Hybrid/Heterogeneous Programming with StarSs

Jesus Labarta
Director Computer Sciences Research Dept.
BSC
Index of the talk

- Introduction
- StarSs
- StarSs @ heterogenous nodes
- Hybrid MPI/StarSs
- Conclusion
Plethora of architectures

- Heterogeneity
  - system deployment
    - Cathedrals took more than 100 years to build → mix of styles
  - Multicore host/multi accelerator
- Performance
  - on purpose or not
  - Controllable or not
  - Dynamic

- Different levels, causes, degree
  - Design, manufacturing variability, frequency setting, fault tolerance,…

- Dynamic variability
Need

- Asynchrony
- Decouple algorithm – target architecture

To achieve
- Productivity: Portability, incremental,
- Dynamic, malleable, Auto matched to resources (~liquid)
A “unified” model

- StarSs
  - A “node” level programming model
  - C/Fortran + directives
  - Nicely integrates in hybrid MPI/StarSs
  - Natural support for heterogeneity

- Programmability
  - Incremental parallelization/restructure
  - Abstract/separate algorithmic issues from resources
  - Disciplined programming

- Portability
  - “Same” source code runs on “any” machine
    - Optimized task implementations will result in better performance.
  - “Single source” for maintained version of a application

- Performance
  - Asynchronous (data-flow) execution and locality awareness
  - Intelligent Runtime: specific for each type of target platform.
    - Automatically extracts and exploits parallelism
    - Matches computations to resources

Open Source     http://www.bsc.es/smpsuperscalar
                http://nanos.ac.upc.edu/
StarSs: just a few directives

```c
#pragma css task [input (<parameters>)] \n [output (<parameters>)] \n [inout (<parameters>)] \n [target device([cell, smp, cuda]) \n [implements (<task name>)\n [reduction (<parameters>)\n [highpriority]

#pragma css wait on(<data address>)
#pragma css barrier
#pragma css mutex lock (<variable>)
#pragma css mutex unlock(<variable>)
```

parameters: parameter [,parameter]*
parameter: variable_name {[dimensions]}* ranges*
ranges: {} | {index} | {start..end} | {start:length}
void Cholesky( float *A ) {
    int i, j, k;
    for (k=0; k<NT; k++) {
        spotrf (A[k*NT+k]) ;
        for (i=k+1; i<NT; i++)
            strsm (A[k*NT+k], A[k*NT+i]);
        // update trailing submatrix
        for (i=k+1; i<NT; i++) {
            for (j=k+1; j<i; j++)
                sgemm( A[k*NT+i], A[i*NT+i], A[j*NT+i]);
            ssyrk (A[k*NT+i], A[i*NT+i]);
        }
    }
}

#pragma css task inout (A[TS][TS])
void spotrf (float *A);
#pragma css task input (A[TS][TS]) inout (C[TS][TS])
void ssyrk (float *A, float *C);
#pragma css task input (A[TS][TS], B[TS][TS]) inout (C[TS][TS])
void sgemm (float *A, float *B, float *C);
#pragma css task input (T[TS][TS]) inout (B[TS][TS])
void strsm (float *T, float *B);
StarSs: the potential of data access information

- **Flat global address space seen by programmer**
- **Flexibility to dynamically traverse dataflow graph “optimizing”**
  - Concurrency. Critical path
  - Memory access: data transfers performed by run time
- **Opportunities for**
  - Prefetch
  - Reuse
  - Eliminate antidependences (rename)
  - Replication management
    - Coherency/consistency handled by the runtime
History / Strategy

- CompSs ~2007
- CellSs ~2006
- GPUSs ~2009
- SMPSs V2 ~2009
- SMPSs V1 ~2007
- NANOS++ ~2008
- GridSs ~2002
- PERMAS ~1994
- NANOS ~1996
- NanoSs ~2006
More relevant versions

SMPSs V2:
C, No Fortran?
overlap & strided
No/yes data transfers
Locality aware scheduling

SMPSs V1
contiguous, C/F, deps
MPI/SMPSs optims.

NANOS++ (OMPSs)
C/C++, No Fortran, OpenMP
Inlined/outlined pragmas
Contiguous regions
Separate dependences/transfers
SMP/GPU/Cluster
Tracing infrastructure
No renaming, No locality aware sched.
OMPSs: extending OpenP with StarSs

Dependences: not all arguments in directionality clause

Heterogeneous devices

Different implementations

Separation dependences/transfers

E. Ayguade, et al., “A Proposal to Extend the OpenMP Tasking Model for Heterogeneous Architectures” LNCS, IWOMP 2009, IJPP 2010
SMPSs implementation

- Main thread
  - User main program
  - Original task code
  - Renaming table
  - Memory
    - Data dependence, Data renaming

- Worker thread 1
  - Original task code
  - SMPSs runtime library
  - Scheduling, Task execution
  - Ready task queue
  - Work stealing

- Worker thread 2
  - Original task code
  - SMPSs runtime library
  - Scheduling, Task execution
  - Ready task queue
  - Work stealing

- IFU → DEC → REN → IQ → ISS → REG → FU

GPUSs

- First implementation

- New implementation on top of OMPSs
  - Cache management
    - Write-through
    - Coherence


Work done by Judit Planas
void blocked_cholesky(int NT, float *A) {
    int i, j, k;
    for (k=0; k<NT; k++) {
        spotrf (A[k*NT+k]);
        for (i=k+1; i<NT; i++)
            strsm (A[k*NT+k], A[k*NT+i]);
        // update trailing submatrix
        for (i=k+1; i<NT; i++) {
            for (j=k+1; j<i; j++)
                sgemm( A[k*NT+i], A[k*NT+j], A[j*NT+i]);
                ssysr (A[k*NT+i], A[i*NT+i]);
        }
    }
}

#pragma css task inout (A[TS][TS])
void spotrf (float *A);
#pragma css task input (A[TS][TS]) inout (C[TS][TS])
void ssysr (float *A, float *C);
#pragma css task input (A[TS][TS], B[TS][TS]) inout (C[TS][TS])
void sgemm (float *A, float *B, float *C);
#pragma css task input (T[TS][TS]) inout (B[TS][TS])
void strsm (float *T, float *B);
void blocked_cholesky( int NT, float *A )
{
    int i, j, k;
    for (k=0; k<NT; k++) {
        spotrf (A[k*NT+k]) ;
        for (i=k+1; i<NT; i++)
            strsm (A[k*NT+k], A[k*NT+i]);
        // update trailing submatrix
        for (i=k+1; i<NT; i++) {
            for (j=k+1; j<i; j++)
                sgemm( A[k*NT+i], A[k*NT+j], A[j*NT+i]);
            ssyrk (A[k*NT+i], A[i*NT+i]);
        }
    }
}

#pragma css task input(A[NB*NB], B[NB*NB], NB) inout(C[NB*NB]) target device (cuda)
void sgemm(float  *A, float *B, float *C, unsigned long NB)
{
    unsigned char TR='T', NT='N';
    float DONE=1.0, DMONE=-1.0;
    cublasSgemm( NT, TR, NB, NB, NB, DMONE, A, NB, B, NB, DONE,C, NB );
}

Block matrix storage @ GPU

- Source code independent of # devices

Spotrf:
Slow task @ GPU
In critical path (scheduling problem)

n = 8192; bs = 1024
Heterogeneous execution

- Spotrf more efficient at CPU
- Overlap between CPU and GPU

```c
#pragma css task input(NB) inout(A[NB*NB]) target device (smp)
void spotrf_tile(float *A, int NB)
{
    long INFO;
    char L = 'L';

    spotrf_( &L, &NB, A, &NB, &INFO );
}
```
Standard row-wise matrix association

```c
void flat_cholesky( int N, float *A ) {
    float **Ah;
    int nt = n/BS;
    Ah = allocate_block_matrix();
    convert_to_blocks(n, nt, A, Ah);
    blocked_cholesky (nt, Ah);
    convert_to_linear(n, bs, Ah, A);
    #pragma css barrier
    free_block_matrix(Ah)
}
```

```c
void convert_to_block( int n, int nt, float * A , float **Ah) {
    for (i=0; i<nt; i++)
        for (j=0; j<nt; j++)
            gather_block (n, A, i, j, Ah[i*nt+j]);
}
```

```c
void convert_to_linear(int n, int bs, float **Ah, float * A ) {
    for (i=0; i<nt; i++)
        for (j=0; j<nt; j++)
            scatter_block (n, bs, A, Ah[i*nt+j], I, j);
}
```

```c
#pragma css task input (n, I, j, A[n][n]) output (bA[bs][bs])
void gather_block (int n, float *A, int i, int j, float *bA);
#pragma css task input (n, bs, I,j, bA[bs][bs]) inout (A[n][n]) reduction (A)
void scatter_block (int n, bs, float *bA, float *A, i,j);
```
#pragma css task input (n) inout (A[n][n])
void flat_cholesky( int N, float *A ) {
  float **Ah;
  int nt = n/BS;
  Ah = allocate_block_matrix();
  convert_to_blocks(n, nt, A, Ah);
  blocked_cholesky (nt, Ah);
  convert_to_linear(n, bs, Ah, A);
  #pragma css barrier
  free_block_matrix(Ah)
}

void convert_to_block( int n, int nt, float * A , float **Ah) {
  for (i=0; i<nt; i++)
    for (j=0; j<nt; j++) gather_block (n, A, i, j, Ah[i*nt+j]);
}

void convert_to_linear(int n, int bs, float **Ah, float * A ) {
  for (i=0; i<nt; i++)
    for (j=0; j<nt; j++) scatter_block (n, bs, A, Ah[i*nt+j], i, j);
}

#pragma css task input (n, I, j, A[n][n]) output (bA[bs][bs])
void gather_block (int n, float *A, int i, int j, float *bA);
#pragma css task input (n, bs, I, j, bA[bs][bs]) inout (A[n][n]) reduction (A)
void scatter_block (int n, bs, float *bA, float *A, i,j);

Standard row-wise matrix association

BS
NB
BS
BS
BS
BS
BS
BS
BS
BS
BS
BS
BS
#pragma css task input (n, nt) inout (A[n][n])
void cholesky(int n, float *A, int nt ) {
    if (n < SMALL) { spotrf(...); return;}

    float **Ah;
    int bs= n/nt
    Ah = allocate_block_matrix();

    convert_to_blocks(n, nt, A, Ah);

    for (k=0; k<NT; k++) {
        cholesky (bs, A[k*NT+k], 2) ;
        for (i=k+1; i<NT; i++)  strsm (bs, A[k*NT+k], A[k*NT+i]);
        for (i=k+1; i<NT; i++) {
            for (j=k+1; j<i; j++) sgemm( bs, A[k*NT+i], A[k*NT+j], A[j*NT+i]);
            ssyrk (bs, A[k*NT+i], A[i*NT+i]);
        }
    }

    convert_to_linear(Ah);

    #pragma css barrier
    free_block_matrix(Ah)
}
Standard row-wise matrix association

Scheduling “problem”
Serialized copy tasks → can be overlapped

Tasks

Direction of data transfer

Transfered bytes

Bandwidth

n = 8192; bs = 1024
Standard row-wise matrix association

Scheduling “problem”
Standard row-wise matrix association

Scheduling “problem”
Cholesky task I critical path should be advanced
Standard row-wise matrix association

Cholesky matrix 8192x8192; block 1024x1024

Cholesky matrix 16384x16384; block 1024x1024

Performance (GFlops)
1 – All GPU
2 – SPOTRF @ SMP
3 – SPOTRF recursive
Cholesky @ StarSs

- Simple program + some basic, general purpose and optimizable tasks
- Focused on algorithm
  - Understandable to numerical analyst
- Easy modification of application parallel structure at source code level
  - Programmers focus on algorithm and generate "potential parallel" structure
  - Could provide hints on scheduling
- Potential for very dynamic mapping to resources
  - Scheduling more important than overhead.
  - Runtime developers focusing on scheduling optimizations with potential high level hints form the programmer
Hybrid MPI/StarSs
Hybrid MPI/SMPSs

- 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 css task inout(A[SIZE])
void Factor_panel(float *A);
#pragma css task input(A[SIZE]) inout(B[SIZE])
void update(float *A, float *B);

Hybrid MPI/SMPSs

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

```c
#pragma css task inout(A[SIZE])
void Factor_panel(float *A);

#pragma css task input(A[SIZE])
inout(B[SIZE])
void update(float *A, float *B);

#pragma css task input(A[SIZE])
void send(float *A);

#pragma css task output(A[SIZE])
void receive(float *A);

#pragma css task input(A[SIZE])
void resend(float *A);
```

... 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]);
...
Hybrid MPI/SMPSs

- **Performance**
  - Higher at smaller problem sizes
  - Improved Load balance (less processes)
  - Higher IPC
  - Overlap communication/computation

- **Tolerance to bandwidth and OS noise**

![Graph 1: Sensitivity to low bandwidth - 512 processors](image1)

![Graph 2: Sensitivity to the process preemptions - 512 processors](image2)
Hybrid MPI/SMPSs

- **Performance**
  - Higher at smaller problem sizes
  - Improved Load balance (less processes)
  - Higher IPC
  - Overlap communication/computation

- **Tolerance to bandwidth and OS noise**
Hybrid MPI/SMPSs

- **Performance**
  - Higher at smaller problem sizes
  - Improved Load balance (less processes)
  - Higher IPC
  - Overlap communication/computation

- **Tolerance to bandwidth and OS noise**
Better, but still some time in MPI !!!
Locality, memory management, architecture, ...
Renaming

- Renaming table
- Type, size,...
  - *prev
  - *obj
  - *producer
  - # users
- Object instance
- Object instance
- Object instance

Main memory: cold
- Killed transfers
- Main memory: capacity
- Global software cache
- Main memory: cold
- Local software cache

DMA Writes

DMA Reads

choleskyC 32*32 64*64

P. Belens et al, “Making the Best of Temporal Locality: Just-In-Time Renaming and Lazy Write-Back on the Cell/B.E.” IJHPC 2010
TaskSim: StarSs impact on architecture

- Tolerance to latency, need bandwidth
- Use a shared last-level cache to filter bandwidth (not for latency)

Impact of Memory Latency

Speedup

Local Cache
Remote Cache
Real DDR3 Latency

Impact latency (cycles)

Cache interleave every 128 bytes
Cache interleave every 128 KB
Cache interleave every 256 KB


A. Vega et al., “Comparing Last-level Cache Designs for CMP Architectures.” IFMT’10
If overhead was a problem

- Can the dependence detection process be supported in hardware?
  - Pipelined?

- Design: Task-level pipeline frontend that can be embedded into virtually any manycore fabric
  - Gateway:
    - Allocate resources for task meta-data
    - ORT (Object Renaming Table)
      - Map memory objects to producer tasks
    - OVT (Object Versioning Table)
      - Maintain object use-count, rename memory buffers
    - TRS (Task Reservation Stations)
      - Store task meta-data and manage dataflow firing rules

Conclusion
The TEXT project

- Towards EXaflop applications
- Demonstrate that **Hybrid MPI/SMPSSs** addresses the Exascale challenges in a productive and efficient way.
  - Deploy at supercomputing centers: Julich, EPCC, HLRS, BSC
  - Port Applications (HLA, SPECFEM3D, PEPC, PSC, BEST, CPMD, LS1 MarDyn) and develop algorithms.
- Develop additional environment capabilities
  - tools (debug, performance)
  - improvements in runtime systems (load balance and GPUss)
- Support other users
  - Identify users of TEXT applications
  - Identify and support interested application developers
- Contribute to Standards (OpenMP ARB, PERI-XML)
Conclusions

• What is important in parallel programming?
  • Asynchrony/dataflow
  • Data access awareness
  • Malleability
  • Scheduling more important than overhead

• StarSs
  • An syntax for coarse grain dataflow on C/C++/FORTRAN: incremental
  • An active research project
  • With a global view on multicore and parallel programming.
    • Homogeneous / Heterogeneous multicores
    • Nicely propagates asynchrony and overlap to Hybrid.
  • Stable enough to start porting relevant production codes at BSC
  • With a lot of potential still to be investigated.
SMPSs for SMP and multicores

- HPL Linpack: Comparison of SMPSs, OpenMP and MPI on a dual socket in Istanbul
The importance of flexibility to expose concurrency

- StarSs version allowing strided and aliased references
- Extremely simple application
- Extremely complex dependence detection

```c
void Cholesky (int N, int BS, float A[N][N]) {
    for (int j = 0; j < N; j+=BS) {
        for (int k= 0; k< j; k+=BS)
            for (int i = j+BS; i < N; i+=BS)
                sgemm_tile(BS, N, &A[k][i], &A[k][j], &A[j][i]);
        for (int i = 0; i < j; i+=BS)
            ssyrk_tile(BS, N, &A[i][j], &A[j][j]);
        spotrf_tile(BS, N, &A[j][j]);
        for (int i = j+BS; i < N; i+=BS)
            strsm_tile(BS, N, &A[j][j], &A[j][i]);
    }
}
```

Using MKL kernels/tiles

```c
#pragma css task
input(T{0:BS}{0:BS}, BS, N) inout(B{0:BS}{0:BS})
void strsm_tile (integer BS, integer N, float T[N][N], float B[N][N]) {
    unsigned char LO='L', TR='T', NU='N', RI='R';
    float DONE=1.0;
    integer LDT = sizeof(*T)/sizeof(float);
    integer LDB = sizeof(*B)/sizeof(float);
    strsm_(&RI, &LO, &TR, &NU, &BS, &BS, &DONE, T, &LDT, B, &LDB);
}
```

```c
void gs (float A[(NB+2)*BS][(NB+2)*BS]) {
    int it,i,j;
    for (it=0; it<NITERS; it++)
        for (i=0; i<N-2; i+=BS)
            for (j=0; j<N-2; j+=BS)
                gs_tile(&A[i][j]);
}
```

```c
#pragma css task
input(A{0}{1:BS}, A{BS+1}{1:BS}, \ A{1:BS}{0}, A{1:BS}{BS+1}) \ inout(A{1:BS}{1:BS})
void gs_tile (float A[N][N]) {
    for (int i=1; i <= BS; i++)
        for (int j=1; j <= BS; j++)
}
```
The importance of flexibility to expose concurrency
Is overhead important?

- Large overheads may not be a problem
  - If not in the critical path
  - And compensated by huge flexibilities
- Should overhead drive designs?
  - Of course be considered, but drive?

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Threads</th>
<th>Cholesky</th>
<th>FFT2D</th>
<th>Gauss-Seidel</th>
<th>GMRES</th>
<th>Multisort</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>32</td>
<td>1</td>
<td>32</td>
<td>1</td>
<td>32</td>
</tr>
<tr>
<td>Overhead per task</td>
<td>27µs</td>
<td>9µs</td>
<td>19µs</td>
<td>6µs</td>
<td>40µs</td>
<td>3µs</td>
</tr>
<tr>
<td>Overhead %</td>
<td>0.05%</td>
<td>0.44%</td>
<td>0.92%</td>
<td>1.87%</td>
<td>0.72%</td>
<td>1.71%</td>
</tr>
<tr>
<td>Overhead main thr.</td>
<td>0.05%</td>
<td>9.80%</td>
<td>0.92%</td>
<td>24.20%</td>
<td>0.77%</td>
<td>25.32%</td>
</tr>
<tr>
<td>Time idle</td>
<td>0.00%</td>
<td>6.42%</td>
<td>0.04%</td>
<td>2.41%</td>
<td>0.01%</td>
<td>6.16%</td>
</tr>
<tr>
<td>Number of tasks</td>
<td>5984</td>
<td>4224</td>
<td>14400</td>
<td>21930</td>
<td>21930</td>
<td>832</td>
</tr>
<tr>
<td>Average task time</td>
<td>48.5ms</td>
<td>57.5ms</td>
<td>2.0ms</td>
<td>10.1ms</td>
<td>5.1ms</td>
<td>5.6ms</td>
</tr>
</tbody>
</table>

Acceptable fraction of total time
Even if ties up a significant fraction of a core, not yet the bottleneck
HUGE absolute value

Significant reduction of task performance
The importance of Locality aware scheduling
Challenges on the way to Exascale

- Complexity
- Efficiency (..., power, ...)
- Variability
- Memory
- Faults
- Scale (..., concurrency, strong scaling, ...)

J. Labarta, et al., “BSC Vision towards Exascale”
IJHPCA vol 23, n. 4 Nov 2009
void flat_cholesky(float *A) {
    int i, j, k;
    convert_to_blocks();
    blocked_Cholesky();
    convert_to_linear();
    #pragma css barrier
}

void convert_to_block() {
    int i, j, k;
}

void convert_to_linear() {
    int i, j, k;
}

#pragma css task inout (A[TS][TS])
void gather_block(float *A);
#pragma css task input (A[TS][TS]) inout (C[TS][TS])
void scatter_block(float *A, float *C);
Scheduling “problem”
for( i = 0; i < processes; i++ )
{
    stag = i + 1; rtag = stag;
    indx = (me + nodes - i)%processes;
    shift = ((offset[indx][0]/BS)*nDIM*BS*BS);
    size = vsize*1;
    if(i%2 == 0) {
        mxm ( lda, sizes[indx][0], lda, hsize, a, b, (c+shift) );
        MPI_Sendrecv (a, size, MPI_DOUBLE, down, stag, rbuf, size, MPI_DOUBLE, up, rtag, comm, &stats );
    } else {
        mxm (lda, sizes[indx][0], lda, hsize, rbuf, b, (c+shift) );
        MPI_Sendrecv (rbuf, size, MPI_DOUBLE, down, stag, a, size, MPI_DOUBLE, up, rtag, comm, &stats );
    }
}

void mxm ( int lda, int m, int l, 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, &l, &alpha, a, &lda, b, &m, &beta, c, &m);
... to hybrid MPI/SMPSs ...

```c
for( i = 0; i < processes; i++ )
{
    stag = i + 1; rtag = stag;
    indx = (me + nodes - i )%processes;
    shift = ((offset[indx][0]/BS)*nDIM*BS*BS);
    size = vsize*l;
    if(i%2 == 0) {
        mxm( lda, sizes[indx][0], lda, hsize, a, b, (c+shift) );
        callSendRecv (a, size, down, stag, rbuf, up, rtag);
    } else {
        mxm (lda, sizes[indx][0], lda, hsize, rbuf, b, (c+shift) );
        callSendRecv (rbuf, size, down, stag, a, up, rtag);
    }
}

#pragma css task input (size, down, stag, up, rtag, a[size]) output (rbuf[size])
void callSendRecv (double *a, int size, int down, int stag, double *rbuf, int up, int rtag)
{
    MPI_Status stats;
    MPI_Sendrecv( a, size, MPI_DOUBLE, down, stag, rbuf, size, MPI_DOUBLE, up, rtag, MPI_COMM_WORLD, &stats );
}

#pragma css task input( lda, m, l, n, a[m*l], b[l*n]) inout(c[m*n])
void mxm ( int lda, int m, int l, 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, &l, &alpha, a, &lda, b, &m, &beta, c, &m);
}
```
for( i = 0; i < processes; i++ )
{
    stag = i + 1; rtag = stag;
    indx = (me + nodes - i )%processes;
    shift = ((offset[indx][0]/BS)*nDIM*BS*BS);
    size = vsize*l;
    if(i%2 == 0) {
        mxm( lda, sizes[indx][0], lda, hsize, a, b, (c+shift) );
        callSendRecv (a, size, down, stag, rbuf, up, rtag);
    } else {
        mxm (lda, sizes[indx][0], lda, hsize, rbuf, b, (c+shift) );
        callSendRecv (rbuf, size, down, stag, a, up, rtag);
    }
}

#pragma css task input (size, down, stag, up, rtag, a[size]) output rbuf[size]
void callSendRecv (double *a, int size, int down, int stag, double *rbuf, int up, int rtag)
{
    MPI_Status stats;
    MPI_Sendrecv( a, size, MPI_DOUBLE, down, stag, rbuf, size, MPI_DOUBLE, up, rtag, MPI_COMM_WORLD, &stats );
}

#pragma css task input( lda, m, l, n, a[m*l], b[l*n]) inout(c[m*n])
void mxm ( int lda, int m, int l, 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, &l, &alpha, a, &lda, b, &m, &beta, c, &m);
Hybrid MPI/SMPSs linpack @ BG/P

BG/P 16cores(4node)

BG/P 1024 cores(256nodes)
TaskSim: Architectural simulator

- Event-driven simulator for large-scale accelerator-based architectures
  - Trace-driven, CPU bursts
  - StarSs programs

- Cycle-accurate simulation of:
  - DMA/Cache controllers
  - Caches
  - Interconnect
  - Memory controller and DRAM DIMMs

- Example architecture
  - 256 processors in 32 clusters of 8
  - 256KB Local Memory + DMA
  - No cache. All DMA accesses go to off-chip memory