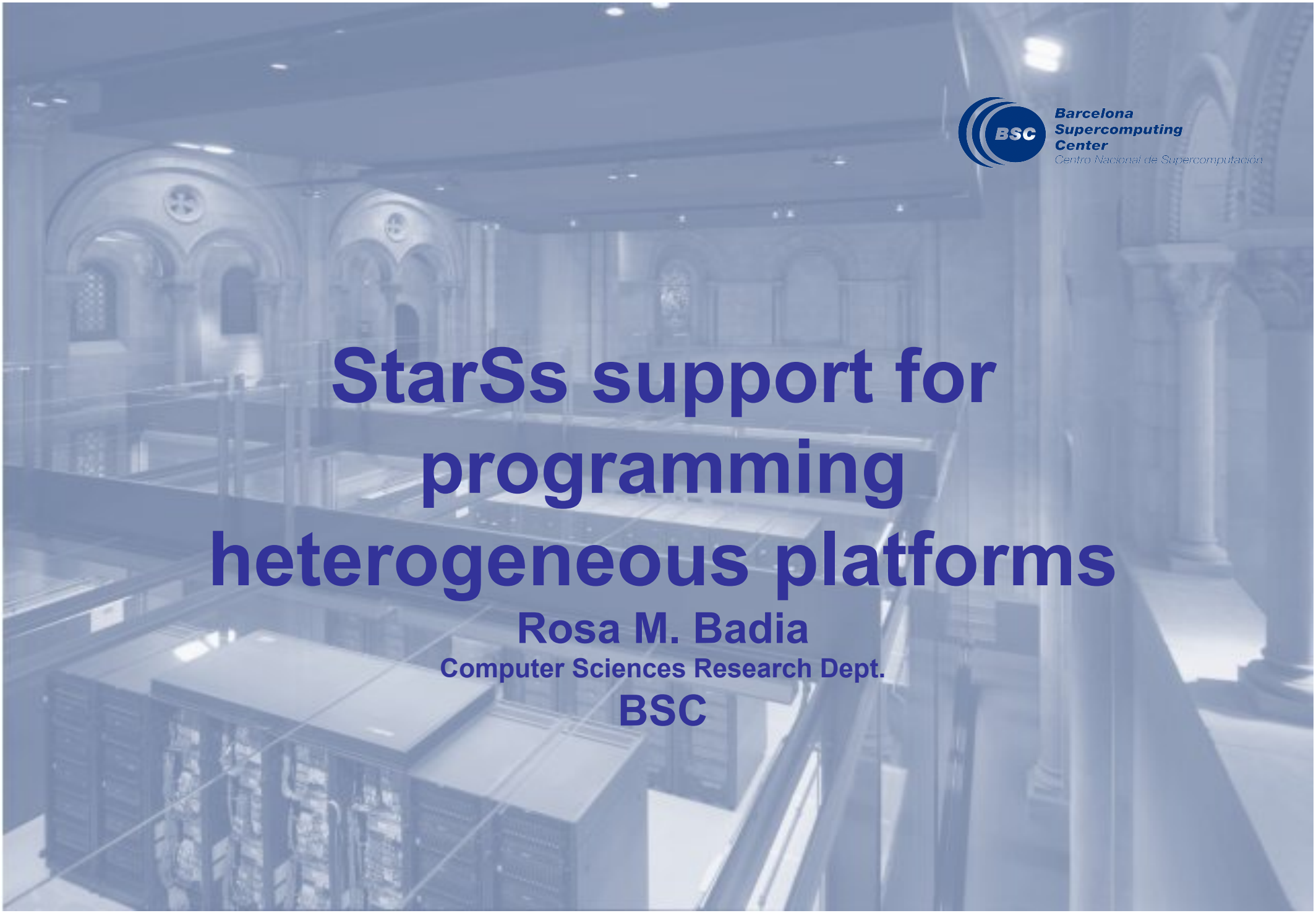
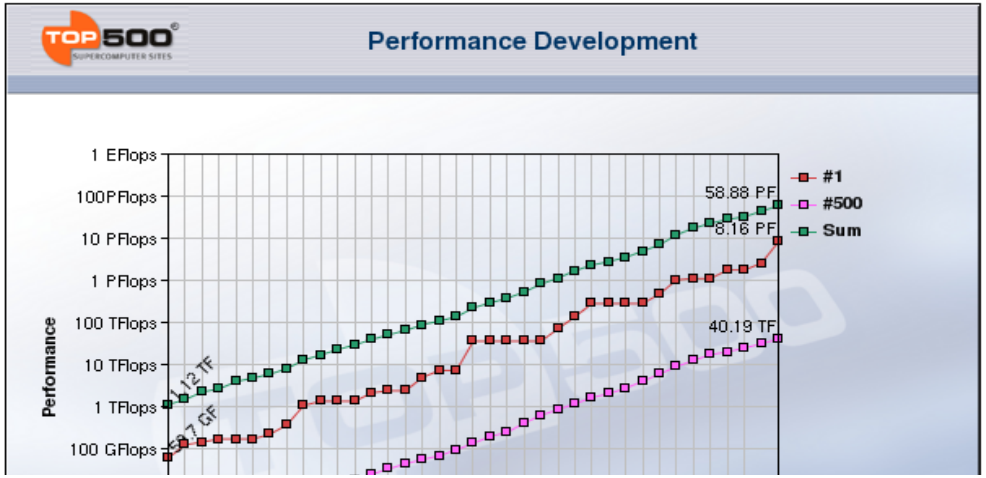
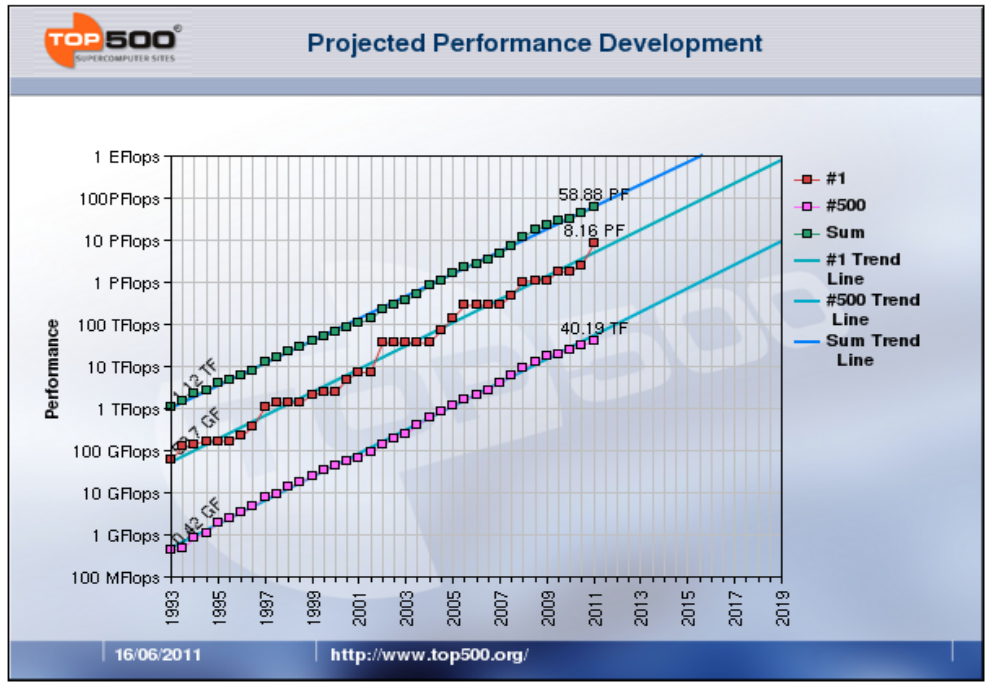


StarSs support for programming heterogeneous platforms

Rosa M. Badia
Computer Sciences Research Dept.
BSC

Evolution of computers





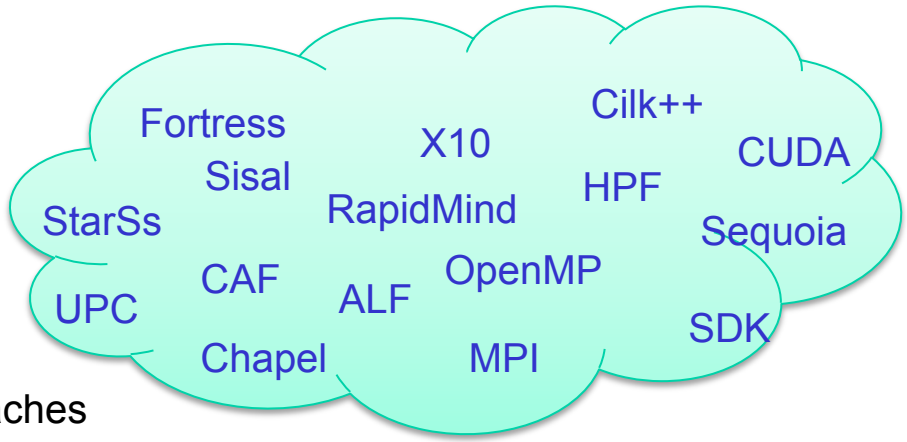
GPU, accelerator

multicore

Site	Computer
RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu
National Supercomputing Center in Tianjin China	Tianhe-1A - NUDT TH MPP, X5670 2.93Ghz 6C, NVIDIA GPU, FT-1000 8C NUDT
DOE/SC/Oak Ridge National Laboratory United States	Jaguar - Cray XT5-HE Opteron 6-core 2.6 GHz Cray Inc.
National Supercomputing Centre in Shenzhen (NSCS) China	Nebulae - Dawning TC3600 Blade, Intel X5650, NVidia Tesla C2050 GPU Dawning
GSIC Center, Tokyo Institute of Technology Japan	TSUBAME 2.0 - HP ProLiant SL390s G7 Xeon 6C X5670, Nvidia GPU, Linux/Windows NEC/HP
DOE/NNSA/LANL/SNL United States	Cielo - Cray XE6 8-core 2.4 GHz Cray Inc.
NASA/Ames Research Center/NAS United States	Pleiades - SGI Altix ICE 8200EX/8400EX, Xeon HT QC 3.0/Xeon 5570/5670 2.93 Ghz, Infiniband SGI
DOE/SC/LBNL/NERSC United States	Hopper - Cray XE6 12-core 2.1 GHz Cray Inc.
Commissariat a l'Energie Atomique (CEA) France	Tera-100 - Bull bullx super-node S6010/S6030 Bull SA
DOE/NNSA/LANL United States	Roadrunner - BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz, Voltaire Infiniband IBM

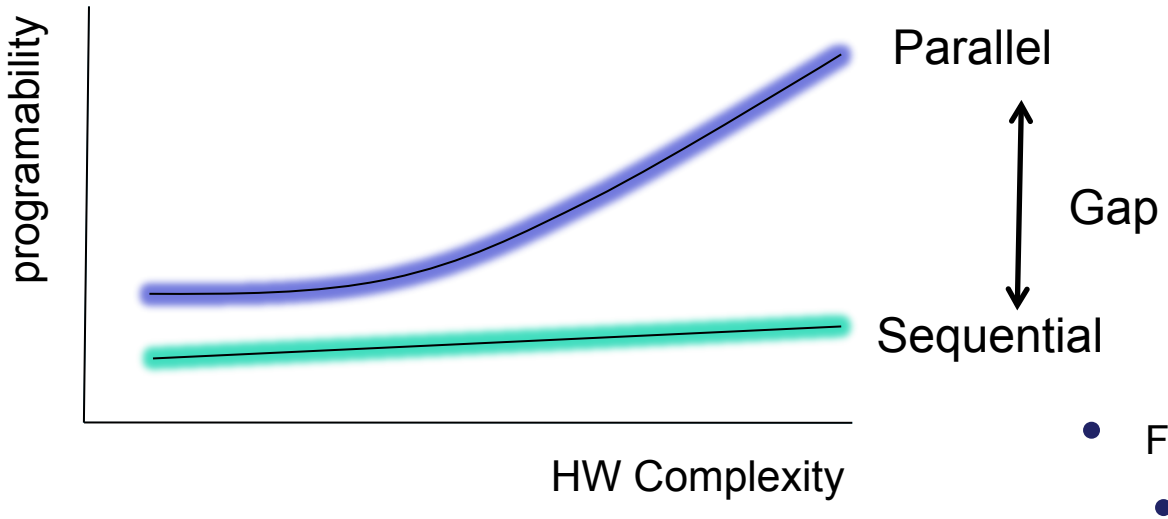
Parallel programming models

- Traditional programming models
  - Message passing (MPI)
  - OpenMP
  - Hybrid MPI/OpenMP
- Heterogeneity
  - CUDA
  - OpenCL
  - ALF
  - RapidMind
- Alternative approaches
  - Partitioned Global Address Space (PGAS) programming models
    - UPC, X10, Chapel
- ...



Simple programming paradigms that enable easy application development are required

Programability wall



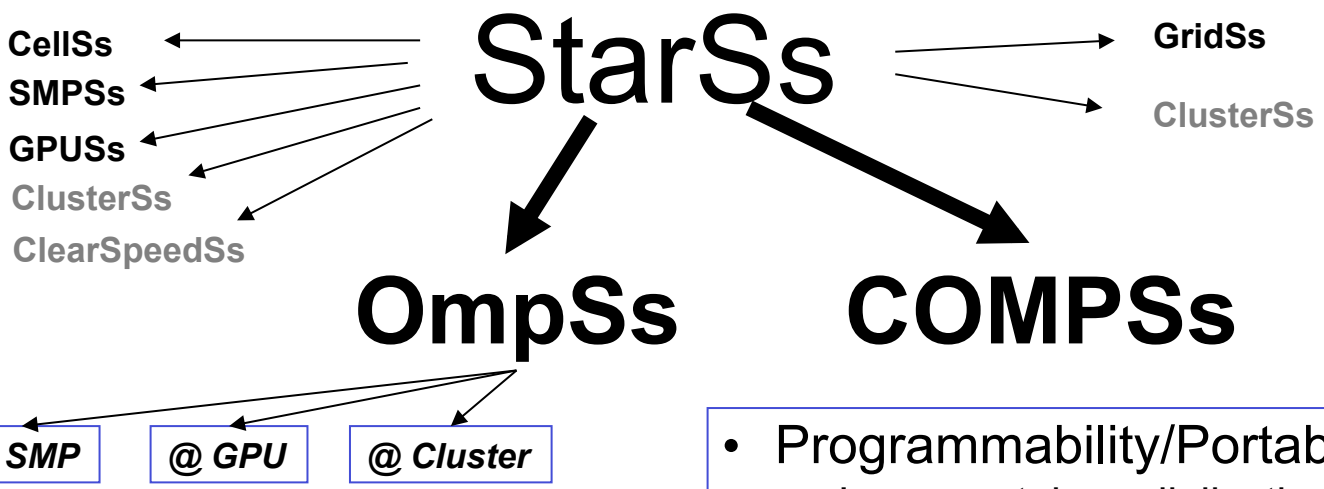
- Features that can help reducing the gap:
  - Sequential code closer to parallel code
  - Support for heterogeneity
  - Same source for different platforms
  - Code independent of underlying HW
  - Tools that help on
    - Application parallelization
    - Application debug
    - Application performance analysis

Agenda

- StarSs overview
- StarSs + OpenMP = OmpSs
  - Syntax
  - Infrastructure: Compiler & Runtime
  - Developing StarSs applications
  - Examples
- Conclusions

The StarSs programming model

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



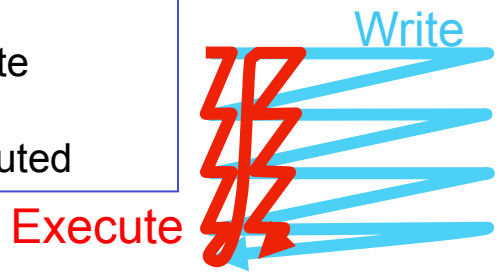
- StarSs
  - A "node" level programming model
  - Sequential C/Fortran/Java + annotations
  - Task based. Asynchrony, data-flow.
  - "Simple" linear address space
  - Directionality annotations on tasks arguments
  - Malleable
  - Nicely integrates in hybrid MPI/StarSs
  - Natural support for heterogeneity

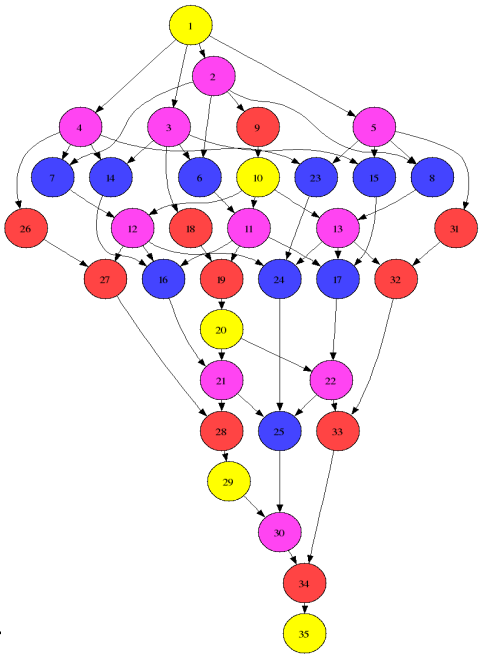
- Programmability/Portability
  - Incremental parallelization/restructure
  - Separate algorithm from resources
  - Disciplined programming
  - **"Same" source code runs on "any" machine**
    - Optimized task implementations will result in better performance.
- Performance
  - Intelligent Runtime
    - Automatically extracts and exploits parallelism
      - Dataflow, workflow
    - Matches computations to specific resources on each type of target platform
  - Asynchronous (data-flow) execution and locality awareness

Data-flow/Asynchrony in StarSs

- Graph dynamically generated at run time from execution of sequential program

Decouple
how we write
form
how it is executed



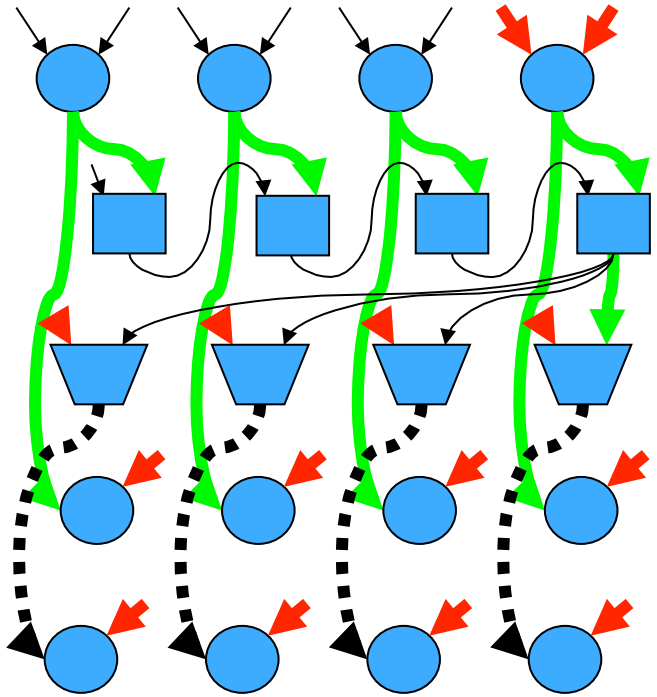


```
void Cholesky( float *A ) {
   int i, j, k;
   for (k=0; k<NT; k++) {
      spotrf (A[k*NT+k]) ;
      for (i=k+1; i<NT; i++)
         strsm (A[k*NT+k], A[k*NT+i]);
      // update trailing submatrix
      for (i=k+1; i<NT; i++) {
         for (j=k+1; j<i; j++)
            sgemm( A[k*NT+i], A[k*NT+j],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

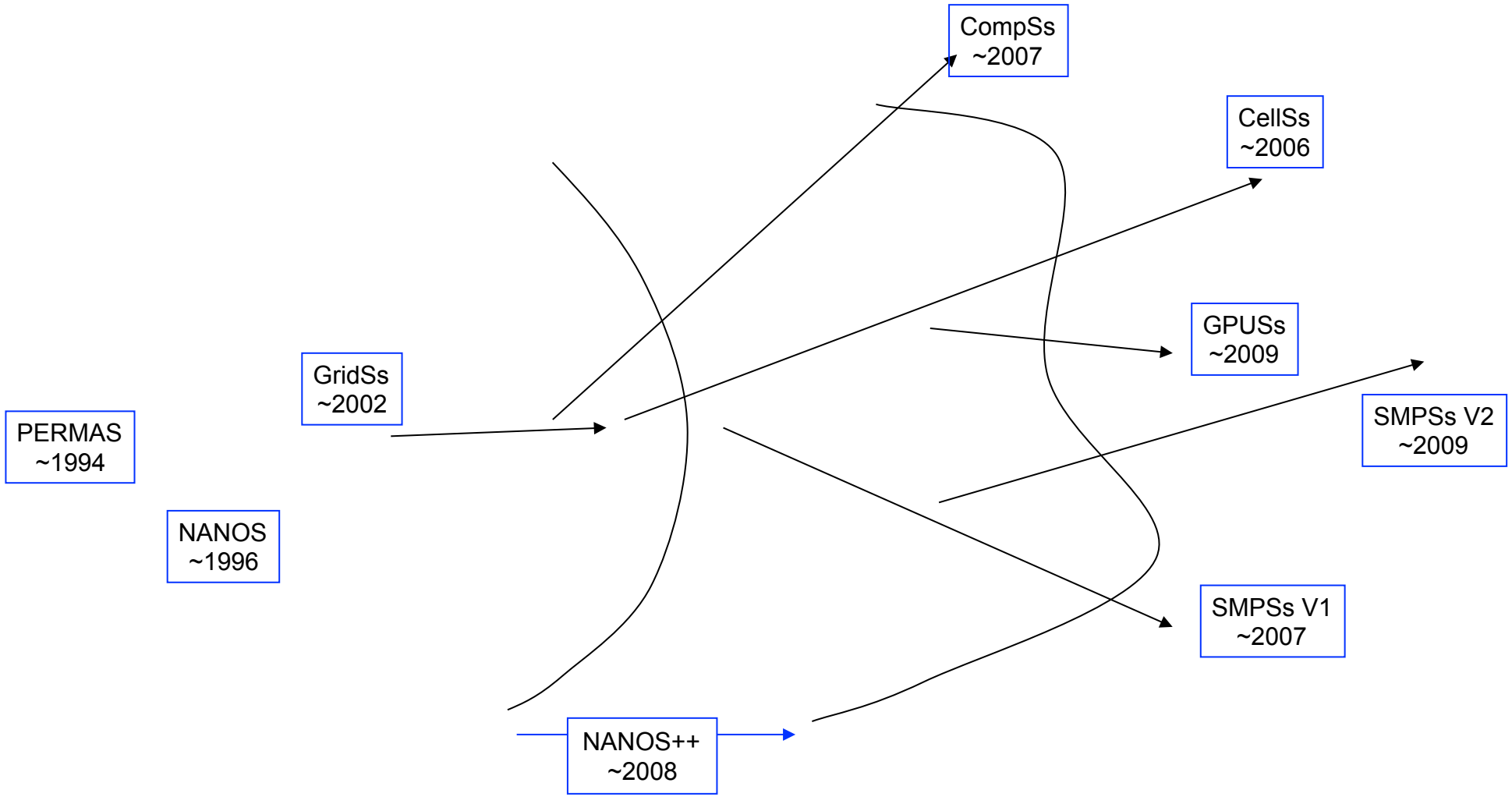
```
#pragma omp task inout ([TS][TS]A)
void spotrf (float *A);
#pragma omp task input ([TS][TS]A) inout ([TS][TS]C)
void ssyrk (float *A, float *C);
#pragma omp task input ([TS][TS]A, [TS][TS]B) inout ([TS][TS]C)
void sgemm (float *A, float *B, float *C);
#pragma omp task input ([TS][TS]T) inout ([TS][TS]B)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

StarSs: the potential of data access information

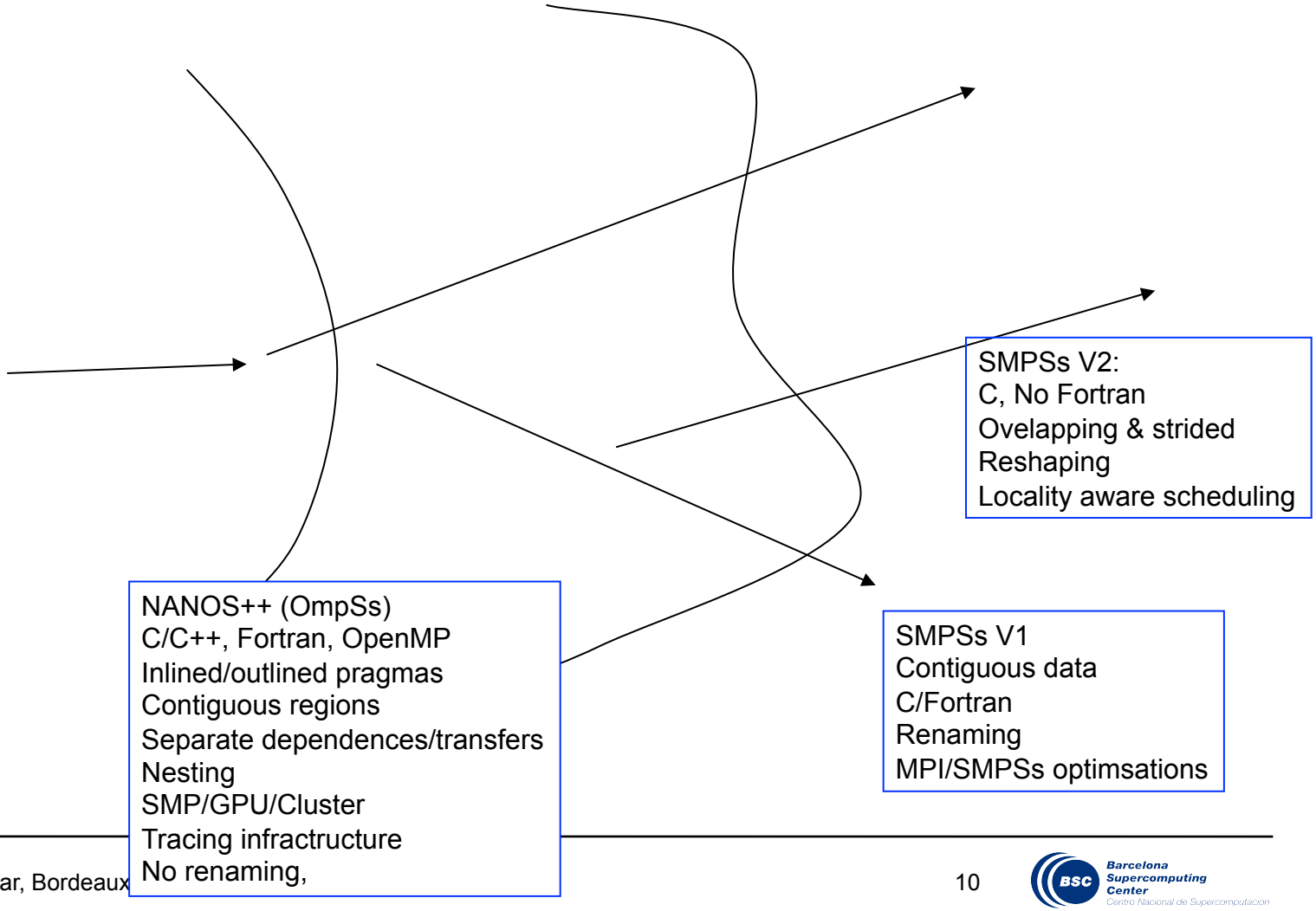
- **Flat global address space seen by programmer**
- Flexibility to dynamically traverse dataflow graph "optimizing"
  - Concurrency. Critical path
  - Memory access: data transfers performed by run time
- **Opportunities for**
  - Prefetch
  - Reuse
  - Eliminate antidependences (rename)
  - Replication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



History / Strategy



More relevant versions



OmpSs

- OpenMP compatibility and extension
- Integrating StarSs concepts
- A few directives

```
# pragma omp target device ({ smp | cell | cuda })     \
                  [ implements ( function_name )]          \
                  { copy_deps | [ copy_in ( array_spec ,...)] [ copy_out (...)] [ copy_inout (...)] }


# pragma omp task [ input (...)] [ output (...)] [ inout (...)] \
                           [concurrent(...)]
 { function or code block }

# pragma omp taskwait
# pragma omp taskwait on (...)
# pragma omp taskwait noflush
```

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 (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:

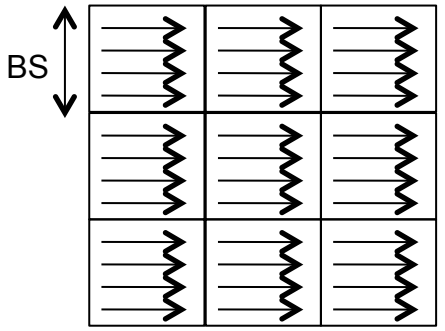
  #pragma omp target [clauses]

  - 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

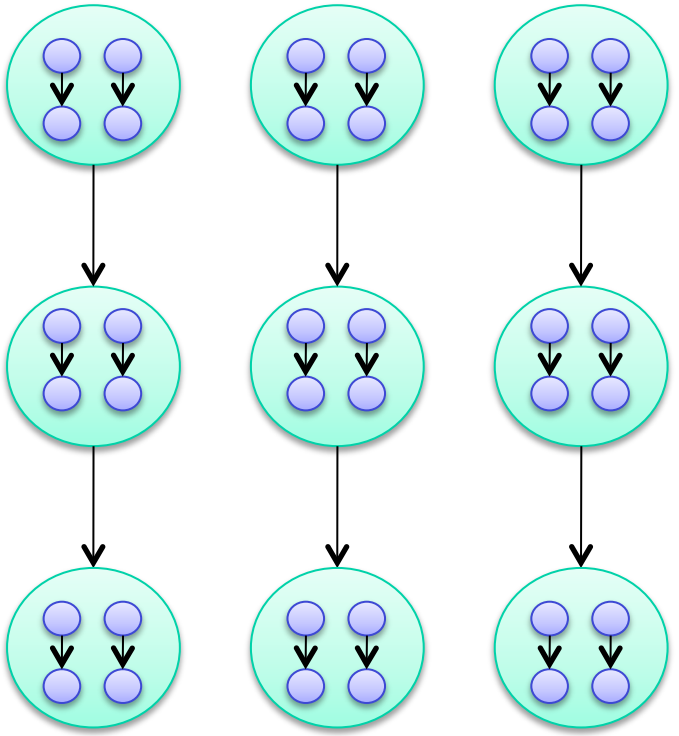
```
#pragma omp target device (cuda) copy_deps implements (scale_task)
#pragma omp task input ([size] c) output ([size] b)
void scale_task_cuda(double *b, double *c, double scalar, int size)
{

    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);
}
```

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() {
(
  ...
block_dgemm(A,B,C);
block_dgemm(D,E,F);
#pragma omp taskwait

}
```



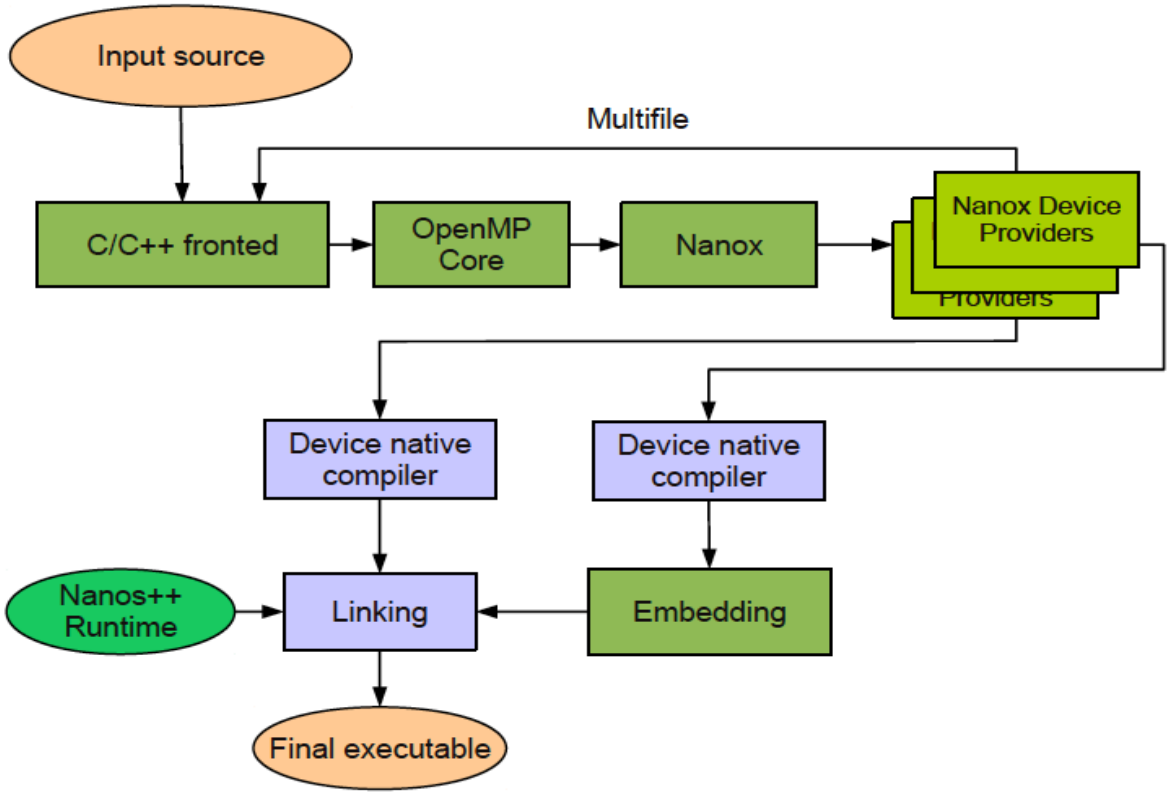
Avoiding data transfers

- 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
}
```

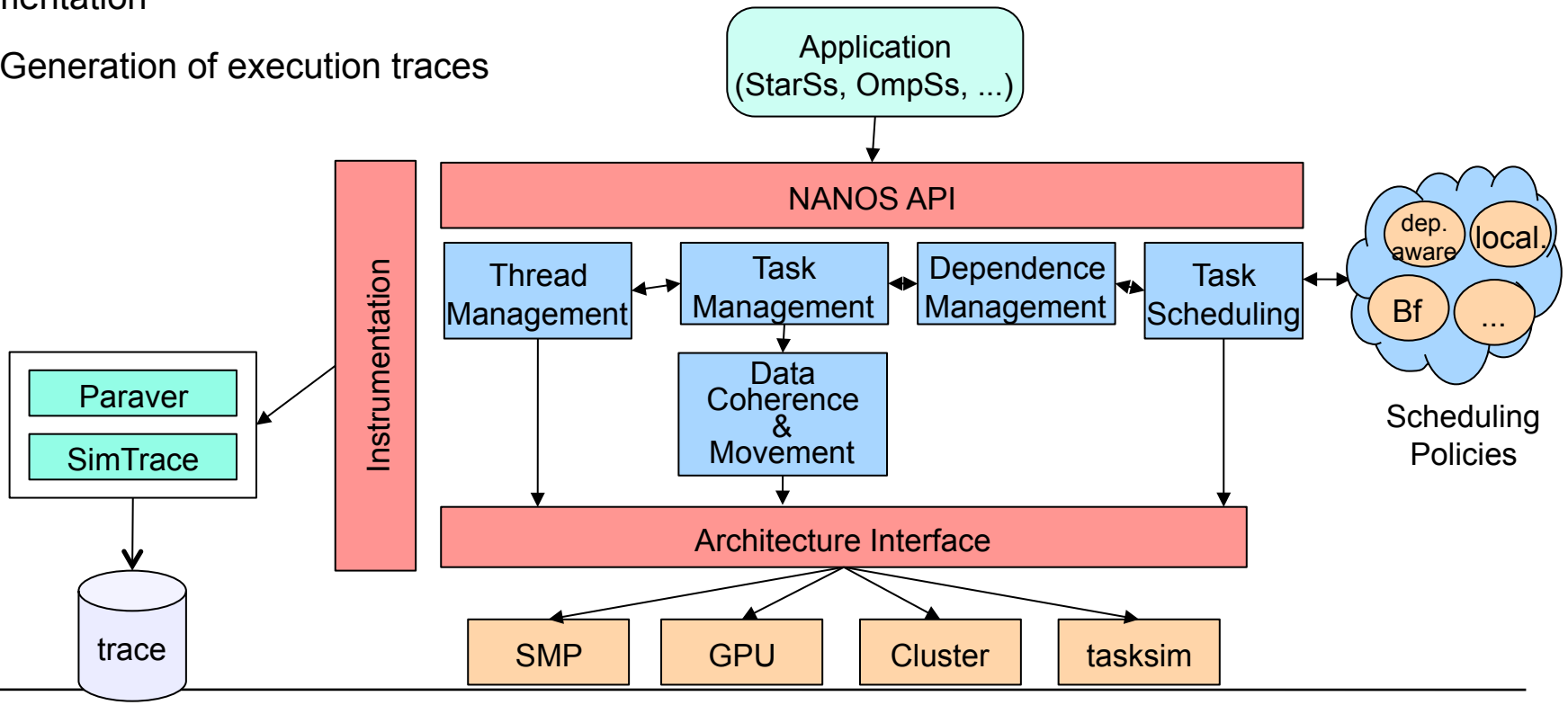
Mercurium Compiler

- 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



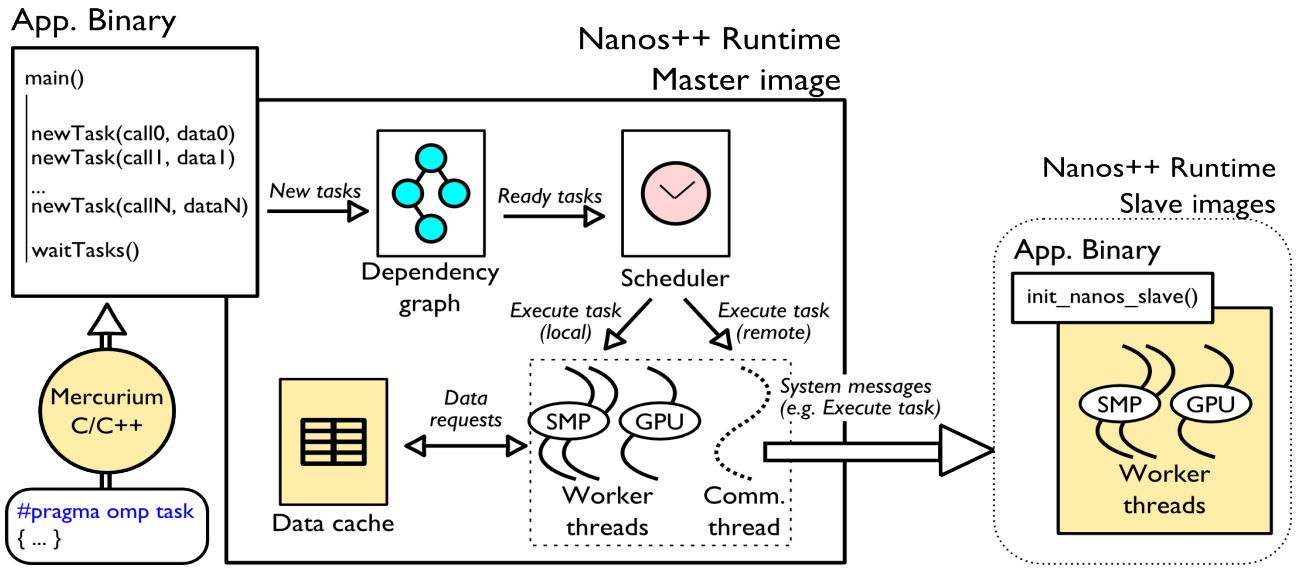
Runtime structure

- 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



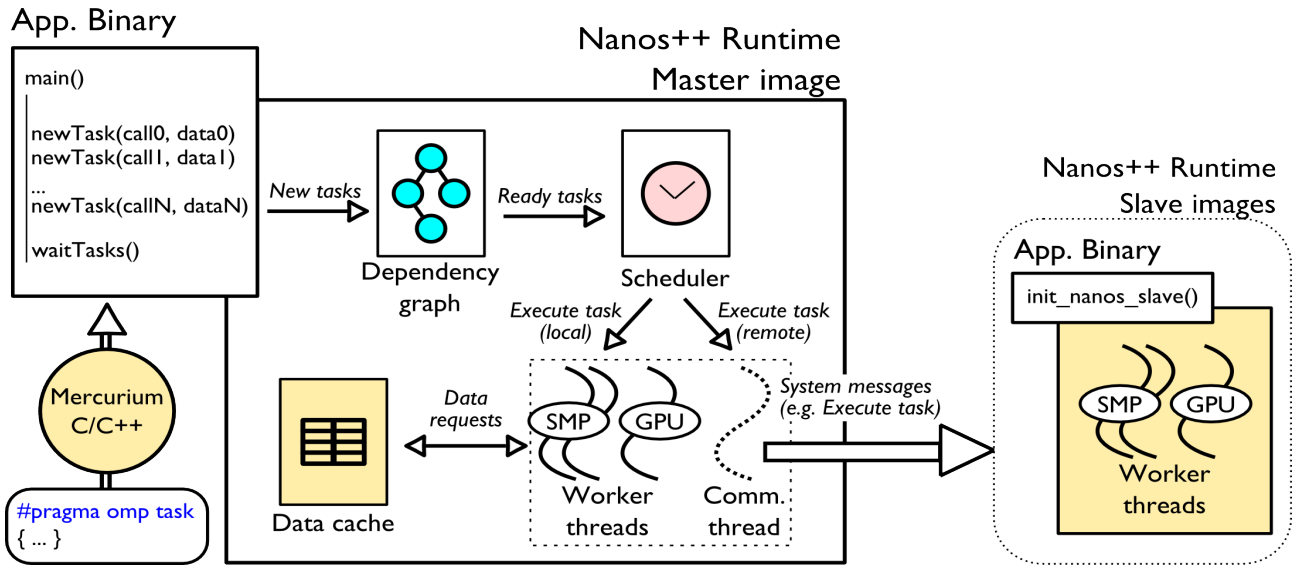
Runtime structure behaviour: task handling

- Task generation
- Data dependence analysis
- Task scheduling



Runtime structure behaviour: coherence support

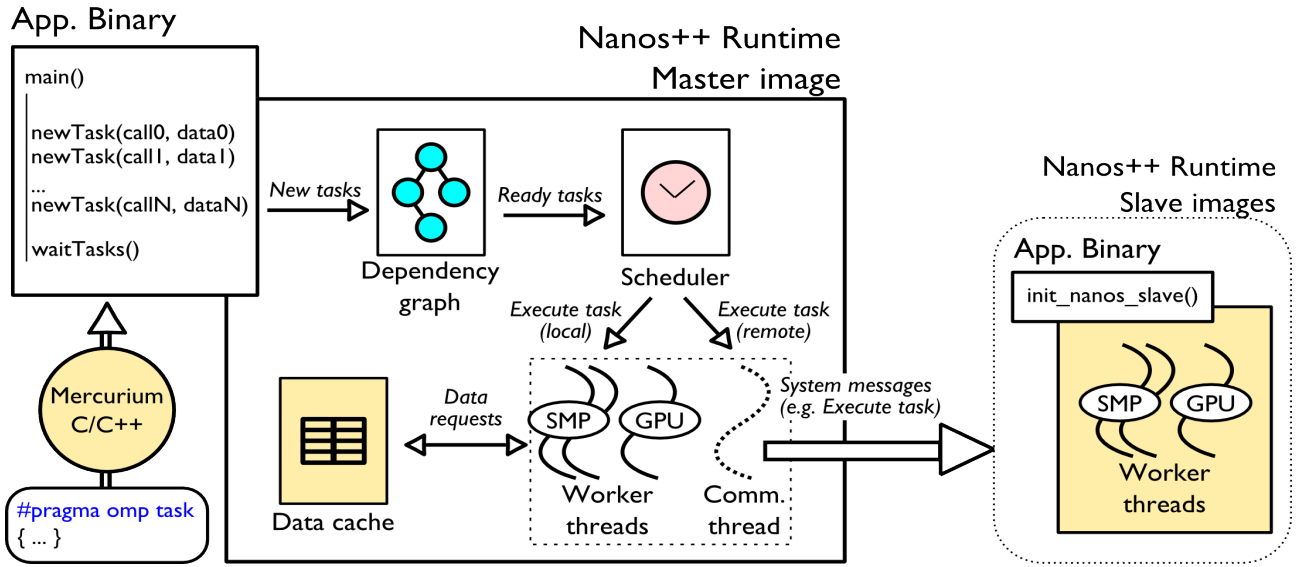
- Different address spaces managed with:
  - A hierarchical directory
  - A software cache per each:
    - Cluster node
    - GPU
- Data transfers between different memory spaces only when needed
  - Write-through
  - Write-back



Runtime structure behaviour: clusters

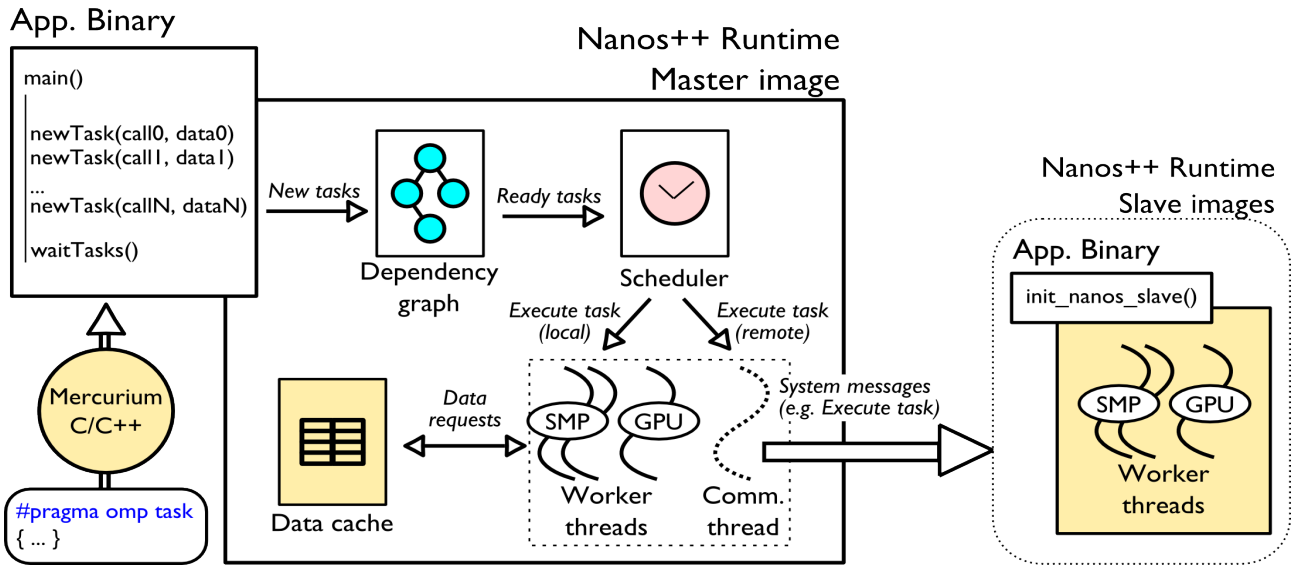
More on Friday, session G3
"Productive Cluster Programming with OmpSs",
Javier Bueno et al.

- 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



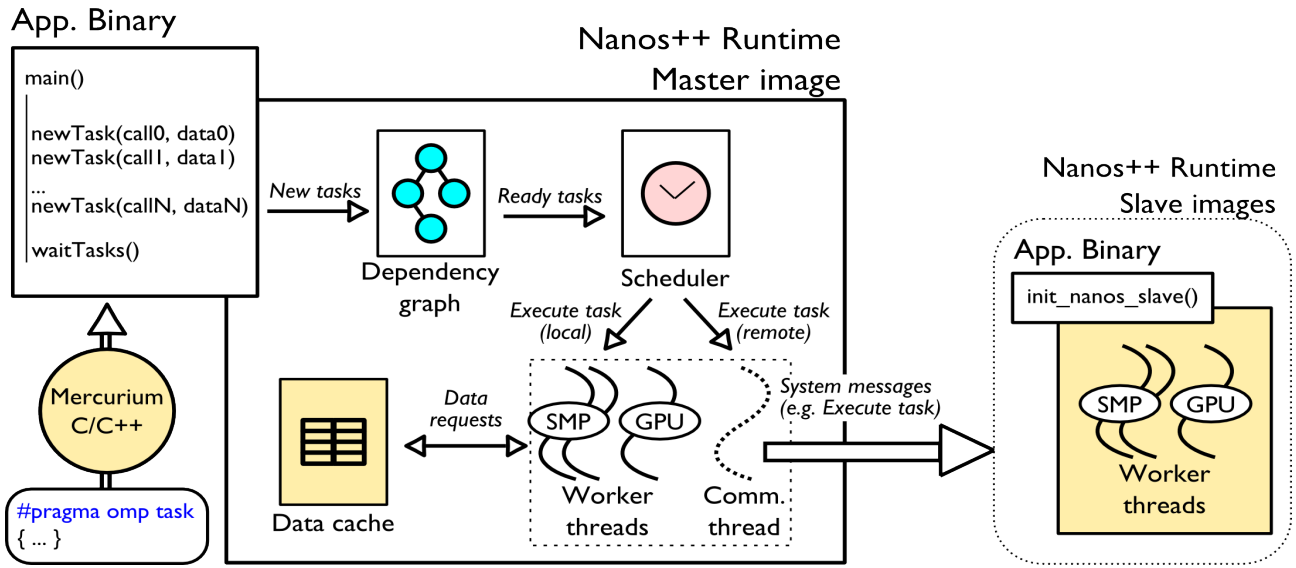
Runtime structure behaviour: GPUs

- 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 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



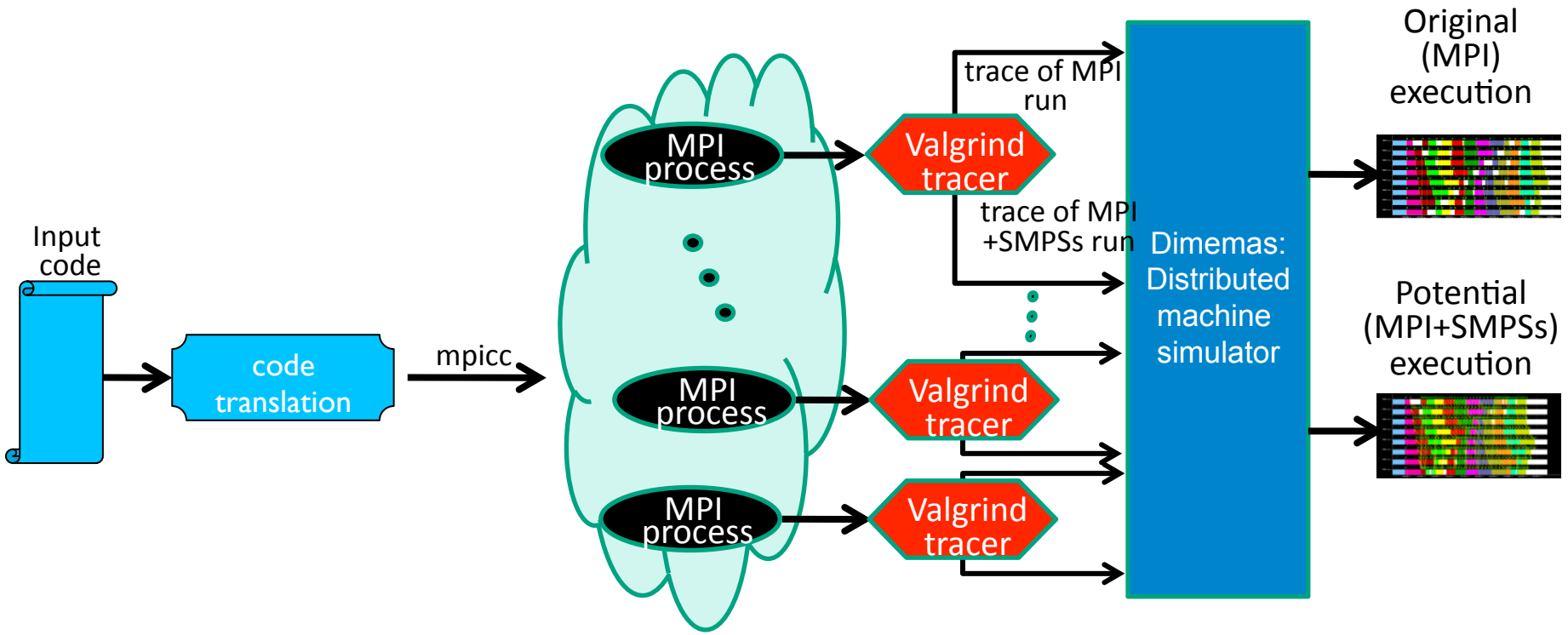
Development tools

- Development tools for StarSs applications
  - Valgrind based tool for task identification and correctness check
  - Ayudame/Temanejo StarSs debugging

Task identification and correctness validation: Valgrind-based tool

- Help porting applications to StarSs
  - From an MPI code:
    - How to partition the execution into tasks
  - For the MPI code partitioned into tasks:
    - What is the directionality of parameters passed to each tasks
- Check correctness of MPI/SMPSs applications
  - For an MPI/SMPSs code:
    - Assure the correctness of the specified directionality of parameters
- Simulate MPI/SMPSs execution
  - For various mappings of MPI processes onto computation nodes
  - For various configurations of each computation node
  - For various configurations of the interconnect

The Environment



- Input: MPI code with annotated taskification (incomplete MPI/SMPSs code)
- Executed as MPI app ➔ all SMPSs tasks are executed in the order of their instantiation
- All memory accesses of each task are automatically instrumented
  - Data dependencies among tasks are derived
- Tracefile of the potential MPI/SMPSs execution is generated
- Dimemas/Venus simulator reconstructs the potential MPI/SMPSs behavior

Input

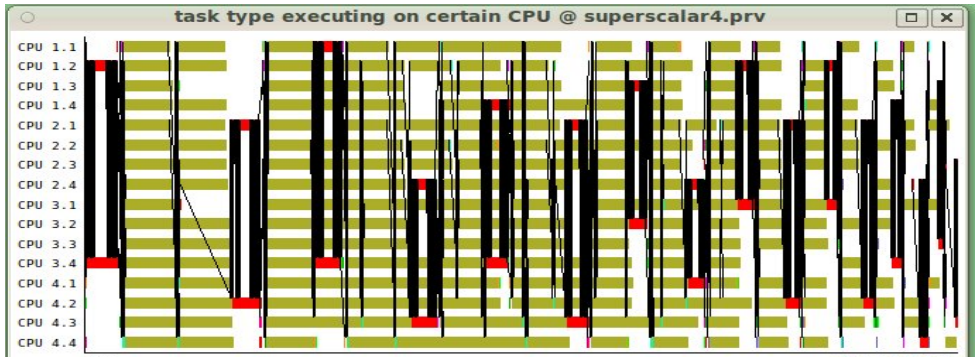
More on Wednesday, session C1
"Quantifying the potential task-based dataflow parallelism in MPI applications",
Vladimir Subotic et al.

- MPI code with proposed taskification (incomplete MPI/SMPSs code)

```
#pragma css task
void copy( float A[NUM_ELEM], float B[NUM_ELEM) {
        for (i = 0; i < NUM_ELEM; i++)
                B[i] = A[i];
        }
```

Pragma is incomplete:
It specifies that function *copy* should be considered as task,
but not directionality of parameters

- Simulation of the proposed taskification (as if the MPI/SMPSs code was complete)



- Directionality of parameters (for completing pragma directives)

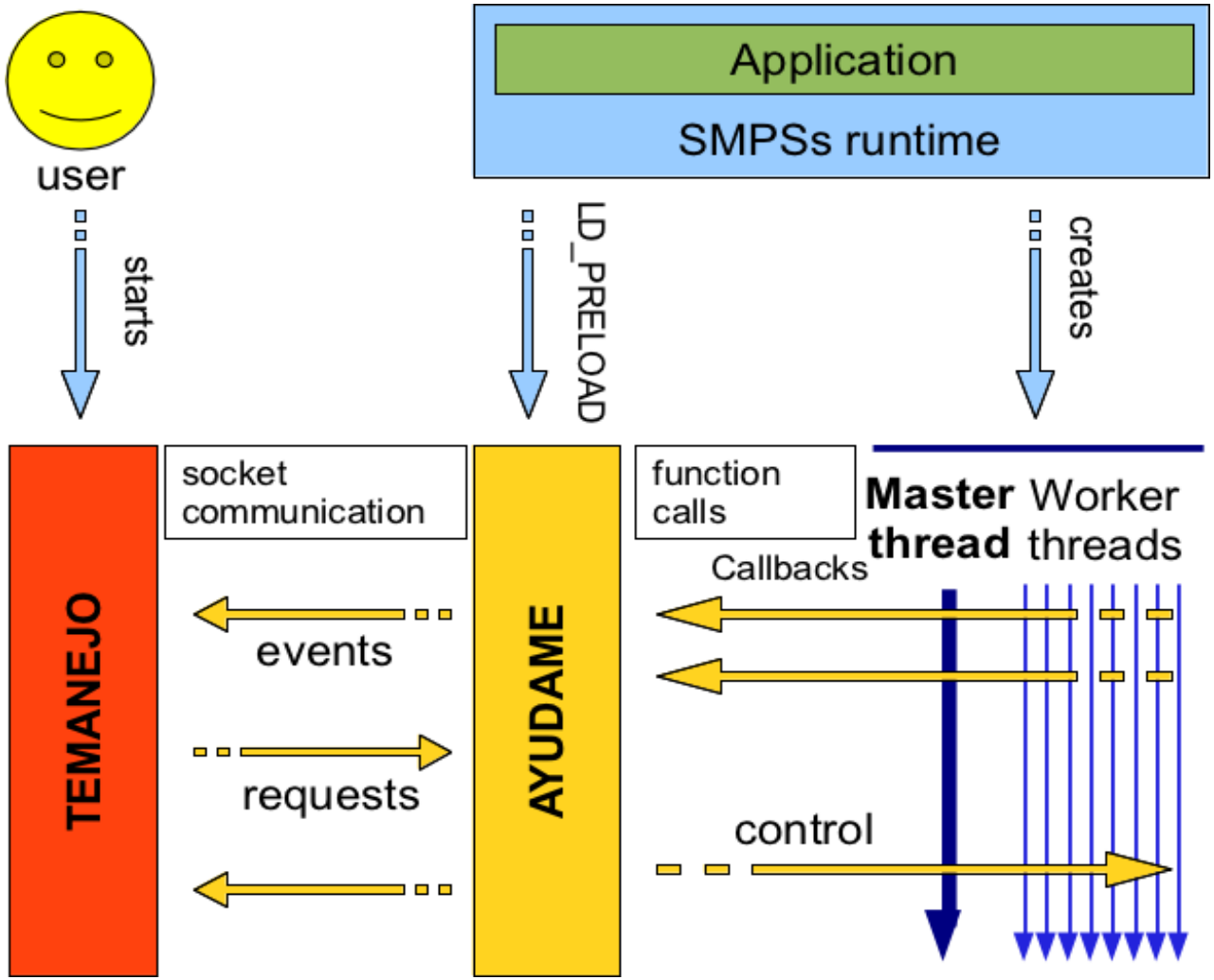
```
#pragma css task        input (A) output(B)
void copy( float A[NUM_ELEM], float B[NUM_ELEM) {
        for (i = 0; i < NUM_ELEM; i++)
                B[i] = A[i];
}
```

Debugging with Ayudame/Temanejo

- TEMANEJO is a graphical debugger for task based parallel programming models

- It allows to:
  - Visualize dependency graph and synchronization
  - Analyze dependencies and data involved
  - Control the scheduling of tasks
  - Interface with gdb for traditional debugging
    - on function level (i.e., stepping, value inspection, …)
- where the application may be running on a remote compute node or locally

TEMANEJO – The StarSs Debugger



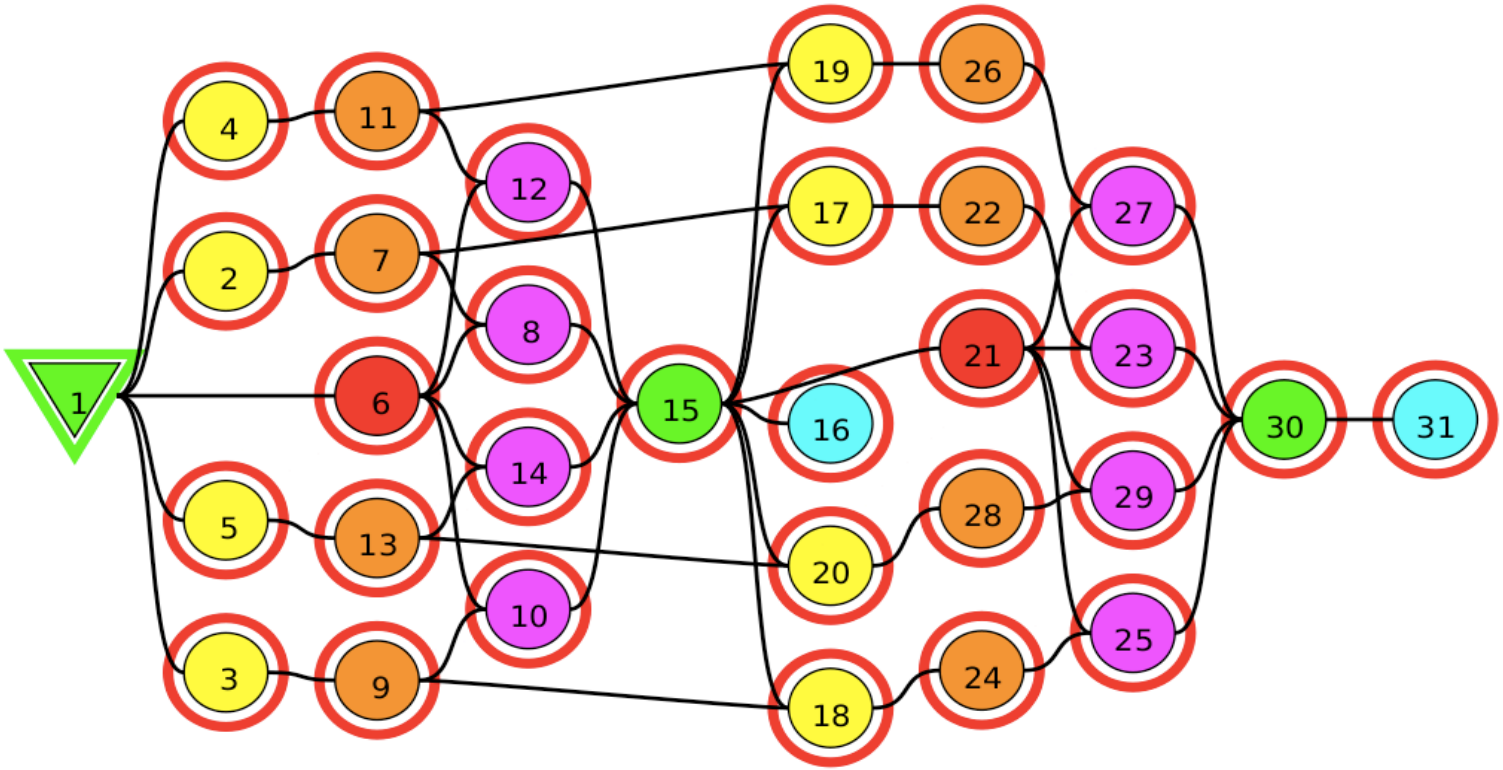
Basic Usage: Visualize Dep Graph

function	NQ	Q	R	F	total
bnd	2	0	1	0	3
decompose	8	0	0	0	8
dummy	2	0	0	0	2
outp	2	0	0	0	2
recompose	8	0	0	0	8
step	8	0	0	0	8
total	30	0	1	0	31

Breakpoints

Stop scheduling tasks

Stop on any event



Features

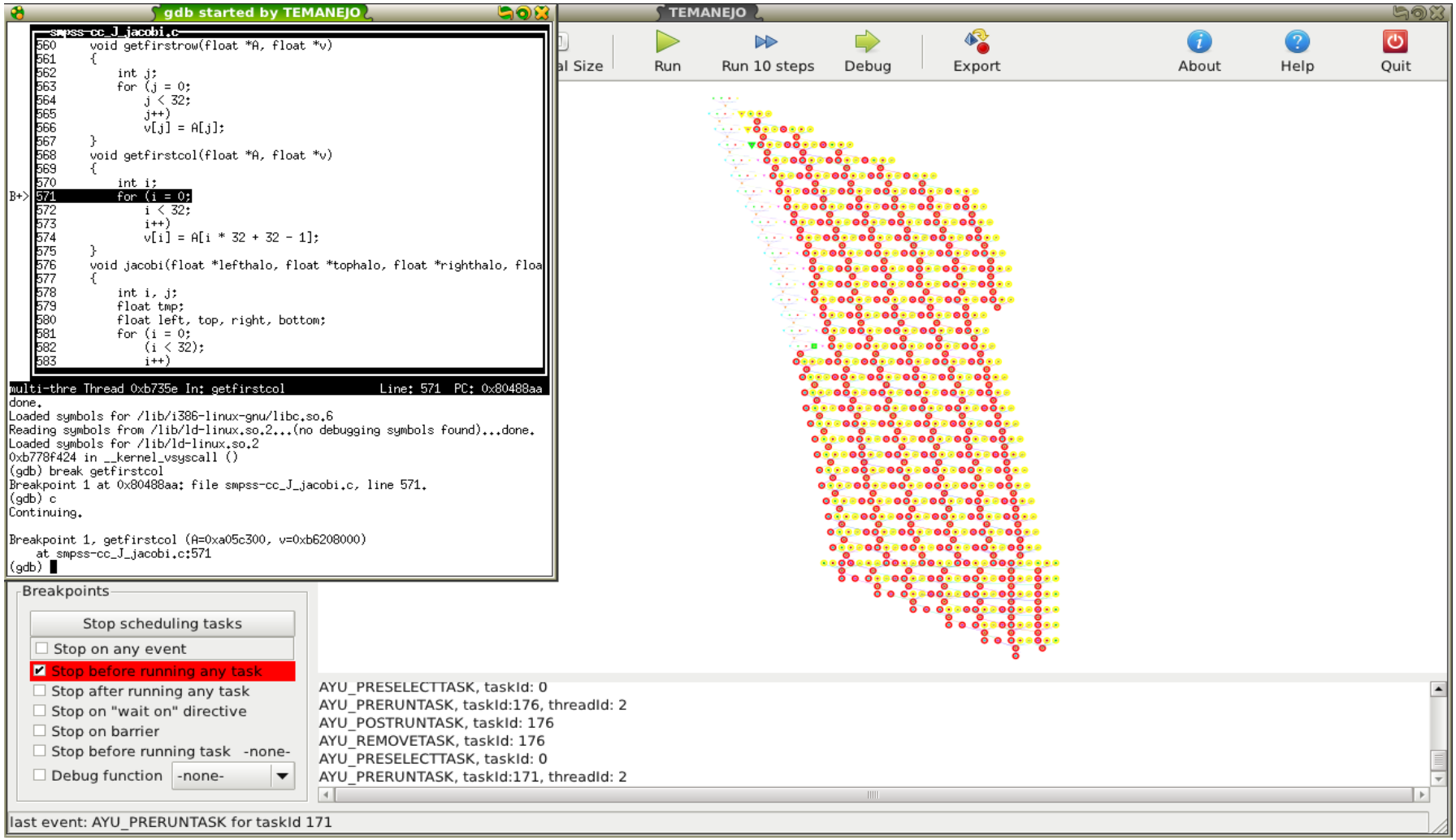
Information

- Task dependency graph
  - Nodes (task id, function id)
  - Edges (data memory addresses)
  - Parents and children of tasks (neighbors)
- Scheduling status of task
  - Running
  - Queued (all dependencies resolved)
  - Not-queued (some dependencies unresolved)
- Execution status of task once running
  - Thread id of executing worker
  - Timings/duration
- Information shown in node color, shape and border color is freely configurable

Controls

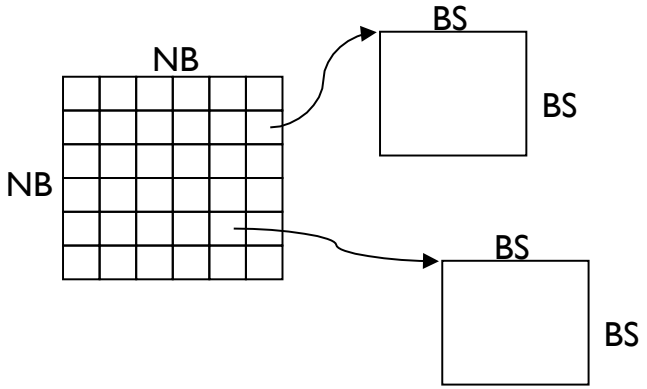
- Breakpoints on master thread
  - Barrier directive
  - Wait_on directive
- Breakpoints on worker / task
  - Before running a task
  - After running a task
- Scheduling
  - Don't schedule any task
  - Stepping through events
  - (Serialization of tasks)
  - (Blocking/running of tasks)
- Interactive gdb window to step on function/source line level

Integration with gbd



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]);
}
```



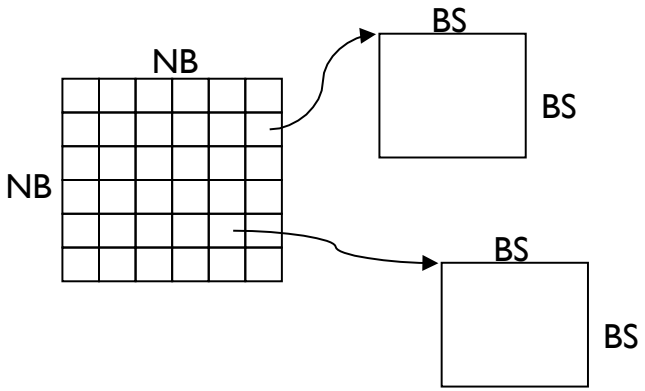
Will work on matrices of any size

Will work on any number of cores/devices

```
#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];
}
```

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, NB) \
                 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, 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], 
}
```

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

   dim3 dimBlock(BLOCK_SIZ
    dimBlock.x = BLOCK_SIZ
    dimBlock.y = BLOCK_SIZ
   dim3 dimGrid;
    dimGrid.x = (wB / dimB
    dimGrid.y = (hA / dimB
   Muld <<<dimGrid, dimBlo
}
```

```
__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;
}
```

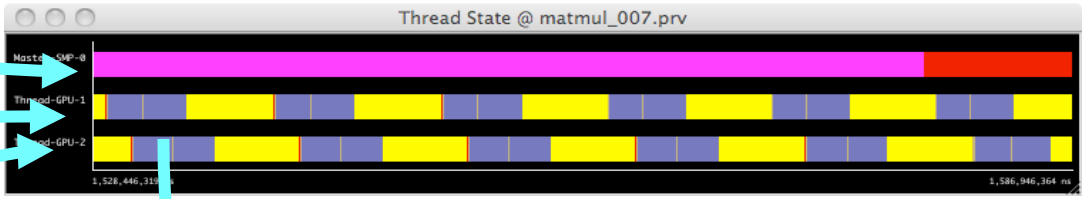
Matrix multiply: multi-GPUs

Two Intel Xeon E5440, 4 cores
4 Tesla S2050 GPUs

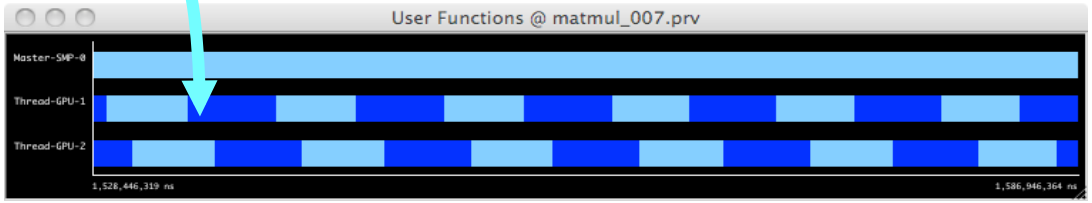
main thread

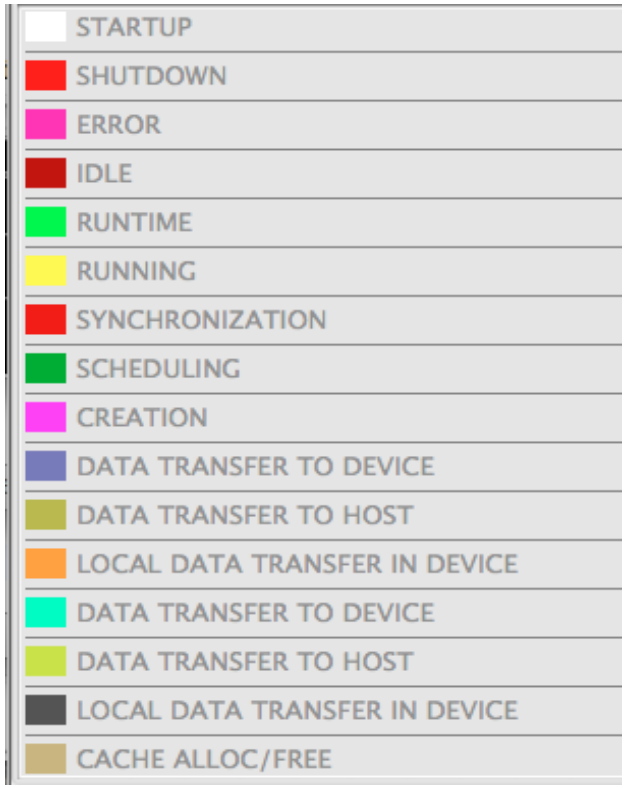
GPU thread

GPU thread

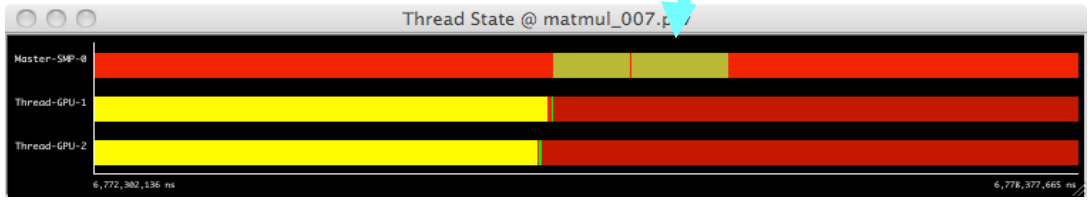


Copying data to GPU

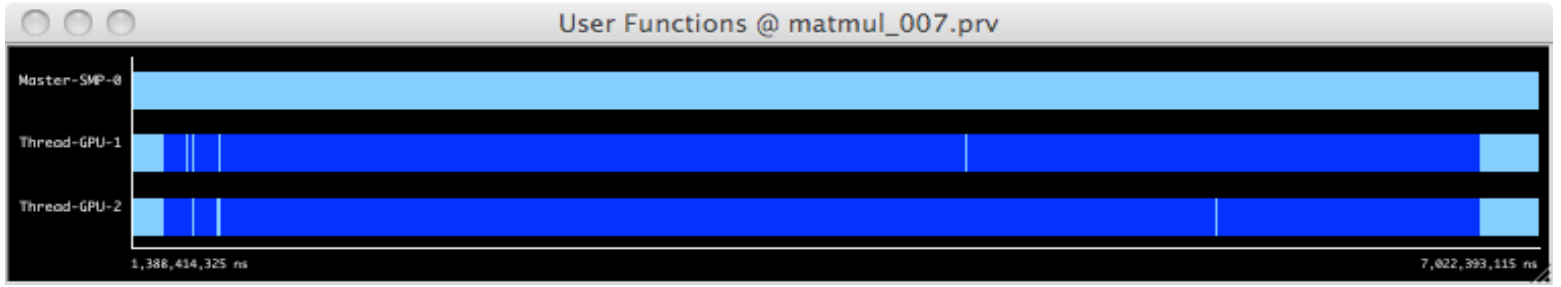


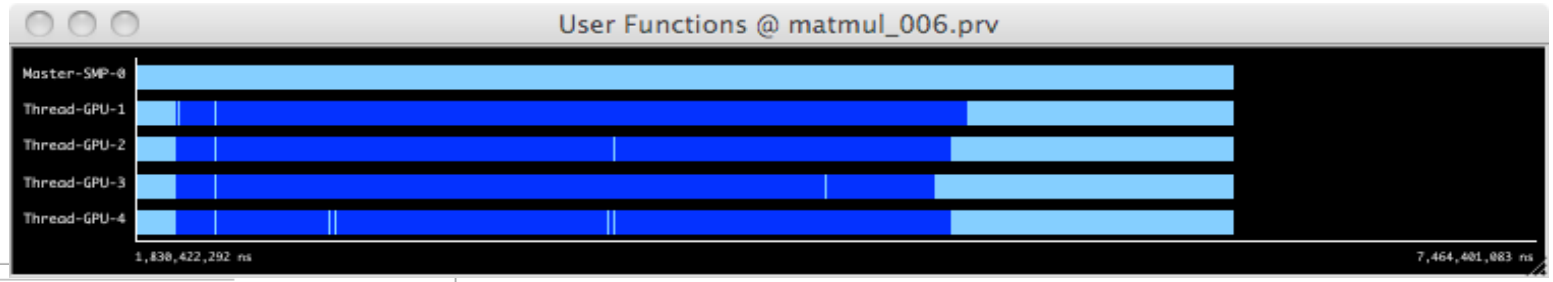


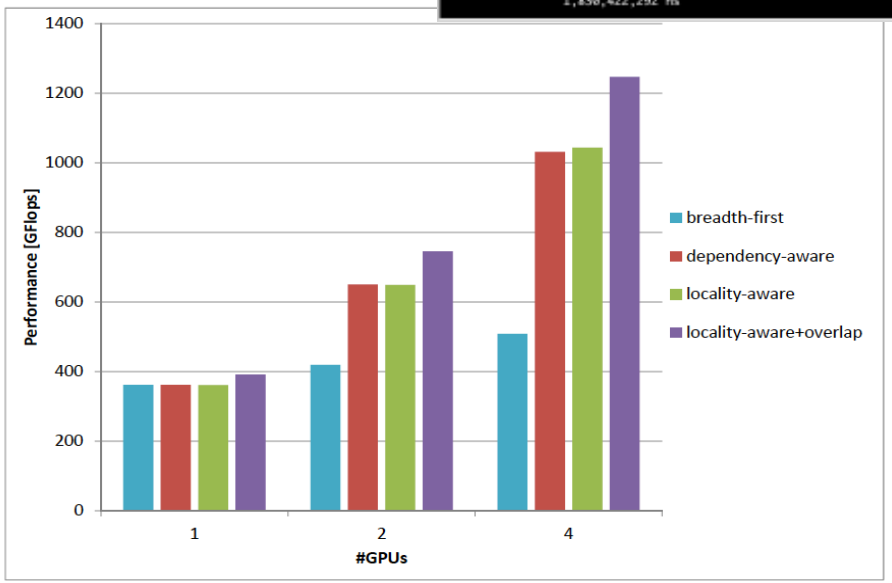
All copies to host at end of application



Matrix multiply: multi-GPUs

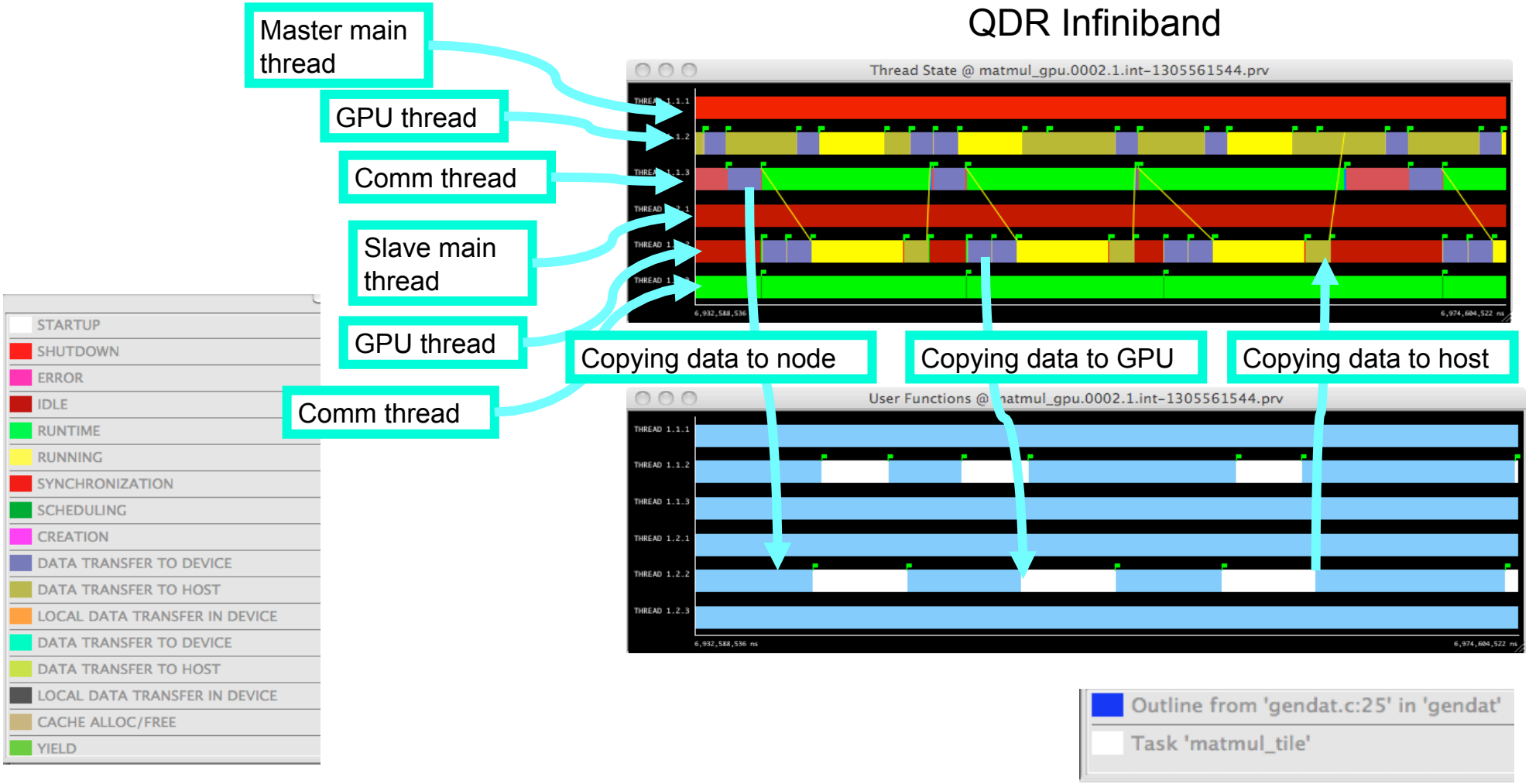






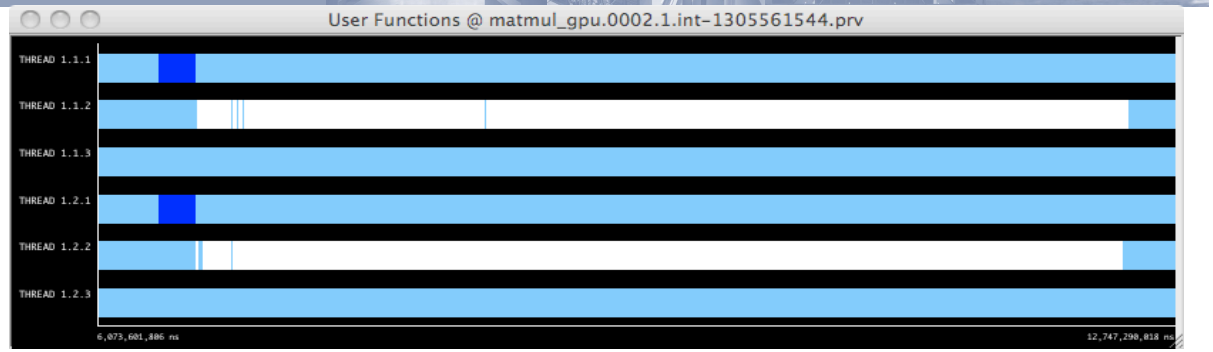
Matrix multiply: clusters of GPUs

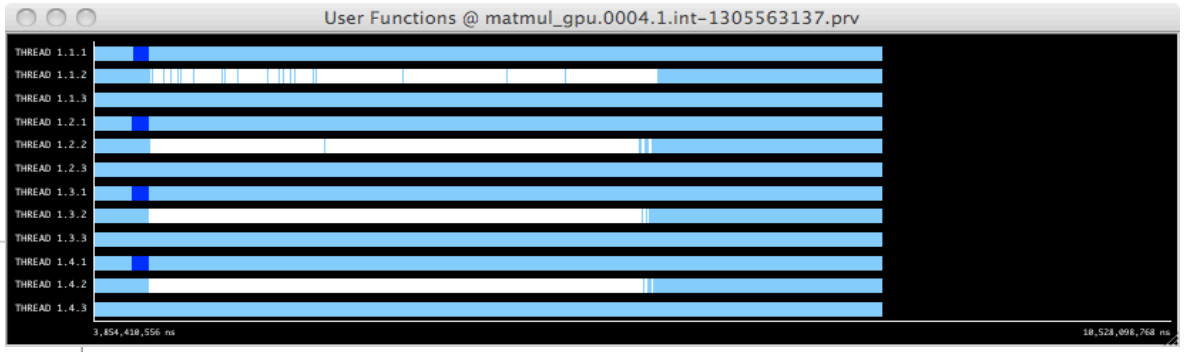
2 nodes of DAS-4
each node:
Two Intel Xeon E5620, 4 cores
1 GTX 480 GPU
QDR Infiniband

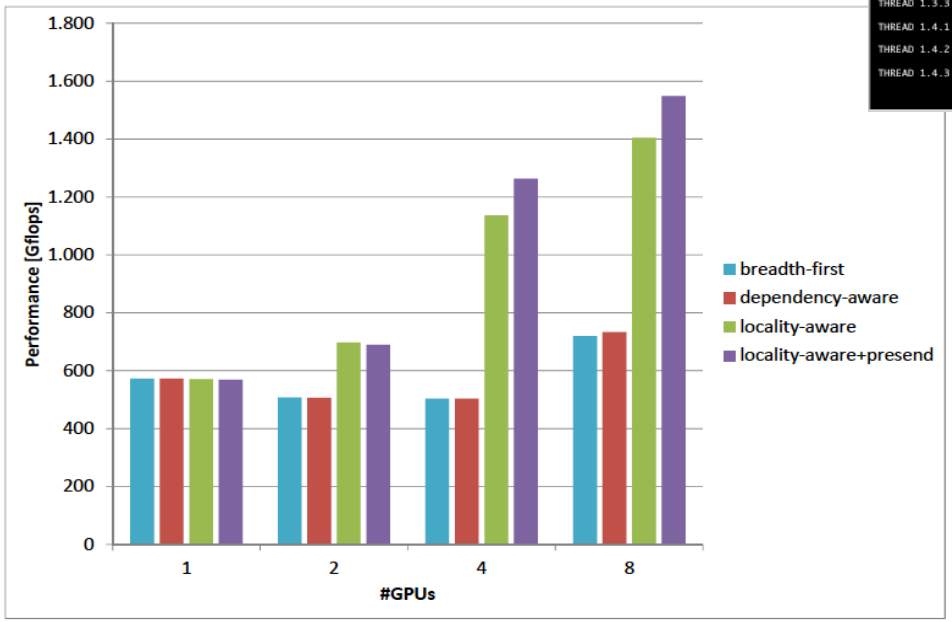


Matrix multiply: clusters of GPUs

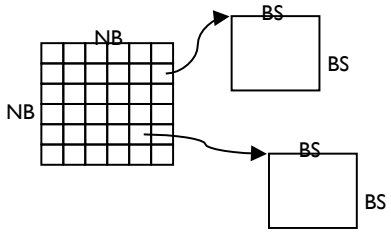
2 and 4 nodes of DAS-4
each node:
Two Intel Xeon E5620, 4 cores
1 GTX 480 GPU
QDR Infiniband

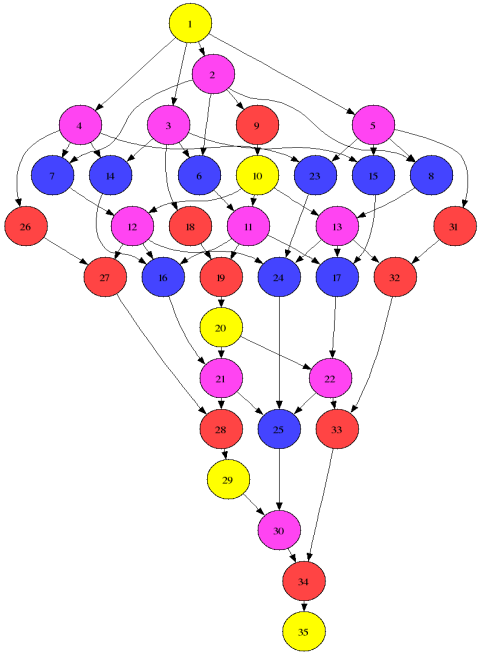






Cholesky: Block matrix storage



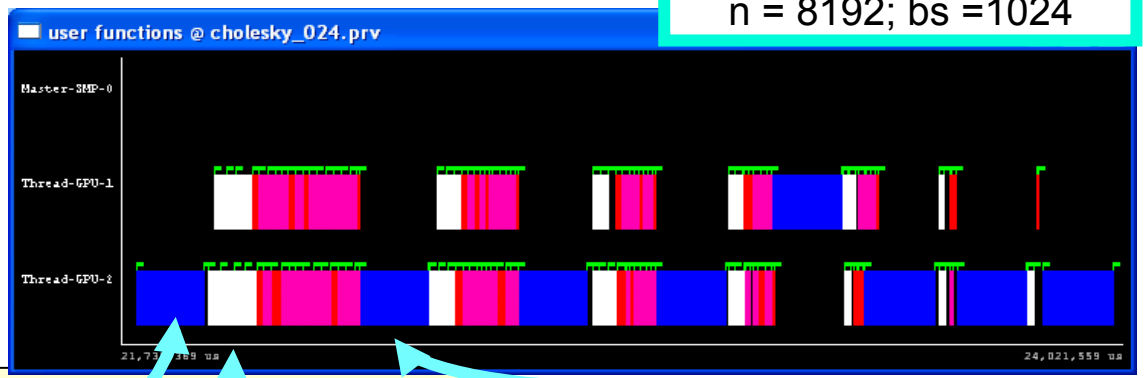


```
void blocked_cholesky( int NT, float *A ) {
   int i, j, k;
   for (k=0; k<NT; k++) {
      spotrf (A[k*NT+k]) ;
      for (i=k+1; i<NT; i++)
         strsm (A[k*NT+k], A[k*NT+i]);
      // update trailing submatrix
      for (i=k+1; i<NT; i++) {
         for (j=k+1; j<i; j++)
            sgemm(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

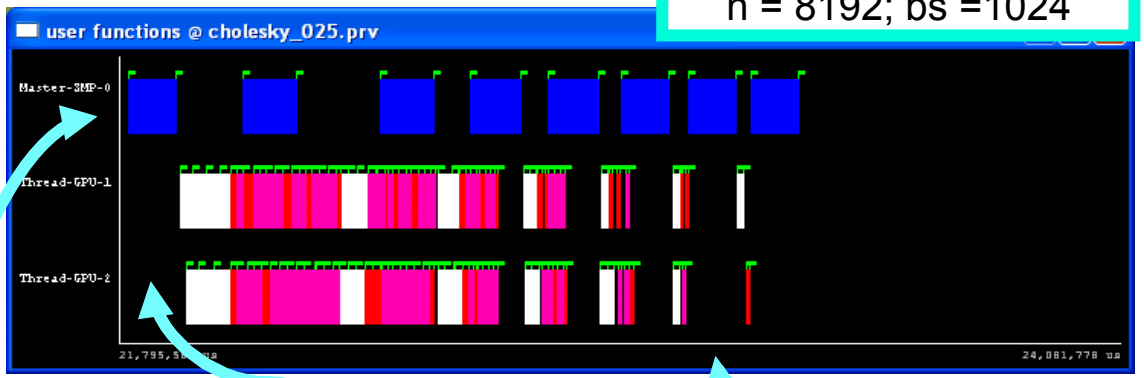
```
void blocked_cholesky( int NT, float *A ) {
   int i, j, k;
   for (k=0; k<NT; k++) {
      spotrf (A[k*NT+k]) ;
      for (i=k+1; i<NT; i++)
         strsm (A[k*NT+k], A[k*NT+i]);
      // update trailing submatrix
      for (i=k+1; i<NT; i++) {
         for (j=k+1; j<i;
            sgemm( A[k*NT+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

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

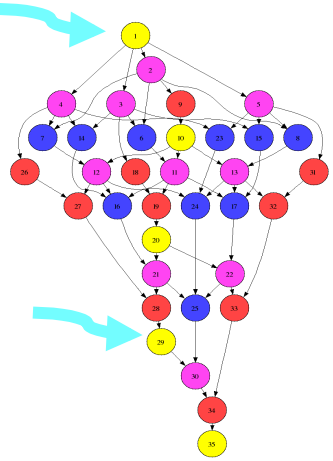
```
#pragma omp target device (cuda) copy_deps
#pragma omp task inout ([TS][TS]A)
void spotrf (float *A);
#pragma omp target device (cuda) copy_deps
#pragma omp task input ([TS][TS]A) inout ([TS][TS]C)
void ssyrk (float *A, float *C);
#pragma omp target device (cuda) copy_deps
#pragma omp task input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



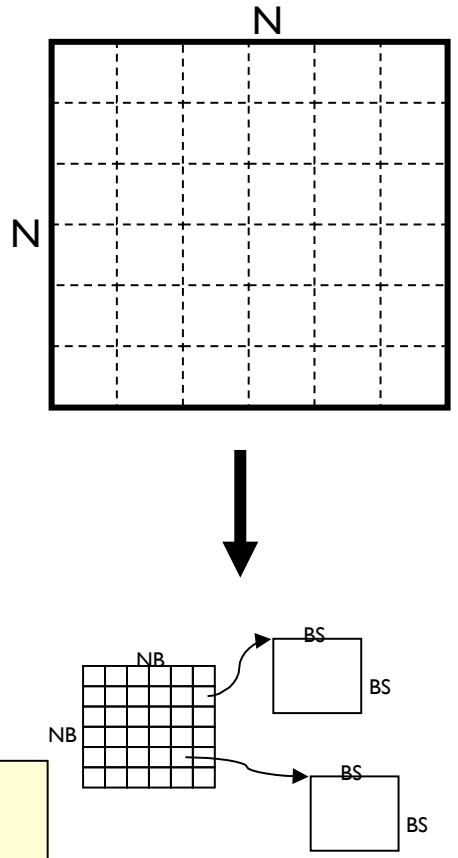

```
#pragma omp target device (smp) copy_deps
#pragma omp task inout([NB][NB]A)
void spotrf_tile(float *A, int NB)
{
    long INFO;
    char L = 'L';

    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



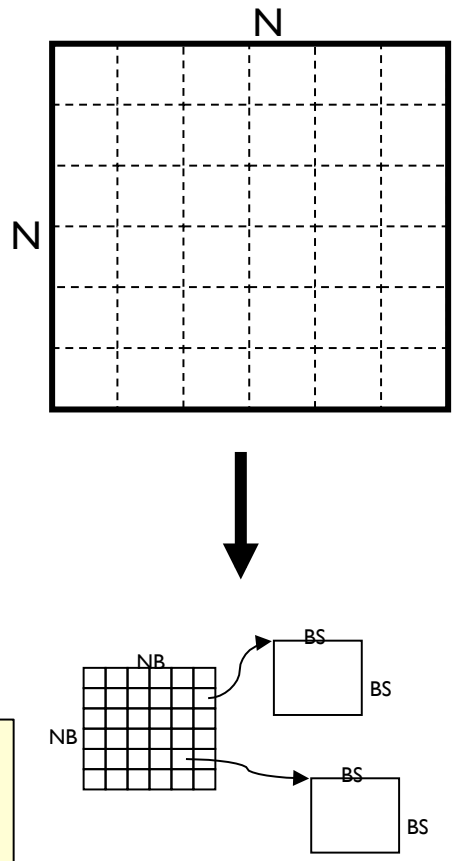

```
void convert_to_block( int n, int nt, float * A , float **Ah) {
  for (i=0; i<nt; i++)
    for (j=0; j<nt; j++) gather_block (n, A, i, j, Ah[i*nt+j]]);
}
```

```
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)
}
```

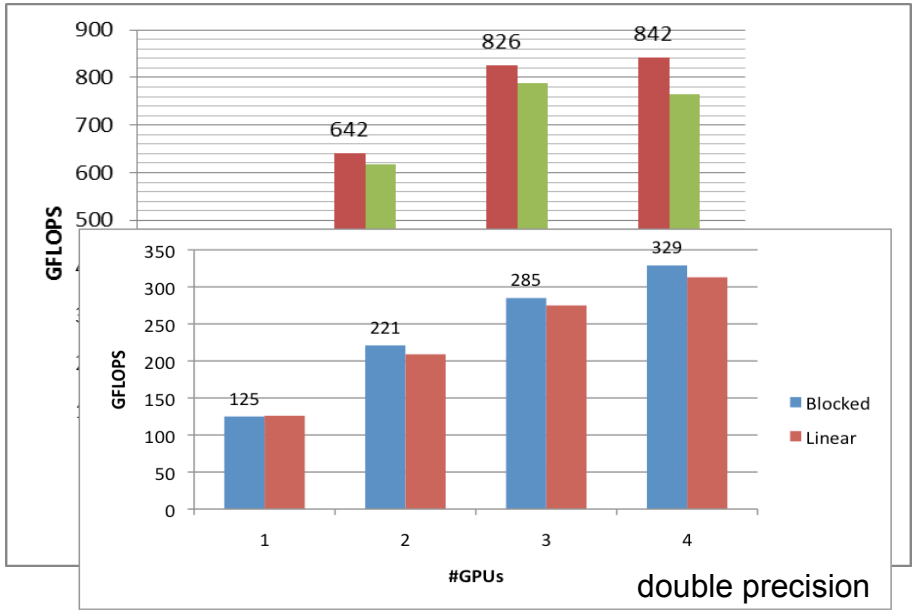
Recursion, a nice way to
refine parallelism
reduce granularity

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

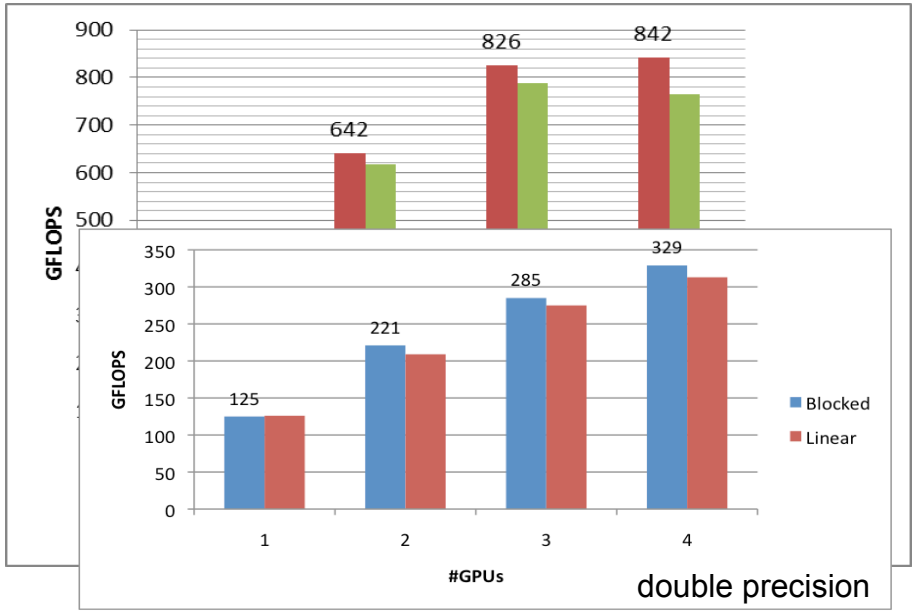


Blocked

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

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
  - Come with development/performance tools
- 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:

    pm.bsc.es/ompss

The TEXT project

- Towards EXaflop applicaTions

- Demonstrate that **Hybrid MPI/SMPSs** addresses the Exascale challenges in a productive and efficient way.
  - Deploy at supercomputing centers
  - 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)

THANKS!!!