Introducing Task-Containers as an Alternative to Runtime Stacking

EuroMPI, Edinburgh, UK
September 2016

Jean-Baptiste BESNARD
jbbesnard@paratools.fr

Julien ADAM, Sameer SHENDE, Allen MALONY (ParaTools)
Marc PÉRACHE, Patrick CARRIBAULT, Julien JAEGER (CEA, DAM, DIF)

Introduction (1/2)

MPI is a programming standard which shaped the use of Supercomputers for the last two decades. It is now deeply embedded in large industrial code-bases.

Is message passing suitable for new architectures ?

- Growing number of cores (Millions)
- Less memory per thread
- Larger nodes
- Many-core

Opened discussions around hybrid parallelism (MPI+X)

Introduction (2/2)

What are the alternatives ?

MPI + X:

- OpenMP
- Cuda
- PGAS
- OpenCL
- Threads
- TBB
- …

Complex decision for a « legacy » code. Which of these X will remain in a five year time-frame ?

Hybridization has a software development cost.

MPI Challenges on Many-Core (1/4)

68 Cores

272 Hyper-threads

How many MPI processes ?

272 ?

68 * 4 ?

1 * 272 ?

MPI Challenges on Many-Core (2/4)

	272	68 * 4	1 * 272
Parallelism	Pure MPI	Hybrid mostly MPI	Hybrid mostly X

Increasing OS processes poses problems of :

- Memory replication (halo cells, distributed memory)
- Polling replication (no collaborative polling)
- Communication buffers replication (SHM, IB QP)
- Pressure on the batch manager (launch time)
- Scalability (problem size per core VS communication)

MPI Challenges on Many-Core (3/4)

MPI is now installed on every machine and used in the vast majority of HPC codes. What if we were able to transition MPI codes, mitigating the need for hybridization ?

Evolutions in MPI

Evolutions in Codes

Trade-off

Evolutions in runtimes

MPI Challenges on Many-Core (4/4)

Evolutions in MPI	Evolutions in runtimes	Evolutions in codes
Endpoints	Optimizations for many-core	Hybridization
MPI Sessions	Unified runtimes (MPI + PGAS) (MPI+OpenMP) …	Use of RMAs (Shared windows)
Ownership passing	Thread-based MPI	Manual parallelism

↔

Minimal impact on end-users

Still an open research question

MPI Challenges on Many-Core (4/4)

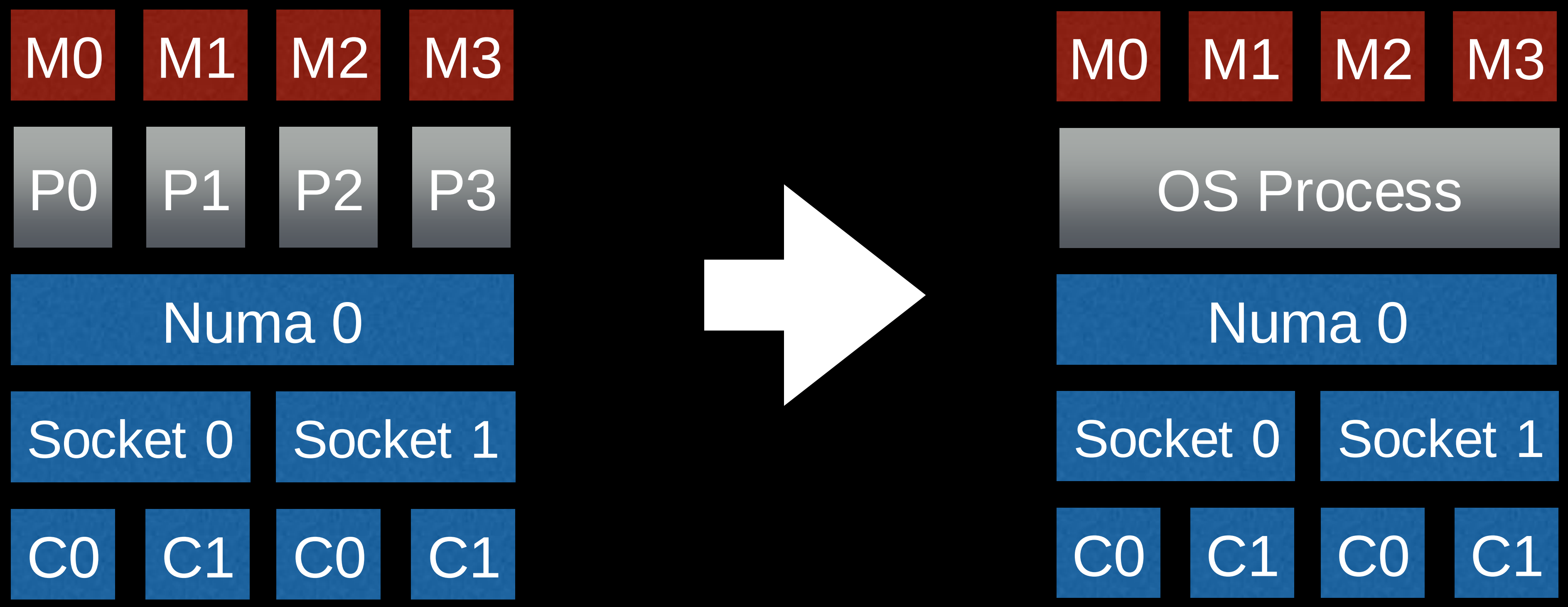
Evolutions in MPI	Evolutions in runtimes	Evolutions in codes
Endpoints	Optimization for many-core	Hybridization
MPI Sessions	Unified runtimes (MPI + PGAS) (MPI+OpenMP)	Use of RMAs (Shared windows)
Ownership passing	Thread-based MPI	Manual parallelism

←→

Minimal impact on end-users

Running MPI in Threads (1/3)

IDEA : transpose the MPI standard to lightweight threads **transparently** to get MPI+X advantages while programming in pure MPI.



Running MPI in Threads (2/3)

Increasing OS processes poses problems of :

- Memory replication (halo cells, distributed memory)
- Polling replication (no collaborative polling)
- Communication buffers replication (SHM, IB QP)
- Pressure on the batch manager (launch time)
- Scalability (problem size per core VS communication)

Addressed by thread-based

Requires « shared-memory » MPI constructions

Running MPI in Threads (3/3)

The Extended TLS Library

is gathering the work done in the MPC runtime around compiler level privatization in order to transpose MPI codes to user-level threads (ULT) without code modification. Compatible with both GCC and ICC.

We open this work to the community (outside of MPC itself) to ease the exploration of solutions involving ULT

TLS (Thread-Local Storage)

User-Level Threads and MPI (1/3)

We want to run processes as threads while preserving the initial semantic, in particular global variables.

```
#include <mpi.h>
#include <stdio.h>

int rank = -1;

int main( int argc, char ** argv )
{
    MPI_Init( &argc, &argv );

    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

        printf("My rank %d\n», rank );

    MPI_Finalize();

    return 0;
}
```

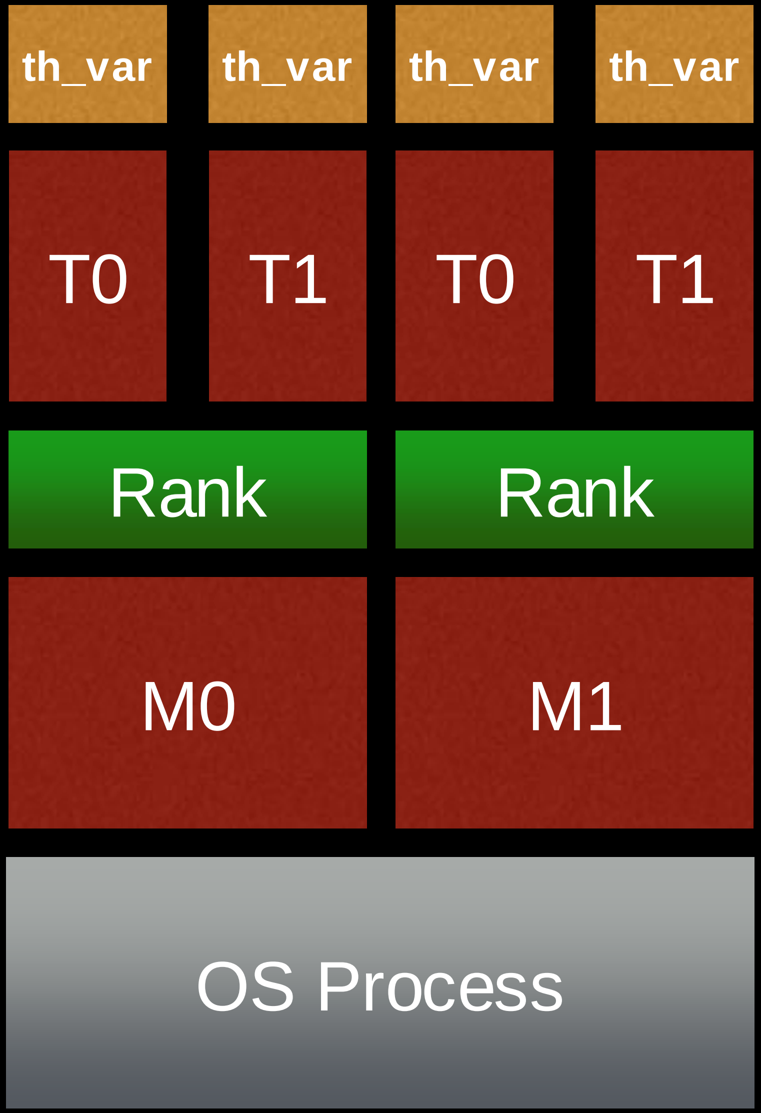
User-Level Threads and MPI (2/3)

Global variables privatization, our alternatives :

Source-to-source	Compilation time	Runtime
Array expansion	**Privatization to TLS (AMPI, MPC)**	GOT switching (AMPI)
AMPI Photran	Code transformation (globals as argument)	HMPI common heap

OS Level : Thread-Shared Private Library (TSPL)

User-Level Threads and MPI (3/3)



th_var th_var th_var th_var

T0 T1 T0 T1

Rank Rank

M0 M1

OS Process

```
int rank = -1;
__thread int th_var = -1;
```

```
int omprank = -1;
#pragma omp threadprivate(omprank)
```

The compilation approach was retained to support **context stacking**, allowing the « transparent » transposition of OS processes to threads.

ExTLS Privatisation Interface (1/3)

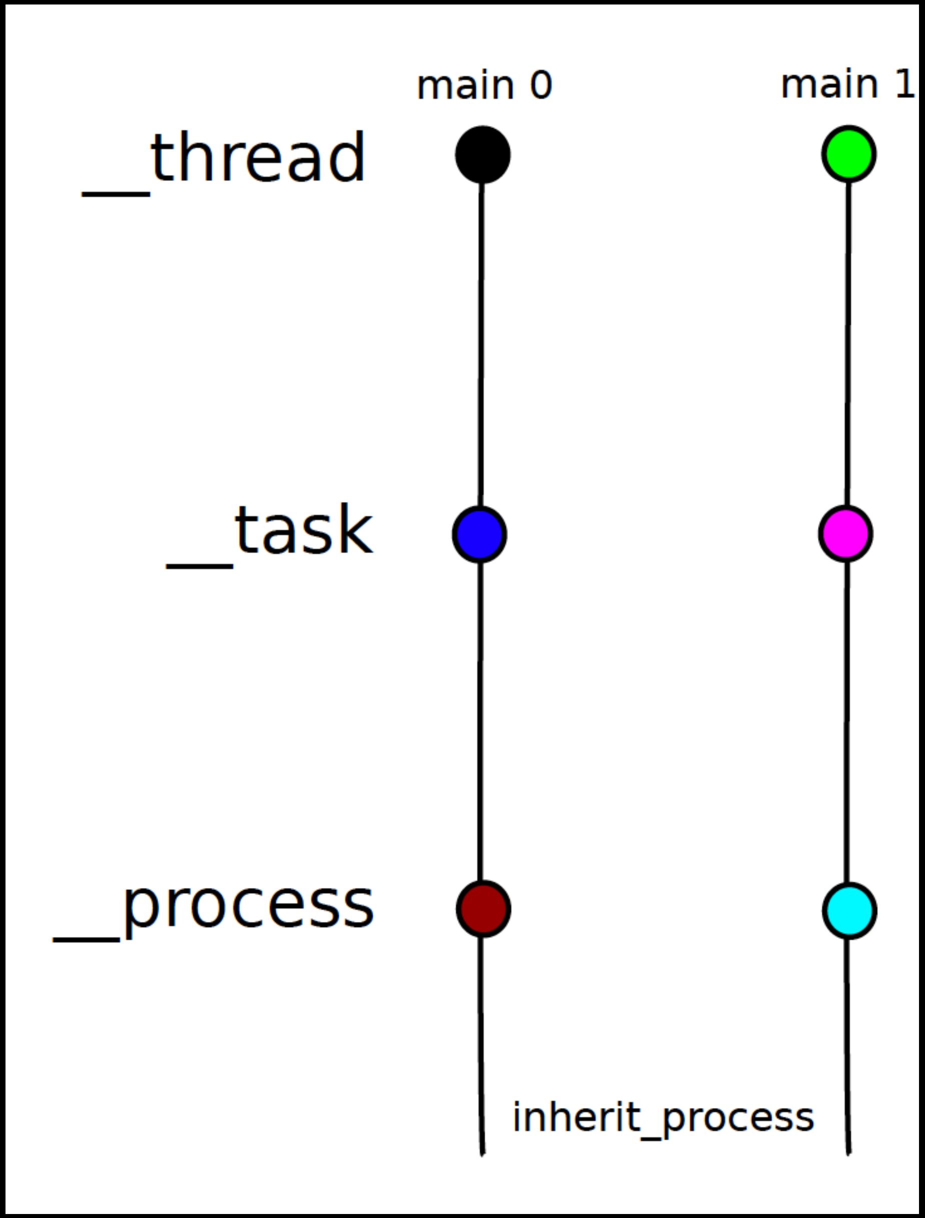
New keywords

Standard case	Privatization (C)	Privatization (C++)
-	**__openmp**	**openmp_local**
__thread	__thread	thread_local
-	**__task**	**task_local**
Global	**__process**	**process_local**

By default Global variables are privatized to task level (automatic privatization).

This lead to the definition of the task-container.

ExTLS Privatisation Interface (2/3)



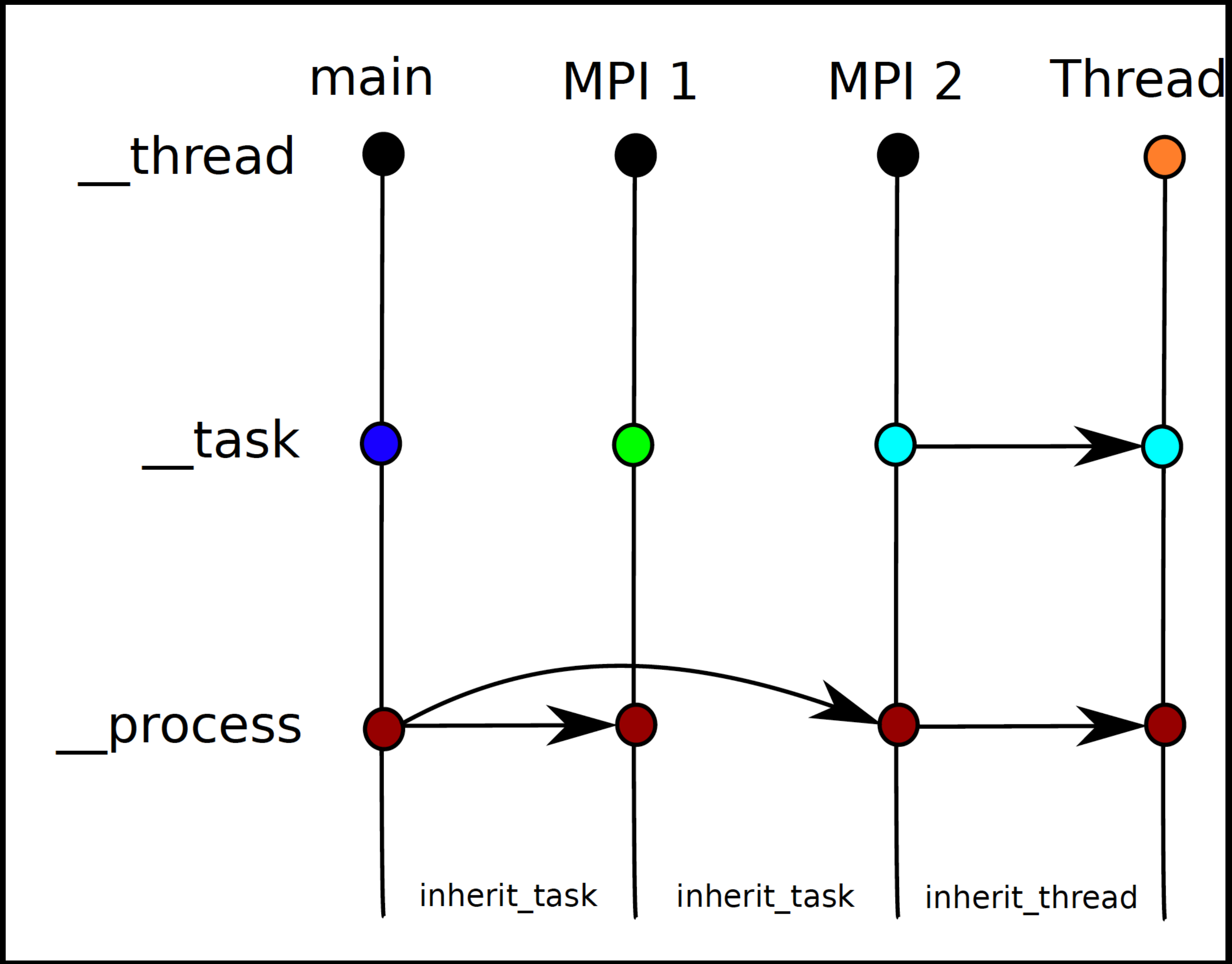
inherit_process()

main 0

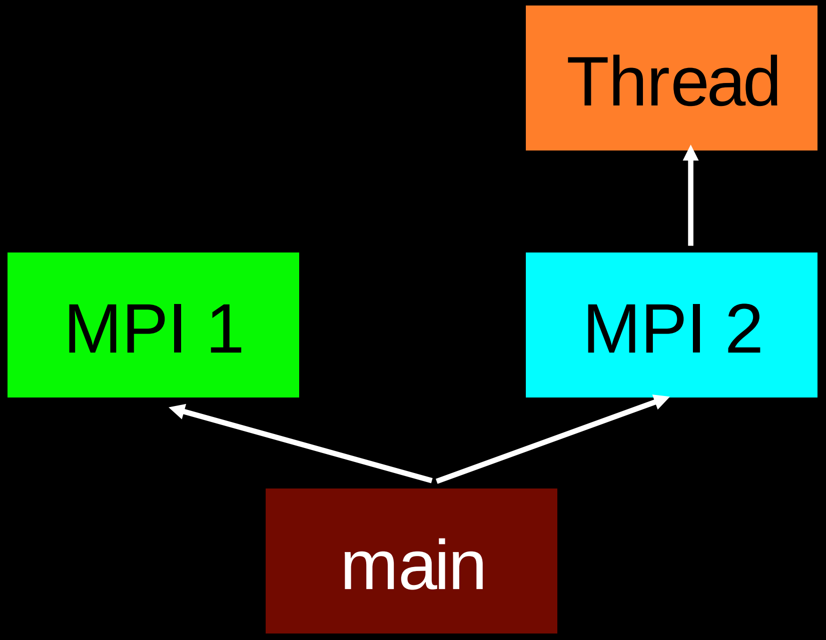
main 1

Mimic disjoint processes
in shared-memory

ExTLS Privatisation Interface (3/3)



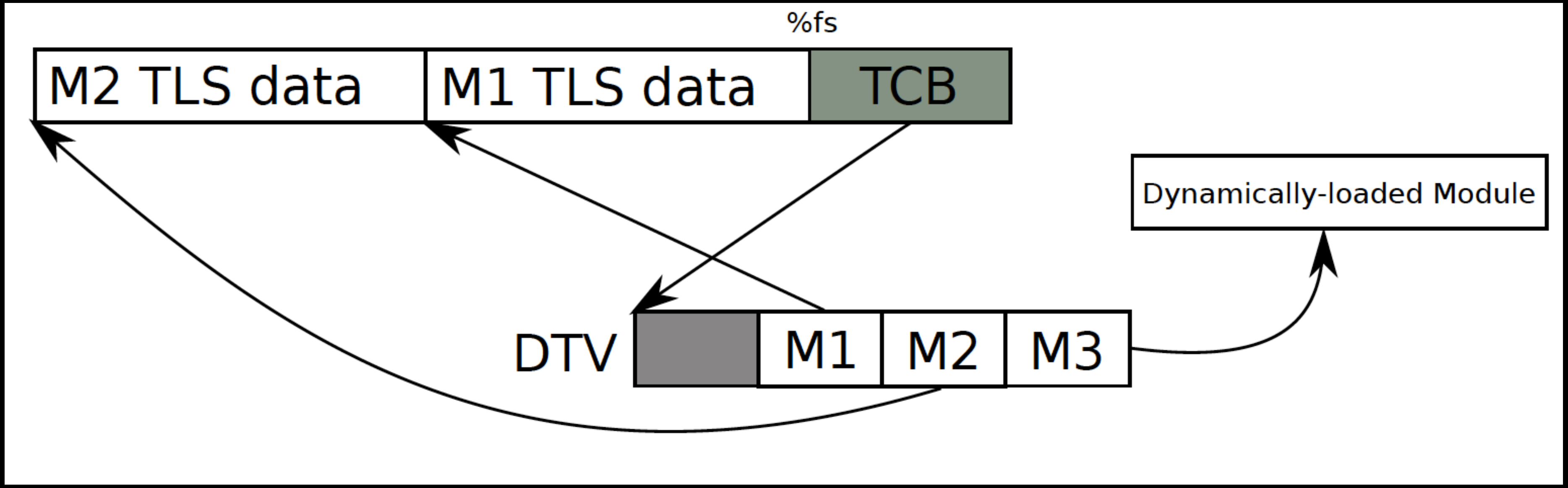
inherit_task()
inherit_thread()



Inheritance as an alternative to *fork*,
in order to express contexts' hierarchy.

Building block for a thread-based MPI using « task-containers »

ExTLS Implementation (1/2)



%fs

M2 TLS data

M1 TLS data

TCB

Dynamically-loaded Module

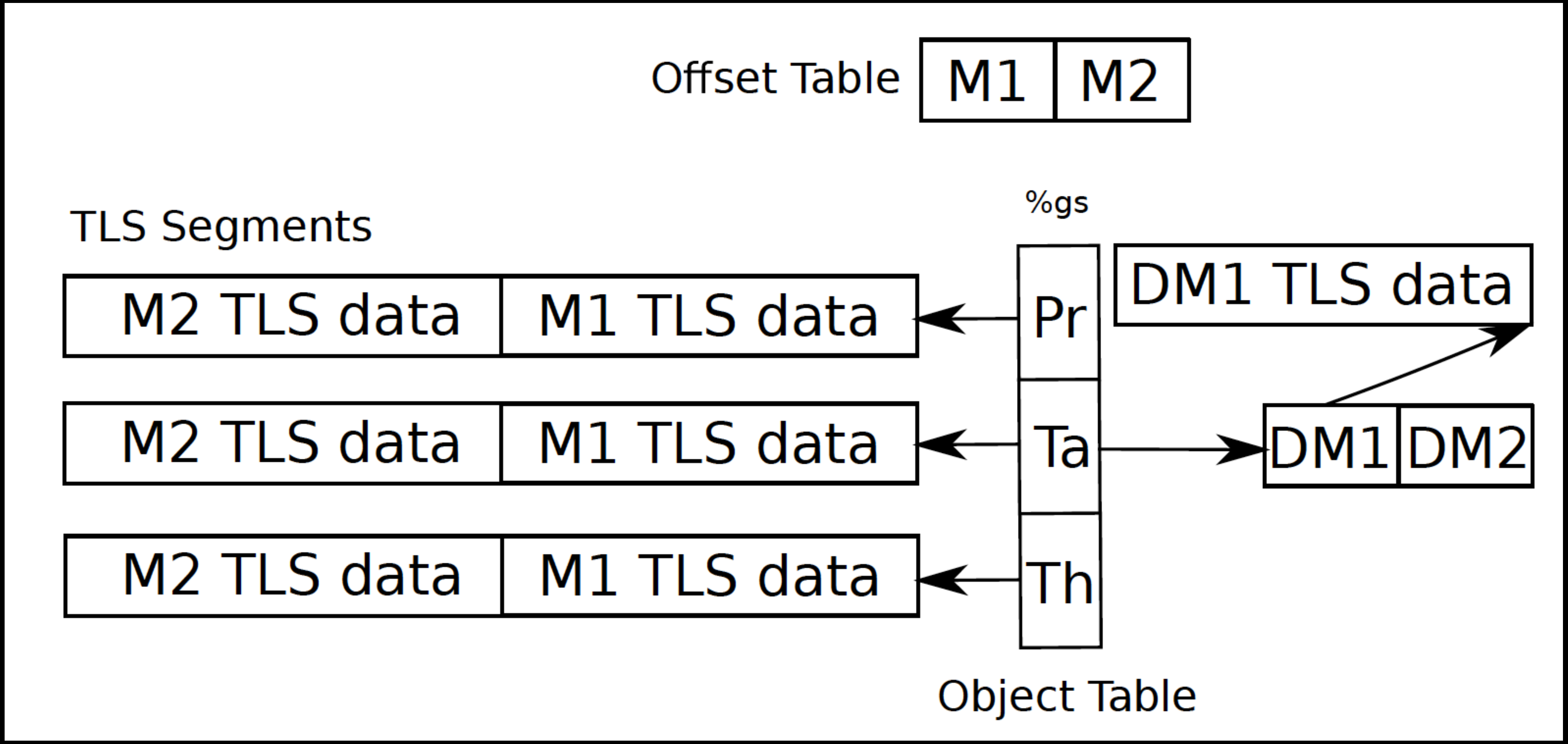
DTV

M1

M2

M3

libC



Offset Table

M1

M2

TLS Segments

%gs

M2 TLS data

M1 TLS data

Pr

DM1 TLS data

M2 TLS data

M1 TLS data

Ta

DM1

DM2

M2 TLS data

M1 TLS data

Th

Object Table

libExTLS

ExTLS Implementation (2/2)

GCC	GCC Plugin	Linker	Runtime
Add keywords in the C and C++ front-end	Handle dynamic initialization chains	Implement TLS optimizations	Inheritance calls
Add new TLS levels to handle ExTLS	Wrap TLS initializers for each thread	Support for TLS levels	Context switching (ULT)
Add TLS definitions to the x86_64 backend	Inject Initializers (optional)	Export initializers in the dynamic symbol table	Topology detection (HLS)

- Patched GCC (Front-end & Backend)
- GCC Plugin (Dynamic initializers)
- Patched linker (TLS optimizations)
- Runtime library (libExTLS)

ParaTools

Dynamic Initializers (1/3)

main.c

```
extern int * a;

int main( int argc, char ** argv )
{
	printf(« Value %d\n», *a );
   return 0;
}
```

a.c

```
extern int b;
int * a = &b;
```

b.c

```
int b = 99;
```

When privatized, this code does not compile as « *&b* » is not a constant (« *b* » being transitioned to TLS).
The role of the privatization plugin is to support this pattern
To do so, we proposed two methods:

- Code injection
- Introspection

ParaTools

Dynamic Initializers (2/3)

main.c

```
extern int * a;

int main( int argc, char ** argv )
{
        tls_init_a();
        printf(« Value %d\n», *a );
    return 0;
}
```

b.c

```
int b = 99;

void tls_init_b()
{
     tls_init();
}

static void tls_init()
{
}
```

a.c

```
extern int b;
int * a = &b;

void tls_init_a()
{
     tls_init();
}



static void tls_init()
{
   tls_init_b();
     static int init_done = 0;
     if( !init_done )
     {
          init_done = 1;
          a = &b;
     }
}
```

Code injection

Dynamic Initializers (3/3)

main.c

```
extern int * a;

int main( int argc, char ** argv )
{
        //tls_init_a();
        printf(« Value %d\n», *a );
    return 0;
}
```

Introspection

Do not inject function calls at each function start. Self-scan the executable for « *tls_init_** » symbols and call them when launching each thread. TLS symbols must be exported in the dynamic symbol table (reachable through *dlsym* even in the main exe).

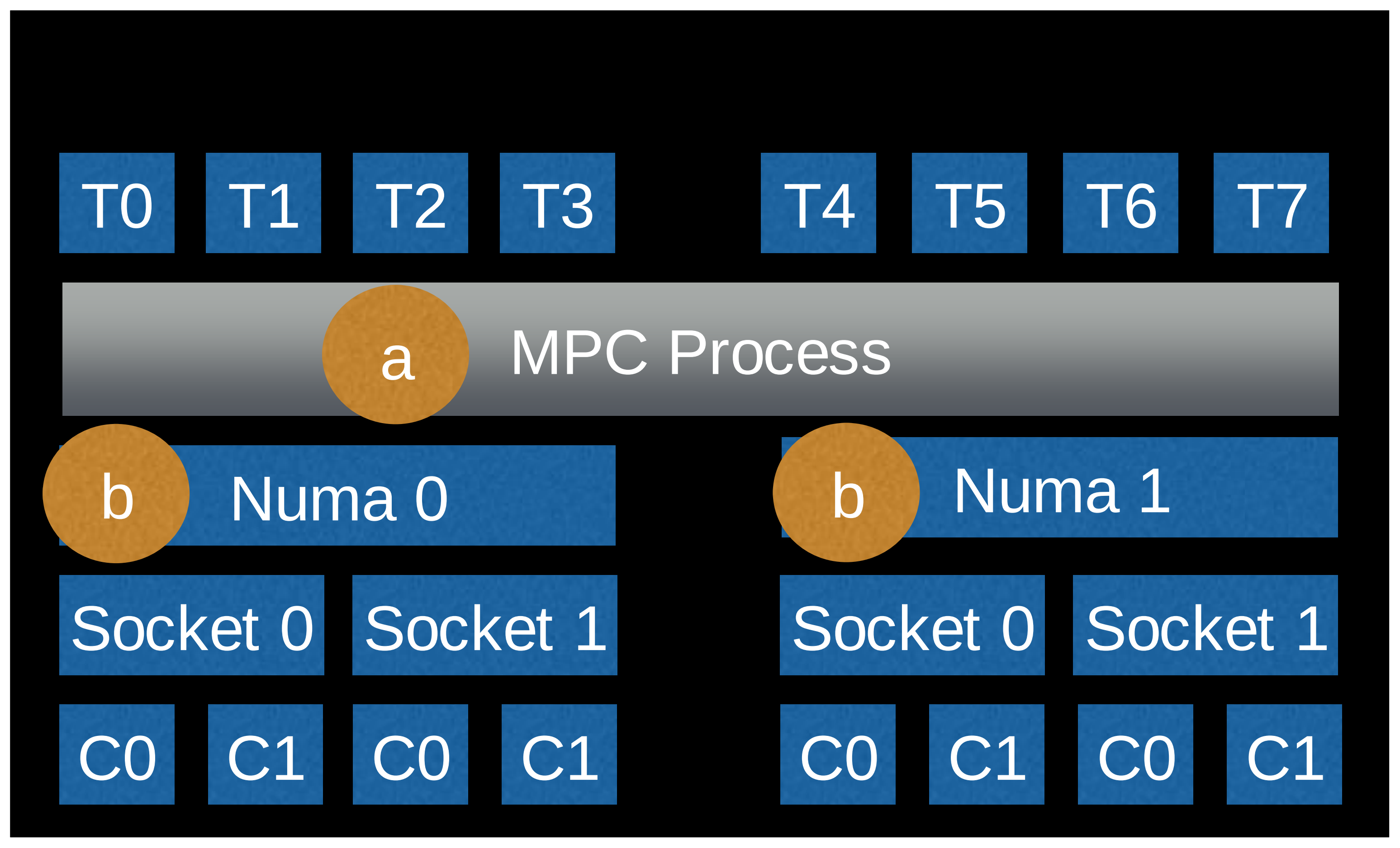
Performance Gains

	Code Injection	**Introspection**
HDF5 Test-Suite Wall-time	244,8	94,5

Hierarchical Local Storage

Use Hierarchical Local Storage to store data at a given topological level to avoid data replication.

```
int a,b ;
#pragma hls node(a)
#pragma hls numa(b)
```



M. Tchiboukdjian, P. Carribault, and M. Pérache, Hierarchical Local Storage: Exploiting Flexible User-Data Sharing Between MPI Tasks, in IEEE International Parallel and Distributed Processing (IPDPS'12), 2012.

MPI One-Sided in Thread-Based

In a thread-based, one-sided operations are simple *memcpy*, providing multiple advantages:

- Direct access to remote data (no shared pools)
- Immediate flush (shared window proprieties)
- Simpler synchronization primitives (RW locks)
- Optimized support for dynamic windows
- Easier support for derived data-types

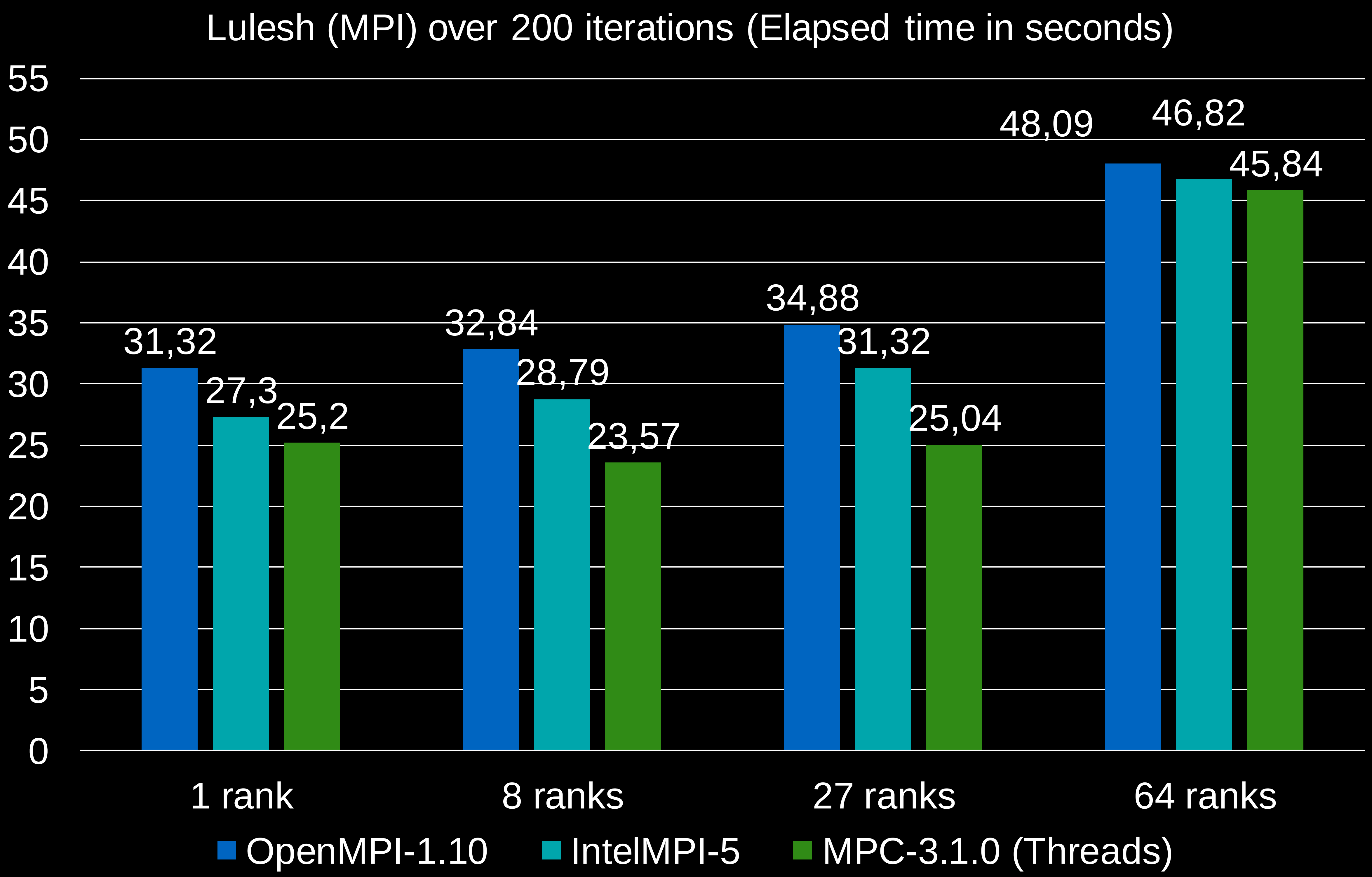
Current development version of MPC implements MPI 3.1 One-sided taking advantage of thread-based with promising results.

Already Working With MPC

- HDF5
- Pastix
- Scotch
- TBB
- Parallel Ocean Program (POP)
- MCB
- UMT2013
- Nekbone
- Lulesh
- Graph500

Process facilitated by libExTLS automatic privatization support

Lulesh on the KNL



All tests done with ICC

Conclusion

There are challenges for MPI when running on many-core processors
- Launch time, memory replication, buffer replication, …

There are different alternatives to address these challenges
- MPI, Applications or runtimes

Our solution approach is to run MPI on lightweight threads
- Addresses several of the challenges but introduces privatization issues

The ExTLS library is proposed to address these issues.
- Compatible with our own GCC and ICC (except dynamic initializers)

ExTLS is demonstrated in the context of MPC
- HLS, MPI One-sided, code porting

Future Work

- Use of task-containers for In-Situ
  Orthogonally embed IO processsing in an MPI program
- Define a multi-main MPMD program
  Multiple collaborating components in shared-memory
- MPC with One-sided is to be released in the next version of MPC

The ExTLS library implementation is already available online as part of the MPC 3.1.0 MPI runtime: http://mpc.hpcframework.com

Introducing Task-Containers as an Alternative to Runtime Stacking

EuroMPI, Edinburgh, UK
September 2016

Jean-Baptiste BESNARD
jbbesnard@paratools.fr

Julien ADAM, Sameer SHENDE, Allen MALONY (ParaTools)
Marc PÉRACHE, Patrick CARRIBAULT, Julien JAEGER (CEA, DAM, DIF)

ParaTools