www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Day 4, Session 2: Hybrid MPI + OpenMP

Rosa Badia, Xavier Martorell

Session 4.2: Hybrid MPI + OpenMP

- Hybrid programming
  - Issues and basic examples

Hybrid MPI/OpenMP

MPI + OpenMP 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

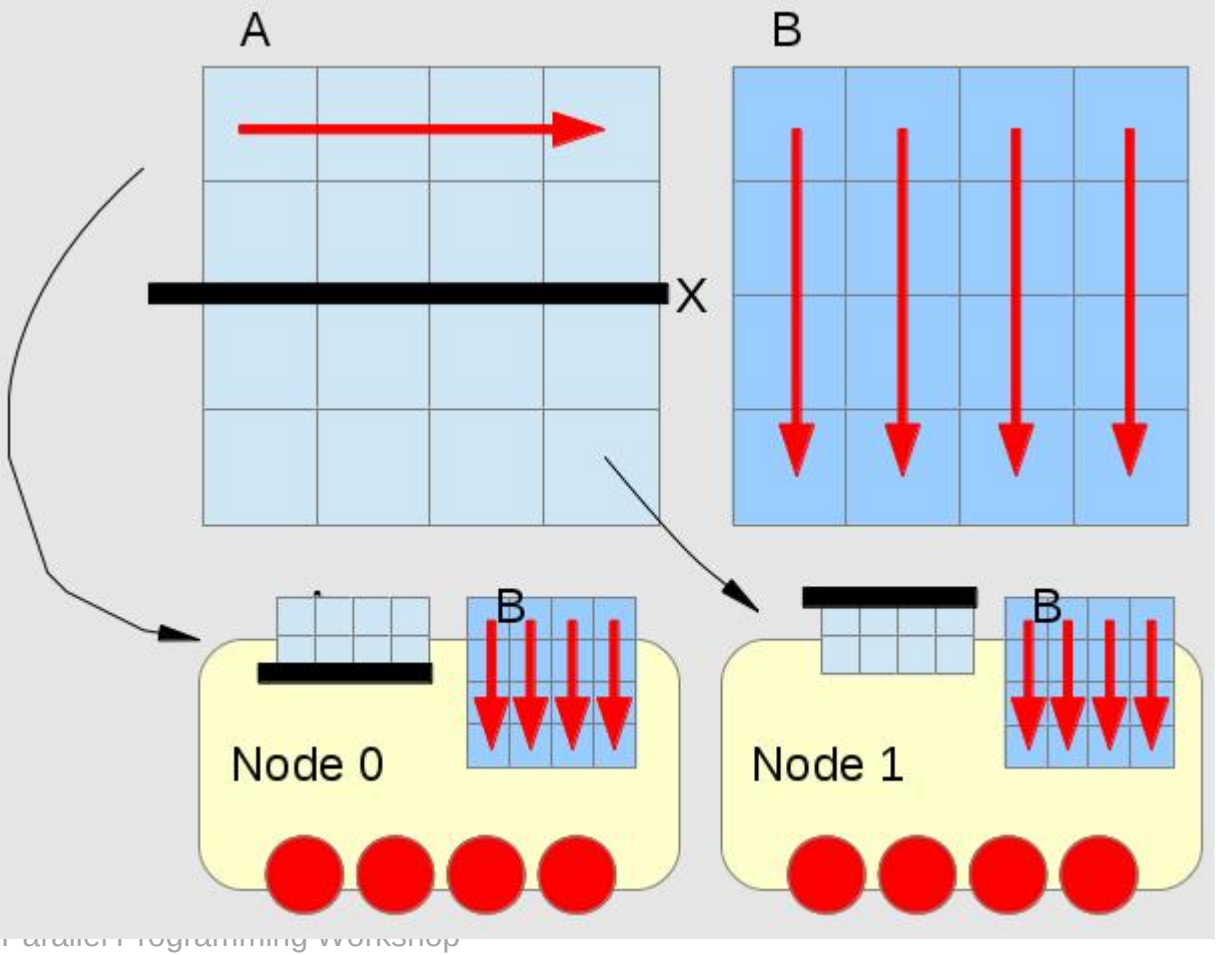
MPI + OpenMP hybrid programming

- 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

MPI + OpenMP hybrid programming

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

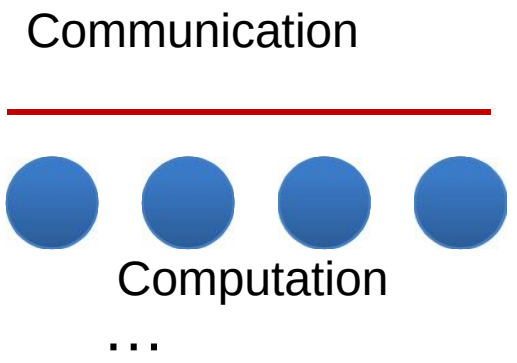


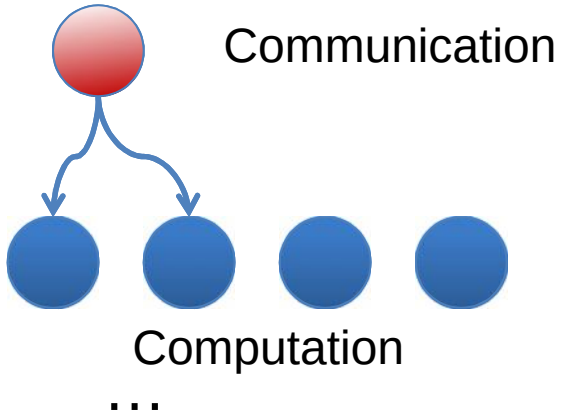
Opportunities for MPI+OpenMP

- When to use MPI+OpenMP
  - Starting from OpenMP, and moving to clusters with MPI
  - Starting from an MPI application, and exploiting further parallelism inside each node
    - In this case the parallelism can also be exploited with MPI
- Improvements
  - OpenMP can mitigate/solve MPI imbalance

Interaction between MPI and StarSs 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





Taskifying communications

- Communication Tasks:
  - Single MPI communication
    - Could be done just modifying mpi.h

```
#pragma omp task depend(in: buffer[0;N])
int mpi_send(... void * 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

Taskifying communications

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

Deadlock potential when taskifying communications

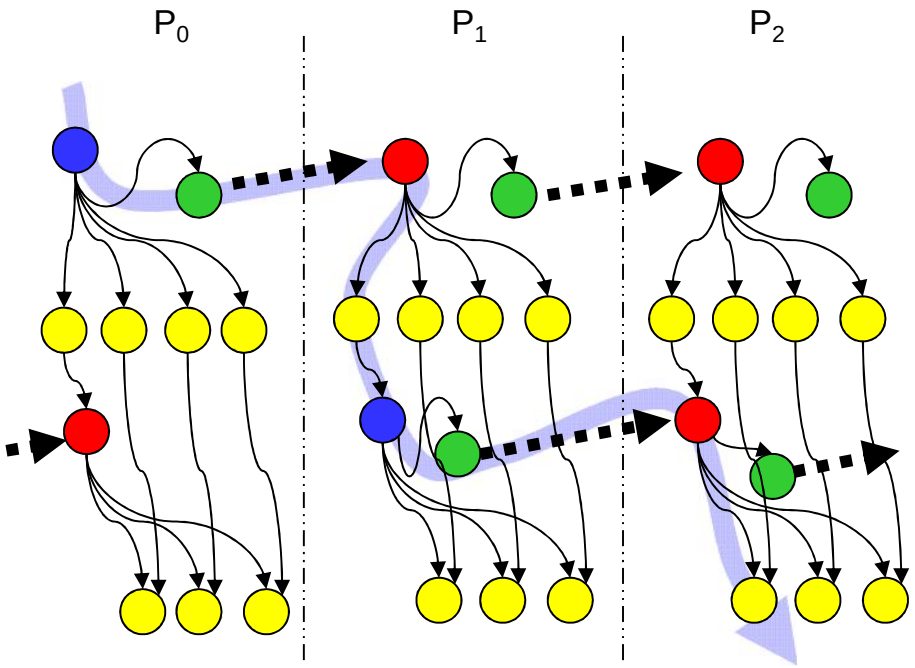
- Approaches to handle
  - Algorithmic structure may not expose loop and thus be deadlock safe
  - Enforce “in order” execution of communicating tasks
    - In the source code, possibly using a single sentinel specified as inout for all communication tasks
    - Specialized logical device (implemented in SMPSs, not in OmpSs)

      #target device (COMM_THREAD)

      - Modification in runtime scheduler to execute tasks allocated to it in instantiation order
    - Serialization of communication can have an impact on performance
  - Virtualize communication resource. Allow infinite communication tasks
    - Integrated implementation of MPI interface in the OmpSs runtime
    - Implement blocking MPI calls by issuing nonblocking MPI calls blocking the task in NANOS and reactivate then when communication completes

Hybrid MPI/OmpSs: Linpack example

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



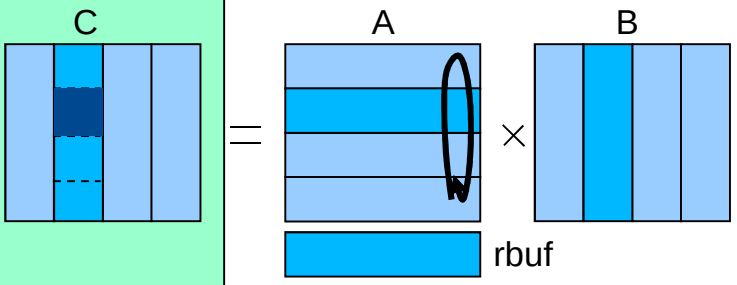

```
…
for (k=0; k<N; k++) {
   if (mine) {
      Factor_panel(A[k]);
      send (A[k])
   } else {
      receive (A[k]);
      if (necessary) resend (A[k]);
   }
   for (j=k+1; j<N; j++)
      update (A[k], A[j]);
…
```

```
#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);
```

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);
```

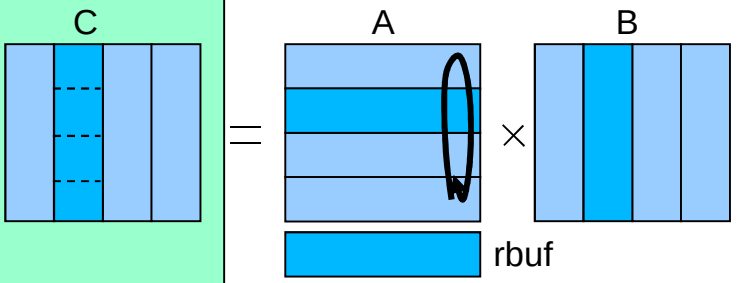



```
void mxm ( int m, int n, double *a, double *b, double *c ) {
   double alpha=1.0, beta=1.0;
   int i, j;
   char tr = 't';
   dgemm(&tr, &tr, &m, &n, &n, &alpha, a, &m, b, &n, &beta, c, &n);
}
```

```
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);
}
```

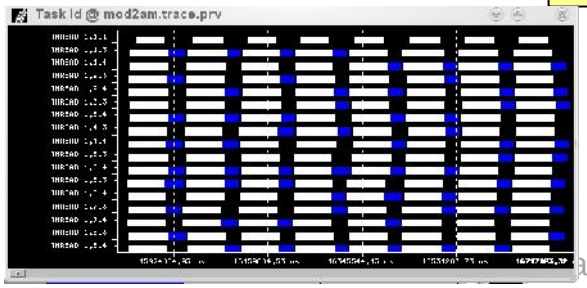
MxM @ MPI + OmpSs: MPI calls not taskified

```
// 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);
```



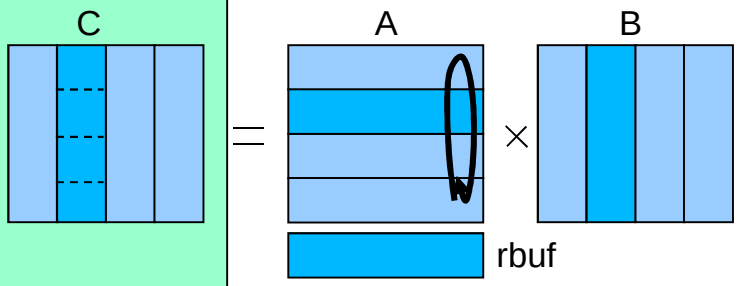

```
void mxm ( int m, int n, double *a, double *b, double *c ) {
   double alpha=1.0, beta=1.0;
   int i, j, l=1;
   char tr = 't';
   for (i=0; i<n; i++) {
      #pragma omp task
      dgemm(&tr, &tr, &m, &n, &l, &alpha, a, &m, b, &n, &beta, c, &n);
      c+=n;
      a+=m;
      b+=n;
   }
   #pragma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



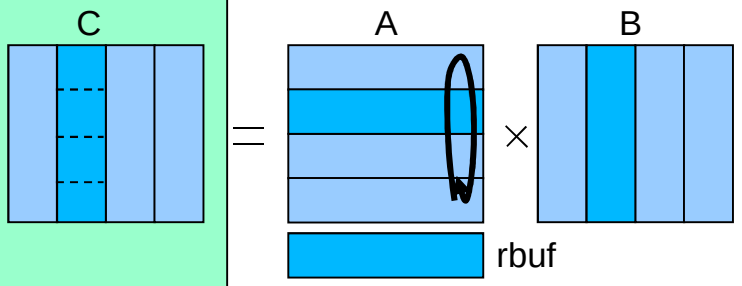

```
void mxm ( int m, int n, double *a, double *b, double *c ) {
   double alpha=1.0, beta=1.0;
   int i, j, l=1;
   char tr = 't';
   for (i=0; i<n; i++) {
      #pragma omp task
      dgemm(&tr, &tr, &m, &n, &l, &alpha, a, &m, b, &n, &beta, c, &n);
      c+=n;
      a+=m;
      b+=n;
   }

}
```

```
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);
```



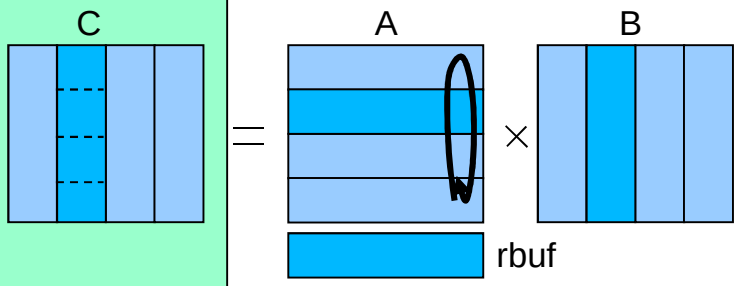

```
void mxm ( int m, int n, double *a, double *b, double *c ) {
   double alpha=1.0, beta=1.0;
   int i, j, l=1;
   char tr = 't';
   for (i=0; i<n; i++) {
      #pragma omp task
      dgemm(&tr, &tr, &m, &n, &l, &alpha, a, &m, b, &n, &beta, c, &n);
      c+=n;
      a+=m;
      b+=n;
   }

}
```

```
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);
```



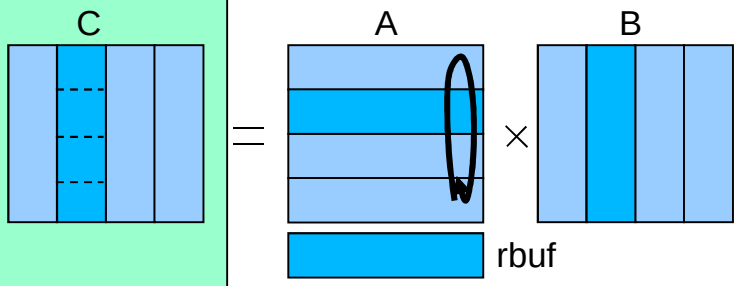

```
void mxm ( int m, int n, double *a, double *b, double *c ) {
   double alpha=1.0, beta=1.0;
   int i, j, l=1;
   char tr = 't';
   for (i=0; i<n; i++) {
      #pragma omp task
      dgemm(&tr, &tr, &m, &n, &l, &alpha, a, &m, b, &n, &beta, c, &n);
      c+=n;
      a+=m;
      b+=n;
   }
   #pragma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



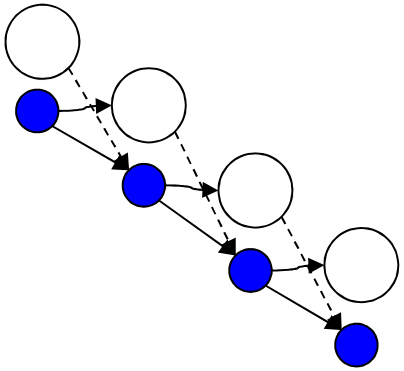

```
void mxm ( int m, int n, double *a, double *b, double *c ) {
   double alpha=1.0, beta=1.0;
   int i, j, l=1;
   char tr = 't';
   for (i=0; i<n; i++) {
      #pragma omp task
      dgemm(&tr, &tr, &m, &n, &l, &alpha, a, &m, b, &n, &beta, c, &n);
      c+=n;
      a+=m;
      b+=n;
   }
   #pragma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

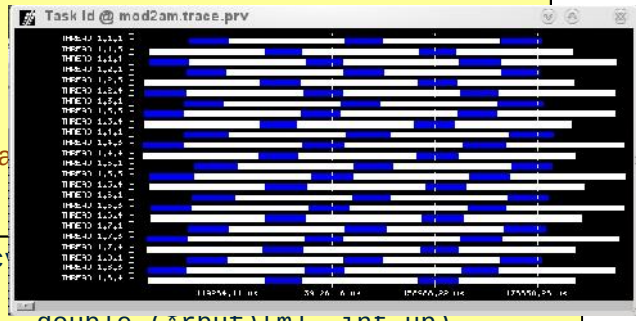
- Can start computation of next block of C as soon as communication terminates

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


BSC
Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación
THANKS