Sequoia Implementation

CS149 Lecture 13

Recall

```
A B C
Located at level X
```

```
Located at level Y
```

```
matmul::inner
```

```
matmul::leaf
```

```
float A[M][T],
float B[T][N],
inout float C[M][N]
```

```
tunable int P, Q, R;
```

```
mappar(int i=0 to M/P,
int j=0 to N/R) {
mapseq(int k=0 to T/Q) {
    matmul(A[P*i:P*(i+1);P][Q*k:Q*(k+1);Q],
          B[Q*k:Q*(k+1);Q][R*j:R*(j+1);R],
          C[P*i:P*(i+1);P][R*j:R*(j+1);R]);
}
}
```

```
for (int i=0; i<M; i++)
for (int j=0; j<N; j++)
for (int k=0; k<T; k++)
C[i][j] += A[i][k] * B[k][j];
```

Recall: Blocked Matrix Multiplication

```
void matmul(int M, int N, int T,
            float* A,
            float* B,
            float* C) {
    ... call matmul recursively ...
}
```

Mapping

Abstract Task Hierarchy

```
task matmul::inner in float A[M][T],
in float B[T][N],
out float C[M][N] ) {
    tunable int P, Q, R;
    mappar(int i=0 to M/P,
           int j=0 to N/R) {
        mapseq(int k=0 to T/Q) {
            matmul(A[P*i:P*(i+1);P][Q*k:Q*(k+1);Q],
                   B[Q*k:Q*(k+1);Q][R*j:R*(j+1);R],
                   C[P*i:P*(i+1);P][R*j:R*(j+1);R]);
        }
    }
}
```

```
task matmul::leaf in float A[M][T],
in float B[T][N],
out float C[M][N] ) {
    for (int i=0; i<M; i++)
        for (int j=0; j<N; j++)
            for (int k=0; k<T; k++)
                C[i][j] += A[i][k] * B[k][j];
}
```

How Mapping Works

Sequoia task definitions (parameterized)

```
matmul::inner
```

```
matmul::leaf
```

Task instances (not parameterized)

```
matmul_node_inst
```

```
matmul_L1_inst
```

```
matmul_L2_inst
```

Mapping specification

```
instance {
    name = matmul_node_inst
    variant = inner
    runs_at = main_memory
    tunable P=256, Q=256, R=256
}
instance {
    name = matmul_L2_inst
    variant = inner
    runs_at = L2_cache
    tunable P=32, Q=32, R=32
}
instance {
    name = matmul_L1_inst
    variant = leaf
    runs_at = L1_cache
}
```
**Task Mapping Specification**

- **instance**
  - name = matmul_node_inst
  - task = matmul
  - variant = inner
  - runs_at = main_memory
  - tunable P=256, Q=256, R=256
  - calls = matmul_L2_inst

- **instance**
  - name = matmul_L2_inst
  - task = matmul
  - variant = inner
  - runs_at = L2_cache
  - tunable P=32, Q=32, R=32
  - calls = matmul_L1_inst

- **instance**
  - name = matmul_L1_inst
  - task = matmul
  - variant = leaf
  - runs_at = L1_cache

---

**Specializing Matmul**

- Tasks instances placed at each memory level

**Task Instances: Cell**

- Sequoia task definitions (parameterized)
  - matmul::inner
  - matmul::leaf

- Cell task instances (not parameterized)
  - matmul node inst
  - matmul L2 inst
  - matmul L1 inst

**Compiler Optimization**

- Sequoia compilation works on hierarchical programs
- Many "standard" optimizations
  - But done at all levels of the hierarchy
  - Greatly increases leverage of optimization

**Inter-Level Copy Elimination (1)**

Copy elimination near the root removes not one instruction, but thousands/millions
Inter-Level Copy Elimination (2)

Copy elimination near the root removes not one instruction, but thousands/millions

Communication and Computation

Asynchronous Communication

- Preload batch of data
- Compute on data
- Initiate write of results (this data is done)
- Compute on next batch (which should be loaded)

Runtime System

- Uniform scheme for explicitly describing memory hierarchies
  - Capture common traits important for performance
  - Allow composition of memory hierarchies
- Simple, portable API interface for many parallel machines
  - Mechanism independence for communication and management of parallel resources
**Design Requirements**

- Resource allocation
  - Data allocation and naming
  - Setup parallel resources
- Explicit bulk asynchronous communication
  - Transfer lists
  - Transfer commands
- Parallel execution
  - Launch tasks on children
  - Asynchronous
- Synchronization
  - Make sure tasks/transfer complete before continuing
- Runtime isolation
  - No direct knowledge of other runtimes

**Top Interface**

```cpp
// create and free runtime
Runtime(TaskTable table, int numChildren);
virtual ~Runtime();

// allocate and deallocate arrays
virtual Array_t* AllocArray (Size_t elmtSize, int dimensions, Size_t* dim_sizes, ArrayDesc_t descriptor, int alignment) = 0;
virtual void FreeArray(Array_t* array) = 0;

// array naming
virtual void AddArray(Array_t array);
virtual Array_t GetArray(ArrayDesc_t descriptor);
virtual void RemoveArray(ArrayDesc_t descriptor);

// launch and synchronize on tasks
virtual TaskHandle_t CallChildTask(TaskID_t taskid, ChildID_t start, ChildID_t end) = 0;
virtual void WaitTask(TaskHandle_t handle) = 0;
```

**Bottom Interface**

```cpp
// array naming
virtual Array_t* GetArray (ArrayDesc_t descriptor);

// create, free, invoke, and synchronize on transfer lists
virtual XferList* CreateXferList (Array_t* dst, Array_t* src, Size_t* dst_idx, Size_t* src_idx, Size_t* lengths, int count) = 0;
virtual void FreeXferList (XferList* list) = 0;
virtual XferHandle_t Xfer (XferList* list) = 0;
virtual void WaitXfer (XferHandle_t handle) = 0;

// get number of children in bottom level, get local processor id, and barrier
int GetSiblingCount();
int GetID();
virtual void Barrier (ChildID_t start, ChildID_t stop) = 0;
```

**Compiler/Runtime Interaction**

- Compiler initializes runtime for each pair of memories in the hierarchy
- Initialize runtime for root memory
  - Machine description specifies runtime to
- If more levels in hierarchy
  - Initialize runtimes for child levels
- Runtime cleanup is inverse
  - Call exit on children, wait, cleanup local resources, return to parent

**Abstraction Review**

- Tree of nodes
- 1 control thread per node
- 1 memory per node
- Threads can:
  - Transfer in bulk from/to parent memory asynchronously
  - Wait for transfers from/to parent to complete
  - Allocate data
  - Only access their memory directly
  - Transfer control to child node(s)
  - Non-leaf threads only operate to move data and control
  - Synchronize with siblings

**SMP Runtime**

- Abstraction issues
  - Shared memory, so transfer not required
  - No processor at parent level
SMP Implementation

- Data transfers
  - Memory copy from source to destination
  - Optimizations
    - Pass reference to parent array
    - Feedback to compiler to remove transfers
    - Machine file information

- No processor at parent level
  - Node 0 represents the parent node and a child node

Disk Implementation

- No processor at top level
  - Host CPU represents the parent node and child node

- Implementation issues
  - Allocation
    - Open file on disk
  - Data transfers
    - Use Async I/O API to read/write data to disk

Cell Implementation

- Overlay handling
  - At runtime creation, load overlay loader into SPEs
  - On task call
    - PowerPC notifies SPE of function to load
    - SPE loads overlay and executes

- Data alignment
  - All data allocated to 128 byte boundaries
  - Multi-dimensional arrays padded to maintain alignment for dimensions

Disk Runtime

- Abstraction issues
  - No processor at parent level

Cell Runtime

- Abstraction issues
  - Code overlay handling
  - Data allocation, alignment, padding

Cluster Runtime

- Abstraction issues
  - Virtual level used to represent aggregate memory
  - Allocation
  - Data transfers
  - Execution at parent node
Cluster Implementation

• No processor at top level
  - Node 0 represents the parent node and a child node

• Virtual level
  - Distributed Shared Memory (DSM)
    • Many software and hardware solutions
  - Overkill for Sequoia
    • No sharing between sibling processors = no coherence requirements

Cluster Implementation, cont.

• Virtual level implementation
  - Interval trees
    • Represent arrays as covering an range of 0→N elements
    • Each node can have any subportion (interval) of range
    • Tree structure allows fast lookup of all nodes that cover the interval of interest
    • Allows complex data distributions
  - Array allocation
    • Define distribution as interval tree and broadcast
    • Allocate data for intervals and register data pointers with MPI-2 (MPI_win_create)
    • Align data to page alignment for network fast transfers

Cluster Implementation, cont.

• Virtual level implementation
  - Data transfer
    • Compare requested data range against interval tree
    • Read from parent: issue MPI_Get to any nodes with matching intervals (MPI_LOCK_SHARED)
    • Write to parent: issue MPI_Put to all nodes with matching intervals (MPI_LOCK_EXCLUSIVE)
  - Optimizations
    • If requested range is local, return reference to parent memory

Runtime Composition

Autotuner

• Many parameters to tune
  - Sequoia codes parameterized by tunables
  - Abstract away from machine particulars
    • E.g., memory sizes
  - The tuning framework sets these parameters
    • Search-based
    • Programmer defines the search space
    • Bottom line: Autotuner is a big win
      • Never worse than hand tuning (and much easier)
      • Often better (up to 15% in experiments)
Software-Managed Memory

Smooth with high-frequency components
(due to alignment)

Hierarchical Search

Bottom Up
M2
set of tunables: S1
M1
set of tunables: S0
M0

Search Algorithm

grid spacing
Start with a coarse grid
Refine the grid when no further progress can be made

Performance Results

Sequoia Benchmarks

Lin. Alg. Blas Level 1 SAXPY, Level 2 SGEMV, and Level 3 SGEMM benchmarks
Conv2D 2D single precision convolution with 9x9 support (non-periodic boundary constraints)
FFT3D Complex single precision FFT
Gravity 100 time steps of N-body stellar dynamics simulation (N^2) single precision
HMMER Fuzzy protein string matching using HMM evaluation (Horn et al. SC2005 paper)

Best available implementations used as leaf task

Single Runtime System Configurations

- Scalar
  - 2.4 GHz Intel Pentium4 Xeon, 16B
- 8-way SMP
  - 4 dual-core 2.66GHz Intel P4 Xeons, 8GB
- Disk
  - 2.4 GHz Intel P4, 160GB disk, ~50MB/s from disk
- Cluster
  - 16, Intel 2.4GHz P4 Xeons, 16B/node, Infiniband interconnect (780MB/s)
- Cell
  - 3.2 GHz IBM Cell blade (1 Cell – 8 SPE), 16B
- PS3
  - 3.2 GHz Cell in Sony Playstation 3 (6 SPE), 256MB (160MB usable)
System Utilization

<table>
<thead>
<tr>
<th>Function</th>
<th>SMP</th>
<th>Disk</th>
<th>Cluster</th>
<th>Cell</th>
<th>PS3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Percentage of Runtime</td>
<td>100</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Multi-Runtime System Configurations

- **Cluster of SMPs**
  - Four 2-way, 3.16GHz Intel Pentium 4 Xeons connected via GigE (80MB/s peak)
- **Disk + PS3**
  - Sony Playstation 3 bringing data from disk (~30MB/s)
- **Cluster of PS3s**
  - Two Sony Playstation 3's connected via GigE (60MB/s peak)

Multi-Runtime Utilization

<table>
<thead>
<tr>
<th>Function</th>
<th>Cluster of SMPs</th>
<th>Disk + PS3</th>
<th>Cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>Percentage of Runtime</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sequoia Summary

- **Problem:**
  - Deep memory hierarchies pose perf. programming challenge
  - Memory hierarchy different for different machines
- **Solution:**
  - Hierarchical memory in the programming model
  - Program the memory hierarchy explicitly
  - Expose properties that affect performance
- **Approach:** Express hierarchies of tasks
  - Execute in local address space
  - Call-by-value-result semantics exposes communication
  - Parameterized for portability