Tutorial OmpSs: single node programming

PATC course
Parallel Programming Workshop

Rosa M Badia, Xavier Martorell
Tutorial OmpSs

Agenda

10:00 – 11:00  Tasking in OpenMP 3.0 and 4.0  60 min

11:00 – 11:15  Coffee break  15 min

11:15 – 12:15  Introduction to OmpSs programming model  60 min
• Introduction to StarSs
• OmpSs syntax
• Simple examples
• Development methodology and infrastructure

12:15 – 12:45  Practical: heat equation example and divide-and-conquer (part I)  30 min

12:45 – 14:00  Lunch  75 min

14:00 – 15:00  Practical: heat equation example and divide-and-conquer (part I)  90 min

15:00 – 15:30  Programming using a hybrid MPI/OmpSs approach  15 min

15:30 – 17:00  Practical: heat equation example and matrix-multiply  105 min
Outline

- Introduction to StarSs
- OmpSs overview
- OmpSs syntax
- OmpSs environment
- Hands-on

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

- Basic concept: write sequential on a flat single address space + directionality annotations
  - Order IS defined
  - Dependence and data access related **information** (NOT specification) in a single mechanism
  - Think global, specify local
  - Intelligent runtime
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 in ([TS][TS]T) inout ([TS][TS]B)
void strsm (float *T, float *B);
#pragma omp task in ([TS][TS]A,[TS][TS]B) inout ([TS][TS]C)
void sgemm (float *A, float *B, float *C);
#pragma omp task in ([TS][TS]A) inout ([TS][TS]C)
void ssyrk (float *A, float *C);
```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[k*NT+j], A[j*NT+i]);
            ssyrk (A[k*NT+i], A[i*NT+i]);
        }
    }
}
```
OmpSs syntax
OmpSs execution model and memory model

- Inlined pragmas
- Outlined pragmas
- Array sections
- Concurrent
- Commutative
- Nesting
- Sentinels
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
Sequential control flow
  - Defines a single address space
  - Executes sequential code that
    • Can spawn/instantiate tasks than will be executed sometime in the future
    • Can stall/wait for tasks

Tasks annotated with directionality clauses
  - Input, output, inout
  - Used to build dependences between tasks and for main to wait for data to be produced
  - Can be used for memory management functionalities (replication, locality, movement,...)
**Execution Model**

- **Thread-pool model**
  - OpenMP parallel not required
- **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 and target platform, 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
OmpSs: Directives

- **Task implementation for a GPU device**: The compiler parses CUDA/OpenCL kernel invocation syntax.

- **Support for multiple implementations of a task**:

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

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

- **To compute dependences**:

  ```
  #pragma omp task [ in (...)] [ out (...) ] [ inout (...)] [ concurrent (...) ] [ commutative (...) ] [ priority(…) ]\n  [label(tasklabel)]

  { function or code block }  
  ```

- **Ask the runtime to ensure data is accessible in the address space of the device**:

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

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

- **To set priorities to tasks**:

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

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

- **To relax dependence order allowing concurrent execution of tasks**:

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

- **To relax dependence order allowing change of order of execution of commutative tasks**:

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

- **Wait for sons or specific data availability**:

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

- **Relax consistency to main program**:

  ```
  #pragma omp taskwait [on (...)] [noflush]
  ```
OpenMP: Directives

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

Direct contribution of BSC at 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.
Inlined and outlined tasks

**Pragmas inlined**

- Pragma applies to immediately following statement
- The compiler outlines the statement (as in OpenMP)

**Pragmas outlined:**

- Attached to function declaration
  - All function invocations become a task
  - The programmer gives a name, this enables later to provide several implementations
Main element: inlined tasks

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

**Pragmas parallel and single not required**

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

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

    #pragma omp taskwait

    ...
}
```
Defining dependences

Clauses that express data direction:
- in
- out
- inout

Dependences computed at runtime taking into account these clauses

```c
#pragma omp task out(x)
x = 5;        //1
#pragma omp task in(x)
printf("%d\n", x );    //2
#pragma omp task inout(x)
x++;          //3
#pragma omp task in(x)
printf ("%d\n", x );    //4
```
Graph automatically generated

User functions:
- smp_ol_main_0
- smp_ol_main_1
- smp_ol_main_2
- smp_ol_main_3

Dependence type:
- Input dependence
- Output dependence
Partial control flow synchronization

#pragma taskwait on ( expression )

- Expressions allowed are the same as for the directionality clauses
- Stalls the encountering control flow until the data is available

```c
double A[N][N], B[N][N], C[N][N], D[N][N], E[N][N],
     F[N][N], G[N][N], H[N][N], I[N][N], J[N][N];

main() {
    #pragma omp task in(A, B) inout(C)
    dgemm(A, B, C); //1
    #pragma omp task in(D, E) inout(F)
    dgemm(D, E, F); //2
    #pragma omp task in(C, F) inout(G)
    dgemm(C, F, G); //3
    #pragma omp task in(A, D) inout(H)
    dgemm(A, D, H); //4
    #pragma omp task in(C, H) inout(I)
    dgemm(C, H, I); //5

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

    #pragma omp task in(C, H) inout(I)
    dgemm(H, G, J); //6

    #pragma omp taskwait
    printf ("result J = %f\n", J[0][0]);
}
```
Pragmas outlined: attached to function definition

- All function invocations become a task
- The programmer gives a name, this enables later to provide several implementations

```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
    ...
}
```
Defining dependences for outlined tasks

Clauses that express data direction:
- Input, output, inout
- The argument is an lvalue expression based on data visible at the point of declaration (global variables and arguments)
- The object pointed by the lvalue expression will be used to compute dependences.

```c
#pragma omp task out(*px)
void set (int *px, int v) {*px = v;}

#pragma omp task inout(*px)
void incr (int *px) {(*px)++;

#pragma omp task in(x)
void do_print (int x) {
    printf(“from do_print %d\n”, x );
}
```

```c
set(&x,5);      //1
do_print(x);   //2
incr(&x);       //3
do_print(x);   //4
#pragma omp taskwait
```

antidependence
Mixing inlined and outlined tasks

```c
#pragma omp task input (x)
void do_print (int x) {
    printf("from do_print \%d\n", x);
}

int main() {
    int x;
    x=3;
    #pragma omp task out (x)
    x = 5;           //1
    #pragma omp task in( x )
    printf("from main \%d\n", x); //2
    do_print(x);       //3
    #pragma omp task inout( x )
    x++;          //4
    #pragma omp task in( x )
    printf ("from main \%d\n", x); //5
}
```
Partial control flow 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 in([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:**
  
  in (A[i])
  → the input argument is element $i$ of $A$

- **Indicating an array section:**
  
  In ([BS]A)
  → the input argument is a block of size BS from address $A$

  in (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)

  In (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

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

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

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

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

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

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

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

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

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

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

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

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

void main(){
  int actual_size;
  for (int j=0; j<N; j+=BS){
    actual_size = (N- j> BS ? BS : N-j);
    sum_task (&vec[j], actual_size, &total);
  }
}
```

`BS` represents the block size, and `vec` is the array section. The diagram illustrates the dependency and array sections.
for (int j=0; j<N; j+=BS){
    actual_size = (N- j> BS ? BS : N-j);
    #pragma omp task in (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([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];
}

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);
}
```
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];
}

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++)
                #pragma omp task input([NB][NB]A[i][k], [NB][NB]B[k][j]) inout([NB][NB]C[i][j])
firstprivate (i, j, k)
                matmul (A[i][k], B[k][j], C[i][j], NB);
}
#pragma omp task in (...) out (...) 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
  - Dependences 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

#pragma omp task in([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 in (total)
    printf ("TOTAL is %d\n", total);
    #pragma omp taskwait
}
#pragma omp task in ( ...) out (...) 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
Commutative

```c
#pragma omp task in ([n]vec ) commutative(*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] ;
    *results += local_sum;
}

void main(){
    for (int j=0; j<N; j+=BS) sum_task (&vec[j], BS, &total);
    #pragma omp task in(total)
    printf ("TOTAL is %d\n", total);
    #pragma omp taskwait
}```
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
Nesting inlined tasks

int Y[4]={1,2,3,4}

int main( )
{
    int X[4]={5,6,7,8};

    for (int i=0; i<2; i++) {
        #pragma omp task out(Y[i]) firstprivate(i,X)
        {
            for (int j=0 ; j<3; j++) {
                #pragma omp task inout(X[j])
                X[j]=f(X[j], j);
                #pragma omp task in (X[j]) inout (Y[i])
                Y[i] +=g(X[j]);
            }
            #pragma omp taskwait
        }
        #pragma omp task wait
    }
    #pragma omp task inout(Y[0;2])
    for (int i=0; i<2; i++) Y[i] += h(Y[i]);
    #pragma omp task inout (v) inout(Y[3])
    for (int i=1; i<N; i++) Y[3]=h(Y[3]);

    #pragma omp task wait
}
```c
#pragma omp task in([BS][BS] A, [BS][BS] B) inout([BS][BS] C)
void block_dgemm(float *A, float *B, float *C);

#pragma omp task in([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_dgemm(&A[i][k*BS], &B[k][j*BS], &C[i][j*BS]);
}

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

- Directionality not required for all arguments
- May even be used with variables not accessed in that way or even used
  - used to force dependences under complex structures (graphs, ... )

```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++)
                #pragma omp task in(A[i][k], B[k][j]) inout(C[i][j])
                matmul (A[i][k], B[k][j], C[i][j], NB);
}
```

Using entry in C matrix of pointers as representative/sentinel for the whole block it points to.

Will build proper dependences between tasks.

Does NOT provide actual information of data access pattern. (see copy clauses)
Example sentinels

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

#pragma omp task in (*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
Thank you!

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