Programming with StarSs

Computer Sciences Research Dept.
BSC

Agenda

• StarSs overview
• OmpSs syntax
• OmpSs environment
• Single node handson
• Hybrid model MPI/OmpSs
• MPI/OmpSs handson

• Slides and single-node source code: /tmp/tutorial_ompss_PATC_single.ppt
• /tmp/tutorial_PATC_singlenode.tar.bz2
• /tmp/StarSs hands_PATC.pdf
• Contact: pm-tools@bsc.es
• Source code available from http://pm.bsc.es/ompss/
The StarSs family of programming models

- Task based. Asynchrony, data-flow.
- Simple linear address space
- Malleable
- Nicely hybridizes (MPI/StarSs)
- Natural support for heterogeneity

Key concepts:
- Sequential program
- Directionality annotations on tasks arguments

Key objectives:
- Same program any platform
- Productive / Incremental programming

The StarSs “Granularities”

Average task Granularity:
- 100 microseconds – 10 milliseconds
- 1 second - 1 day

Address space to compute dependences:
- Memory
- Files, Objects

Language binding:
- C, C++, FORTRAN
- Java, Python
StarSs: history/strategy/versions

Basic SMPSs
- must provide directionality argument
- Contiguous, non partially overlapped
- Renaming
- Several schedulers (priority, locality,...)
- No nesting
- C/Fortran
- MPI/SMPSs opts.

SMPSs regions
- C, No Fortran
- must provide directionality argument
- overlapping & strides
- Reshaping strided accesses
- Priority and locality aware scheduling

OMPSs
- C, C++, Fortran
- OpenMP compatibility (~)
- Contiguous and strided args.
- Separate dependences/transfer
- Inlined/outlined pragmas
- Nesting
- Heterogeneity: SMP/GPU/Cluster
- No renaming.
- Several schedulers: "Simple" locality aware sched,…

Evolving research since 2005

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 runtime to
  - Prefetch
  - Reuse
  - Eliminate antidependences (rename)
  - Replication management
  - Coherency/consistency handled by the runtime

- C, C++, Fortran
- OpenMP compatibility (~)
- Contiguous and strided args.
- Separate dependences/transfer
- Inlined/outlined pragmas
- Nesting
- Heterogeneity: SMP/GPU/Cluster
- No renaming.
- Several schedulers: "Simple" locality aware sched,…
StarSs: data-flow execution of sequential programs

```c
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]);
        }
    }
}
```

### StarSs tasks

- **Task**
  - Computation unit. Amount of work (granularity) may vary in a wide range (microseconds to milliseconds or even seconds), may depend on input arguments,
  - Once started can execute to completion independent of other tasks
- **States:**
  - Instantiated: when task is created by main program. Dependences are computed at the moment of instantiation.
  - 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.
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]);
    
    #pragma omp task
    for ((j=k+1; j<i; j++) {
        #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
Memory Model

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

OmpSs: Directives

- Task implementation for a GPU device
  - The compiler parses CUDA kernel invocation syntax
  - Support for multiple implementations of a task
    - #pragma omp target device ({ smp | cuda })
    - [ implements { function_name }]
    - { copy_deps | [ copy_in ( array_spec ,...)] [ copy_out (...)] [ copy_inout (...)]}
  - Ask the runtime to ensure data is accessible in the address space of the device
    - #pragma omp taskwait [on (...)] [noflush]
  - To compute dependences
    - #pragma omp task [ input (...)] [ output (...)] [ inout (...)] [ concurrent (...)]
    - ( function or code block )
    - To allow concurrent execution of commutative tasks
  - Wait for sons or specific data availability
    - #pragma omp taskwait [on (...)] [noflush]
    - Relax consistency to main program
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;
}
```
Main element: outlined tasks

- 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
  ...
}```
Synchronization

#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.

Defining dependences

• Clauses that express data direction:
  • input
  • output
  • 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
```
Defining dependences

```c
#include <stdio.h>

#define MAX_THREADS 4

void do_print (int *px) {
    printf("from do_print \%d\n", *px);
}

int main() {
    int x;
    x = 3;
    #pragma omp task output(x)
    x = 5;
    printf(\"from main \%d\n\", x);
    #pragma omp task input(x)
    printf(\"from main \%d\n\", x);
    do_print(x);
    #pragma omp task inout(x)
    x++;
    #pragma omp task input(x)
    printf(\"from main \%d\n\", x);
    do_print(&x);
    #pragma omp task inout(x)
    x++;
    #pragma omp task input(x)
    printf(\"from main \%d\n\", x);
}

Output:
(4) from print_do 3
(2) from main 5
(3) from do_print 5
(6) from main 6
```

1. Non-taskified: executed sequentially
2. "value captured at instantiation time"
3. Compiler warning, input clause discarded
Defining dependences

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

#pragma omp task input (x) // compiler warning, input clause discarded
void print_do (int x) {
    printf("from print_do %d
", x );
}

int main()
{
    int x;
    x=3;
    #pragma omp task output( x )
    x = 5;//1
    #pragma omp task input( x )
    printf("from main %d\n", x );//2
do_print(*px);//3
print_do(x);//4
#pragma omp task inout( x )
x++;//5
#pragma omp task input( x )
printf ("from main %d\n", x );//6
}
```

but also:

(2) from main 5
(4) from print_do 3
(3) from do_print 5
(6) from main 6

Defining dependences

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

#pragma omp task input (x) // compiler warning, input clause discarded
void print_do (int x) {
    printf("from print_do %d\n", x );
}

int main()
{
    int x;
    x=3;
    #pragma omp task output( x )
    x = 5;//1
    #pragma omp task input( x )
    printf("from main %d\n", x );//2
do_print(*px);//3
print_do(x);//4
#pragma omp task inout( x )
x++;//5
#pragma omp task input( x )
printf ("from main %d\n", x );//6
}
```

and also:

(2) from main 5
(3) from do_print 5
(4) from print_do 3
(6) from main 6

depending on actual schedule
Defining dependences

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

#define omp task input (x) // compiler warning, input clause discarded
void print_do (int x) {
    printf("from print_do %d\n", x );
}

int main() {
    int x;
    x=3;
    #pragma omp task output( x )
    x = 5;//1
    #pragma omp task input( x )
    printf("from main %d\n", x );//2
do_print(&x);//3
    #pragma omp task inout( x )
    x++;//5
    #pragma omp task input( x )
    printf("from main %d\n", x );//6
do_print(&x);//3
    print_do(x);//4
}
```

and even:

1. (2) from main 5
2. (3) from do_print 5
3. (4) from print_do 5
4. (6) from main 6

• because there is no dependence to 4
• the value of x is captured at instantiation time
• other tasks may alter the value
• in this case, execution of 1 before instantiation of 4

Synchronization

```c
#pragma taskwait on ( expression )
• Expressions allowed are the same as for the dependency clauses
• Blocks the encountering task until the data is available

#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]);
}
#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:
  
  \[ \text{input} (\text{A}[i]) \]
  
  \( \rightarrow \) the input argument is element \( i \) of \( A \)

- Indicating an array section:
  
  \[ \text{input} ([\text{BS}]A) \]
  
  \( \rightarrow \) the input argument is a block of size \( \text{BS} \) from address \( A \)

  \[ \text{input} (\text{A}[i:\text{BS}]) \]
  
  \( \rightarrow \) the input argument is a block of size \( \text{BS} \) from address \&A[i]
  
  \( \rightarrow \) the lower bound can be omitted (default is 0)

  \[ \text{input} (\text{A}[i:]) \]
  
  \( \rightarrow \) the input argument is a block from element \( A[i] \) to element \( A[j] \) (included)
  
  \( \rightarrow \) \( A[i:i+\text{BS}-1] \) equivalent to \( A[i:j] \)

  \( \rightarrow \) the upper bound can be omitted if size is known (default is \( N-1 \), being \( N \) the size)

  \[ \text{input} (\text{A}[i:j]) \]
  
  \( \rightarrow \) the input argument is a block from element \( A[i] \) to element \( A[j] \) (included)

Examples dependency clauses, array sections

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

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

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

```
int a[N];
#pragma omp task input(a[2:3])
// elements 2 and 3 of the array used to compute dependences
```
### Examples dependency clauses, array sections

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

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

```c
int *a;
#pragma omp task input(a[2:3])
//elements 2 and 3 of the array used to compute dependences
```

```c
int *a;
#pragma omp task input(a[2:N-1])
//elements 2 to N-1 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[0:3][0:M])
//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
```
Examples dependency clauses, array sections (multidimensions)

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

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

```c
int *a;
#pragma omp task input([N][M]a)
// whole matrix
```

Examples dependency clauses, array sections

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

```c
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);
    }
}
```

BS

< BS

vec

dynamic size of argument

results
Examples dependency clauses, array sections

```c
void loop1_task(int n, double *pl, double *pr, double *tl, double *tr, double *pn)
{
    double t1, t2;
    int i;
    for (i = 0; i < n; i++)
    {
        pn[0] = t1 * t2;
        pn[1] = t1 * t2;
        pn[2] = t1 * t2;
        pn[3] = t1 * t2;
    }
}
```

Not all parameters necessary in the dependence clauses

```c
#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];
```

Not all parameters necessary in the dependence clauses

```c
for (int j; j<N; j+=BS)
    actual_size = (N- j> BS ? BS: N-j);
```
Examples dependency clauses, array sections

```c
for (iPattern = 0; iPattern < g_ds.nPattern; iPattern += TASK_ITERATIONS)
{
    int n = g_ds.nPattern - iPattern > TASK_ITERATIONS ? TASK_ITERATIONS: g_ds.nPattern - iPattern;

    #pragma omp task input (pl[iPattern*4;n*4], pr[iPattern*4;n*4]) \
    output (pn[iPattern*4;n*4]) firstprivate (n) private (i)
    for (i = iPattern; i < iPattern+n; i++)
    {
        pn[i*4+0] = t1 * t2;

        pn[i*4+1] = t1 * t2;

        pn[i*4+2] = t1 * t2;

        pn[i*4+3] = t1 * t2;
    }
}
```

---

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

#pragma omp task input([DIM][DIM]A, [DIM][DIM]B) inout([DIM][DIM]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);
}
Examples dependency clauses, array sections

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

```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 < NB; k++)
        matmul ((double (*)(*)[NB])A[i][k], (double (*)(*)[NB])B[k][j],
                 (double (*)(*)[NB])C[i][j], NB);
}
```

Examples dependency clauses, array sections

```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*Nt+k] * B[k*Nt+j];
}
```

```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 < NB; k++)
        #pragma omp task input([NB][NB]A[i][k], [NB][NB]B[k][j]) inout([NB][NB]C[i][j])
          matmul (A[i][k], B[k][j], C[i][j], NB);
}
```
Examples dependency clauses, array sections

```
void matmul(double (*A)[NB], double (*B)[NB], double (*C)[NB],
            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][k] * B[k][j];
}
```

```
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;
    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, [NB][NB]B)
                inout([NB][NB]C)
                matmul ((double (*)(*)[NB])A[i][k],
                         (double (*)(*)[NB])B[k][j],
                         (double (*)(*)[NB])C[i][j], NB);
}
```

**Examples dependency clauses, array sections**

- **N** = total size of matrix
- **NB** = block size
- **DIM** = number of blocks

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

```
void compute(unsigned long NB, unsigned long DIM, double *A, double *B, double *C)
{
    unsigned i, j, k;
    for (i = 0; i < N; i += NB)
        for (j = 0; j < N; j += NB)
            for (k = 0; k < N; k += NB)
                matmul (&A[i*N+k*NB], &B[k*N+j*NB], &C[i*N+j*NB], NB);
}
```
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)[n], float *Ah[nt][nt]) {
    for (i=0; i<nt; i++)
        for (j=0; j<nt; j++) gather_block (n, bs, A, i, j, Ah[i][j]);
}

void convert_to_linear(int n, int bs, float *Ah[nt][nt], float (*A)[n]) {
    for (i=0; i<nt; i++)
        for (j=0; j<nt; j++) scatter_block (n, bs, A, Ah[i][j], i, j);
}

#pragma omp task input ([n]A) output ([bs][bs]bA)
void gather_block (int n, int bs, float (*A)[n], int I, int J, float *bA);
#pragma omp task input ( [bs][bs]bA) concurrent([n]A)
void scatter_block (int n, bs, float (*A)[n], float *bA, I, J);

#pragma omp task input ([n]A) output ([bs][bs]bA)
void gather_block (int n, int bs, float (*A)[n], int I, int J, float *bA);
#pragma omp task input ( [bs][bs]bA) concurrent([n]A)
void scatter_block (int n, bs, float (*A)[n], float *bA, I, J);
Concurrent

```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
  - 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, ...

```c
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);
}
```
Hierarchical task graph

- Nesting
  - Tasks can generate tasks themselves
- Hierarchical task dependences
  - Dependences only checked between siblings
    - Several task graphs
    - Hierarchical
    - 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

```c
#pragma omp task input([BS][BS]A, [BS][BS] B) inout([BS][BS]C)
void block_dgem(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]);
}

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

```
#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) output([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);
dgemm(C,F,G);
#pragma omp taskwait

//Hierarchical task graph
```

Block data-layout

```
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:
    - consistent copy needed in the device, may require a transfer
  - copy_deps: same as above, to copy data specified in input/output/inout clauses
  - implements: specifies alternate implementations

Not only for tasks, also to indicate to the compiler that a given function or kernel is specific of a device
```

```
#pragma target device (smp) copy_inps
#pragma target device (gpu) copy_outps
#pragma omp task input ([size] c) copy_out([size]b)
void scale_task (double *b, double *c, double scalar, int size) {
    int j;
    for (j=0; j < BSIZE; j++)
        b[i] = 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.x = 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
    }
}
Example sentinels

```c
#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

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

- 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

OmpSs Environment

- Code
- Mercurium (compiler)
- Transformed code
- Native compiler
- Executable
- Object code
- Nanos++ (runtime)
- Executable
- Extrae
- Trace
Compiling

- 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>Description</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 (In development)</td>
</tr>
</tbody>
</table>

Compiling

- Compatibility flags:
  - `-I`, `-g`, `-L`, `-l`, `-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>--instrument</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>
Executing

- No LD_LIBRARY_PATH or LD_PRELOAD needed
  ```
  ./bin
  ```
- Adjust number of threads with OMP_NUM_THREADS
  ```
  OMP_NUM_THREADS=4 ./bin
  ```

Nanos++ options

- 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 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`

Schedulers

- Available schedulers for `--schedule` option
  - default
    - centralize queue, LIFO scheduler, follows dependency edges
  - bf
    - centralized queue, FIFO scheduler
  - dbf
    - multiple queues, FIFO scheduler with work-stealing
  - affinity
    - multiple queues, uses copy-information to decide which thread to schedule to
  - wf
    - multiple queues, work-first approach with work-stealing
  - Several options to modify behavior
- Example
  `NX_ARGS="--schedule=bf" ./exec`
**Throttle policies**

- Available policies for \(--\text{throttle} \) option
  - num-tasks (default)
    - stop creating tasks if more than limit * #threads are in flight
  - taskdepth
    - stop creating tasks if recursion depth is bigger than limit
  - idlethreads
    - stop creating tasks if no thread is idle
  - readytasks
    - stop creating tasks if more than limit * #threads tasks are ready
- Example
  
  \[
  \text{NX\_ARGS=}\text{"--throttle=taskdepth --throttle-limit=4" \ ./exec}
  \]

**Tracing**

- Compile and link with \(--\text{instrument} \)
  
  \[
  \text{mcc --ompss --instrument -c bin.c} \\
  \text{mcc -o bin --ompss --instrument bin.o}
  \]
- When executing specify which instrumentation module to use:
  
  \[
  \text{NX\_ARGS=}\text{"--instrumentation=extrae" \ ./bin}
  \]
- Will leave 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, throttle=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

Tasks’ profile

- 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 estadistic; i.e., number of tasks instances

different colours represent different task type
Tasks duration histogram

- 3dh_duration_task.cfg

Gradient color indicates given statistic: i.e., number of tasks instances.
Tasks duration histogram

- 3dh_duration_task.cfg

3D window:
- task type

Tasks duration histogram

- 3dh_duration_task.cfg

3D window:
- task type
- chooser:
  - task type
Threads state profile

- 2dp_threads_state.cfg

control window: timeline where each color represents the runtime state of each thread

Single node hands-on