Asynchronous MPI communication with OpenMP tasks spawning task dependency graphs across nodes

Dr. Joachim Jenke (jenke@itc.rwth-aachen.de)

NHR4CES NHR for Computational Engineering Science

i12 High Performance Computing | RWTH AACHEN UNIVERSITY

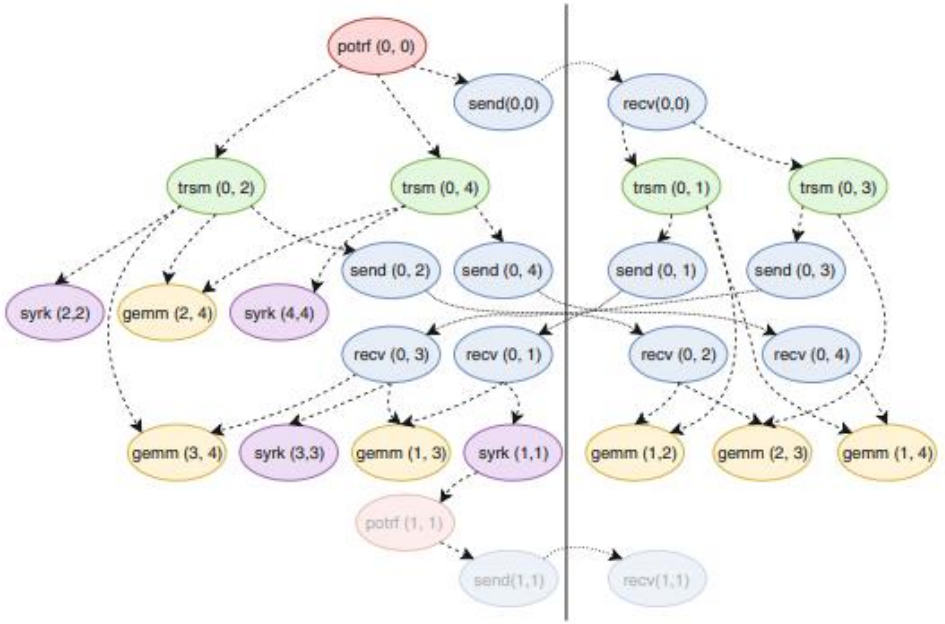
MPI Communication in OpenMP Tasking Programs

- Work initially presented by Schuchart et al. at IWOMP'18
- Calculation tasks generate data
- Sender tasks depend on the data to communicate
- Receiver tasks have out dependencies for the received data

➢ Current MPI + OpenMP provide no scalable solution

➢ TAMPI + OmpSs-2 show a feasible implementation

➢ MPI – Detach presented at EuroMPI'20



OpenMP Task Dependencies

OpenMP Task Dependencies Overview

- **`task`**, **`taskwait`**, **`target`** constructs can have a **`depend`** clause to express dependencies

- **`depend(`***[depend-modifier`,`] dependence-type* **`:`** *locator-list*`)`

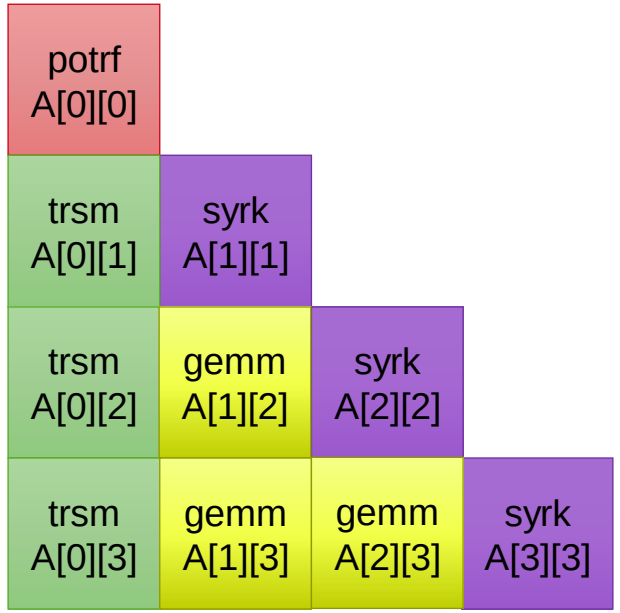
- *dependence-type* is any in:
  - In, inout/out, mutexinoutset, inoutset, depobj

- *locator-list* is a list of variables representing their location.

Is ordered with	in	out/inout	mutexinoutset	inoutset
in		X	X	X
out/inout	X	X	X	X
mutexinoutset	X	X	O	X
inoutset	X	X	X	

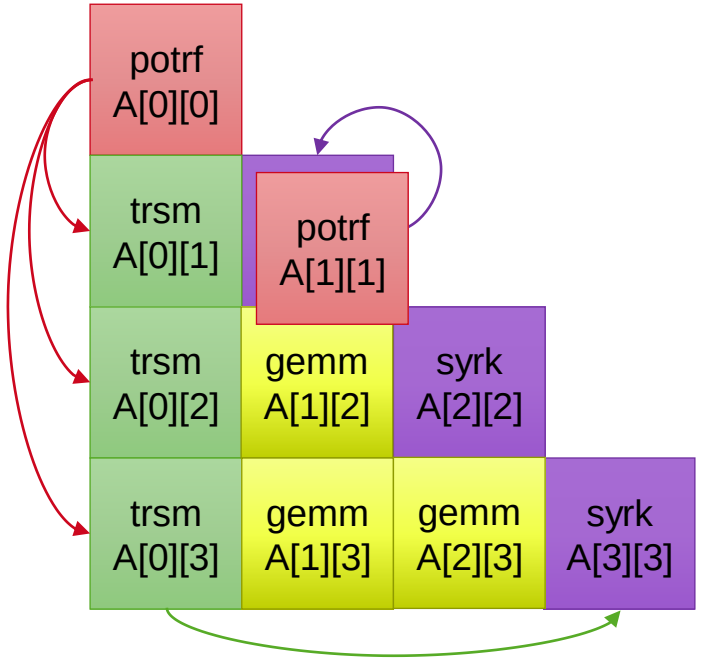
OpenMP Tasks: Block Cholesky Decomposition

```
for (int i = 0; i < n; i++) {
#pragma omp task
   potrf (A[i][i], ...);
#pragma omp taskwait
   for (int y = i + 1; y < n; y++)
#pragma omp task
      trsm(A[i][y], A[i][i], ...);
#pragma omp taskwait
   for (int y = i + 1; y < n; y++)
#pragma omp task
      syrk(A[y][y], A[i][y], ...);
#pragma omp taskwait
   for (int x = i + 1; x < n; x++)
      for (int y = x + 1; y < n; y++)
#pragma omp task
         gemm(A[x][y], A[i][x], A[i][y], ...);
#pragma omp taskwait
}
```

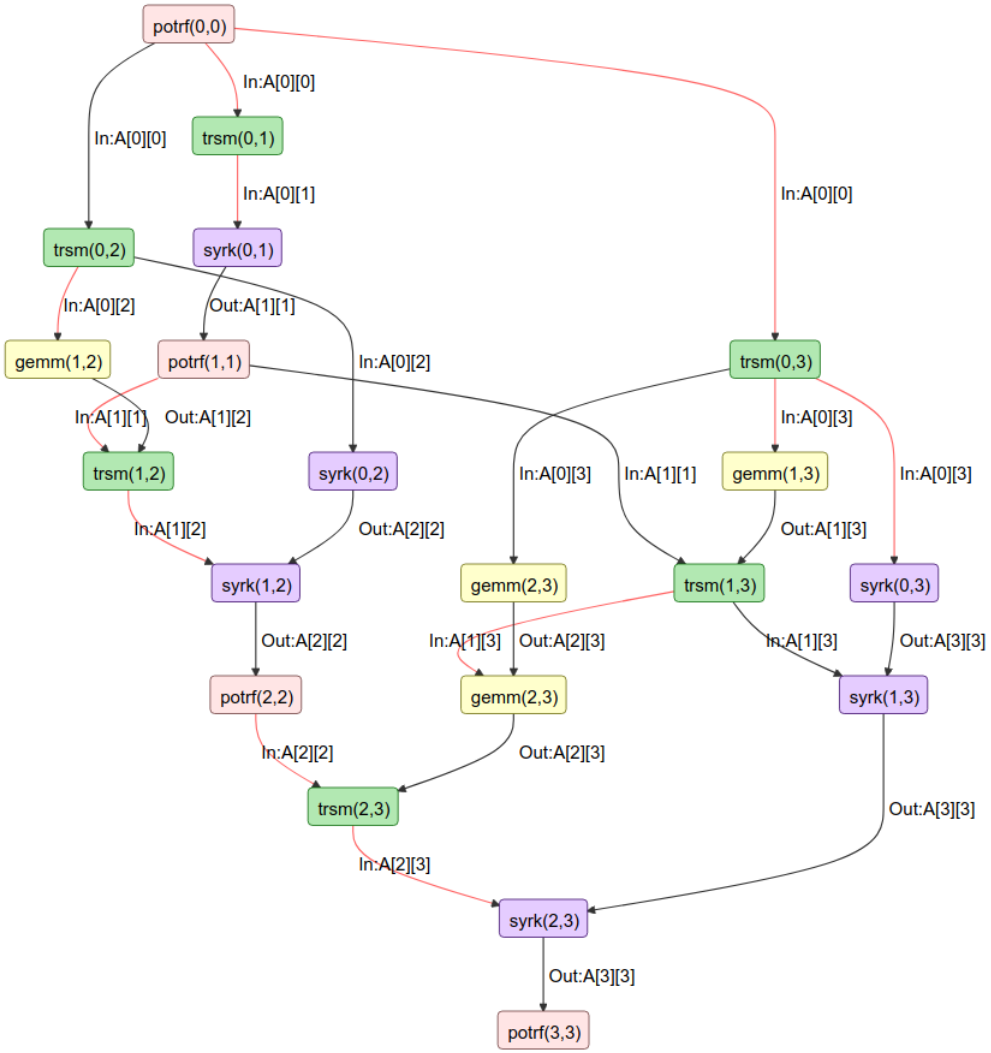


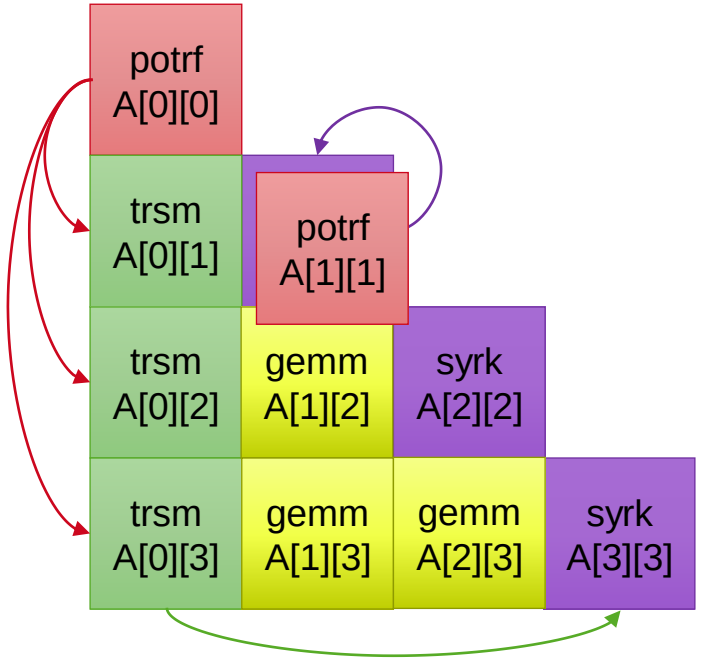
OpenMP Task Dependencies: Block Cholesky Decomposition

```
for (int i = 0; i < n; i++) {
#pragma omp task depend(inout:A[i][i])
   potrf (A[i][i], ...);
   for (int y = i + 1; y < n; y++)
#pragma omp task depend(in:A[i][i]) \
                 depend(inout:A[i][y])
      trsm(A[i][y], A[i][i], ...);
   for (int y = i + 1; y < n; y++)
#pragma omp task depend(in:A[i][y]) \
                 depend(inout:A[y][y])
      syrk(A[y][y], A[i][y], ...);
   for (int x = i + 1; x < n; x++)
      for (int y = x + 1; y < n; y++)
#pragma omp task depend(in:A[i][x], A[i][y]) \
                 depend(inout:A[x][y])
         gemm(A[x][y], A[i][x], A[i][y], ...);
}
```

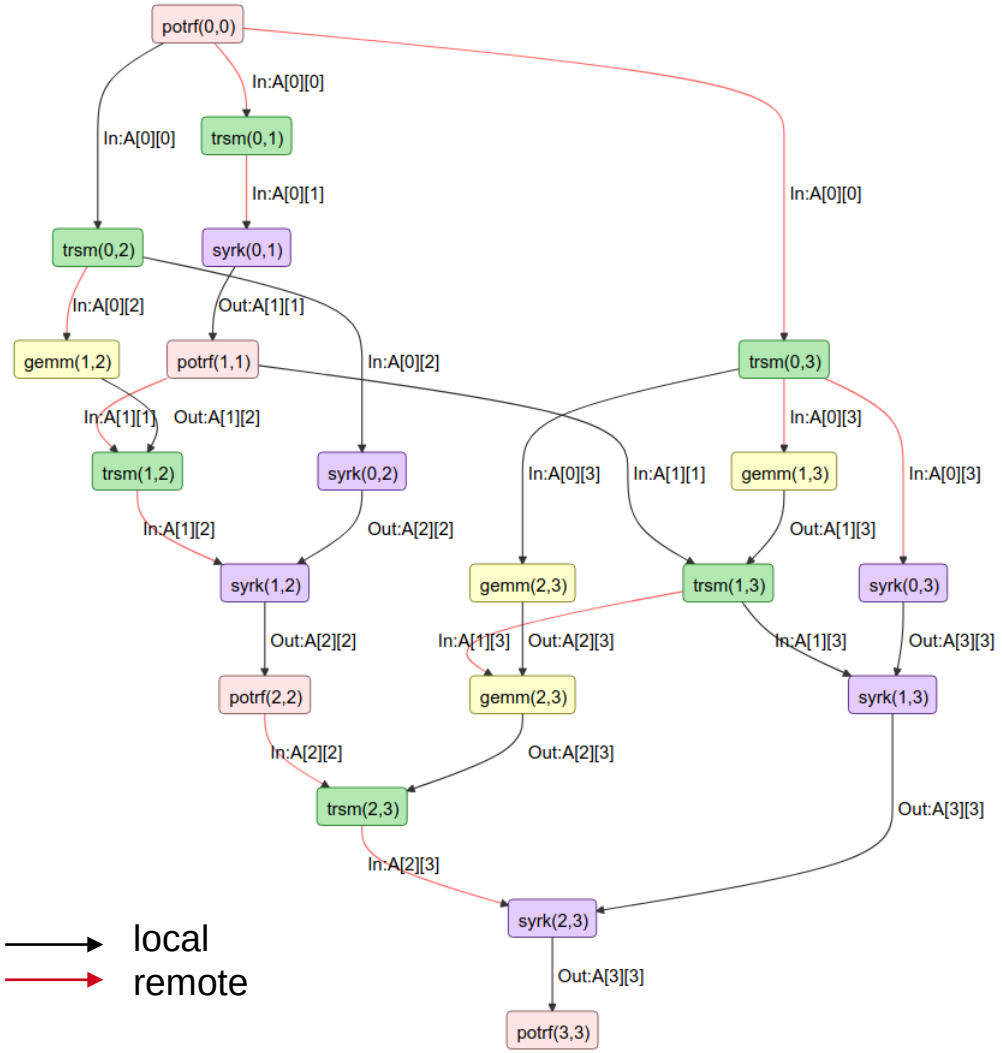


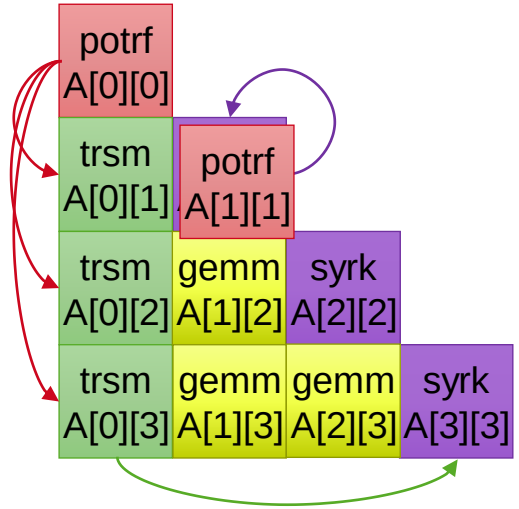
OpenMP Task Dependencies as a DAG

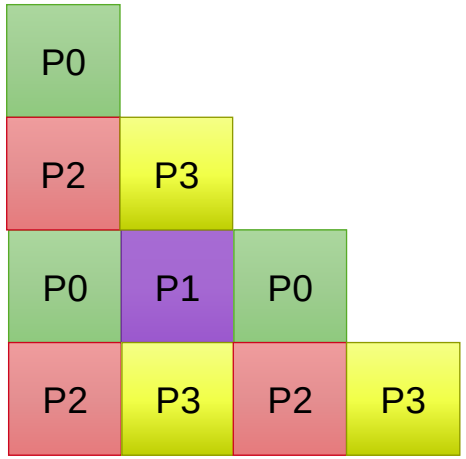




Decomposition for Distributed Memory Calculation







Distributed task dependencies

- Out/inout dependence produces data
  - Needs to **send** data to dependent tasks
- In/inout dependence consumes data
  - Needs to **receive** and wait for data

- OpenMP 5.0 introduced detachable tasks
- Semantically:

Executing `omp_fulfill_event(req)` completes the task

```
#pragma omp task depend(out:recvbuf) detach(req)
{
  MPI_Request req;
  MPI_Irecv(recvbuf, …, &req);
} // the task finishes execution and returns, but completion
  // is coupled with completion of the receive

#pragma omp task depend(in:sendbuf)
  MPI_Send(sendbuf, …);

#pragma omp task depend(in:recvbuf)
  do_something_with_buf(recvbuf);
```

Integration of OpenMP detached tasks and MPI Communication

Proposed MPI detach functions

- Hand back request to MPI library and register for a completion callback.
- Compare to `int MPI_Wait(MPI_Request *request, MPI_Status *status)`:

```
typedef void MPIX_Detach_status_function(void *data, MPI_Status status);

int MPIX_Detach_status(
  MPI_Request *request,
  MPIX_Detach_status_function *callback,
  void *data);
```

- If we are not interested in the status, we cannot pass `MPI_STATUS_IGNORE`:

```
typedef void MPIX_Detach_function(void *data);

int MPIX_Detach(
  MPI_Request *request,
  MPIX_Detach_function *callback,
  void *data);
```

Proposed MPI detach functions

- Compare to `int MPI_Waitall(int count, MPI_Request requests[], MPI_Status statuses[])`:

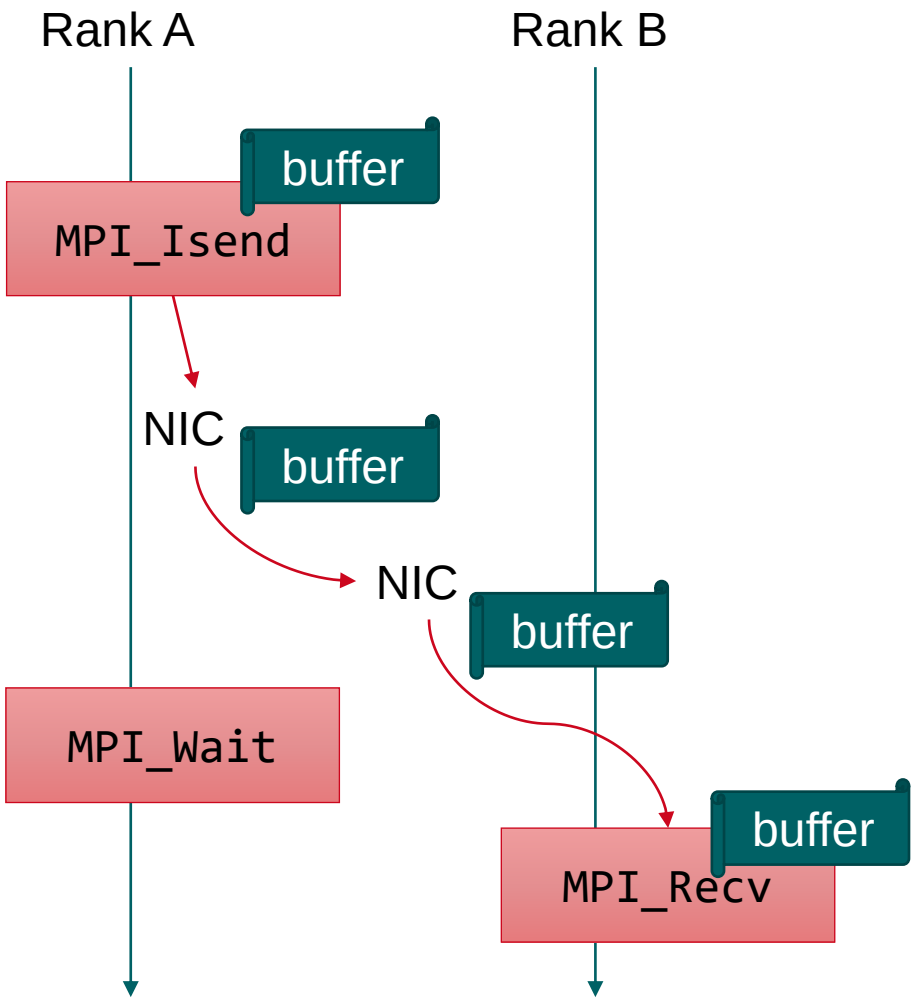
```
typedef void MPIX_Detach_statuses_function(void *data, int count, MPI_Status statuses[]);

int MPIX_Detach_all_status(
  int count,
  MPI_Request requests[],
  MPIX_Detach_status_function *callback,
  void *data);
```

- Again, we cannot pass `MPI_STATUSES_IGNORE`:

```
int MPIX_Detach_all(
  MPI_Request *request,
  MPIX_Detach_function *callback,
  void *data);
```

Progress in MPI communication

- MPI guarantees that once a matching pair of send and receive is initiated, one of them will (eventually) complete.
- Choices:
  - Is there a progress thread, supervising communication progress?
  - Is progress only driven during API calls?



Driving progress

- When should MPI make progress after calling `MPIX_Detach`?
- The new function `MPIX_Progress` can be registered with a polling service:

```
int MPIX_Progress(void *data);
```

- The polling service calls the registered function when ever it is convenient.
- The function must be non-blocking.
- Calling this function regularly will finally allow the completion callback to be called.
- The prototype starts a background polling thread, if the following env is set:
  - **`export MPIX_DETACH = progress`**

Usage

OpenMP tasks using detach clause (OpenMP 5.0)

```
omp_event_handle_t ev_handle;
#pragma omp task detach(ev_handle) depend(out: recvbuf)
{
  MPI_Request req;
  MPI_Irecv(recvbuf, …, &req);
  MPIX_Detach(&req, (MPI_Detach_callback *) omp_fulfill_event, (void*) ev_handle);
}


#pragma omp task depend(in: recvbuf)
  do_something_with_buf(recvbuf);
```

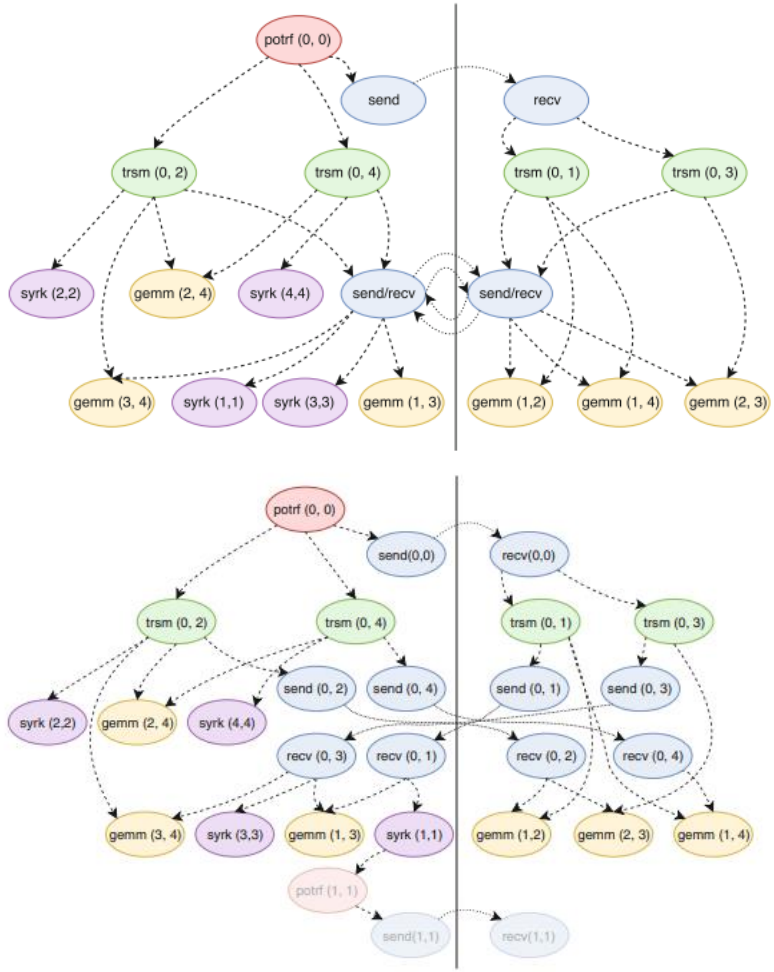
Evaluation

Extending distributed Block Cholesky Factorization with MPI detach

- Presented by Schuchart et al. at IWOMP'18:
https://github.com/devreal/cholesky_omptasks

- Coarse-grain communication (***singlecom***) using a single communication task per level

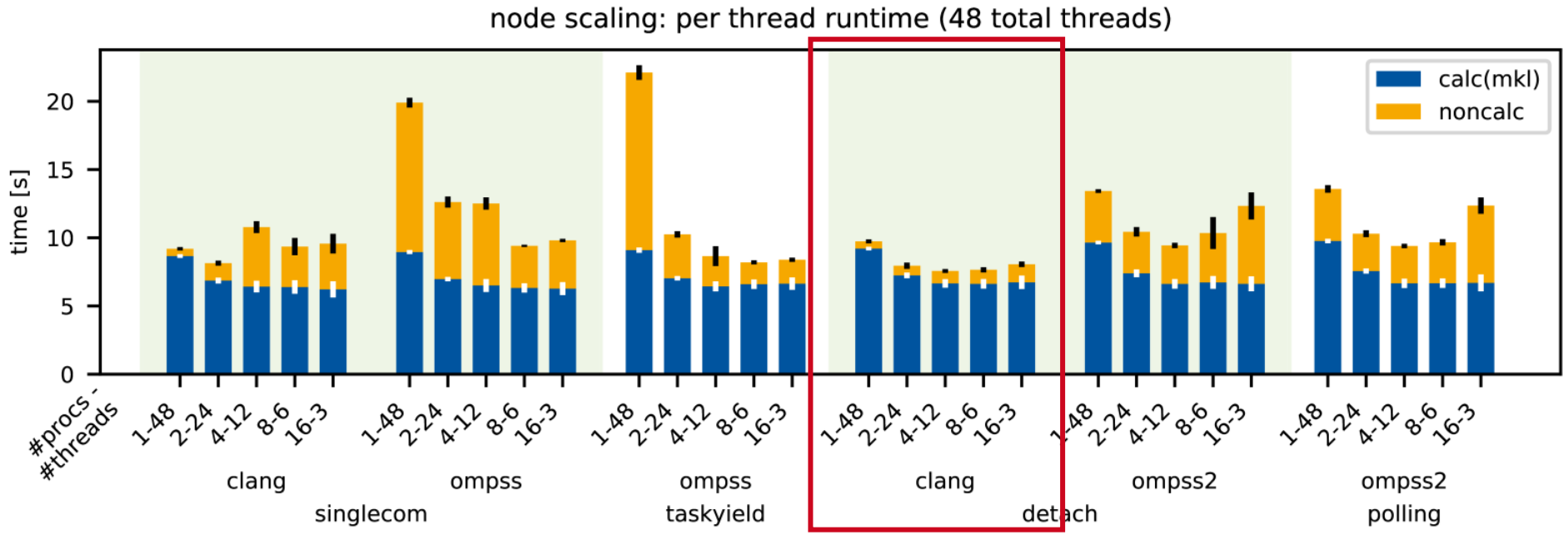
- Fine-grain communication using a communication task per block
  - ***taskyield***: test-loop with taskyield
  - ***detach***: MPIX-detach replaces test-loop
  - ***polling***: MPIX-progress replaces progress thread



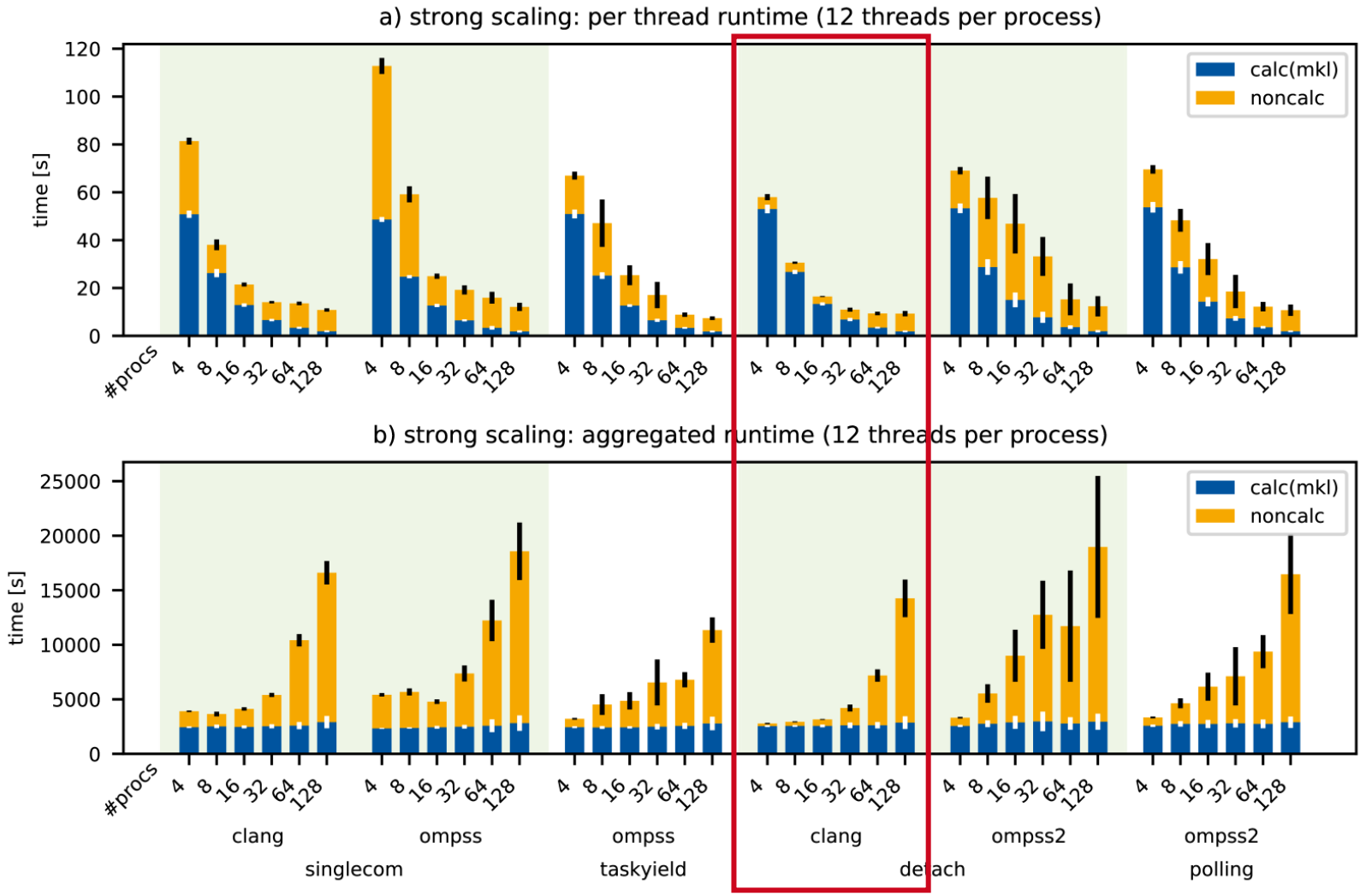
Graphs from: Schuchart et al., *The Impact of Taskyield on the Design of Tasks Communicating Through MPI*, IWOMP'18

Node scaling experiment: $(32k)^2$ matrix, 256^2 blocks

- Filling 48 cores of a node with MPI processes/OpenMP threads



Strong scaling experiment: (64k)² matrix, 256² blocks

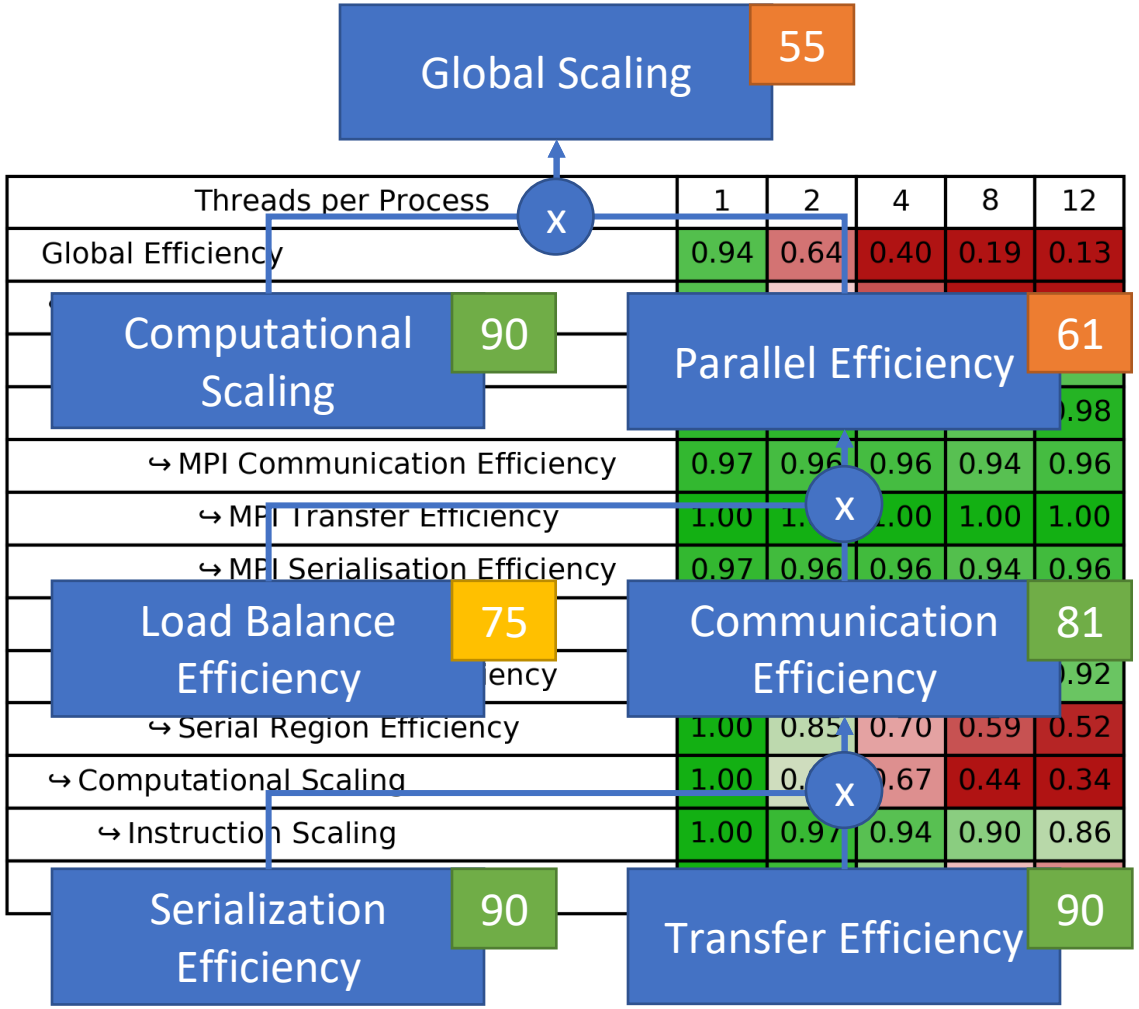


Understanding the scalability limits

For details see EuroPar‘22 paper: *On-the-fly Performance Model Factors for Multi-Level Parallelism*

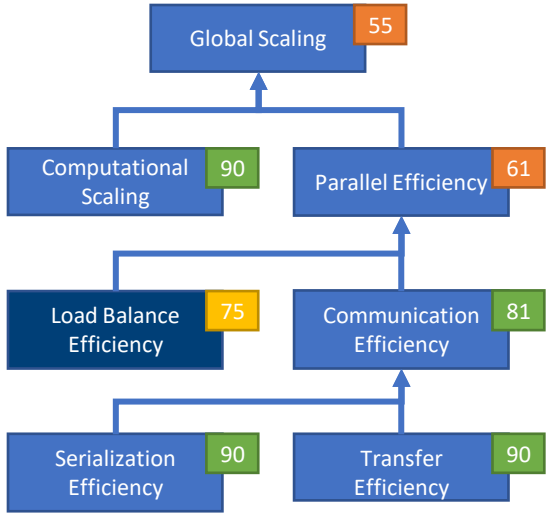
Performance Model Factors

- Hierarchy of metrics developed at BSC
- Highlight issues in the parallel structure of an application
- Parallel Efficiency breaks down into
  - Load balance
  - Serialization
  - Transfer
- Computational Scaling captures impact of scaling to node-level performance



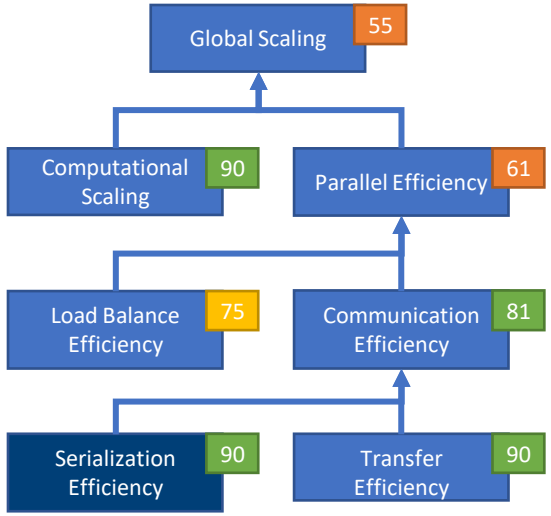
Load Balance

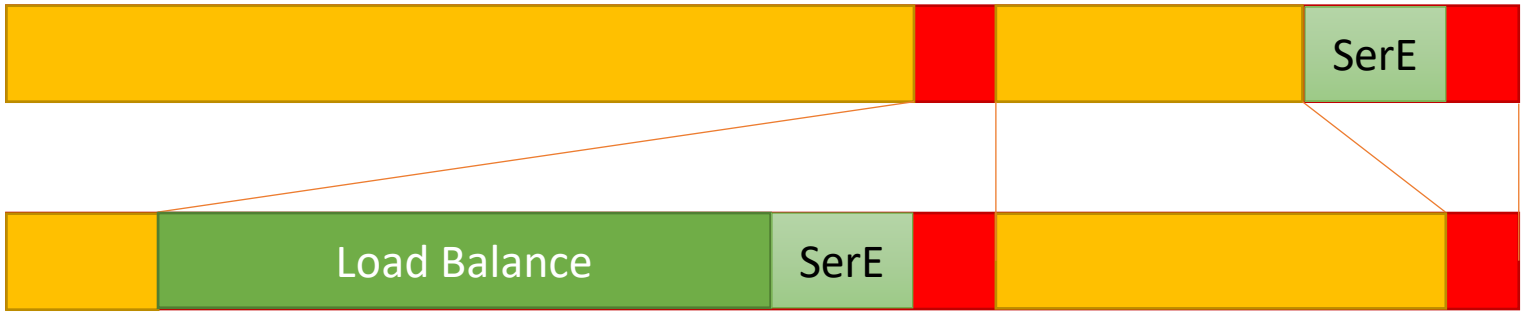
- Reflects global imbalance of work between execution units
- $LB = \frac{avg(useful\ time)}{\max(useful\ time)}$
- *Useful time*: execution time outside parallel runtimes



Serialization Efficiency

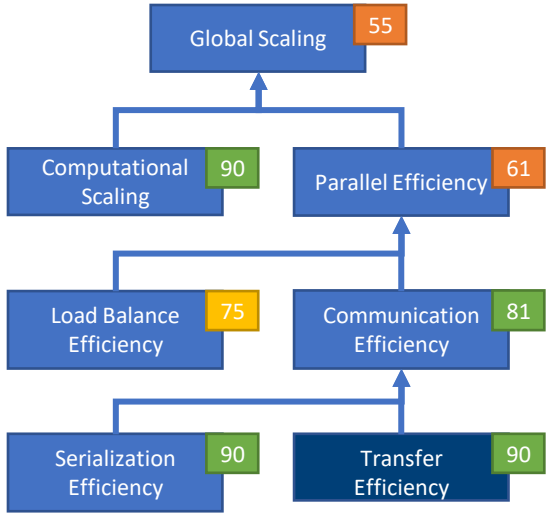
- Reflects moving imbalance of work between execution units, resp., alternating dependencies
- $SerE = \frac{max(useful\ time)}{ideal\ runtime}$
- *Ideal runtime*: execution time on an ideal machine with 0 communication cost (inf. BW / 0 lat)

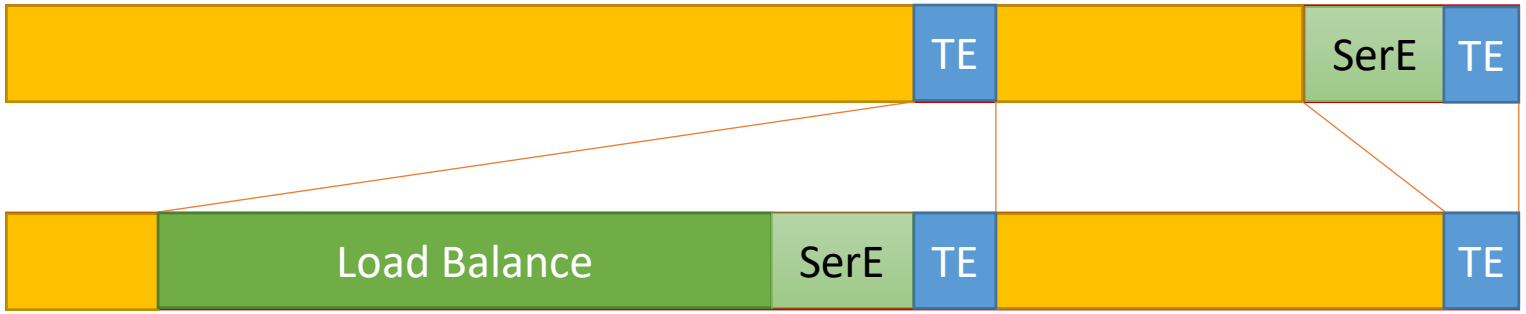




Transfer Efficiency

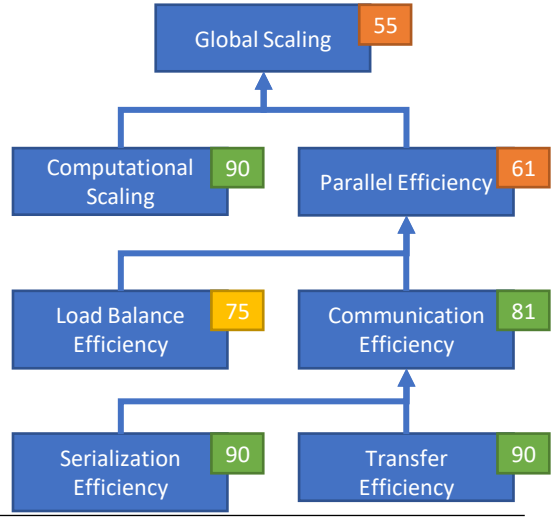
- Cost of transfer / communication / synchronization
- $TE = \frac{ideal\ runtime}{real\ runtime}$
- *Real runtime*: observed execution time





Which metrics to measure?

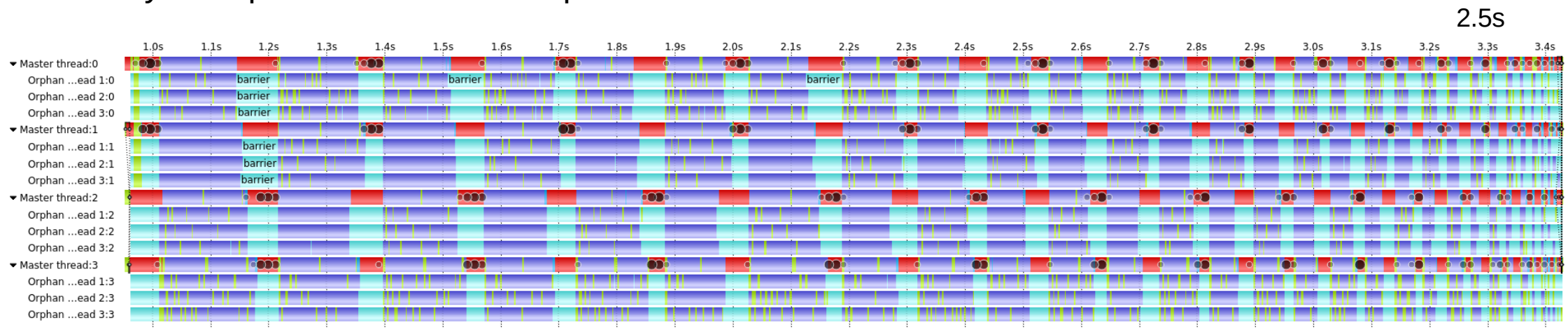
- *Useful time*: execution time outside parallel runtimes
  - Track execution time on each thread excluding time inside MPI / OpenMP runtimes
- *Real runtime*: observed execution time
  - Track wall clock time from start to end.
- *Ideal runtime*: execution time on an ideal machine with 0 communication cost (inf. BW / 0 lat)
  - Track *useful time* on **critical path** → assumes 0 communication cost



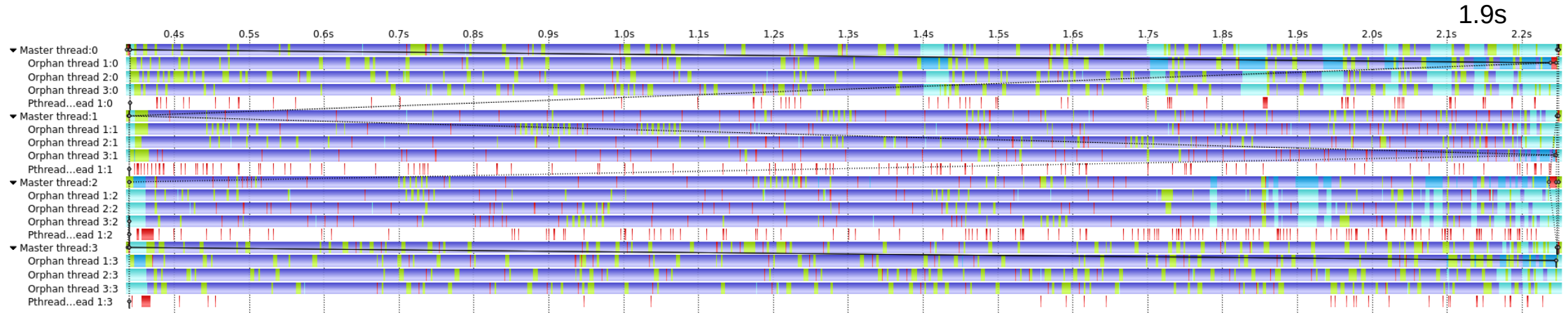
Execution trace with tricks and tweaks

- Based on user-instrumentation API used in an OMPT tool
  - ➢ No automatic analysis of performance metrics possible

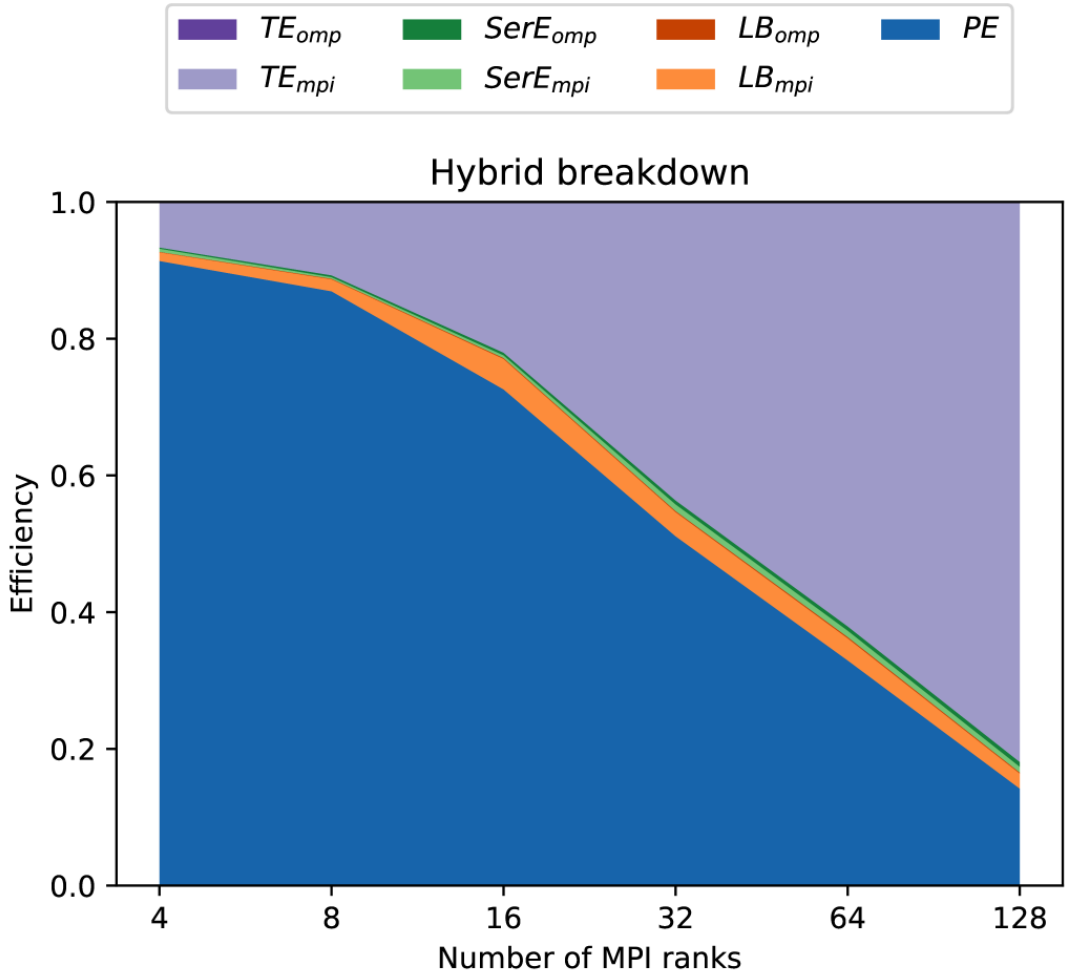
Block-synchronous

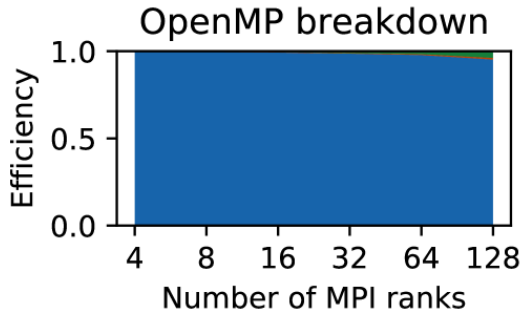


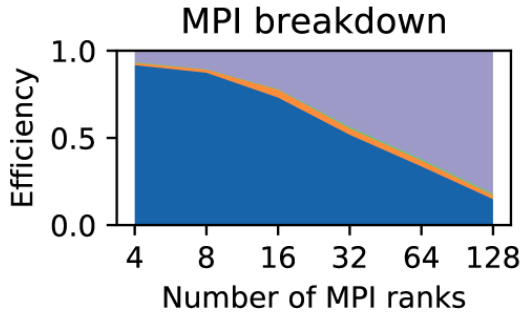
Asynchronous

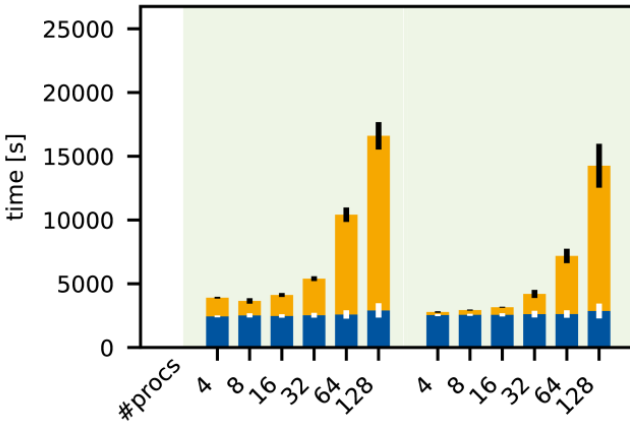


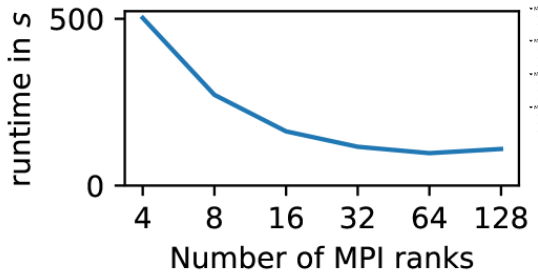
Block-synchronous execution with 12 threads / MPI process

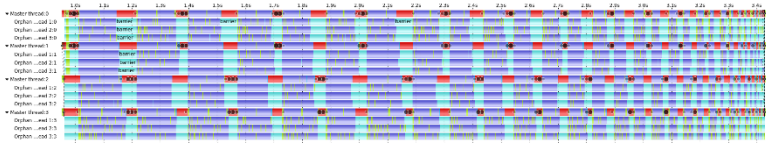




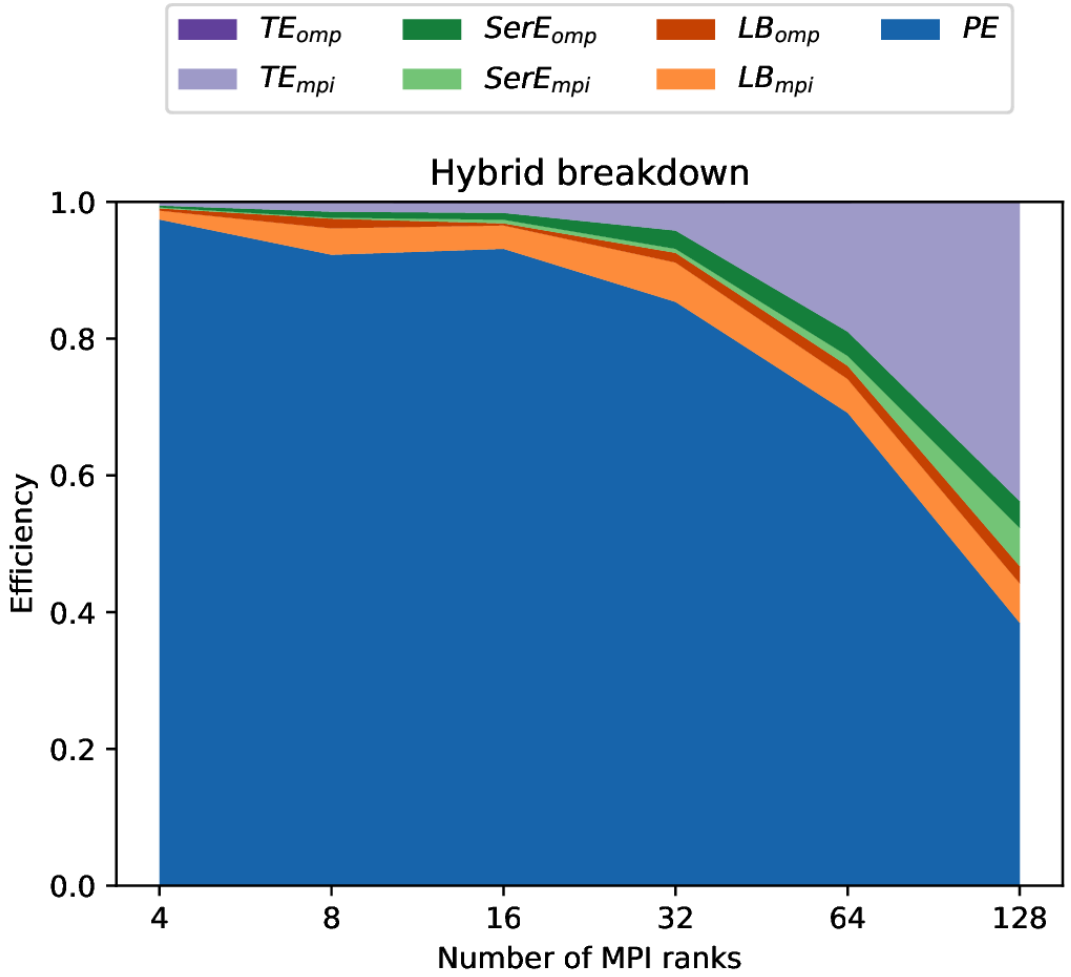


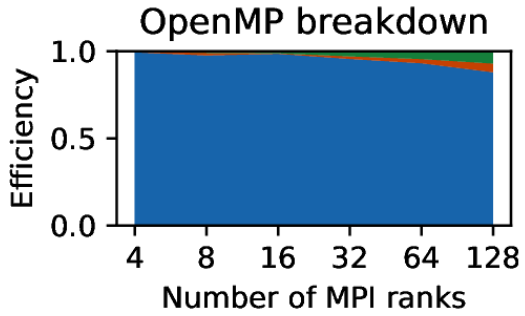


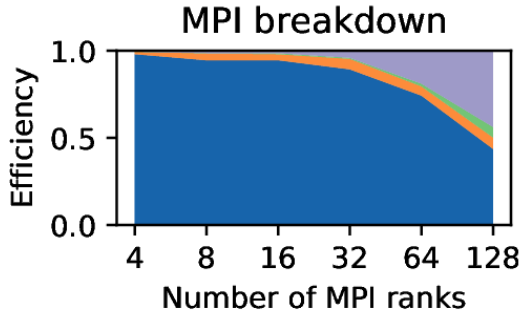


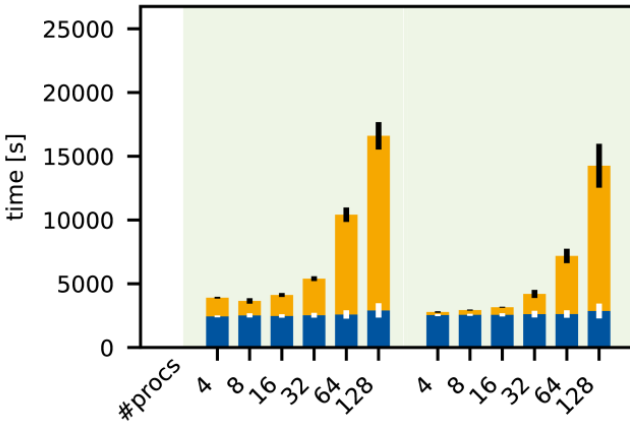


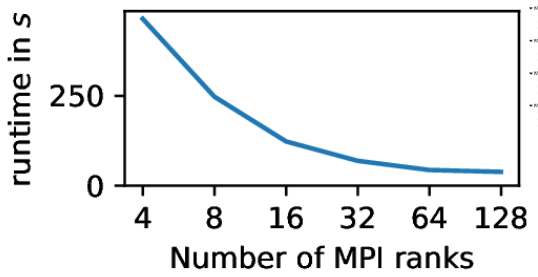
Asynchronous execution with 12 threads / MPI process

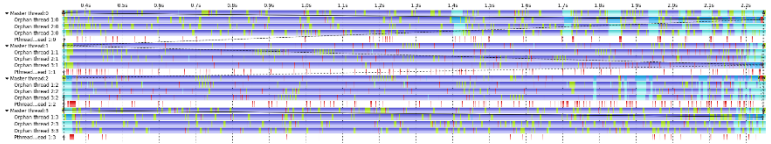












Conclusion

Conclusion

- Slim extension of MPI (7 new functions) to provide asynchronous local completion
- Usage presented for OpenMP tasks. (see hidden slides for C++ futures)
  - Can also be used in other tasking models
- Prototype implemented as PMPI wrapper:
  - https://github.com/RWTH-HPC/mpi-detach (thread-safe and just 347 sloc)
- Performance study based on distributed block cholesky factorization:
  - https://github.com/RWTH-HPC/cholesky_omptasks
- What we missed during the implementation:
  - API to determine whether a given `MPI_Request` is a persistent request and what is the state
- Hybrid performance model factors as a means to understand scalability limits