www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Hybrid approach MPI+OmpSs

PATC Tutorial

Xavier Martorell, Xavier Teruel, Antonio J. Peña

Barcelona, 22-23 May, 2019

Outline

- MPI + OpenMP/OmpSs hybrid programming
- Interaction between MPI and OpenMP/OmpSs
- TAMPI – Task-Aware MPI
- Examples of applications using MPI+OpenMP/OmpSs

MPI + OpenMP/OmpSs hybrid programming

- Distributed-memory programming
  - Separate processes
    - Private variables are unaccessible from others
  - Point-to-point and collective communication
    - Implicit synchronization
- Shared-memory programming
  - Multiple threads share the same address space
  - Communication is implicit
    - Needs explicit synchronization

MPI + OpenMP/OmpSs hybrid programming

- Why hybrid?
  - MPI is here to stay
  - A lot of HPC applications already written in MPI
  - MPI scales well to tens/hundreds of thousands of nodes
  - Everybody talks about MPI + X:
    - MPI exploits intra-node parallelism while X can exploit node parallelism but ...

- MPI + OpenMP/OmpSs
  - … propagates OpenMP/OmpSs features to global program behavior
  - Potential for automatic overlap of communication and computation through the inclusion of communication in the task graph

Why hybrid MPI+OpenMP/OmpSs programming?

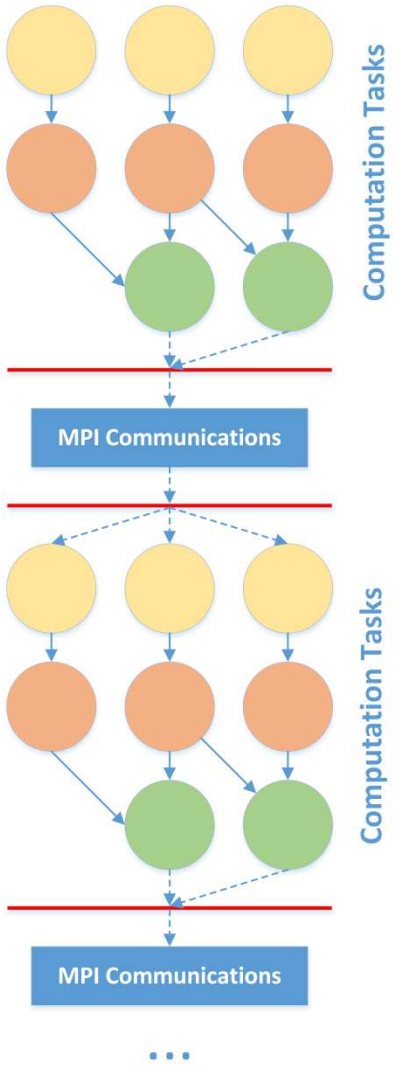
- Try to leverage best of both programing models ...
  - Message Passing Interface (MPI)
    - Designed to exploit distributed memory systems
    - Efficient and scalable message passing interface
  - OpenMP/OmpSs tasking model
    - Designed to exploit shared memory system
    - Write sequential code, but execute it in parallel
    - Fine grained synchronizations
    - Automatic load-balancing

Why hybrid MPI+OpenMP/OmpSs programming?

- … but also to exploit some potential synergies ☺
  - Fine-grained synchronization across nodes
  - Overlap of computation and communication phases
  - Intra-node load-balancing
  - Leverage intra-node application parallelism to hide network latency and maximize network throughput
- However, **interoperability issues** between MPI and OpenMP/OmpSs prevents application developers to achieve most of these goals ☹

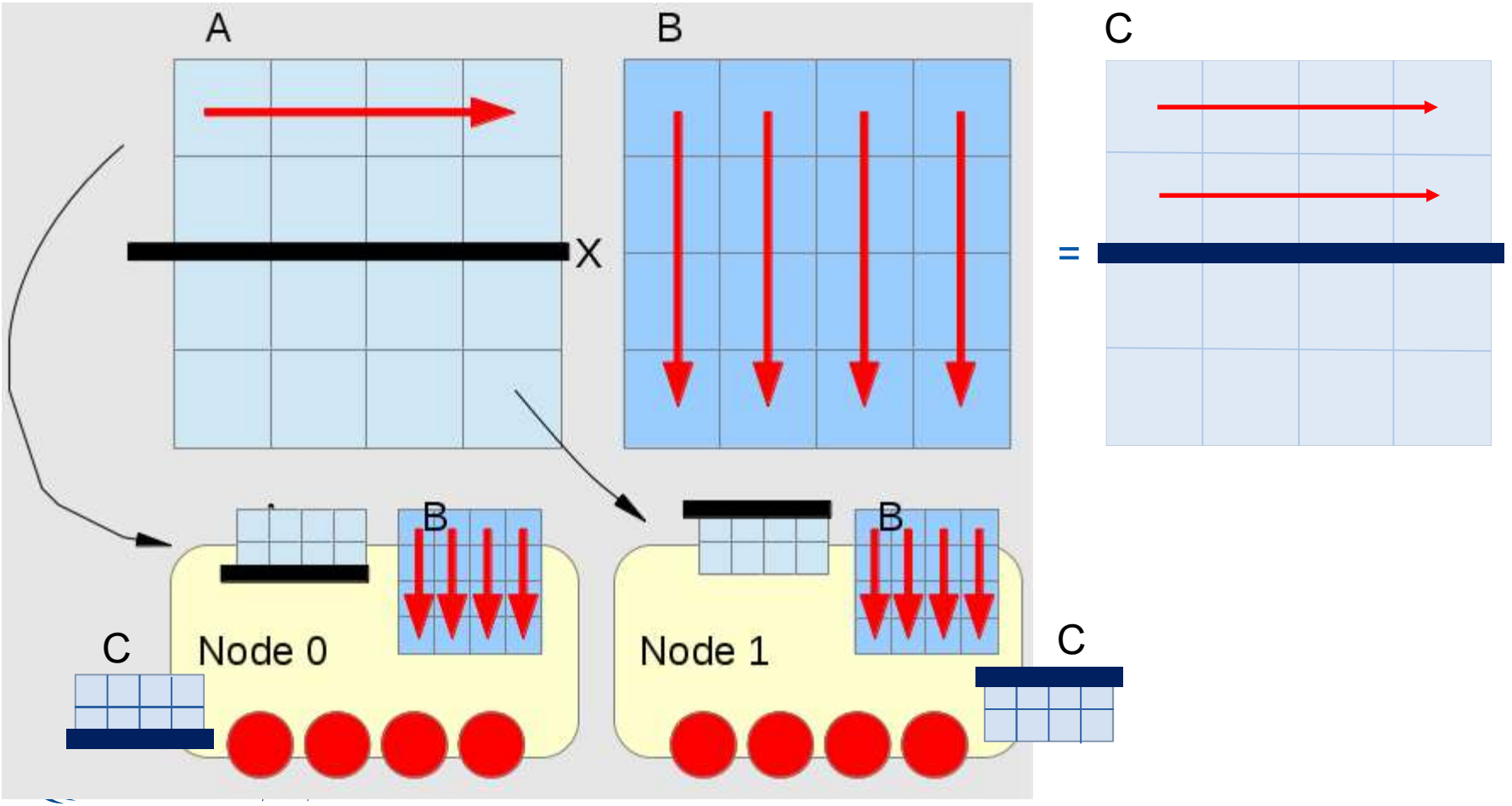
MPI + OpenMP/OmpSs (fork-join model)

- The common approach is to interleave computation and communication phases: **fork-join** model
  - Computational phases executed in parallel
    - ex: “parallel for” or “tasks + taskwait”
  - Communication phases executed sequentially
- Advantages
  - Straightforward to implement
  - No need of MPI_THREAD_MULTIPLE
- Disadvantages
  - No overlap of computation and communication phases
  - Communications delayed until all computations done



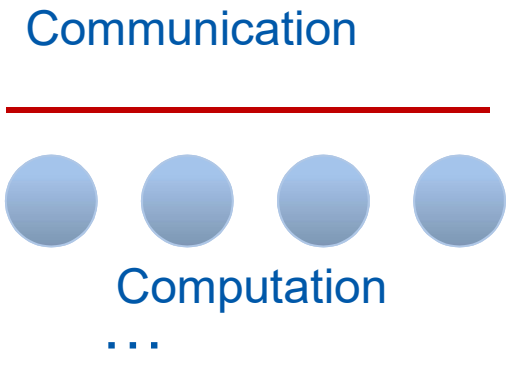
MPI + OpenMP/OmpSs hybrid programming

- Combining MPI and OpenMP/OmpSs
  - Distributed algorithms spread over nodes
  - Shared memory for computation within each node

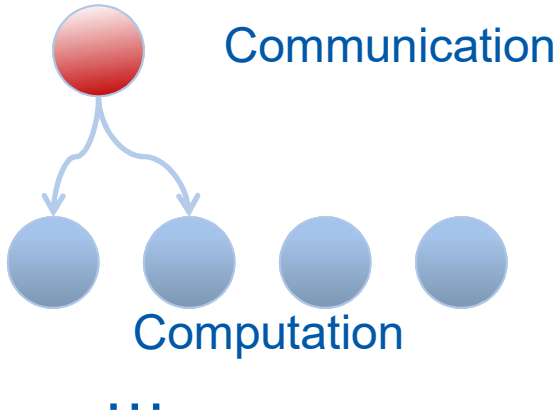


Interaction between MPI and OpenMP/OmpSs models

- Communication can be taskified or not
  - If not taskified:
    - Main program stalls till data to be sent and the reception buffers are available.
    - Still communication performed by main thread can overlap with previously generated tasks if they do not refer to communicated variables

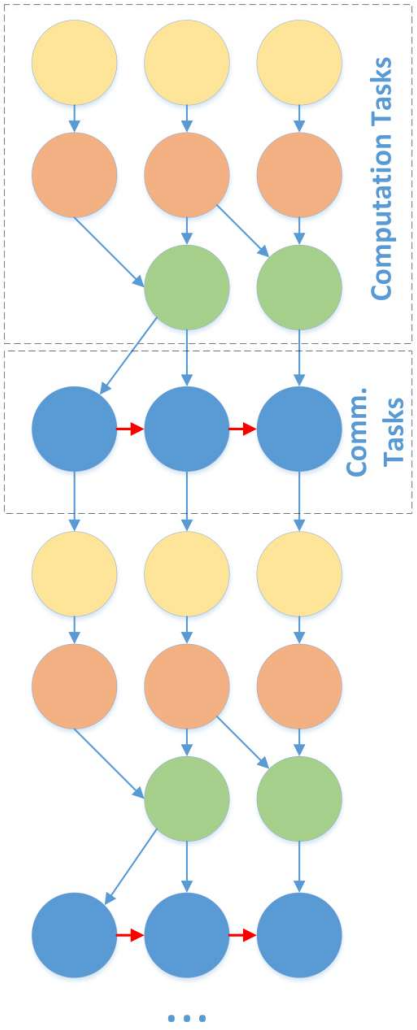


  - If taskified:
    - It can be instantiated and program can proceed generating work
    - Tasks calling MPI will execute out of order with respect to any other task just honoring local dependences within the process



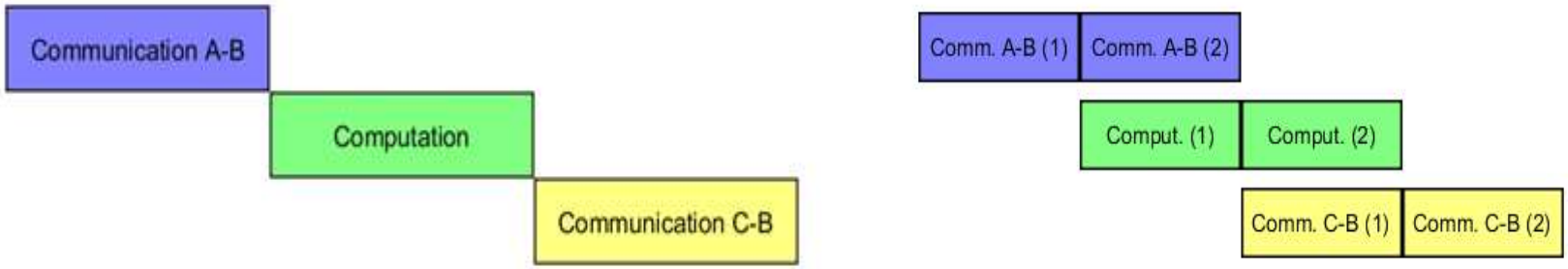
MPI + OpenMP/OmpSs (tasking model)

- In this approach *tasks* are used **both** for computational phases and communication phases
- Advantages
  - Potential to overlap computation and communication phases
  - No ”artificial” barrier between phases
- Disadvantages
  - Manually match MPI operations (MPI **tag**)
  - Due to **interoperability issues** between MPI and OpenMP the communication *tasks* have to be serialized (**red arrows**)
  - These limitations severely limit the potential benefits of this approach



Taskifying communications: performance

- Communication and computation overlapping
  - performing useful work while the message data is still in transit
  - using double buffering allows communication and computation overlap

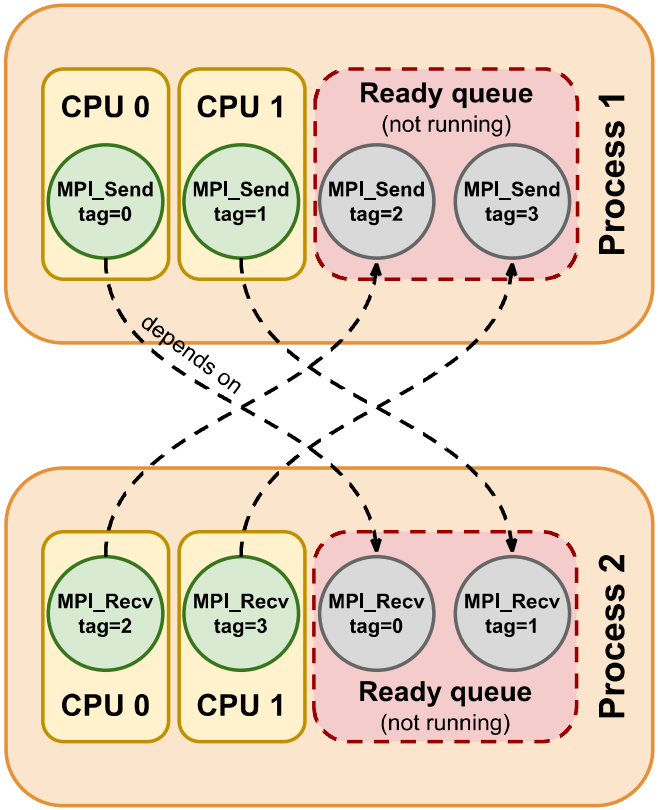


  - non-blocking communication allows routines to return immediately
    - Check the results of the MPI call
    - Taskyield otherwise → opportunity to do other independent work

MPI + OpenMP/OmpSs (tasking model) Deadlock!

- Tasks can be scheduled out-of-order, so MPI operations can also be executed out-of-order ... needs MPI_THREAD_MULTIPLE
- All CPUs can stall executing MPI **receive** operations that depend on the eventual completion of MPI **send** operations of this rank, but these will never be executed!

```
int main(int argc, char **argv) {
  int provided, rank;
  MPI_Init_thread(&argc, &argv,
       MPI_THREAD_MULTIPLE, &provided);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  int data[4]; init(data, 4);

  if (rank == 0) {
    for (int i = 0; i < 4; ++i) {
      int tag = i;
      #pragma oss task in(data[i])
      MPI_Send(&data[i], 1, MPI_INT, 1, tag,
                             MPI_COMM_WORLD);
    }
  } else if (rank == 1) {
    for (int i = 0; i < 4; ++i) {
      int tag = i;
      #pragma oss task out(data[i])
      MPI_Recv(&data[i], 1, MPI_INT, 0, tag,
        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
  }
  #pragma oss taskwait
  MPI_Finalize();
}
```



Taskifying communications

- Communication Tasks:
  - Single MPI communication

```
#pragma omp task depend(in: buffer[0;N])
MPI_send(... buffer..., int N);
```

  - Burst of MPI communications (+ some small but potentially critical computation between them)
    - Less overhead
    - Typically the inter-process synchronization cost is paid in the first MPI call within the task

```
#pragma omp task depend(in: buffer[0;N]...)
{
    MPI_Send(... buffer..., int N);
    MPI_Send...
    MPI_Send...
}
```

Taskifying communications

- Issues to consider
  - Possibly several concurrent MPI calls → thread safe MPI
    - Might not be available or really concurrent in some installations

```
res = MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
```

  - MPI tasks waste cores if communication partner delays or long transfer times
    - Less problem as more and more cores per node become available
  - Reordering of MPI calls + blocking calls + limited number of cores → potential to introduce deadlock
    - → need to control order of communication tasks, improve use of tags

Taskifying communications: false-matching

- Parallel intra-node programs perform communication at any order
  - Thus they can be received at any order and produce false-matching

> "A message can be received by a receive operation if its envelope matches the source, tag and comm values specified by the receive operation. The receiver may specify a wildcard MPI_ANY_SOURCE value for source, and/or a wildcard MPI_ANY_TAG value for tag, indicating that any source and/or tag are acceptable."

  - False matching may lead into different problems
    - Incorrect results (having in a buffer the contents expected in other buffer)
    - Buffer overflow (if buffer sizes are not the same)
    - Deadlock (if ends up having all threads/processes blocked in receive calls)
  - Importance to identify communication uniquely using different tags

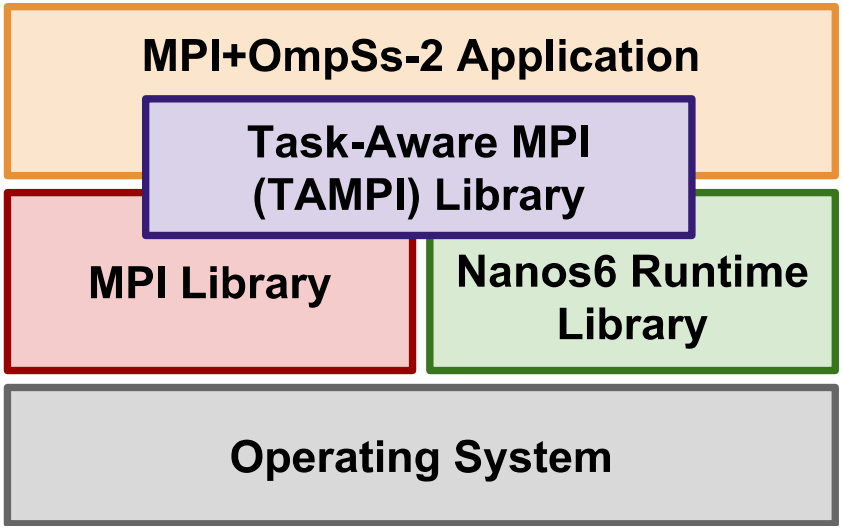
TAMPI – Task-aware MPI

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

TAMPI. Support for synchronous primitives

- TAMPI intercepts all synchronous MPI primitives to avoid a deadlock when called from inside a task
- This behavior is available to end users using a new threading support level: **MPI_TASK_MULTIPLE**
- TAMPI is compiled against a native MPI library and the Nanos6 runtime
- Very useful for converting legacy MPI codes to hybrid ones

```c
int main(int argc, char **argv)
{
  int prov;
  MPI_Init_thread(&argc, &argv,
       MPI_TASK_MULTIPLE, &prov);
  assert(prov == MPI_TASK_MULTIPLE);

  /* ... */

  MPI_Finalize();
  return 0;
}
```



TAMPI. Support for synchronous primitives

- Multiple MPI call can be safely executed from inside a task
- Calls to external libraries that uses MPI are also supported
- Even calls to asynchronous MPI primitives!
- Collective operations not supported (no **tag** parameter, MPI only allows one collective operation in-flight per communicator)

```
#pragma oss task inout(data[i])
{

  MPI_Send(&data[i], 1, MPI_INT, 1, tag,
           MPI_COMM_WORLD);
  // The send() operation has already been completed

  MPI_Recv(&data[i], 1, MPI_INT, 0, tag,
           MPI_COMM_WORLD, MPI_STATUS_IGNORE);

  // The recv() operation has already been completed
  printf("%d", data[i]);

  // Now we call and 3rd party library that uses
  // MPI inside => Ok!
  foo(data);
}
```

```
#pragma oss task ...
{
 /* ... */

  MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1,
                     MPI_COMM_WORLD, &reqs[0]);
  MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2,
                     MPI_COMM_WORLD, &reqs[1]);

  MPI_Isend(&rank, 1, MPI_INT, prev, tag2,
                     MPI_COMM_WORLD, &reqs[2]);
  MPI_Isend(&rank, 1, MPI_INT, next, tag1,
                     MPI_COMM_WORLD, &reqs[3]);

    /*  do some work  */

  MPI_Waitall(4, reqs, stats);
}
```

TAMPI. Support for synchronous primitives

- Leverage the OmpSs-2 low-level **pause/resume** and **polling service** API to improve the interoperability between MPI and tasking
- Expose this new feature as a new threading support level in MPI: **MPI_TASK_MULTIPLE**
- When TAMPI is initialized using the **MPI_TASK_MULTIPLE** threading model, all the blocking operations are intercepted and converted to their non-blocking counter parts. Ex: MPI_Recv() executed inside a task.

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,
        int tag, MPI_Comm comm, MPI_Status *status) {
    int err, completed = 0;
    if (Interop::isEnabled()) {
        MPI_Request request;
        err = MPI_Irecv(buf, count, datatype, source, tag, comm, &request);
        MPI_Test(&request, &completed, status);
        if (!completed) {
            Ticket ticket(&request, status);
            ticket._waiter = get_current_blocking_context();
            _pendingTickets.add(ticket);
            block_current_task(ticket._waiter);
        }
        return err;
    }
    return PMPI_Recv(buf, count, datatype, source, tag, comm, status);
}
```

TAMPI. Support for asynchronous primitives

- **TAMPI_Iwait/all**(MPI_Request *req, MPI_status *status)
  - Asynchronous function that **links** the release of task's dependencies with the completion of the in-flight MPI requests
  - Once the tasks finishes AND all the MPI operations registered with *TAMPI_Iwait/all()* have completed, the dependencies are released.

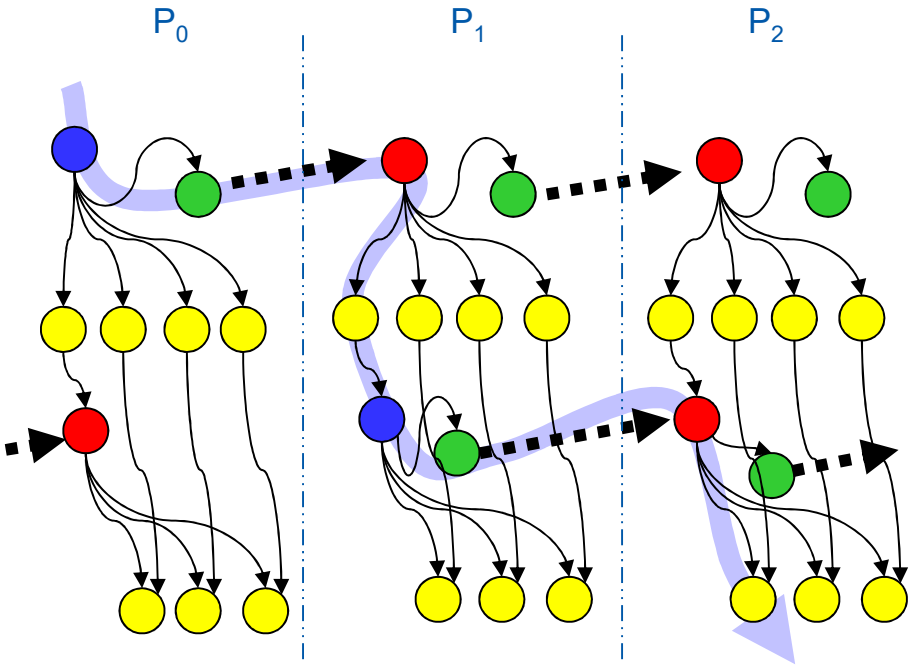
```
int TAMPI_Iwait(MPI_Request *request, MPI_Status *status);
int TAMPI_Iwaitall(int count, MPI_Request *requests, MPI_Status *statuses);
```

```
void sendLastComputeRow(block_t *matrix, int nbx, int nby,
                                         int rank, int rank_size)
{
    for (int by = 1; by < nby-1; ++by) {
        #pragma oss task in(([nbx][nby]matrix)[nbx-2][by])
            {
                    MPI_Request request;
                    MPI_Isend(&matrix[(nbx-2)*nby + by][BSX-1], BSY, MPI_DOUBLE,
                                  rank + 1, by, MPI_COMM_WORLD, &request);
                    TAMPI_Iwait(&request, MPI_STATUS_IGNORE);
            }
    }
}
```

Application examples

Hybrid MPI/OmpSs: Linpack example

- Linpack example
- Overlap communication/computation
- Extend asynchronous data-flow execution to outer level
- Automatic lookahead



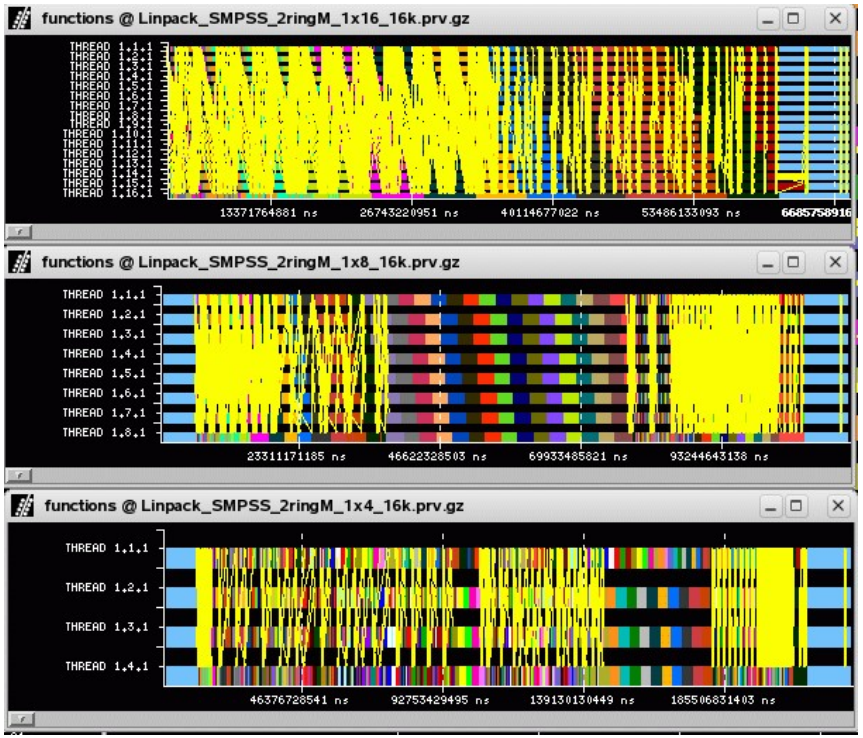

```
…
for (k=0; k<N; k++) {
   if (mine) {
      Factor_panel(A[k]);
      send (A[k])
   } else {
      receive (A[k]);
      if (necessary) resend (A[k]);
   }
   for (j=k+1; j<N; j++)
      update (A[k], A[j]);
…
```

```
#pragma omp task inout([SIZE]A)
void Factor_panel(float *A);
#pragma omp task in([SIZE]A) inout([SIZE]B)
void update(float *A, float *B);
```

```
#pragma omp task in([SIZE]A)
void send(float *A);
#pragma omp task out([SIZE]A)
void receive(float *A);
#pragma omp task in([SIZE]A)
void resend(float *A);
```

Hybrid MPI/OmpSs: Linpack example

- Linpack example
- Overlap communication/computation
- Extend asynchronous data-flow execution to outer level
- Automatic lookahead




```
…
for (k=0; k<N; k++) {
   if (mine) {
      Factor_panel(A[k]);
      send (A[k])
   } else {
      receive (A[k]);
      if (necessary) resend (A[k]);
   }
   for (j=k+1; j<N; j++)
      update (A[k], A[j]);
…
```

```
#pragma omp task inout([SIZE]A)
void Factor_panel(float *A);
#pragma omp task in([SIZE]A) inout([SIZE]B)
void update(float *A, float *B);
```

```
#pragma omp task in([SIZE]A)
void send(float *A);
#pragma omp task out([SIZE]A)
void receive(float *A);
#pragma omp task in([SIZE]A)
void resend(float *A);
```

Executing MPI + OmpSs applications (1)

- Make sure to export environment variables

-x <env>
Export the specified environment variables to the remote nodes before executing the program. Only one environment variable can be specified per -x option. Existing environment variables can be specified or new variable names specified with corresponding values. For example: % mpirun -x DISPLAY -x OMP_NUM_THREADS=16
The parser for the -x option is not very sophisticated; it does not even understand quoted values. Users are advised to set variables in the environment, and then use -x to export (not define) them.

Executing MPI + OmpSs applications (2)

Number of processes and CPUs

-c, -n, --n, -np <#>
Run this many copies of the program on the given nodes.
-cpus-per-proc, --cpus-per-proc <#perproc>
Bind each process to the specified number of cpus. (deprecated in favor of --map-by <obj>:PE=n)

Process binding information

-bind-to-core, --bind-to-core
Bind processes to cores (deprecated in favor of --bind-to core)
-bind-to-socket, --bind-to-socket
Bind processes to processor sockets (deprecated in favor of --bind-to socket)
-bind-to-none, --bind-to-none
Do not bind processes (deprecated in favor of --bind-to none)
--bind-to <foo>
Bind processes to the specified object, defaults to *core*. Supported options include slot, hwthread, core, l1cache, l2cache, l3cache, socket, numa, board, and none.

Executing MPI + OmpSs applications (3)

Displaying MPI deployment (to nodes)

-display-map, --display-map
Display a table showing the mapped location of each process prior to launch.
-display-devel-map, --display-devel-map
Display a more detailed table showing the mapped location of each process prior to launch (usually of interest to developers).
-display-allocation, --display-allocation
Display the detected resource allocation.

Example: 2 CPUs per process, 2 processes

```
$ mpirun -np 2 --cpus-per-proc 2 --display-map a.out
 ========================   JOB MAP   ========================

 Data for node: nvblogin2       Num procs: 2
        Process OMPI jobid: [57024,1] Process rank: 0
        Process OMPI jobid: [57024,1] Process rank: 1

 =============================================================
```

Executing MPI + OmpSs applications (4)

Report binding information (to CPUs)

-report-bindings, --report-bindings
Report any bindings for launched processes.

Example: 3 CPUs per process, bind to core

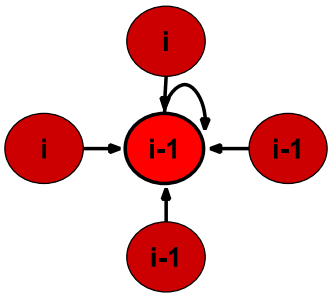
```
$ mpirun -np 4 -cpus-per-proc 3  -bind-to-core --report-bindings a.out
[log1:2773] rank 0 bound to socket 0[core 0-2]: [B B B . . .][. . . . . .]
[log1:2773] rank 1 bound to socket 0[core 3-5]: [. . . B B B][. . . . . .]
[log1:2773] rank 2 bound to socket 1[core 0-2]: [. . . . . .][B B B . . .]
[log1:2773] rank 3 bound to socket 1[core 3-5]: [. . . . . .][. . . B B B]
```

Gauss-Seidel method: Sequential

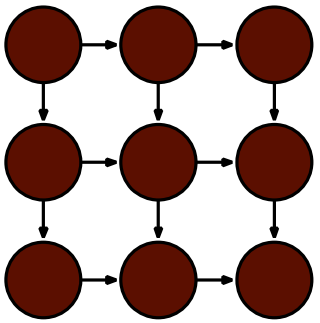
- In-place iterative algorithm
- Ex: 3 x 3 tile domain

0	1	2
3	4	5
6	7	8

- Each tile depend on top and left tile from current iteration and right and bottom tile from previous iteration

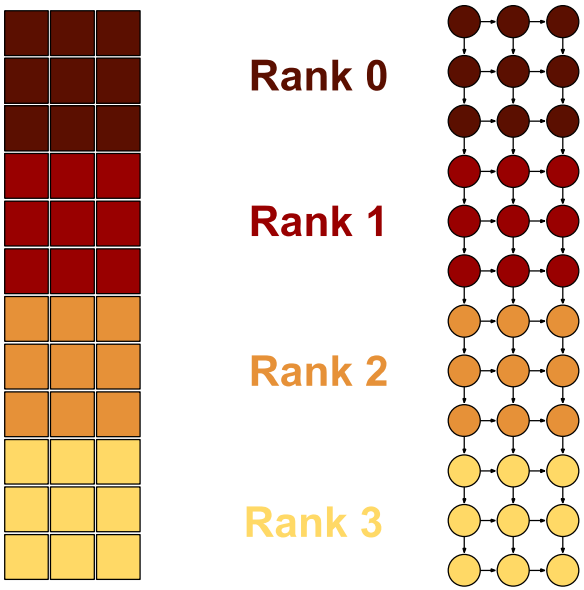


Task that computes a block on the *i-th* iteration



Gauss-Seidel method: Pure MPI

- Ex: 12 x 3 blocks domain, decomposition across 4 MPI ranks

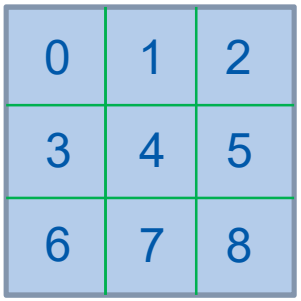


- After each iteration, neighbor MPI ranks has to exchange halos

Gauss-Seidel method: Pure MPI

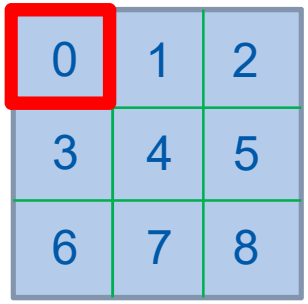
```
void solve(block_t *matrix, int rowBlocks, int colBlocks, int timesteps)
{
    int rank, rank_size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &rank_size);
    for (int t = 0; t < timesteps; ++t) {
        solveGaussSeidel(matrix, rowBlocks, colBlocks, rank, rank_size);
    }
    MPI_Barrier(MPI_COMM_WORLD);
}
```

```
void solveGaussSeidel(block_t *matrix, int nbx, int nby, int rank, int rank_size)
{
    if (rank != 0) {
        sendFirstComputeRow(matrix, nbx, nby, rank, rank_size);
        receiveUpperBorder(matrix, nbx, nby, rank, rank_size);
    }
    if (rank != rank_size - 1) {
        receiveLowerBorder(matrix, nbx, nby, rank, rank_size);
    }
    for (int bx = 1; bx < nbx-1; ++bx) {
        for (int by = 1; by < nby-1; ++by) {
            solveBlock(matrix, nbx, nby, bx, by);
        }
    }
    if (rank != rank_size - 1) {
        sendLastComputeRow(matrix, nbx, nby, rank, rank_size);
    }
}
```



Gauss-Seidel method: Pure MPI

```
void solveBlock(block_t *matrix, int nbx, int nby, int bx, int by)
{
    block_t &targetBlock = matrix[bx*nby + by];
    const block_t &centerBlock = matrix[bx*nby + by];
    const block_t &topBlock    = matrix[(bx-1)*nby + by];

    for (int x = 0; x < BSX; ++x) {
        const row_t &topRow = (x > 0) ? centerBlock[x-1] : topBlock[BSX-1];
        const row_t &bottomRow = (x < BSX-1) ? centerBlock[x+1] : bottomBlock[0];

        for (int y = 0; y < BSY; ++y) {
            double left = (y > 0) ? centerBlock[x][y-1] : leftBlock[x][BSY-1];
            double right = (y < BSY-1) ? centerBlock[x][y+1] : rightBlock[x][0];
            targetBlock[x][y] = 0.25 * (topRow[y] + bottomRow[y] + left + right);
        }
    }
}
```

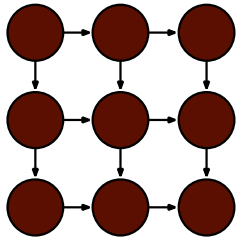


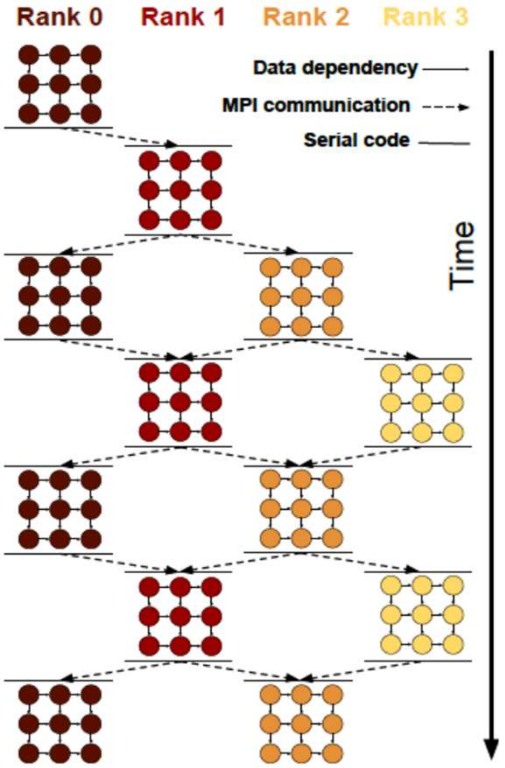

```
void sendLastComputeRow(block_t *matrix, int nbx, int nby, int rank, int rank_size)
{
    for (int by = 1; by < nby-1; ++by) {
        MPI_Send(&matrix[(nbx-2)*nby + by][BSX-1], BSY, MPI_DOUBLE, rank + 1, by, MPI_COMM_WORLD);
    }
}
void receiveUpperBorder(block_t *matrix, int nbx, int nby, int rank, int rank_size)
{
    for (int by = 1; by < nby-1; ++by) {
        MPI_Recv(&matrix[by][BSX-1], BSY, MPI_DOUBLE, rank - 1, by, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
}
```

Gauss-Seidel method: Pure MPI

- Fully sequential execution
- No overlapping of communication an computation phases
- One rank per core

```
void solveGaussSeidel(block_t *matrix, int nbx, int nby, int rank, int rank_size)
{
    if (rank != 0) {
        sendFirstComputeRow(matrix, nbx, nby, rank, rank_size);
        receiveUpperBorder(matrix, nbx, nby, rank, rank_size);
    }
    if (rank != rank_size - 1) {
        receiveLowerBorder(matrix, nbx, nby, rank, rank_size);
    }
    for (int bx = 1; bx < nbx-1; ++bx) {
        for (int by = 1; by < nby-1; ++by) {
            solveBlock(matrix, nbx, nby, bx, by);
        }
    }

    if (rank != rank_size - 1) {
        sendLastComputeRow(matrix, nbx, nby, rank, rank_size);
    }
}
```



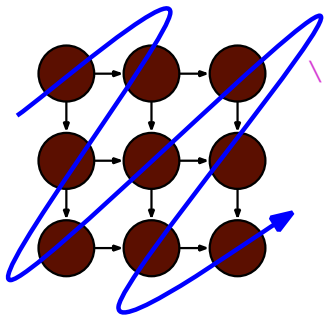


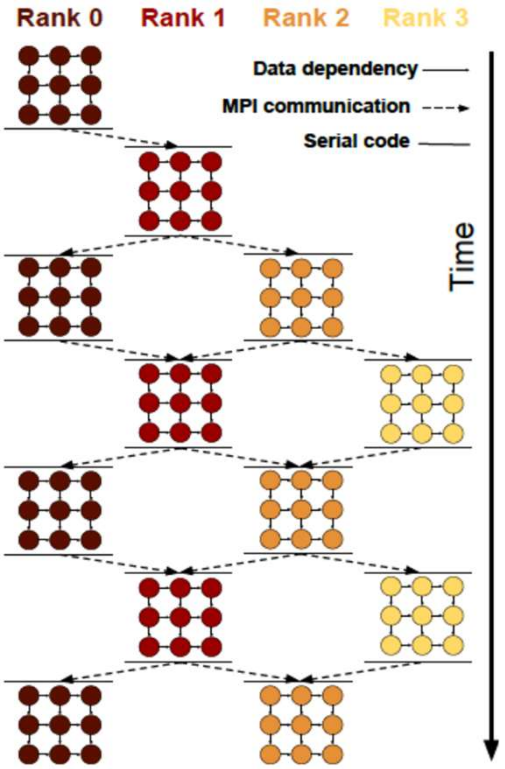
1 Rank x Core!!!

Gauss-Seidel method: Fork-Join

- OmpSs-2 used to execute in parallel the computational phase of the program
- MPI used only on sequential phase of the program for communications
- No overlapping of communication an computation phases

```
void solveGaussSeidel(block_t *matrix, int nbx, int nby, int rank, int rank_size)
{
    if (rank != 0) {
        sendFirstComputeRow(matrix, nbx, nby, rank, rank_size);
        receiveUpperBorder(matrix, nbx, nby, rank, rank_size);
    }
    if (rank != rank_size - 1) {
        receiveLowerBorder(matrix, nbx, nby, rank, rank_size);
    }
    for (int bx = 1; bx < nbx-1; ++bx) {
        for (int by = 1; by < nby-1; ++by) {
                #pragma oss task
                  in(([nbx][nby]matrix)[bx-1][by]) \
                  in(([nbx][nby]matrix)[bx][by-1]) \
                  in(([nbx][nby]matrix)[bx][by+1]) \
                  in(([nbx][nby]matrix)[bx+1][by]) \
                  inout(([nbx][nby]matrix)[bx][by])
            solveBlock(matrix, nbx, nby, bx, by);
        }
    }
    #pragma oss taskwait
    if (rank != rank_size - 1) {
        sendLastComputeRow(matrix, nbx, nby, rank, rank_size);
    }
}
```





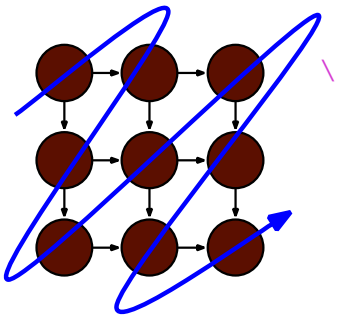
1 Rank x Node!!!

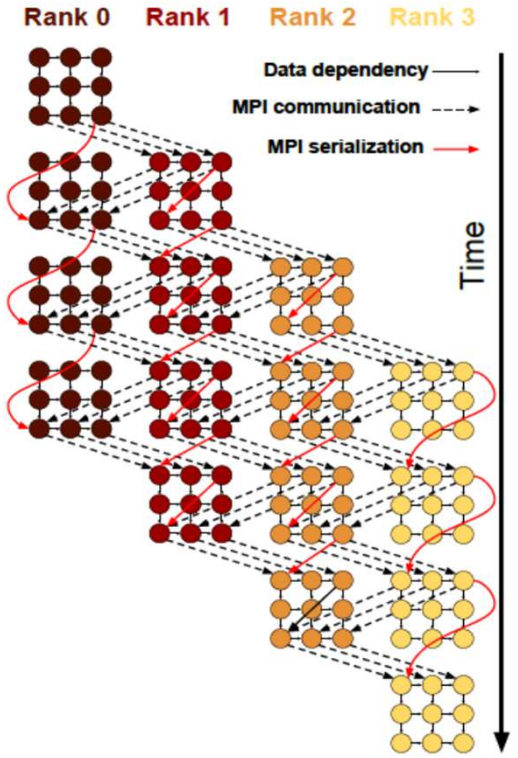
Gauss-Seidel method: Tasks + sentinel

- Tasks used for both computations and communications
- **Tags** used to match send and receive operations but ...
- Communication tasks have to be serialized to avoid **deadlocks**
- Partial overlapping of communication an computation phases

```
void solveGaussSeidel(block_t *matrix, int nbx, int nby, int rank, int rank_size)
{
    if (rank != 0) {
        sendFirstComputeRow(matrix, nbx, nby, rank, rank_size);
        receiveUpperBorder(matrix, nbx, nby, rank, rank_size);
    }
    if (rank != rank_size - 1) {
        receiveLowerBorder(matrix, nbx, nby, rank, rank_size);
    }
    for (int bx = 1; bx < nbx-1; ++bx) {
        for (int by = 1; by < nby-1; ++by) {
                #pragma oss task
                  in(([nbx][nby]matrix)[bx-1][by]) \
                  in(([nbx][nby]matrix)[bx][by-1]) \
                  in(([nbx][nby]matrix)[bx][by+1]) \
                  in(([nbx][nby]matrix)[bx+1][by]) \
                  inout(([nbx][nby]matrix)[bx][by])
            solveBlock(matrix, nbx, nby, bx, by);
        }
    }

    if (rank != rank_size - 1) {
        sendLastComputeRow(matrix, nbx, nby, rank, rank_size);
    }
}
```

No taskwait!





1 Rank x Node!!!

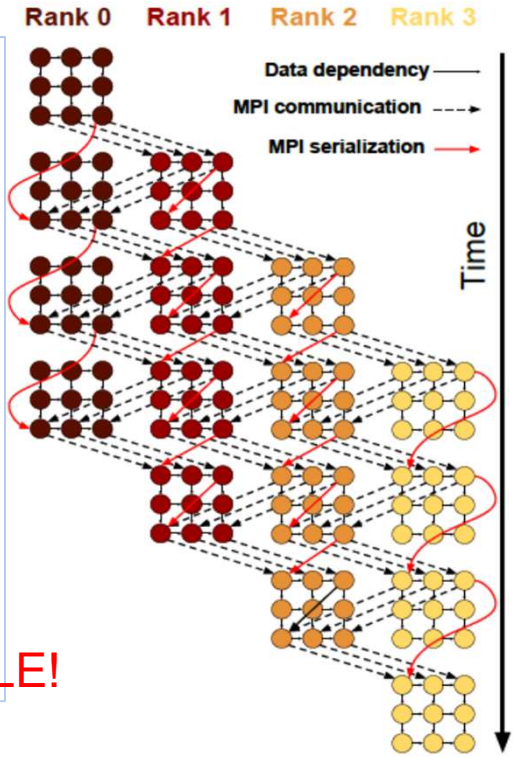
Gauss-Seidel method: Tasks + sentinel

- Tasks used for both computations and communications
- **Tags** used to match send and receive operations but ...
- Communication tasks have to be serialized to avoid **deadlocks**
- Partial overlapping of communication an computation phases

```
void sendLastComputeRow(block_t *matrix, int nbx, int nby,
                                                int rank, int rank_size)
{
    for (int by = 1; by < nby-1; ++by) {
        #pragma oss task in(([nbx][nby]matrix)[nbx-2][by]) inout(*serial)
        MPI_Send(&matrix[(nbx-2)*nby + by][BSX-1], BSY, MPI_DOUBLE, rank + 1,
                                    by, MPI_COMM_WORLD);
    }
}

void receiveUpperBorder(block_t *matrix, int nbx, int nby,
                                               int rank, int rank_size)
{
    for (int by = 1; by < nby-1; ++by) {
        #pragma oss task out(([nbx][nby]matrix)[0][by]) inout(*serial)
        MPI_Recv(&matrix[by][BSX-1], BSY, MPI_DOUBLE, rank - 1, by,
                                    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
}
```

Sentinel to serialize communication task

MPI_THREAD_MULTIPLE!



1 Rank x Node!!!

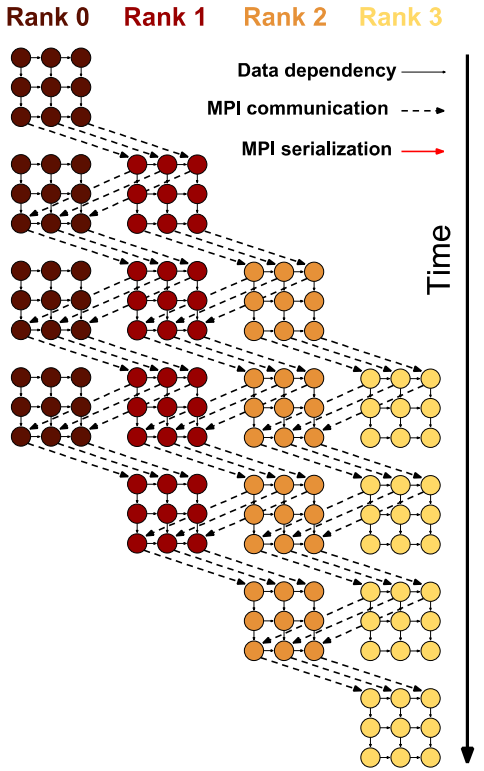
Gauss-Seidel method: Tasks + TAMPI MPI_TASK_MULTIPLE

- Tasks used for both **computations** and **communications**
- **Tags** used to match send and receive operations but ...
- **Full** overlapping of communication an computation phases

```
void sendLastComputeRow(block_t *matrix, int nbx, int nby,
                                                int rank, int rank_size)
{
    for (int by = 1; by < nby-1; ++by) {
        #pragma oss task in(([nbx][nby]matrix)[nbx-2][by])
        MPI_Send(&matrix[(nbx-2)*nby + by][BSX-1], BSY, MPI_DOUBLE, rank + 1,
                                    by, MPI_COMM_WORLD);
    }
}

void receiveUpperBorder(block_t *matrix, int nbx, int nby,
                                               int rank, int rank_size)
{
    for (int by = 1; by < nby-1; ++by) {
        #pragma oss task out(([nbx][nby]matrix)[0][by])
        MPI_Recv(&matrix[by][BSX-1], BSY, MPI_DOUBLE, rank - 1, by,
                                    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
}
```

No sentinel needed!

MPI_**TASK**_MULTIPLE!



1 Rank x Node!!!

Gauss-Seidel method: Tasks + TAMPI TAMPI_Iwait()

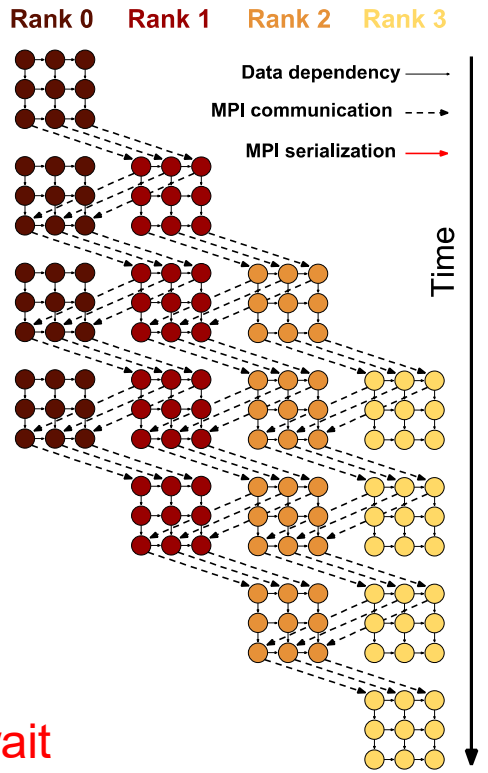
- Tasks used for both **computations** and **communications**
- **Tags** used to match send and receive operations but ...
- **Full** overlapping of communication an computation phases

```
void sendLastComputeRow(block_t *matrix, int nbx, int nby,
                                                int rank, int rank_size)
{
    for (int by = 1; by < nby-1; ++by) {
        #pragma oss task in(([nbx][nby]matrix)[nbx-2][by])
        MPI_Send(&matrix[(nbx-2)*nby + by][BSX-1], BSY, MPI_DOUBLE, rank + 1,
                                    by, MPI_COMM_WORLD);
    }
}
}
```

MPI_TASK_MULTIPLE

```
void sendLastComputeRow(block_t *matrix, int nbx, int nby,
                                                int rank, int rank_size)
{
    for (int by = 1; by < nby-1; ++by) {
        #pragma oss task in(([nbx][nby]matrix)[nbx-2][by])
                                    {
          MPI_Request request;
          MPI_Isend(&matrix[(nbx-2)*nby + by][BSX-1], BSY, MPI_DOUBLE,
                                rank + 1, by, MPI_COMM_WORLD, &request);
          TAMPI_Iwait(&request, MPI_STATUS_IGNORE);
       }
    }
}
```

MPI_TASK_MULTIPLE+TAMPI_Iwait

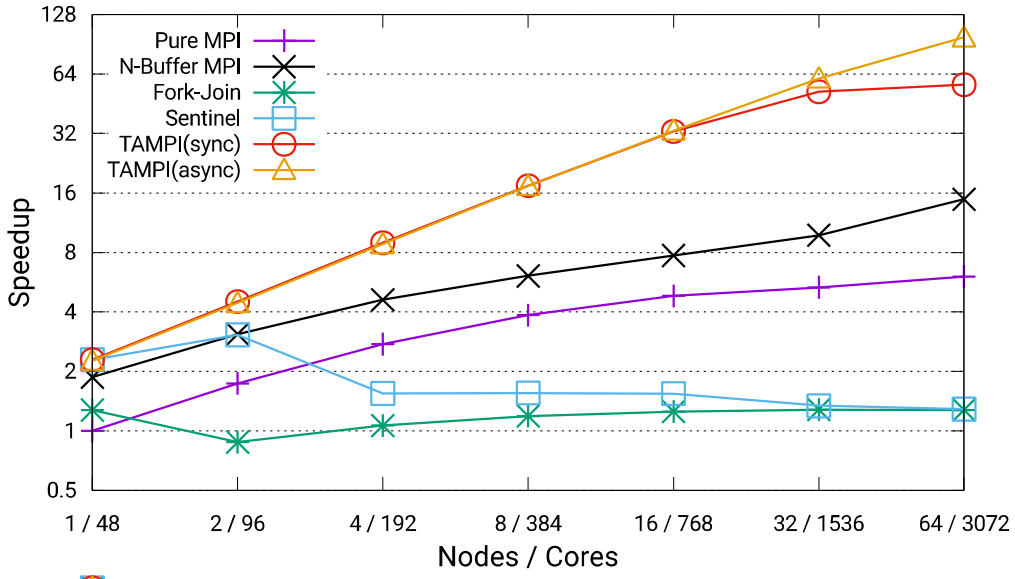


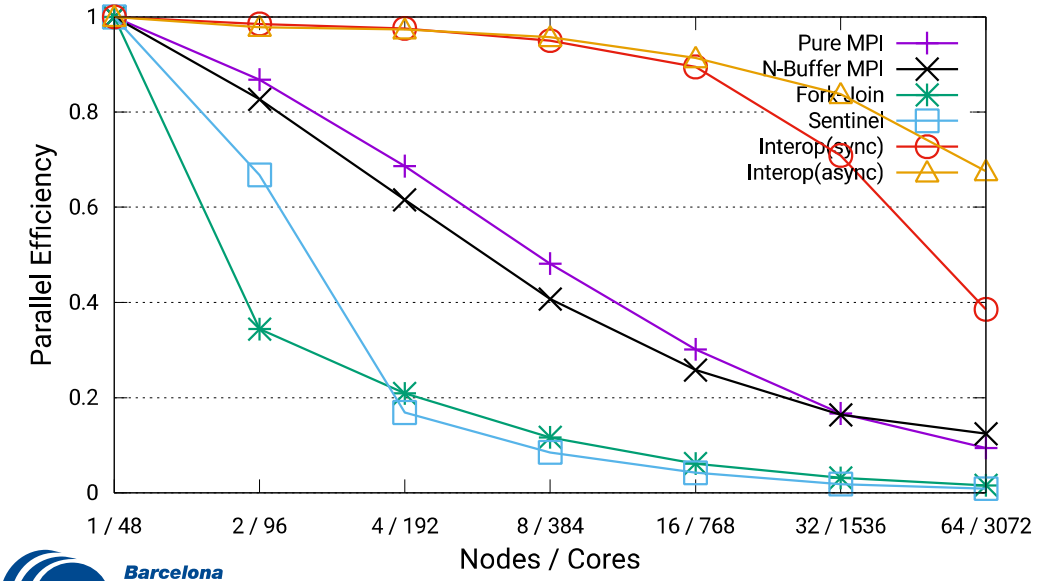
1 Rank x Node!!!

VB1

VB7

Tiled Gauss-Seidel(64Kx64K): Results





- BS=512/1024 & 1000 iterations
- Pure MPI: 48 ranks/node
- 1 rank/node & 48 cores/rank
- **Baseline: Pure MPI**

- BS=512/1024 & 1000 iterations
- Pure MPI: 48 ranks/node
- 1 rank/node & 48 cores/rank
- **Baseline: results with one node**

Slide 38

VB1 Update results and remove interop(mpich) interop(poolong). Change interop(lib) by TAMPI(sync) and TAMPI(async)
Vicenç Beltran, 9/18/2018

VB7 Vicenç Beltran, 9/18/2018

www.bsc.es

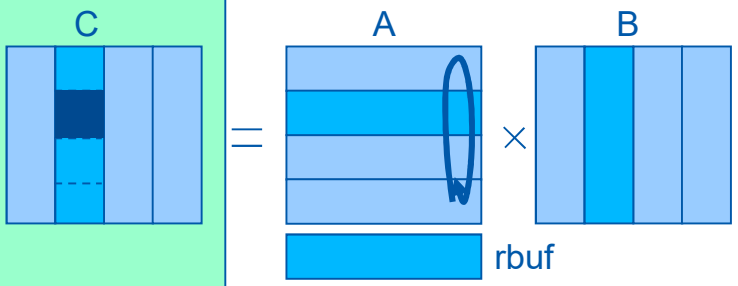
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Thank you!

For further information please contact
xavier.martorell@bsc.es

MxM @ MPI

```
// m: matriz size; n: local row/columns

double A[n][m], B[m][n], C[m][n], *a, *rbuf;

orig_rbuf = rbuf = (double (*)[m])malloc(m*n*sizeof(double));
if (nodes >1) { up=me<nodes-1 ? me+1:0; down=me>0 ? me-1:nodes-1;}
else { up = down = MPI_PROC_NULL; }

a=A;
i = n*me;  // first C block (different for each process)

for( it=0; it<nodes; it++ ) {

   mxm( m, n, a, B, (double (*)[n])&C[i][0]);

   callSendRecv(m, n, a, down, rbuf, up);

   //next C block circular
   i = (i+n)%m;
   //swap pointers
   ptmp=a; a=rbuf; rbuf=ptmp;

}


free (orig_rbuf);
```

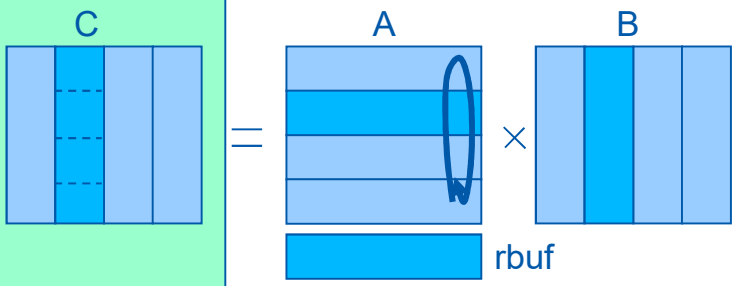



```
void mxm ( int m, int n, double *a, double *b, double *c ) {
   double alpha=1.0, beta=1.0;
   int i, j;
   char tr = 't';
   dgemm(&tr, &tr, &m, &n, &n, &alpha, a, &m, b, &n, &beta, c, &n);
}
```

```
void callSendRecv(int m, int n,
                  double (*a)[m], int down,
                  double (*rbuf)[m], int up)
{
   int tag = 1000;
   int size = m*n;
   MPI_Status stats;

   MPI_Sendrecv( a, size, MPI_DOUBLE, down, tag,
                 rbuf, size, MPI_DOUBLE, up, tag,
                 MPI_COMM_WORLD, &stats);
}
```

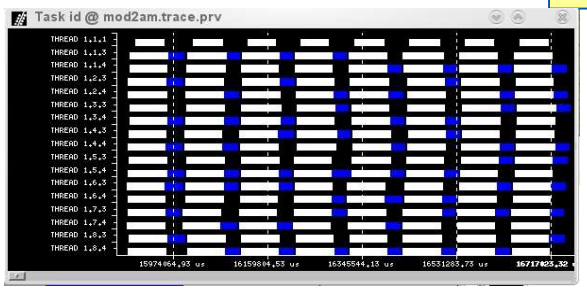
MxM @ MPI + OmpSs: MPI calls not taskified

```
// m: matriz size; n: local row/columns

double A[n][m], B[m][n], C[m][n], *a, *rbuf;

orig_rbuf = rbuf = (double (*)[m])malloc(m*n*sizeof(double));
if (nodes >1) { up=me<nodes-1 ? me+1:0; down=me>0 ? me-1:nodes-1;}
else { up = down = MPI_PROC_NULL; }

a=A;
i = n*me;  // first C block (different for each process)

for( it=0; it<nodes; it++ ) {

   mxm( m, n, a, B, (double (*)[n])&C[i][0]);

   callSendRecv(m, n, a, down, rbuf, up);

   //next C block circular
   i = (i+n)%m;
   //swap pointers
   ptmp=a; a=rbuf; rbuf=ptmp;

}


free (orig_rbuf);
```



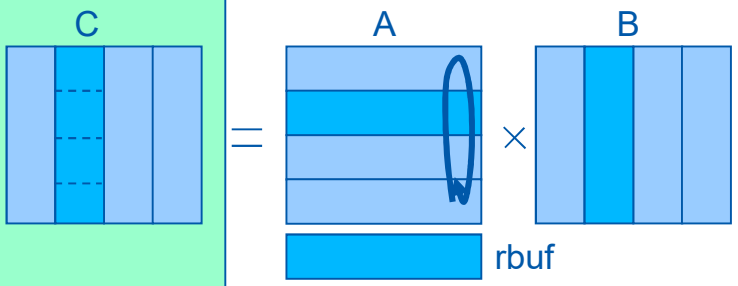

```
void mxm ( int m, int n, double *a, double *b, double *c ) {
   double alpha=1.0, beta=1.0;
   int i, j, l=1;
   char tr = 't';
   for (i=0; i<n; i++) {
      #pragma omp task
      dgemm(&tr, &tr, &m, &n, &l, &alpha, a, &m, b, &n, &beta, c, &n);
      c+=n;
      a+=m;
      b+=n;
   }
   #pragma omp taskwait
}
```

```
void callSendRecv(int m, int n,
                  double (*a)[m], int down,
                  double (*rbuf)[m], int up)
{
   int tag = 1000;
   int size = m*n;
   MPI_Status stats;

   MPI_Sendrecv( a, size, MPI_DOUBLE, down, tag,
                 rbuf, size, MPI_DOUBLE, up, tag,
                 MPI_COMM_WORLD, &stats);
}
```



MxM @ MPI + OmpSs: MPI calls not taskified

```
// m: matriz size; n: local row/columns

double A[n][m], B[m][n], C[m][n], *a, *rbuf;

orig_rbuf = rbuf = (double (*)[m])malloc(m*n*sizeof(double));
if (nodes >1) { up=me<nodes-1 ? me+1:0; down=me>0 ? me-1:nodes-1;}
else { up = down = MPI_PROC_NULL; }

a=A;
i = n*me;  // first C block (different for each process)

for( it=0; it<nodes; it++ ) {

   mxm( m, n, a, B, (double (*)[n])&C[i][0]);

   callSendRecv(m, n, a, down, rbuf, up);

   //next C block circular
   i = (i+n)%m;
   //swap pointers
   ptmp=a; a=rbuf; rbuf=ptmp;
   #pragma omp taskwait
}


free (orig_rbuf);
```



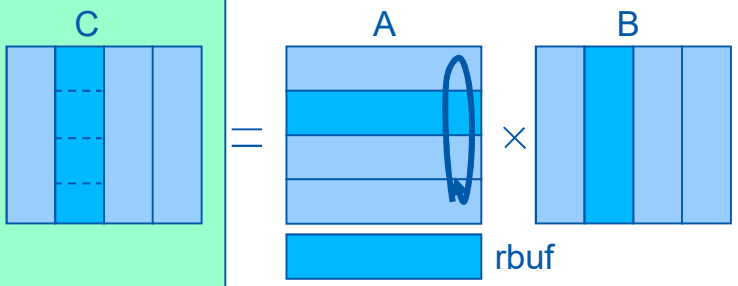

```
void mxm ( int m, int n, double *a, double *b, double *c ) {
   double alpha=1.0, beta=1.0;
   int i, j, l=1;
   char tr = 't';
   for (i=0; i<n; i++) {
      #pragma omp task
      dgemm(&tr, &tr, &m, &n, &l, &alpha, a, &m, b, &n, &beta, c, &n);
      c+=n;
      a+=m;
      b+=n;
   }

}
```

```
void callSendRecv(int m, int n,
                  double (*a)[m], int down,
                  double (*rbuf)[m], int up)
{
   int tag = 1000;
   int size = m*n;
   MPI_Status stats;

   MPI_Sendrecv( a, size, MPI_DOUBLE, down, tag,
                 rbuf, size, MPI_DOUBLE, up, tag,
                 MPI_COMM_WORLD, &stats);
}
```

- Overlap computation in tasks and communication in master

MxM @ MPI + OmpSs: MPI calls taskified

```
// m: matriz size; n: local row/columns

double A[n][m], B[m][n], C[m][n], *a, *rbuf;

orig_rbuf = rbuf = (double (*)[m])malloc(m*n*sizeof(double));
if (nodes >1) { up=me<nodes-1 ? me+1:0; down=me>0 ? me-1:nodes-1;}
else { up = down = MPI_PROC_NULL; }

a=A;
i = n*me;  // first C block (different for each process)

for( it=0; it<nodes; it++ ) {

   mxm( m, n, a, B, (double (*)[n])&C[i][0]);
   #pragma omp task in([n][m]a) out([n][m]rbuf)
   callSendRecv(m, n, a, down, rbuf, up);

   //next C block circular
   i = (i+n)%m;
   //swap pointers
   ptmp=a; a=rbuf; rbuf=ptmp;
   #pragma omp taskwait
}


free (orig_rbuf);
```



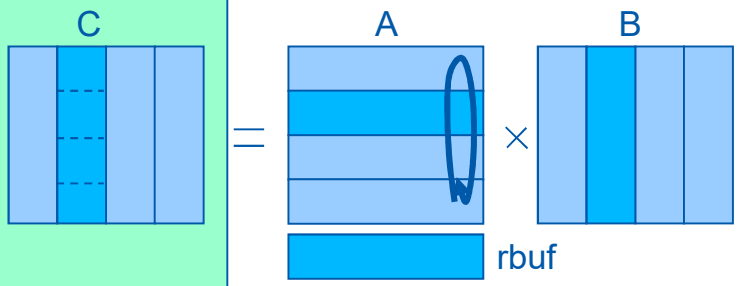

```
void mxm ( int m, int n, double *a, double *b, double *c ) {
   double alpha=1.0, beta=1.0;
   int i, j, l=1;
   char tr = 't';
   for (i=0; i<n; i++) {
      #pragma omp task
      dgemm(&tr, &tr, &m, &n, &l, &alpha, a, &m, b, &n, &beta, c, &n);
      c+=n;
      a+=m;
      b+=n;
   }

}
```

```
void callSendRecv(int m, int n,
                  double (*a)[m], int down,
                  double (*rbuf)[m], int up)
{
   int tag = 1000;
   int size = m*n;
   MPI_Status stats;

   MPI_Sendrecv( a, size, MPI_DOUBLE, down, tag,
                 rbuf, size, MPI_DOUBLE, up, tag,
                 MPI_COMM_WORLD, &stats);
}
```

- Overlap computation in tasks and communication in task

MxM @ MPI + OmpSs: MPI calls taskified

```
// m: matriz size; n: local row/columns

double A[n][m], B[m][n], C[m][n], *a, *rbuf;

orig_rbuf = rbuf = (double (*)[m])malloc(m*n*sizeof(double));
if (nodes >1) { up=me<nodes-1 ? me+1:0; down=me>0 ? me-1:nodes-1;}
else { up = down = MPI_PROC_NULL; }

a=A;
i = n*me;  // first C block (different for each process)

for( it=0; it<nodes; it++ ) {
   #pragma omp task in( [n][m]a, B ) inout ( C[i;n][0;n] )
   mxm( m, n, a, B, (double (*)[n])&C[i][0]);
   #pragma omp task in([n][m]a) out([n][m]rbuf)
   callSendRecv(m, n, a, down, rbuf, up);

   //next C block circular
   i = (i+n)%m;
   //swap pointers
   ptmp=a; a=rbuf; rbuf=ptmp;
   #pragma omp taskwait
}


free (orig_rbuf);
```



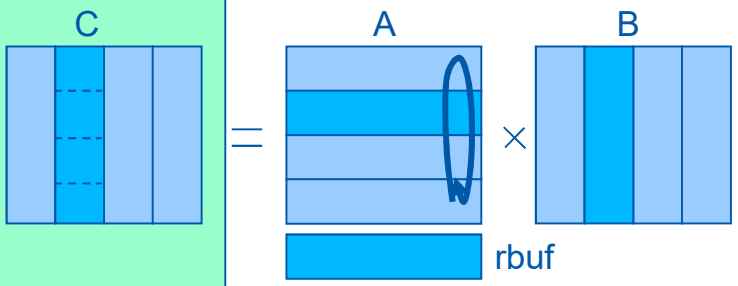

```
void mxm ( int m, int n, double *a, double *b, double *c ) {
   double alpha=1.0, beta=1.0;
   int i, j, l=1;
   char tr = 't';
   for (i=0; i<n; i++) {
      #pragma omp task
      dgemm(&tr, &tr, &m, &n, &l, &alpha, a, &m, b, &n, &beta, c, &n);
      c+=n;
      a+=m;
      b+=n;
   }
   #pragma omp taskwait
}
```

```
void callSendRecv(int m, int n,
                  double (*a)[m], int down,
                  double (*rbuf)[m], int up)
{
   int tag = 1000;
   int size = m*n;
   MPI_Status stats;

   MPI_Sendrecv( a, size, MPI_DOUBLE, down, tag,
                 rbuf, size, MPI_DOUBLE, up, tag,
                 MPI_COMM_WORLD, &stats);
}
```

- Nested. Overlap between computation and communication in tasks

MxM @ MPI + OmpSs: MPI calls taskified

```
// m: matriz size; n: local row/columns

double A[n][m], B[m][n], C[m][n], *a, *rbuf;

orig_rbuf = rbuf = (double (*)[m])malloc(m*n*sizeof(double));
if (nodes >1) { up=me<nodes-1 ? me+1:0; down=me>0 ? me-1:nodes-1;}
else { up = down = MPI_PROC_NULL; }

a=A;
i = n*me;  // first C block (different for each process)

for( it=0; it<nodes; it++ ) {
   #pragma omp task in( [n][m]a, B ) inout ( C[i;n][0;n] )
   mxm( m, n, a, B, (double (*)[n])&C[i][0]);
   #pragma omp task in([n][m]a) out([n][m]rbuf)
   callSendRecv(m, n, a, down, rbuf, up);

   //next C block circular
   i = (i+n)%m;
   //swap pointers
   ptmp=a; a=rbuf; rbuf=ptmp;

}
#pragma omp taskwait

free (orig_rbuf);
```



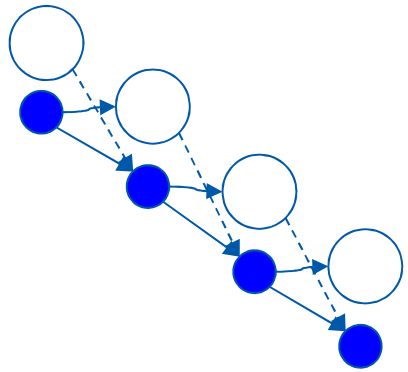

```
void mxm ( int m, int n, double *a, double *b, double *c ) {
   double alpha=1.0, beta=1.0;
   int i, j, l=1;
   char tr = 't';
   for (i=0; i<n; i++) {
      #pragma omp task
      dgemm(&tr, &tr, &m, &n, &l, &alpha, a, &m, b, &n, &beta, c, &n);
      c+=n;
      a+=m;
      b+=n;
   }
   #pragma omp taskwait
}
```

```
void callSendRecv(int m, int n,
                  double (*a)[m], int down,
                  double (*rbuf)[m], int up)
{
   int tag = 1000;
   int size = m*n;
   MPI_Status stats;

   MPI_Sendrecv( a, size, MPI_DOUBLE, down, tag,
                 rbuf, size, MPI_DOUBLE, up, tag,
                 MPI_COMM_WORLD, &stats);
}
```

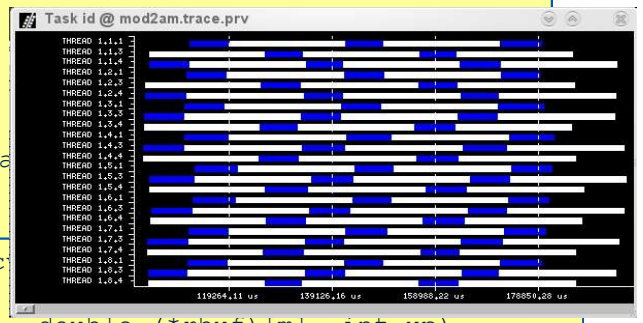
- Can start computation of next block of C as soon as communication terminates

MxM @ MPI + OmpSs: MPI calls taskified

```
// m: matriz size; n: local row/columns

double A[n][m], B[m][n], C[m][n], *a, *rbuf;

orig_rbuf = rbuf = (double (*)[m])malloc(m*n*sizeof(double));
if (nodes >1) { up=me<nodes-1 ? me+1:0; down=me>0 ? me-1:nodes-1;}
else { up = down = MPI_PROC_NULL; }

a=A;
i = n*me;  // first C block (different for each process)

for( it=0; it<nodes; it++ ) {
   #pragma omp task in( [n][m]a, B ) inout ( C[i;n][0;n] ) label (white)
   mxm( m, n, a, B, (double (*)[n])&C[i][0]);
   #pragma omp task in([n][m]a) out([n][m]rbuf) label (blue)
   callSendRecv(m, n, a, down, rbuf, up);

   //next C block circular
   i = (i+n)%m;
   //swap pointers
   ptmp=a; a=rbuf; rbuf=ptmp;

}
#pragma omp taskwait

free (orig_rbuf);
```



```
void mxm ( int m, int n, double *a, double *b, double *c ) {
   double alpha=1.0, beta=1.0;
   int i, j, l=1;
   char tr = 't';
   for (i=0; i<n; i++) {

      dgemm(&tr, &tr, &m, &n, &l, &alpha, a
      c+=n;
      a+=m;
      b+=n;
   }

}
```




```
void callSendRec
                                  double (*rbuf)[m], int up)
{
   int tag = 1000;
   int size = m*n;
   MPI_Status stats;

   MPI_Sendrecv( a, size, MPI_DOUBLE, down, tag,
                 rbuf, size, MPI_DOUBLE, up, tag,
                 MPI_COMM_WORLD, &stats);
}
```

- Can obtain parallelism even without parallelizing the computation