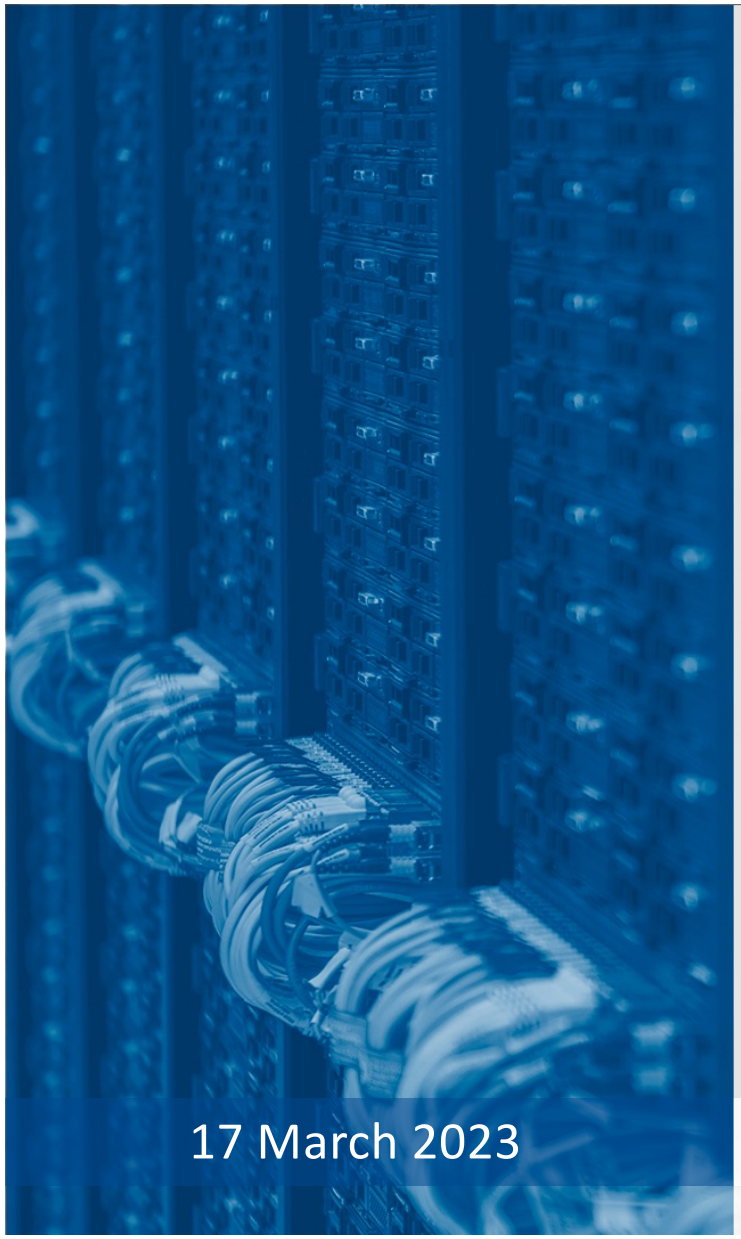


Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Core- and node-level malleability using OmpSs-2@Cluster

Paul Carpenter

Jimmy Aguilar Mena, Omar Shaaban, Isabel Piedrahita, Juliette Fournis, Vicenç Beltran

17 March 2023

DEEP-SEA seminar

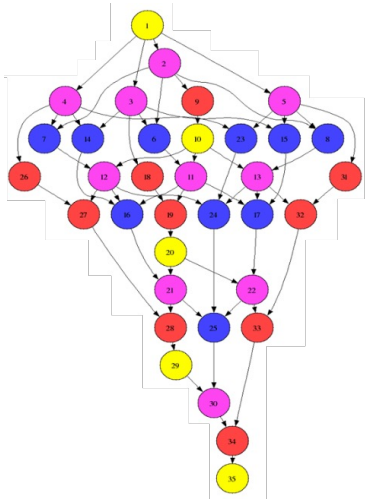
Outline

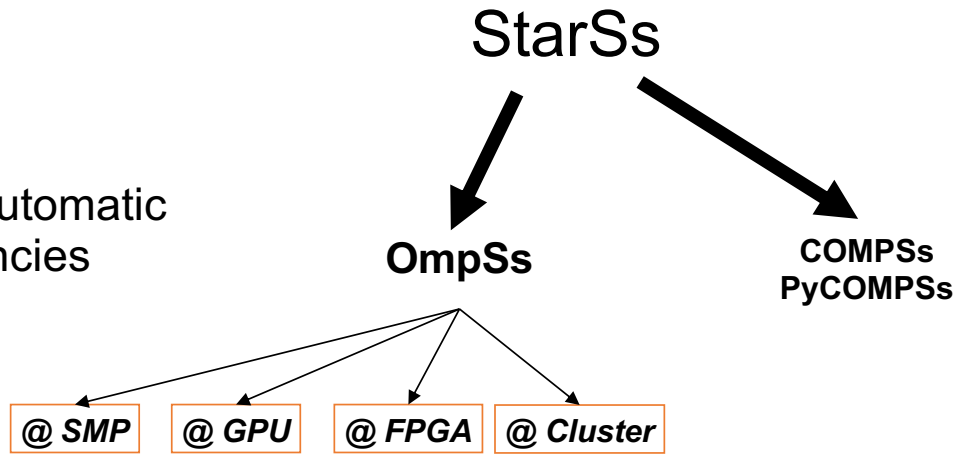
- OmpSs and OmpSs-2
- Pure OmpSs-2@Cluster
- MPI + OmpSs: Dynamic load balancing (core-level malleability)
- Node-level malleability
- Conclusion

OmpSs and OmpSs-2

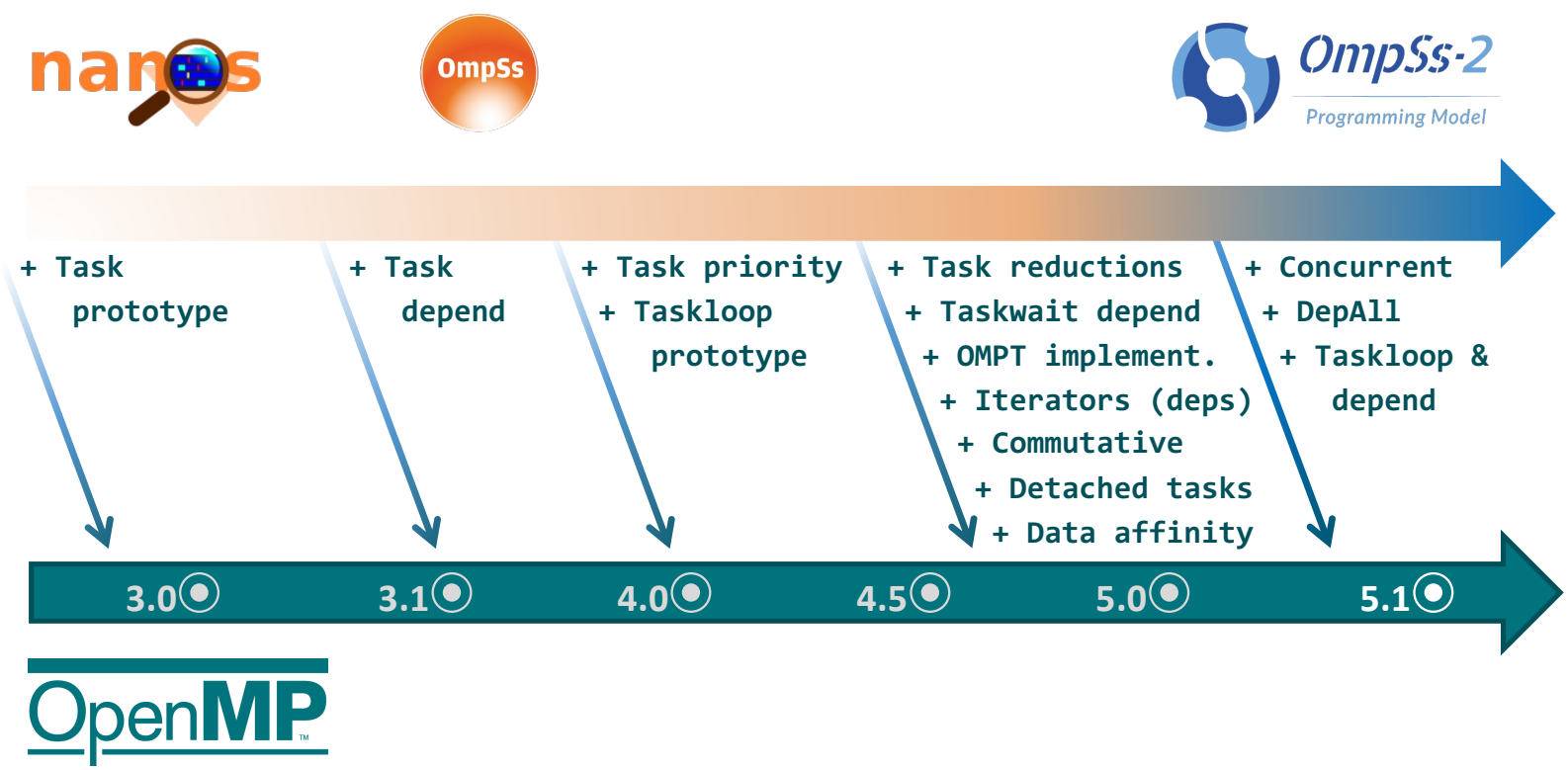
StarSs family of parallel programming models

- StarSs key concepts
  - Sequential task-based program
  - Tasks with accesses in a single address/name space
  - Happens to execute in parallel: automatic runtime computation of dependencies



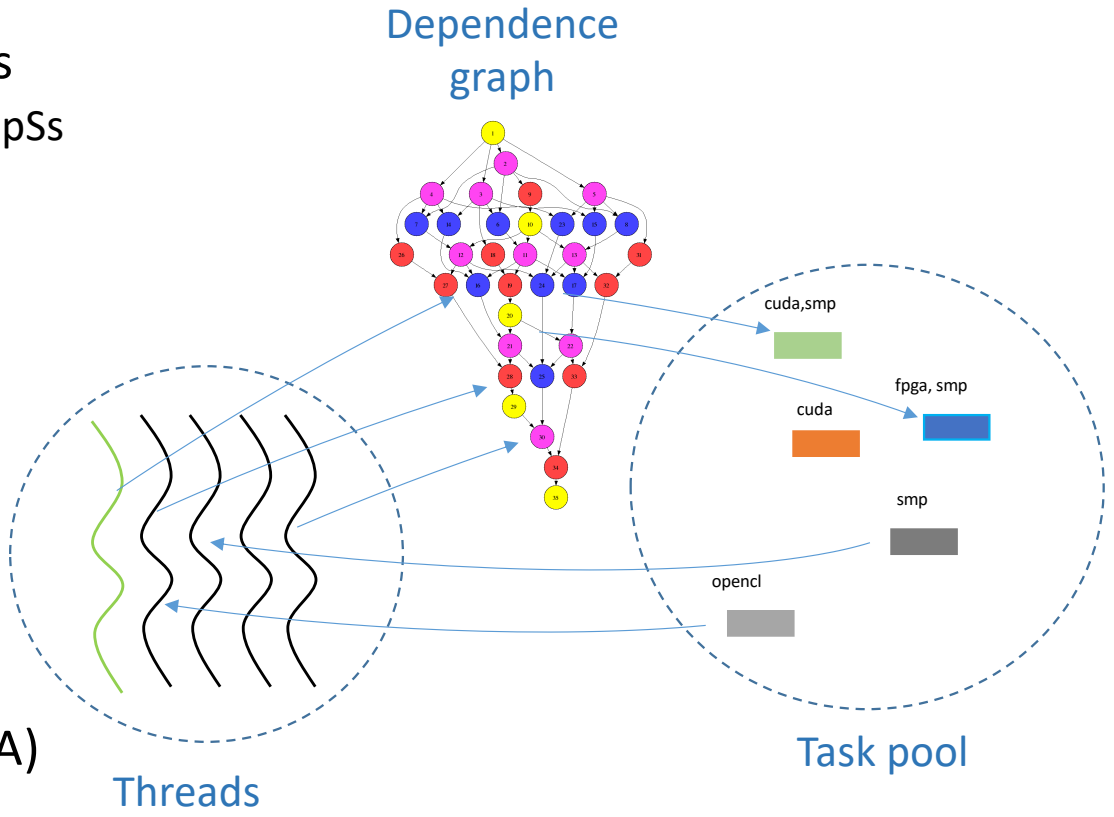


OmpSs: a forerunner for OpenMP Tasking



Main difference between OmpSs and OpenMP: execution model

- Both models have tasks with annotations
  - Generally small changes for OpenMP => OmpSs
  - BSC can help
- Thread pool model, not fork–join
- Kernel threads created on start-up
- All threads get work from a task pool
  - One thread executes main on an SMP core
  - Tasks generate subtasks
  - Work is labeled with possible “targets”
  - Single or multiple work-queues
- One representative (OpenCL/CUDA/FPGA) per device/accelerator



Data accesses: single mechanism to describe concurrency and data

- **Concurrency**: Runtime computes task dependencies
- **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);
      }
   }
}
```

Pure OmpSs-2@Cluster

OmpSs-2@Cluster

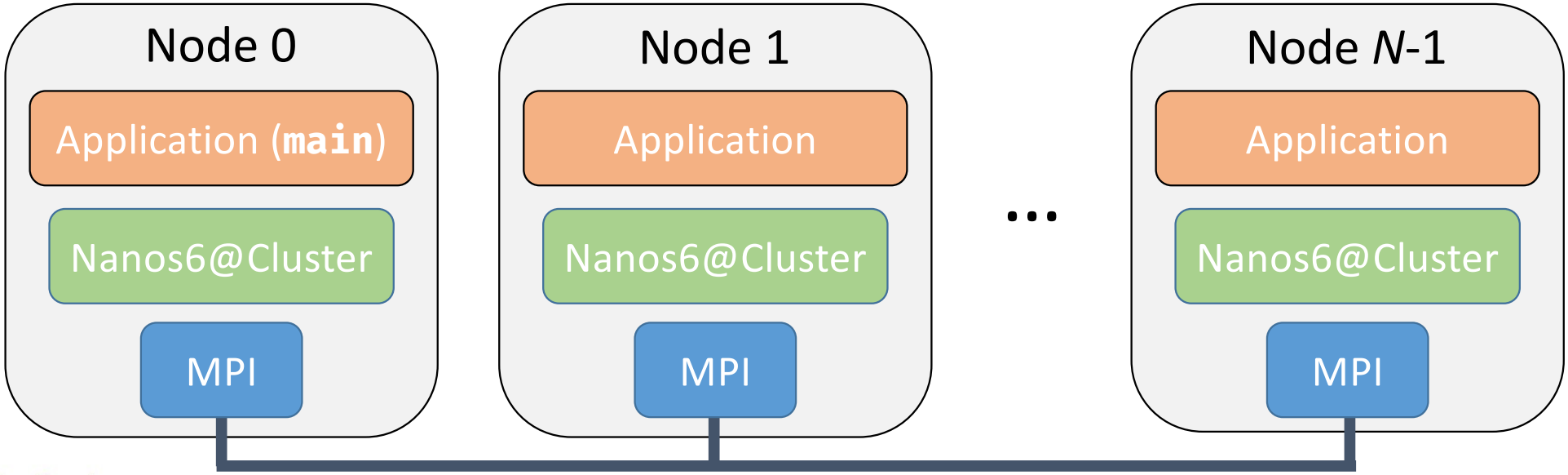
- Extends OmpSs-2 tasking model to multiple nodes
  - Tasks may be transparently offloaded to other nodes (MPI processes)
  - Task dependencies and data transfers are managed by the runtime system
  - Scheduling is based on locality and load balancing
  - Communication is done via MPI

- Objectives
  1. Alternative to MPI for small scale (up to about 8 or 16 nodes)
  2. Improved load balancing for MPI + OmpSs-2 programs
  3. Node-level malleability

[*] Jimmy Aguilar Mena, Omar Shaaban, Vicenç Beltran, Paul Carpenter, Eduard Ayguade, and Jesus Labarta. OmpSs-2@Cluster: Distributed memory execution of nested OpenMP-style tasks. *European Conference on Parallel Processing, Euro-Par 2022*

OmpSs-2@Cluster architecture

- Program is compiled exactly the same as OmpSs-2 on SMP (compatible run time API)
- 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 node (otherwise symmetrical across nodes)



OmpSs-2@Cluster example: Optimization for one task per node

```
1 const size_t numNodes = nanos6_get_num_cluster_nodes();
2 const size_t elemsPerNode = N / numNodes;
3 for (size_t i = 0; i < N; i += elemsPerNode) {
4     int nodeid = i / elemsPerNode;
5     #pragma oss taskfor depend(inout: a[i;elemsPerNode]) node(nodeid)
6      for (size_t j = i; j < i+elemsPerNode; j ++) {
7           a[j]++;
8     }
9 }
```

- One task is offloaded per node
- The offloaded task is a taskfor, which occupies all cores

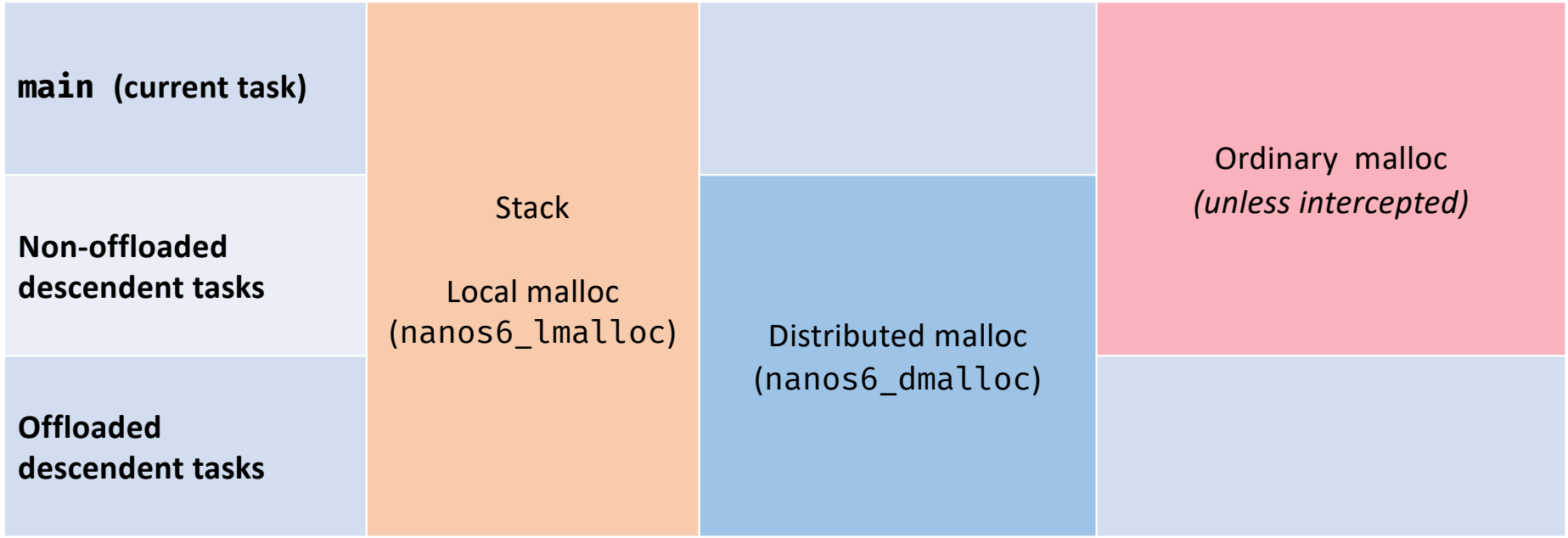
OmpSs-2@Cluster example: Distributed taskloop

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

- Taskloop is part of OpenMP 4.0
  - Proposed by BSC based on earlier work on OmpSs-2
  - Similar to taskfor, but accesses can be function of induction variable
- Taskloop loop iterations are automatically distributed by runtime
  - In SMP mode: across cores
  - In OmpSs-2@Cluster mode: across nodes and cores
- In this form, the runtime has flexibility to optimize
  - Load balance
  - Dynamic concurrency throttling
  - Malleability

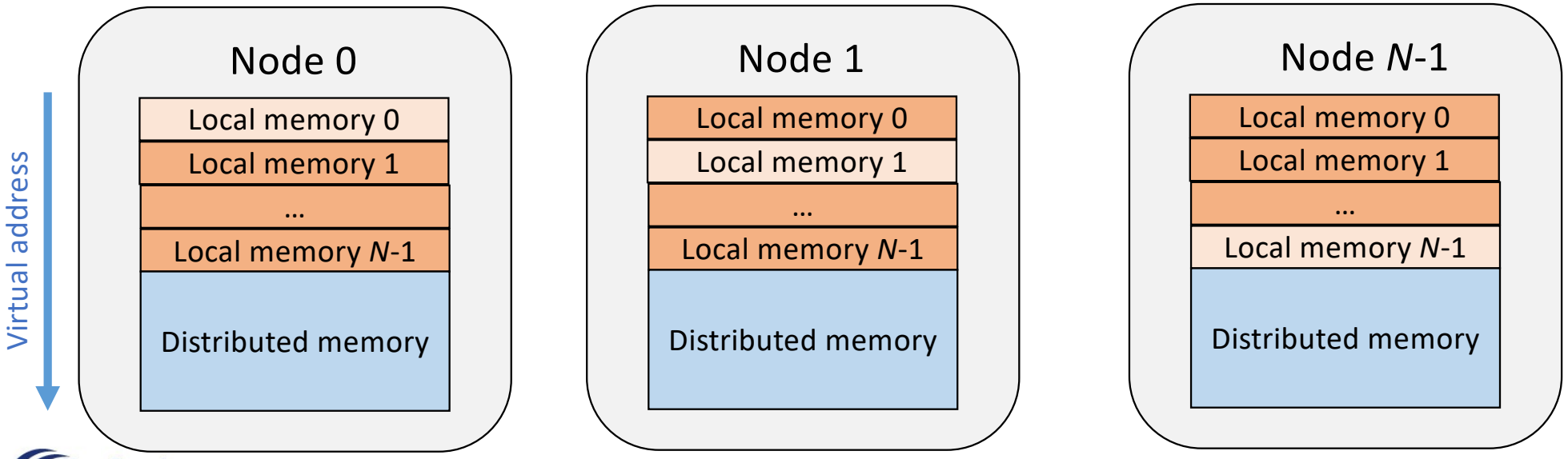
Address space and data management: programmer's view

- Data allocated on any node can be used by task on any node
- Allocation of data provides information to the runtime on the kind of data



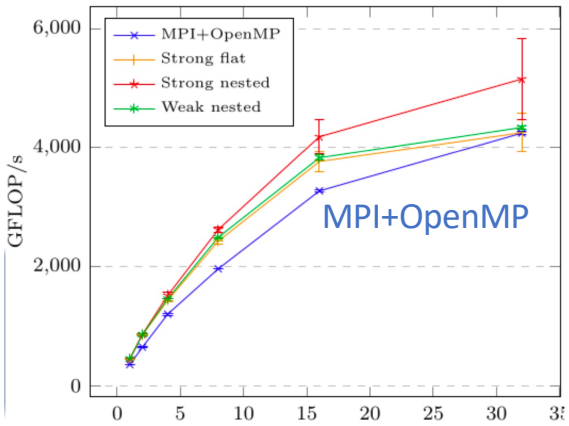
Address space and data management: implementation

- Runtimes coordinate to locate and mmap a common region of virtual memory
  - Modern CPUs have 48 bits of virtual memory, which is more than sufficient
- Local memory is allocated/freed by the current node without synchronization
- Distributed memory is allocated/freed centrally (by first node)

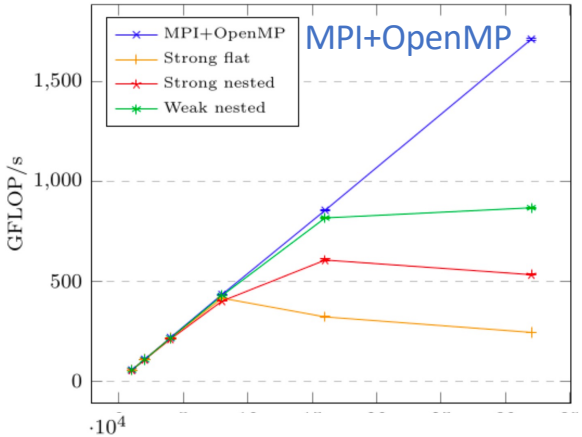


Results (strong scaling)

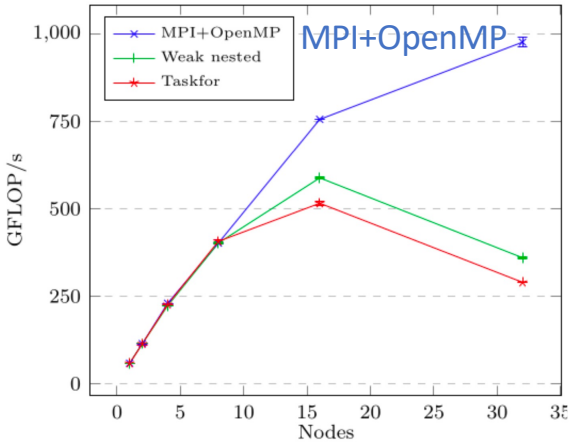
matmul:
Matrix–matrix multiplication
32k matrix



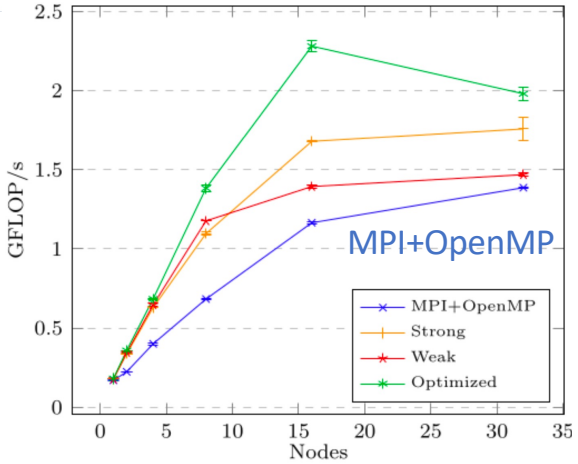
matvec:
Matrix–vector multiplication
64k matrix



jacobi:
Iterative Jacobi solver
64k matrix



choleksy:
Cholesky factorization
64k matrix



- Competitive with MPI for matmul
- Scales to about 8 nodes for matvec and jacobi
- Better than MPI+OpenMP (without TAMPI) for cholesky, due to irregular dataflow execution

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

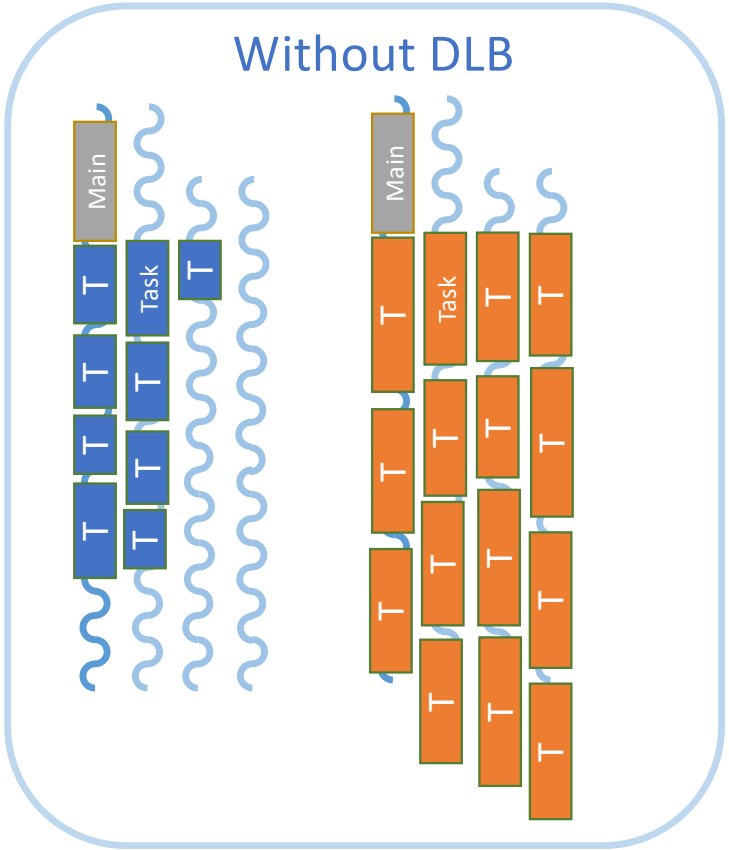
OmpSs-2@Cluster load balancing

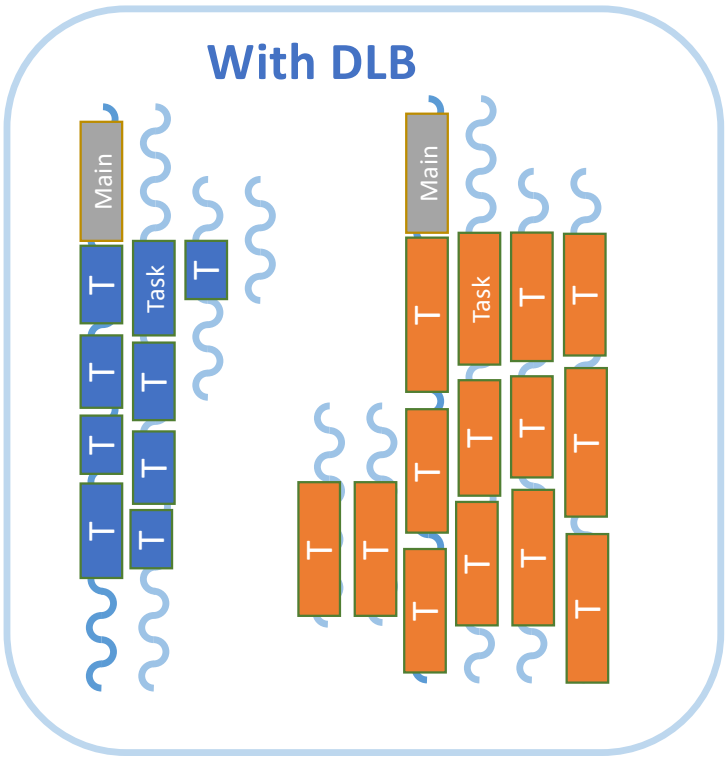
Dynamic load balancing: motivation

- 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*.

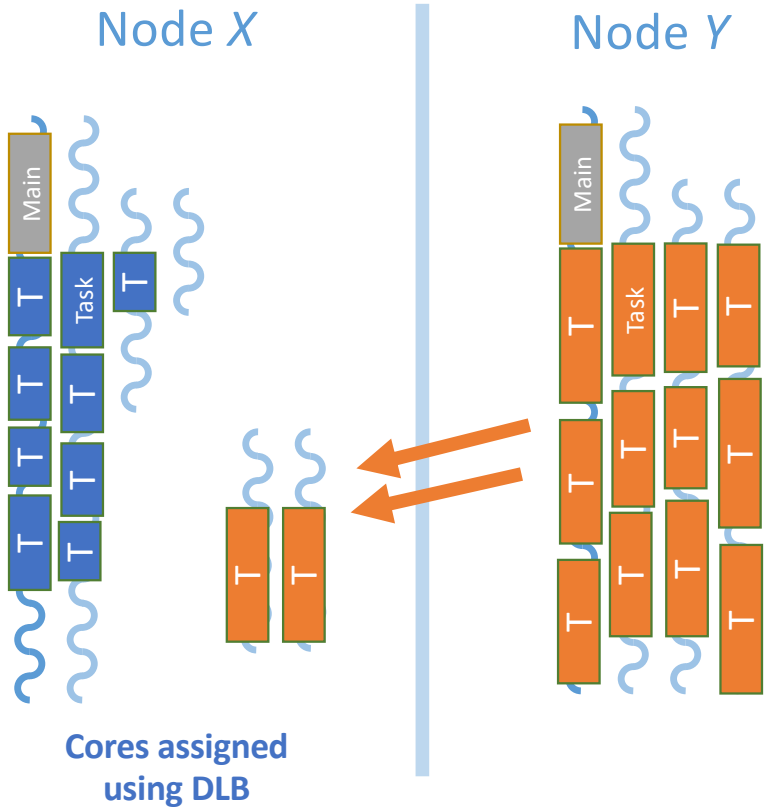
Dynamic load balancing with OpenMP/OmpSs and DLB





Marta Garcia, Julita Corbalan, and Jesus Labarta. LeWI: A Runtime Balancing Algorithm for Nested Parallelism. International Conference on Parallel Processing, 2009. ICPP '09.

OmpSs-2@Cluster addresses load imbalance on different nodes

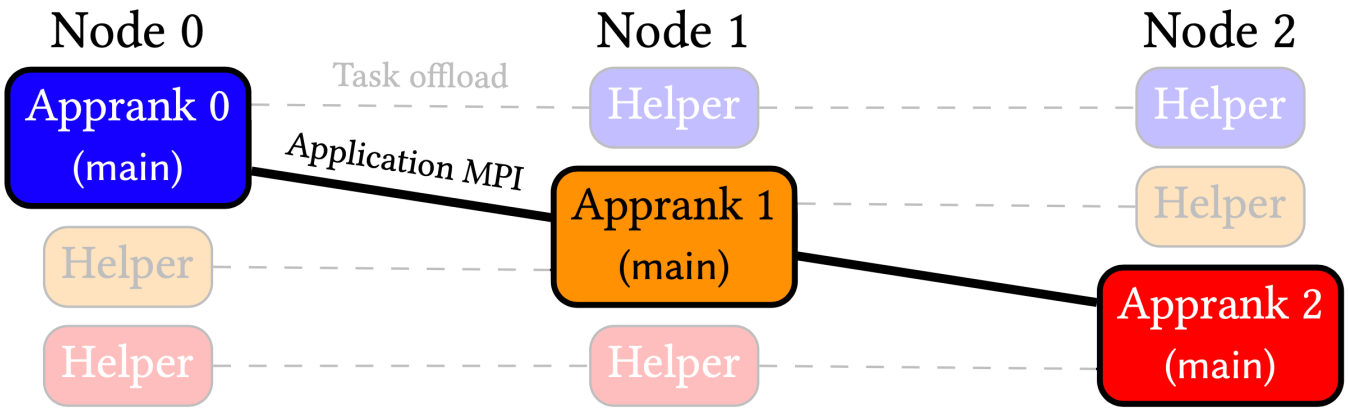


Addresses:

- Greater level of load imbalance
- One or few application ranks per node
- Correlation in load imbalance of MPI ranks on a node

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
  - Separate processes: isolated address space from other appranks on same node
  - DLB assigns cores among the processes on each node

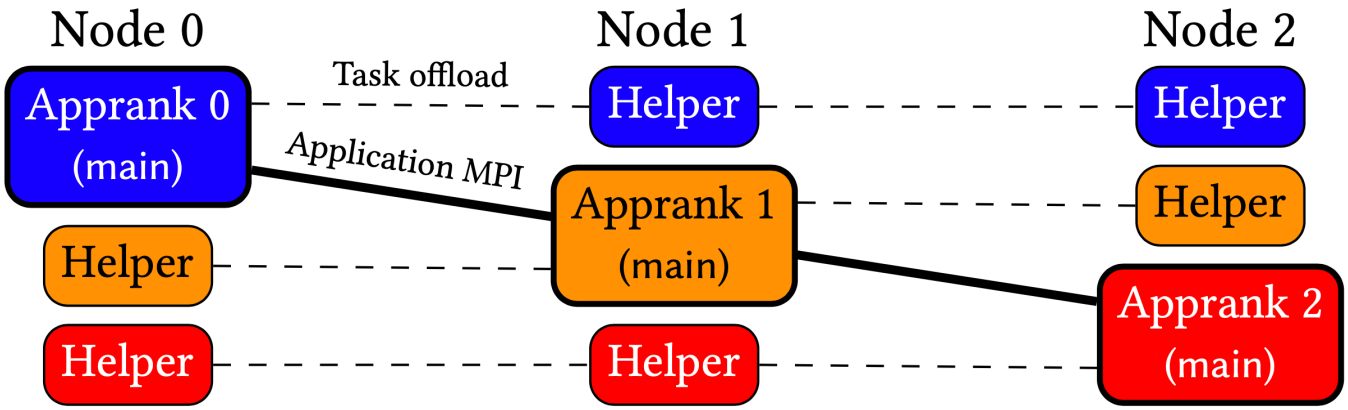
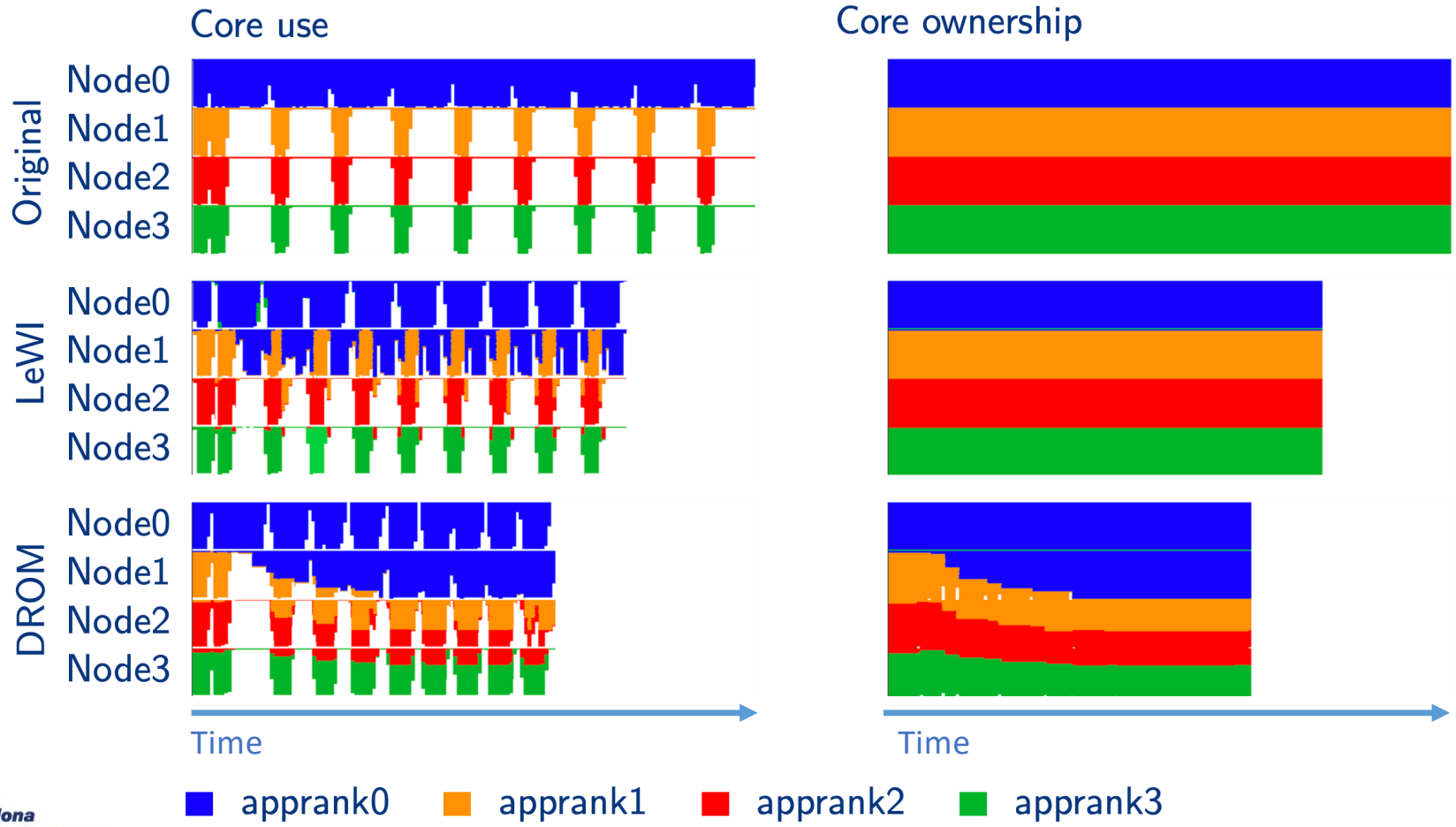


Illustration of load balancing using BSC's Alya MicroPP

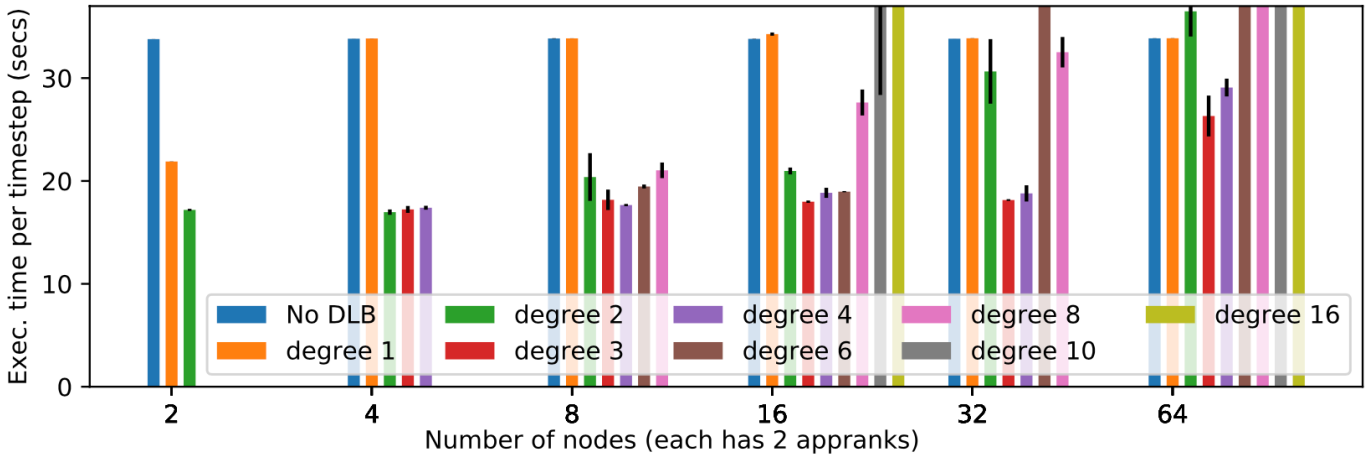


Results (weak scaling)

Unbalanced application

MicroPP

=> Reduction in time to solution by 49% (4 nodes) and 47% (32 nodes)

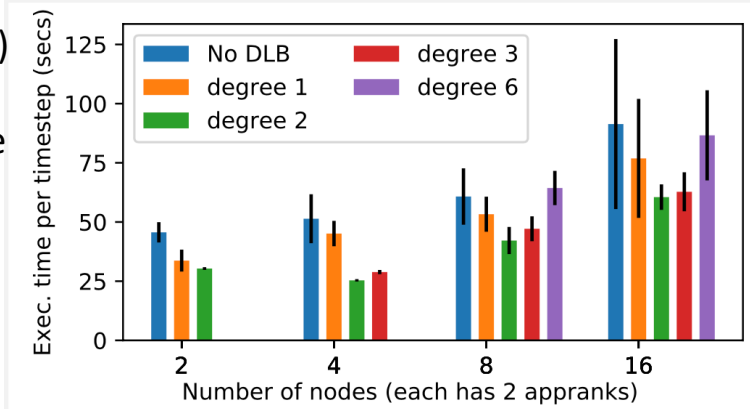


System with one slow node

N-body on Nord 3 (1.8 GHz vs 3.0 GHz)

Orthogonal Recursive Bisection balances load assuming constant node performance

=> 36% reduction in time to solution compared with baseline (using Orthogonal Recursive Bisection)



OmpSs-2@Cluster node-level malleability

Motivation

- Malleability: improve system throughput and support varying resource requirements
- Difficult to modify MPI programs to support malleability (data redistribution)
  - Libraries like DMRlib can help… (Antonio Peña + Sergio Iserte)
- OmpSs-2@Cluster simplifies malleability for the application
  - Few (or no) changes to program (if already using tasks)
  - Data redistribution transparently handled by runtime system

Programmer's interface for malleability

- Who makes the decision?
  - **Runtime controlled:** Runtime measures resource utilization, negotiates with job scheduler, decides when to initiate or accept malleability operations, and decides how many nodes to add/remove
  - **User controlled:** Application decides when to initiate or accept malleability operation and how many nodes to add/remove; runtime (only) implements user's decision

- When is the decision applied?
  - **Synchronous:** At defined points in the sequential code
  - **Asynchronous:** At any instant during the execution of the dependency graph, which may not correspond to a point in the sequential code

- We currently have a **User controlled Synchronous** model
  - But will investigate the other approaches

Programmer's interface for malleability: Synchronous with taskwait

- Taskwait stops task execution before malleability operation
- User controlled variant has API function
  - Taskwait throttles execution, otherwise may create all tasks, then it would be too late
  - Blocks to allocate resources from Slurm (timeout argument)

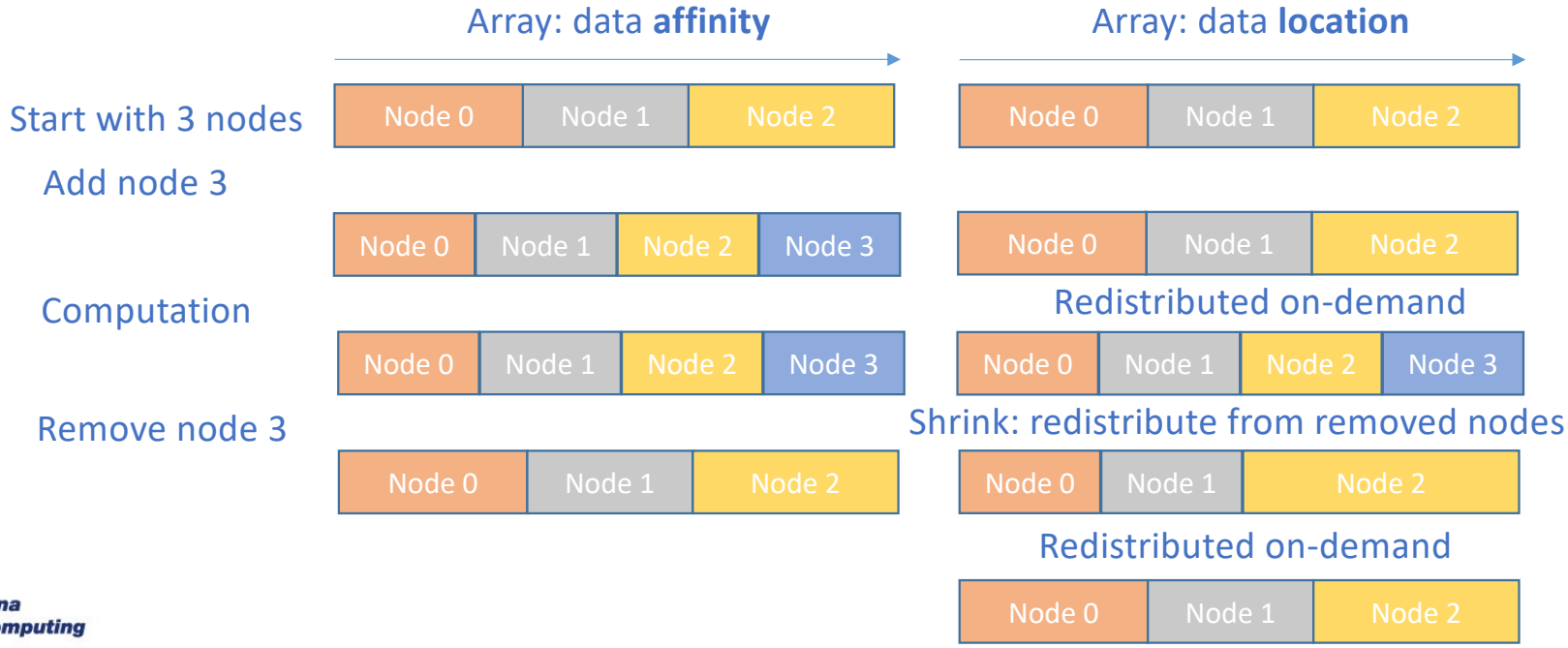
```
while (…) {

    #pragma oss taskloop inout(a[j])
    for(j=0; j<N; j++) {
        a[j]++;
    }

    #pragma oss taskwait
    int delta = ... // determine change
    nanos6_cluster_resize(delta);
}
```

Using the current implementation

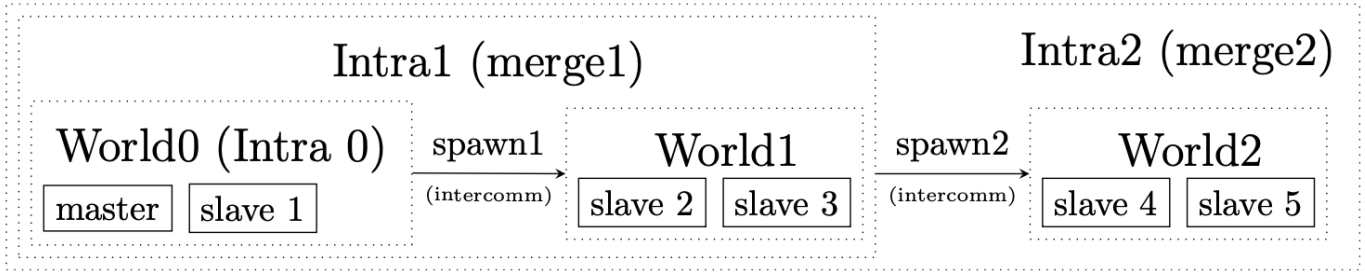
- Interaction with Slurm
  - If Slurm API is available and permit_job_expansion is true
- Minimum number of processes: from `mpirun`, `mpiexec` or `srun` command
- Maximum number of processes: `nanos6.toml` configuration file
- Number of processes per node: taken from `SLURM_*` environment variables

Data redistribution policy: Lazy

- Current lazy approach
  - On spawn: update affinity, but don't redistribute
  - On shrink: update affinity, move from removed nodes to node with affinity
  - Example: `nanos6_dmalloc(…)` with equipartition (blocked) distribution



Process spawning with MPI_Comm_spawn (temporary until MPI Session interface available)

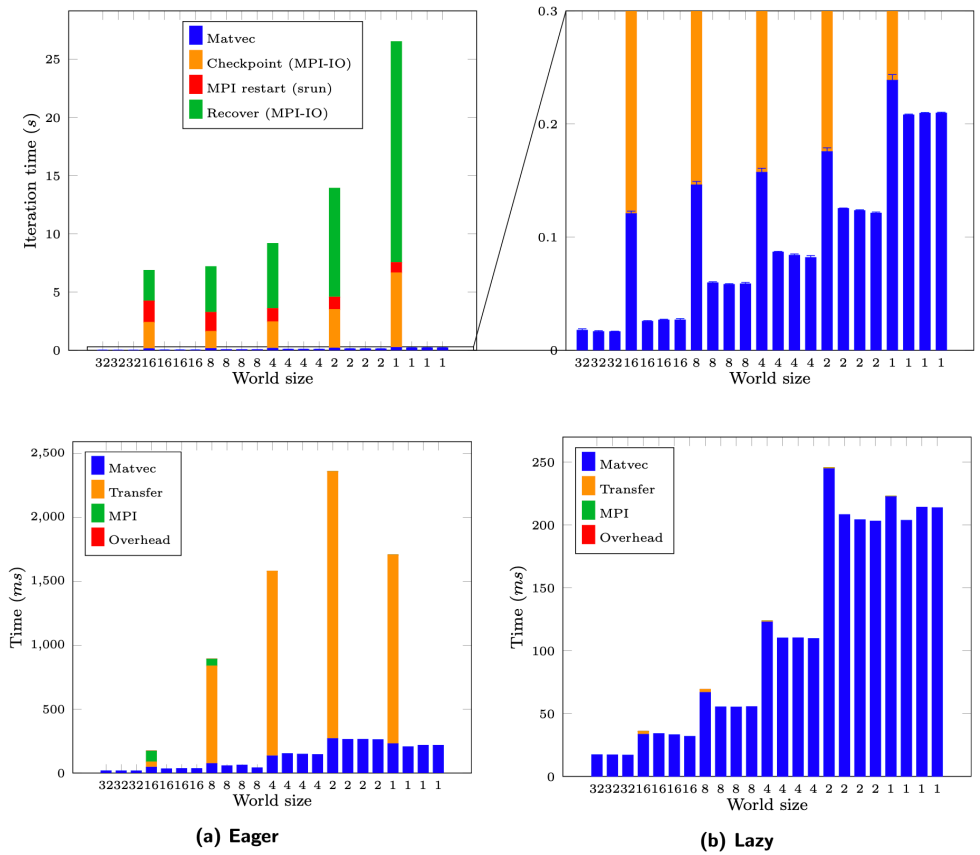


- Each spawn operation adds a layer
  - Previous intra-communicator spawns new nodes
- Last-in First-out
  - Release operation must match most recent spawn => undo layers one by one
- Matching granularities
  - Release operation must release all ranks that were spawned together
  - Implicit barrier in `MPI_Finalize`: does not return until all processes call `MPI_Comm_disconnect`
  - Cannot break children's `MPI_COMM_WORLD`

Results: Matvec timelines (64k) when shrinking

Checkpoint restart

Dynamic processes



(a) Eager (b) Lazy

Note: Matvec: Most data (the matrix) is read only

Conclusion


BSC
Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación

Conclusions

- OmpSs-2@Cluster is the distributed memory variant of OmpSs-2
- Objectives
  1. Alternative to MPI for small-scale (up to 8 or 16 nodes for ~ms task granularity)
  2. Dynamic load balancing of MPI+OmpSs-2 programs through task offload
  3. Node-level malleability transparent to application

- **We are looking for feedback and application use cases**
- OpenMP tasks are sufficient – we can take it from there
- Please contact Paul Carpenter paul.carpenter@bsc.es