Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Programming with StarSs

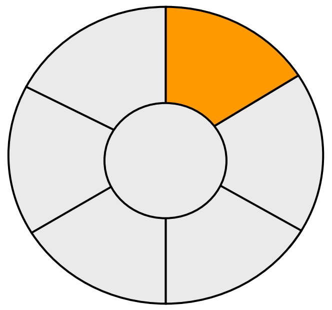
Computer Sciences Research Dept.
BSC

Agenda

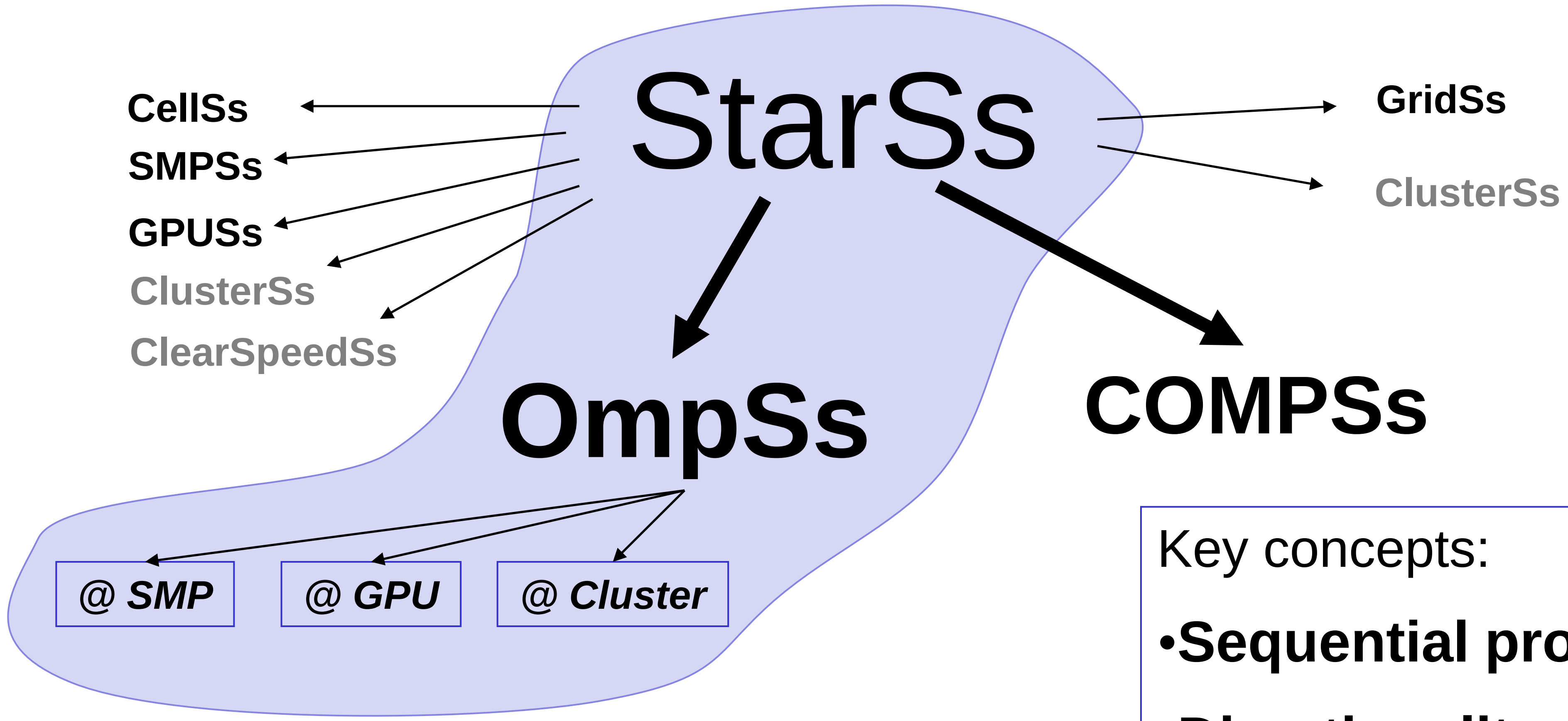
- StarSs overview
- OmpSs
- OmpSs examples
- Single node hands-on
- Hybrid model MPI/OmpSs
- Programming examples
- MPI/OmpSs hands-on

- Slides available in:
  - /gpfs/scratch/bsc19/bsc19776/TutorialRES2011/tutorial_RES_starss_2011.ppt

StarSs overview



The StarSs family of programming models



- A "node" level programming model
- C/Fortran/Java
- Task based. Asynchrony, data-flow.
- Single linear address space
- Malleable
- Nicely hybridizes (MPI/StarSs)
- Natural support for heterogeneity

Key concepts:

•**Sequential program**

•**Directionality annotations on tasks arguments**

Focus:

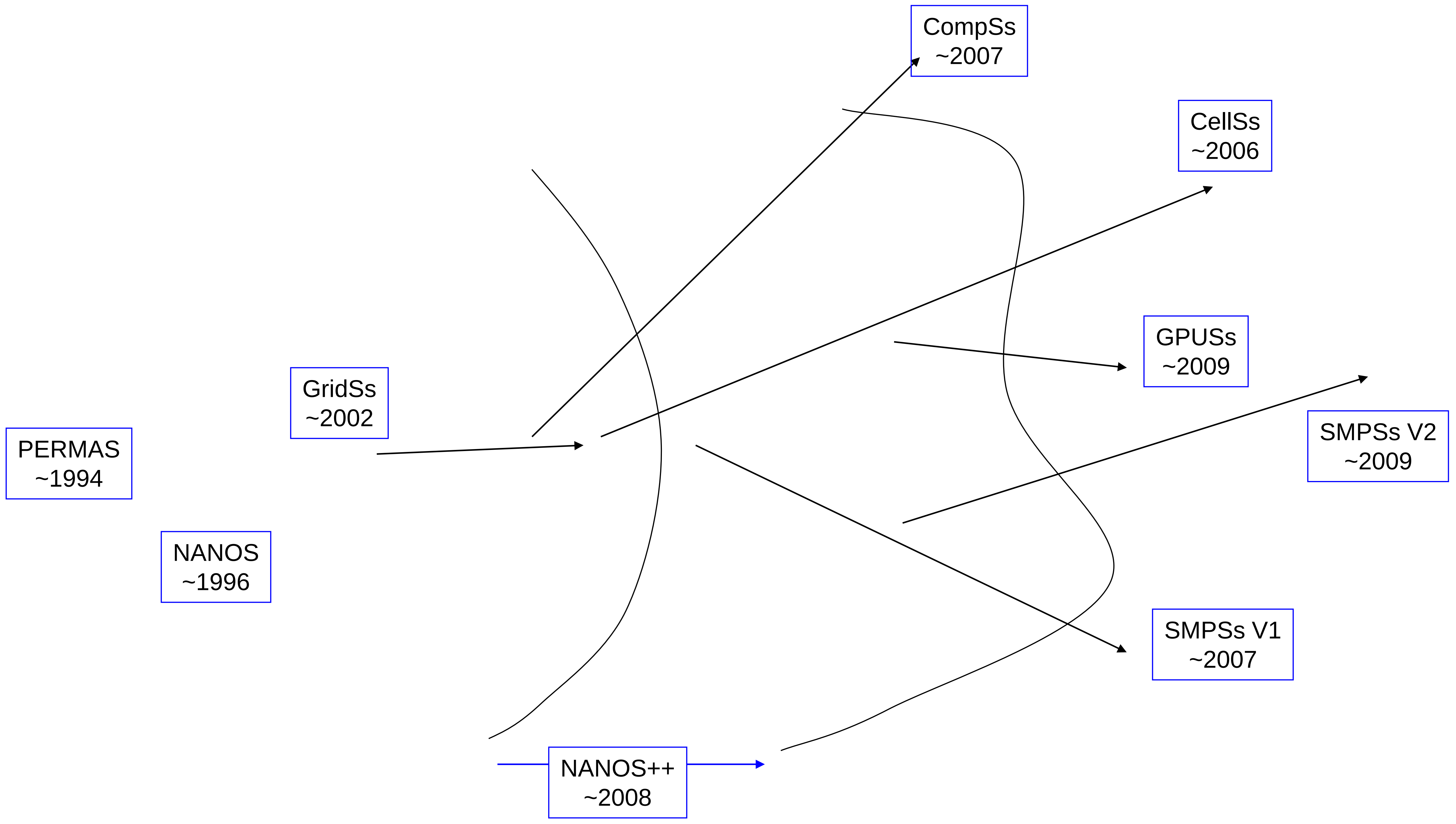
•**Programmability/Portability**

•**Intelligence in the runtime**

Open Source
http://pm.bsc.es/ompss/

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

History / Strategy



StarSs: a sequential program …

```
void vadd3 (float A[BS], float B[BS],
            float C[BS]);

void scale_add (float sum, float A[BS],
               float B[BS]);

void accum (float A[BS], float *sum);
```

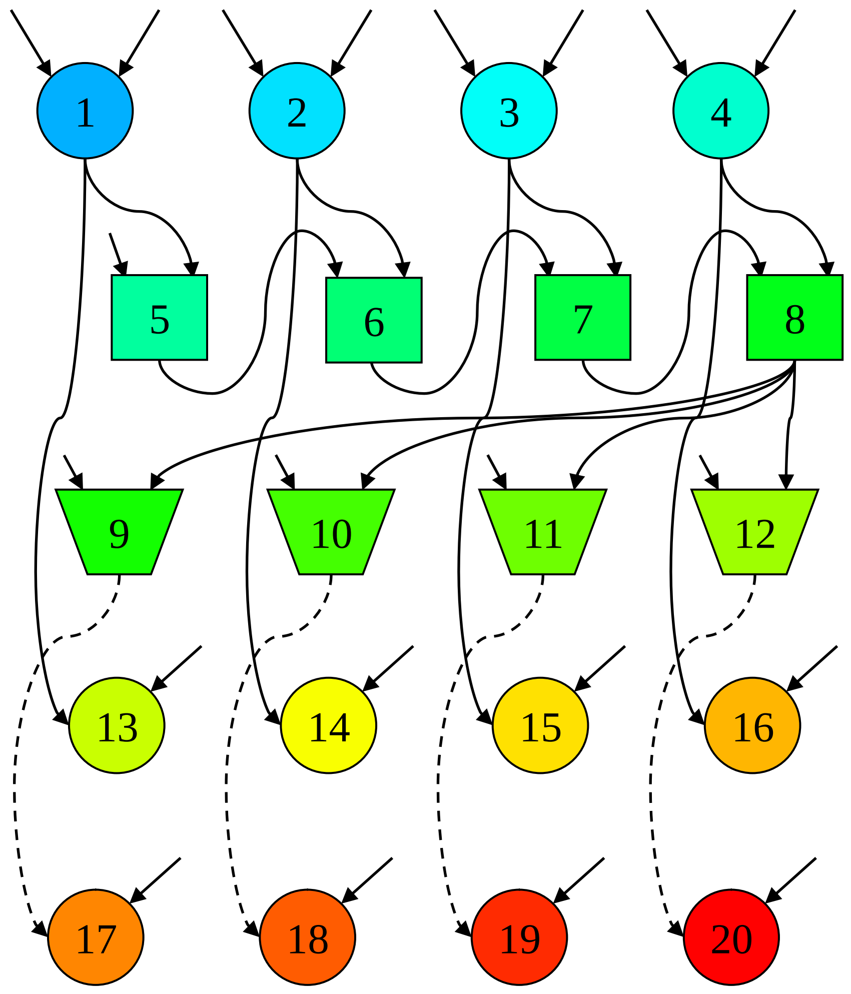
```
for (i=0; i<N; i+=BS)              // C=A+B
   vadd3 ( &A[i], &B[i], &C[i]);
...
for (i=0; i<N; i+=BS)             // sum(C[i])
   accum (&C[i], &sum);
...
for (i=0; i<N; i+=BS)             // B=sum*A
   scale_add (sum, &E[i], &B[i]);
...
for (i=0; i<N; i+=BS)             // A=C+D
   vadd3 (&C[i], &D[i], &A[i]);
...
for (i=0; i<N; i+=BS)             // E=G+F
   vadd3 (&G[i], &F[i], &E[i]);
```

StarSs: … taskified …

```
#pragma css task input(A, B) output(C)
void vadd3 (float A[BS], float B[BS],
            float C[BS]);
#pragma css task input(sum, A) inout(B)
void scale_add (float sum, float A[BS],
               float B[BS]);
#pragma css task input(A) inout(sum)
void accum (float A[BS], float *sum);
```

```
for (i=0; i<N; i+=BS)                    // C=A+B
   vadd3 ( &A[i], &B[i], &C[i]);
...
for (i=0; i<N; i+=BS)                   // sum(C[i])
   accum (&C[i], &sum);
...
for (i=0; i<N; i+=BS)                   // B=sum*A
   scale_add (sum, &E[i], &B[i]);
...
for (i=0; i<N; i+=BS)                   // A=C+D
   vadd3 (&C[i], &D[i], &A[i]);
...
for (i=0; i<N; i+=BS)                   // E=G+F
   vadd3 (&G[i], &F[i], &E[i]);
```

Compute dependences @ task instantiation time



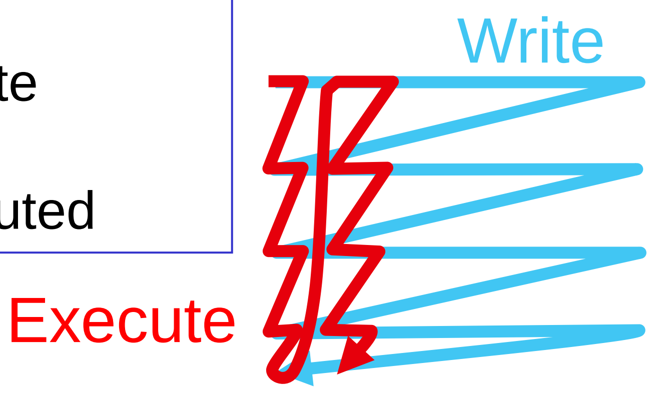
Color/number: order of task instantiation
Some antidependences covered by flow dependences not drawn

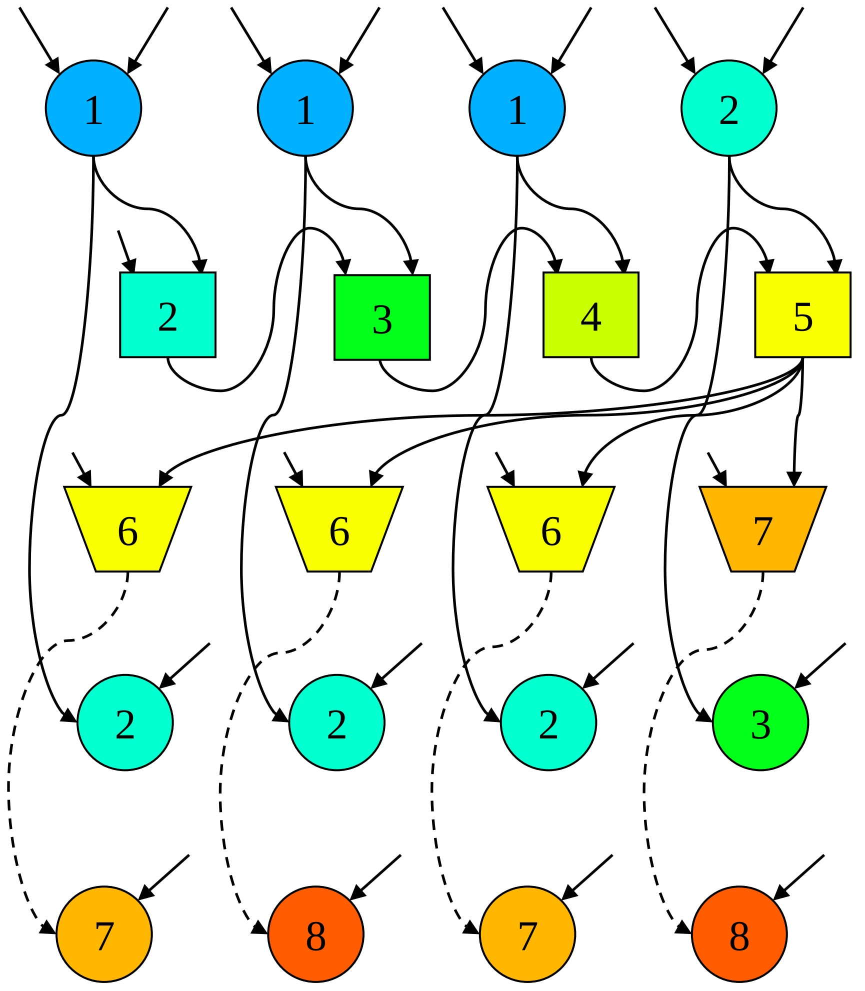
StarSs: … and executed in a data-flow model

```
#pragma css task input(A, B) output(C)
void vadd3 (float A[BS], float B[BS],
            float C[BS]);
#pragma css task input(sum, A) inout(B)
void scale_add (float sum, float A[BS],
               float B[BS]);
#pragma css task input(A) inout(sum)
void accum (float A[BS], float *sum);
```

```
for (i=0; i<N; i+=BS)                // C=A+B
   vadd3 ( &A[i], &B[i], &C[i]);
...
for (i=0; i<N; i+=BS)               // sum(C[i])
   accum (&C[i], &sum);
...
for (i=0; i<N; i+=BS)               // B=sum*A
   scale_add (sum, &E[i], &B[i]);
...
for (i=0; i<N; i+=BS)               // A=C+D
   vadd3 (&C[i], &D[i], &A[i]);
...
for (i=0; i<N; i+=BS)               // E=G+F
   vadd3 (&G[i], &F[i], &E[i]);
```

Decouple
how we write
form
how it is executed

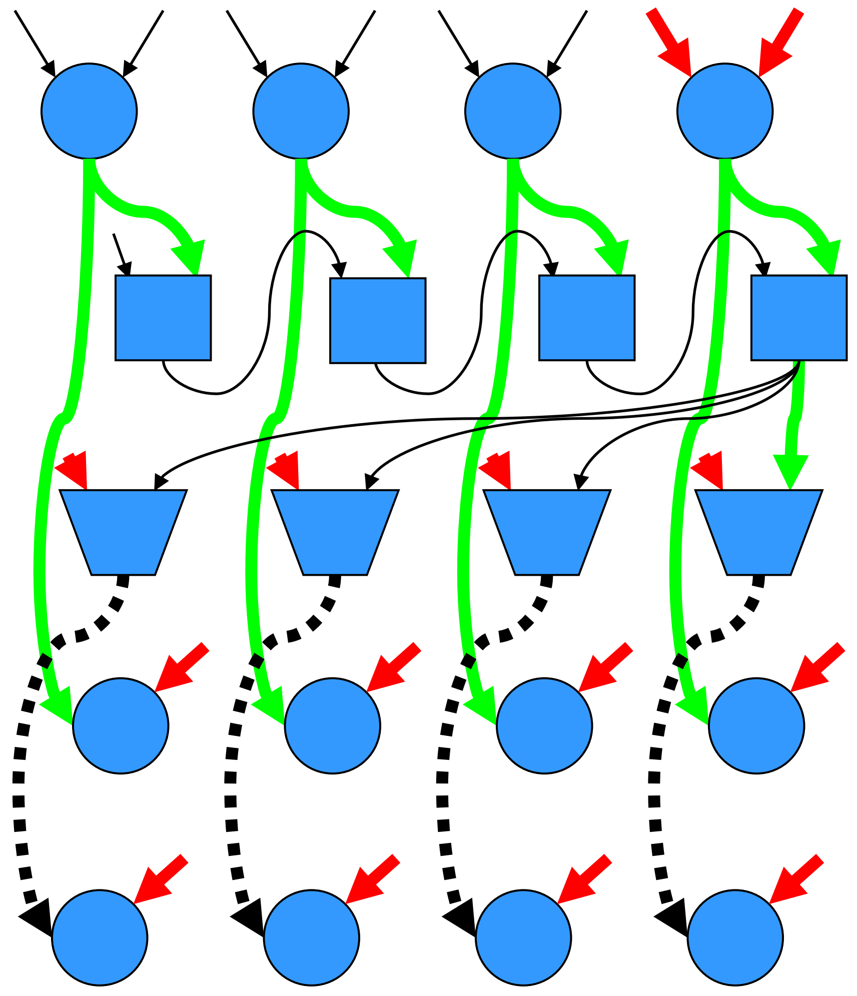




Color/number: a possible order of task execution

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

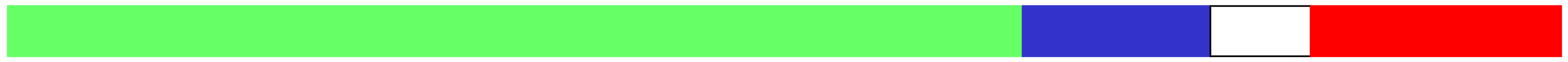


StarSs Tasks

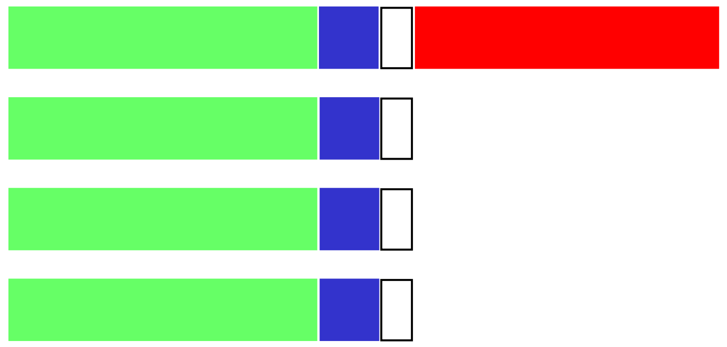
- Task states:
  - **Instantiated**: when task is created
    - Dependences are computed at the moment of instantiation. At that point in time a task may or may not be ready for execution
  - **Ready**: When all its input dependences are satisfied, typically as a result of the completion of other tasks
    - The task is ready to be executed.
  - **Active**: the task has been scheduled to a processing element.
    - Will take a finite amount of time to execute.
  - **Completed**: the task terminates, its state transformations are guaranteed to be globally visible and frees its output dependences to other tasks.

Fighting Amdahl's law: StarSs. a chance for lazy programmers

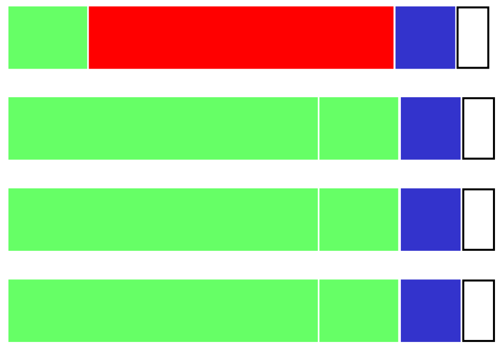
Four loops/routines
Sequential program order



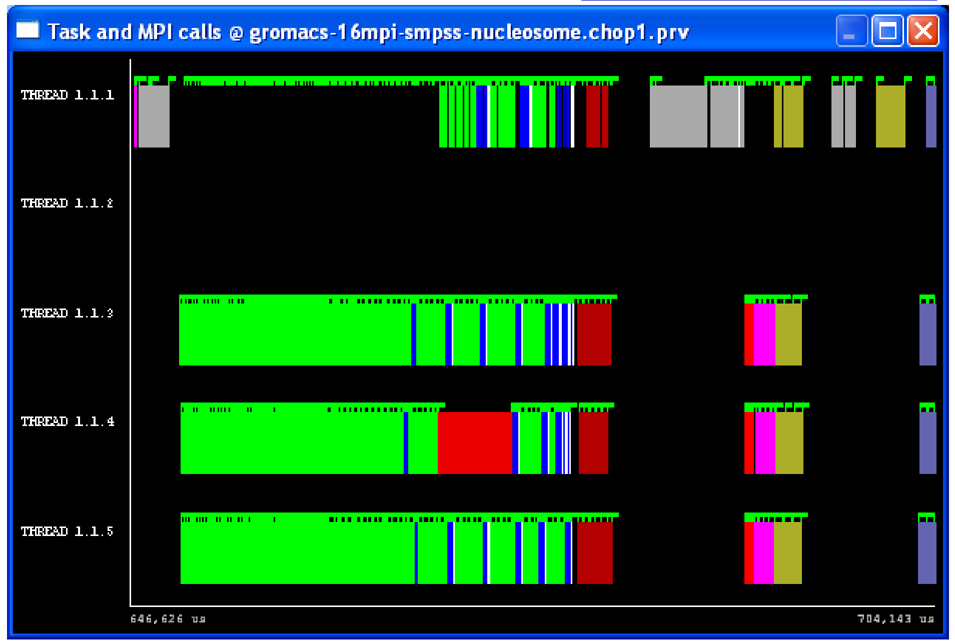
OpenMP
not parallelizing one loop



SMPSs
not parallelizing one loop



GROMACS

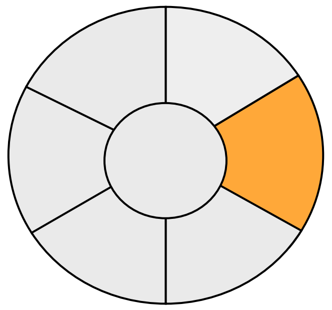


StarSs: just a few directives

```
#pragma css task  [input ( parameters ) ] \
                  [output ( parameters ) ] \
                  [inout ( parameters )] \
                  [target device( [cell, smp, cuda] ) ] \
                  [implements ( task_name ) ] \
                  [reduction ( parameters ) ] \
                  [ 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* {[*dimension*]}*

OmpSs

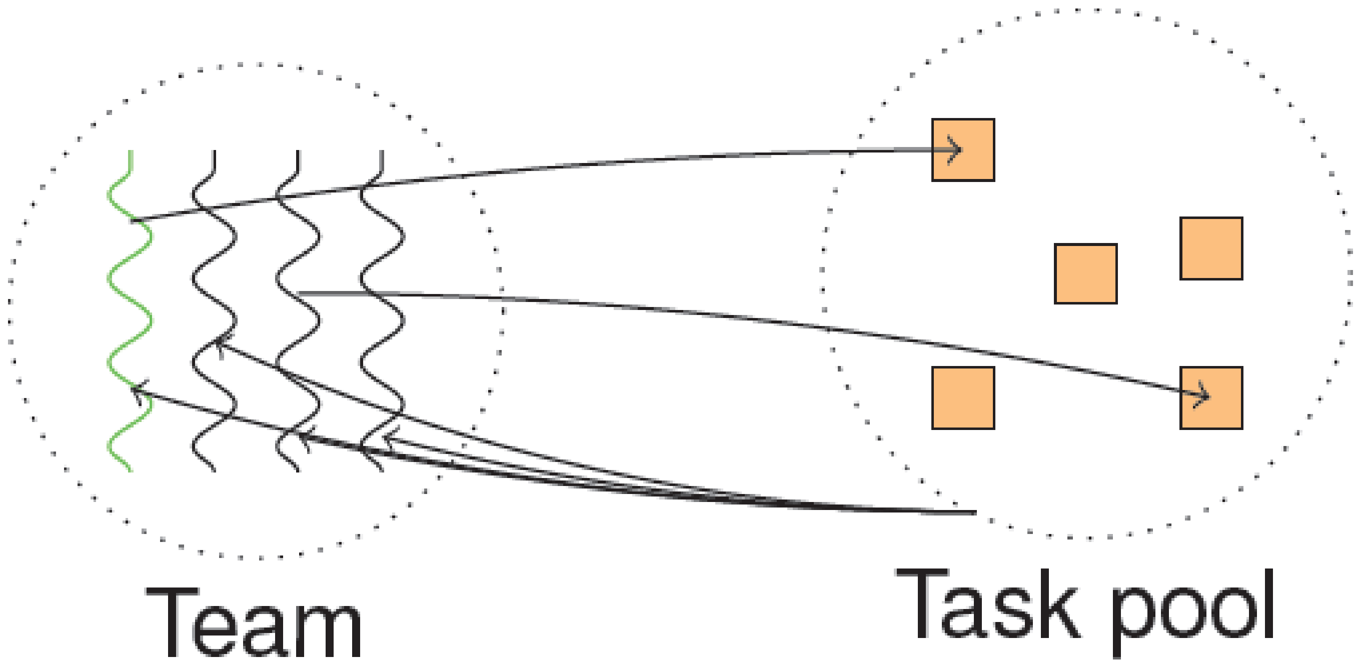


OmpSs = OpenMP + StarSs extensions

- OmpSs is based on OpenMP + StarSs with some differences:
  - Different execution model
  - Extended memory model
  - Extensions for point-to-point inter-task synchronizations
    - data dependencies
  - Extensions for heterogeneity
  - Other minor extensions

Execution Model

- Thread-pool model
  - OpenMP parallel “ignored”
- All threads created on startup
  - One of them starts executing main
- All get work from a task pool
  - And can generate new work



Memory Model

- From the point of view of the programmer a single naming space exists
- From the point of view of the runtime, different possible scenarios
  - pure SMP:
    - Single address space
  - distributed/heterogeneous (cluster, gpus, ...):
    - Multiple address spaces exist
      - Versions of same data may exist in multiple of these
    - Data consistency ensured by the implementation

Main element: task

- Task: unit of computation
- Task definition
  - Pragmas inlined
  - Pragmas attached to function definition

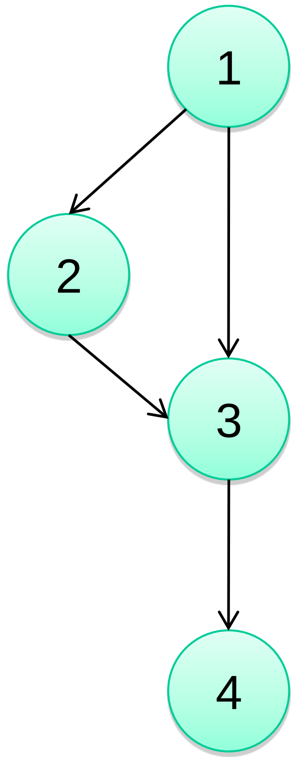
```
int   main   (    )
{
   int X[100];
   #pragma  omp   task
   for (int i =0; i< 100; i++) X[i]=i;
}
```

```
#pragma  omp   task
void   foo (int Y[size], int size) {
int j;

   for (j=0; j < size; j++) Y[j]= j;
}

int main()
{
int X[100]
foo (X, 100) ;
}
```

Defining dependences

- Clauses that express data direction:
  - input
  - output
  - inout
- Dependences computed at runtime taking into account these clauses

```
#pragma omp task output( x )
x = 5;
#pragma omp task input( x )
printf("%d\n" , x ) ;
#pragma omp task inout( x )
x++;
#pragma omp task input( x )
printf ("%d\n" , x ) ;
```



$$\begin{tikzcd}
 & 1 \arrow[dl] \arrow[ddr, start anchor=south, end anchor=north] & \\
2 \arrow[dr] & & \\
 & 3 \arrow[d] & \\
 & 4 &
\end{tikzcd}$$

Heterogeneity: the target directive

- Directive to specify device specific information:

  #pragma omp target [clauses]

  - Clauses:
    - device: which device (smp, gpu)
    - copy_in, copy_out, copy_inout: data to be moved in and out
    - copy_deps: same as above, to copy data specified in input/output/inout clauses
    - implements: specifies alternate implementations

```
#pragma target device (smp) copy_deps
#pragma omp task input ([size] c) output ([size] b)
void scale_task (double *b, double *c, double scalar, int size)
{
int j;
for (j=0; j < BSIZE; j++)
    b[j] = scalar*c[j];
}
```

Heterogeneity: the target directive

- Directive to specify device specific information:

  #pragma omp target [clauses]

  - Clauses:
    - device: which device (smp, gpu)
    - copy_in, copy_out, copy_inout: data to be moved in and out
    - copy_deps: same as above, to copy data specified in input/output/inout clauses
    - implements: specifies alternate implementations

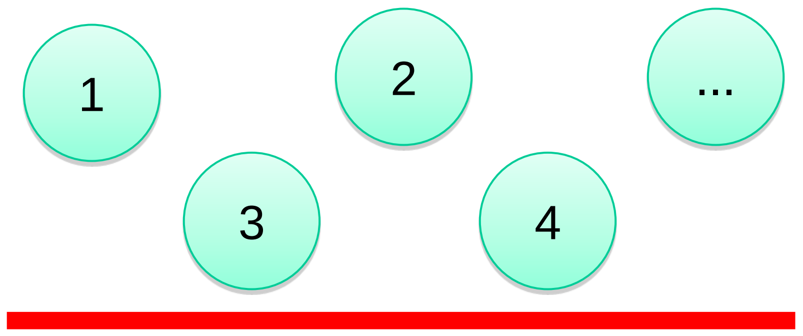
```
#pragma omp target device (cuda) copy_deps implements (scale_task)
#pragma omp task input ([size] c) output ([size] b)
void scale_task_cuda(double *b, double *c, double scalar, int size)
{

const int threadsPerBlock = 128;
   dim3 dimBlock;
   dimBlock.x = threadsPerBlock;
   dimBlock.y = dimBlock.z = 1;

   dim3 dimGrid;
   dimGrid.x = size/threadsPerBlock+1;

   scale_kernel<<<dimGrid,dimBlock>>>(size, 1, b, c, scalar);

}
```

Synchronization

#pragma omp taskwait

- Suspends the current task until all children tasks are completed
- Just direct children, not descendants

```
void traverse_list ( List l )
{
   Element e ;
   for ( e = l-> first; e ; e = e->next )
   #pragma omp task
    process ( e ) ;

#pragma omp taskwait
}
```

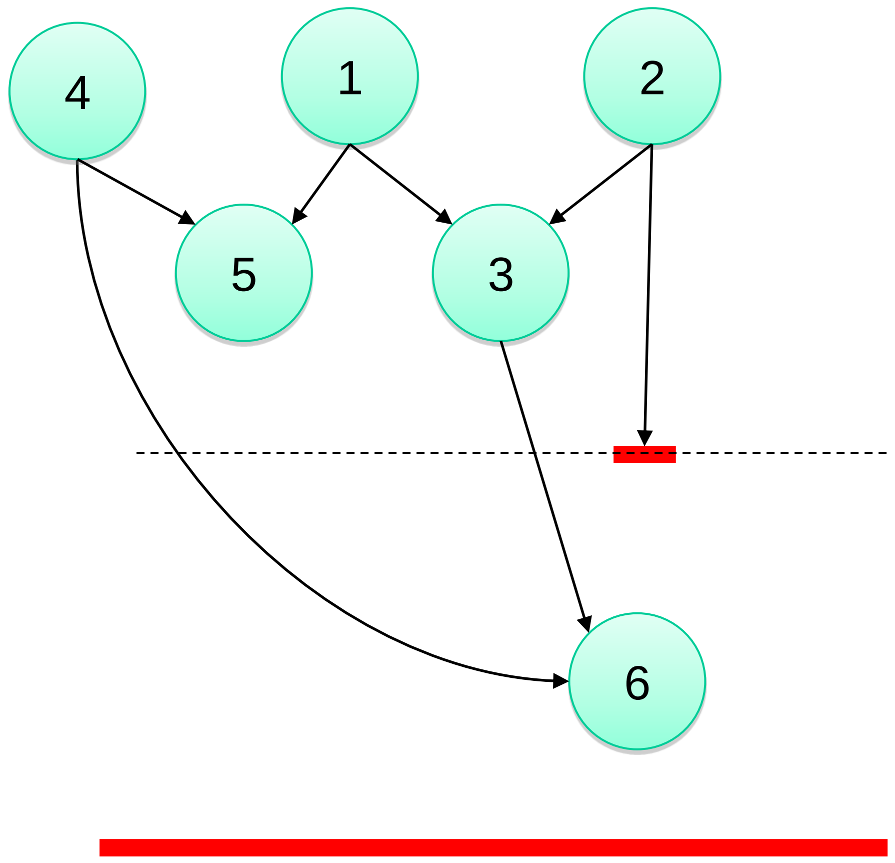


Synchronization

#pragma taskwait on (expression)

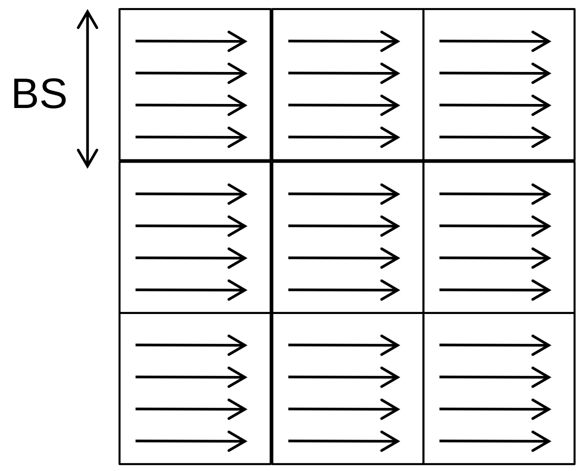
- Expressions allowed are the same as for the dependency clauses
- Blocks the encountering task until the data is available

```
#pragma css task input([N][N]A, [N][N] B) inout([N][N]C)
void dgemm(float *A, float *B, float *C);
main() {
(
 ...
dgemm(A,B,C); //1
dgemm(D,E,F); //2
dgemm(C,F,G); //3
dgemm(A,D,H); //4
dgemm(C,H,I); //5

#pragma omp taskwait on (F)
prinft ("result F = %f\n", F[0][0]);


dgemm(H,G,C); //6


#pragma omp taskwait
prinft ("result C = %f\n", C[0][0]);
}
```

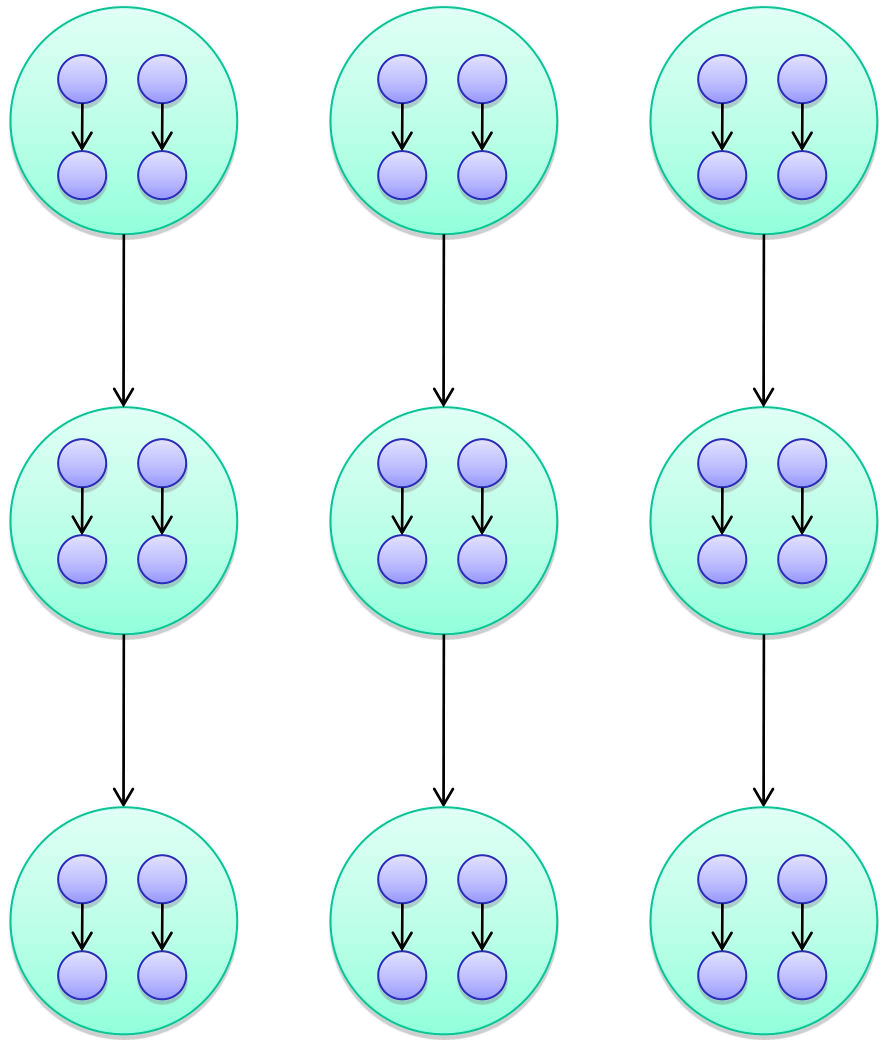


Hierarchical task graph

- Nesting
- Hierarchical task dependences
- Block data-layout

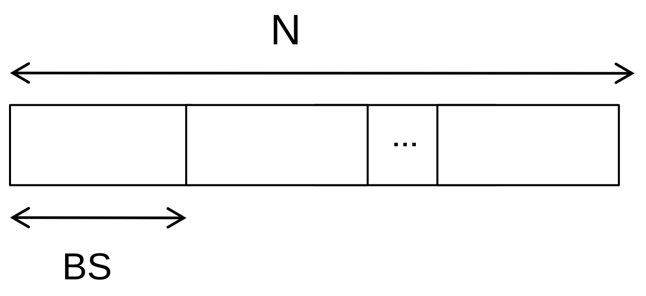


```
#pragma omp task input([BS][BS]A, [BS][BS] B)\
inout([BS][BS]C)
void block_dgemm(float *A, float *B, float *C);

#pragma omp task input([N][N]A, [N][N] B)\
inout([N][N]C)
void dgemm(float *A, float *B, float *C){
int i, j, k;
int NB= N/BS;

for (i=0; i< NB; i++)
for (j=0; j< NB; j++)
   for (k=0; k< NB; k++)
      block_dgem(&A[i*N*BS+k*BS*BS],
         &B[k*N*BS+j*BS*BS], &C[i*N*BS+j*BS*BS])
}
main() {
(
 ...
dgemm(A,B,C);
dgemm(D,E,F);
#pragma omp taskwait

}
```



Concurrent

- Less-restrictive inout clause → *Concurrent* tasks can run in parallel
  - Dependences with other tasks will be handled normally
    - I.e., the printf task will wait for all sum_task's to finish
- The task may require additional synchronization
  - i.e., atomic accesses



```
#pragma omp task input ( [ n ] vec ) concurrent (results)
void sum_task ( int *vec , int n , int  *results)
{
   int i ;
   int local_sum=0;
   for ( i = 0; i < n ; i ++)
    local_sum += vec [i] ;

   #pragma omp atomic
 results += local_sum;
}

void main(){
   for (int j; j<N; j+=BS) sum_task (&vec[j], BS, &total);
   #pragma omp task input (total)
printf ("TOTAL is %d\n", total);
}
```

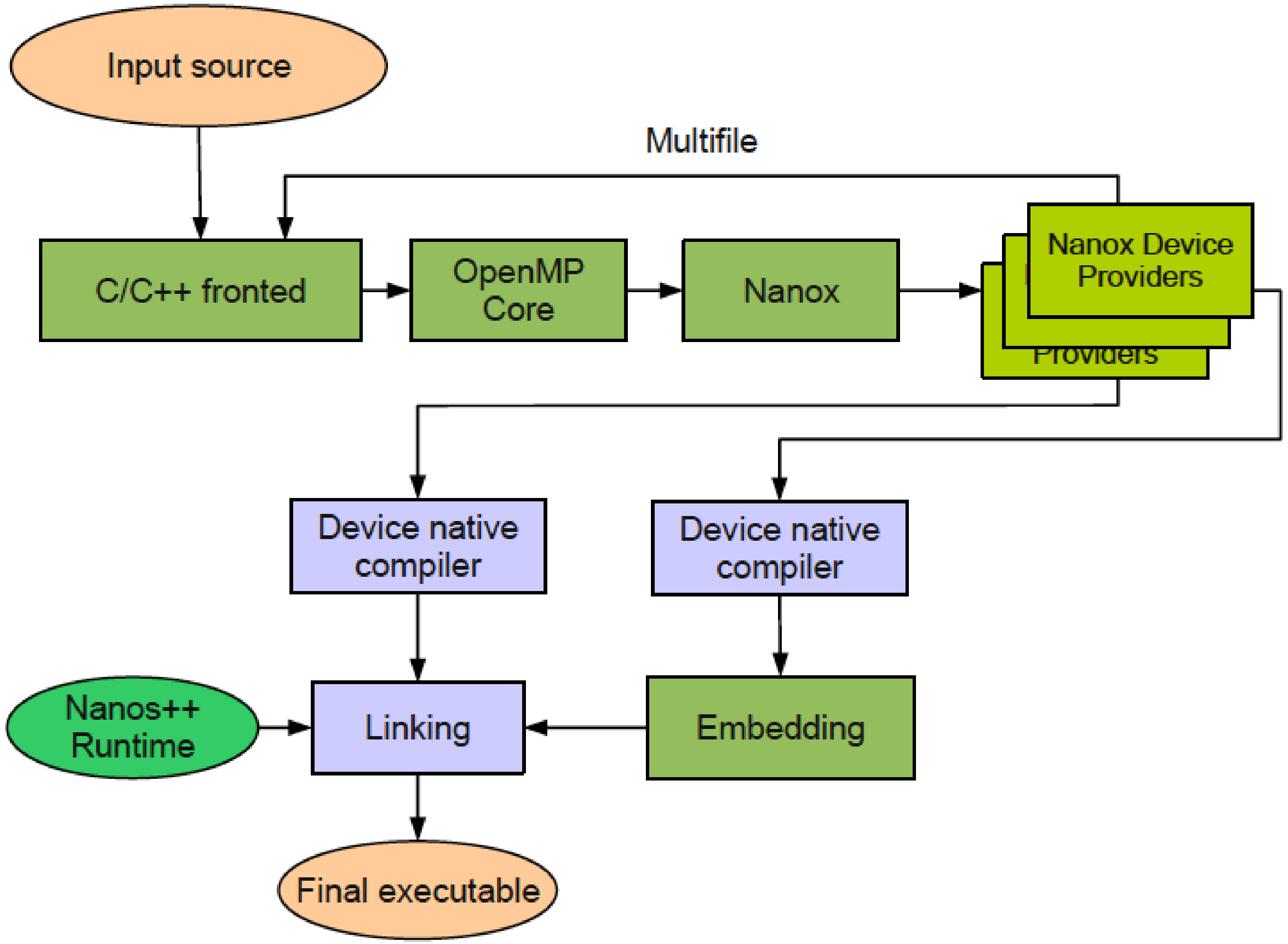
Avoiding data transfers

- Need to synchronize
- No need for synchronous data output

```
void compute_perlin_noise_device(pixel * output, float time, unsigned int
rowstride, int img_height, int img_width)
{
    unsigned int i, j;
    float vy, vt;
    const int BSy = 1;
    const int BSx = 512;
    const int BS = img_height/16;

    for (j = 0; j < img_height; j+=BS) {
#pragma omp target device(cuda) copy_out(output[j*rowstride;BS*rowstride])
#pragma omp task
        {
            dim3 dimBlock, dimGrid;
            dimBlock.x = (img_width < BSx) ? img_width : BSx;
            dimBlock.y = (BS < BSy) ? BS : BSy;
            dimBlock.z = 1;
            dimGrid.x = img_width/dimBlock.x;
            dimGrid.y = BS/dimBlock.y;
            dimGrid.z = 1;
            cuda_perlin <<<dimGrid, dimBlock>>> (&output[j*rowstride],
                    time, j, rowstride);
        }
    }
#pragma omp taskwait noflush
}
```

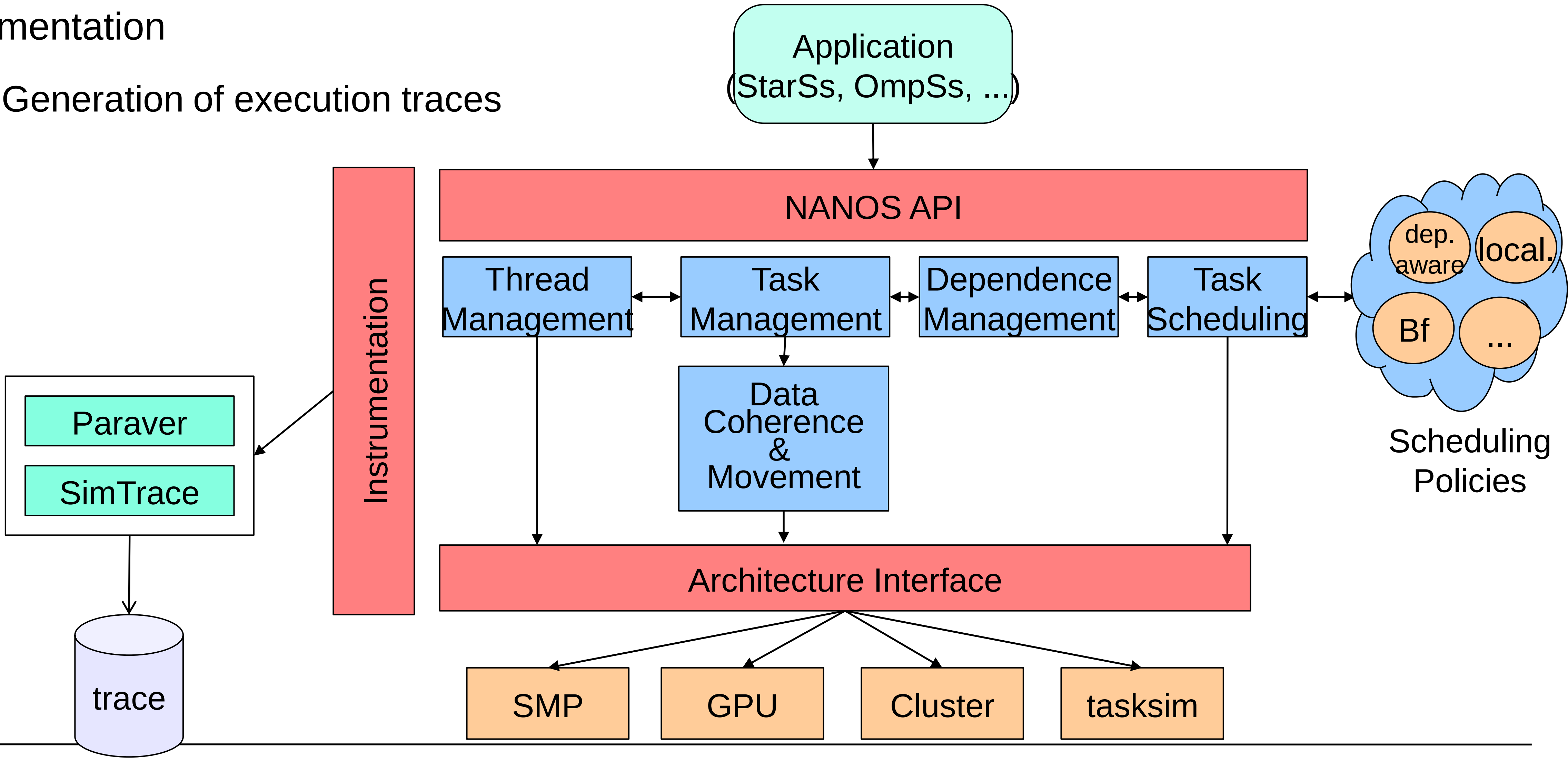
Mercurium Compiler

- Minor role
- Recognizes constructs and transforms them to calls to the runtime
- Manages code restructuring for different target devices
  - Device-specific handlers
  - May generate code in a separate file
  - Invokes different back-end compilers
    → nvcc for NVIDIA



Runtime structure

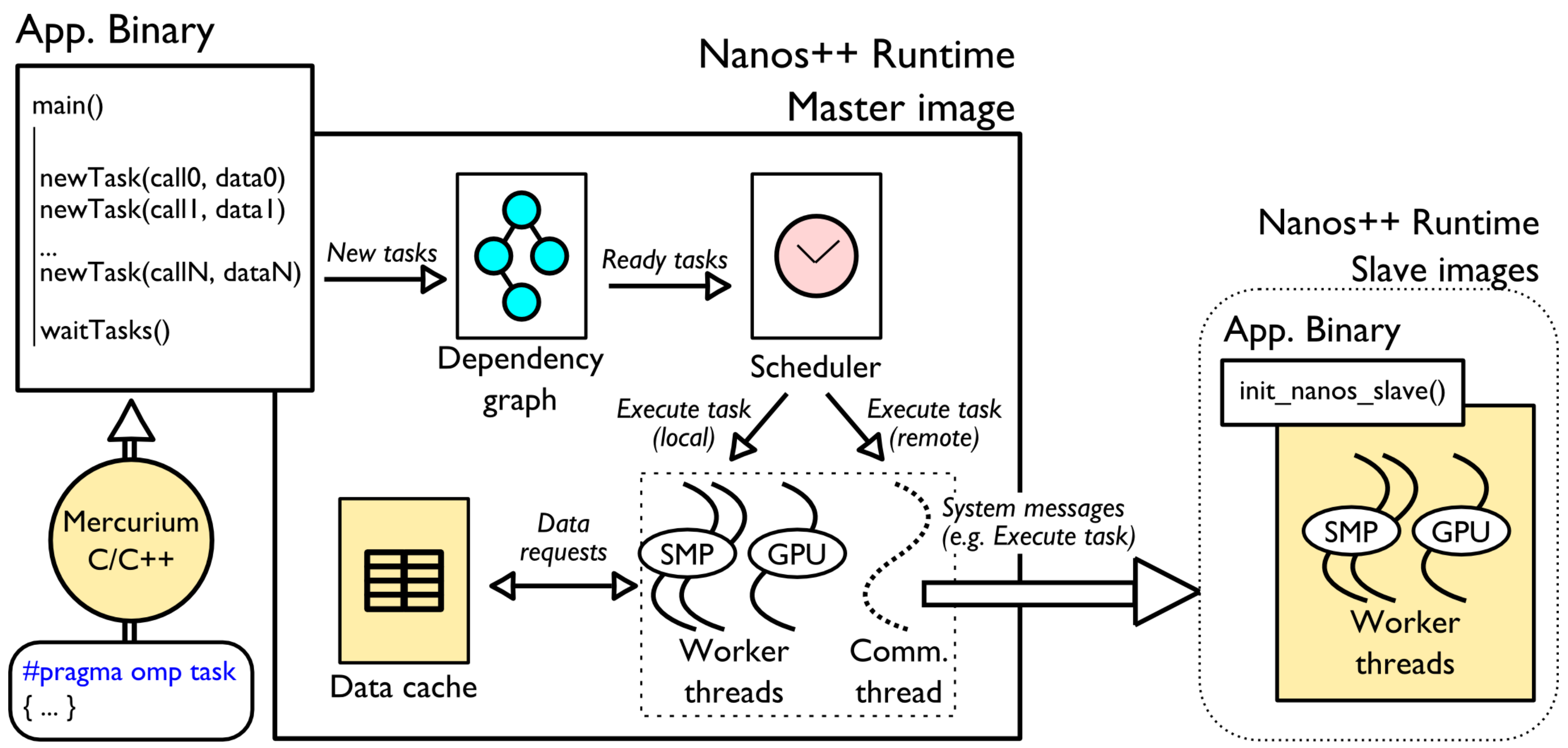
- Support to different programming models: OpenMP (OmpSs), StarSs, Chapel
- Independent components for thread, task, dependence management, task scheduling, ...
- Most of the runtime independent of the target architecture: SMP, GPU, tasksim simulator, cluster
- Support to heterogeneous targets
  - → i.e., threads running tasks in regular cores and in GPUs
- Instrumentation
  - →Generation of execution traces



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

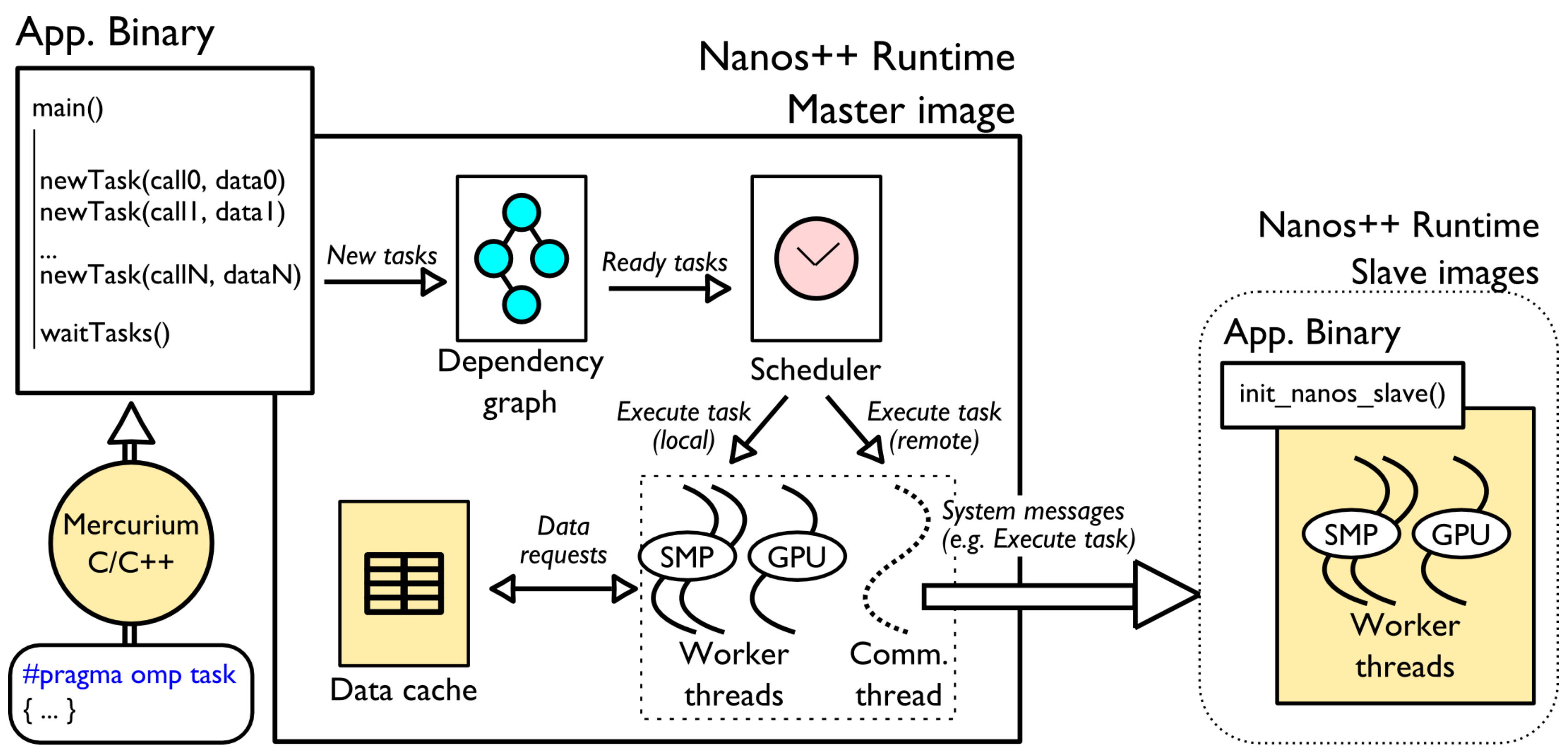
Runtime structure behaviour: task handling

- Task generation
- Data dependence analysis
- Task scheduling



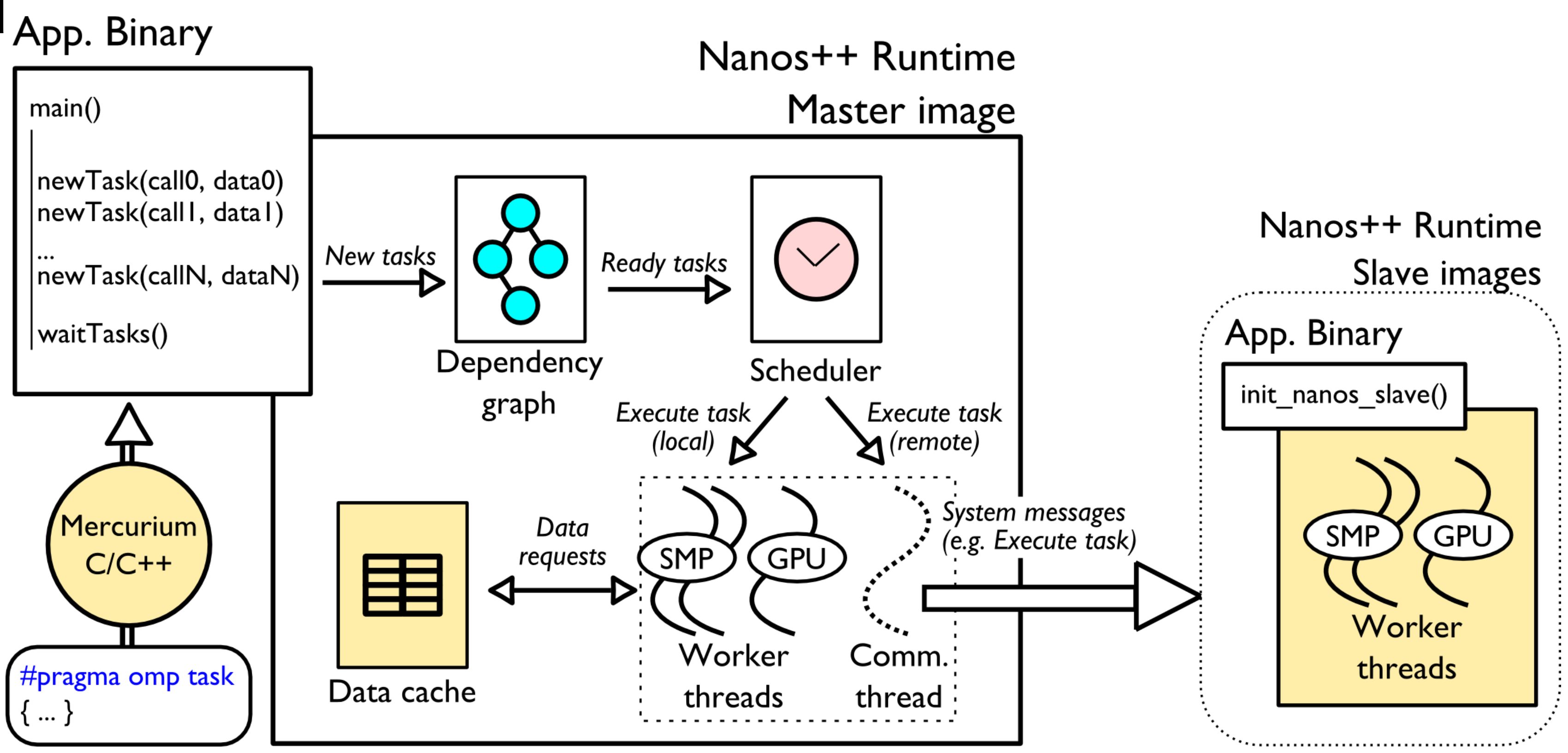
Runtime structure behaviour: coherence support

- Different address spaces managed with:
  - A hierarchical directory
    - Information about where the data is
  - A software cache per each:
    - Cluster node
    - GPU
- Data transfers between different memory spaces only when needed
  - Write-through
  - Write-back



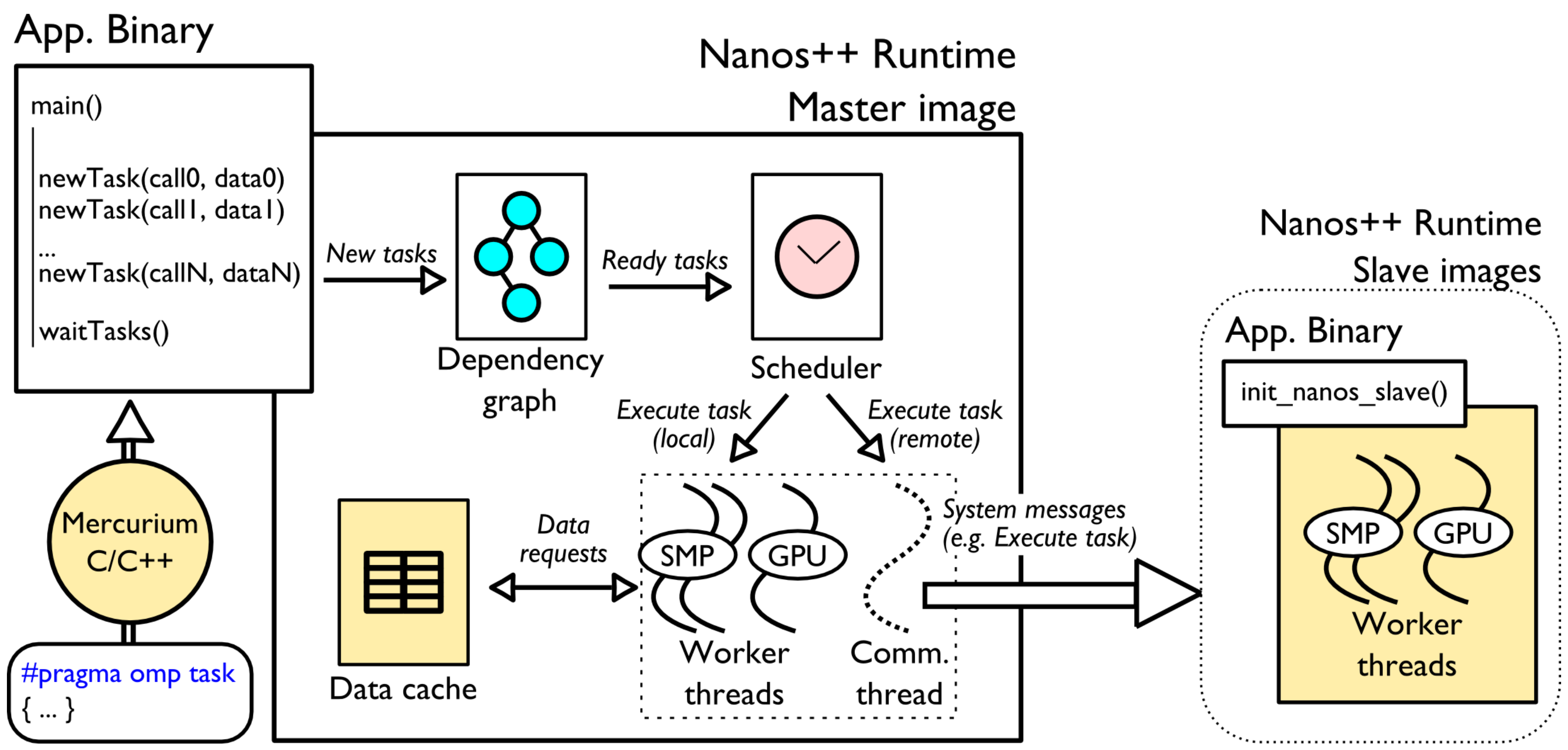
Runtime structure behaviour: GPUs

- Automatic handling of Multi-GPU execution
- Transparent data-management on GPU side (allocation, transfers, ...) and synchronization
- One manager thread in the host per GPU. Responsible for:
- Transferring data from/to GPUs
- Executing GPU tasks
- Synchronization
- Overlap of computation and communication
- Data pre-fetch
- Enabled with CUDA 3 and CUDA 4



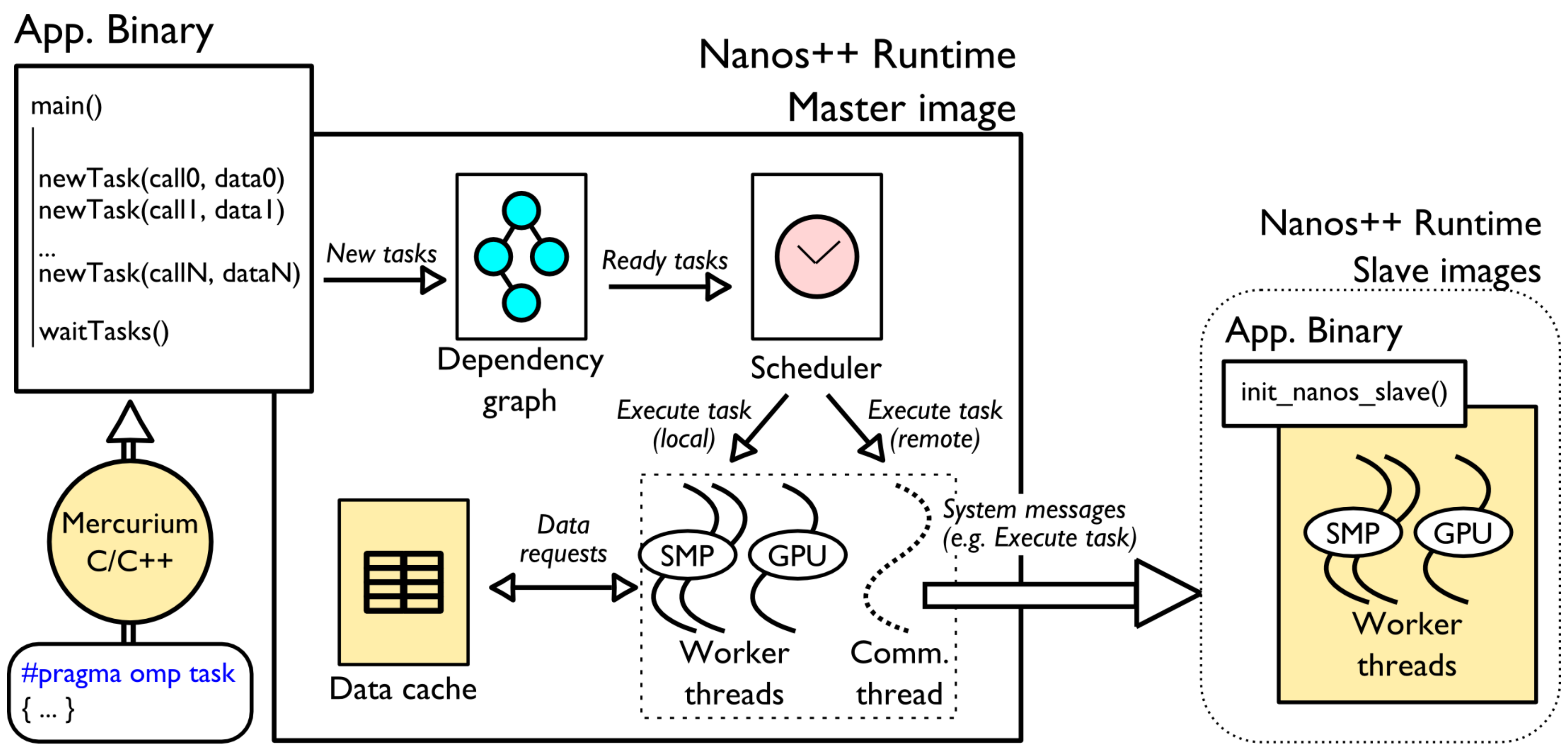
Runtime structure behaviour: clusters

- One runtime instance per node
  - One master image
  - N-1 worker images
- Low level communication through active messages
- Tasks generated by master
  - Tasks executed by worker threads in the master
  - Tasks delegated to slave nodes through the communication thread
- Remote task execution:
  - Data transfer (if necessary)
  - Overlap of computation with communication
  - Task execution
    - Local scheduler

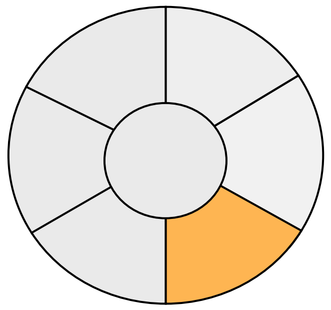


Runtime structure behavior: clusters of GPUs

- Composes previous approaches
- Supports for heterogeneity and hierarchy:
  - Application with homogeneous tasks: SMP or GPU
  - Applications with heterogeneous tasks: SMP and GPU
  - Applications with hierarchical and heterogeneous tasks:
    - I.e., coarser grain SMP tasks
    - Internally generating GPU tasks

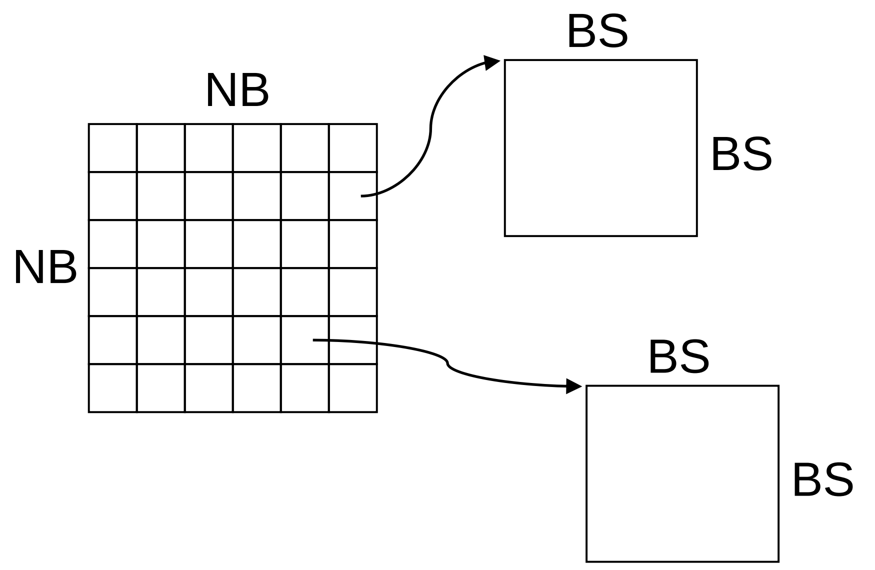


OmpSs Programming examples



MxM on matrix stored by blocks

```
int main  (int argc, char **argv) {
int i, j, k;
…

initialize(A, B, C);

for (i=0; i < NB; i++)
   for (j=0; j < NB; j++)
      for (k=0; k < NB; k++)
         mm_tile( C[i][j], A[i][k], B[k][j]);
}
```



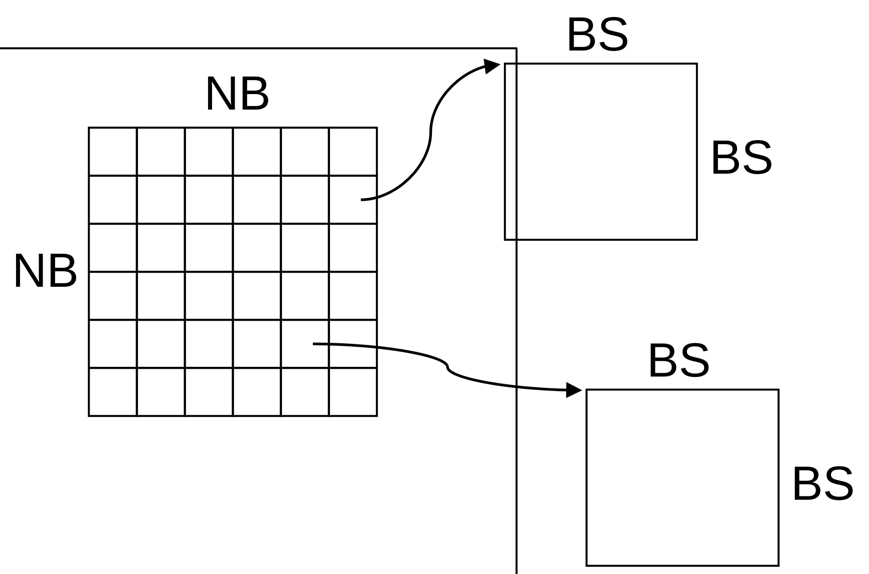
```
#pragma omp task input([BS][BS]A, [BS][BS]B)\
              inout([BS][BS]C)
static void mm_tile ( float C[BS][BS], float A[BS][BS],
                      float B[BS][BS]) {
int i, j, k;

for (i=0; i < BS; i++)
   for (j=0; j < BS; j++)
      for (k=0; k < BS; k++)
         C[i][j] += A[i][k] * B[k][j];
}
```

Will work on matrices of any size

Will work on any number of cores/devices

MxM on matrix stored by blocks

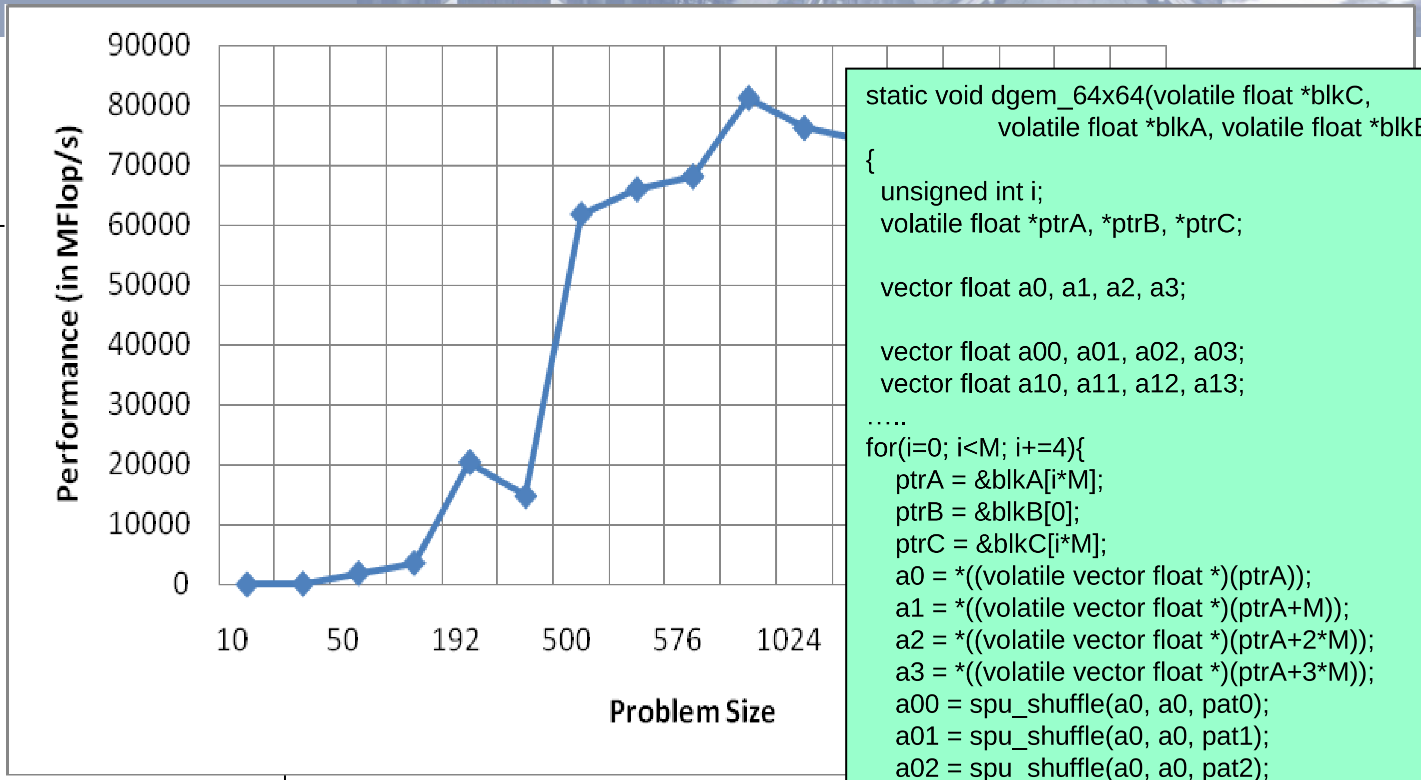
```
int main  (int argc, char **argv) {
int i, j, k;
int l, m, n;
…

initialize(A, B, C);

for (i=0; i < NB; i++)
   for (j=0; j < NB; j++)
      for (k=0; k < NB; k++)
          #pragma omp task input([BS][BS]A[i][k], [BS][BS]B[k][j])\
                           inout([BS][BS]C[i][j])
          for (l=0; l < BS; l++)
             for (m=0; m < BS; m++)
                for (n=0; n < BS; n++)
                   (*C[i][j])[l][m] += (*A[i][k])[l][n] * (*B[k][j])[n][m];
}
```



Without outlining

MxM @ CellSs

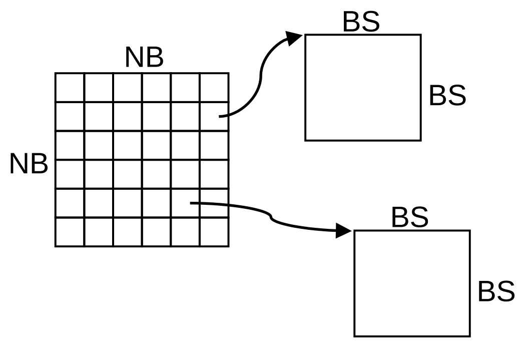
```
int main  (int argc, char **argv) {
int i, j, k;
…

initialize(A, B, C);

for (i=0; i < NB; i++)
   for (j=0; j < NB; j++)
      for (k=0; k < NB; k++)
         dgem_64x64( C[i][j], A[i][k], B[k][j]);
}
```



```
static void dgem_64x64(volatile float *blkC,
            volatile float *blkA, volatile float *blkB)
{
  unsigned int i;
  volatile float *ptrA, *ptrB, *ptrC;

  vector float a0, a1, a2, a3;

  vector float a00, a01, a02, a03;
  vector float a10, a11, a12, a13;
…..
for(i=0; i<M; i+=4){
    ptrA = &blkA[i*M];
    ptrB = &blkB[0];
    ptrC = &blkC[i*M];
    a0 = *((volatile vector float *)(ptrA));
    a1 = *((volatile vector float *)(ptrA+M));
    a2 = *((volatile vector float *)(ptrA+2*M));
    a3 = *((volatile vector float *)(ptrA+3*M));
    a00 = spu_shuffle(a0, a0, pat0);
    a01 = spu_shuffle(a0, a0, pat1);
    a02 = spu_shuffle(a0, a0, pat2);
    a03 = spu_shuffle(a0, a0, pat3);
    a10 = spu_shuffle(a1, a1, pat0);
    a11 = spu_shuffle(a1, a1, pat1);
……
 a33 = spu_shuffle(a3, a3, pat3);
    Loads4RegSetA(0);
    Ops4RegSetA();
    Loads4RegSetB(8);
    StageCBA(0,0);
    StageACB(8,0);
    StageBAC(16,0);
    StageCBA(24,0);
    StageACB(32,0);
    StageBAC(40,0);
    StageMISC(0,0);
    StageCBAmod(0,4);
    StageACB(8,4);
    StageBAC(16,4);
    StageCBA(24,4);
    StageACB(32,4);
    StageBAC(40,4);
    StageMISC(4,4);
    StageCBAmod(0,8);
    StageACB(8,8);
…..
```

```
#ifdef SPU_CODE
#include <blas_s.h>
#endif
```

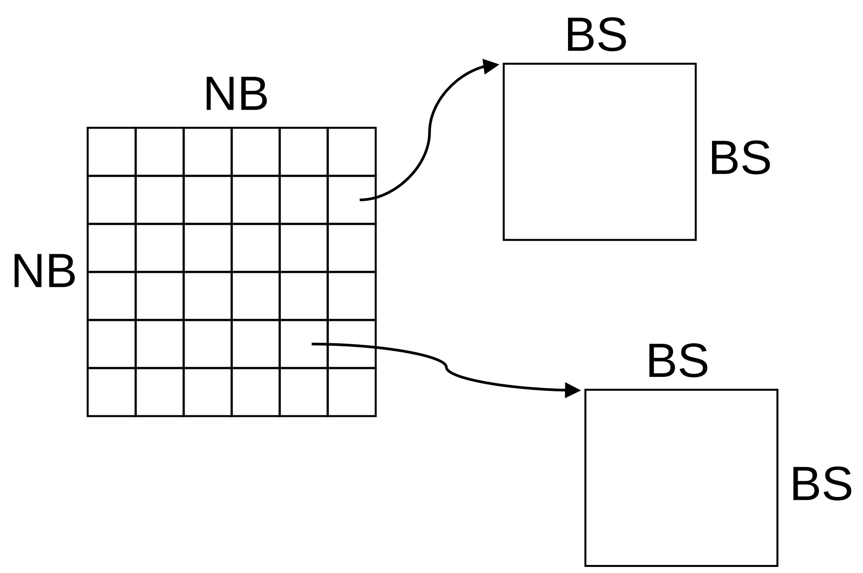
```
#define StageCBA(OFFSET,OFFB)
{                                                             \
  ALIGN8B;                                                    \
  SPU_FMA(c0_0B,a00,b0_0B,c0_0B); c0_0C = *((volatile vector float *)(
  SPU_FMA(c1_0B,a10,b0_0B,c1_0B); c1_0C = *((volatile vector float *)(
  SPU_FMA(c2_0B,a20,b0_0B,c2_0B); c2_0C = *((volatile vector float *)(
  SPU_FMA(c3_0B,a30,b0_0B,c3_0B); SPU_LNOP;
  SPU_FMA(c0_1B,a00,b0_1B,c0_1B); c3_0C = *((volatile vector float *)(
  SPU_FMA(c1_1B,a10,b0_1B,c1_1B); c0_1C = *((volatile vector float *)(
  SPU_FMA(c2_1B,a20,b0_1B,c2_1B); c1_1C = *((volatile vector float *)(ptrC+M+OFFSET+20));      \
  SPU_FMA(c3_1B,a30,b0_1B,c3_1B); SPU_LNOP;                                   \
  SPU_FMA(c0_0B,a01,b1_0B,c0_0B); c2_1C = *((volatile vector float *)(ptrC+2*M+OFFSET+20));    \
…….
```



Leverage existing kernels, libraries,…

MxM @ GPUs using CUBLAS kernel

```
int main  (int argc, char **argv) {
int i, j, k;
…

initialize(A, B, C);

for (i=0; i < NB; i++)
  for (j=0; j < NB; j++)
    for (k=0; k < NB; k++)
      mm_tile( C[i][j], A[i][k], B[k][j], BS);
}
```



```
#pragma omp target device (cuda) copy_deps
#pragma omp task input([NB][NB]A, [NB][NB]B) \
                 inout([NB][NB]C)
void mm_tile (float *A, float *B, float *C, int NB)
{
  unsigned char TR = 'T', NT = 'N';
  float DONE = 1.0, DMONE = -1.0;
  float *d_A, *d_B, *d_C;

  cublasSgemm (NT, NT, NB, NB, NB, DMONE, A,
               NB, B, NB, DONE, C, NB);
}
```

MxM @ GPUs using CUDA kernel

```
int main  (int argc, cha
int i, j, k;
…

initialize(A, B, C);

for (i=0; i < NB; i++)
  for (j=0; j < NB; j++)
    for (k=0; k < NB; k+
      mm_tile( C[i][j], 
}
```

```
#pragma omp target device (
#pragma omp task input([NB]
void mm_tile (float *A, flo
{
   int hA, wA, wB;
   hA = NB; wA = NB; wB = N

   dim3 dimBlock(BLOCK_SIZE
dimBlock.x = BLOCK_SIZE;
dimBlock.y = BLOCK_SIZE;
   dim3 dimGrid;
dimGrid.x = (wB / dimBlock.x);
dimGrid.y = (hA / dimBlock.y);
   Muld <<<dimGrid, dimBlock>>> ( A, B, wA, wB, C );
}
```

```
__global__ void Muld(float* A, float* B, int wA, int wB, float* C) {
  int bx = blockIdx.x; int by = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;
  int aBegin = wA * BLOCK_SIZE * by;
  int aEnd   = aBegin + wA - 1;
  int aStep = BLOCK_SIZE;
  int bBegin = BLOCK_SIZE * bx;
  int bStep = BLOCK_SIZE * wB;
  float Csub = 0;

  for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep) {
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
    As[ty][tx] = A[a + wA * ty + tx];
    Bs[ty][tx] = B[b + wB * ty + tx];
    __syncthreads();

    for (int k = 0; k < BLOCK_SIZE; ++k)
      Csub += As[ty][k] * Bs[k][tx];
    __syncthreads();
  }
  int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
  C[c + wB * ty + tx] += Csub;
}
```

Differences/similarities with the direct use of CUDA

Differences

- No need to allocate GPU memory (~~cudaMalloc~~)
- No need to copy host to gpu, nor gpu to host (~~cudaMemcpy~~)
- No need for synchonization (~~cudaThreadSynchronize~~)
- Multi-GPU, multi-node for free

Similarities

- Same CUDA kernels
- Same kernel invocation methodology (<<dimgrid,dimblock>>)
- Underneath uses the CUDA runtime

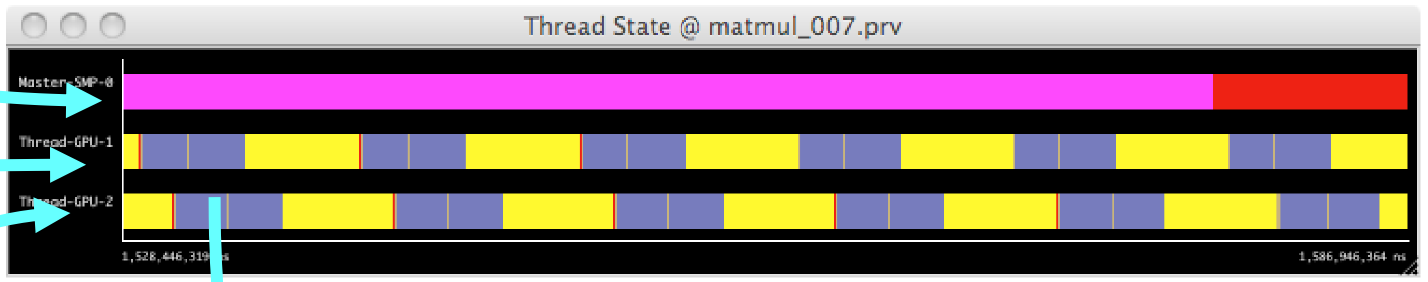
Matrix multiply: multi-GPUs

Two Intel Xeon E5440, 4 cores
4 Tesla S2050 GPUs

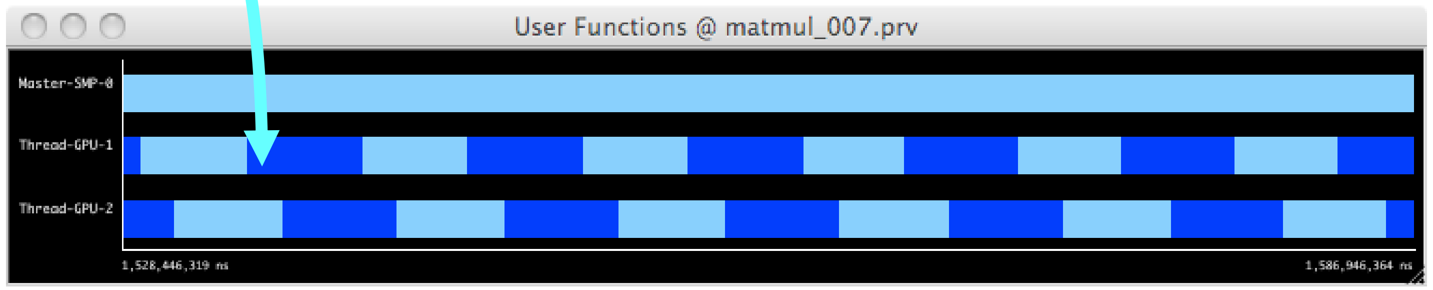
main thread

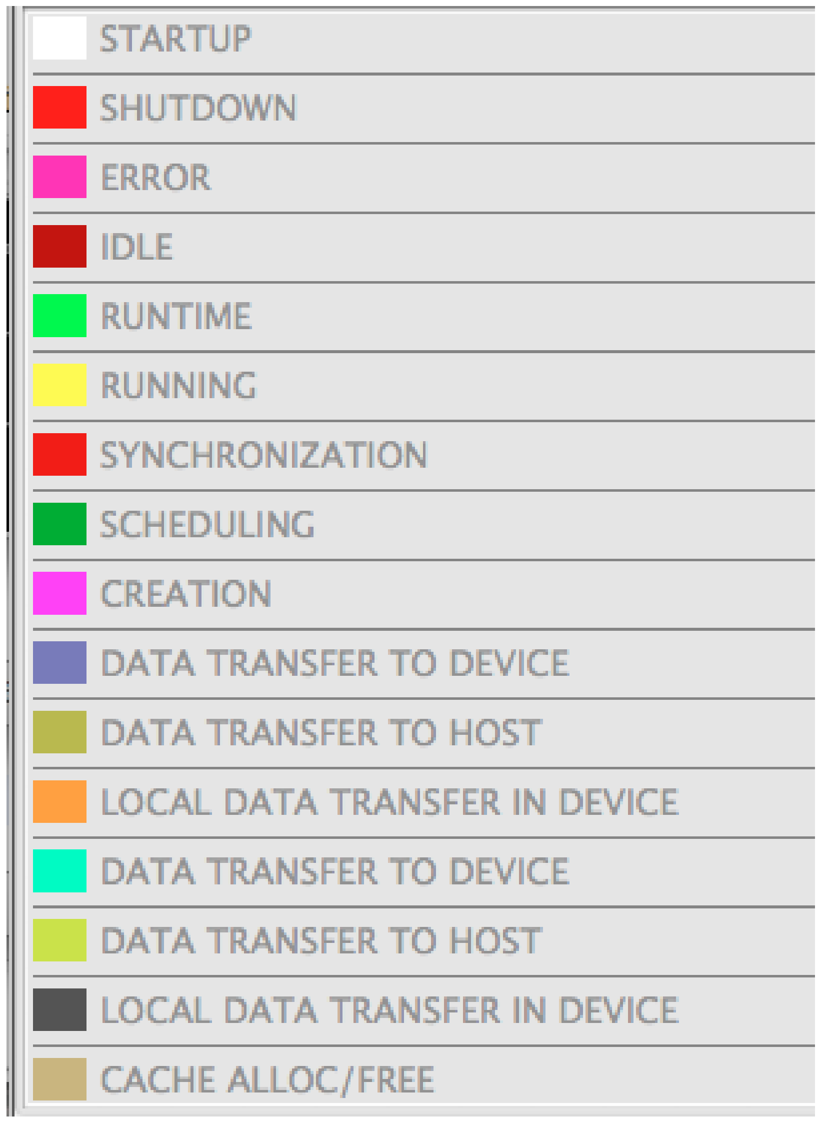
GPU thread

GPU thread



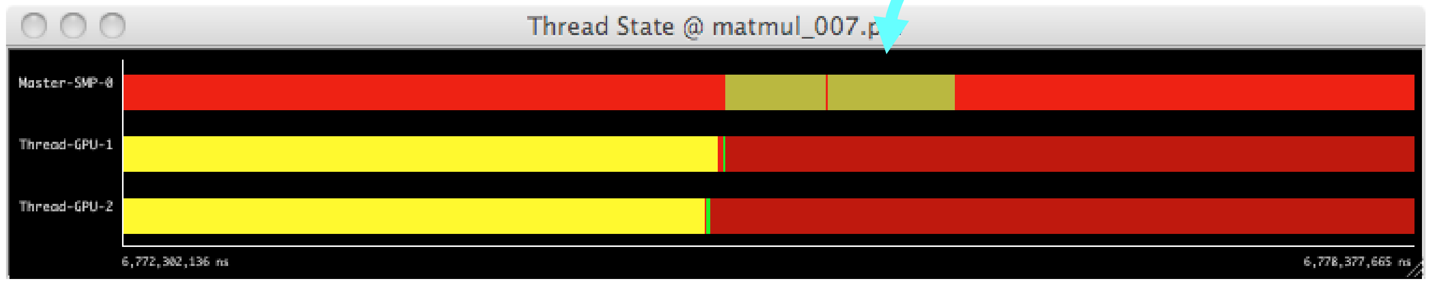
Copying data to GPU



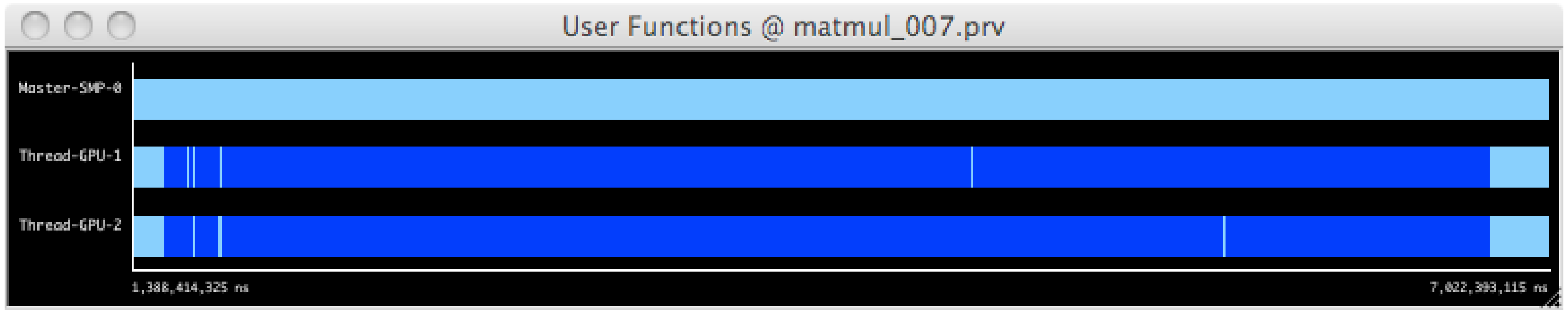


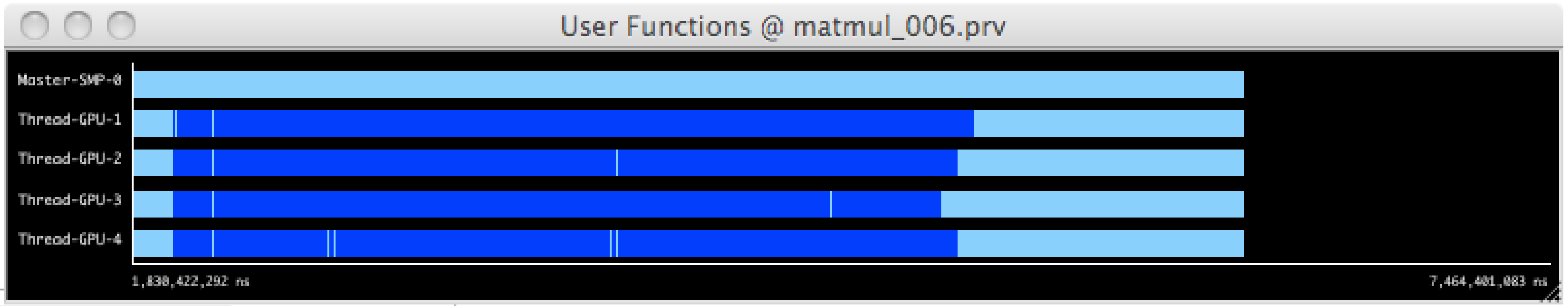
- ⬜ STARTUP
- 🟥 SHUTDOWN
- 🟪 ERROR
- 🟥 IDLE
- 🟩 RUNTIME
- 🟨 RUNNING
- 🟥 SYNCHRONIZATION
- 🟩 SCHEDULING
- 🟪 CREATION
- 🟦 DATA TRANSFER TO DEVICE
- 🟨 DATA TRANSFER TO HOST
- 🟧 LOCAL DATA TRANSFER IN DEVICE
- 🟦 DATA TRANSFER TO DEVICE
- 🟩 DATA TRANSFER TO HOST
- ⬛ LOCAL DATA TRANSFER IN DEVICE
- 🟫 CACHE ALLOC/FREE

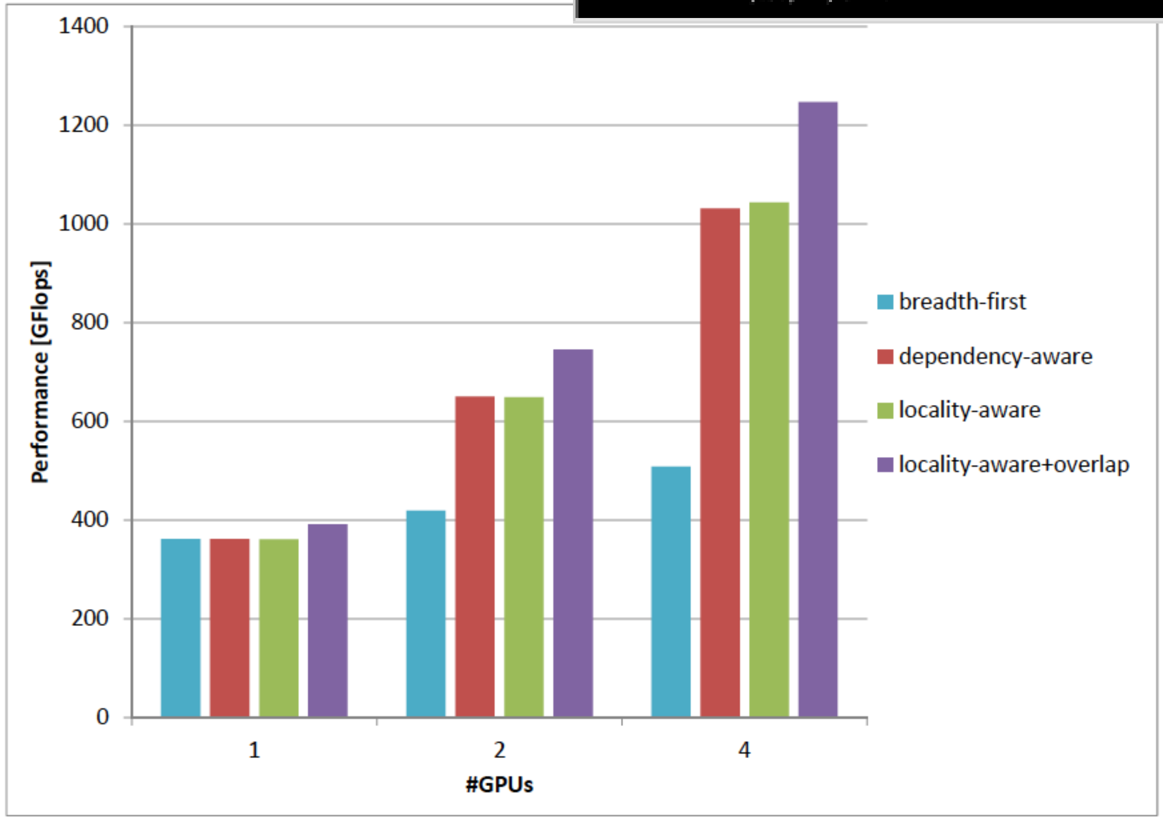
All copies to host at end of application



Matrix multiply: multi-GPUs

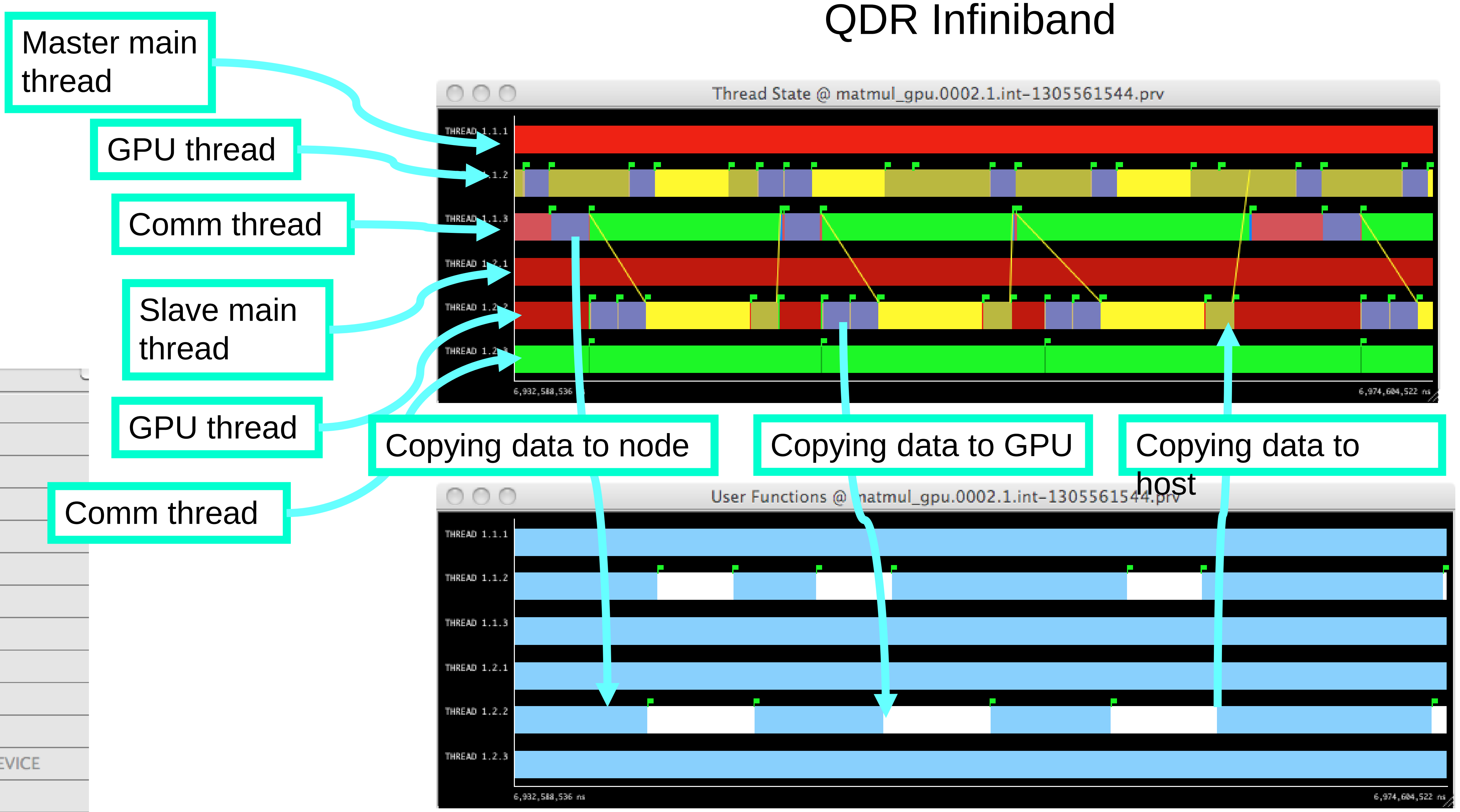


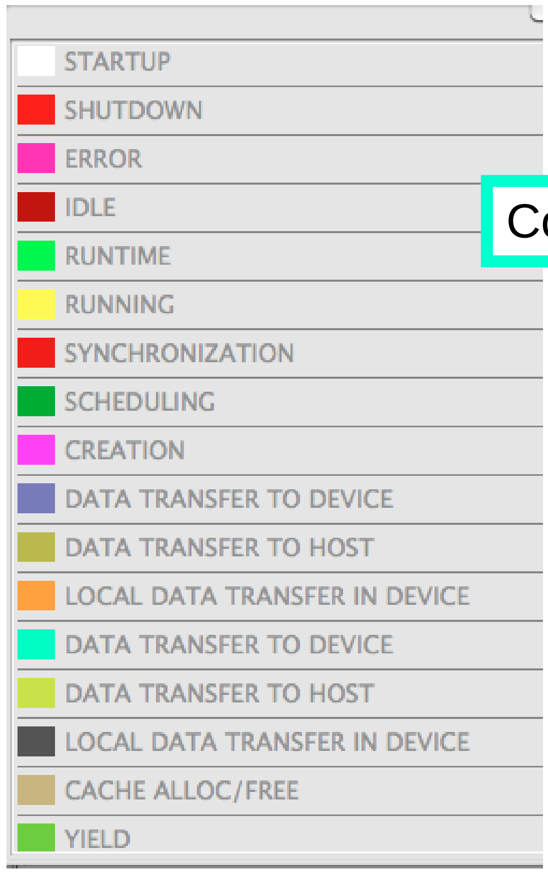




Matrix multiply: clusters of GPUs

2 nodes of DAS-4
each node:
Two Intel Xeon E5620, 4 cores
1 GTX 480 GPU
QDR Infiniband



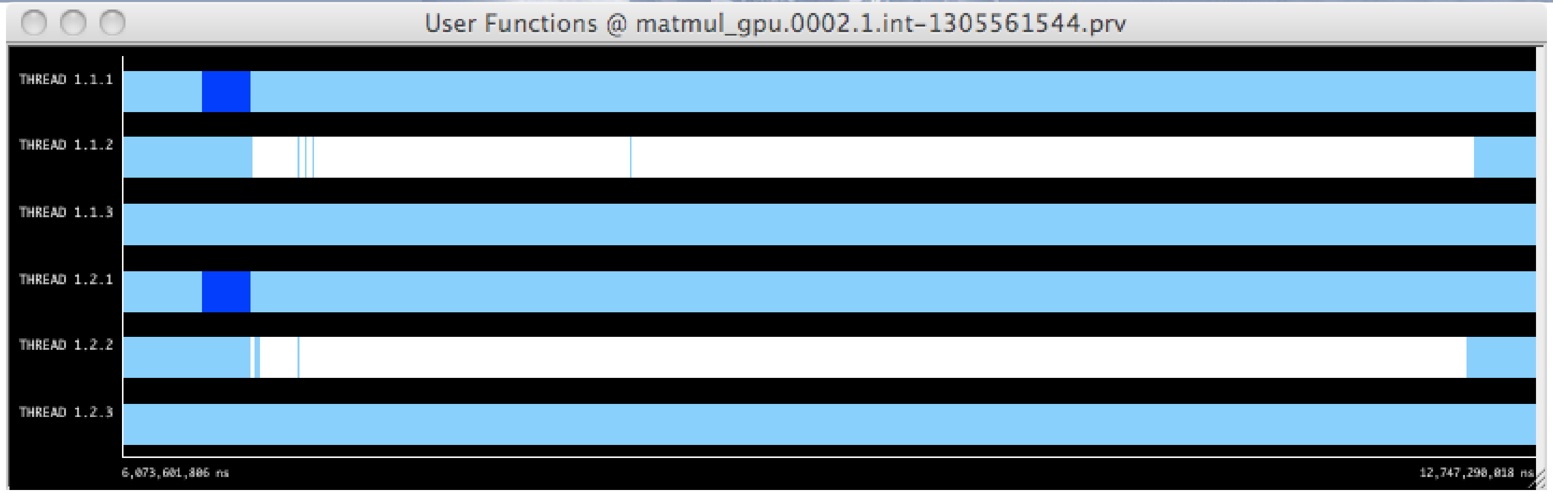


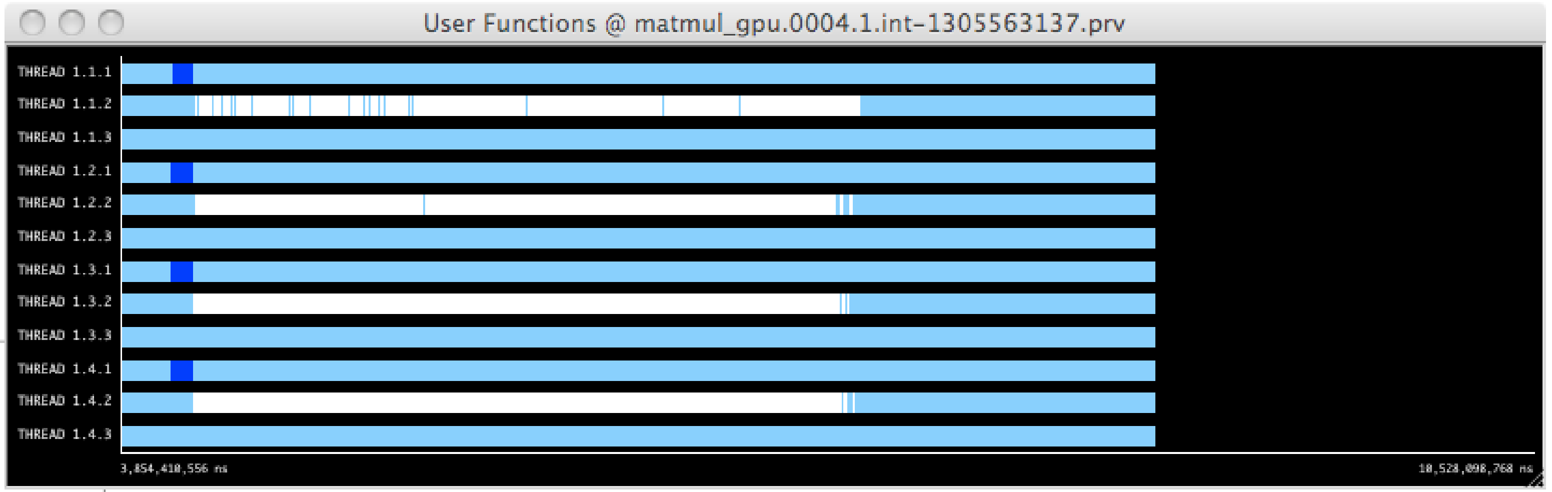
- ⬜ STARTUP
- 🟥 SHUTDOWN
- 🟪 ERROR
- 🟥 IDLE
- 🟩 RUNTIME
- 🟨 RUNNING
- 🟥 SYNCHRONIZATION
- 🟩 SCHEDULING
- 🟪 CREATION
- 🟦 DATA TRANSFER TO DEVICE
- 🟨 DATA TRANSFER TO HOST
- 🟧 LOCAL DATA TRANSFER IN DEVICE
- 🟩 DATA TRANSFER TO DEVICE
- 🟨 DATA TRANSFER TO HOST
- ⬛ LOCAL DATA TRANSFER IN DEVICE
- 🟫 CACHE ALLOC/FREE
- 🟩 YIELD

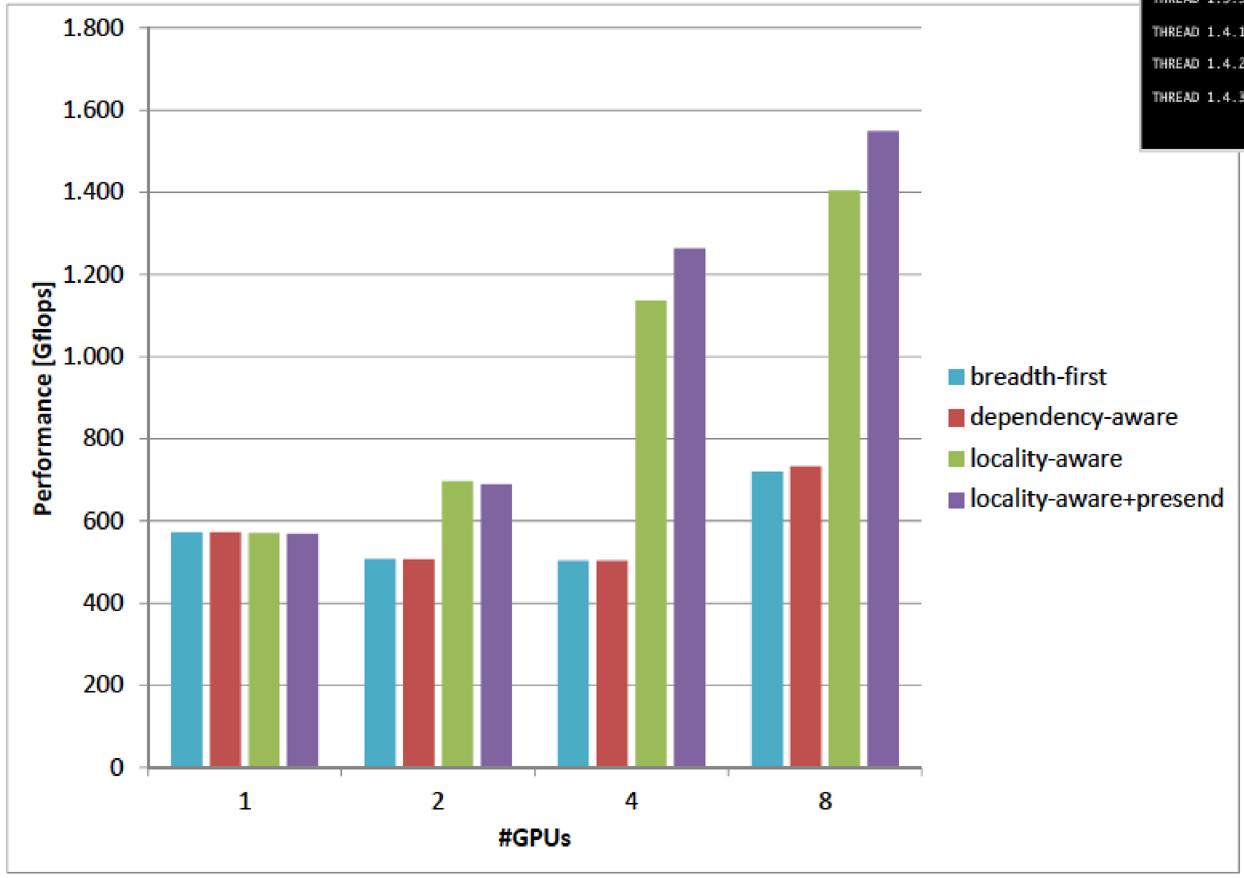
- 🟦 Outline from 'gendat.c:25' in 'gendat'
- ⬜ Task 'matmul_tile'

Matrix multiply: clusters of GPUs

2 and 4 nodes of DAS-4
each node:
Two Intel Xeon E5620, 4 cores
1 GTX 480 GPU
QDR Infiniband







Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Cholesky factorization

- Cholesky factorization
- Common matrix operation used to solve normal equations in linear least squares problems.
- Calculates a triangular matrix (L) from a symetric and positive defined matrix A.

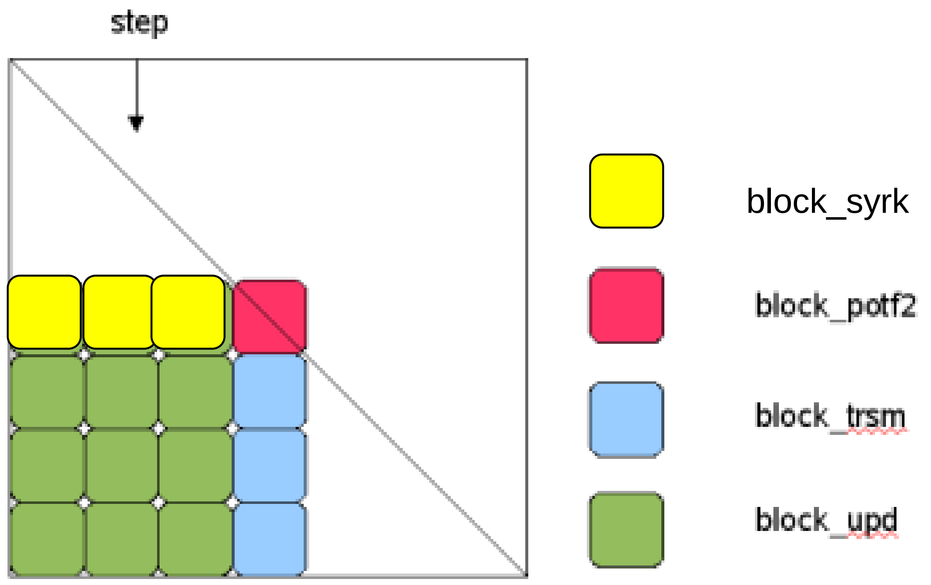
$$\text{Cholesky}(A) = L$$

$$L \cdot L^t = A$$

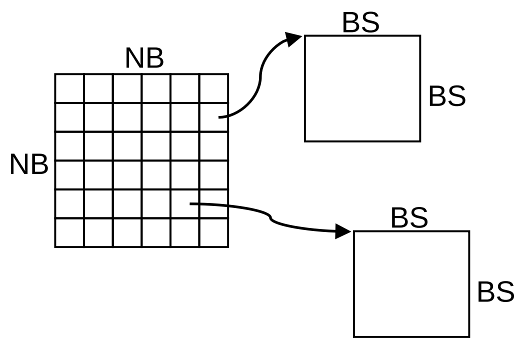
- Different possible implementations, depending on how the matrix is traversed (by rows, by columns, left-looking, right-looking)
- It can be decomposed in block operations

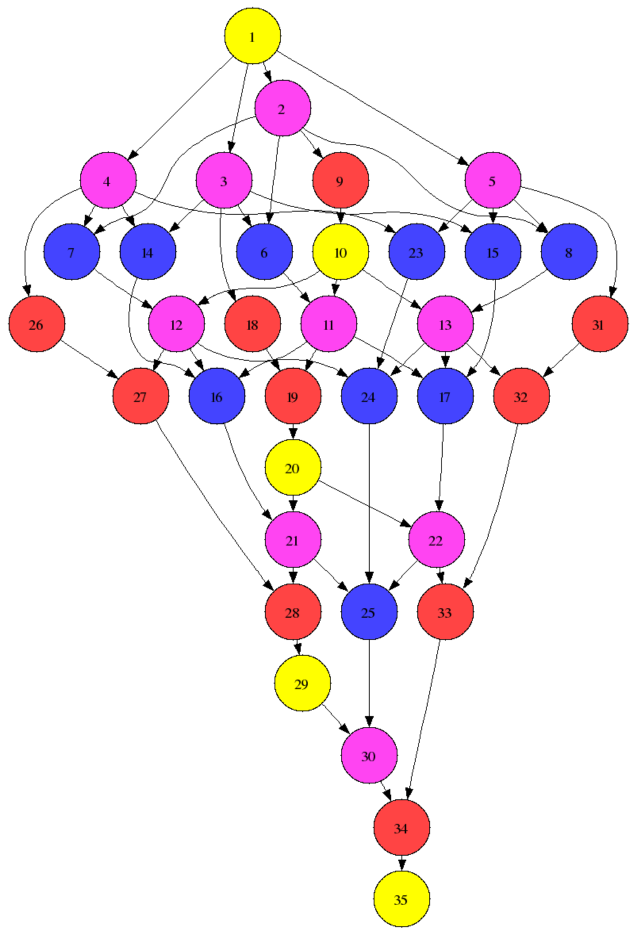
Cholesky factorization

- In each iteration red and blue blocks are updated
  - SPOTRF: Computes the Cholesky factorization of the diagonal block .
  - STRSM: Computes the column panel
  - SSYRK: Computes the row panel
  - SGEMM: Updates the rest of the matrix



Cholesky: Block matrix storage



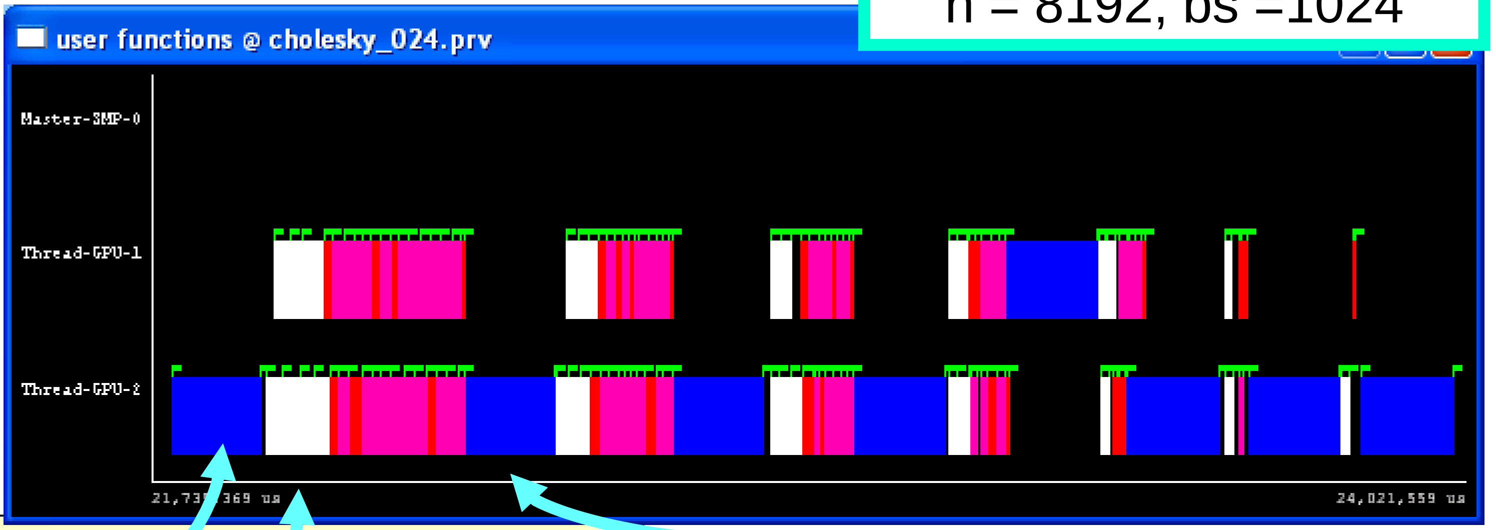


```
void blocked_cholesky( int NT, float *A ) {
   int i, j, k;
   for (k=0; k<NT; k++) {
      spotrf (A[k*NT+k]) ;
      for (i=k+1; i<NT; i++)
         strsm (A[k*NT+k], A[k*NT+i]);
      // update trailing submatrix
      for (i=k+1; i<NT; i++) {
         for (j=k+1; j<i; j++)
            sgemm( A[k*NT+i], A[k*NT+j], A[j*NT+i]);
         ssyrk (A[k*NT+i], A[i*NT+i]);
   }
#pragma omp taskwait
}
```

```
🟡 #pragma omp task inout ([TS][TS]A)
   void spotrf (float *A);
🔵 #pragma omp task input ([TS][TS]A) inout ([TS][TS]C)
   void ssyrk (float *A, float *C);
🔴 #pragma omp task input ([TS][TS]A, [TS][TS]B) inout ([TS][TS]C)
   void sgemm (float *A, float *B, float *C);
🟣 #pragma omp task input ([TS][TS]T) inout ([TS][TS]B)
   void strsm (float *T, float *B);
```

Cholesky: Block matrix storage @ GPU

- Source code independent of # devices



n = 8192; bs =1024

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

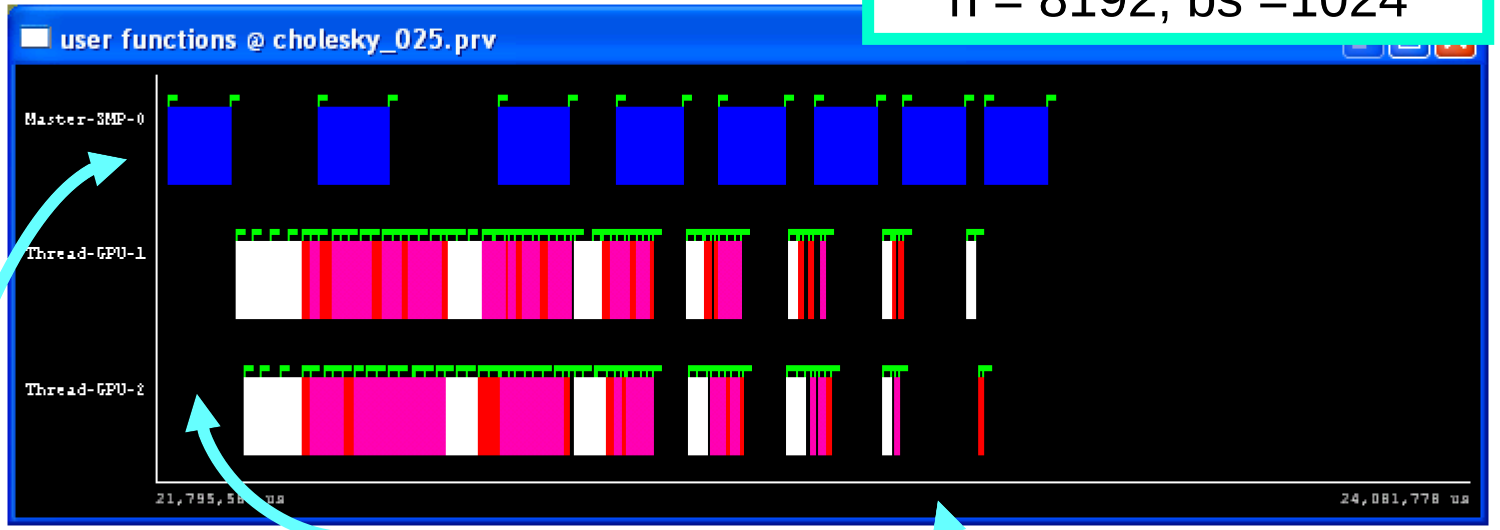
```
void blocked_cholesky( int NT, float *A ) {
   int i, j, k;
   for (k=0; k<NT; k++) {
      spotrf (A[k*NT+k]) ;
      for (i=k+1; i<NT; i++)
         strsm (A[k*NT+k], A[k*NT+i]);
      // update trailing submatrix
      for (i=k+1; i<NT; i++) {
         for (j=k+1; j<i;
            sgemm( A[k*NT
         ssyrk (A[k*NT+i]
   }
#pragma omp taskwait
}
```

```
#pragma omp target device (cuda) copy_deps
#pragma omp task inout ([TS][TS]A)
void spotrf (float *A);
#pragma omp target device (cuda) copy_deps
#pragma omp task input ([TS][TS]A) inout ([TS][TS]C)
void ssyrk (float *A, float *C);
#pragma omp target device (cuda) copy_deps
#pragma omp task input ([TS][TS]A, [TS][TS]B) inout ([TS][TS]C)
void sgemm (float *A, float *B, float *C);
#pragma omp target device (cuda) copy_deps
#pragma omp task input ([TS][TS]T) inout ([TS][TS]B)
void strsm (float *T, float *B);
```

Cholesky: Heterogeneous execution

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

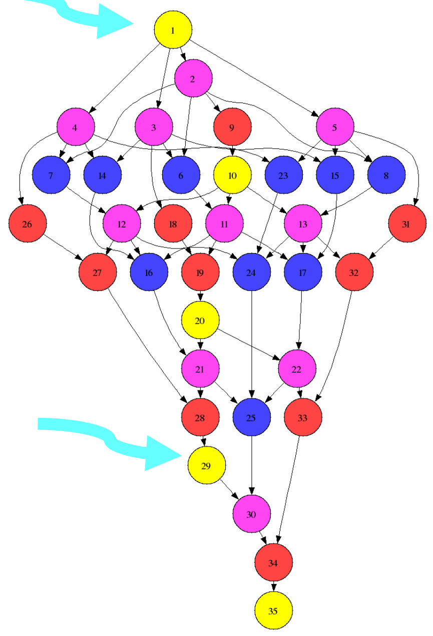
n = 8192; bs =1024



Late start due to data-dependences

Not enough concurrency

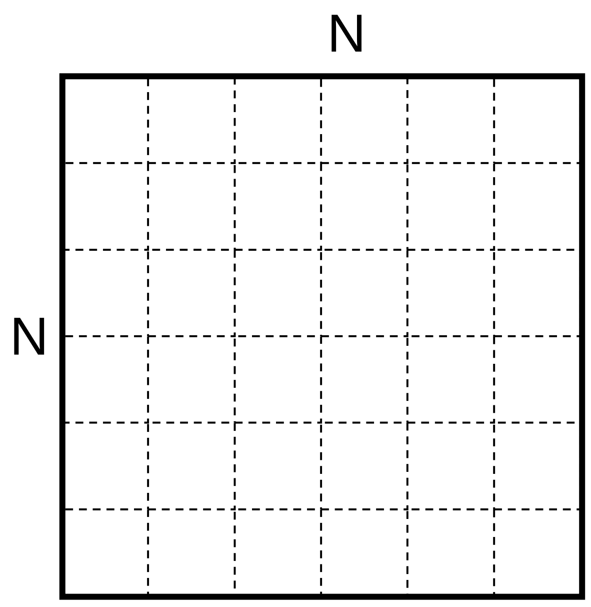
```
#pragma omp target device (smp) copy_deps
#pragma omp task inout([NB][NB]A)
void spotrf_tile(float *A, int NB)
{
    long INFO;
    char L = 'L';

    spotrf_( &L, &NB, A, &NB, &INFO );
}
```

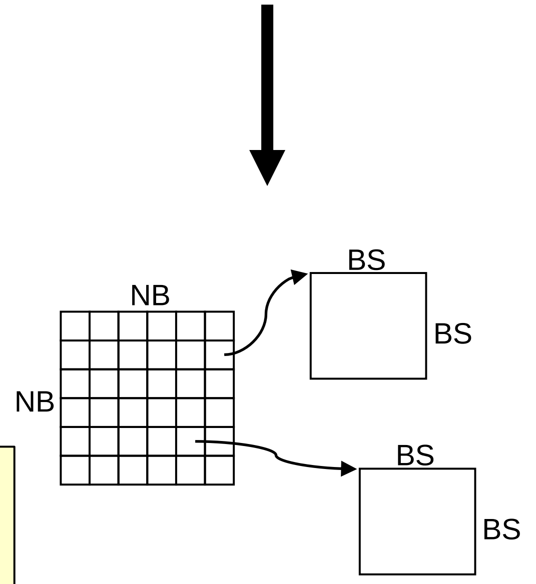


Cholesky: Standard row-wise matrix association

```
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 omp taskwait
   free_block_matrix(Ah)
}
```

Local memory management
Temporary work arrays





```
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);
}
```

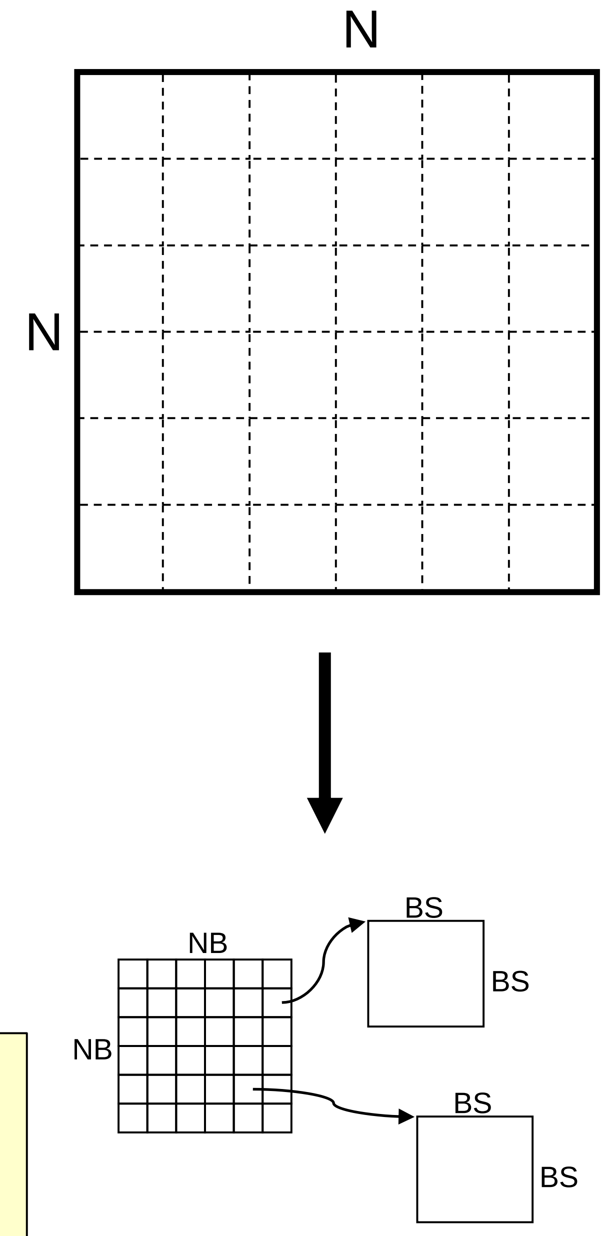
```
#pragma omp task input ([n][n]A) output ([bs][bs]bA)
void gather_block (int n, float *A, int i, int j, float *bA);
#pragma omp task input ( [bs][bs]zA) inout ([n][n]A) concurrent(A)
void scatter_block (int n, bs, float *bA, float *A, i,j);
```

Cholesky: The elegance of nesting …

```
#pragma omp task inout ([n][n]A)
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 omp taskwait
   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 omp task input ([n][n]A) output ([bs][bs]bA)
void gather_block (int n, float *A, int i, int j, float *bA);
#pragma omp task input ( [bs][bs]zA) inout ([n][n]A) concurrent(A)
void scatter_block (int n, bs, float *bA, float *A, i,j);
```

… and recursion

```
#pragma omp task inout ([n][n]A)
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 omp taskwait
   free_block_matrix(Ah)
}
```

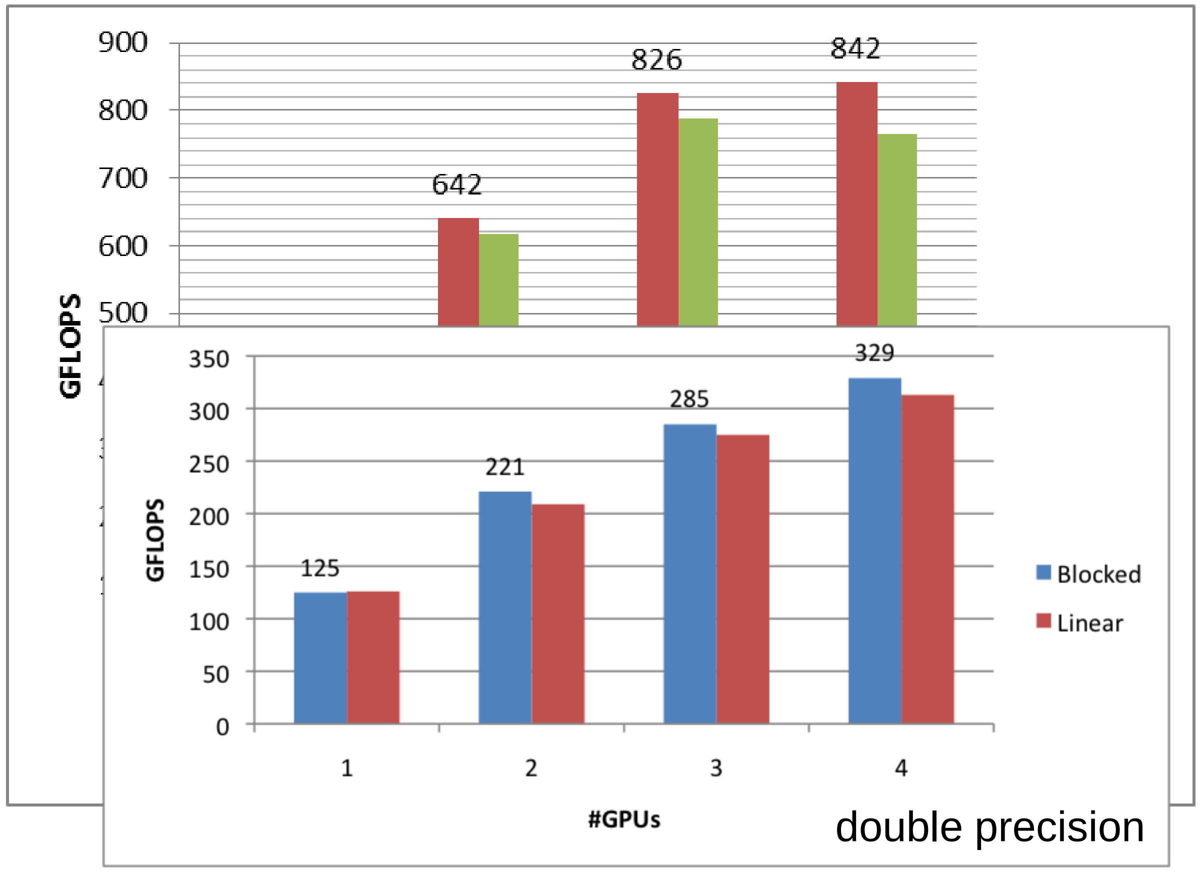
Recursion, a nice way to
refine parallelism
reduce granularity

Algorithmic level
Enables many potential execution schedules

Cholesky performance

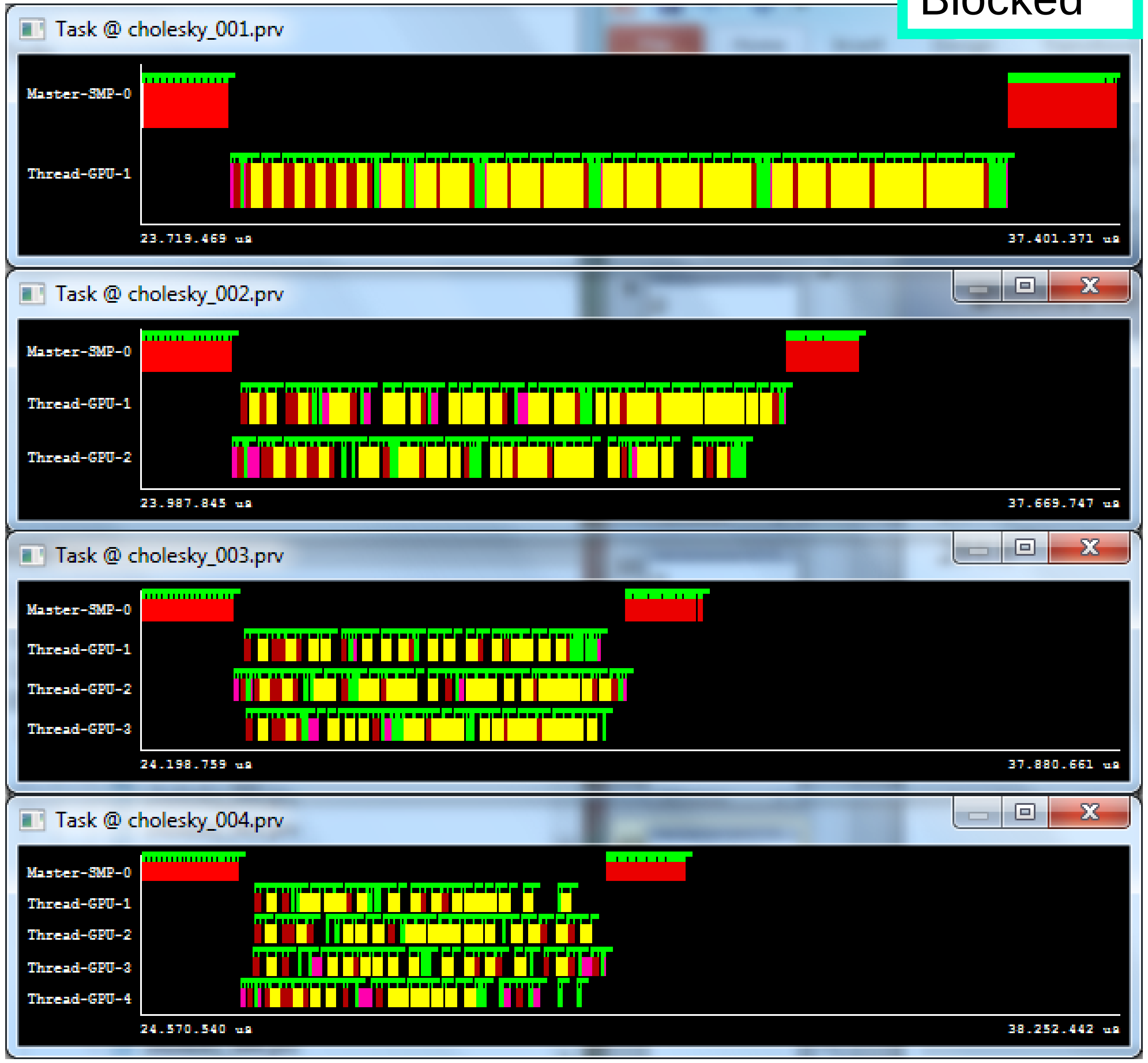
Two Intel Xeon E5440, 4 cores
4 Tesla S2050 GPUs

- Matrix size: 16K x 16K
- Block size: 2K x 2K
- Storage: Blocked / linear
- Tasks:
  - spotrf: Magma
  - trsm, syrk, gemm: CUBLAS



double precision

Blocked

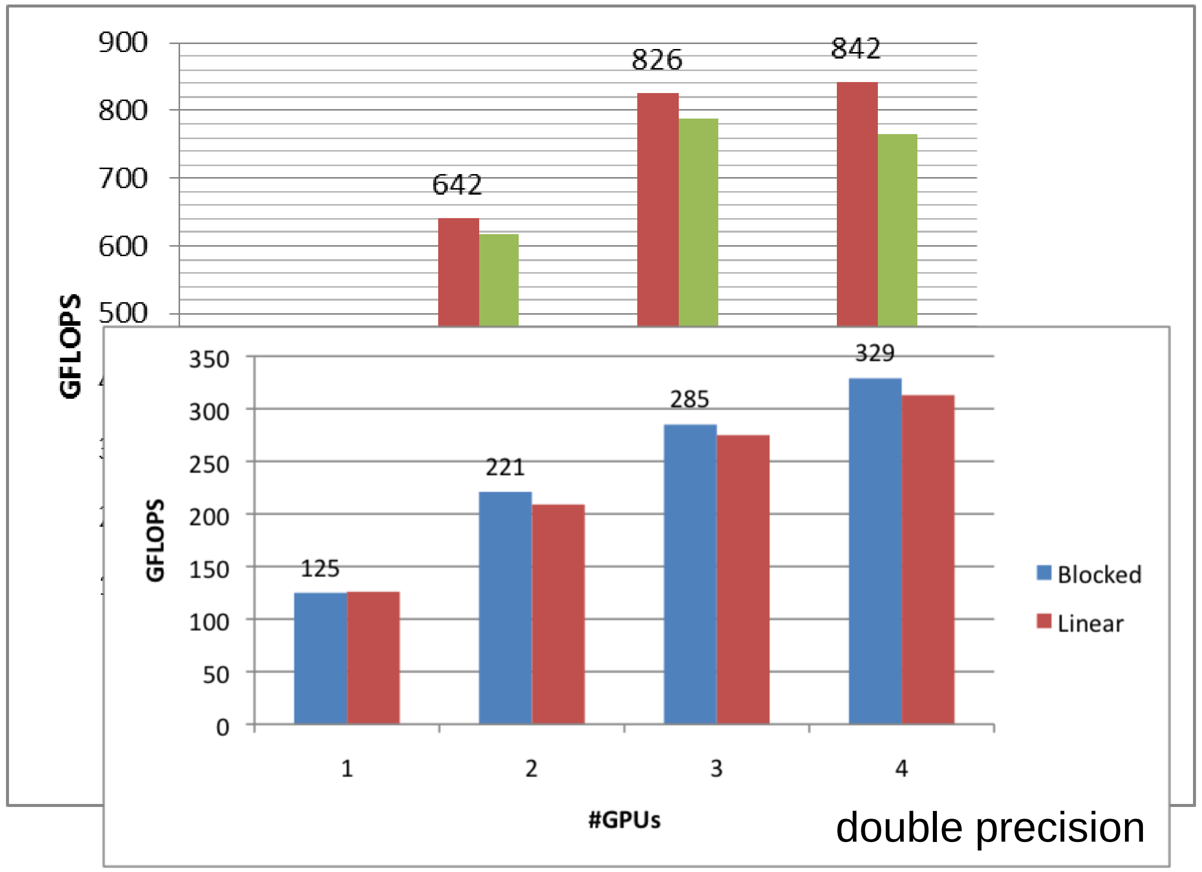


Barcelona Supercomputing Center
Centro Nacional de Supercomputación

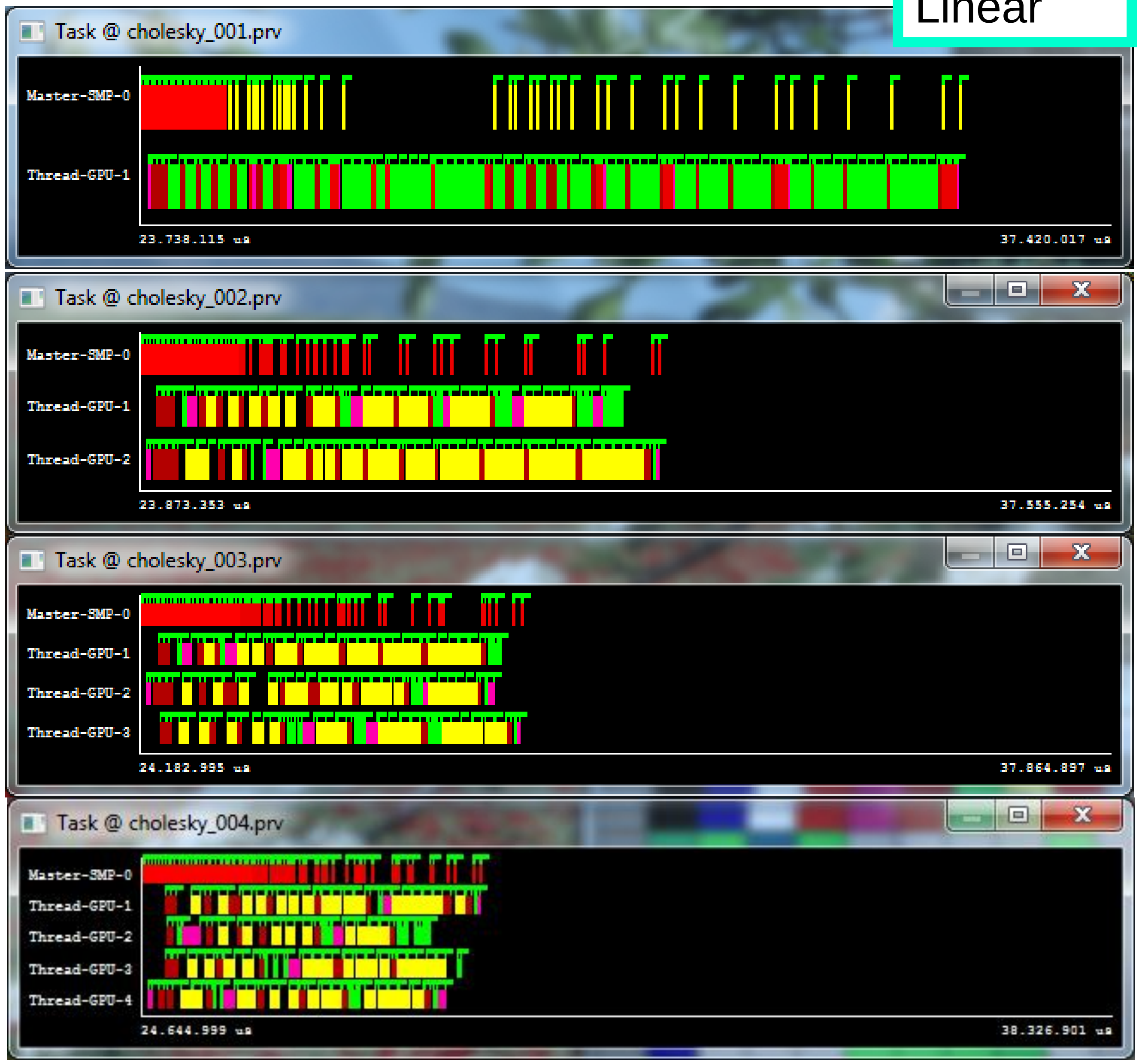
Cholesky performance

Two Intel Xeon E5440, 4 cores
4 Tesla S2050 GPUs

- Matrix size: 16K x 16K
- Block size: 2K x 2K
- Storage: Blocked / linear
- Tasks:
  - spotrf: Magma
  - trsm, syrk, gemm: CUBLAS



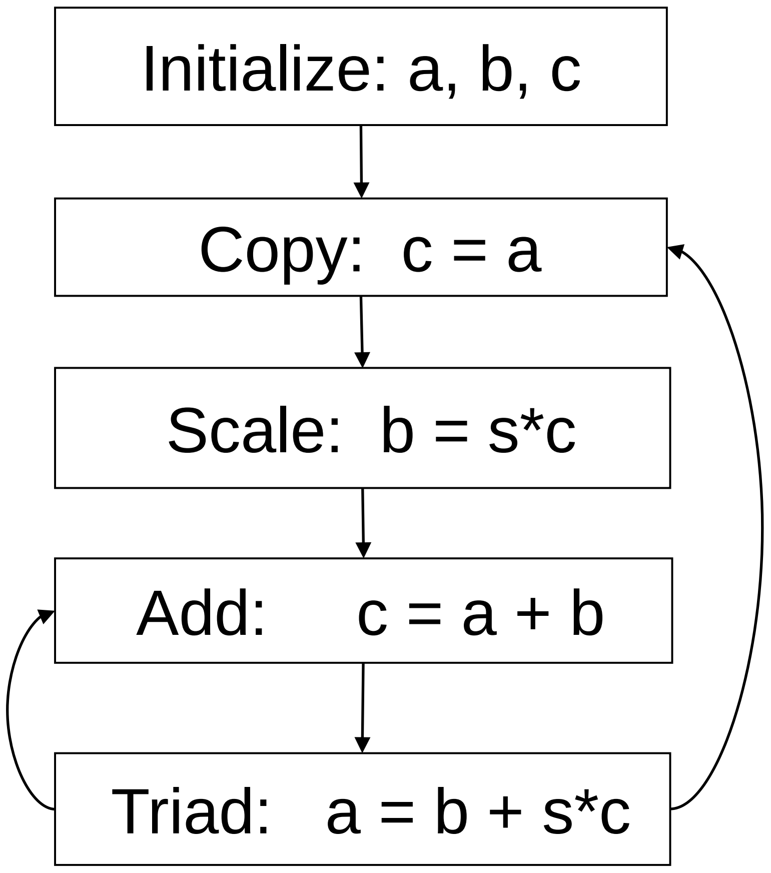
Linear



Stream

- Stream is one of the HPC Challenge benchmark suite

  http://icl.cs.utk.edu/hpcc/
- Stream is a simple synthetic benchmark program that measures sustainable memory bandwidth (in GB/s) and the corresponding computation rate for simple vector kernel.
- Original OpenMP version: inserts barriers after each operation has processed all elements of the array
- Two StarSs versions: with and without barriers
- Without barriers the correctness is guaranteed thanks to the data-dependence analysis



Stream: mimicking original version

```
 main (){
tuned_initialization();
#pragma omp taskwait
    scalar = 3.0;
    for (k=0; k<NTIMES; k++)}
        tuned_STREAM_Copy();
#pragma omp taskwait
        tuned_STREAM_Scale(scalar);
#pragma omp taskwait
        tuned_STREAM_Add();
#pragma omp taskwait
        tuned_STREAM_Triad(scalar);
#pragma omp taskwait
     }
}
```

```
#pragma omp task input ([bs]a) output ([bs]c)
void copy_task(double *a, double *c, int bs)
{  int j;
        for (j=0; j < bs; j++)   c[j] = a[j];
}
void tuned_STREAM_Copy()
{ int j;
        for (j=0; j<N; j+=BSIZE)
            copy_task (&a[j], &c[j], BSIZE);
}

#pragma omp task input ([bs]c) output ([bs]b)
void scale_task (double *b, double *c, double scalar,
      int bs)
{ int j;
        for (j=0; j < bs; j++) b[j] = scalar*c[j];
}
void tuned_STREAM_Scale(double scalar)
{ int j;
        for (j=0; j<N; j+=BSIZE)
               scale_task (&b[j], &c[j], scalar, BSIZE);
}
```

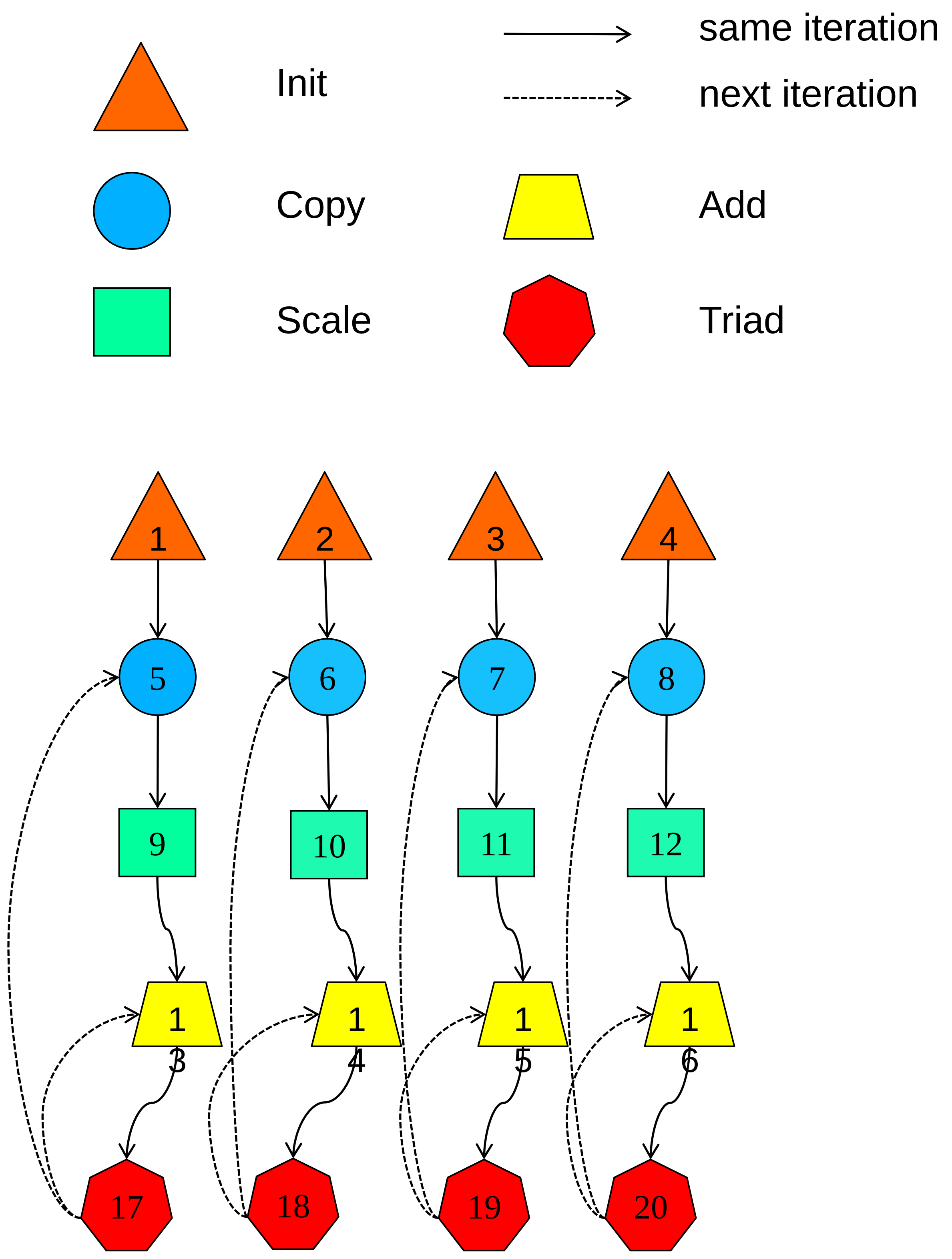
Stream

```
#pragma css task input ([bs]a, [bs]b) output ([bs]c)
void add_task (double *a, double *b, double *c, int bs)
{ int j;
        for (j=0; j < bs; j++) c[j] = a[j]+b[j];
}
void tuned_STREAM_Add()
{ int j;
        for (j=0; j<N; j+=BSIZE)
            add_task(&a[j], &b[j], &c[j], BSIZE);
}
#pragma css task input ([bs]b, [bs]c) output ([bs]a)
void triad_task (double *a, double *b, double *c, double scalar, int bs)
{ int j;
        for (j=0; j < bs; j++)
            a[j] = b[j]+scalar*c[j];
}
void tuned_STREAM_Triad(double scalar)
{ int j;
        for (j=0; j<N; j+=BSIZE)
            triad_task (&a[j], &b[j], &c[j], scalar, BSIZE);
}
```

Stream: version without barriers

- Version without barriers: SMPSs/CellSs data dependence analysis guarantees correctness

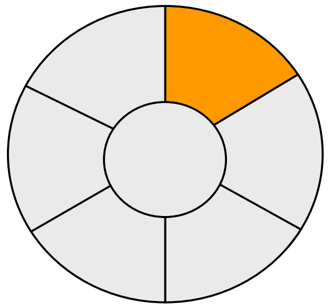
```
 main (){
tuned_initialization();
#pragma omp taskwait
    scalar = 3.0;
    for (k=0; k<NTIMES; k++)}
        tuned_STREAM_Copy();
        tuned_STREAM_Scale(scalar);
        tuned_STREAM_Add();
        tuned_STREAM_Triad(scalar);
     }
#pragma omp taskwait
   }
```



Hands-on

- Copy files from:
  - /gpfs/scratch/bsc19/bsc19776/TutorialRES2011/tutorial_RES_2011.tar.gz

Hybrid MPI/OmpSs



MPI + StarSs hybrid programming

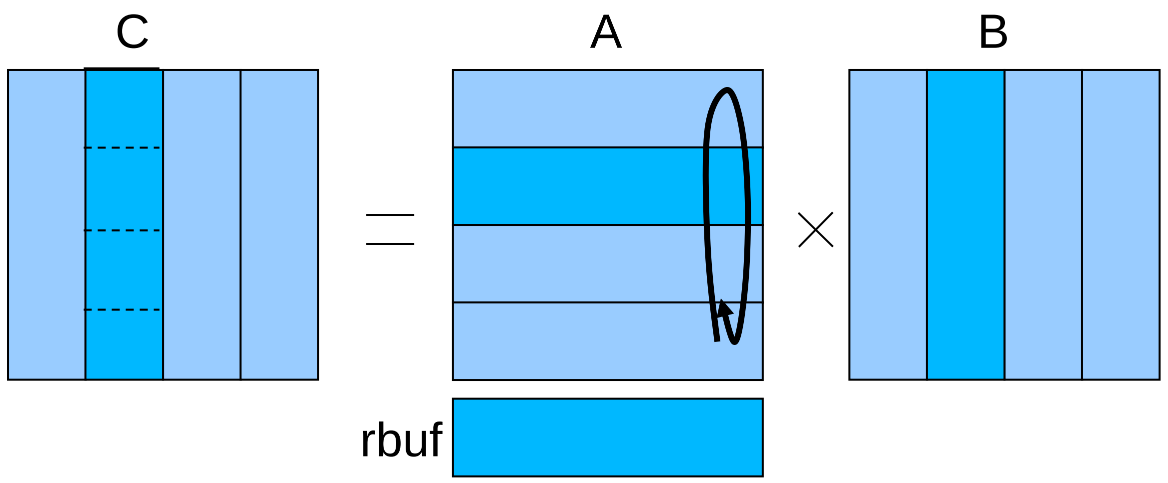
- Why?
  - MPI is here to stay.
  - A lot of HPC applications already written in MPI.
  - MPI scales well to tens/hundreds of thousands of nodes.
  - MPI exploits intra-node parallelism while StarSs can exploit node parallelism.

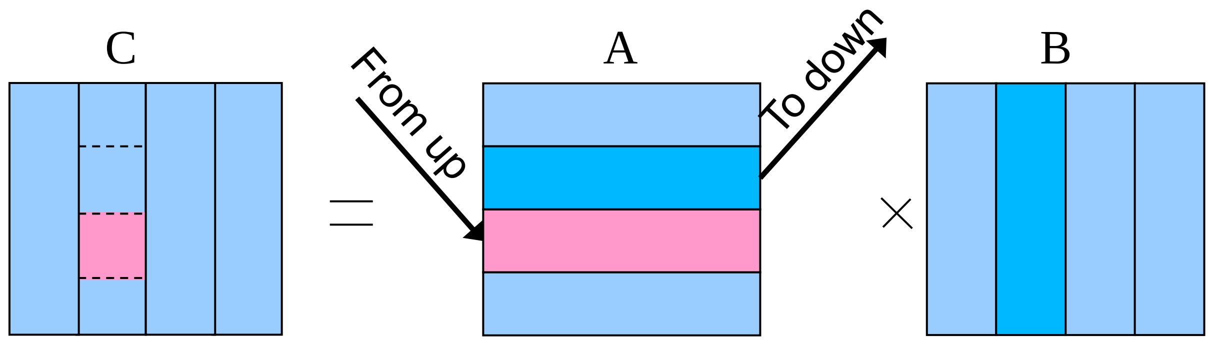
MPI + StarSs

- **Performance**: Overcomes the too synchronous structure of MPI/OpenMP, propagating the dataflow asynchronous behavior to the MPI level.
- **Flexibility**: making the application malleable
  - ntroduces the possibility of flexible resource management in otherwise rigid process structure of MPI or MPI/OpenMP
  - Supports automatic fine grain load balance, reaction to faults and dynamic system level resource management policies.
- **Productivity**: Allows for very flexible overlap with simple code structures
- **Portability**: Offers incremental parallelization even for heterogeneous systems

Mod2am: Original MPI code

```
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) );
    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 );
  }
}
```

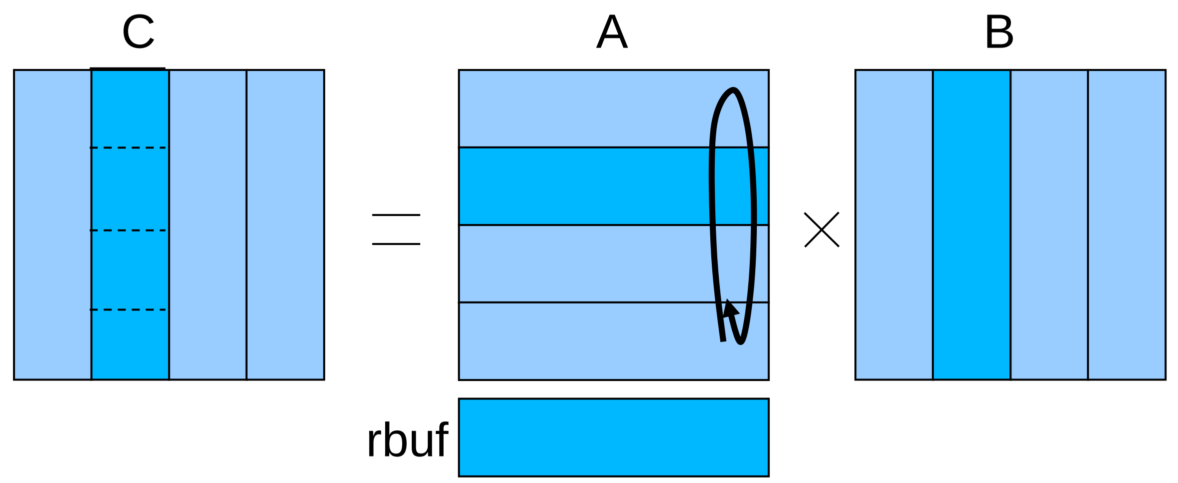


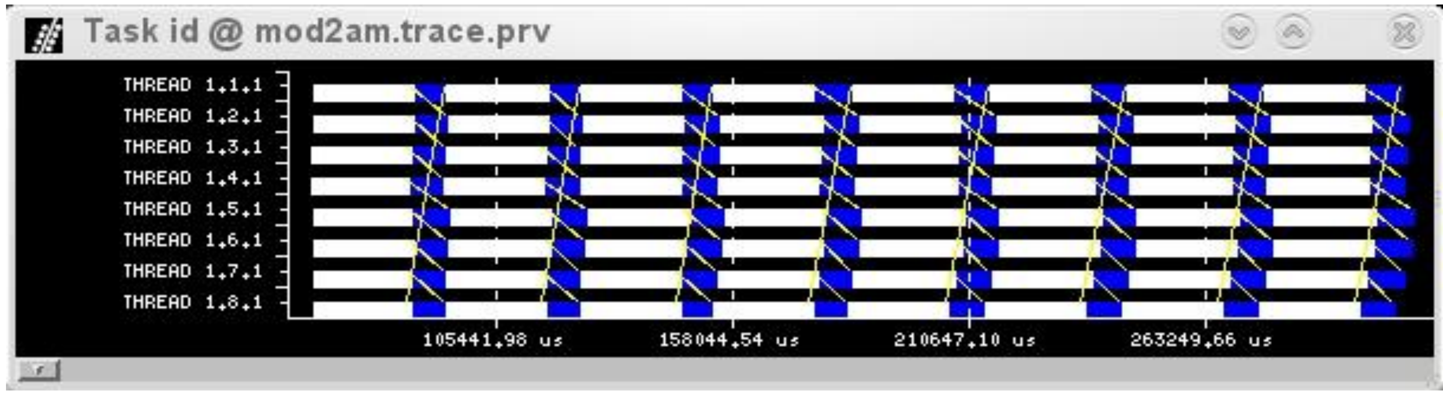


```
void mxm ( int lda, int m, int l, int n, double *a, double *b,
           double *c )
{
         double alpha=1.0, beta=1.0;
         int i, j;
         char tr = 't';
         dgemm(&tr, &tr, &m, &n, &l, &alpha, a, &lda, b, &m, &beta, c, &m);
}
```

Mod2am: Original MPI code

```
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) );
    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 );
  }
}
```

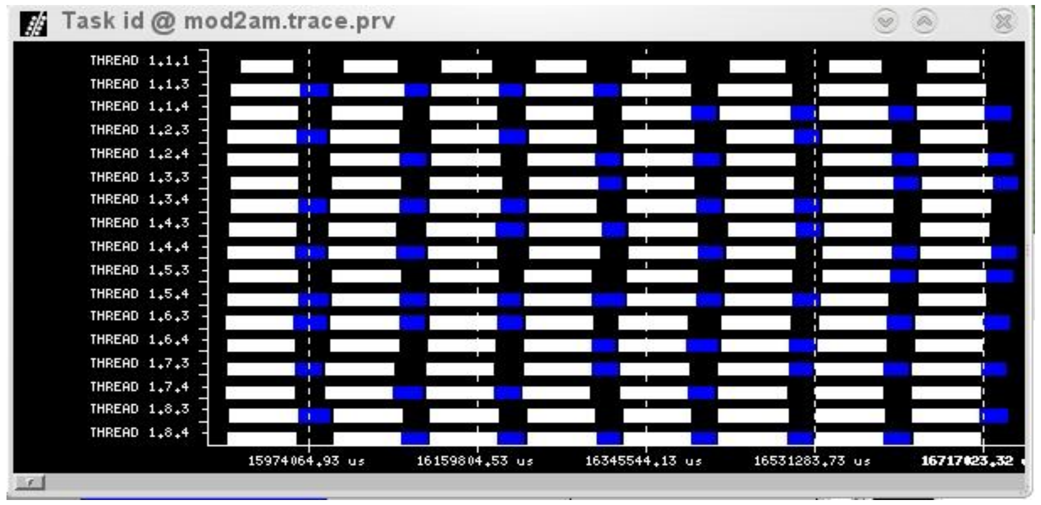




```
void mxm ( int lda, int m, int l, int n, double *a, double *b,
           double *c )
{
        double alpha=1.0, beta=1.0;
        int i, j;
        char tr = 't';
        dgemm(&tr, &tr, &m, &n, &l, &alpha, a, &lda, b, &m, &beta, c, &m);
}
```

MPI+StarSs example – Matrix multiply

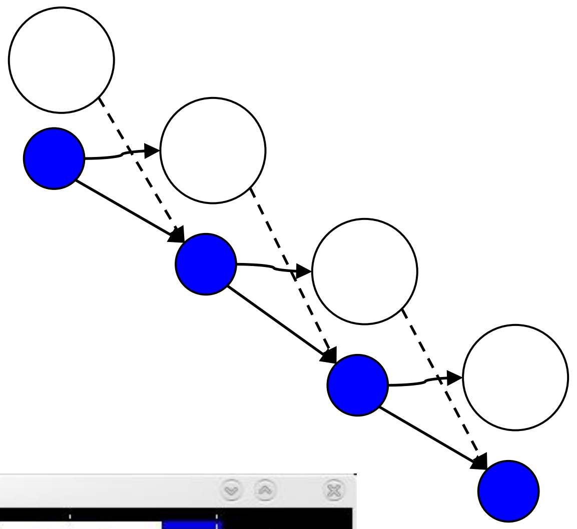
- Typical hybrid paralelization:
  - Parallelize computation phase.
  - Serialize communication part.

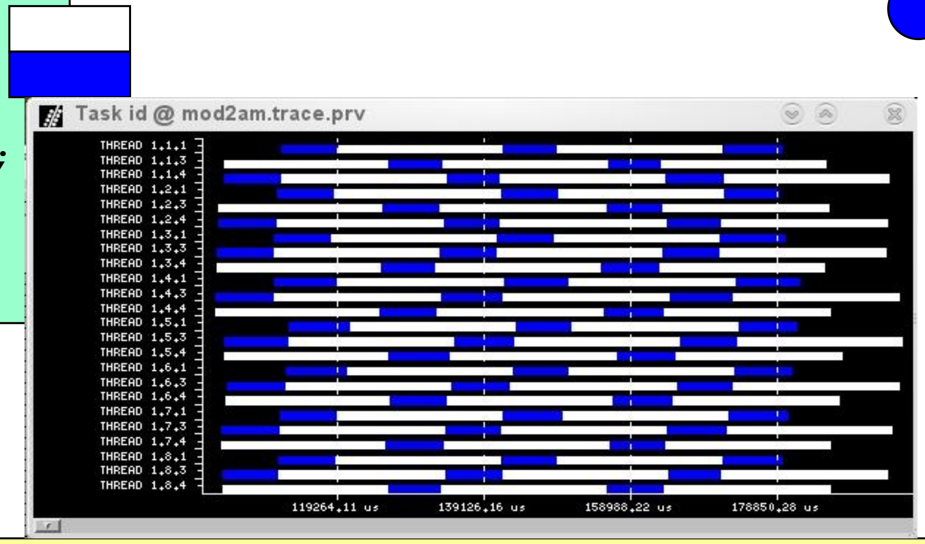


- How to overlap of communication and computation?
  - Using double buffering and
  - Asynchronous MPI calls
- Easier with StarSs...

Mod2am: Overlap communication and computation with StarSs

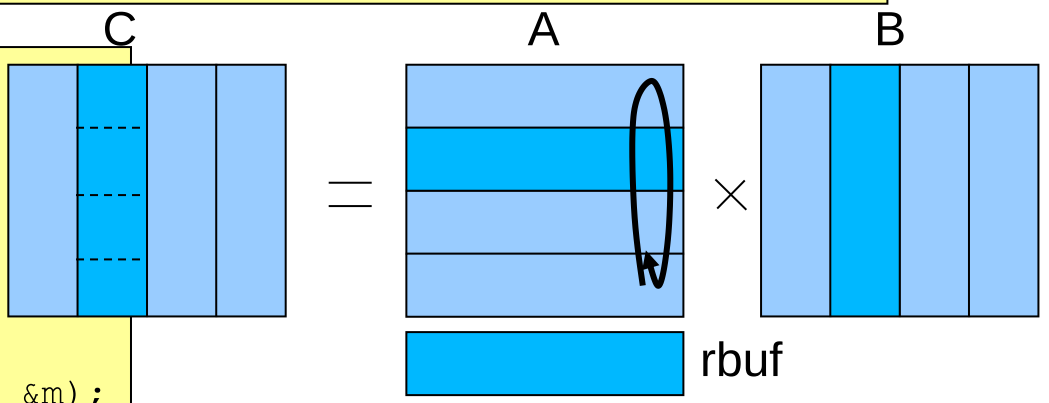
```
   for( i = 0; i < processes; i++ )
   {
     stag = i + 1; rtag = stag;
     indx = (me + nodes - i )%processes;
     shift = ((offset[indx][0]/BS)*nDIM*BS*BS);
     size = vsize*l;
     if(i%2 == 0) {
       mxm( lda, sizes[indx][0], lda, hsize, a, b, (c+shift) );
       callSendRecv (a, size,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

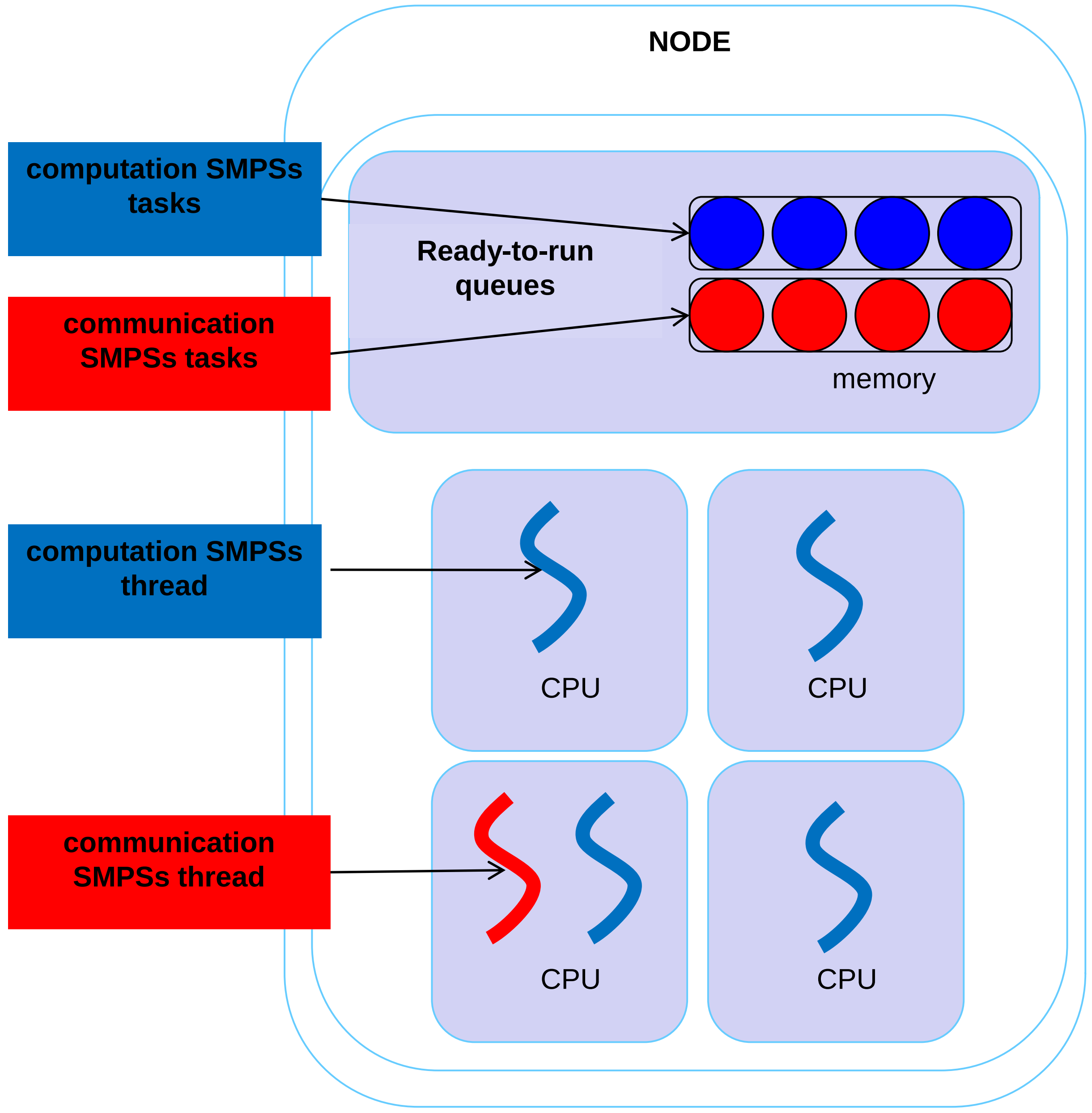
```
#pragma omp task input([m*l]a, [l*n]b) inout([m*n]c)
void mxm ( int lda, int m, int l, int n, double *a, double *b,
           double *c )
{
        double alpha=1.0, beta=1.0;
        int i, j;
        char tr = 't';
        dgemm(&tr, &tr, &m, &n, &l, &alpha, a, &lda, b, &m, &beta, c, &m);
}
```



Problems taskifying communications

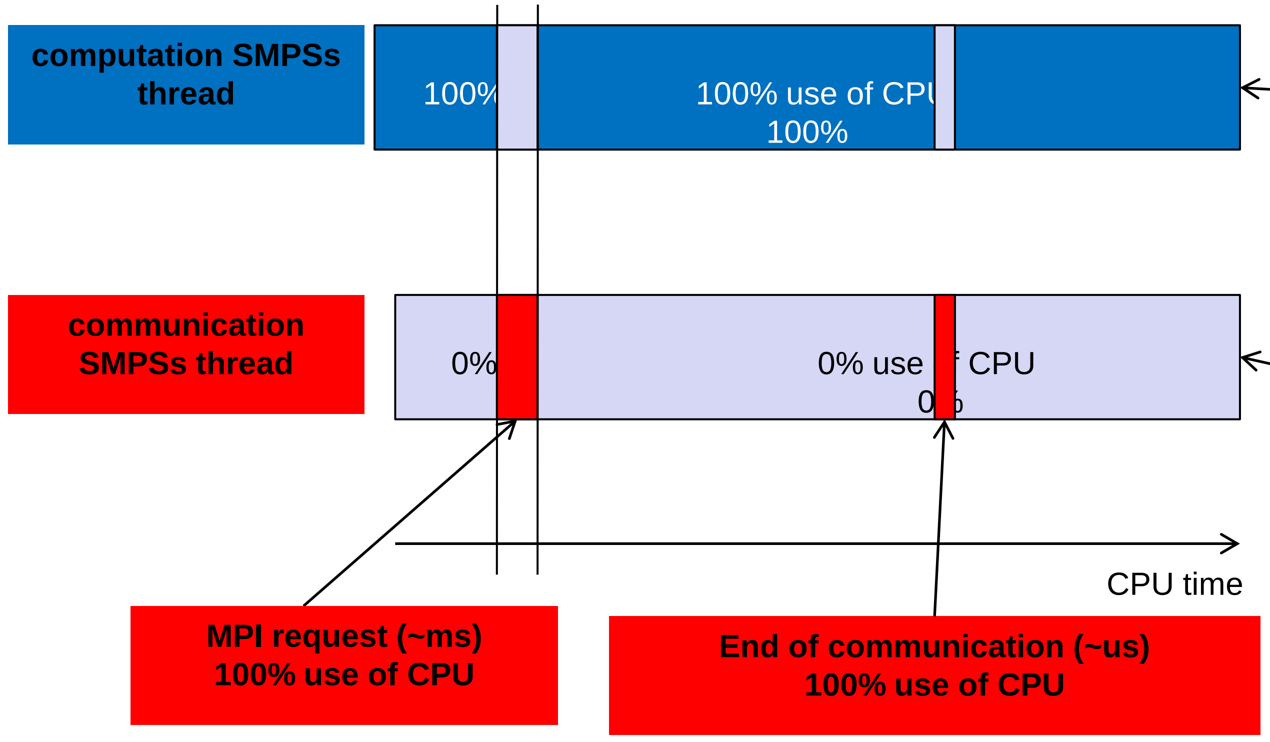
- Problems:
  - Possibly several concurrent MPI calls
    - Need to use thread safe MPI
  - Reordering of MPI calls + limited number of cores → potential source of deadlocks
    - Need to control order of communication tasks.
  - MPI task waste cores (busy waiting if communication partner delays or long transfer times)
    - An issue today, may be not with many core nodes.
- A solution: Communication Thread…

Communication thread for MPI calls



- 1 MPI process per node (N cores)
- N computation threads per node
- 1 communication thread per node
- Communication and computation threads share CPU time
- Communication threads should wake up as soon as possible and do not waste CPU time while it is blocked
- Time sharing optimization by setting OS thread priorities and MPI waiting mode

OS thread priorities & MPI waiting mode



Reducing the priority of the computation threads accelerates execution of the communication thread

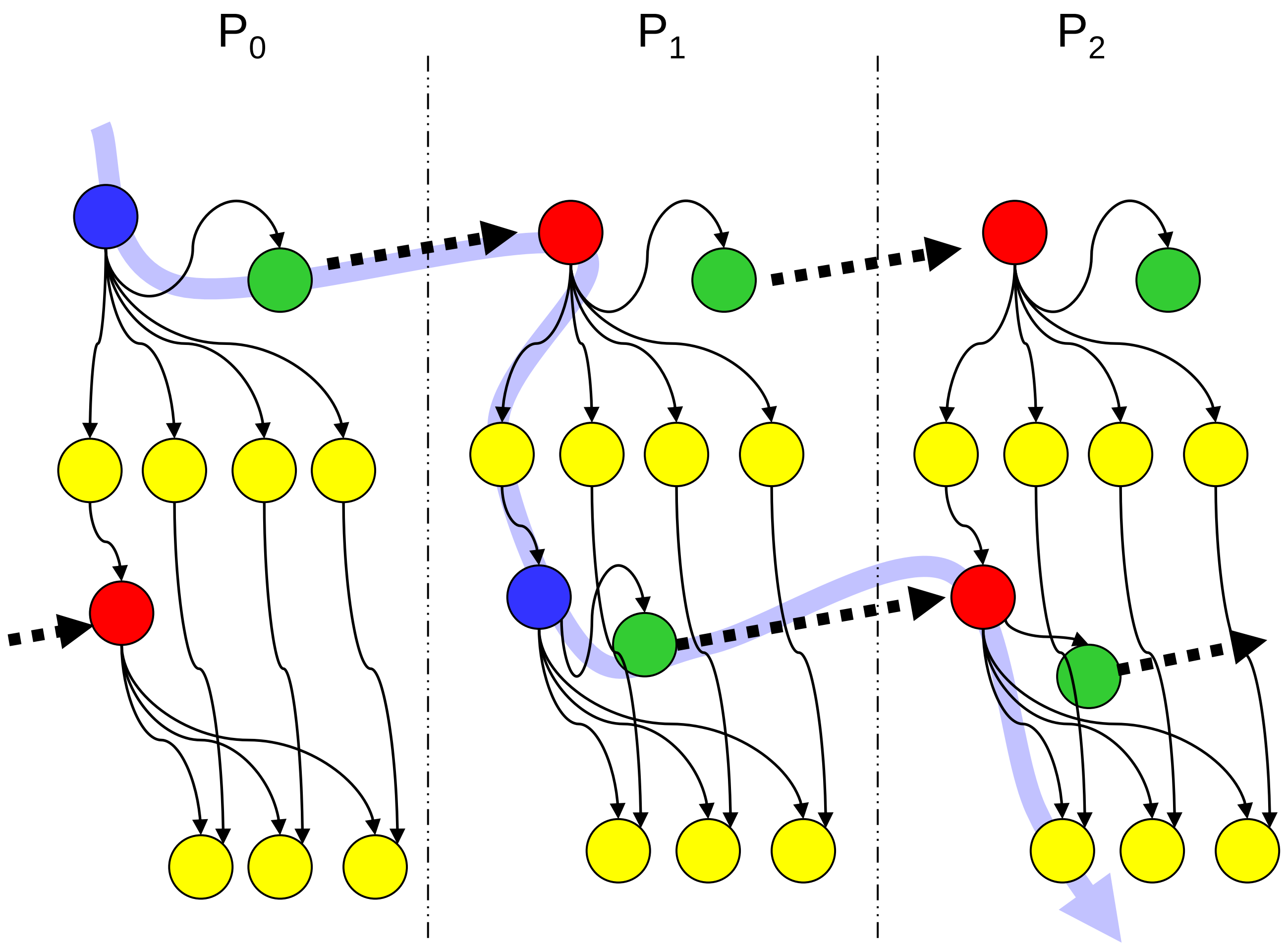
MPI calls uses blocking mode and communication thread does not waste resources while it is blocked

Hybrid MPI/SMPSs

- Overlap communication/computation
- Extend asynchronous data-flow execution to outer level
- Linpack example: 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 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);
```

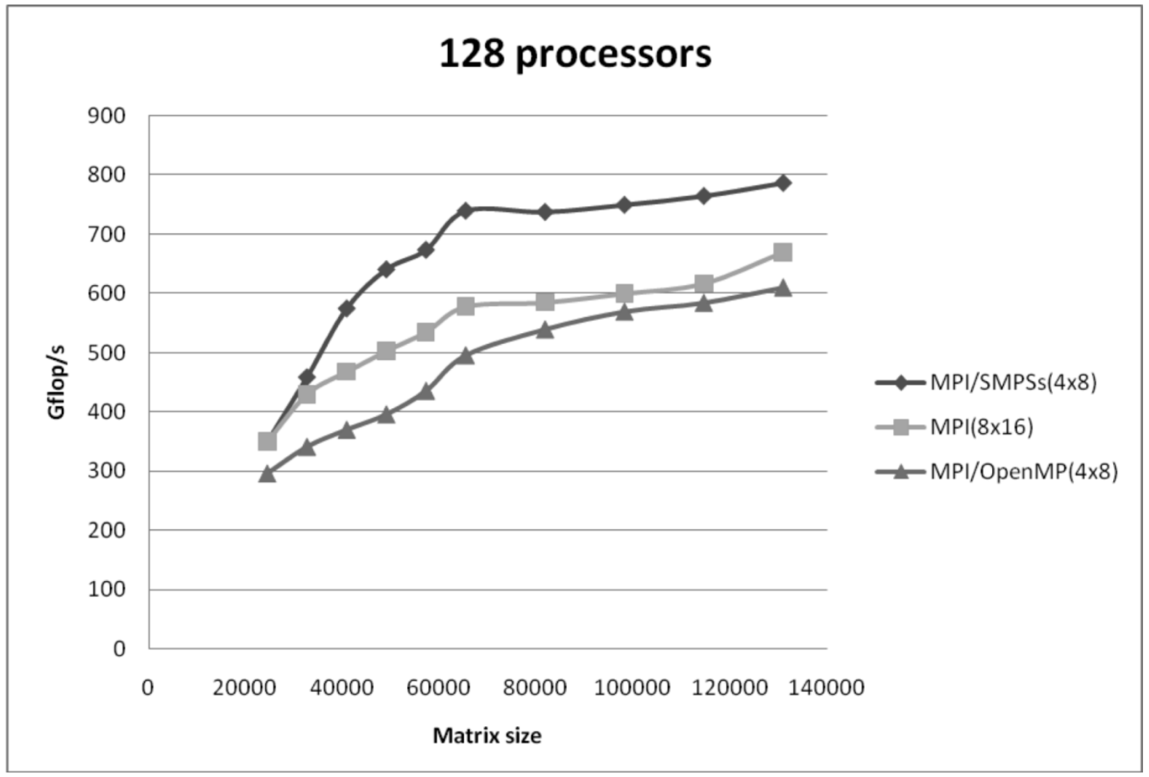


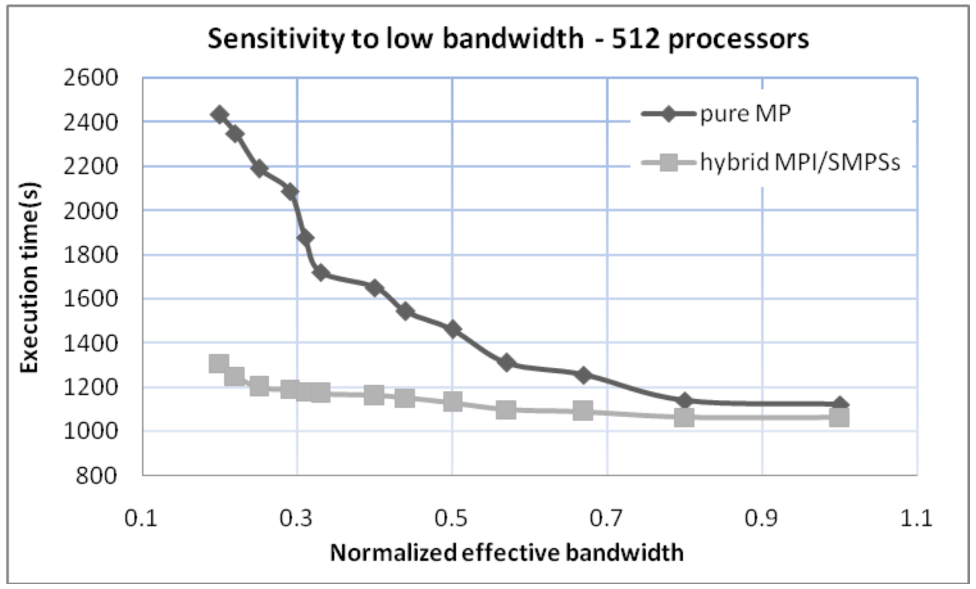
```
#pragma css task input(A[SIZE])
void send(float *A);
#pragma css task output(A[SIZE])
void receive(float *A);
#pragma css task input(A[SIZE])
void resend(float *A);
```

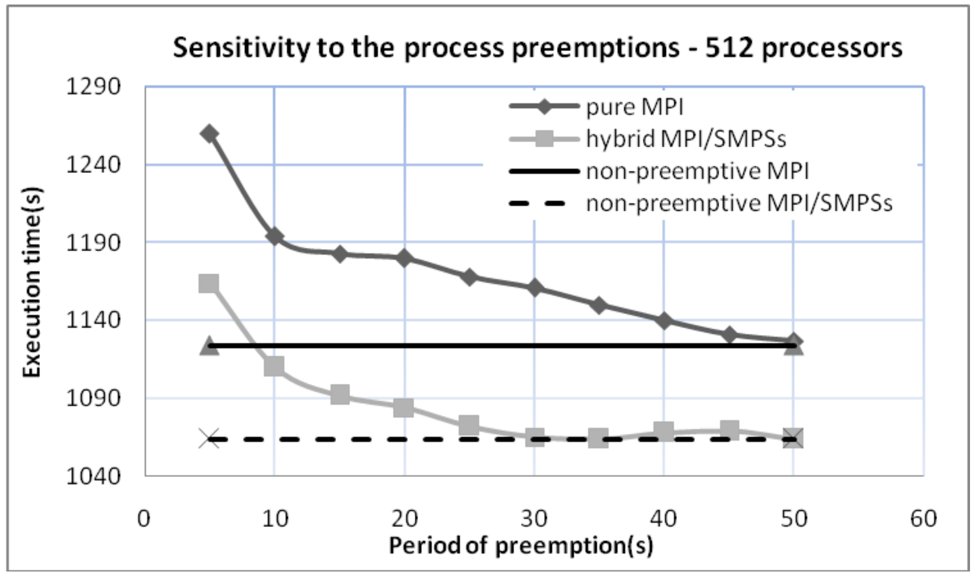
V. Marjanovic, et al, “Overlapping Communication and Computation by using a Hybrid MPI/SMPSs Approach” ICS 2010

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

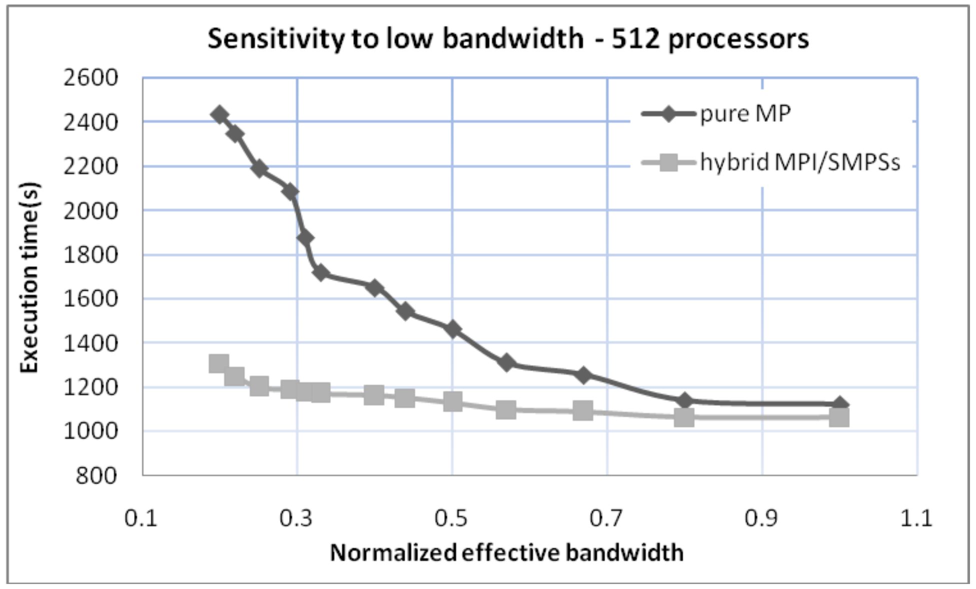


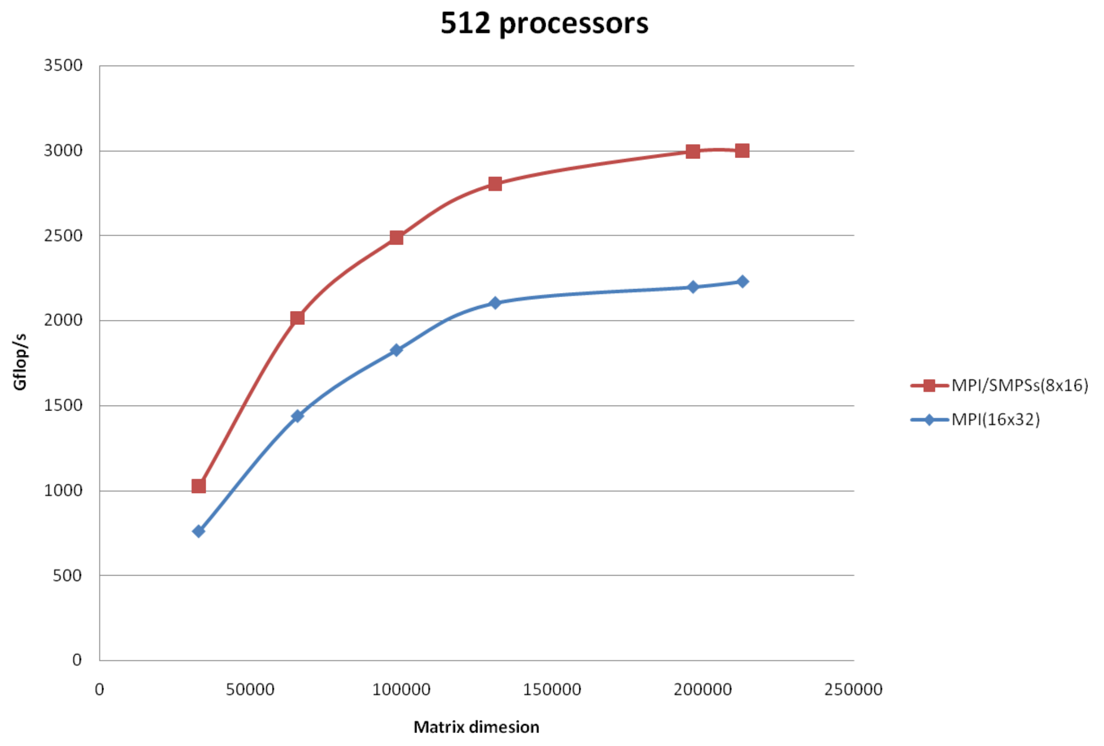


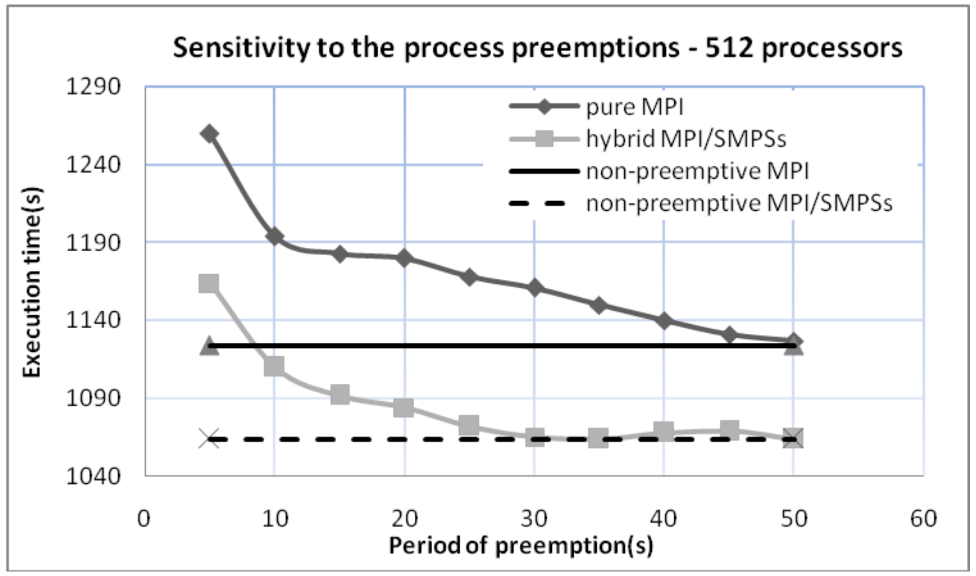


Hybrid MPI/SMPSs

- Performance
  - Higher at smaller problem sizes
  - Improved Load balance (less processes)
  - Higher IPC
  - Overlap communication/computation
- Tolerance to bandwidth and OS noise

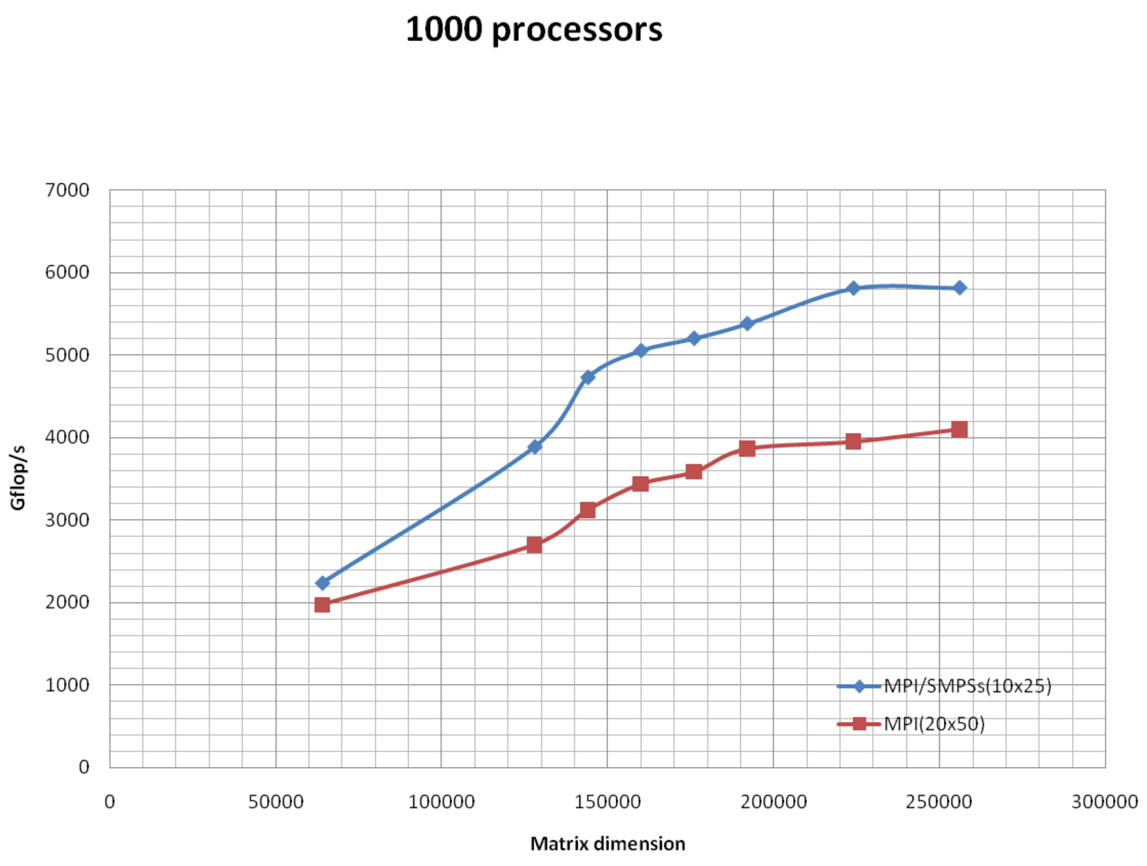


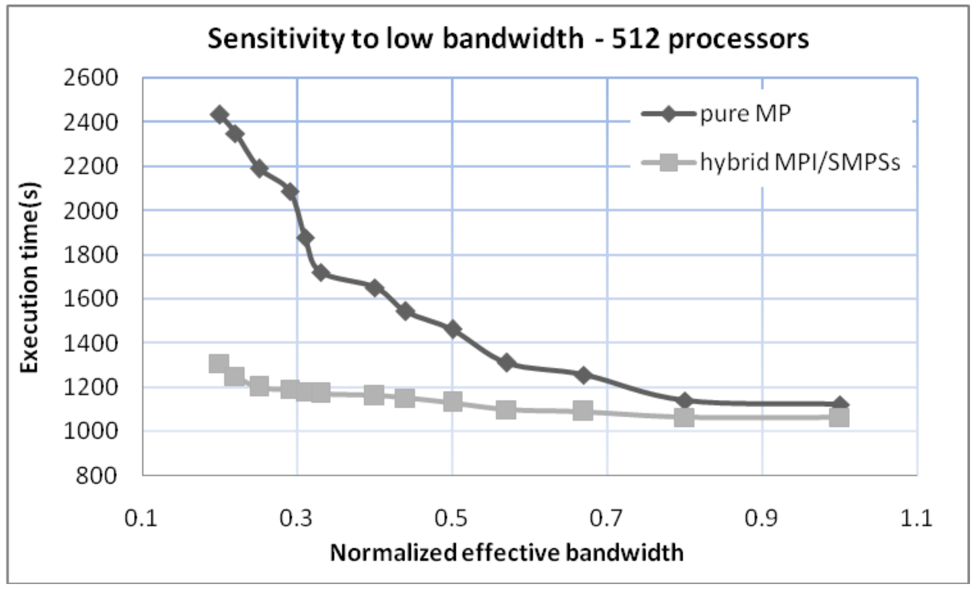


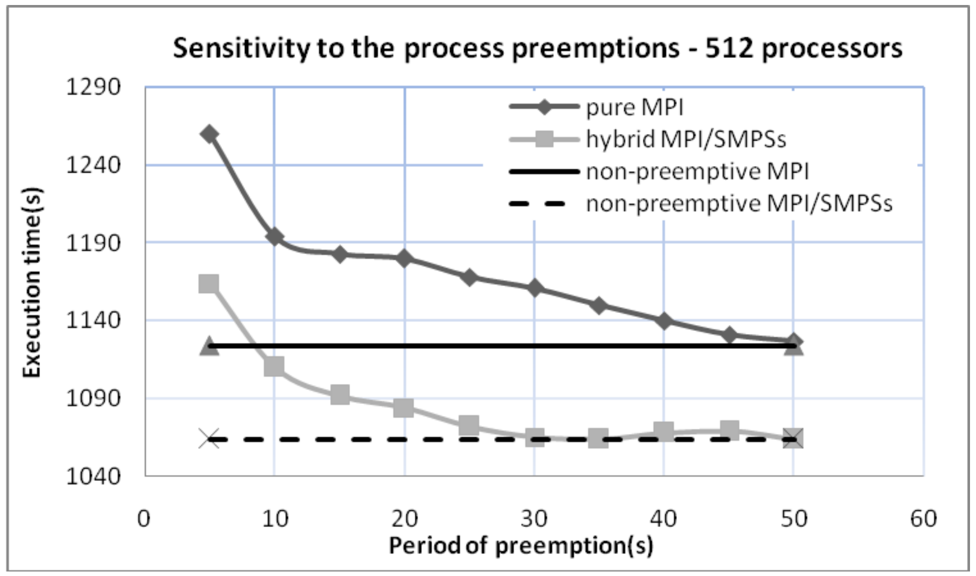


Hybrid MPI/SMPSs

- Performance
  - Higher at smaller problem sizes
  - Improved Load balance (less processes)
  - Higher IPC
  - Overlap communication/computation
- Tolerance to bandwidth and OS noise

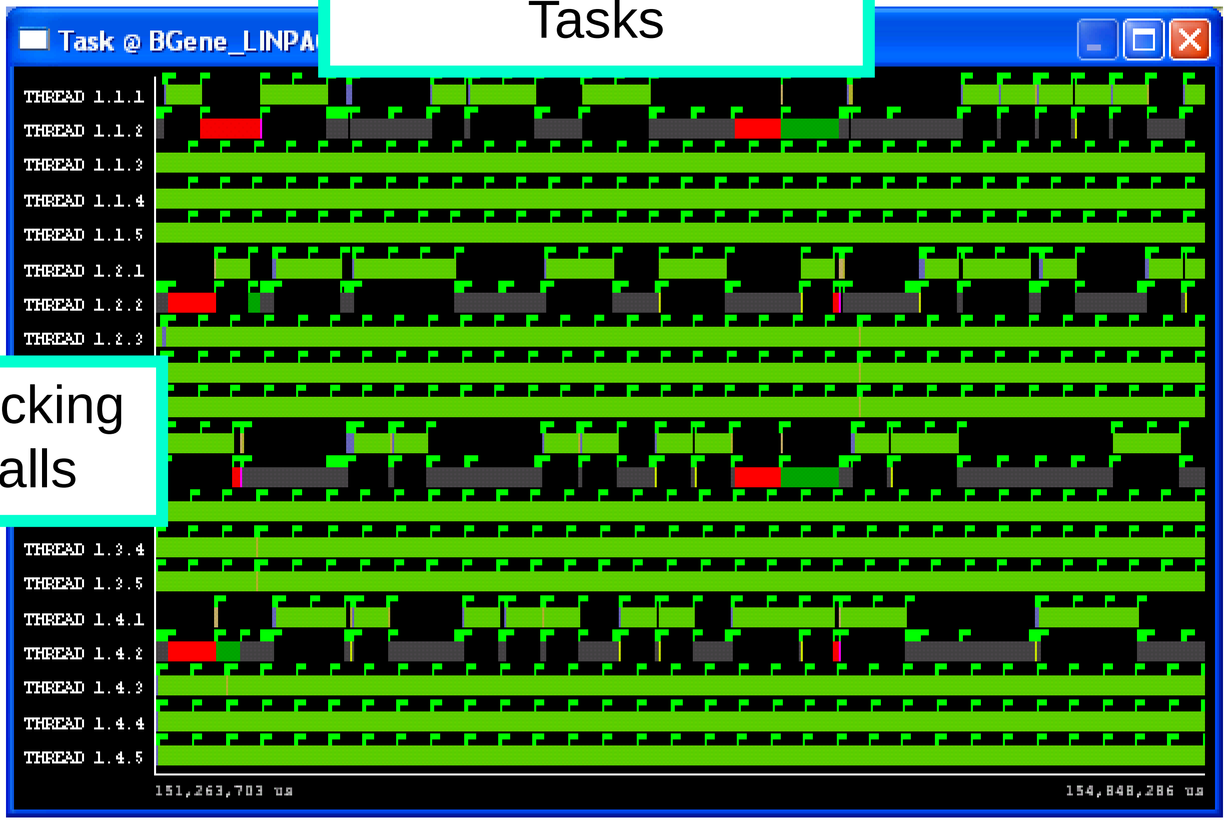




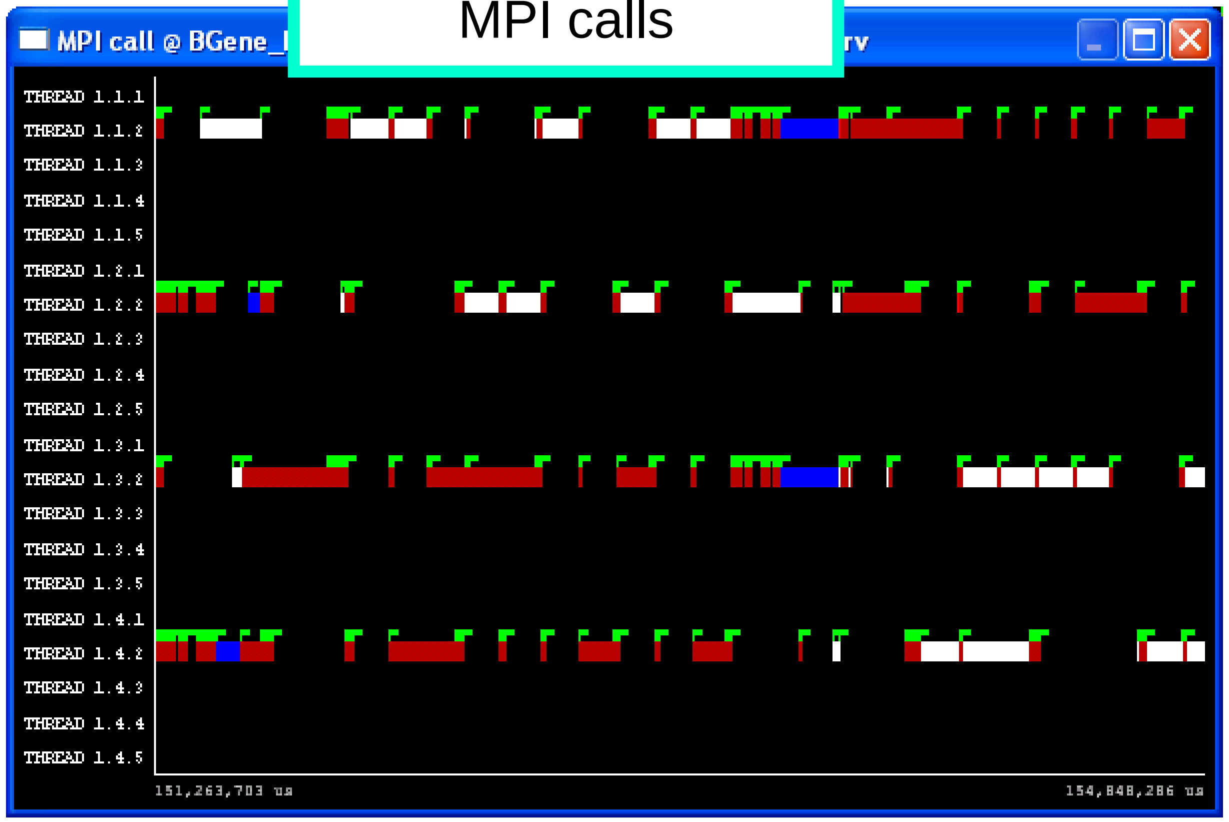


MPI/SMPSs @ BG/P

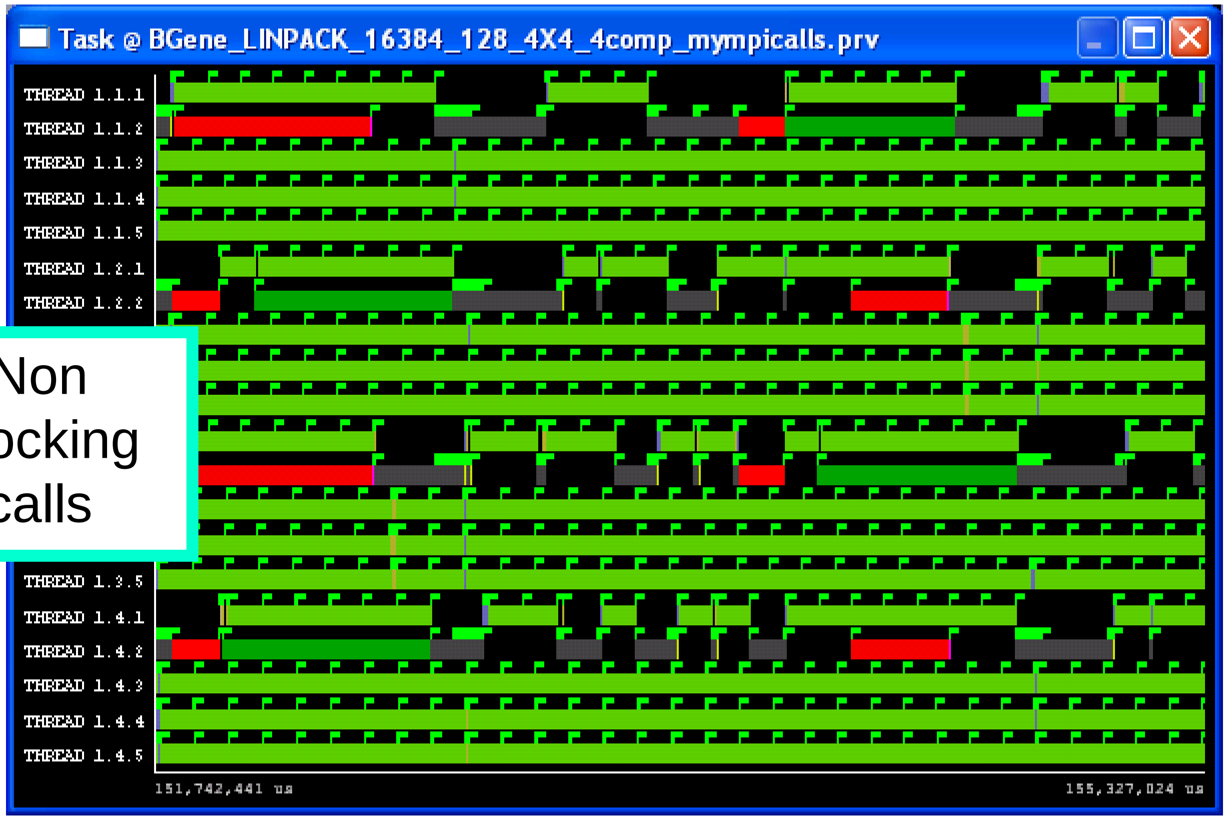
Tasks



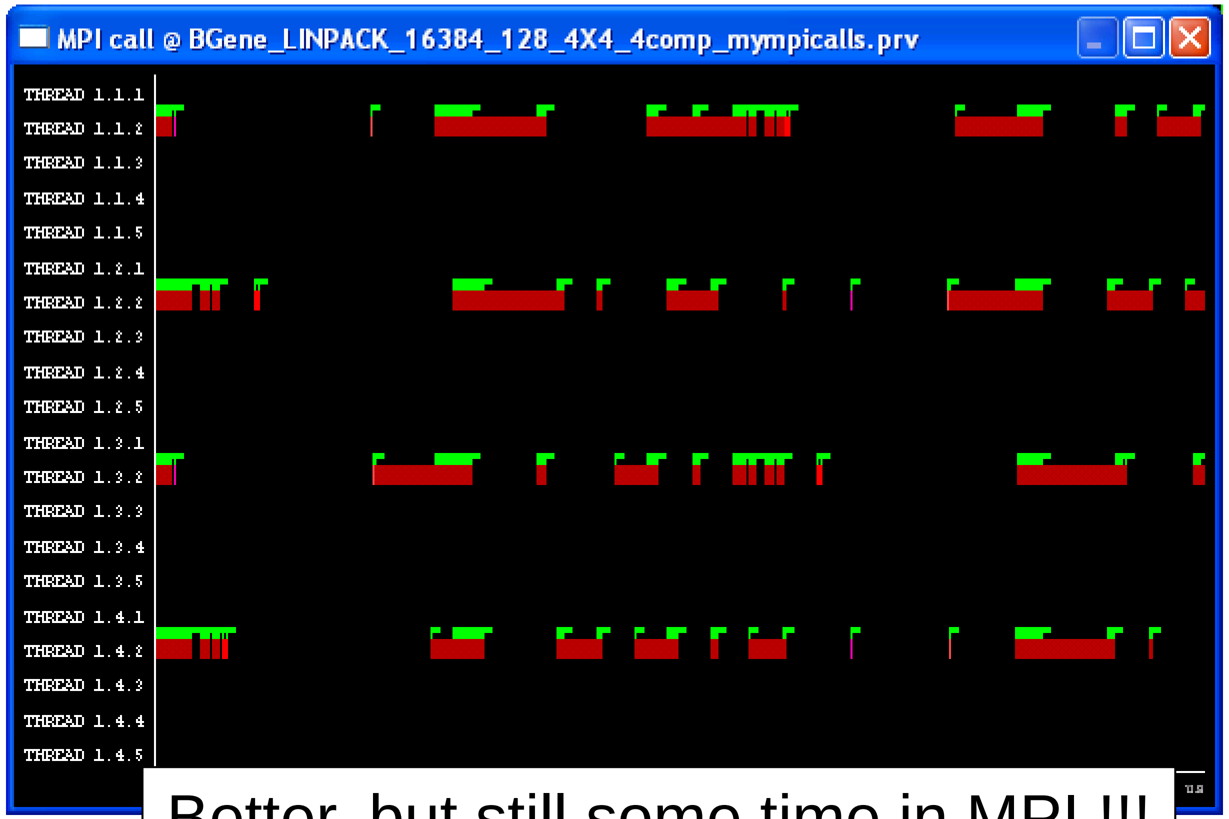
MPI calls



Blocking calls



Non Blocking calls



Better, but still some time in MPI !!!

Conclusions

- Future programming models should:
  - Enable productivity and portability
  - Support for heterogeneous/hierarchical architectures
  - Support asynchrony → global synchronization in systems with large number of nodes is not an answer anymore
  - Be aware of data locality
- OmpSs is a proposal that enables:
  - Incremental parallelization from existing sequential codes
  - Data-flow execution model that naturally supports asynchrony
  - Nicely integrates heterogeneity and hierarchy
  - Support for locality scheduling
  - Active and open source project:

git clone http://pm.bsc.es/git/mcxx.git (compiler)

git clone http://pm.bsc.es/git/nanox.git (runtime)

Pingpong

```
for(i = 0; i < NUM_OF_ITE; i++){
     for(j = 0; j < SIZE_OF_ARRAY; j+=BLOCK_SIZE)
         compute(&array[j]);

     expensive(&array[0], &sum);
     communication(partner, &array[0], &array[SIZE_OF_ARRAY-BLOCK_SIZE]);

     for(j = 0; j < SIZE_OF_ARRAY-BLOCK_SIZE; j+=BLOCK_SIZE)
       shift(&array[j+BLOCK_SIZE], &array[j]);
  }
  printf(" MAX = %i\n", css_get_max_threads());
```

```
#pragma omp task inout([BLOCK_SIZE]local_array)
void compute(double *local_array)
{
  int i,j,k;
  k=random()%20;
  for (i = 0; i < BLOCK_SIZE; i++)
       for(j = 0; j < k*20; j++)
           local_array[i]+= (local_array[i]+2)/local_
}
```

```
#pragma omp task input([BLOCK_SIZE]array_right) \
input([BLOCK_SIZE]array_left)
void shift(double *array_right, double *array_left)
{
  int i,j;

  for (i = 0; i < BLOCK_SIZE; i++)
       for(j = 0; j < 50; j++)
               array_left[i] += array_right[i]/50;
}
```

```
#pragma omp task input([BLOCK_SIZE]local_array) inout(sum)
void expensive(double local_array[BLOCK_SIZE], double *sum)
{
  int i, j;
```

```
#pragma css task input([BLOCK_SIZE]bufsend) output([BLOCK_SIZE]bufrecv)
void communication(int partner, double *bufsend, double *bufrecv)
{
  int ierr;
  MPI_Request request[2];
  MPI_Status status[2];

  ierr = MPI_Isend(bufsend, BLOCK_SIZE, MPI_DOUBLE, partner, 0, MPI_COMM_WORLD, &request[0]);
  ierr = MPI_Irecv(bufrecv, BLOCK_SIZE, MPI_DOUBLE, partner, 0, MPI_COMM_WORLD, &request[1]);
  MPI_Waitall(2, request, status);
}
```