time vs parallelism degree for Sh-Mem, No-KT, KT-W, and MPI implementations.](image-url)
STREAM-PARALLEL EXPERIMENT

Overlap communications Workers → Collector

Service time of different task-farm implementations.

- **SH-Mem**: shared memory implementation ⇒ **Best one**
- **MPI**: message passing using Intel MPI library ⇒ **Worst one**
- **No-KT**: m.p. using our library w/o KT's;
- **KT-W**: m.p. using our library with KT's.
STREAM-PARALLEL EXPERIMENT

Overlap communications Workers → Collector

Service time of different task-farm implementations.

- **SH-Mem**: shared memory implementation ⇒ **Best one**
- **MPI**: message passing using Intel MPI library ⇒ **Worst one**
- **No-KT**: m.p. using our library w/o KTs; ⇒ **Better than MPI but still limited by the Emitter**
- **KT-W**: m.p. using our library with KTs.
STREAM-PARALLEL EXPERIMENT

Overlap communications Workers → Collector

<table>
<thead>
<tr>
<th>$N$</th>
<th>Sh-Mem $T_S^{(n)}$</th>
<th>MPI $T_S^{(n)}$</th>
<th>No-KT $T_S^{(n)}$</th>
<th>KT-W $T_S^{(n)}$</th>
<th>gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13614</td>
<td>16558</td>
<td>14789</td>
<td>14022</td>
<td>5.18%</td>
</tr>
<tr>
<td>4</td>
<td>3407</td>
<td>4177</td>
<td>3703</td>
<td>3515</td>
<td>5.07%</td>
</tr>
<tr>
<td>8</td>
<td>1709</td>
<td>2200</td>
<td>1856</td>
<td>1764</td>
<td>4.97%</td>
</tr>
<tr>
<td>12</td>
<td>1141</td>
<td>1650</td>
<td>1241</td>
<td>1193</td>
<td>3.85%</td>
</tr>
<tr>
<td>16</td>
<td>860</td>
<td>1665</td>
<td>1229</td>
<td>1228</td>
<td>$\sim 0%$</td>
</tr>
<tr>
<td>58</td>
<td>247</td>
<td>2183</td>
<td>1240</td>
<td>1239</td>
<td>$\sim 0%$</td>
</tr>
</tbody>
</table>

- **SH-Mem**: shared memory implementation ⇒ **Best one**
- **MPI**: message passing using Intel MPI library ⇒ **Worst one**
- **No-KT**: m.p. using our library w/o KTs; ⇒ **Better than MPI but still limited by the Emitter**
- **KT-W**: m.p. using our library with KTs.
STREAM-PARALLEL EXPERIMENT

Overlap communications Workers $\rightarrow$ Collector

<table>
<thead>
<tr>
<th>$N$</th>
<th>Sh-Mem $T_S^{(n)}$</th>
<th>MPI $T_S^{(n)}$</th>
<th>No-KT $T_S^{(n)}$</th>
<th>KT-W $T_S^{(n)}$</th>
<th>gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13614</td>
<td>16558</td>
<td>14789</td>
<td>14022</td>
<td>5.18%</td>
</tr>
<tr>
<td>4</td>
<td>3407</td>
<td>4177</td>
<td>3703</td>
<td>3515</td>
<td>5.07%</td>
</tr>
<tr>
<td>8</td>
<td>1709</td>
<td>2200</td>
<td>1856</td>
<td>1764</td>
<td>4.97%</td>
</tr>
<tr>
<td>12</td>
<td>1141</td>
<td>1650</td>
<td>1241</td>
<td>1193</td>
<td>3.85%</td>
</tr>
<tr>
<td>16</td>
<td>860</td>
<td>1665</td>
<td>1229</td>
<td>1228</td>
<td>$\sim$ 0%</td>
</tr>
<tr>
<td>58</td>
<td>247</td>
<td>2183</td>
<td>1240</td>
<td>1239</td>
<td>$\sim$ 0%</td>
</tr>
</tbody>
</table>

- **SH-Mem**: shared memory implementation $\Rightarrow$ Best one
- **MPI**: message passing using Intel MPI library $\Rightarrow$ Worst one
- **No-KT**: m.p. using our library w/o KT; $\Rightarrow$ Better than MPI but still limited by the Emitter
- **KT-W**: m.p. using our library with KT.
STREAM-PARALLEL EXPERIMENT

Parallelization of the distribution phase using a set of KTs.
STREAM-PARALLEL EXPERIMENT

Parallelization of the distribution phase using a set of KTs.

![Graph](attachment:image.png)

Two different allocation policies:
- **KT-Specialized**: $K_T^E$ used for the Emitter, $K_T^W$ for the Workers
- **KT-Generalized**: one KT per core, which is in charge of executing send operations directed/generated to/from the Workers mapped onto that core
Parallelization of the distribution phase using a set of KTs.

<table>
<thead>
<tr>
<th>N</th>
<th>Sh-Mem</th>
<th>KT Specialized</th>
<th>KT Generalized</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T_S^{(n)}$</td>
<td>$S^{(n)}$</td>
<td>$T_S^{(n)}$</td>
</tr>
<tr>
<td>3 per core</td>
<td>156.2</td>
<td>87.2</td>
<td>177</td>
</tr>
<tr>
<td>4 per core</td>
<td>151.3</td>
<td>90</td>
<td>169.7</td>
</tr>
</tbody>
</table>
Parallelization of the distribution phase using a set of KTs.

<table>
<thead>
<tr>
<th>$N$</th>
<th>Sh-Mem</th>
<th>KT Specialized</th>
<th>KT Generalized</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T_S^{(n)}$</td>
<td>$S^{(n)}$</td>
<td>$T_S^{(n)}$</td>
</tr>
<tr>
<td>3 per core</td>
<td>156.2</td>
<td>87.2</td>
<td>177</td>
</tr>
<tr>
<td>4 per core</td>
<td>151.3</td>
<td>90</td>
<td>169.7</td>
</tr>
</tbody>
</table>
Data-parallel five-point stencil computation
Data-parallel *five-point stencil* computation

Consider a two dimensional grid of points (represented as a matrix), the value of every point at time step $t + 1$ is updated by a function applied to the point itself and to its four neighbor points at time step $t$; repeated for a certain number of time steps.
Data-parallel *five-point stencil* computation

Consider a two dimensional grid of points (represented as a matrix), the value of every point at time step $t + 1$ is updated by a function applied to the point itself and to its four neighbor points at time step $t$; 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 choose a partitioning by *blocks of rows*, assuming that each Worker starts the computation holding its partition.
Two shared-memory implementations:
Two shared-memory implementations:

- **Sh-Mem**: operates on two matrix copies $A^t$ (current step) and $A^{t-1}$ (previous step that will be overwritten by new results). Synchronization at the end of each iteration by means of a `pthread barrier`. 
Two shared-memory implementations:

- **Sh-Mem**: operates on two matrix copies $A^t$ (current step) and $A^{t-1}$ (previous step that will be overwritten by new results). Synchronization at the end of each iteration by means of a *pthread barrier*;

- **OMP**: the OpenMP version obtained by adding the `parallel for` annotation over the loop on the lines of the matrix
DATA-PARALLEL EXPERIMENT

Three message-passing implementations:

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

for $t=1$ to $T$
do
  send $(A^t_1, \cdots, W_k)$;
  send $(A^t_g, \cdots, W_k+1)$;
  for $i=2$ to $g-1$
do
    for $j=1$ to $M$
do
      $A^{t+1}_{i,j} = \frac{(A^t_{i,j} + A^t_{i-1,j} + A^t_{i,j-1} + A^t_{i+1,j} + A^t_{i,j+1})}{5}$;
    end
  end
  receive $(A^t_0, \cdots, W_k)$;
  receive $(A^t_{g+1}, \cdots, W_k+1)$;
  for $j=1$ to $M$
do
    $A^{t+1}_{1,j} = \frac{(A^t_{1,j} + A^t_{0,j} + A^t_{2,j} + A^t_{1,j-1} + A^t_{1,j+1})}{5}$;
  end
  $A^{t+1}_{g,j} = \frac{(A^t_{g,j} + A^t_{g-1,j} + A^t_{g+1,j} + A^t_{g,j-1} + A^t_{g,j+1})}{5}$;
end

//Communication of border elements.
//Internal calculation overlapped with previous sends.
//Calculation of border parts of the partition.

$\bullet$ KT -$W_k$: one KT per Worker;
$\bullet$ SKT: KT-s are shared among group of Workers.
DATA-PARALLEL EXPERIMENT

Three message-passing implementations:

- **No-KT**: without communication threads;

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

\[
\begin{align*}
&\text{for } t=1 \text{ to } T \text{ do} \\
&\quad \text{send } (A_{t1}, \cdots, W_{k1}); \\
&\quad \text{send } (A_{tg}, \cdots, W_{k+1}); \\
&\quad \text{for } i=2 \text{ to } g-1 \text{ do} \\
&\quad\quad \text{for } j=1 \text{ to } M \text{ do} \\
&\quad\quad\quad A_{t+1,i,j} = \left( A_{t,i,j} + A_{t,i-1,j} + A_{t,i+1,j} + A_{t,i,j-1} + A_{t,i,j+1} \right) / 5; \\
&\quad \quad \text{end} \\
&\quad \text{end} \\
&\quad \text{receive } (A_{t0}, \cdots, W_{k1}); \\
&\quad \text{receive } (A_{tg+1}, \cdots, W_{k+1}); \\
&\quad \text{for } j=1 \text{ to } M \text{ do} \\
&\quad\quad A_{t+1,1,j} = \left( A_{t,1,j} + A_{t,0,j} + A_{t,2,j} + A_{t,1,j-1} + A_{t,1,j+1} \right) / 5; \\
&\quad\quad A_{t+1,g,j} = \left( A_{t,g,j} + A_{t,g+1,j} + A_{t,g-1,j} + A_{t,g,j-1} + A_{t,g,j+1} \right) / 5; \\
&\quad \quad \text{end} \\
&\quad \text{end}
\end{align*}
\]

//Communication of border elements.
//Internal calculation overlapped with previous sends.
//Calculation of border parts of the partition.

- **KT-W**: one KT per Worker;
- **SKT**: KTs are shared among group of Workers.
Three message-passing implementations:

- **No-KT**: without communication threads;
- with communication threads: code rewritten for overlapping

---

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

```
for $t = 1$ to $T$ do
    send($A^t_{1,*}$, $W_{k-1}$);
    send($A^t_{g,*}$, $W_{k+1}$);
    for $i = 2$ to $g - 1$ do
        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
        $A^{t+1}_{1,j} = (A^t_{1,j} + A^t_{0,j} + A^t_{2,j} + A^t_{1,j-1} + A^t_{1,j+1}) / 5$;
        $A^{t+1}_{g,j} = (A^t_{g,j} + A^t_{g-1,j} + A^t_{g+1,j} + A^t_{g,j-1} + A^t_{g,j+1}) / 5$;
    end
end
```
Three message-passing implementations:

- **No-KT**: without communication threads;
- **with communication threads**: code rewritten for overlapping

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

```plaintext
for $t=1$ to $T$ do
    send($A_{t,1}$, $\cdot$, $W_{k-1}$);
    send($A_{t,g}$, $\cdot$, $W_{k+1}$);
    for $i=2$ to $g-1$ do
        for $j=1$ to $M$ do
            $A_{i,j}^{t+1} = (A_{i,j}^t + A_{i,j-1}^t + A_{i+1,j}^t + A_{i,j-1}^t + A_{i,j+1}^t) / 5$;
        end
    end
    receive($A_{0,1}$, $\cdot$, $W_{k-1}$);
    receive($A_{g+1,1}$, $\cdot$, $W_{k+1}$);
    for $j=1$ to $M$ do
        $A_{1,j}^{t+1} = (A_{1,j}^t + A_{0,j}^t + A_{1,j-1}^t + A_{1,j-1}^t + A_{1,j+1}^t) / 5$;
        $A_{g,j}^{t+1} = (A_{g,j}^t + A_{g-1,j}^t + A_{g+1,j}^t + A_{g,j-1}^t + A_{g,j+1}^t) / 5$;
    end
```

- **KT-W**: one KT per Worker;
- **SKT**: KT's are shared among group of Workers.
DATA-PARALLEL EXPERIMENT

Matrices of $1080 \times 1080$ elements, 100 iterations

Completion time of five-point stencil.

Parallelism degree

Completion time (msec)

Sh-Mem.
No-KT.
KT-W.
SKT.
OMP.

$2$ per core
DATA-PARALLEL EXPERIMENT

Matrices of $1080 \times 1080$ elements, 100 iterations

Completion time of five-point stencil.

<table>
<thead>
<tr>
<th>$N$</th>
<th>Sh-Mem</th>
<th>OMP</th>
<th>No-KT</th>
<th>SKT</th>
<th>KT-W</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T_C^{(n)}$</td>
<td>$S^{(n)}$</td>
<td>$T_C^{(n)}$</td>
<td>$T_C^{(n)}$</td>
<td>$T_C^{(n)}$</td>
</tr>
<tr>
<td>30</td>
<td>37.1</td>
<td>25.8</td>
<td><strong>123</strong></td>
<td>44.1</td>
<td>41.1</td>
</tr>
<tr>
<td>60</td>
<td>22.1</td>
<td>43.4</td>
<td>157</td>
<td>25.5</td>
<td>-</td>
</tr>
<tr>
<td>2 per core</td>
<td><strong>19.7</strong></td>
<td>48.3</td>
<td>197</td>
<td><strong>23.1</strong></td>
<td>26.6</td>
</tr>
</tbody>
</table>
Matrices of $1080 \times 1080$ elements, 100 iterations

![Completion time of five-point stencil graph]

<table>
<thead>
<tr>
<th>$N$</th>
<th>Sh-Mem $T_C^{(n)}$</th>
<th>Sh-Mem $S^{(n)}$</th>
<th>OMP $T_C^{(n)}$</th>
<th>OMP $S^{(n)}$</th>
<th>No-KT $T_C^{(n)}$</th>
<th>No-KT $S^{(n)}$</th>
<th>SKT $T_C^{(n)}$</th>
<th>SKT $S^{(n)}$</th>
<th>KT-W $T_C^{(n)}$</th>
<th>KT-W $S^{(n)}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>37.1</td>
<td>25.8</td>
<td>123</td>
<td></td>
<td>44.1</td>
<td>41.1</td>
<td>41.7</td>
<td>23.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>22.1</td>
<td>43.4</td>
<td>157</td>
<td></td>
<td>25.5</td>
<td>-</td>
<td>22.8</td>
<td>42.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 per core</td>
<td>19.7</td>
<td>48.3</td>
<td>197</td>
<td>23.1</td>
<td>26.6</td>
<td>402</td>
<td>2.4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Matrices of $1080 \times 1080$ elements, 100 iterations

Completion time of five-point stencil.

<table>
<thead>
<tr>
<th>$N$</th>
<th>Sh-Mem</th>
<th>OMP</th>
<th>No-KT</th>
<th>SKT</th>
<th>KT-W</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T^{(n)}_C$</td>
<td>$S^{(n)}$</td>
<td>$T^{(n)}_C$</td>
<td>$T^{(n)}_C$</td>
<td>$T^{(n)}_C$</td>
</tr>
<tr>
<td>30</td>
<td>37.1</td>
<td>25.8</td>
<td>123</td>
<td>44.1</td>
<td>41.1</td>
</tr>
<tr>
<td>60</td>
<td>22.1</td>
<td>43.4</td>
<td>157</td>
<td>25.5</td>
<td>-</td>
</tr>
<tr>
<td>2 per core</td>
<td>19.7</td>
<td>48.3</td>
<td>197</td>
<td>23.1</td>
<td>26.6</td>
</tr>
</tbody>
</table>
Matrices of $8640 \times 8640$ elements, 100 iterations
Matrices of $8640 \times 8640$ elements, 100 iterations

Completion time of five-point stencil.

<table>
<thead>
<tr>
<th>$N$</th>
<th>Sh-Mem</th>
<th>OMP</th>
<th>No-KT</th>
<th>SKT</th>
<th>KT-W</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T_C^{(n)}$</td>
<td>$S^{(n)}$</td>
<td>$T_C^{(n)}$</td>
<td>$T_C^{(n)}$</td>
<td>$T_C^{(n)}$</td>
</tr>
<tr>
<td>1</td>
<td>58.6</td>
<td>1</td>
<td>52.6</td>
<td>59.1</td>
<td>58.5</td>
</tr>
<tr>
<td>54</td>
<td>13.4</td>
<td>43.7</td>
<td>14.5</td>
<td>14.1</td>
<td><strong>13.9</strong></td>
</tr>
<tr>
<td>60</td>
<td><strong>13.1</strong></td>
<td>45.1</td>
<td><strong>14.4</strong></td>
<td><strong>13.6</strong></td>
<td>-</td>
</tr>
</tbody>
</table>
DATA-PARALLEL EXPERIMENT

Matrices of $8640 \times 8640$ elements, 100 iterations

![Graph showing completion time of five-point stencil for different parallelism degrees and methods.]

<table>
<thead>
<tr>
<th>$N$</th>
<th>Sh-Mem $T_C^{(n)}$</th>
<th>Sh-Mem $S^{(n)}$</th>
<th>OMP $T_C^{(n)}$</th>
<th>No-KT $T_C^{(n)}$</th>
<th>SKT $T_C^{(n)}$</th>
<th>KT-W $T_C^{(n)}$</th>
<th>SKT $S^{(n)}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>58.6</td>
<td>1</td>
<td>52.6</td>
<td>59.1</td>
<td>58.5</td>
<td>59.3</td>
<td>1</td>
</tr>
<tr>
<td>54</td>
<td>13.4</td>
<td>43.7</td>
<td>14.5</td>
<td>14.1</td>
<td><strong>13.9</strong></td>
<td>13.9</td>
<td>45.7</td>
</tr>
<tr>
<td>60</td>
<td><strong>13.1</strong></td>
<td>45.1</td>
<td><strong>14.4</strong></td>
<td><strong>13.6</strong></td>
<td>-</td>
<td><strong>13.6</strong></td>
<td>43.7</td>
</tr>
</tbody>
</table>
CONCLUSIONS

We have shown how a message-passing model represents a valuable alternatives on multy-/many-core. The performance gap w.r.t. shared-memory is quite limited thanks to:

1. Introduction of a lightweight support able to outperform MPI on fine grain computations;
2. Development of a delegation mechanism for overlapping communications with computation;
3. Exploitation of HMT that does not effect the Workers computation time.
CONCLUSIONS

We have shown how a message-passing model represents a valuable alternatives on multy-/many-core. The performance gap w.r.t. shared-memory is quite limited thanks to:

- introduction of a lightweight support able to outperform MPI on fine grain computations;
- development of a delegation mechanism for overlapping communications with computation;
- exploitation of HMT that does not effect the Workers computation time.
THANK YOU FOR YOUR ATTENTION

QUESTIONS?