StarSs and CellSs: Data Flow Programming for Multicore Systems

Jesus Labarta
Computer Science Research Department Director
BSC
Back to Babel?

Book of Genesis

“Now the whole earth had one language and the same words” …

…”Come, let us make bricks, and burn them thoroughly. ”…

…”Come, let us build ourselves a city, and a tower with its top in the heavens, and let us make a name for ourselves”…

And the LORD said, "Look, they are one people, and they have all one language; and this is only the beginning of what they will do; nothing that they propose to do will now be impossible for them. Come, let us go down, and confuse their language there, so that they will not understand one another's speech."
Key issues

- Programmability
- Productivity
- Abstraction
- Accepted by developers
- Asynchronism
- Scalability
- Portability
- Heterogeneous functionality and performance
- Memory association
- % memory used
- Memory
- Variance
- Malleability
- Hierarchy
- Processor-memory bandwidth
- Load balance
- Maintainability
- Dynamic environment
- Accepted by developers
- Abstraction
- Asynchronism
- Scalability
- Portability
- Heterogeneous functionality and performance
- Memory association
- % memory used
- Memory
- Variance
- Malleability
- Hierarchy
- Processor-memory bandwidth
- Load balance
- Maintainability
- Dynamic environment
Key issues

Programmability
Productivity
Maintainability
Portability
Dynamic environment
Variance
Salleability
Load balance
Scalability
Memory association
Malleability
Load balance
Processor-memory bandwidth
Hierarchy
Abstraction
Accepted by developers
Asynchronism
Performance
Dazzled by Performance

Jesus Labarta, STI workshop on Soft. and Apps. for the Cell/B.E., 2008
Address spaces in multicores

Different cache hierarchies

Homogeneous in structure

Niagara, Nehalem, Power…

A really fuzzy space

How much visible to the programmer???
The presentation

- Environment, philosophy,…
- CellSs
- Work in Progress
- SMPSs
- MPI+SMPSs
- Summary
Mare Incognito

An IBM-BSC joint research project towards a 10+ PF supercomputer

“Homogeneous” Cell based system

We believe it is possible to build a

“cheap”, “efficient”, “not architecture specific” and

“wide applicability” machines based on it

We know it is risky

We have “a vision” of
relevant technologies to develop

Many of these technologies are not Cell specific
and will be also evaluated/used for other architectures
Project Structure

- Model and prototype
- Fine-grain programming models
- Load balancing
- Processor and node
- Interconnect
- Performance analysis and Prediction Tools
- Application development and tuning
Programming model for MareIncognito?

- How much pressure can we put on our (BSC) program developers?
  - Drastic changes?

- Hierarchy
  - MPI
  - OpenMP, StarSs, ..... 

- “beneficial” transformations:
  - Blocking
  - Better understanding of inputs and outputs
  - Understanding potential asynchronism

- Incremental
How my daughter sees the Cell
How Mateo sees the Cell

- Dual XDRTM (up to 25.2 GByte/s)
- PPE dual-threaded
- EIB (up to 96 Bytes/cycle)
  4 x 16B data rings at half core clock frequency
- FlexIO 6.4 GByte/s per link (76.8 total)
- L2 512 KB
- L1 32 KB I$ 32 KB D$
- MIC 16B/cycle 16B/cycle 16B/cycle (2x)
- SXU 32B/cycle
- MFC 16B/cycle
- LS
- BIC
How I see the Cell

Simple hardware
Aggregate performance
Vector instructions

Hard to optimize
Only vector instructions
Alignment

Thin processor
SMT

PPE
SPE
SPE
SPE
SPE
SPE
SPE
SPE
SPE
SPE

Separate address spaces
Explicit DMAs
Tiny local memory

Explicit many memory transaction
Bandwidth

Necessity → virtue
A perspective on architectures and programming models

Mapping of concepts:
- Instructions ➔ Block operations
- Functional units ➔ SPUs
- Fetch & decode unit ➔ PPE
- Registers (name space) ➔ Main memory
- Registers (storage) ➔ SPU memory

Granularity
Stay sequential
Just look at things from a bit further away
Architects do know how to run parallel
StarSs

CellSs

SMPSs

GPUSs

GridSs

**Programability**
- Standard sequential look and feel (C, Fortran)
- Incremental parallelization/restructure
- Abstract/separate algorithmic issues from resources
- Methodology/practices
  - Block algorithms: modularity
  - “No” side effects: local addressing
  - Promote visibility of “Main” data
  - Explicit synchronization variables

**Portability**
- Runtime for each type of target platform.
  - Matches computations to resources
  - Achieves “decent” performance
- Even to sequential platform
- Single source for maintained version of a application
CellSs
CellSs concept: matrix multiply example

```c
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]);
}
```

```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 < BS; i++)
        for (j=0; j < BS; j++)
            for (k=0; k < BS; k++)
                C[i][j] += A[i][k] * B[k][j];
}
```
CellSs concept: matrix multiply example

```c
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]);
}
```

```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]) {
    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];
}
```

Dynamically defined instruction set

Modularity
CellSs compilation environment

- **app.c**
- **app_spe.c**
- **app_ppe.c**
- **llib_css-spe.so**
- **llib_css-ppe.so**

**SDK**

- **CSS compiler**
- **SPE Compiler**
- **app_spe.o**
- **SPE Linker**
- **SPE executable**
- **SPE Embedder**
- **PPE Compiler**
- **app_ppe.o**
- **PPE Linker**
- **PPE Object**
- **Cell executable**

**CellSs compilation environment**
CellSs execution environment
CellSs: argument renaming

- **Supports:**
  - Decouple data declaration/allocation from storage
  - Increase parallelism potential
  - Adapt to available memory
- **When**
  - Eager: at task creation time
  - Lazy: at task execution time
- **Where to dynamically allocate memory**
  - Main memory (GB)
  - LS: avoid transfers
- **How much**
  - Limit the size of memory used for renaming. Stall when reached.
  - Avoid swapping
  - Impact on locality
- **Programmability**
  - Restrict/focus Important data declarations
CellSs Programming model: Dependence detection

Instruction Window: 40000+  !!!!
Speculation: needed?
Memory

- Object cache
  - Reuse SPE Local Store
  - Avoid DMA transfers

- Double buffer
  - Automatic: before executing a task, launch
    - DMA in for arguments for the next
    - DMA out for results of previous
  - Dependent on object sizes – LS pressure
  - Handle alignment issues

- Locality:
  - can scheduler
    - Maximize the reuse of LS?
  - Can alleviate the off chip bandwidth bottleneck
  - Knowing data accesses
CellSs: continuous runtime tuning

- Master overhead
  - ~3 us. Something to control, minimize

- Dependence
  - storage capacity/addressing ↔ task granularity

- How will it evolve?
  - Future Cell
  - Other platforms (Roadrunner)

- Distribute implementation? Offload?
- Multiple *Ss engines
Flexible/dynamic model: SparseLU example

```c
int main(int argc, char **argv) {
    int ii, jj, kk;
    ...

    for (kk=0; kk<NB; kk++) {
        lu0(A[kk][kk]);
        for (jj=kk+1; jj<NB; jj++)
            if (A[kk][jj] != NULL)
                fwd(A[kk][kk], A[kk][jj]);
        for (ii=kk+1; ii<NB; ii++)
            if (A[ii][kk] != NULL) {
                bdiv (A[kk][kk], A[ii][kk]);
                for (jj=kk+1; jj<NB; jj++)
                    if (A[kk][jj] != NULL) {
                        if (A[ii][jj]==NULL)
                            A[ii][jj]=allocate_clean_block();
                        bmod(A[ii][kk], A[kk][jj], A[ii][jj]);
                    }
            }
    }
}
```

void lu0(float *diag);
void bdiv(float *diag, float *row);
void bmod(float *row, float *col, float *inner);
void fwd(float *diag, float *col);
int main(int argc, char **argv) {
    int ii, jj, kk;
    ...

    for (kk=0; kk<NB; kk++) {
        lu0(A[kk][kk]);
        for (jj=kk+1; jj<NB; jj++)
            if (A[kk][jj] != NULL)
                fwd(A[kk][kk], A[kk][jj]);
        for (ii=kk+1; ii<NB; ii++)
            if (A[ii][kk] != NULL) {
                bdiv (A[kk][kk], A[ii][kk]);
                for (jj=kk+1; jj<NB; jj++)
                    if (A[kk][jj] != NULL) {
                        if (A[ii][jj]==NULL)
                            A[ii][jj]=allocate_clean_block();
                        bmod(A[ii][kk], A[kk][jj], A[ii][jj]);
                    }
            }
    }
}

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);
pragma css task input(row[B][B],col[B][B]) inout(inner[B][B])
void bmod(float *row, float *col, float *inner);
pragma css task input(diag[B][B]) inout(col[B][B])
void fwd(float *diag, float *col);
Distant/interprocedural parallelism: Test LU example

- Ideally: unrolling of whole program

```c
int main(int argc, char* argv[]) {
    long t_start, t_end;
    double time;

genmats(L, U);
clean_mat (A);
sparse_matmult (L, U, A);
copy_mat (A, origA);
t_start=usecs();
LU (A);
t_end=usecs();
time = ((double)(t_end-t_start))/1000000;
split_mat (A, L, U);
clean_mat (A);
sparse_matmult (L, U, A);
compare_mat (origA, A, &are_equal);
#pragma css wait on (&are_equal)
    if (are_equal) printf("Great!\nAnd only took %f\n", time);
}
```
Work in Progress
N Dimensional data structures: Syntax

```plaintext
#pragma css task
input(parameter_list)opt output(parameter_list)opt inout(parameterlist)opt highpriority opt
```

function_definition or function_declaration

parameter:

```
name dimension_specifier_list opt range_specifier_list opt
```

dimension_specifier:  range_specifier:

```
[length]               {}
{index}                {index}
{begin..end}           {begin..end}
{begin:length}         {begin:length}
```

A

A{}

A[i..j]

A{k..m}

A[i..j]{k..m}
N Dimensional data structures

- How to specify subsections of array structures

```c
#pragma css task input( a{i:BS}{k:BS}, b{k:BS}{j:BS}, i, j, k )
  inout( c{i:BS}{j:BS} )
void matmul(float a[N][N], float b[N][N], float c[N][N], int i, int j, int k)

int main(...) {
  ...
  for (i=0; i<N; i+=BS)
    for (j=0; j<N; j+=BS)
      for (k=0; k<N; k+=BS)
        matmul(a,b,c,i,j,k);
  ...
```
N Dimensional data structures

- How to specify subsections of array structures

```c
#pragma css task input(transa, transb, m, n, k, alpha, lda, ldb, beta, ldc, \
          a[*k][*lda] {}{0:*m}, \n          b[*n][*ldb] {}{0:*k}) \ninout(c[*n][*ldc] {}{0:*m}

void dgemm(char *transa, char *transb, \
          integer *m, integer *n, integer *k, double *alpha, \
          double *a, integer *lda, \n          double *b, integer *ldb, \n          double *beta, double *c, integer *ldc);
```
Global memory addressing

- **Local addressing**
  - Enables automatic double buffering
    - Large no speculative prefetching

- **Task granularity**
  - Depends on operands size
    - Limited by local storage capacity
    - Limits scalability given a task handling cost

- **Global addressing decouples granularity and storage capacity**
  - **Manual:**
    - Already possible (mfc_read,.....)
    - Escapes the dependence control mechanism. User responsibility.
  - **Automatic:**
    - Requires caching mechanism

- **Can be cleanly combined?**
  - Separation of dependence and data transfer
Nested StarSs

- Hierarchical task graphs
- Concurrent generation to flat task graph

Model: OpenMP + StarSs, ??
Opportunity for Transactional Memory within Run Time
Reductions

- Similar to OpenMP syntax
  - Private copy initialized to operator neutral value
  - Local update during task
  - Atomic update of global value

- Dependence semantics
  - Commutative
  - Count of number of users

To appear on next distribution
Memory association

• Data layout
  • Main memory and LS layout of data blocks
    • Necessarily different because of LS size.
    • If we have to address the issue, can we exploit it more?
  • ≈ work arrays
  • Potential to handle automatically
    • Avoid specific user declarations and code. Abstraction.
    • Several instances of an object, with different layouts.
    • Dynamically managed by run time. (cache)

• Code restructuring
  • change access patterns
SMPSs
SMPSs

- “Same” source code
- Although more flexibility (!block size,…..)
- Same compiler
- Although not force to address the memory association issue
- SMP run time
- Shared memory implementation

2 ways POWER 5

LU performance

Number of threads

Gflops

Machine peak
1024
2048
4096

Number of threads

Gflops

LU performance

SMPs

 Machine peak
1024
2048
4096

Number of threads

Gflops

LU performance

SMPs
Asynchronism: I/O

- Trace handling and visualization tools
- Statistics
- Filters

```c
#pragma css task inout(fd)
    output(buffer, R_recs, end_trace) highpriority
void Read (FILE *fd, char buffer[500][4096], int *R_recs,
        int *end_trace);
#pragma css task input(buffer_in, R_recs, control)
    output(buffer_out, W_recs)
void Process (char buffer_in[500][4096], int R_recs,
               proc_ctrl control,
               char buffer_out[500][4096], int *W_recs);
#pragma css task input(buffer, W_recs)
    inout(fd, total_records)
void Write (FILE *fd, char buffer[500][4096], int W_recs,
           unsigned long long *total_records);

int main()
{
 ...  
while(!end_trace) {
    Read (infile, buffer_in, &RR, &end_trace);
    Process (buffer_in, RR, control, &RW, buffer_out);
    Write (outfile, buffer_out, RW, &total_records);
    #pragma css wait on (&end_trace)
}
}```
Asynchronism: I/O

- Asynchronism
- Decoupling/overlapping I/O and processing
- Serialization of I/O
  - Request for specific thread,…
- Task duration variance
- Dynamic schedule
SMPSs vs OpenMP 3.0 vs Cilk

- Benchmarks used for OpenMP 3.0 development
- Similar performance in some ranges
- Overlap potential in SMPSs
- Programmability issues
  - Reductions, memory allocations, synchronization representatives, nesting,…

Sparse LU

N queens

Multi sort

Graphs showing speedup vs. sequential for different benchmarks and thread counts.
MPI + SMPSs
**MPI + SMPSs**

- **Linpack**
  - Simplify program
    - Same source code for any level of lookahead (0-n). No specific code needed inside each computation routine
  - Decouple/overlap computation and synchronization
    - Tolerate slow networks
    - Tolerance to OS noise
    - Programmer can concentrate on other issues
      - Load balance,…
  - Smaller problem size to get max performance
    - Memory used if necessary by run time (within limited renaming space)

Linpack distribution
- Include directory: 19 files, 1946 code lines
- src directory: 113 files, 11945 code lines
- Parameters: P, Q, Broadcast, lookahead
**MPI + SMPSs**

- Extend asynchronism to outer level

```c
for (k=0; k<N; k++) {
    if (mine) {
      Factor_panel(A[k]);
      send (A[k])
    } else {
      receive (A[k]);
      if (necessary) resend (A[k]);
    }
    for (j=k+1; j<N; j++)
      update (A[k], A[j]);
...}
```

```c
#pragma css task inout(A[SIZE])
void Factor_panel(float *A);
#pragma css task input(A[SIZE]) inout(B[SIZE])
void update(float *A, float *B);
#pragma css task input(A[SIZE])
void send(float *A);
#pragma css task output(A[SIZE])
void receive(float *A);
#pragma css task input(A[SIZE])
void resend(float *A);
```
MPI + SMPSs

- Overlap communication and computation
- Asynchronous/immediate MPI_calls + wait tasks
- Restartable task
- Avoid deadlock
- Avoid inefficient use of resources (busy wait)

```c
#pragma css task input(A[SIZE]) output(send_req)
void Isend(float *A, int *send_req)
{
    MPI_isend (A,...);
}

#pragma css task input(send_req) inout(A{SIZE})
void Wait_Isend(int *send_req, float *A)
{
    int ierr, go;
    ierr = MPI_Test(send_req,&go,...)
    if(go==0) #pragma css restart
    ierr = MPI_Wait(send_req,...);
}

#pragma css task output(recv_req,A[SIZE])
void Ireceive(int *recv_req, float *A)
{
    MPI_Irecv(A,...);
}

#pragma css task input(recv_req) inout(A[SIZE])
void Wait_Ireceive(int *recv_req, float *A)
{
    int ierr, go;
    ierr = MPI_Test(recv_req,&go,...)
    if(go==0) #pragma css restart
    ierr = MPI_Wait(recv_req,...);
}
```
MPI + SMPSs
MPI + SMPSs
Summary
StarSs

- **Parallelism**
  - Automatic run time parallelism detection and exploitation
  - Unstructured (DAG)
  - Data dependent parallelism
  - Exploit distant, interprocedural parallelism
  - Asynchronous - OoO - Dynamic data flow execution
    - Tolerates variance
    - Extends to outer programming models/issues
      - I/O
      - MPI + StarSs
  - Malleable
  - Incremental code parallelization

- **Memory**
  - Single global address space
  - Renaming:
    - Decouple data declaration/allocation from storage (available size and structure)
    - Increase parallelism potential
    - Adapt to available memory
  - Potential for automatic handling of memory association issues

- **Memory wall:**
  - Latency tolerance
    - Automatic double buffering
    - DMA transfers: ~ vector loads
  - Automatic locality optimization
StarSs

- Programming practices
  - Make visible relevant data structures
  - Explicit synchronization variables
  - Does require some effort, but globally beneficial.
  - Clean/structure code
Convergence? (Hierarchical) Integration?

**OpenMP**
- Explicit parallelism
- Nested tasks
- Global Addressing

**StarSs**
- Implicit parallelism
- “atomic” tasks
- Single work generator
- Local addressing

---

**TM**
- Mechanisms guaranteeing atomicity
- Retry mechanisms
- Speculation $\rightarrow$ detection of dependences,....

**OpenMP 3.X?**
StarSs: issues and ongoing efforts

- CellSs programming model
  - Memory association
  - Array regions
  - Subobject accesses
  - Access to global memory by tasks?
    - Blocks larger than Local Store.
  - Inline directives
  - Reductions
  - Separation Sync – data transfer
  - Resource constraints
  - Nesting:
    - Hierarchical task graph
    - Flat task graph
- Porting:
  - P6 (dcbc)
  - BG

- Runtime
  - Further optimization of overheads.
  - Architectural support
  - Minimal set of synchronization arcs
  - Scheduling algorithms: overhead, locality
  - Overlays
  - Short circuiting (SPE → SPE transfers)

- Debugging
  - Amenable to race detection tools

- Compiler
  - Dusty deck → StarSs

- Applications
  - Porting from sequential
  - Port from parallel
  - New algorithms

http://www.bsc.es/cellsuperscalar
http://www.bsc.es/smpsупerscalar
The importance of ...

- Language
- Transport / integrate
  - Experience → Good “predictor”
- Structures mind
  - Abstraction

- Details
  - Very “simple/small/subtle” differences → very important impact
The importance of ...

- Language
- Transport / integrate
  - Experience → Good “predictor”
- Structures mind

Wish we were able to develop language:

- Communicated to humans
- Exploitable by machines

- Details
  - Very “simple/small/subtle” differences → very important impact