## Agenda

<table>
<thead>
<tr>
<th>Time</th>
<th>Session</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>10:00 – 11:00</td>
<td>Tasking in OpenMP 3.0 and 4.0</td>
<td>60 min</td>
</tr>
<tr>
<td>11:00 – 11:15</td>
<td>Coffee break</td>
<td>15 min</td>
</tr>
<tr>
<td>11:15 – 12:15</td>
<td>Introduction to OmpSs programming model</td>
<td>60 min</td>
</tr>
<tr>
<td></td>
<td>• Introduction to StarSs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• OmpSs syntax</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Simple examples</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Development methodology and infrastructure</td>
<td></td>
</tr>
<tr>
<td>12:15 – 12:45</td>
<td>Practical: heat equation example and divide-and-conquer (part I)</td>
<td>30 min</td>
</tr>
<tr>
<td>12:45 – 14:00</td>
<td>Lunch</td>
<td>75 min</td>
</tr>
<tr>
<td>14:00 – 15:00</td>
<td>Practical: heat equation example and divide-and-conquer (part I)</td>
<td>90 min</td>
</tr>
<tr>
<td>15:00 – 15:30</td>
<td>Programming using a hybrid MPI/OmpSs approach</td>
<td>15 min</td>
</tr>
<tr>
<td>15:30 – 16:35</td>
<td>Practical: heat equation example and divide-and-conquer (part II)</td>
<td>105 min</td>
</tr>
</tbody>
</table>
Agenda

- Motivation for a hybrid model
- Overlapping communication with computation
- Examples
- Dynamic Load Balancing

Contact: pm-tools@bsc.es
Source code available from http://pm.bsc.es/ompss/
MPI/StarSs 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 + OmpSs
- … propagates OmpSs features to global program behavior
- Potential for automatic overlap of communication and computation through the inclusion of communication in the task graph
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

- Burst of MPI communications (+ some small but potentially critical computation between them)
  - Less overhead
  - Typically the interprocess 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
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/StarSs

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

```c
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 in([SIZE] A)
void send(float *A);
#pragma omp task out([SIZE] A)
void receive(float *A);
#pragma omp task inout([SIZE] A)
void Factor_panel(float *A);
#pragma omp task inout([SIZE] A) inout([SIZE] B)
void update(float *A, float *B);
```
Hybrid MPI/OmpSs

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

```c
for (k=0; k<N; k++) {
    if (mine) {
        #pragma omp task inout([SIZE]A)
        void Factor_panel(float *A);
        send (A[k])
    } else {
        receive (A[k]);
        if (necessary) resend (A[k]);
    }
    for (j=k+1; j<N; j++)
        update (A[k], A[j]);
}
```

```c
#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);
```
// 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 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);
}

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);
}
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);
}

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);
}
Overlap computation in tasks and communication in master

```c
// m: matrix 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);
```

```c
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;
    }
}
```

```c
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

```c
// 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);
}
```
Nested. Overlap between computation and communication in tasks

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.
Can obtain parallelism even without parallelizing the computation

Can obtain parallelism even without parallelizing the computation

```
// 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);
}

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, &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);
Cholesky factorization

- Example of the issues in porting legacy code
- Demonstration that it is feasible
- Synchronization tasks to emulate array sections behavior
  - Overhead more than compensated by flexibility
- The importance of scheduling
  - ~ 10% difference in global performance
- Some difficulties with legacy codes
  - Structure of sequential code
  - Memory allocation
UniMAN

Dense linear algebra for multicore (QR and LU)

Scheduling of computations and communications
- Impact of serialization in communication thread

Potential to integrate libraries and application parallelism
Dynamic Load Balancing: LeWI

- Automatically achieved by the runtime
  - Load balance within node
  - Fine grain
  - Complementary to user level load balance
  - Leverage OmpSs malleability

LeWI: Lend core When Idle
- User level Run time Library (DLB) coordinating processes within node
- Lend cores to other processes in the node when entering blocking MPI calls
- Retrieve when leaving blocking MPI
- Fighting the Linux kernel: Explicit pinning of threads to cores

“LeWI: A Runtime Balancing Algorithm for Nested Parallelism”. M.Garcia et al. ICPP09
**LeWI**: Lend When Idle

- An MPI process lends its cpus when inside a blocking MPI call.
- Another MPI process in the same node can use the lend cpus to run with more threads.
- When the MPI call is finished the MPI process retrieves it’s cpus.

- Can handle irregular imbalance

- Its performance depends on the malleability of the second level of parallelism...
LeWI - Malleability

- **Unbalanced application**
  - Application
  - Node
    - MPI 0
    - MPI 1
  - MPI call

- **Execution with DLB MPI+OpenMP**
  - Application
  - Node
    - MPI 0
    - MPI 1
  - Lend
  - Retrieve
  - OpenMP parallel

- **Execution with DLB MPI+OmpSs**
  - Application
  - Node
    - MPI 0
    - MPI 1
  - Lend
  - Retrieve
**DLB in action**

**Original**
- 6 MPIs
- 2 threads x MPI

**DLB**
- 6 MPIs
- Initial 2 threads x MPI
- Up to 12 threads x MPI
- 12 threads active at most

**Legend**
- Idle
- Barrier
- Tasks

[Diagram showing tasks and threads with idle and barrier states]
Thank you!

For further information please contact
rosa.m.badia@bsc.es
Problems taskifying communications

Problems:

- Possibly several concurrent MPI calls
  - Need to use thread safe MPI
- Reordering of MPI calls + limited number of cores $\rightarrow$ potential source of deadlocks
  - Need to control order of communication tasks
  - Use of sentinels if necessary
- MPI task waste cores (busy waiting if communication partner delays or long transfer times)
  - An issue today, may be not with many core nodes.