Barcelona Supercomputing Center
Centro Nacional de Supercomputación

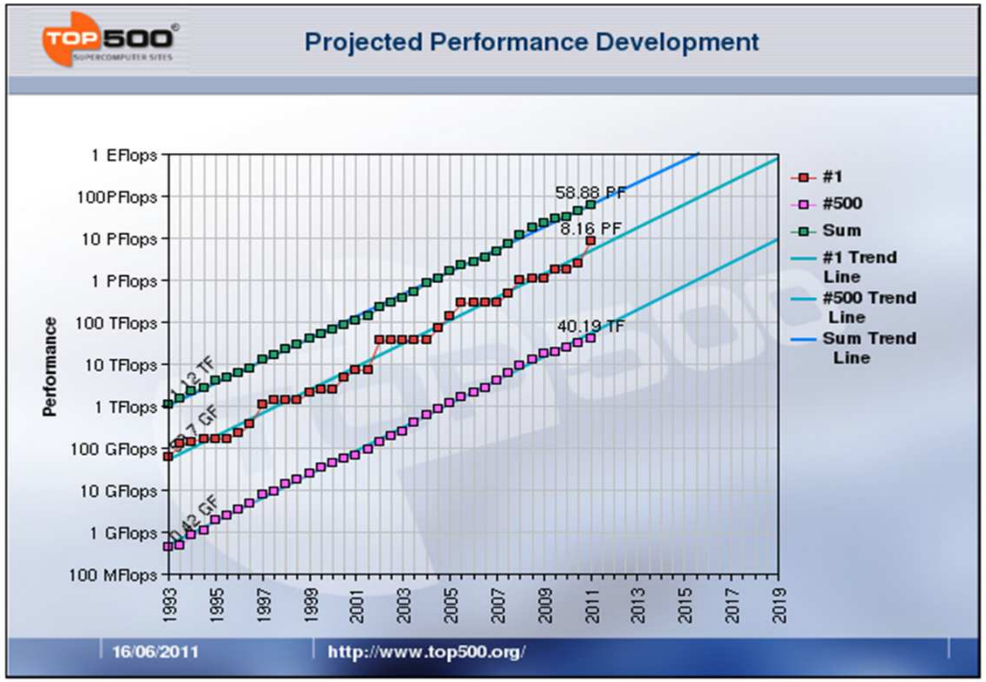
Dataflow programming models and OmpSs

Rosa M. Badia, Eduard Ayguadé, Jesús Labarta, Alejandro Duran, Xavier Martorell

Computer Sciences Research Dept.

BSC

Evolution of computers



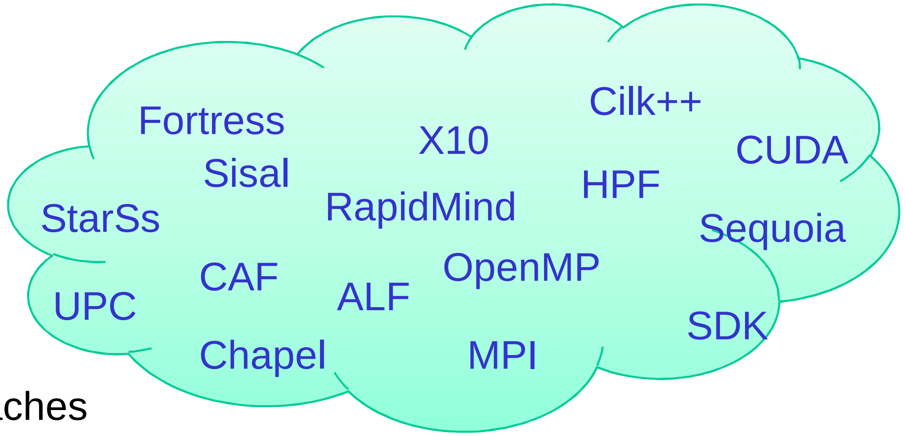
GPU, accelerator

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

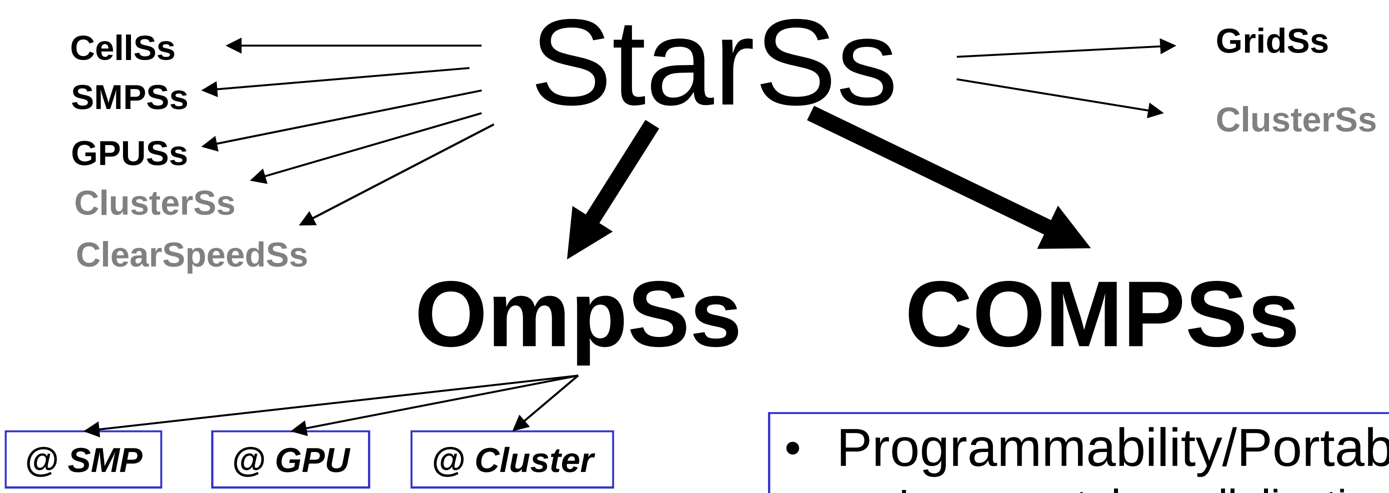
- Traditional programming models
  - Message passing (MPI)
  - OpenMP
  - Hybrid MPI/OpenMP
- Heterogeneity
  - CUDA
  - OpenCL
  - ALF
  - RapidMind
- Alternative approaches
  - Partitioned Global Address Space (PGAS) programming models
    - UPC, X1
- ...



Simple programming paradigms that enable easy application development are required

The StarSs programming model

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



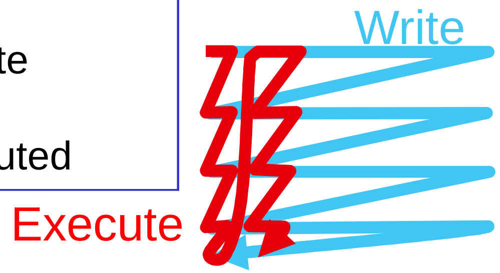
- StarSs
  - A "node" level programming model
  - Sequential C/Fortran/Java + annotations
  - Task based. Asynchrony, data-flow.
  - "Simple" linear address space
  - Directionality annotations on tasks arguments
  - Malleable
  - Nicely integrates in hybrid MPI/StarSs
  - Natural support for heterogeneity

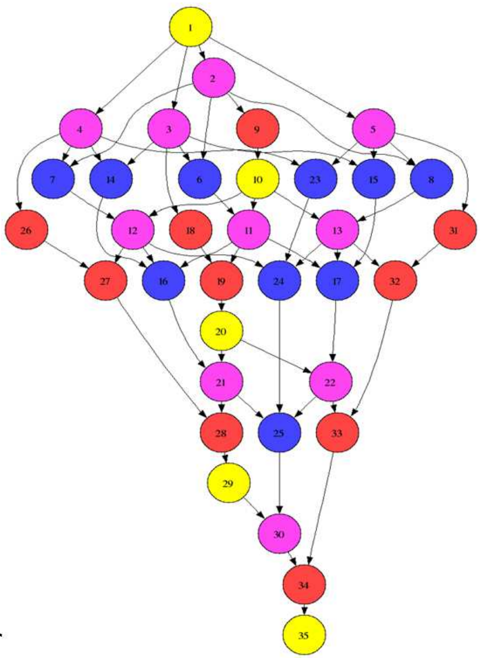
- Programmability/Portability
  - Incremental parallelization/restructure
  - Separate algorithm from resources
  - Disciplined programming
  - **"Same" source code runs on "any" machine**
    - Optimized task implementations will result in better performance.
- Performance
  - Intelligent Runtime
    - 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



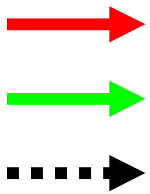


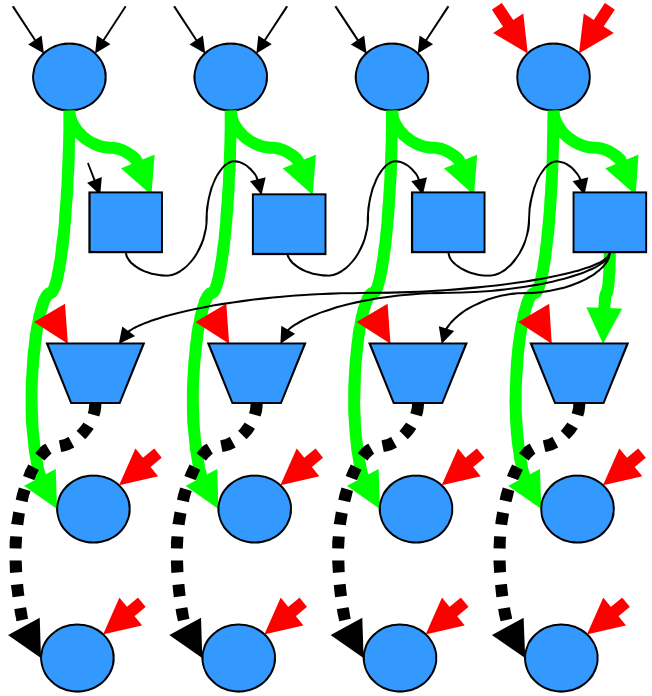
```
void Cholesky( float *A ) {
   int i, j, k;
   for (k=0; k<NT; k++) {
      spotrf (A[k*NT+k]) ;
      for (i=k+1; i<NT; i++)
         strsm (A[k*NT+k], A[k*NT+i]);
      // update trailing submatrix
      for (i=k+1; i<NT; i++) {
         for (j=k+1; j<i; j++)
            sgemm( A[k*NT+i], A[k*NT+j],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





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);
}
```

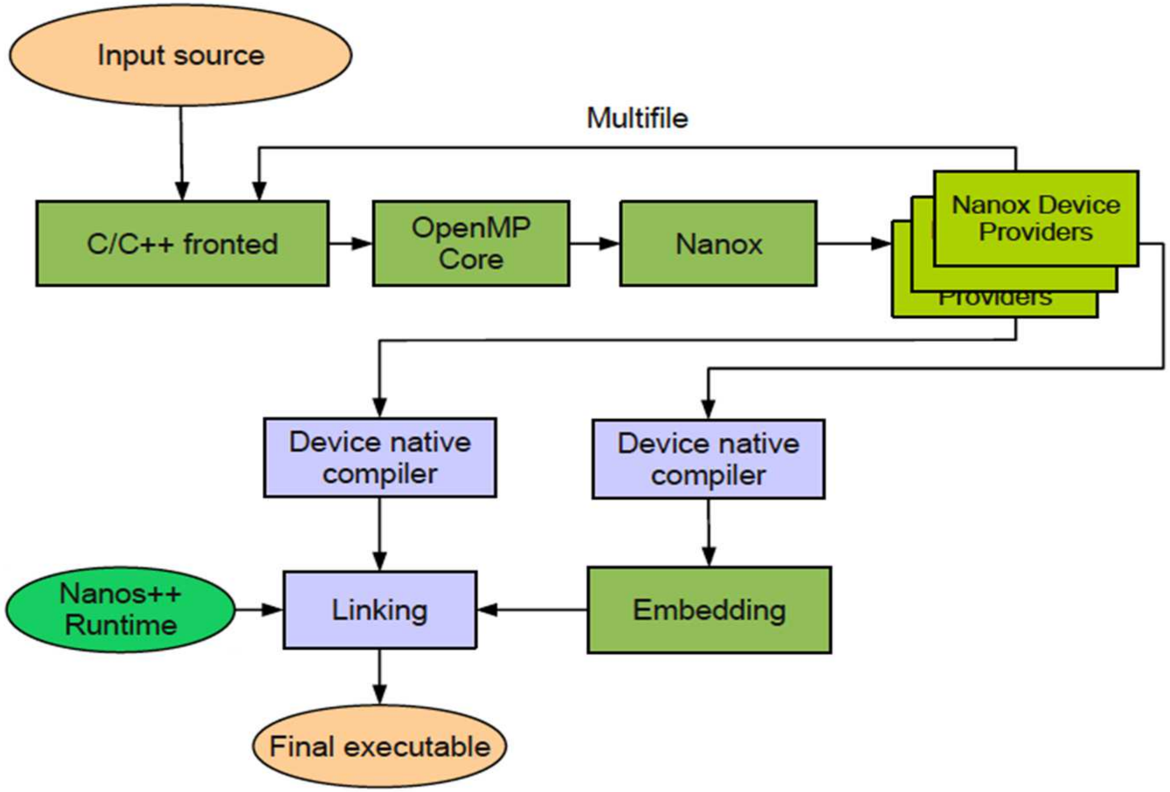
Avoiding data transfers

- 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
}
```

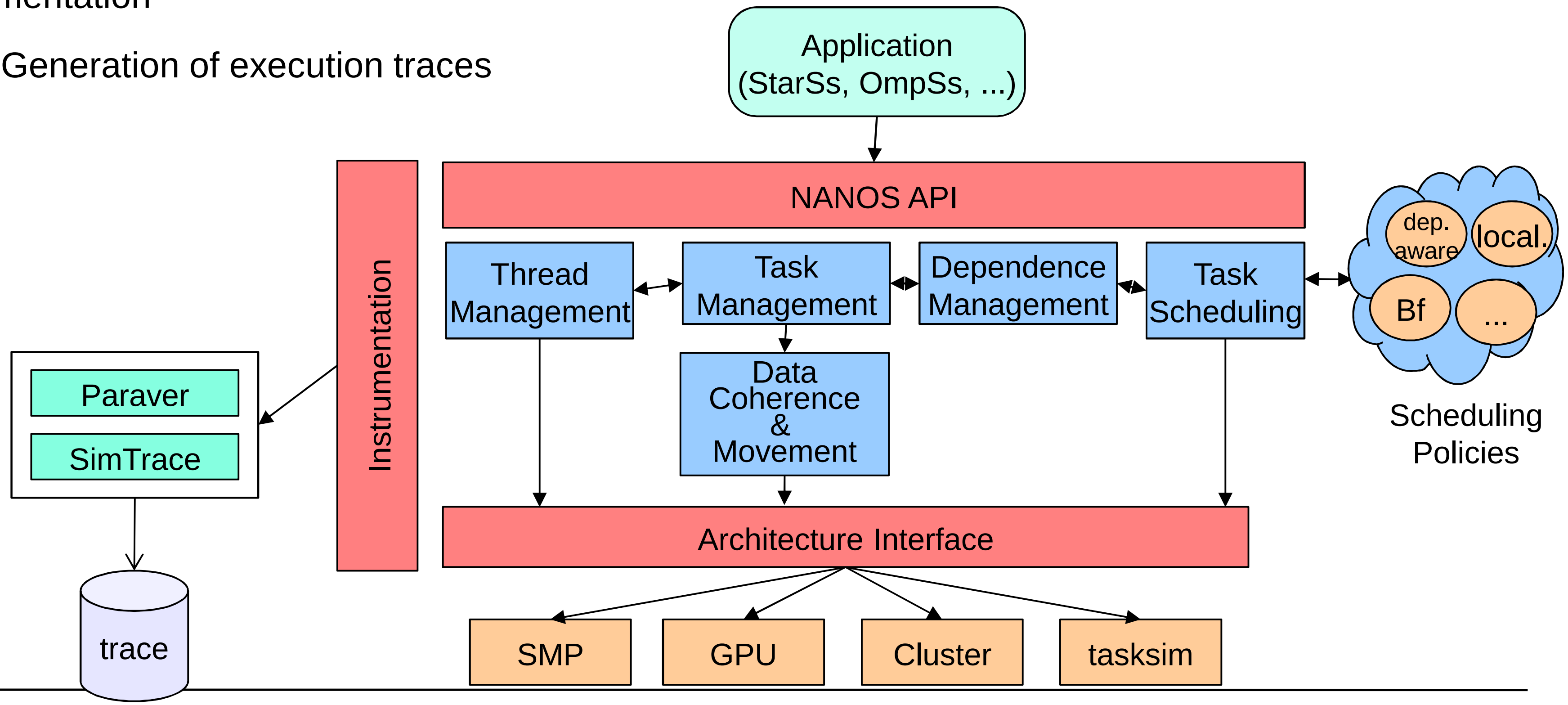
Mercurium Compiler

- Source to source compiler
- 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



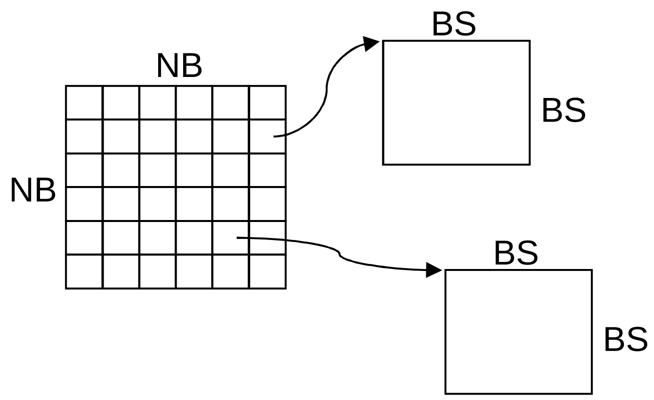
Runtime structure

- 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



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]);
}
```



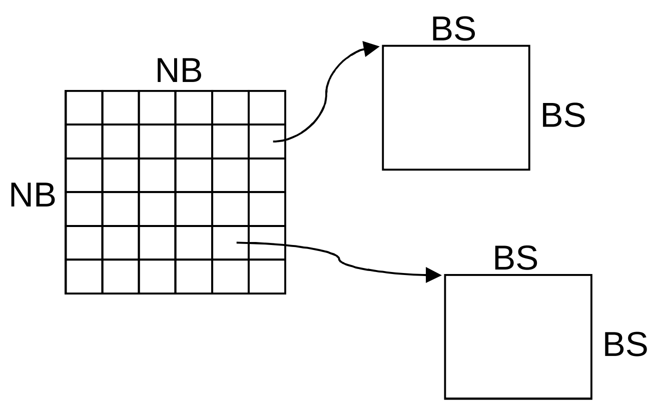

```
#pragma omp task input([BS][BS]A, [BS][BS]B)\
              inout([BS][BS]C)
static void mm_tile ( float C[BS][BS], float A[BS][BS],
                      float B[BS][BS]) {
int i, j, k;

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 @ 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

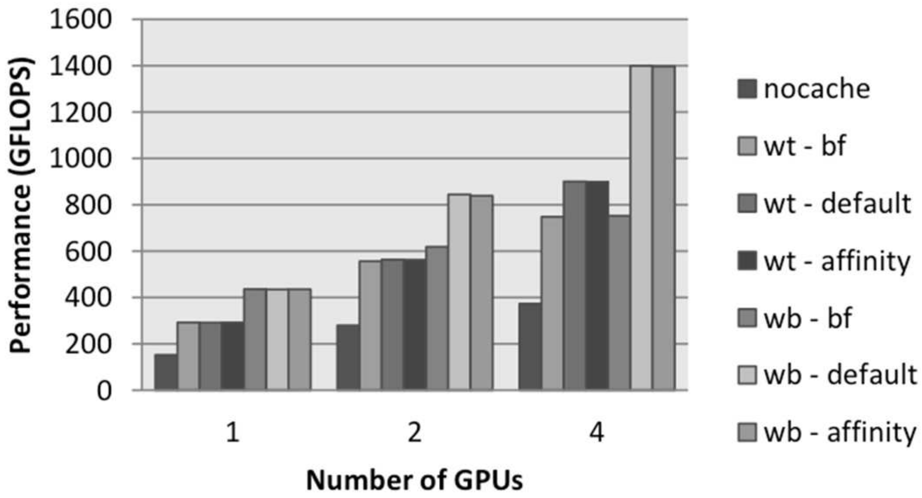
```
int main  (int argc, char **argv) {
int i, j, k;
…

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], A
}
```

```
#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_SIZE;
dimBlock.y = BLOCK_SIZE;
   dim3 dimGrid;
dimGrid.x = (wB / dimBlock
dimGrid.y = (hA / dimBlock
   Muld <<<dimGrid, dimBlo
}
```

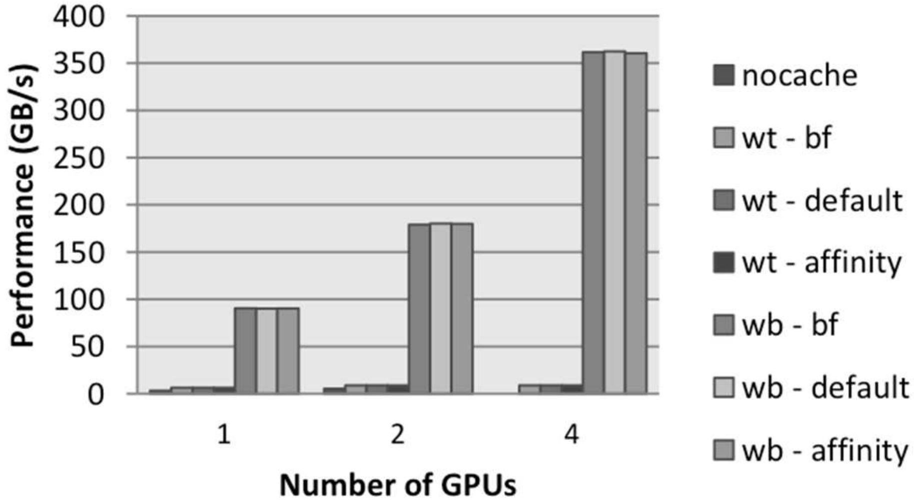
```
__global__ void Muld(float* A, float* B, int wA, int wB, float* C) {
  int bx = blockIdx.x; int by = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;
  int aBegin = wA * BLOCK_SIZE * by;
  int aEnd   = aBegin + wA – 1;
  int aStep = BLOCK_SIZE;
  int bBegin = BLOCK_SIZE * bx;
  int bStep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

A few results in the multi-GPU environment

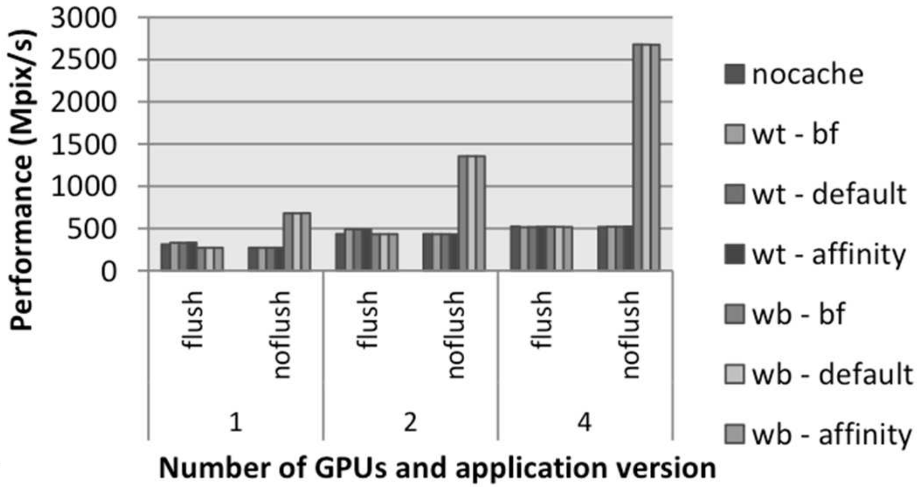
Matrix multiply



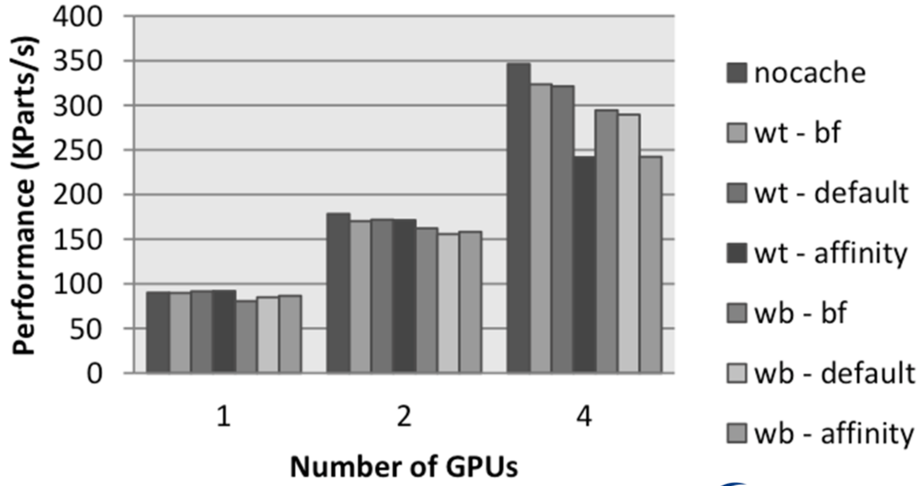
Stream



Perlin Noise

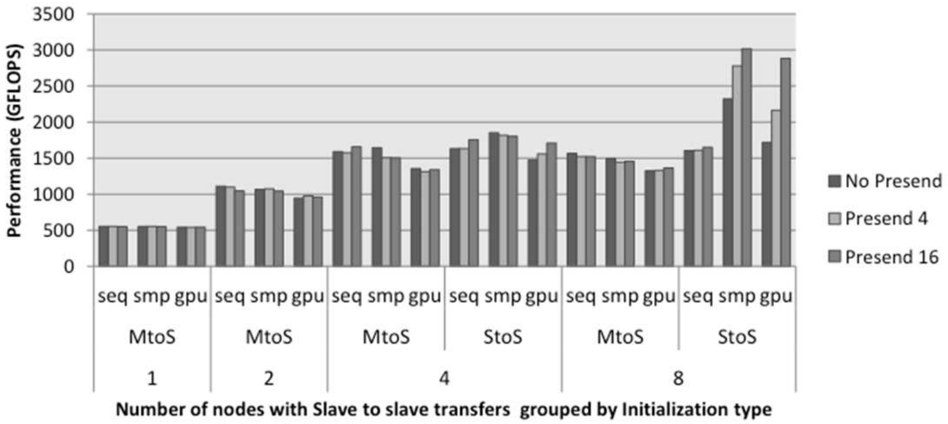


n-Body

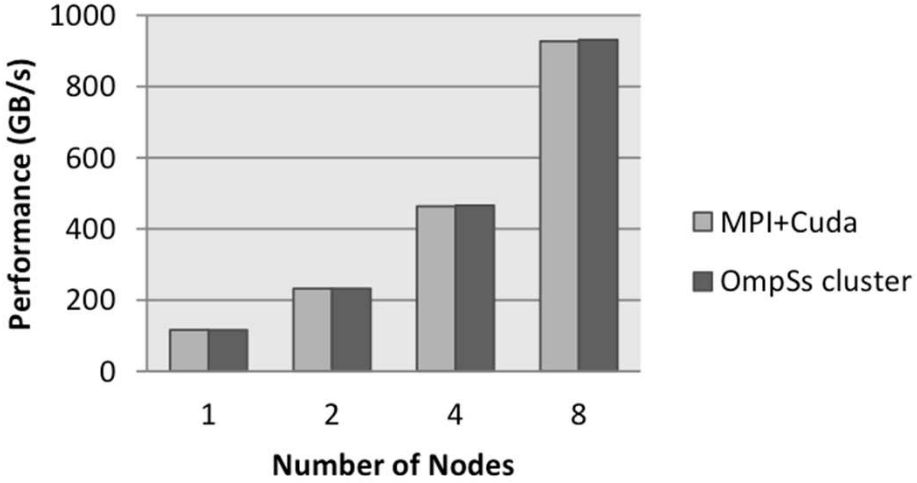


A few results in the Cluster of GPUs environment

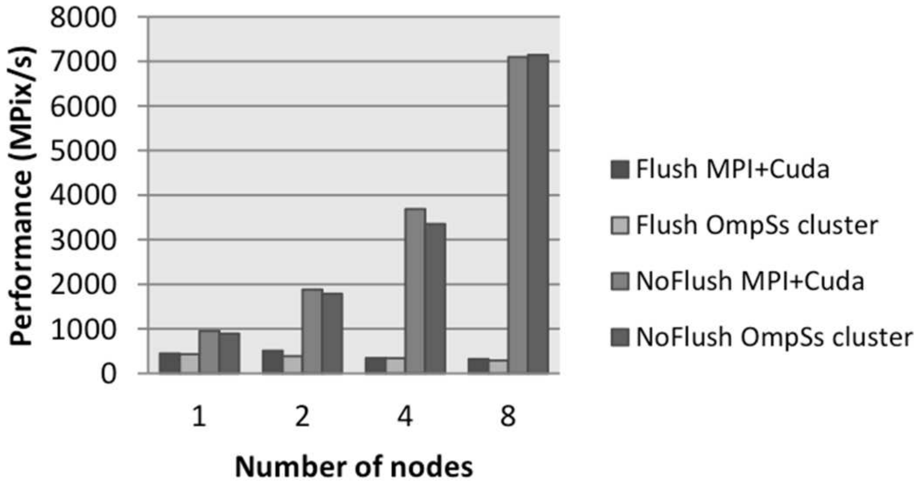
Matrix multiply



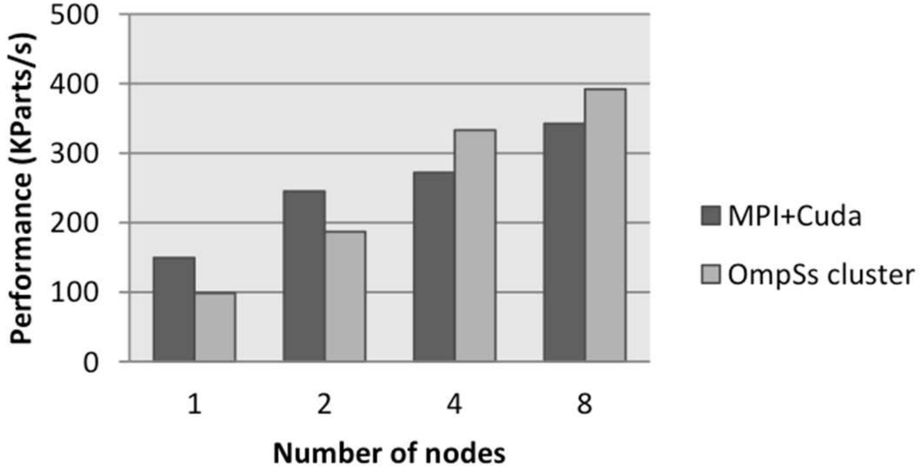
Stream



Perlin Noise



n-Body



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

THANKS!!!