BARCELONA **ZETTASCALE** LAB

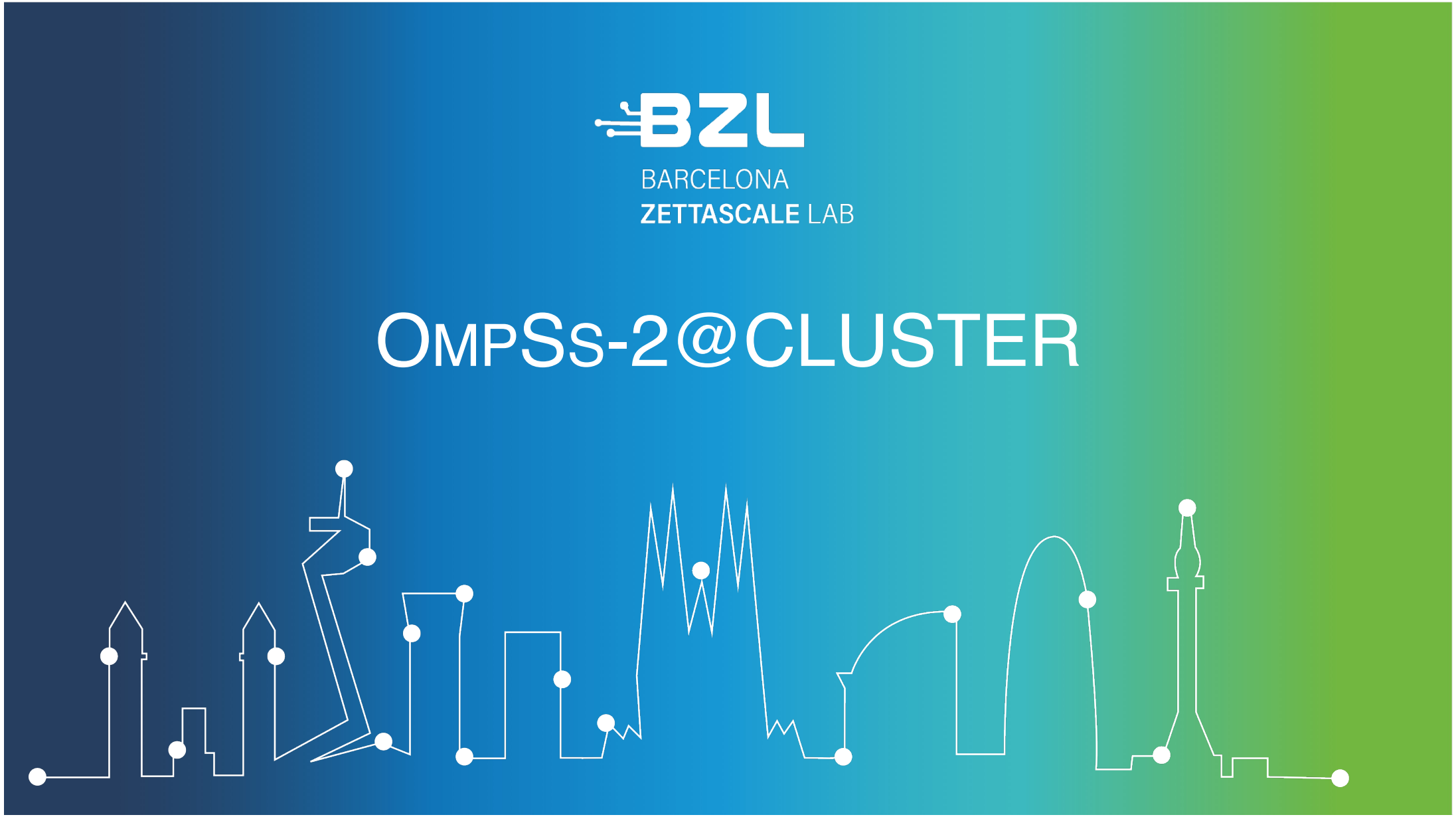
Distributed task execution: Opportunities, challenges and lessons learnt from OmpSs-2@Cluster

Paul Carpenter – Barcelona Supercomputing Center
PARMA-DITAM Workshop - Krakow – 27/1/2026

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

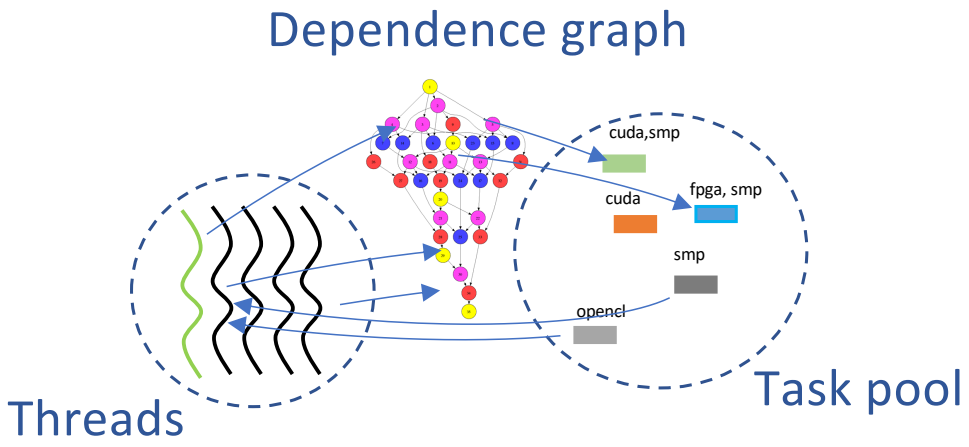
Introduction

- **Tasking works extremely well at node level**
  - Mainstream in HPC via OpenMP
  - Also employed in Intel TBB, Cilk, HPX, StarPU, PaRSEC, Legion, OmpSs
- **… and at workflow level**
  - COMPSs, Pegasus, etc.
- **However, at intermediate scale (distributed memory HPC) it hits fundamental scalability limits**
  - Sequential task creation
  - Centralized dependency tracking
- **This talk is about what breaks, what we fixed, and where tasking makes sense**


BZL
BARCELONA
ZETTASCALE LAB
OMPSS-2@CLUSTER

OmpSs/OmpSs-2

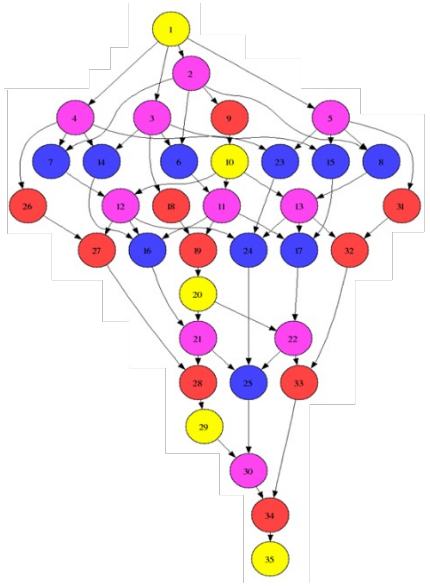
- **In common with rest of StarSs family (incl. COMPSs)**
  - Sequential code + annotations to define tasks and accesses
  - Single address space (or namespace)
  - Runtime builds DAG, which happens to execute in parallel
  - Automatic runtime computation of dependencies



Data accesses: single mechanism for concurrency and data

- **Concurrency**: Runtime computes task dependencies from accesses
- **Data locality**: NUMA, accelerator, node
- **Data transfers**: host–accelerator, among nodes

```
void Cholesky(int NT, float *A[NT][NT] ) {
   for (int k=0; k<NT; k++) {
      #pragma omp task inout ([TS][TS](A[k][k]))
      spotrf (A[k][k], TS) ;
      for (int i=k+1; i<NT; i++) {
        #pragma omp task in([TS][TS](A[k][k])) inout ([TS][TS](A[k][i]))
        strsm (A[k][k], A[k][i], TS);
      }
      for (int i=k+1; i<NT; i++) {
        for (j=k+1; j<i; j++) {
          #pragma omp task in([TS][TS](A[k][i]), [TS][TS](A[k][j])) \
                           inout ([TS][TS](*A[j][i]))
          sgemm( A[k][i], A[k][j], A[j][i], TS);
        }
        #pragma omp task in ([TS][TS](A[k][i])) inout([TS][TS](A[i][i]))
        ssyrk (A[k][i], A[i][i], TS);
      }
   }
}
```

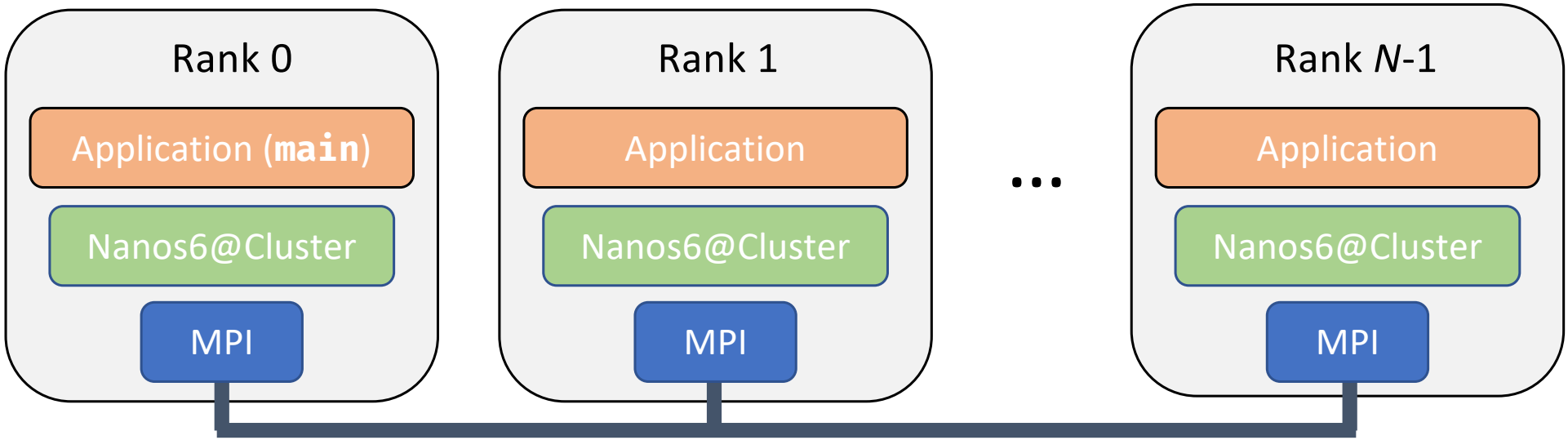


OmpSs-2@Cluster

- **Extends OmpSs-2 tasking model to multiple nodes**
  - Tasks are transparently offloaded to other MPI ranks
  - Task dependencies and data transfers managed by runtime system
  - Scheduling is based on locality and load balancing
  - Communication is done via MPI
- **Objectives**
  - Alternative to MPI for small scale
    - Medium to large scale if there is program structure
  - Can be used to address inter-node load imbalance for MPI + OmpSs-2

OmpSs-2@Cluster architecture

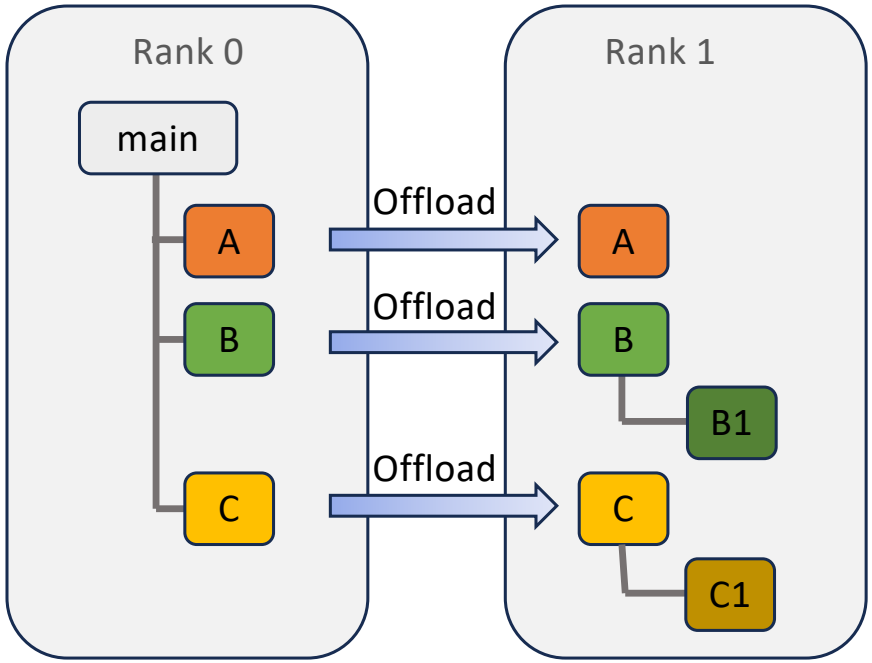
- Program is compiled using regular OmpSs-2 LLVM compiler
- Executed using `mpirun`/`mpiexec`/`srun` as normal for an MPI program
- Each process has instance of Nanos6@Cluster runtime
- `main` is executed as a task on the first rank (ranks are otherwise symmetrical)



OmpSs-2: Improved nesting + dependencies

- **Before Task A completes execution…**
- **Offload B and C to Rank 1 and run them to create their subtasks**
  - Would not be possible if B and C had “inout” accesses

```
1  #pragma oss task out(a) label("A")
2  { a = 1; }
3
4  #pragma oss task weakinout(a) label("B")
5  {
6      #pragma oss task inout(a) label("B1")
7      { a++; }
8  }
9
10 #pragma oss task weakinout(a) label("C")
11 {
12     #pragma oss task inout(a) label("C1")
13     { a++; }
14 }
```

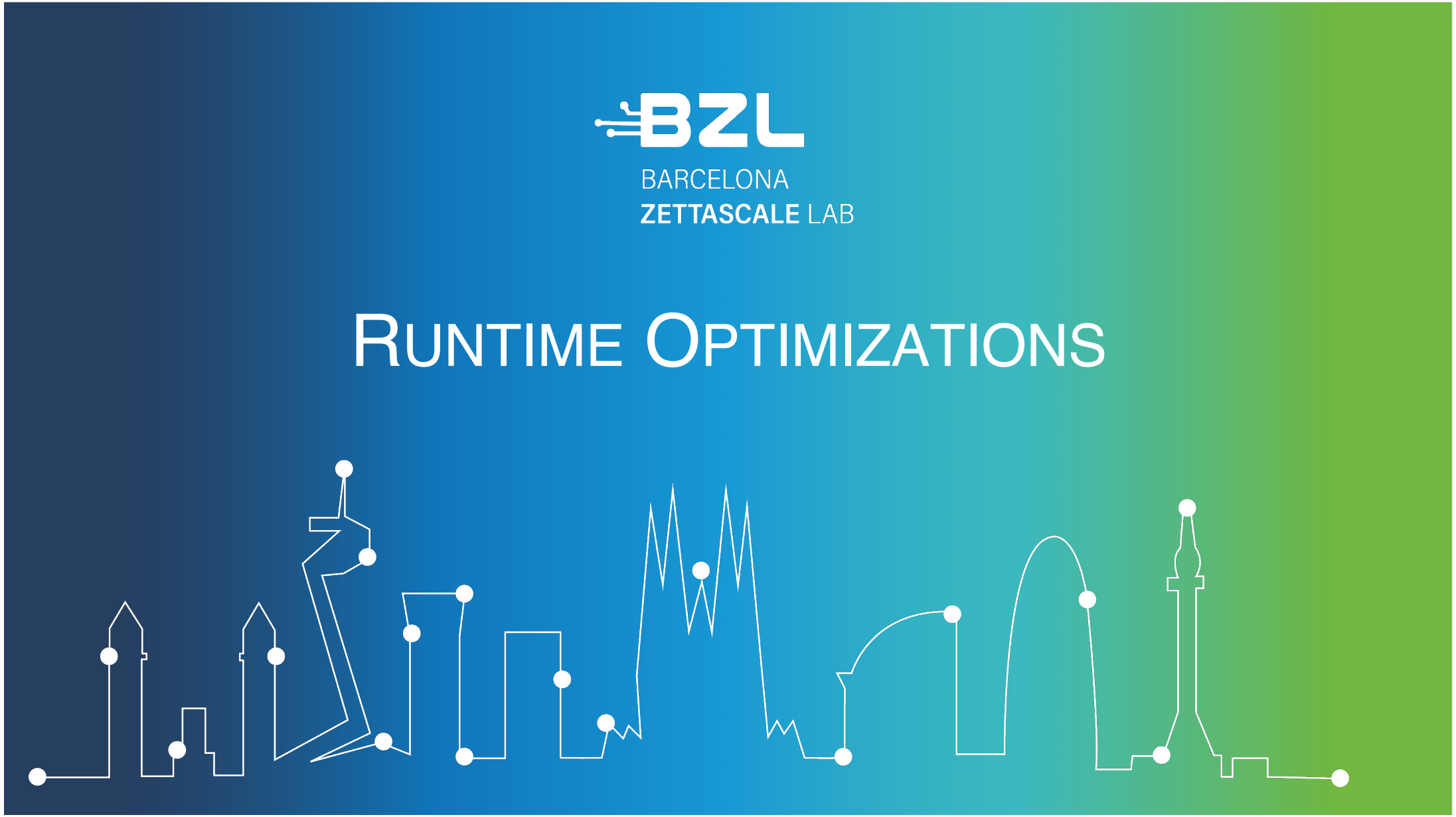


OmpSs-2@Cluster example: Nested tasks

```
1 #pragma oss taskloop depend(inout: a[i])
2 for (size_t i = 0; i < N*M; i++) {
3         a[i]++;
4     }
5 }
```

```
1 for (size_t i = 0; i < N*M; i+= M) {
2 #pragma oss task depend(weakinout: a[i;M])
3    for (size_t j = i; j < i+M; j++) {
4        #pragma oss task depend(inout: a[j])
5        a[i]++;
6     }
7 }
```

- **Taskloop loop iterations are automatically distributed by runtime**
  - In SMP mode: across cores
  - In OmpSs-2@Cluster mode: across nodes and cores
- **Weak accesses**
  - Region not directly accessed by the parent task
  - But links dependencies so that the data can be accessed by subtasks
- **Achilles heel: requires fragmented regions dependency system**
  - Either that or more complex multidependencies for parent tasks
  - We chose regions dependencies to help the programmer


BZL
BARCELONA
ZETTASCALE LAB
RUNTIME OPTIMIZATIONS

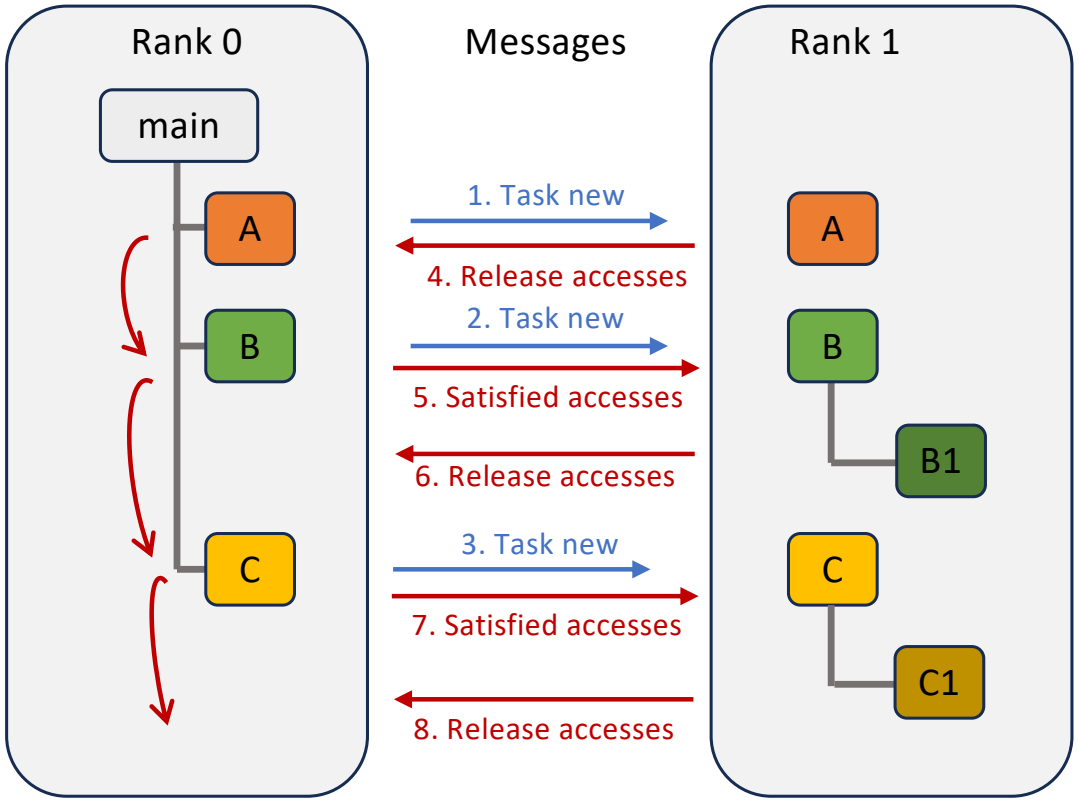
Runtime optimizations

- Seems simple, but took years…
- Sensitive to implementation details

Optimization	Description
WriteID	Avoid redundant data transfers via a versioning cache (centralized dependency tracking remembers only one valid node for correctness)
Leader Thread	Reduce latency / CPU contention using a dedicated thread for incoming MPI messages (e.g. newly offloaded tasks) and to process message completions
Namespace	Reduce overhead of centralized dependency tracking by delegating dependency tracking of consecutive tasks offloaded to the same rank
Message aggregation	Reduce MPI contention by coalescing control messages when multiple accesses become ready
Helpers	Parallelize message handling and runtime progress using multiple low-priority tasks

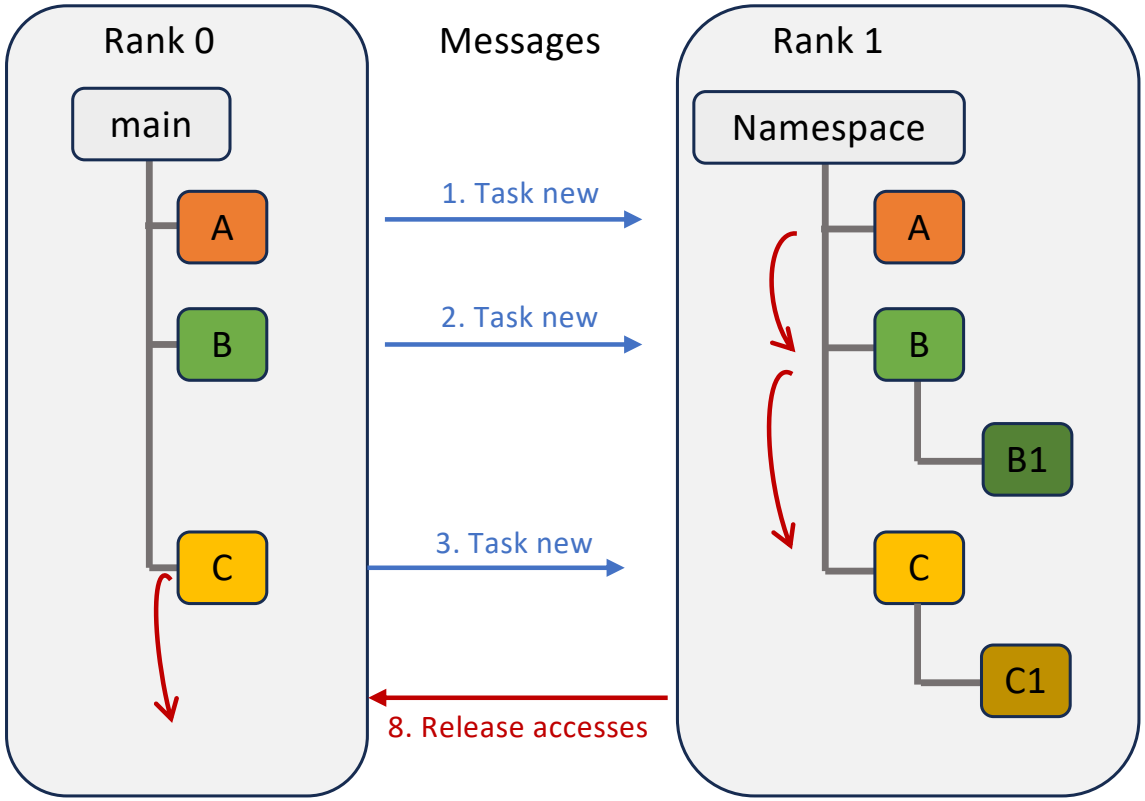
Without namespace optimization

```
1  #pragma oss task out(a) label("A")
2  { a = 1; }
3
4  #pragma oss task weakinout(a) label("B")
5  {
6      #pragma oss task inout(a) label("B1")
7      { a++; }
8  }
9
10 #pragma oss task weakinout(a) label("C")
11 {
12     #pragma oss task inout(a) label("C1")
13     { a++; }
14 }
```



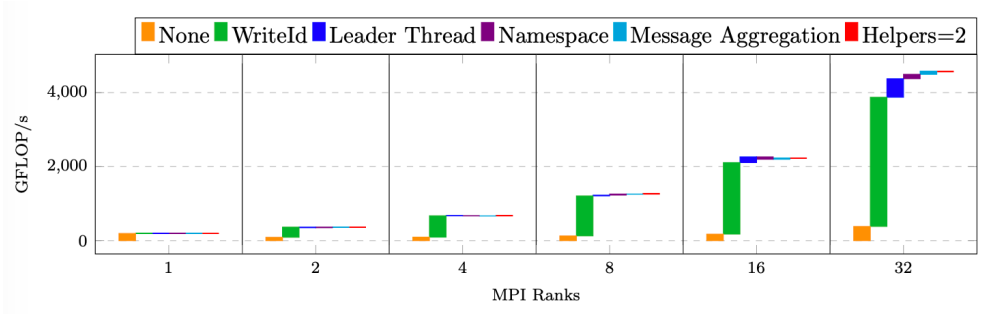
With namespace optimization

```
1  #pragma oss task out(a) label("A")
2  { a = 1; }
3
4  #pragma oss task weakinout(a) label("B")
5  {
6      #pragma oss task inout(a) label("B1")
7      { a++; }
8  }
9
10 #pragma oss task weakinout(a) label("C")
11 {
12     #pragma oss task inout(a) label("C1")
13     { a++; }
14 }
```

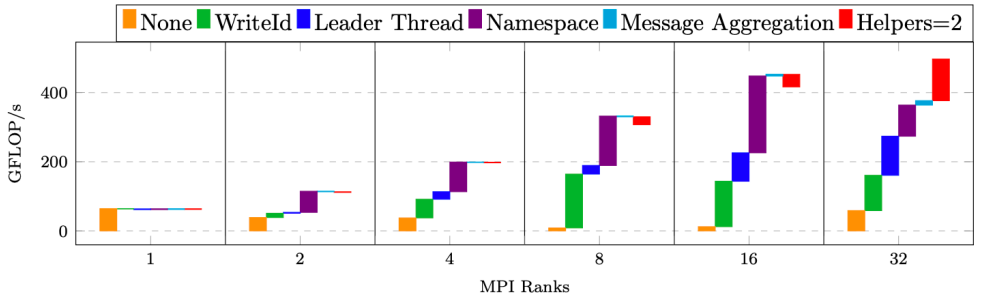


Lessons from runtime engineering

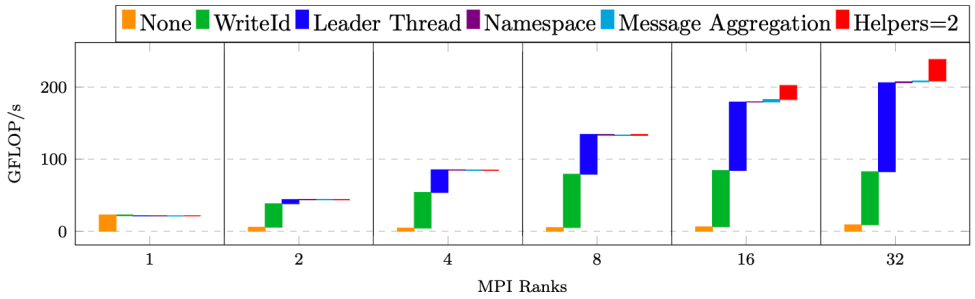
- Different benchmarks and different #nodes benefit from different optimizations (also depends on the system, MPI library, etc.)



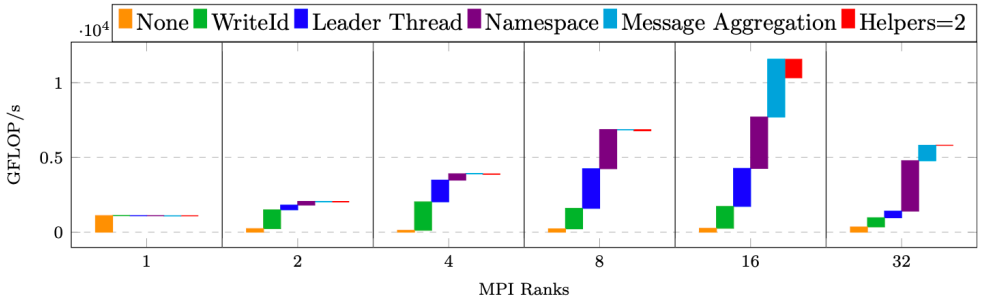
Matrix–matrix multiplication 16384×16384



Matrix–vector multiplication 32768×32768



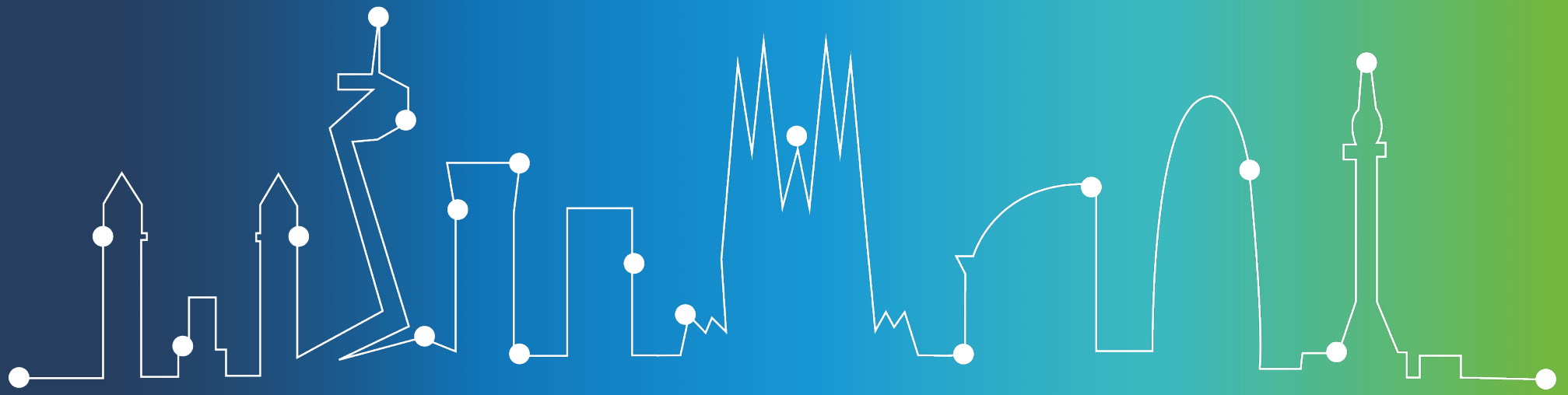
Jacobi benchmark 32768×32768



Cholesky factorization, 32768×32768

BZL
BARCELONA
ZETTASCALE LAB

DYNAMIC LOAD BALANCING (DLB)



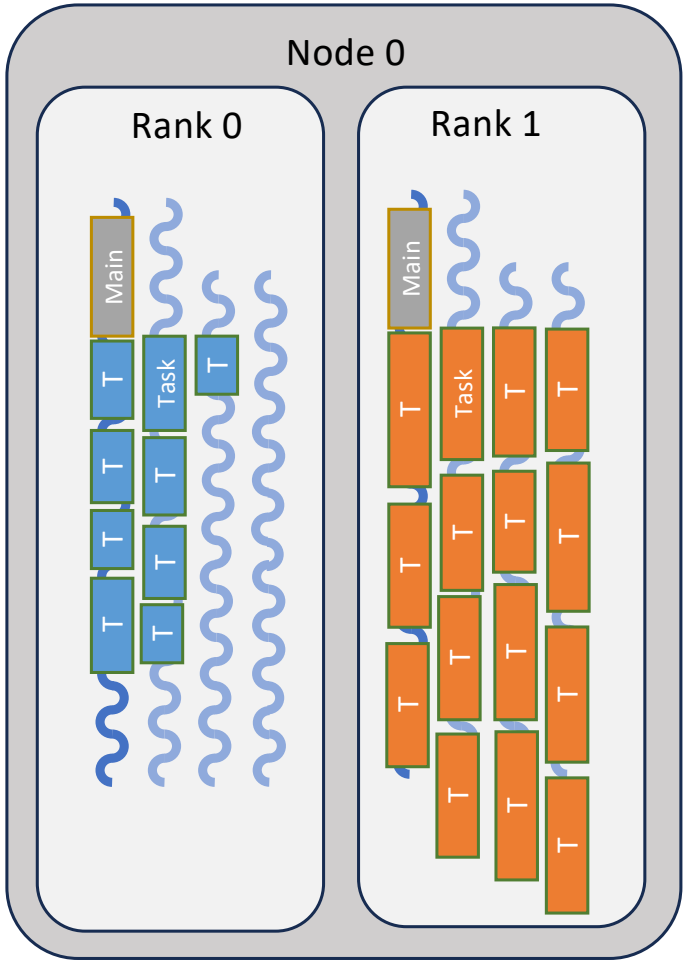
Dynamic load balancing: motivation

- **Load imbalance is a problem as old as parallel programming**
- **Most solutions are done in the application**
  - e.g. BT-MZ, discrete event simulation, Monte–Carlo
- **Mesh partitioning**
  - e.g. METIS
  - Need an accurate cost model
  - Static approaches cannot handle dynamic load imbalance
  - Dynamic approaches: not trivial when to repartition
- **Second level of parallelism**
  - e.g. BSC's DLB library
  - Compute resources can be redistributed among the processes
  - But current approaches are restricted to processes on the node

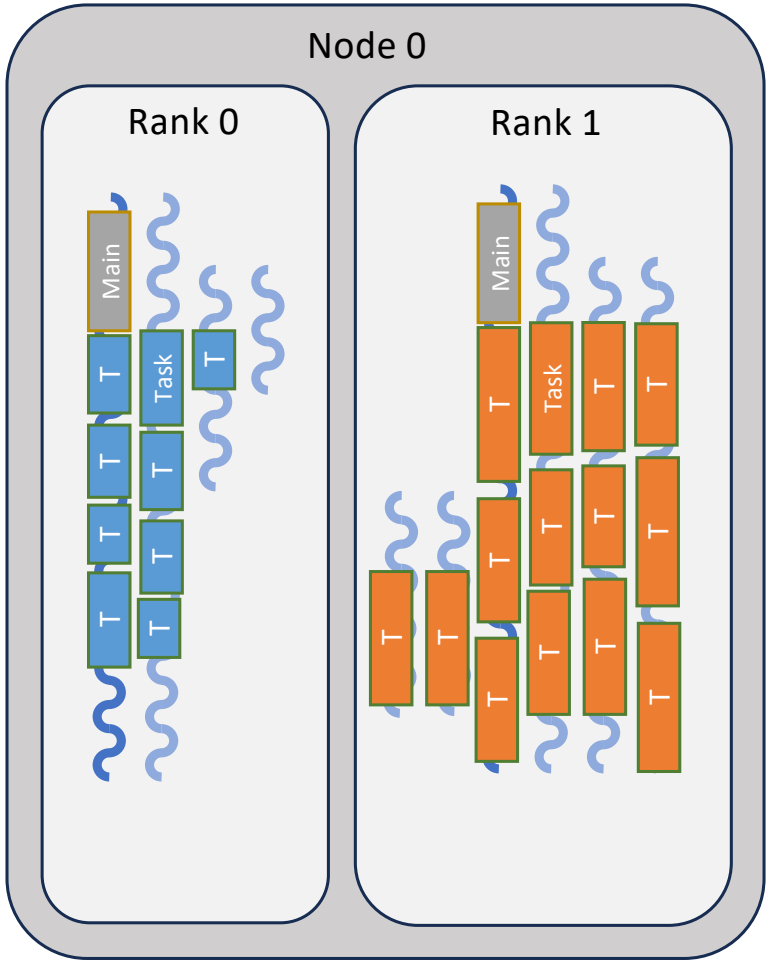
Jimmy Aguilar Mena, Omar Shaaban, Victor Lopez, Marta Garcia, Paul Carpenter, Eduard Ayguade, and Jesus Labarta. Transparent load balancing of MPI programs using OmpSs-2@Cluster and DLB. *ICPP2022*.

Dynamic load balancing with DLB*

Before: Without DLB



After: With DLB

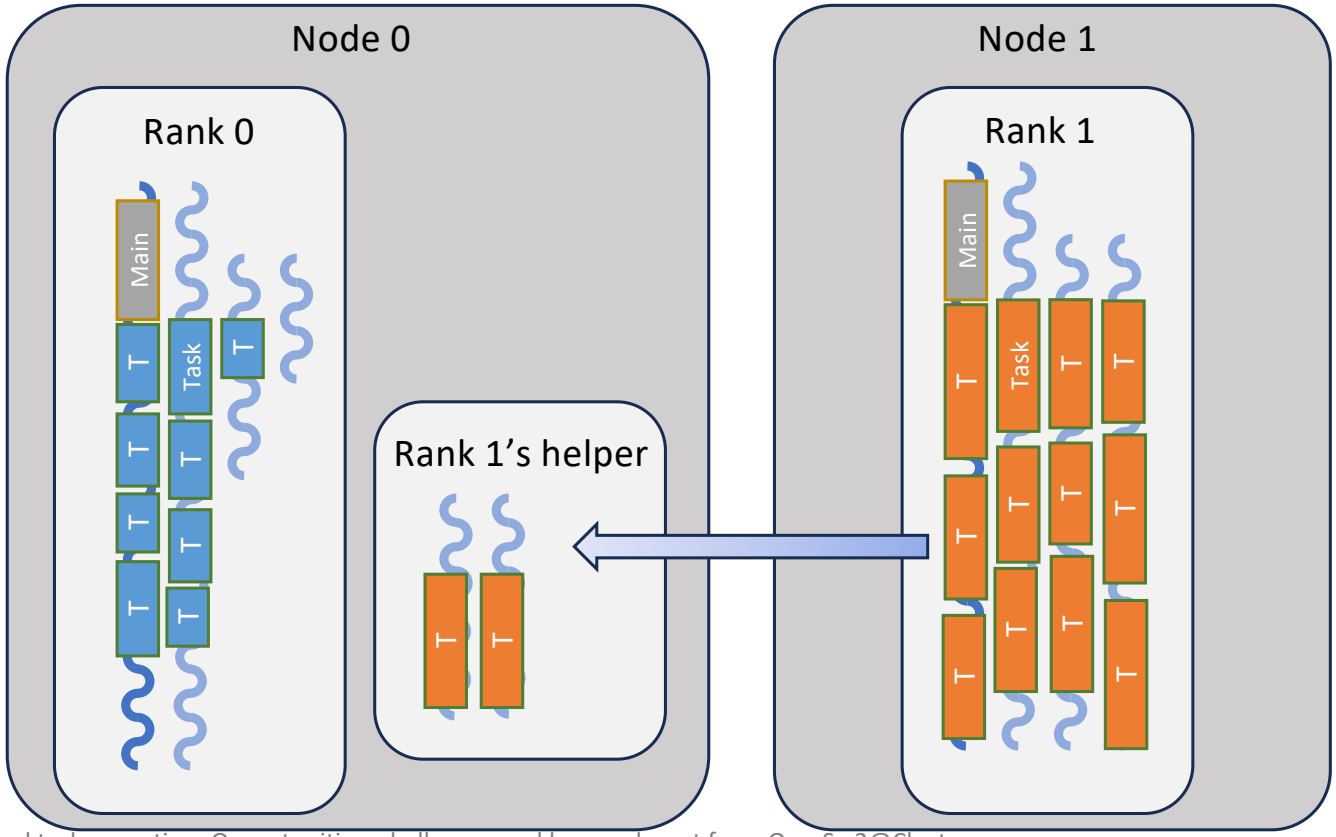


P. Carpenter, Distributed task execution: Opportunities, challenges and lessons learnt fror

* Marta Garcia, Julita Corbalan, and Jesus Labarta. LeWI: A Runtime Balancing Algorithm for Nested Parallelism. ICPP 2009.

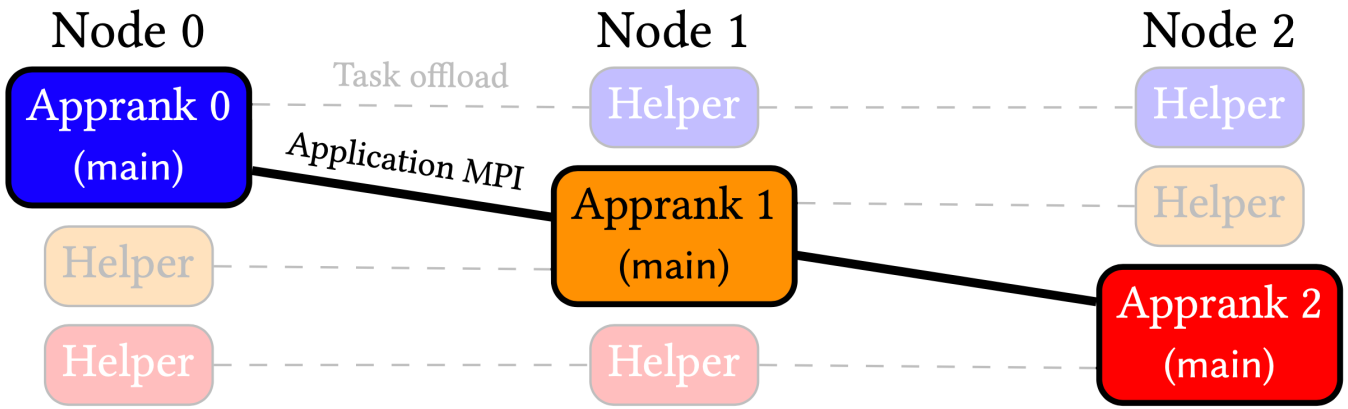
OmpSs-2@Cluster addresses inter-node load imbalance

- Handles greater level of imbalance and intra-node correlation in imbalance



OmpSs-2@Cluster + MPI: Implementation approach (1/2)

- **Small number of helper processes are launched on each node**
  - Sparse expander graph: few helpers but high connectivity to spread the work
  - If load is balanced, helpers remain inactive



MPI + OmpSs-2@Cluster: Implementation approach (2/2)

- **But if load imbalance, helpers will execute offloaded tasks**
  - Helpers are full Nanos6 runtime instances but invisible to the application
  - Separate processes: isolated address space from other appranks on same node
  - DLB assigns cores among the processes on each node

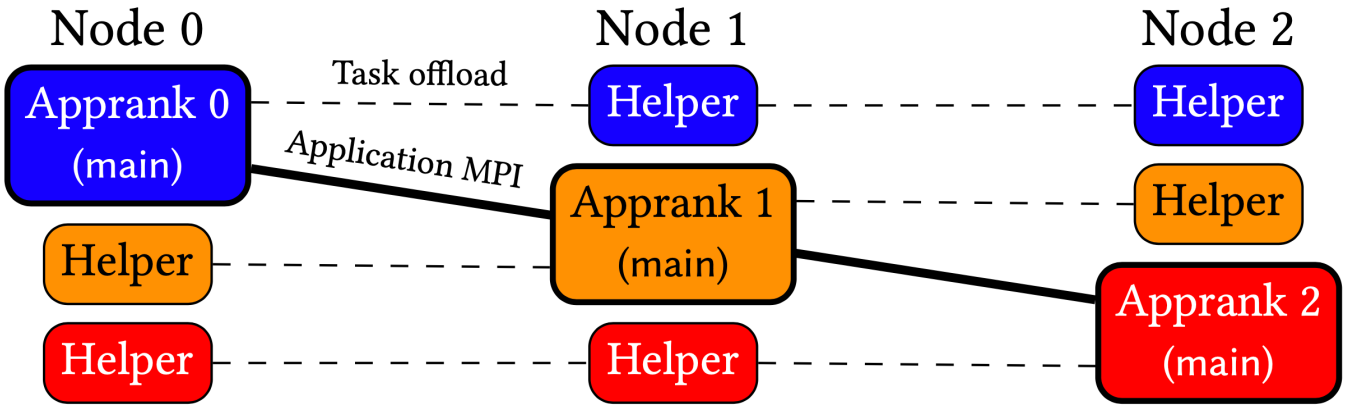
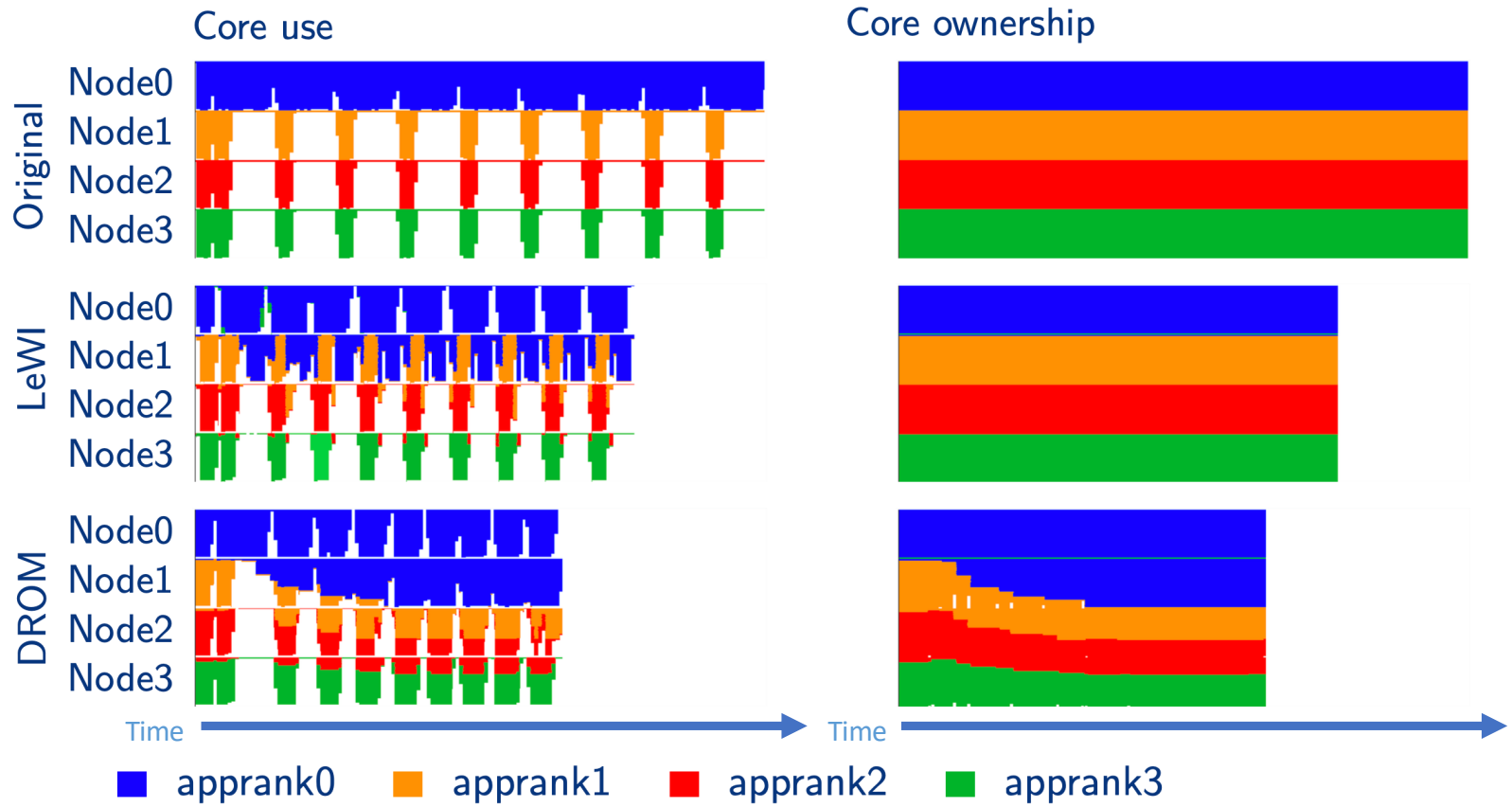
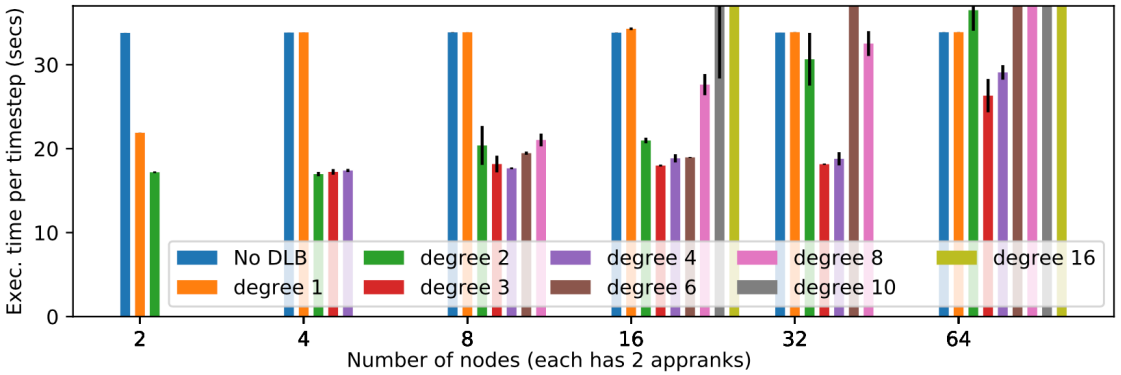


Illustration of load balancing using BSC's Alya MicroPP

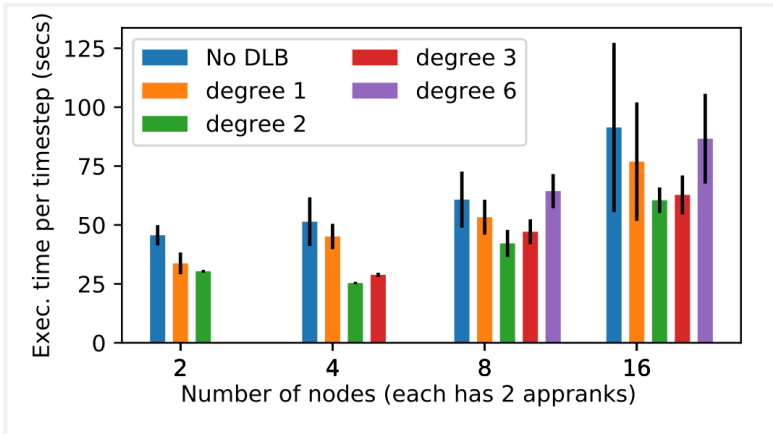


Results (weak scaling)

- **Unbalanced application (MicroPP)**
  - Reduction in time to solution by 47% on 32 nodes



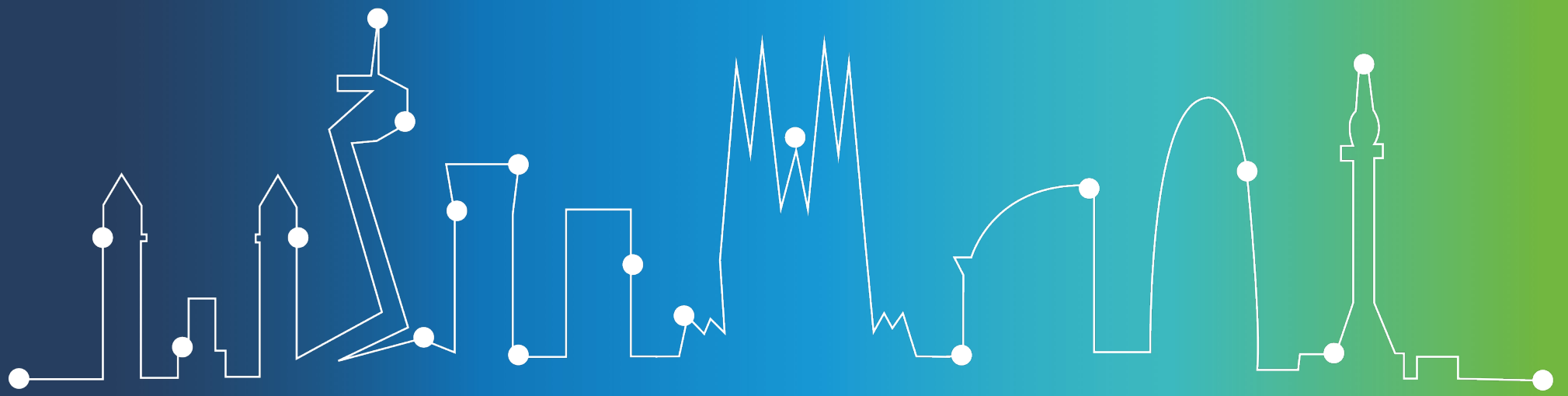
- **System with one slow node (N-body on Nord 3)**
  - One node at 1.8 GHz vs 3.0 GHz
  - Orthogonal Recursive Bisection assumes equal performance nodes
  - 36% reduction in time to solution



BZL

BARCELONA
ZETTASCALE LAB

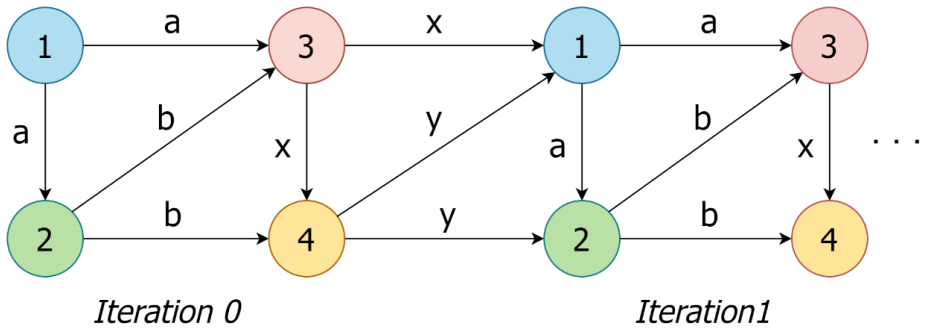
DISTRIBUTED TASKITER



Iterative Applications

- **Many HPC applications create the same DAG on each timestep / iteration**
  - e.g., iterative methods or multi-step simulation.

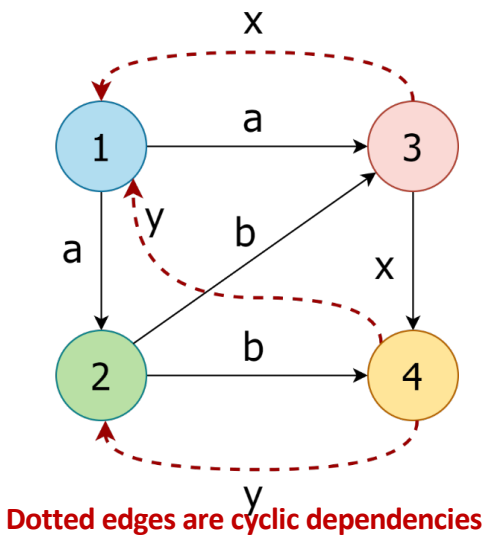
```
1  for (int it=0; i<NUM_ITERATIONS;it++) {
2
3     #pragma oss task in(x,y) out(a) label("1")
4     { ... }
5
6     #pragma oss task in(a,y) out(b) label("2")
7     { ... }
8
9     #pragma oss task in(a,b) out(x) label("3")
10    { ... }
11
12    #pragma oss task in(x,b) out(y) label("4")
13    { ... }
14 }
```



Taskiter construct*

- Proposed in 2023 for iterative OpenMP/OmpSs-2 applications on SMP
  - Task creation and calculation of dependencies only done once
  - Enables scheduling decisions to be precalculated
- **Annotate loops where:**
  - Each iteration creates same top-level tasks and accesses on each iteration
  - Program remains valid if code in loop body but outside tasks is executed once

```
1  #pragma oss taskiter
2  for (int it=0; i<NUM_ITERATIONS;it++) {
3
4     #pragma oss task in(x,y) out(a) label("1")
5     { ... }
6
7     #pragma oss task in(a,y) out(b) label("2")
8     { ... }
9
10    #pragma oss task in(a,b) out(x) label("3")
11    { ... }
12
13    #pragma oss task in(x,b) out(y) label("4")
14    { ... }
15 }
```



Dotted edges are cyclic dependencies

* David Álvarez and Vicenç Beltran. 2023. Optimizing Iterative Data-flow Scientific Applications using Directed Cyclic Graphs. IEEE access (2023).

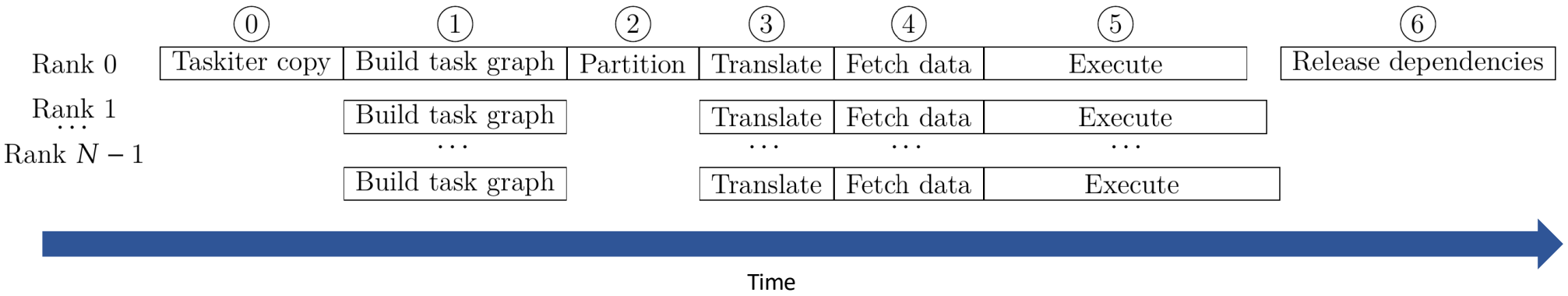
Distributed taskiter

- **Taskiter is perfect for distributed execution**
- **Amortize sequential task creation bottleneck**
- **Accesses fully fragmented**
  - Translate to simple DAG rather than fragmented region accesses
- **Precompute all MPI data transfers**
  - Execute in same way as MPI + OmpSs-2 with asynchronous tasks
  - Only approach that completely avoids centralized dependency tracking

- HiPEAC Conference technical program this afternoon
  - **Omar Shaaban**, Juliette Fournis d'Albiat, Isabel Piedrahita, Vicenç Beltran, Paul Carpenter, Eduard Ayguade and Jesus Labarta. Leveraging Iterative Applications to Improve the Scalability of Task-Based Programming Models on Distributed Systems. *ACM Transactions on Architecture and Code Optimization.*

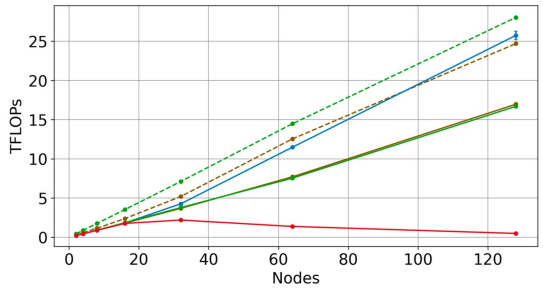
Overview of execution Flow

- Rank 0 distributes taskiter body to all other ranks
- All ranks create full dependency graph concurrently
- Translate into local dependency graph (in simplified form)

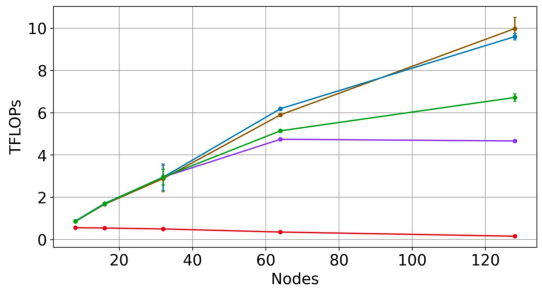


Results (1): Distributed taskiter matches state-of-the-art

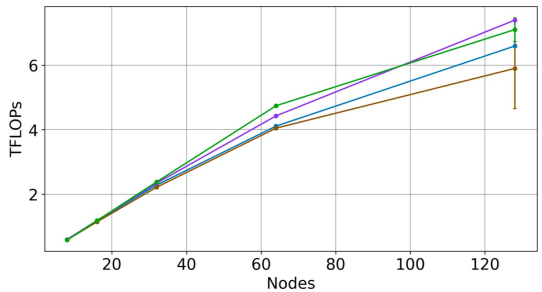
Matrix–vector multiplication 128K× 128K



Jacobi iteration 128K×128K



Heat equation (Jacobi iteration) 128K×128K



Distributed Taskiter

MPI + OpenMP (fork–join)

MPI + OmpSs-2 (fork–join)

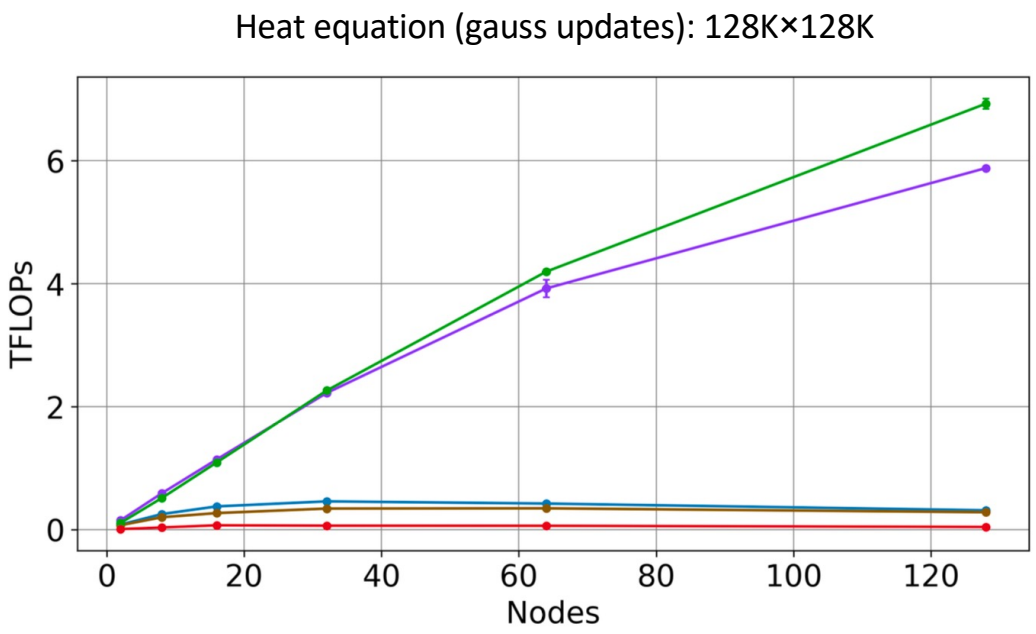
TAMPI + OmpSs-2

- Range due to different runtimes and scheduling policies
  - Dotted: immediate successor, Solid line: by iteration
- Distributed taskiter similarwhen comparing like-to-like

Baseline OmpSs-2@Cluster

- Poor scaling with these problem sizes

Results (2): Benefits from asynchronous tasking



Distributed Taskiter

TAMPI + OmpSs-2

- Both benefit from asynchronous tasks and communication

MPI + OpenMP (fork–join)

MPI + OmpSs-2 (fork–join)

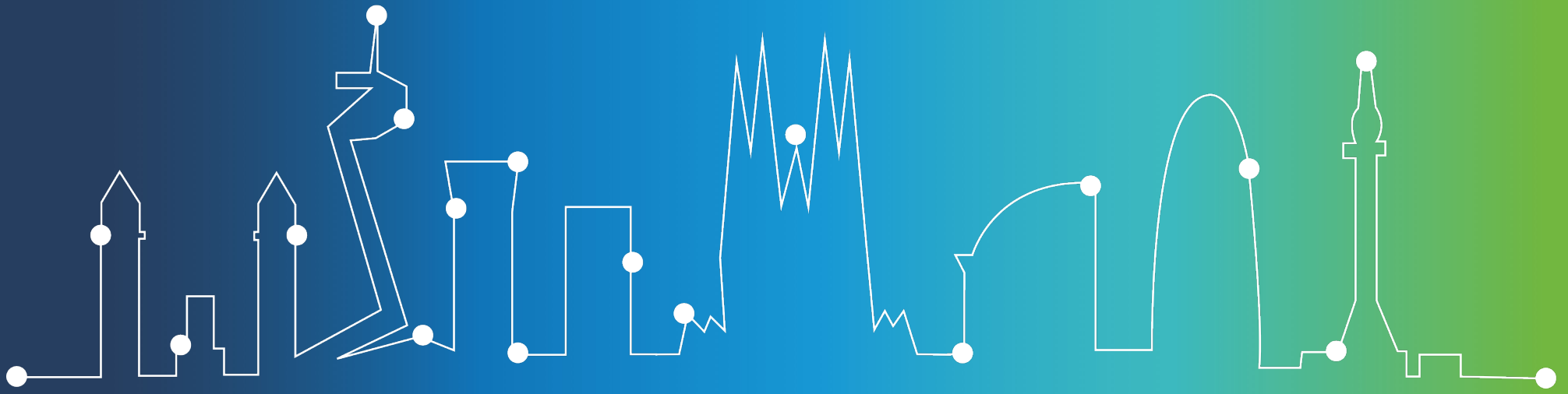
- Fork-join parallelism limits parallelism

Baseline OmpSs-2@Cluster

- Poor scaling with these problem sizes

BZL
BARCELONA
ZETTASCALE LAB

CONCLUSIONS



Conclusions

- Distributed tasking is expensive – but possible
- It works best when applied selectively (e.g. for load balancing)
- Structure (e.g. iterative) changes everything

- HiPEAC Conference technical program this afternoon
  - **Omar Shaaban**, Juliette Fournis d'Albiat, Isabel Piedrahita, Vicenç Beltran, Paul Carpenter, Eduard Ayguade and Jesus Labarta. Leveraging Iterative Applications to Improve the Scalability of Task-Based Programming Models on Distributed Systems. *ACM Transactions on Architecture and Code Optimization.*

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

BARCELONA **ZETTASCALE** LAB

Financiado por la Unión Europea
NextGenerationEU

MINISTERIO
PARA LA TRANSFORMACIÓN DIGITAL
Y DE LA FUNCIÓN PÚBLICA

SECRETARÍA DE ESTADO
DE TELECOMUNICACIONES
E INFRAESTRUCTURAS DIGITALES

Plan de Recuperación, Transformación y Resiliencia

Este proyecto se impulsa por el Ministerio para la Transformación Digital y de la Función Pública, en el marco del Plan de Recuperación, Transformación y Resiliencia - Financiado por la Unión Europea – NextGenerationEU