Multicore programming environments: CellSs and SMPSs

Rosa M. Badia

BSC
Outline

- StarSs programming model
- CellSs/SMPSs
- Contributions to standards: OpenMP
- Conclusions
STARSs programming model

- Superscalar processor
  - Instructions
  - Functional units
  - Registers
  - Memory
- Flow sequential program
- Concurrent execution, out of order, speculation, ...
**STARSs programming model**

- **Basic idea**

Sequential Application

```c
for (i=0; i<N; i++){
    T1 (data1, data2);
    T2 (data4, data5);
    T3 (data2, data5, data6);
    T4 (data7, data8);
    T5 (data6, data8, data9);
}
```

Task graph creation based on data precedence

Task selection + parameters direction (input, output, inout)

Synchronization, results transfer

Parallel Resources (multicore, SMP, cluster, grid)

Scheduling, data transfer, task execution

Resource 1

Resource 2

Resource 3

Resource N
STARSs programming model

- Main objective: Reduce the complexity of applications development
  - Complexity of writing an application for a parallel platform comparable to writing it for a sequential platform

- Main characteristics
  - Task: unit of parallel work
  - Non intrusive programming model
  - Data dependence detection
  - Data renaming
  - Exploitation of distant parallelism
StarSs programming model

- **GRIDSs, COMPSs**
  - Tailored for Grids or clusters
  - Data dependence analysis based on files
  - C/C++, Java

- **SMPSs**
  - Tailored for SMPs or homogeneous multicores
  - Altix, JS21 nodes, Power5, Intel-Core2
  - C or Fortran

- **CellSs**
  - Tailored for Cell/B.E. processor
  - C or Fortran
GRIDSs applications: Analysis of protein and function diversity on earth

Largest protein comparison and classification done so far

- 15 million protein sequences
- BLAST orchestrated by GRIDSs
- Query and Database file size: ~ 5 Gb
- 4000 CPUs (= 1000 exclusive nodes)
- Total CPU time: 311,112 hours
- 5 Tb of results
- 100,000 tasks in each GRIDSs run

Proteins and organisms classification
GRIDSs applications: BEinGRID BE14 - Design of products and processes

- Applied to the acetic acid production process
- Around 150,000 tasks per execution
- Multisite runs: Barcelona, Madrid & Surrey

![Diagram of Process Network](image)

- Reactor network
- Separation network
- Recycles
- Feeds
- Products

![Graph of Speedup vs. # CPUs](image)

- Intensity 6
- Intensity 7
- Intensity 8

![Institution Logos](image)
GRIDSs + MPI + OpenMP: Reverse Time Migration (RTM)

Only RTM produces proper subsalt imaging
Computationally more intensive

- 1 GRID superscalar application per image
  - 350,000 – 500,000 tasks per image
- Domain Decomposition (MPI) to process one shot between several blades
- Threads
  - OpenMP to execute one MPI process per JS21blade
- SIMD capabilities
  - VMX code
Cell/B.E. Architecture

So, what is the Cell/B.E.?

Architecture point of view

Programmer's point of view

Thin processor
SMT
16B/cycle

Synergistic processor elements for high (R)ops/Watt

SPE
SPE
SPE
SPE
SPE
SPE
SPE
SPE

L2
32B/cycle

L1
64-bit Power Architecture with VMX for traditional computation

PPE
16B/cycle

EIB (up to 64B/cycle)

Hard to optimize

16B/cycle (2x)

SPE

Separate address spaces
Tiny local memory
Bandwidth

Users' point of view

The Cell/BE is like a tiny Grid!
CellSs/SMPSs

- Pragma based programming model: programmer specifies tasks (functions) and direction of arguments
  - `#pragma css task input (…) output (…) inout (…)`
  - `function-definition|funtion-declaration`

- Dependences replace barriers
  - Sources of unbalance and overhead
  - Exploitation of distant parallelism

- Portability: sequential, SMP, homogeneous multicore, Cell, …

- Constraints
  - Blocked algorithms, task granularity
  - Tasks can only access function arguments and local data
CellSs/SMPSs syntax

```c
#pragma css task input(A, B) inout(C)
static void block_addmultiply( float C[BS][BS], float A[BS][BS], float B[BS][BS]);

#pragma css task inout(diag[B][B]) highpriority
void lu0(float *diag);
#pragma css task input(diag[B][B]) inout(row[B][B])
void bdiv(float *diag, float *row);
```

```c
interface
  !$CSS TASK
  subroutine velocity(BSIZE, ii, jj, xi, yi, zi, xj, yj, zj, vx, vy, vz)
    implicit none
    integer, intent(in) :: BSIZE, ii, jj
    real, intent(in), dimension(BSIZE) :: xi, yi, zi, xj, yj, zj
    real, intent(inout), dimension(BSIZE) :: vx, vy, vz
  end subroutine

  !$CSS TASK
  subroutine v_mod(BSIZE, v, vx, vy, vz)
    implicit none
    integer, intent(in) :: BSIZE
    real, intent(out) :: v(BSIZE)
    real, intent(in), dimension(BSIZE) :: vx, vy, vz
  end subroutine
end interface
```
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++)
                block_addmultiply (C[i][j], A[i][k], B[k][j]);
}

static void block_addmultiply (float C[BS][BS], float A[BS][BS],
    float B[BS][BS]) {
    int i, j, k;
    for (i=0; i < B; i++)
        for (j=0; j < B; j++)
            for (k=0; k < B; k++)
                C[i][j] += A[i][k] * B[k][j];
}
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++)
        block_addmultiply( C[i][j], A[i][k], B[k][j]);
}

#pragma css task input(A, B) inout(C)
static void block_addmultiply( float C[BS][BS], float A[BS][BS],
                              float B[BS][BS]) {
  int i, j, k;

  for (i=0; i < B; i++)
    for (j=0; j < B; j++)
      for (k=0; k < B; k++)
        C[i][j] += A[i][k] * B[k][j];
}
main (){...
for (int j = 0; j < DIM; j ++ ){
for (int k = 0; k < j; k ++ ){
   for (int i = j + 1; i < DIM; i ++ ){
      css_sgemm_tile( A[i][k], A[j][k], A[i][j] );
   }
for (int i = 0; i < j; i ++ ){
   css_ssyrk_tile(A[j][i], A[j][j]);
}
// Cholesky Factorization of A[j,j]
   css_spotrf_tile( A[j][j] );
   for (int i = j + 1; i < DIM; i ++ ){
      css_strsm_tile( A[j][j], A[i][j] );
   }
   // Cholesky Factorization of A[i,j]
   css_spotrf_tile( A[i][j] );
}
for (int i = 0; i < DIM; i ++ ){
   for (int j = 0; j < DIM; j ++ ){
      //pragma css wait on (A[i][j])
      print_block(A[i][j]);
   }
   //pragma css task input(A[64][64], B[64][64]) inout(C[64][64])
   void sgemm_tile(float *A, float *B, float *C)
   #pragma css task input (T[64][64]) inout(B[64][64])
   void strsm_tile(float *T, float *B)
   #pragma css task input(A[64][64]) inout(C[64][64])
   void ssyrk_tile(float *A, float *C)
   #pragma css task inout(A[64][64])
   void spotrf_tile(float *A)
   void print_block(float *A)
CellSs: Runtime

Main features
- Scheduling of chains/clusters of tasks
- Double buffering
- Early callback
- Performance traces
CellSs compilation environment
SMPSs: runtime

Main features
- Scheduling of individual tasks
- Each thread has private ready list
- Performance traces
SMPSs hierarchical

- Ideally: allow nesting at different levels of an application
- Current ongoing effort: SMPSs + CellSs
int main (int argc, char **argv) {
    int i, j, k;

    float A[NBB][NBB][NSB][NSB][BS][BS], B[NBB][NBB][NSB][NSB][BS][BS], C[NBB][NBB][NSB][NSB][BS][BS];

    for (i = 0; i < NBB; i++)
        for (j = 0; j < NBB; j++)
            for (k = 0; k < NBB; k++)
                dgem2(&C[i][j][0][0][0][0], &A[i][k][0][0][0][0], &B[k][j][0][0][0][0]);

#pragma css task input(A, B) inout(C)
void dgem2 (float C[NSB][NSB][BS][BS], float A[NSB][NSB][BS][BS], float B[NSB][NSB][BS][BS]) {
    int i, j, k;
    for (i = 0; i < NSB; i++)
        for (j = 0; j < NSB; j++)
            for (k = 0; k < NSB; k++)
                dgem1(  &C[i][j][0][0], &A[i][k][0][0], &B[k][j][0][0]);

#pragma css task input(A, B)  inout(C)
#pragma css target device(cell)
void dgem1 (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];

}
OpenMP 3.0 has been extended with tasks

- Both loop parallelism and parallel tasks can conform an OpenMP application
- However, no data dependence is supported

BSC is on the process of proposing an extension based on StarSs experience (IWOMP08, IWOM09)

- To extend OpenMP to handle dependences
- To support the heterogeneity of the current and forthcoming platforms
Contribution to OpenMP

```c
#pragma omp task inout(C[BS][BS])
void matmul( float *A, float *B, float *C) {
    // Regular C code
}
#pragma omp target device(cuda) implements(matmul) \ 
    copy_in(A[BS][BS], B[BS][BS], C[BS][BS]) copy_out(C[BS][BS])
void matmul_cuda ( float *A, float *B, float *C) {
    // optimized kernel for cuda
}
#pragma omp target device(cell) implements(matmul) \ 
    copy_in(A[BS][BS], B[BS][BS], C[BS][BS]) copy_out(C[BS][BS])
void matmul_sdk ( float *A, float *B, float *C);
    // optimized kernel for the Cell SPUs
```
Conclusions

- Task based programming model looks a good approach

- However
  - Is not the only solution
    - Should co-exist with others: MPI, streaming, TM, …
    - Must converge with OpenMP
  - Automatism is not always well accepted
    - Need to educate programmers
  - Overheads in architectures like Cell make it difficult … although more fun
    - Current issues can be solved in forthcoming devices’ generations
  - Need for ways of expressing the hierarchy of resources in the programming model
    - Must for many-cores and heterogeneous devices

- Ongoing work in different platforms: Cell, SMP, GPU, hierarchical, SMPSs + MPI
STARSSs websites

- Contact
  - rosa.m.badia@bsc.es
- GRIDSSs
  - www.bsc.es/grid/gridsuperscalar
- CellSSs
  - www.bsc.es/cellsuperscalar
- SMPSSSs
  - www.bsc.es/smpsuperscalar
- All of them available for download (open source, Apache v2, GPL and LGPL)