OmpSs - programming model for heterogenous and distributed platforms

Rosa M Badia
Evolution of computers

All include multicore or GPU/accelerators

<table>
<thead>
<tr>
<th>Rank</th>
<th>Site</th>
<th>System</th>
<th>Cores</th>
<th>Rmax</th>
<th>Rpeak</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DOE/SC/Oak Ridge National Laboratory United States</td>
<td>Titan - Cray XK7, Opteron 6274 16C 2.20GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.</td>
<td>160640</td>
<td>17560.0</td>
<td>27112.6</td>
<td>8200</td>
</tr>
<tr>
<td>2</td>
<td>DOE/ANL/LBL United States</td>
<td>Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom BM</td>
<td>1572564</td>
<td>16324.8</td>
<td>20132.7</td>
<td>7890</td>
</tr>
<tr>
<td>3</td>
<td>RIKEN Advanced Institute for Computational Science (AICS) Japan</td>
<td>K computer, SPARC64 VIIIfx 2.00GHz, Tohoku interconnect Fujitsu</td>
<td>705024</td>
<td>10510.3</td>
<td>11280.4</td>
<td>12500</td>
</tr>
<tr>
<td>4</td>
<td>DOE/SC/Argonne National Laboratory United States</td>
<td>Mira - BlueGene/Q, Power BQC 16C 1.60GHz, Custom BM</td>
<td>786432</td>
<td>9162.4</td>
<td>10066.3</td>
<td>3946</td>
</tr>
<tr>
<td>5</td>
<td>Forschungszentrum Juelich (FZJ) Germany</td>
<td>JUQUEEN - BlueGene/Q, Power BQC 16C 1.60GHz, Custom Interconnect IBM</td>
<td>393216</td>
<td>4141.2</td>
<td>5033.2</td>
<td>1970</td>
</tr>
<tr>
<td>6</td>
<td>Leibniz Rechenzentrum Germany</td>
<td>SuperMUC - DataPlex DS60M4, Xeon E5-2660 6C 2.70GHz, Infinitband FDR IBM</td>
<td>147456</td>
<td>2897.0</td>
<td>3185.1</td>
<td>3423</td>
</tr>
<tr>
<td>7</td>
<td>Texas Advanced Computing Center/Univ. of Texas United States</td>
<td>Stampede - PowerEdge C8220, Xeon E5-2660 6C 2.70GHz, Infinitband FDR, Intel Xeon Phi</td>
<td>204960</td>
<td>2660.3</td>
<td>3959.0</td>
<td>7260</td>
</tr>
<tr>
<td>8</td>
<td>National Supercomputing Center in Tianjin China</td>
<td>Tianhe-1A - NUDT YH MPP, Xeon X5670 6C 2.93GHz, NVIDIA 2050</td>
<td>186388</td>
<td>2566.0</td>
<td>4701.0</td>
<td>4040</td>
</tr>
<tr>
<td>9</td>
<td>CINECA Italy</td>
<td>Fermi - BlueGene/Q, Power BQC 16C 1.60GHz, Custom BM</td>
<td>163840</td>
<td>1725.5</td>
<td>2097.2</td>
<td>822</td>
</tr>
<tr>
<td>10</td>
<td>IBM Development Engineering United States</td>
<td>DARPA Trial Subset - Power 775, POWER7 8C 3.85GHz, Custom Interconnect IBM</td>
<td>63000</td>
<td>1516.0</td>
<td>1944.4</td>
<td>3676</td>
</tr>
</tbody>
</table>
Parallel programming models

- **Traditional programming models**
  - Message passing (MPI)
  - OpenMP
  - Hybrid MPI/OpenMP

- **Heterogeneity**
  - CUDA
  - OpenCL
  - ALF
  - RapidMind

- **New approaches**
  - Partitioned Global Address Space (PGAS) programming models
    - UPC, X10, Chapel

Simple programming paradigms that enable easy application development are required
Outline

• StarSs overview
• OmpSs syntax
• OmpSs examples
• OmpSs + heterogeneity
• OmpSs compiler & runtime
• OmpSs environment and further examples

• Contact: pm-tools@bsc.es
• Source code available from http://pm.bsc.es/ompss/
StarSs overview
StarSs: a family of **task based** programming models

- Basic concept: **write sequential on a flat single address space** + **directionality annotations**
  - Dependence and data access information in a single mechanism
  - Runtime task-graph dependence generation
  - Intelligent runtime: scheduling, data transfer, support for heterogeneity, support for distributed address space
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]T) inout ([TS][TS]B)
void strsm (float *T, float *B);
#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]A) inout ([TS][TS]C)
void ssyrk (float *A, float *C);
void Cholesky( float *A ) {
    int i, j, k;
    for (k=0; k<NT; k++) {
        spotrf (A[k*NT+k]);
        #pragma omp parallel for
        for (i=k+1; i<NT; i++)
            strsm (A[k*NT+k], A[k*NT+i]);
        for (i=k+1; i<NT; i++) {
            #pragma omp parallel for
            for (j=k+1; j<i; j++)
                sgemm( A[k*NT+i], A[j*NT+i] );
            #pragma omp task
            ssyrk (A[k*NT+i], A[i*NT+i] );
            #pragma omp taskwait
        }
    }
}

void Cholesky( float *A ) {
    int i, j, k;
    for (k=0; k<NT; k++) {
        spotrf (A[k*NT+k]);
        #pragma omp parallel for
        for (i=k+1; i<NT; i++)
            strsm (A[k*NT+k], A[k*NT+i]);
        for (i=k+1; i<NT; i++) {
            #pragma omp task
            #pragma omp parallel for
            for (j=k+1; j<i; j++)
                sgemm( A[k*NT+i], A[k*NT+j], A[j*NT+i] );
            #pragma omp task
            ssyrk (A[k*NT+i], A[i*NT+i] );
            #pragma omp taskwait
        }
    }
}
OmpSs syntax
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
OmpSs: Directives

Task implementation for a GPU device
The compiler parses CUDA/OpenCL kernel invocation syntax

Provides configuration for CUDA/OpenCL kernel

```
#pragma omp target device ( { smp | cuda | opencl } )
   [ ndrange (…)]
   [ implements ( function_name ) ]

   { copy_deps | [ copy_in ( array_spec ,…)] [ copy_out (…)] [ copy_inout (…)] }  
```

Support for multiple implementations of a task

To compute dependences

```
#pragma omp task [ input (…)] [ output (…)] [ inout (…)] [ concurrent (…)] [ commutative (…)] [priority(…)]
   [ label(…)]

   { function or code block }  
```

To give a name

To set priorities to tasks

```
#pragma omp taskwait [on (…)] [noflush]

Wait for sons or specific data availability
```

Relax consistency to main program

To relax dependence order allowing concurrent execution of tasks

To relax dependence order allowing change of order of execution of commutative tasks
OmpSs: new directives

#pragma omp task [ in (...)][ out (...)][ inout (...)][ concurrent (...)][ commutative (...)][priority(...)]
{ function or code block }

- Alternative syntax towards new OpenMP dependence specification
- To relax dependence order allowing concurrent execution of tasks
- To relax dependence order allowing change of order of execution of commutative tasks
- To set priorities to tasks
OpenMP: Directives

#pragma omp task [ depend (in: …)] [ depend(out:…)] [ depend(inout:…)]
{ function or code block }

Direct contribution of BSC to OpenMP promoting dependences and heterogeneity clauses
Main element: tasks

**Task**
- Computation unit. Amount of work (granularity) may vary in a wide range (μsecs to msecs or even seconds), may depend on input arguments,…
- Once started can execute to completion independent of other tasks
- Can be declared inlined or outlined

**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
- **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.
Main element: inlined tasks

Pragmas inlined
- Applies to a statement
- The compiler outlines the statement (as in OpenMP)

```c
int main ( )
{
    int X[100];

    #pragma omp task
    for (int i = 0; i < 100; i++) X[i]=i;
    #pragma omp taskwait

    ...
}
```
Pragmas inlined
  - Standard OpenMP clauses private, firstprivate, ... can be used

```c
int main ( )
{
    int X[100];

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

```c
int main ( )
{
    int X[100];

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

- Clause label can be used to give a name
  - Useful in traces

```c
int main ( )
{
    int X[100];

    #pragma omp task label (foo)
    for (int i =0; i< 100; i++) X[i]=i;
    #pragma omp taskwait
    ...
}
```
Main element: outlined tasks

Pragmas outlined: attached to function definition

- All function invocations become a task

```c
#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) ;
    #pragma omp taskwait
    ...
}
```
Main element: outlined tasks

Pragmas attached to function definition

- The semantic is capture value
  - For scalars is equivalent to firstprivate
  - For pointers, the address is captured

```c
#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);
    #pragma omp taskwait
    ...
}
#pragma omp taskwait

- Suspends the current task until all children tasks are completed

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

    #pragma omp taskwait
}
```

Without taskwait the subroutine will return immediately after spawning the tasks allowing the calling function to continue spawning tasks.
Clauses that express data direction:
- **in**
- **out**
- **inout**

Dependences computed at runtime taking into account these clauses

```c
#pragma omp task output(x)
x = 5; //1
#pragma omp task input(x)
printf("%d\n", x); //2
#pragma omp task inout(x)
x++; //3
#pragma omp task input(x)
printf("%d\n", x); //4
```

Diagram showing the execution order and antidependence.
Synchronization

#pragma taskwait on (expression)

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

```c
#pragma omp 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)
printf ("result F = %.f\n", F[0][0]);

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

#pragma omp taskwait
printf ("result C = %.f\n", C[0][0]);
}
```
Task directive: array regions

**Indicating as input/output/inout subregions of a larger structure:**

- **input (A[i])**
  - the input argument is element $i$ of $A$

**Indicating an array section:**

- **input ([BS]A)**
  - the input argument is a block of size BS from address $A$

- **input (A[i];BS)**
  - the input argument is a block of size BS from address &A[i]
  - the lower bound can be omitted (default is 0)
  - the upper bound can be omitted if size is known (default is $N-1$, being $N$ the size)

- **input (A[i:j])**
  - the input argument is a block from element $A[i]$ to element $A[j]$ (included)
  - $A[i:i+BS-1]$ equivalent to $A[i; BS]$
Examples dependency clauses, array sections

```c
int a[N];
#pragma omp task input(a)
```

```c
int a[N];
#pragma omp task input(a[0:N-1])
//whole array used to compute dependences
```

```c
int a[N];
#pragma omp task input([N]a)
//whole array used to compute dependences
```

```c
int a[N];
#pragma omp task input(a[0;N])
//whole array used to compute dependences
```

```c
int a[N];
#pragma omp task input(a[0:3])
//first 4 elements of the array used to compute dependences
```

```c
int a[N];
#pragma omp task input(a[0;4])
//first 4 elements of the array used to compute dependences
```
Examples dependency clauses, array sections (multidimensions)

```c
int a[N][M];
#pragma omp task input(a[0:N-1][0:M-1])
// whole matrix used to compute dependences
```

```c
int a[N][M];
#pragma omp task input(a[2:3][3:4])
// 2 x 2 subblock of a at a[2][3]
```

```c
int a[N][M];
#pragma omp task input(a[2;2][3;2])
// 2 x 2 subblock of a at a[2][3]
```

```c
int a[N][M];
#pragma omp task input(a[2:3][0:M-1])
// rows 2 and 3
```

```c
int a[N][M];
#pragma omp task input(a[2;2][0;M])
// rows 2 and 3
```
OmpSs examples
Examples dependency clauses, array sections

```
for (int j; j<N; j+=BS){
    actual_size = (N - j > BS ? BS : N-j);
    #pragma omp task input (vec[j:actual_size]) inout(results) firstprivate(actual_size,j)
    for (int count = 0; count < actual_size; count++)
        results += vec [j+count] ;
}
```
Examples dependency clauses, array sections

```c
#pragma omp task input ([n]vec) inout (*results)
void sum_task ( int *vec , int n , int *results);

void main(){
  int actual_size;
  for (int j; j<N; j+=BS){
    actual_size = (N- j) > BS ? BS : N-j);
    sum_task (&vec[j], actual_size, &total);
  }
}
```
Examples dependency clauses, array sections

```c
void compute(unsigned long NB, unsigned long DIM,
             double *A[DIM][DIM], double *B[DIM][DIM], double *C[DIM][DIM])
{
    unsigned i, j, k;

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

```c
#pragma omp task input([NB][NB]A, [NB][NB]B) inout([NB][NB]C)
void matmul(double *A, double *B, double *C,
            unsigned long NB)
{
    int i, j, k;

    for (i = 0; i < NB; i++)
        for (j = 0; j < NB; j++)
            for (k = 0; k < NB; k++)
                C[i][j] += A[i*NB+k] * B[k*NB+j];
}
```
```c
#pragma omp task input (...) output (...) concurrent (var)
```

- Less-restrictive than regular data dependence
  - Concurrent tasks can run in parallel
    - Enables the scheduler to change the order of execution of the tasks, or even execute them concurrently
      - Alternatively the tasks would be executed sequentially due to the inout accesses to the variable in the concurrent clause
    - Dependencies with other tasks will be handled normally
      - Any access input or inout to var will imply to wait for all previous concurrent tasks
  - The task may require additional synchronization
    - i.e., atomic accesses
    - Programmer responsibility: with pragma atomic, mutex, ...
Concurrent

```c
#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=0; j<N; j+=BS) sum_task (&vec[j], BS, &total);

  #pragma omp task input (total)
  printf ("TOTAL is %d\n", total);
}
```
Commutative

```
#pragma omp task input ( ...) output ( ...) commutative(var)
```

Less-restrictive than regular data dependence

- Denoting that tasks can execute in any order but not concurrently
  Enables the scheduler to change the order of execution of the tasks, but without executing them concurrently
    - Alternatively the tasks would be executed sequentially in the order of instantiation due to the inout accesses to the variable in the commutative clause
  - Dependences with other tasks will be handled normally
    - Any access input or inout to `var` will imply to wait for all previous `commutative` tasks
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];
    *results += local_sum;
}

void main(){
    for (int j=0; j<N; j+=BS) sum_task (&vec[j], BS, &total);
    #pragma omp task input (total)
    printf ("TOTAL is %d\n", total);
}
Differences between concurrent and commutative

Tasks timeline: views at same time scale

Histogram of tasks duration: at same control scale

In this case, concurrent is more efficient

... but tasks have more duration and variability
Hierarchical task graph

- **Nesting**
  - Tasks can generate tasks themselves

- **Hierarchical task dependences**
  - Dependences only checked between siblings
    - Several task graphs
    - Hierarchical
      - There is no implicit taskwait at the end of a task waiting for its children
  - Different level tasks share the same resources
    - When ready, queued in the same queues
    - Currently, no priority differences between tasks and its children
Hierarchical task graph

```c
#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]A, [N]B) inout([N]C)
void dgemm(float (*A)[N], float (*B)[N], float (*C)[N]){
  int i, j, k;
  int NB = N/BS;

  for (i=0; i< N; i+=BS)
    for (j=0; j< N; j+=BS)
      for (k=0; k< N; k+=BS)
        block_dgem(&A[i][k*BS], &B[k][j*BS], &C[i][j*BS]);

#pragma omp taskwait
}

main() {
  ...
  dgemm(A,B,C);
  dgemm(D,E,F);
  #pragma omp taskwait
}
```
Example sentinels

```c
#include <omp.h>

#pragma omp task output (*sentinel)
void foo ( .... , int *sentinel){ // used to force dependences under complex structures
  (graphs, ... )

  ...
}

#pragma omp task input (*sentinel)
void bar ( .... , int *sentinel){

  ...
}
main () {
  int sentinel;

  foo (... , &sentinel);
  bar (... , &sentinel)
}
```

- Mechanism to handle complex dependences
- When difficult to specify proper input/output clauses
- To be avoided if possible
- The use of an element or group of elements as sentinels to represent a larger data-structure is valid
- However might made code non-portable to heterogeneous platforms if copy_in/out clauses cannot properly specify the address space that should be accessible in the devices
OmpSs + heterogeneity
#pragma omp target [ clauses ]

- Specifies that the code after it is for a specific device (or devices)
- The compiler parses the specific syntax of that device and hands the code over to the appropriate back end compiler
- Currently supported devices:
  - smp: default device. Back end compiler to generate code can be gcc, icc, xlc,….
  - opencl: OpenCL code will be used from the indicated file, and handed over the runtime system at execution time for compilation and execution
  - cuda: CUDA code is separated to a temporary file and handed over to nvcc for code generation
Heterogeneity: the copy clauses

#pragma omp target [ clauses ]

- Some devices (opencl, cuda) have their private physical address space.
  - The copy_in, copy_out, an copy_inout clauses have to be used to specify what data has to be maintained consistent between the original address space of the program and the address space of the device.
  - The copy_deps is a shorthand to specify that for each input/output/inout declaration, an equivalent copy_in/out/inout is used.
- Tasks on the original program device (smp) also have to specify copy clauses to ensure consistency for those arguments referenced in some other device.
- The default taskwait semantic is to ensure consistency of all the data in the original program address space.
**ndrange**: provides the configuration for the OpenCL/CUDA kernel

\[ \text{ndrange} \left( \text{ndim}, \{\text{global/grid}\}_\text{array}, \{\text{local/block}\}_\text{array} \right) \]
\[ \text{ndrange} \left( \text{ndim}, \{\text{global/grid}\}_\text{dim1}, \ldots \{\text{local/block}\}_\text{dim1}, \ldots \right) \]

- 1 to 3 dimensions are valid
- Values can be provided through
- 1-, 2-, 3-elements arrays (global, local)
- Two lists of 1, 2, or 3 elements, matching the number of dimensions
- Values can be function arguments or globally accessible variables
Example OmpSs@OpenCL

```c
#pragma omp task input ([n]x) inout ([n]y)
void saxpy (int n, float a, float *x, float *y)
{
    for (int i=0; i<n; i++)
        y[i] = a * X[i] + y[i];
}

int main (int argc, char *argv[])
{
    float a, x[1024], y[1024];
    // initialize a, x and y
    saxpy (1024, a, x, y);
    #pragma omp taskwait
    printf ("%f", y[0]);
    return 0;
}
```

```c
#pragma omp task input ([n]x) inout ([n]y)
#pragma omp target device (opencl) 
    ndrange (1, n, 128) copy_deps
__kernel void saxpy (int n, float a, __global 
    float *x, __global float *y)
{
    int i = get_global_id(0);
    if (i<n)
        y[i] = a * X[i] + y[i];
}

int main (int argc, char *argv[])
{
    float a, x[1024], y[1024];
    // initialize a, x and y
    saxpy (1024, a, x, y);
    #pragma omp taskwait
    printf ("%f", y[0]);
    return 0;
}
```
```c
#define BLOCK_SIZE 16
__constant int BL_SIZE = BLOCK_SIZE;

#pragma omp target device(opencl)
#pragma omp task input([NB*NB]A, [NB*NB]B)
__kernel void Muld(__global REAL* A, __global REAL* B, int wA, int wB, __global REAL* C, int NB) {
    // Block index, Thread index
    int bx = get_group_id(0); int by = get_group_id(1);
    int tx = get_local_id(0); int ty = get_local_id(1);

    // Indexes of the first/last sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;
    int aEnd = aBegin + wA - 1;

    // Step size used to iterate through the sub-matrices of A
    int aStep = BLOCK_SIZE;
    ...
```

```c
#include "matmul_auxiliar_header.h"  // defines BLOCK_SIZE

// Device multiplication function
// Compute C = A * B
// wA is the width of A
// wB is the width of B
__kernel void Muld(__global REAL* A, __global REAL* B, int wA, int wB, __global REAL* C, int NB) {
    // Block index, Thread index
    int bx = get_group_id(0); int by = get_group_id(1);
    int tx = get_local_id(0); int ty = get_local_id(1);

    // Indexes of the first/last sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;
    int aEnd = aBegin + wA - 1;

    // Step size used to iterate through the sub-matrices of A
    int aStep = BLOCK_SIZE;
    ...
```
#pragma omp target device(cuda)
copy_deps
ndrange(2,NB,NB,16,16)

#pragma omp task inout([NB*NB]C) in([NB*NB]A,[NB*NB]B)
__global__ void Muld(REAL* A, REAL* B, int wA, int wB, REAL* C, int NB);

OmpSs@CUDA
@CUDA

matmul(NB, NB, DIM, DIM, NB, NB, tileA, tileB, tileC)

#include "matmul_auxiliar_header.h"

// Thread block size
#define BLOCK_SIZE 16

// Device multiplication function called by Mul()
// Compute C = A * B
//  wA is the width of A
//  wB is the width of B
__global__ void Muld(REAL* A, REAL* B, int wA, int wB, REAL* C, int NB) {
    // Block index
    int bx = blockIdx.x; int by = blockIdx.y;
    // Thread index
    int tx = threadIdx.x; int ty = threadIdx.y;
    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;
    // Index of the last sub-matrix of A processed by the block
    int aEnd = aBegin + wA - 1;
    // Step size used to iterate through the sub-matrices of A
    int aStep = BLOCK_SIZE;
    ...
OmpSs compiler and runtime
Mercurium 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

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

[Diagram showing the components and interfaces of the runtime structure]
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: 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
Runtime structure behaviour: clusters

**One runtime instance per node**
- One master image
- N-1 slave 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

```
#pragma omp task { ... }
```
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 environment and further examples
Compiling

frontend --ompss -c bin.c

Linking

frontend --ompss -o bin bin.o

where frontend is one of:

<table>
<thead>
<tr>
<th>frontend</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>mcc</td>
<td>C</td>
</tr>
<tr>
<td>mcxx</td>
<td>C++</td>
</tr>
<tr>
<td>mnvcc</td>
<td>CUDA &amp; C</td>
</tr>
<tr>
<td>mnvcxx</td>
<td>CUDA &amp; C++</td>
</tr>
<tr>
<td>mfc</td>
<td>Fortran</td>
</tr>
</tbody>
</table>
### Compiling

**Compatibility flags:**
- `-l`, `-g`, `-L`, `-I`, `-E`, `-D`, `-W`

**Other compilation flags:**

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-k</code></td>
<td>Keep intermediate files</td>
</tr>
<tr>
<td><code>--debug</code></td>
<td>Use Nanos++ debug version</td>
</tr>
<tr>
<td><code>--instrumentation</code></td>
<td>Use Nanos++ instrumentation version</td>
</tr>
<tr>
<td><code>--version</code></td>
<td>Show Mercurium version number</td>
</tr>
<tr>
<td><code>--verbose</code></td>
<td>Enable Mercurium verbose output</td>
</tr>
<tr>
<td><code>--Wp,flags</code></td>
<td>Pass flags to preprocessor (comma separated)</td>
</tr>
<tr>
<td><code>--Wn,flags</code></td>
<td>Pass flags to native compiler (comma separated)</td>
</tr>
<tr>
<td><code>--Wl,flags</code></td>
<td>Pass flags to linker (comma separated)</td>
</tr>
<tr>
<td><code>--help</code></td>
<td>To see many more options :-</td>
</tr>
</tbody>
</table>
No LD_LIBRARY_PATH or LD_PRELOAD needed
./bin

Adjust number of threads with OMP_NUM_THREADS
OMP_NUM_THREADS=4 ./bin
Other options can be passed to the Nanos++ runtime via `NX_ARGS`

```
NX_ARGS="options" ./bin
```

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--schedule=name</td>
<td>Use name task scheduler</td>
</tr>
<tr>
<td>--throttle=name</td>
<td>Use name throttle-policy</td>
</tr>
<tr>
<td>--throttle-limit=limit</td>
<td>Limit of the throttle-policy (exact meaning depends on the policy)</td>
</tr>
<tr>
<td>--instrumentation=name</td>
<td>Use name instrumentation module</td>
</tr>
<tr>
<td>--disable-yield</td>
<td>Nanos++ won’t yield threads when idle</td>
</tr>
<tr>
<td>--spins=number</td>
<td>Number of spin loops when idle</td>
</tr>
<tr>
<td>--disable-binding</td>
<td>Nanos++ won’t bind threads to CPUs</td>
</tr>
<tr>
<td>--binding-start=cpu</td>
<td>First CPU where a thread will be bound</td>
</tr>
<tr>
<td>--binding-stride=number</td>
<td>Stride between bound CPUs</td>
</tr>
</tbody>
</table>
Nanox helper

Nanos++ utility to

- list available modules:
  
nanox --list-modules

- list available options:
  
nanox --help
Tracing

Compile and link with --instrument

```bash
mcc --ompss --instrument -c bin.c
mcc -o bin --ompss --instrument bin.o
```

When executing specify which instrumentation module to use:

```bash
NX_INSTRUMENTATION=extrae ./bin
```

Will generate trace files in executing directory
- 3 files: prv, pcf, rows
- Use paraver to analyze
Reporting problems

- Compiler problems
  - http://pm.bsc.es/projects/mcxx/newticket

- Runtime problems
  - http://pm.bsc.es/projects/nanox/newticket

- Support mail
  - pm-tools@bsc.es

- Please include snapshot of the problem
Programming methodology

- Correct sequential program
- Incremental taskification
  - Test every individual task with forced sequential in-order execution
    - 1 thread, scheduler = FIFO, throtle=1
- Single thread out-of-order execution
- Increment number of threads
  - Use taskwaits to force certain levels of serialization
Visualizing Paraver tracefiles

Set of Paraver configuration files ready for OmpSs. Organized in directories

- **Tasks**: related to application tasks
- Runtime, nanox-configs: related to OmpSs runtime internals
- **Graph_and_scheduling**: related to task-graph and task scheduling
- DataMgmgt: related to data management
- CUDA: specific to GPU
2dp_tasks.cfg

Tasks’ profile

control window: timeline where each color represent the task been executed by each thread

light blue: not executing tasks

gradient color, indicates given estadístico: i.e., number of tasks instances

different colours represent different task type
Tasks duration histogram

3dh_duration_task.cfg

gradient color, indicates given estadístic: i.e., number of tasks instances

threads

time intervals
Tasks duration histogram

3dh_duration_task.cgf

control window: task duration

gradient color, indicates given statistic: i.e., number of tasks instances

threads
Tasks duration histogram

3dh_duration_task.cfg

3D window: task type

gradient color, indicates given statistic: i.e., number of tasks instances

time intervals

threads
Tasks duration histogram

3dh_duration_task.cfg

3D window: task type

gradient color, indicates given statistic: i.e., number of tasks instances

Threads

Chooser: task type

Statistics

<table>
<thead>
<tr>
<th>Type</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statistic</td>
<td>Time</td>
</tr>
<tr>
<td>Minimum Gradient</td>
<td>833.828</td>
</tr>
<tr>
<td>Maximum Gradient</td>
<td>112342.349</td>
</tr>
</tbody>
</table>

Data

3D

<table>
<thead>
<tr>
<th>3rd Window</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum</td>
<td>1</td>
</tr>
<tr>
<td>Maximum</td>
<td>5</td>
</tr>
<tr>
<td>Delta</td>
<td>1</td>
</tr>
<tr>
<td>Plane</td>
<td>Task 'triad_task'</td>
</tr>
</tbody>
</table>

BSC
Barcelona Supercomputing Center
Civetra Nacional de Supercomputación
Threads state profile

- **2dp_threads_state.cfg**

Control window: timeline where each color represents the runtime state of each thread.
Generating the task graph

- Compile with --instrument
- export NX_INSTRUMENTATION=graph
- export OMP_NUM_THREADS=1
Accessing non-contiguous or partially overlapped regions

Sorting arrays
- Divide by \( \frac{1}{4} \)
- Sort
- Merge

Each small segment is sorted

Merge each set of segments
Accessing non-contiguous or partially overlapped regions

Why is the regions-aware dependences plug-in needed?

- Regular dependence checking uses first element as representative (size is not considered)
- Segment starting at address A[i] with length L/4 will be considered the same as A[i] with length L
- Dependences between A[i] with length L and A[i+L/4] with length L/4 will not be detected

All these is fixed with the regions plugin

Two different implementations:
- **NX_DEPS=** regions
- **NX_DEPS=** perfect-regions
void multisort(long n, T data[], T tmp[]) {
    if (n >= MIN_SORT_SIZE*4L) {
        // Recursive decomposition
        #pragma omp task inout (data[0; n/4L]) firstprivate(n)
        multisort(n/4L, &data[0], &tmp[0]);
        #pragma omp task inout(data[n/4L; n/4L]) firstprivate(n)
        multisort(n/4L, &data[n/4L], &tmp[n/4L]);
        #pragma omp task inout (data[n/2L; n/4L]) firstprivate(n)
        multisort(n/4L, &data[n/2L], &tmp[n/2L]);
        #pragma omp task inout (data[3L*n/4L; n/4L]) firstprivate(n)
        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);

        #pragma omp task input (data[0; n/4L], data[n/4L; n/4L]) output (tmp[0; n/2L])
            firstprivate(n)
        merge_rec(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
        #pragma omp task input (data[n/2L; n/4L], data[3L*n/4L; n/4L])
            output (tmp[n/2L; n/2L]) firstprivate (n)
        merge_rec(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);

        #pragma omp task input (tmp[0; n/2L], tmp[n/2L; n/2L]) output (data[0; n])
            firstprivate (n)
        merge_rec(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
    } else basicsort(n, data);
}
Accessing non-contiguous or partially overlapped regions

T *data = malloc(N*sizeof(T));
T *tmp = malloc(N*sizeof(T));

posix_memalign ((void**)&data, N*sizeof(T), N*sizeof(T));
posix_memalign ((void**)&tmp, N*sizeof(T), N*sizeof(T));

...  
  multisort(N, data, tmp);
#pragma omp taskwait

Current implementation requires alignment of data for efficient data-dependence management
Using task versions

```c
#pragma omp target device (smp) copy_deps
#pragma omp task input([NB][NB]A, [NB][NB]B) inout([NB][NB]C)

void matmul(double *A, double *B, double *C, unsigned long NB)
{
    int i, j, k, I;
    double tmp;
    for (i = 0; i < NB; i++) {
        I = i*NB;
        for (j = 0; j < NB; j++) {
            tmp = C[I+j];
            for (k = 0; k < NB; k++)
                tmp += A[I+k]*B[k*NB+j];
            C[I+j]=tmp;
        }
    }
}

#pragma omp target device (smp) implements (matmul) copy_deps
#pragma omp task input([NB][NB]A, [NB][NB]B) inout([NB][NB]C)

void matmul_mkl(double *A, double *B, double *C, unsigned long NB)
{
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, NB, NB, NB, 1.0,
               (double *)A, NB, (double *)B, NB, 1.0, (double *)C, NB);
}```
Using task versions

```c
void compute(struct timeval *start, struct timeval *stop, unsigned long NB, unsigned long DIM, double *A[DIM][DIM], double *B[DIM][DIM], double *C[DIM][DIM])
{
    unsigned i, j, k;

    gettimeofday(start, NULL);

    for (i = 0; i < DIM; i++)
        for (j = 0; j < DIM; j++)
            for (k = 0; k < DIM; k++)
                matmul ((double *)A[i][k], (double *)B[k][j], (double *)C[i][j], NB);

#pragma omp taskwait
    gettimeofday(stop, NULL);
}
```
Using task versions

- Use of specific scheduling:
  - export NX_SCHEDULE=versioning

- Tries each version a given number of times and automatically will choose the best version
Using socket aware scheduling

- Assign top level tasks (depth 1) to a NUMA node set by the user before task creation
  - nested tasks will run in the same node as their parent.

- nanos_current_socket API function must be called before instantiation of tasks to set the NUMA node the task will be assigned to.

- Queues sorted by priority with as many queues as NUMA nodes specified (see num-sockets parameter).
Using socket aware scheduling

Example: stream

```c
#pragma omp 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 < BSIZE; j++)
        c[j] = a[j]+b[j];
}

void tuned_STREAM_Add()
{
    int j;
    for (j=0; j<N; j+=BSIZE){
        nanos_current_socket( ( j/((int)BSIZE) ) % 2 );
        add_task(&a[j], &b[j], &c[j], BSIZE);
    }
}
```
Using socket aware scheduling

**Usage:**
- export NX_SCHEDULE=socket

**If using less than N threads, being N the cores in a socket:**

**I.E., for a socket of 6 cores:**
- export NX_ARGS="--binding-stride 6"
Using socket aware scheduling

Differences between the use of socket aware scheduling in the stream example:

Non-Socket-aware

Socket-aware
for (k = 0; k < nt; k++) {
    for (i = 0; i < k; i++) {
        #pragma omp task input([ts*ts]Ah[i*nt + k]) inout([ts*ts]Ah[k*nt + k]) \
        priority( (nt-i)+10 ) firstprivate (i, k, nt, ts)
        syrk_tile (Ah[i*nt + k], Ah[k*nt + k], ts, region)
    }
    // Diagonal Block factorization and panel permutations
    #pragma omp task inout([ts*ts]Ah[k*nt + k]) \
    priority( 100000 ) firstprivate (k, ts, nt)
    potr_tile(Ah[k*nt + k], ts, region)

    // update trailing matrix
    for (i = k + 1; i < nt; i++) {
        for (j = 0; j < k; j++) {
            #pragma omp task input ([ts*ts]Ah[j*nt+i], [ts*ts]Ah[j*nt+k]) \
            inout ( [ts*ts]Ah[k*nt+i] ) firstprivate (i, j, k, ts, nt)
            gemm_tile (Ah[j*nt + i], Ah[j*nt + k], Ah[k*nt + i], ts, region)
        }
        #pragma omp task input([ts*ts]Ah[k*nt + k]) inout([ts*ts]Ah[k*nt + i]) \
        priority( (nt-i)+10 ) firstprivate (i, k, ts, nt)
        trsm_tile (Ah[k*nt + k], Ah[k*nt + i], ts, region)
    }
}
#pragma omp taskwait
Giving hints to the compiler: priorities

Potrf: Maximum priority

trsm: priority \( (nt - i) + 10 \)

syrk: priority \( (nt - i) + 10 \)

gemm: no priority
Two policies available:

- Priority scheduler
  - Tasks are scheduled based on the assigned priority.
  - The priority is a number $\geq 0$. Given two tasks with priority $A$ and $B$, where $A > B$, the task with priority $A$ will be executed earlier than the one with $B$.
  - When a task $T$ with priority $A$ creates a task $T_c$ that was given priority $B$ by the user, the priority of $T_c$ will be added to that of its parent. Thus, the priority of $T_c$ will be $A + B$.

- Smart Priority scheduler
  - Similar to the Priority scheduler, but also propagates the priority to the immediate preceding tasks.

Using the schedulers:

- `export NX_SCHEDULE = priority`
- `export NX_SCHEDULE = smartpriority`
Conclusions

**StarSs**
- Asynchronous Task-based programming model
- Key aspect: data dependence detection which avoid global synchronization
- Support for heterogeneity increasing portability

**Encompases a complete programming environment**
- StarSs programming model
- Tareador: finding tasks
- Paraver: Performance analysis
- DLB: dynamic load balancing
- Temanejo: debugger (under development at HLRS)

**Support for MPI**
- Overlap off computation and communication

**Fully open, available at: pm.bsc.es/ompss**
Thank you!

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