Leveraging Multicore Processors for Scientific Computing

Martin Tillenius
Outline

1. Background and Scope
2. Hardware Transactional Memory
3. Multicore Programming Models
The Multicore Revolution

- Performance improvement for single-core processors limited
- Multicore processors are now the norm
- Requires parallel software

Problem: Parallel programming is hard.
Goal and Scope

Goal

Make parallel programming easy.

Find idioms, building blocks, and programming models to
▶ Increase productivity
▶ Reduce programming mistakes
▶ Facilitate efficient implementations

Scope

▶ Scientific Computing
  ▶ Floating point operations
  ▶ High throughput
▶ Shared Memory, User-Level Software
What Makes Parallel Programming Hard?

**Difficulty:** Synchronization between threads
- Waiting for results
- Atomic updates

**Primitives:**
- Atomic read-modify-write instructions such as
  - Compare-And-Swap, Fetch-And-Add, . . .
- Used to build higher level constructs
  - Locks, Condition variables, Barriers, . . .

New sync constructs could simplify parallel programming
Hardware Transactional Memory

Paper I
What is Hardware Transactional Memory?

Example: Double-Ended Queue

Want to concurrently:

▶ Add elements to end of queue
▶ Remove elements from front of queue

▶ Hard to allow this using locks
▶ Simple and efficient with transactions:

```
BEGIN TRANSACTION
    deque.push_back( element );
END TRANSACTION
```
```
BEGIN TRANSACTION
    element = deque.pop_front();
END TRANSACTION
```

Properties:

▶ No intermediate states observable
▶ Aborted if collisions occur
Why Hardware Transactional Memory?

Transactions are optimistic:
- Handle collisions only when they occur
- Locks always first acquire exclusive access

Also:
- Avoids storing and accessing lock variables

Locks
```
pthread_mutex_t lock_variable;

void f() {
    pthread_mutex_lock( &lock_variable );
    counter = counter + 1;
    pthread_mutex_unlock( &lock_variable );
}
```

Transactions
```
void f() {
    BEGIN TRANSACTION
    counter = counter + 1;
    END TRANSACTION
}
```
We use a prototype of Sun’s (later Oracle’s) Rock processor.

<table>
<thead>
<tr>
<th>New Instructions for Transactional Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>chkpt &lt;fail_addr&gt;</code></td>
</tr>
<tr>
<td><code>commit</code></td>
</tr>
<tr>
<td><code>read %cps, &lt;dest_reg&gt;</code></td>
</tr>
</tbody>
</table>

- `chkpt` starts a transaction
- `commit` ends a transaction
- If the transaction fails, jump to `fail_addr`
- If the transaction fails, the reason is stored in the `cps` register.
Best-effort system:

- Transactions are not guaranteed to succeed
- Possible failure reasons:
  - Conflict, Size, Load, Store, Interrupt, Mispredicted branch, Exception, Floating point division, . . .

When a transaction fails:

- Check why
- If load or store: Load the memory into level 1 cache
- If conflict: Use exponential backoff to avoid congestion
Idea: Use transactions to perform atomic floating point updates.

Scenario: Several threads updates a shared matrix.

Alternatives

- Use locks to protect shared memory
  - Access and store lock variables
  - Always need to assure exclusive access (pessimistic)
- Write to private buffers and merge later
  - Occupy and access more memory
  - Additional merge phase
- Use atomic instructions: compare-and-swap
  - Only available for integers: trick needed

Transactions is a good alternative, if collisions are rare.
Experiment: Single Thread Increases a Variable

Benchmark

variable[0] += delta[0];

- CAS: Move data between FPU and CPU
- Locks: Accesses lock variable, library function calls
- Nothing: Lower bound
Experiment: Several Threads Write to Shared Memory

Benchmark

```c
for (i = 0; i < n; ++i)
    if ((i % freq) == 0) shared += delta;
    else local += delta;
```

- 16 threads updating a single element
- Write to shared memory every \( n \)\(^{th} \) iteration
- Transactions much slower than in previous test (105 ns → 254 ns)
- Only about 15 % of the transactions failed at highest contention
Experiment: \textit{n}-Body Simulation

Benchmark

An \textit{n}-body simulation of 1024 particles interacting pair-wise.

- Updates 4 elements at a time
- Small data set

- Transactions slightly faster than locks
- Compare-and-swap slow since it updates a single element
- Avoiding concurrent updates by far most efficient
- About 0.4 \% of the transactions failed (52 \% conflicts, 34 \% reads)
## Conclusions

**Transactions are:**

- More efficient than locks in all tests
- More efficient than compare-and-swap if several elements can be updated at same time
- Sensitive to memory traffic

It is still best to avoid concurrent updates when possible.
Multicore Programming Models

Paper II and III
POSIX threads (or Windows threads)
- Basic functionality provided by the operating system
- Want higher abstraction level

Fork-Join parallel languages
- OpenMP, Cilk
- Limited to Fork-Join parallel structures

Dependency-Aware Task-Based Systems
- OMP Superscalar, ...
Fork-Join vs General Task Graph

**Fork-Join**

- OpenMP: Loop Parallelism

**OpenMP: Tasks**

- A
- B
- C
- D
- E

**Cilk: Fully-Strict**

- A
- B
- C
- D
- E

**General (OMPSs)**

- 0,0
- 1,0
- 2,0
- 3,0
- 4,0
- 5,0
- 0,1
- 1,1
- 2,1
- 3,1
- 4,1
- 5,1
- 0,2
- 1,2
- 2,2
- 3,2
- 4,2
- 5,2
- 0,3
- 1,3
- 2,3
- 3,3
- 4,3
- 5,3

**General Task Graphs**

17/34
Fork-Join vs General Task Graph

**Fork-Join:**
- Well suited for recursive algorithms
- Does not fit all applications

**Example Execution Traces:**

- **General:**
  - Thread 0: A → B → C → D → E
  - Thread 1: A → B → C → E

- **Fork-Join:**
  - Thread 0: A → B → C → D → E
  - Thread 1: A → B → C → E

See also:
Jakub Kurzak, Hatem Ltaief, Jack Dongarra, and Rosa M. Badia.
Fork-Join vs General Task Graph

Fork-Join:
- Well suited for recursive algorithms
- Does not fit all applications

Example Execution Traces:

General:

Fork-Join:

White = Idle = Bad

See also:
Conclusions

- Fork-Join parallelism is not enough
- Support for general dependencies is important for performance
Expressing Parallelism

Task dependencies can be deduced from data-flow:

taskA(write a);
taskB(read a, write b);
taskC(read a, write c);
taskD(read b);
taskE(read b, read c);

- Programmer writes a sequential program
- Annotates tasks and their inputs and outputs
- Dependencies deduced by run-time system
- Tasks are executed in parallel when possible

Used in several task-based systems: Jade, OMP Superscalar, StarPU, Quark, …
SuperGlue
Our Run-Time System for Task-Based Programming
Motivation

- Test bed for experimenting with task-based programming
- Application driven design to suit our needs

Design Goals

- Performance
- Generality
- Ease-of-use
Programming Model:
- Programmer writes a sequential program
- Specifies tasks, and their inputs and outputs
- Run-time deduces dependencies and executes tasks in parallel

Run-Time System:
- One worker thread per core
- One ready task queue per worker thread
- Task stealing for load balancing
Handles are abstract objects for managing dependencies.

Handle x;
taskA(write x);
taskB(read x);

Handles:
- Represents the shared resource to manage:
  - Block of a matrix
  - Slice of a vector
- No coupling needed between handle and actual resource
  - Run-time system does not need to know the data structure
- Represent abstract resource for constrained scheduling
  - Task cache/memroy usage
Dependency management through Data Versioning:

- Tasks have dependencies on handles, not on other tasks
- Each handle has a *version*
- Each task has a *required version* for each accessed handle

### Example

Handle x;

```plaintext
taskA(write x); // taskA requires x version 0
taskB(read x);  // taskB requires x version 1
```

**Note:** We do **not** keep several versions of data. Versions only used for dependency management.
Example

8 tasks accessing the same handle x:
read x, read x, modify x, add x, add x, add x, modify x

- read
- read

Requires version 0
(Run all at once)

- modify

Requires version 2

- add
- add

Requires version 3
(Any order)
(One at the time)

- modify

Requires version 6

Graph View
(Not a DAG)
Implications:

- Another layer of indirection
  Successors are stored in the handles.
- No global view
  A task only knows the handles it accesses.
  A handle only knows tasks that are waiting.
- No coupling between tasks
  Tasks can be deleted at any time.
  Successors need not be known.

Diagram:

```
A
  /\  \\
 B  C  D
   \  /
    A

Classic

handle
B C D
A

Handles
```
**Scheduling**

- When a task is added its dependencies are checked
- The task is enqueued at first unavailable handle
- When a worker finishes a task, it
  - Increases the handle versions
  - Puts the tasks waiting for the new version in its ready queue

**Tasks will be executed by the thread that produced the data.**
Performance Tests
N-Body simulation: 8192 particles, 256 per block, 16 time steps.
4 x AMD Opteron 6276 = 4 x 8 modules, 1 FPU per module
N-Body simulation: 8192 particles, 256 per block, 16 time steps.
4 x AMD Opteron 6276 = 4 x 8 modules, 1 FPU per module
N-Body simulation: 8192 particles, 256 per block, 16 time steps.
4 x AMD Opteron 6276 = 4 x 8 modules, 1 FPU per module.
Speedup at Small Task Sizes

- Speedup over \#cycles \times \#tasks
- 64,000 tasks, no dependencies, varying number of cycles/task
- Tasks only read clock counter
  (no memory accesses or computations)
Conclusion

Dependency-aware task-based models are:

▶ Efficient
▶ Suitable for a large class of applications
▶ User friendly

Version-driven dependency management has nice properties:

▶ Easy, Efficient, and Flexible
▶ No global view:
  ▶ A task only knows the data (handles) it accesses
  ▶ A handle only knows tasks waiting for it

SuperGlue is an efficient and flexible implementation of this.
Outlook

- Generalize to distributed memory
- Support heterogeneous architectures
- Use to implement real applications
- Compiler front-end to make a nice interface
Thank you!

Questions?
class SparseMatVecTask : public Task<Options> {
private:
    const SparseMatrixCSR &DP;
    MatrixRowMajor &H, &T;

public:
    SparseMatVecTask(const SparseMatrixCSR &DP_,
                     MatrixRowMajor &H_, Handle<Options> &hH,
                     MatrixRowMajor &T_, Handle<Options> &hT)
        : DP(DP_), H(H_), T(T_)
    {
        registerAccess(ReadWriteAdd::read, &hH);
        registerAccess(ReadWriteAdd::add, &hT);
    }

    void run() { /* T(r) += DP(r,c) * H(c); */ }
};

for (size_t r = 0; r < numRows; ++r)
    for (size_t c = 0; c < numCols; ++c)
        tl->addTask( new SparseMatVecTask(DPx[r][c],
                                           H, hH[c],
                                           Tx, hTx[r]) );
Computing Required Versions:

- Handle knows *next-required-version* for each access type.
- When task is added:
  - The task asks the handles for which version to require.
  - The handles update the *next-required-version* for accesses that cannot be reordered.

**Example**

```plaintext
Handle x: next read 0
next write 0
```

```plaintext
taskA(read x); // require x version 0
```

```
Handle x: next read 0
next write 1
```

```plaintext
taskB(read x); // require x version 0
```

```
Handle x: next read 0
next write 2
```

```plaintext
taskC(write x); // require x version 2
```

```
Handle x: next read 3
next write 3
```
**Extensions**

**Possible to define other access types**

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Access Types</strong></td>
</tr>
<tr>
<td><em>read</em>: Reorderable, not exclusive</td>
</tr>
<tr>
<td><em>write</em>: Not reorderable</td>
</tr>
<tr>
<td><em>add</em>: Reorderable, exclusive</td>
</tr>
<tr>
<td><em>mult</em>: Reorderable, exclusive</td>
</tr>
<tr>
<td><em>mult x</em></td>
</tr>
</tbody>
</table>

**Example**

<table>
<thead>
<tr>
<th>Access Types</th>
<th>Example</th>
<th>Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>read</em>: Reorderable, not exclusive</td>
<td><em>read x</em></td>
<td></td>
</tr>
<tr>
<td><em>write</em>: Not reorderable</td>
<td><em>read x</em></td>
<td></td>
</tr>
<tr>
<td><em>concurrent</em>: Reorderable, not exclusive</td>
<td><em>conc x</em></td>
<td></td>
</tr>
<tr>
<td><em>conc x</em></td>
<td><em>conc x</em></td>
<td></td>
</tr>
<tr>
<td><em>write x</em></td>
<td><em>write x</em></td>
<td></td>
</tr>
</tbody>
</table>

**Example**

<table>
<thead>
<tr>
<th>Access Types</th>
<th>Example</th>
<th>Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>read</em>: Reorderable, not exclusive</td>
<td><em>read x</em></td>
<td></td>
</tr>
<tr>
<td><em>write</em>: Not reorderable</td>
<td><em>read x</em></td>
<td></td>
</tr>
<tr>
<td><em>concurrent</em>: Reorderable, not exclusive</td>
<td><em>conc x</em></td>
<td></td>
</tr>
<tr>
<td><em>conc x</em></td>
<td><em>conc x</em></td>
<td></td>
</tr>
<tr>
<td><em>write x</em></td>
<td><em>write x</em></td>
<td></td>
</tr>
</tbody>
</table>

**Graph**

- **read**
- **add**
- **mult**
- **write**
- **conc**
Limitations

**Limitation:** Can only reorder accesses of same type.

**Example:** read, write, sort, sum

<table>
<thead>
<tr>
<th>Can be reordered:</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶ read - read</td>
</tr>
<tr>
<td>▶ read - sum</td>
</tr>
<tr>
<td>▶ sort - sum</td>
</tr>
</tbody>
</table>

**Example**

```
read x
sum x
read x
sort x
write x
```

**Graph**

- Sort must wait for both reads to finish
- Sort need not wait for the sum task
- ⇒ Not enough to count the number of executed tasks

This requires more than one version counter per handle.
Allow exclusive accesses to same handle to run concurrently.

- First task writes directly to destination
- If destination is busy, writes to temporary storage
- Reuse existing temporary storages, if one exists
- Temporary storages are merged:
  - Before executing a task with read access to the handle
  - When attaching a temporary storage and one already exist

Properties

- Use as few buffers as possible
- Allow parallel merge
- Good locality
**Example:** Calculate forces between all pairs of particles.

```c
// for each pair (i, j)
for (int i = 0; i < N; i++)
    for (int j = i+1; j < N; j++)
        force = calcForce(i, j);
    A[i] += force;
    A[j] -= force;
```

- Order does not matter
- Two tasks cannot write to same memory concurrently
The Add Access Type

**Example:** Calculate forces between all pairs of particles.

```c
// for each pair (i, j)
for (int i = 0; i < N; i++)
    for (int j = i+1; j < N; j++)
        force = calcForce(i, j);
        A[i] += force;
        A[j] -= force;
```

- Order does not matter
- Two tasks cannot write to same memory concurrently

**No parallelism**

**Span = 9**

**Add**

**Span = 5**

(Possible execution)
Execution Traces: Benefit of Add Accesses

Write:

Add:

N-body simulation, 8192 particles, 512 per block, 4 time steps.
// Custom handle type to include indices

template<typename Options>
struct MyHandle : public Handle_<Options> {
    size_t i, j;
    void set(size_t i_, size_t j_) { i = i_; j = j_; }
    size_t geti() { return i; }
    size_t getj() { return j; }
};

struct Options : public DefaultOptions<Options> {
    typedef MyHandle<Options> HandleType; // Override handle type
    typedef PrioScheduler<Options> Scheduler; // Override scheduler
    typedef Enable TaskPriorities; // Enable task priorities
};
```cpp
struct gemm : public Task<Options, 3> {
    gemm(Handle<Options> &h1, Handle<Options> &h2, Handle<Options> &h3) {
        // register data accesses to manage, with direction
        registerAccess(ReadWriteAdd::read, &h1);
        registerAccess(ReadWriteAdd::read, &h2);
        registerAccess(ReadWriteAdd::add, &h3);
    }

    void run() {
        Handle<Options> &h1(getAccess(0).getHandle());
        Handle<Options> &h2(getAccess(1).getHandle());
        Handle<Options> &h3(getAccess(2).getHandle());

        double *a(Adata[h1->geti()*DIM + h1->getj()]);
        double *b(Adata[h2->geti()*DIM + h2->getj()]);
        double *c(Adata[h3->geti()*DIM + h3->getj()]);

        double DONE=1.0, DMONE=-1.0;
        dgemm("N", "T", &nb, &nb, &nb, &DMONE, a, &nb, b, &nb, ...
    }

    int getPriority() const { return 0; }
};
```
static void cholesky(const size_t numBlocks) {
    // Start the system
    ThreadManager<Options> tm;

    // Create handles, and set the custom indices
    Handle<Options> **A = new Handle<Options>*[numBlocks];
    for (size_t i = 0; i < numBlocks; ++i) {
        A[i] = new Handle<Options>[numBlocks];
        for (size_t j = 0; j < numBlocks; ++j)
            A[i][j].set(i, j);
    }

    // Main code: Generate tasks
    for (size_t j = 0; j < numBlocks; j++) {
        for (size_t k = 0; k < j; k++)
            for (size_t i = j+1; i < numBlocks; i++)
                tm.addTask(new gemm(A[i][k], A[j][k], A[i][j]), i);

        for (size_t i = 0; i < j; i++)
            tm.addTask(new syrk(A[j][i], A[j][j]), j);

        tm.addTask(new potrf(A[j][j]), j);

        for (size_t i = j+1; i < numBlocks; i++)
            tm.addTask(new trsm(A[j][j], A[i][j]), j);
    }

    tm.barrier();
}
HTM Experiment: FEM Stiffness Matrix Assembly

**Benchmark**

Assembly of the stiffness matrix in a finite element scheme (2154 nodes).

- Two versions: many or few computations per triangle
- Scattered memory accesses spread over large address space

- Transactions best when computation bound
- Compare-and-swap best when memory bound
- Locks slowest: One lock per element used
- About 23 % of the transactions failed, most due to failed reads
N-Body simulation: 8192 particles, 128 per block, 4 time steps.
8 x Xeon X6550 = 8 x 8 cores
N-Body simulation: 8192 particles, 128 per block, 4 time steps.
8 x Xeon X6550 = 8 x 8 cores