Processes and Threads Placement of Parallel Applications. Why, How and for What gain?

Joint work with: Guillaume Mercier, François Tessier, Brice Goglin, Emmanuel Agulo, George Bosilca.
COST spring School, Uppsala
1

Runtime Systems and the Inria Runtime Team
Software Stack

Applications

Hardware
Software Stack

Applications

Programming models

Enable and express parallelism
Give abstraction of the parallel machine

Hardware
Software Stack

- Applications
- Programming models
- Compilers
- Hardware

- Enable and express parallelism
- Give abstraction of the parallel machine
- Static optimization
- Parallelism extraction

Process and Thread Placement
Software Stack

Applications

Programming models

Enable and express parallelism
Give abstraction of the parallel machine

Compilers

Static optimization
Parallelism extraction

Libraries

Optimize Computational Kernels

Hardware
Software Stack

Applications

Programming models

Libraries

Compilers

Operating systems

Hardware

- Enable and express parallelism
- Give abstraction of the parallel machine

- Static optimization
- Parallelism extraction

- Hardware abstraction
- Basic services

Optimize Computational Kernels
Software Stack

Applications

Programming models

Compilers

Libraries

Runtime systems

Operating systems

Hardware

- Enable and express parallelism
- Give abstraction of the parallel machine

Optimize Computational Kernels

- Dynamic optimization

Static optimization
- Parallelism extraction

Hardware abstraction
- Basic services
Runtime System

- Scheduling
- Parallelism orchestration (Comm. Synchronization)
- I/O
- Reliability and resilience
- Collective communication routing
- Migration
- Data and task/process/thread placement
- etc.
Runtime Team

Inria Team

Enable performance portability by improving interface expressivity

Success stories:
• MPICH 2 (Nemesis Kernel)
• KNEM (enabling high-performance intra-node MPI communication for large messages)
• StarPU (unified runtime system for CPU and GPU program execution)
• HWLOC (portable hardware locality)
Performance of MPI programs depends on many factors that can be handled when you change the machine:

- Implementation of the standard (e.g. collective com.)
- Parallel algorithm(s)
- Implementation of the algorithm
- Underlying libraries (e.g. BLAS)
- Hardware (processors, cache, network)
- etc.

But...
Process Placement

The MPI model makes little (no?) assumption on the way processes are mapped to resources.

It is often assume that the network **topology is flat** and hence the process mapping has little impact on the performance.
The Topology is not Flat

Due to multicore processors current and future parallel machines are hierarchical

Communication speed depends on:
• Receptor and emitter
• Cache hierarchy
• Memory bus
• Interconnection network
etc.

Almost nothing in the MPI standard help to handle these factors
Example on a Parallel Machine

The higher we have to go into the hierarchy the costly the data exchange
Example on a Parallel Machine

The higher we have to go into the hierarchy the costly the data exchange
Example on a Parallel Machine

The higher we have to go into the hierarchy the costly the data exchange.
The higher we have to go into the hierarchy the costly the data exchange
Example on a Parallel Machine

The higher we have to go into the hierarchy the costly the data exchange
Example on a Parallel Machine

The higher we have to go into the hierarchy the costly the data exchange.
Example on a Parallel Machine

The higher we have to go into the hierarchy the costly the data exchange
Example on a Parallel Machine

The higher we have to go into the hierarchy the costly the data exchange.

The network can also be hierarchical!
Rationale

Not all the processes exchange the same amount of data

The speed of the communications, and hence performance of the application depends on the way processes are mapped to resources.
Do we Really Care: to Bind or not to Bind?

After all, the system scheduler is able to move processes when needed.

Yes, but only for shared memory system. Migration is possible but it is not in the MPI standard (see charm++)

Moreover binding provides better execution runtime stability.
Do we Really Care: to Bind or not to Bind?

Zeus MHD Blast. 64 Processes/Cores. Mvapich2 1.8. + ICC
Process Placement Problem

Given:

- Parallel machine **topology**
- Process **affinity** (communication pattern)

**Map processes** to resources (cores) to reduce communication cost: a nice algorithmic problem:
- Graph partitionning (Scotch, Metis)
- Application tuning [Aktulga et al. Euro-Par 12]
- Topology-to-pattern matching (TreeMatch)
Reduce Communication Cost?

But wait, my application is compute-bound!

Well, but this might not be still true in the future: strong scaling might not always be a solution.
# Reduce Communication Cost?

<table>
<thead>
<tr>
<th>Systems</th>
<th>2010</th>
<th>2018</th>
<th>Difference Today &amp; 2018</th>
</tr>
</thead>
<tbody>
<tr>
<td>System peak</td>
<td>2 Pflop/s</td>
<td>1 Eflop/s</td>
<td>O(1000)</td>
</tr>
<tr>
<td>Power</td>
<td>6 MW</td>
<td>~20 MW</td>
<td></td>
</tr>
<tr>
<td>System memory</td>
<td>0.3 PB</td>
<td>32 - 64 PB</td>
<td>[ .03 Bytes/Flop ]</td>
</tr>
<tr>
<td>Node performance</td>
<td>125 GF</td>
<td>1,2 or 15TF</td>
<td>O(10) - O(100)</td>
</tr>
<tr>
<td>Node memory BW</td>
<td>25 GB/s</td>
<td>2 - 4TB/s</td>
<td>[ .002 Bytes/Flop ]</td>
</tr>
<tr>
<td>Concurrency</td>
<td>12</td>
<td>O(1k) or 10k</td>
<td>O(100) - O(1000)</td>
</tr>
<tr>
<td>Total Node Interconnect BW</td>
<td>3.5 GB/s</td>
<td>200-400GB/s</td>
<td>O(100)</td>
</tr>
<tr>
<td></td>
<td>(1:4 or 1:8 from memory BW)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>System size (nodes)</td>
<td>18,700</td>
<td>O(100,000) or O(1M)</td>
<td>O(10) - O(100)</td>
</tr>
<tr>
<td>Total concurrency</td>
<td>225,000</td>
<td>O(billion) [O(10) to O(100) for latency hiding]</td>
<td>O(10,000)</td>
</tr>
<tr>
<td>Storage</td>
<td>15 PB</td>
<td>500-1000 PB (&gt;10x system memory is min)</td>
<td>O(10) - O(100)</td>
</tr>
<tr>
<td>IO</td>
<td>0.2 TB</td>
<td>60 TB/s (how long to drain the machine)</td>
<td>O(100)</td>
</tr>
<tr>
<td>MTTI</td>
<td>days</td>
<td>O(1 day)</td>
<td>- O(10)</td>
</tr>
</tbody>
</table>

Taken from one of J. Dongarra’s Talk.
How to bind Processes to Core/Node?

MPI standard does not specify process binding

Each distribution has its own solution:

- MPICH2 (hydra manager): `mpiexec -np 2 -binding cpu:sockets`
- OpenMPI: `mpiexec -np 64 -bind-to-board`
- etc.

You can also specify process binding using `numactl` or `taskset` unix command in the mpirun command line:

```
mpiexec -np 1 --host machine numactl --physcpubind=0 ./prg
```
Obtaining the Topology (Shared Memory)

HWLOC (portable hardware locality):
• Runtime and OpenMPI team
• Portable abstraction (across OS, versions, architectures, ...)
• Hierarchical topology
• Modern architecture (NUMA, cores, caches, etc.)
• ID of the cores
• C library to play with
• Etc
HWLOC

http://www.open-mpi.org/projects/hwloc/
Obtaining the Topology (Distributed Memory)

Not always easy (research issue)

MPI core has some routine to get that

Sometime requires to build a file that specifies node adjacency
Getting the Communication Pattern

No automatic way so far...

Can be done through application monitoring:
- During execution
- With a « blank execution »
Results

Processes binding on ZEUS-MP/2 CFD application - metric : msg - processus : 251

- Chaco
- MPIPP1
- MPIPP5
- Packed
- ParMETIS
- RR
- Scotch
- TreeMatch

 Execution time (in seconds)

64 nodes linked with an Infiniband interconnect (HCA: Mellanox Technologies, MT26428 ConnectX IB QDR).
Each node features two Quad-core INTEL XEON NEHALEM X5550 (2.66 GHz) processors.
Results

Processes binding on ZEUS-MP/2 CFD application - metric : msg - processus : 25

64 nodes linked with an Infiniband interconnect (HCA: Mellanox Technologies, MT26428 ConnectX IB QDR).
Each node features two Quad-core INTEL XEON NEHALEM X5550 (2.66 GHz) processors.

36% gain against standard MPI policy
Results

Processes binding on ZEUS-MP/2 CFD application - metric : msg - processus : 256

64 nodes linked with an Infiniband interconnect (HCA: Mellanox Technologies, MT26428 ConnectX IB QDR).
Each node features two Quad-core INTEL XEON NEHALEM X5550 (2.66 GHz) processors.
Results

Processes binding on ZEUS-MP/2 CFD application - metric: msg - processus: 25

64 nodes linked with an Infiniband interconnect (HCA: Mellanox Technologies, MT26428 ConnectX IB QDR).

Each node features two Quad-core INTEL XEON NEHALEM X5550 (2.66 GHz) processors.

400% gain against some graph partitionners

June 4, 2013
Conclusion of this Part

To ensure performance probability one must take into account the topology of target machine.

Process placement according to application behavior and topology helps in increasing performance.

Several open issues:
- Communication pattern
- Metrics
- Dynamic adaptation
- Faster algorithm
- Integration into MPI (dist_graph_create + new communicator)
Thread placement on shared-memory
Multithreading is a good model for shared memory machine.
Multithreading is a good model for shared memory machine.
Multithreading

But threads and/or memory pages can move
Multithreading

But threads and/or memory pages can move
Multithreading

But threads and/or memory pages can move

Process and Thread Placement
Multithreading

But threads and/or memory pages can move
Multithreading

But threads and/or memory pages can move
Multithreading

But threads and/or memory pages can move
Multithreading

But threads and/or memory pages can move

Scheduler decision
Thread Binding
You cannot prevent memory pages from moving but you can:
• Bind threads to nodes/cores (HWLOC)
• Allocate memory pages on specific memory node
Thread Binding
You cannot prevent memory pages from moving but you can:
• Bind threads to nodes/cores (HWLOC)
• Allocate memory pages on specific memory node
Thread Binding

You cannot prevent memory pages from moving but you can:
• Bind threads to nodes/cores (HWLOC)
• Allocate memory pages on specific memory node

![Diagram showing process and thread placement with memory controller and local RAM.]
Thread Binding
You cannot prevent memory pages from moving but you can:
- Bind threads to nodes/cores (HWLOC)
- Allocate memory pages on specific memory node

Mem. Controller
Core
L1/L2
L3
Local RAM

Interconnect

Mem. Controller
Core
L1/L2
L3
Local RAM

Process and Thread Placement
Thread Binding

You cannot prevent memory pages from moving but you can:

- Bind threads to nodes/cores (HWLOC)
- Allocate memory pages on specific memory node

![Diagram of Thread Binding](image-url)

Process and Thread Placement
Thread Binding

You cannot prevent memory pages from moving but you can:

- Bind threads to nodes/cores (HWLOC)
- Allocate memory pages on specific memory node

![Diagram showing process and thread placement with Local RAM, Mem. Controller, L3, L1/L2, and Core components connected through an interconnect.]
Example on the Tiled Version of the Dense Cholesky Factorization

The tiled version. A square matrix, symmetric definite positive decomposed in squares tiles

```plaintext
for k = 0...T - 1 do
    A[k][k] ← DPOTRF(A[k][k])
    for m = k + 1...T - 1 do
        A[m][k] ← DTRSM(A[k][k], A[m][k])
    end
    for n = k + 1...T - 1 do
        A[n][n] ← DSYRK(A[k][k], A[n][n])
        for m = n + 1...T - 1 do
            A[m][n] ← DGEMM(A[k][k], A[n][n], A[m][n])
        end
    end
end
```

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td>0,1</td>
<td>0,2</td>
<td>0,3</td>
<td>0,4</td>
</tr>
<tr>
<td>1,0</td>
<td>1,1</td>
<td>1,2</td>
<td>1,3</td>
<td>1,4</td>
</tr>
<tr>
<td>2,0</td>
<td>2,1</td>
<td>2,2</td>
<td>2,3</td>
<td>2,4</td>
</tr>
<tr>
<td>3,0</td>
<td>3,1</td>
<td>3,2</td>
<td>3,3</td>
<td>3,4</td>
</tr>
<tr>
<td>4,0</td>
<td>4,1</td>
<td>4,2</td>
<td>4,3</td>
<td>4,4</td>
</tr>
</tbody>
</table>

DPOTRF
DTRSM
DSYRK
DGEMM

Process and Thread Placement
Example on the Tiled Version of the Dense Cholesky Factorization

Cholesky Factorization on 160 cores

- SMA with Grouping per node
- SMA Grouping per core
- SMA Grouping per machine

Order of the Matrix (N)

Gflop/s
Example on the Tiled Version of the Dense Cholesky Factorization

On a 20 nodes, 8 cores per node, shared memory machine
20 pool of threads vs. 1 pool of thread
Example on the Tiled Version of the Dense Cholesky Factorization

On a 20 nodes, 8 cores per node, shared memory machine
20 pool of threads vs. 1 pool of thread

Perf degradation!
Problem: System has 600Gb of Ram. But we start swapping at $N=64000$ (30Gb)
What’s Happening at \( N=64000 \)?

Each node has 30Gb of Ram

`malloc`: pages are put on the first node that writes it.
What’s Happening at N=64000?

Each node has 30Gb of Ram
malloc: pages are put on the first node that writes it
What’s Happening at N=64000?

Each node has 30Gb of Ram
malloc: pages are put on the first node that writes it

define *A = malloc( N * LDA * sizeof(double));
fill(A);
What's Happening at N=64000?

Each node has 30Gb of Ram
malloc: pages are put on the first node that writes it

double *A = malloc( N * LDA * sizeof(double));
fill(A);

June 4, 2013
What’s Happening at N=64000?

Solution:
- have a multithreaded I/O for creating the matrix in tiled format
- allocate pages of matrix in round robin fashion onto the nodes (numa_alloc_interleaved)
What’s Happening at N=64000?

Solution:
- have a multithreaded I/O for creating the matrix in tiled format
- allocate pages of matrix in round robin fashion onto the nodes (numa_alloc_interleaved)

June 4, 2013
What’s Happening at N=64000?

Solution:
- have a multithreaded I/O for creating the matrix in tiled format
- allocate pages of matrix in round robin fashion onto the nodes (numa_alloc_interleaved)

```c
double *A = numa_alloc_interleaved( N * LDA * sizeof(double));
fill(A);
```
What’s Happening at N=64000?

Solution:
• have a multithreaded I/O for creating the matrix in tiled format
• allocate pages of matrix in round robin fashion onto the nodes (numa_alloc_interleaved)

double *A = numa_alloc_interleaved( N * LDA * sizeof(double));
fill(A);
Example on the Tiled Version of the Dense Cholesky Factorization

![Graph showing performance comparison between different thread configurations.](image)

- **No NumA Awareness**
- **Thread Binding**
- **Thread Binding + Data Interleaving**

1 group of threads vs. 20 groups of threads

June 4, 2013
Example on the Tiled Version of the Dense Cholesky Factorization

Cholesky Factorization on 160 threads

1 group of threads vs. 20 group of threads

Numa-aware data binding and allocation

June 4, 2013
### Comparison with Block-Cyclic distribution

**Global View**

<table>
<thead>
<tr>
<th></th>
<th>a_{11}</th>
<th>a_{12}</th>
<th>a_{13}</th>
<th>a_{14}</th>
<th>a_{15}</th>
<th>a_{16}</th>
<th>a_{17}</th>
<th>a_{18}</th>
<th>a_{19}</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_{21}</td>
<td>a_{22}</td>
<td>a_{23}</td>
<td>a_{24}</td>
<td>a_{25}</td>
<td>a_{26}</td>
<td>a_{27}</td>
<td>a_{28}</td>
<td>a_{29}</td>
<td></td>
</tr>
<tr>
<td>a_{31}</td>
<td>a_{32}</td>
<td>a_{33}</td>
<td>a_{34}</td>
<td>a_{35}</td>
<td>a_{36}</td>
<td>a_{37}</td>
<td>a_{38}</td>
<td>a_{39}</td>
<td></td>
</tr>
<tr>
<td>a_{41}</td>
<td>a_{42}</td>
<td>a_{43}</td>
<td>a_{44}</td>
<td>a_{45}</td>
<td>a_{46}</td>
<td>a_{47}</td>
<td>a_{48}</td>
<td>a_{49}</td>
<td></td>
</tr>
<tr>
<td>a_{51}</td>
<td>a_{52}</td>
<td>a_{53}</td>
<td>a_{54}</td>
<td>a_{55}</td>
<td>a_{56}</td>
<td>a_{57}</td>
<td>a_{58}</td>
<td>a_{59}</td>
<td></td>
</tr>
<tr>
<td>a_{61}</td>
<td>a_{62}</td>
<td>a_{63}</td>
<td>a_{64}</td>
<td>a_{65}</td>
<td>a_{66}</td>
<td>a_{67}</td>
<td>a_{68}</td>
<td>a_{69}</td>
<td></td>
</tr>
<tr>
<td>a_{71}</td>
<td>a_{72}</td>
<td>a_{73}</td>
<td>a_{74}</td>
<td>a_{75}</td>
<td>a_{76}</td>
<td>a_{77}</td>
<td>a_{78}</td>
<td>a_{79}</td>
<td></td>
</tr>
<tr>
<td>a_{81}</td>
<td>a_{82}</td>
<td>a_{83}</td>
<td>a_{84}</td>
<td>a_{85}</td>
<td>a_{86}</td>
<td>a_{87}</td>
<td>a_{88}</td>
<td>a_{89}</td>
<td></td>
</tr>
<tr>
<td>a_{91}</td>
<td>a_{92}</td>
<td>a_{93}</td>
<td>a_{94}</td>
<td>a_{95}</td>
<td>a_{96}</td>
<td>a_{97}</td>
<td>a_{98}</td>
<td>a_{99}</td>
<td></td>
</tr>
</tbody>
</table>

**Local (distributed) View**

- **P=2, Q=3**

![Diagram showing local distributed view](Image)
Comparison with Block-Cyclic distribution

Cholesky Factorization on 160 cores and NB=256

Cholesky Factorization on 160 cores and NB=512
Conclusion of this Part

Shared memory machine gives the illusion of a flat address space but:
  • data allocation
  • data movement
have a huge impact on the performance

We are lacking model and tools to hide/expose this complexity.
Conclusion
Take Away Message

Performance portability is difficult!

Parallel machines are more complex than you may think!

Important to take care of process/thread/data placement: big gain can be achieved.

Good news: does not require to change application or algorithm

Lot of works are required either:
• to hide complexity from the user while keeping expressivity
• or to expose what is necessary to ensure performance
Thanks!