Local Parallel Iteration in X10

Josh Milthorpe
IBM Research

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research under Award Number DE-SC0008923.
Summary

**foreach**: a new standard mechanism for local parallel iteration in X10

Efficient pattern of parallel activities

Support for parallel reductions, worker-local data

Speedup comparable with OpenMP and TBB for selected kernels

Composable with X10 APGAS model
The Brass Ring: LULESH

LULESH v2.0
– DoE proxy application representing CFD codes
– Simulates shockwave propagation using Lagrangian hydrodynamics for a single material
– Equations solved using a staggered mesh approximation
– Lagrange Leapfrog Algorithm advances solution in 3 parts
  • Advance node quantities
  • Advance element properties
  • Calculate time constraints
LULESH: Parallel Loops with OpenMP

38 parallel loops like this one:

```c
static inline
void CalcFBHourglassForceForElems(
    Domain &domain,
    Real_t *determ,
    Real_t *x8n, Real_t *y8n, Real_t *z8n,
    Real_t *dvdx, Real_t *dvdy, Real_t *dvdz,
    Real_t hourg, Index_t numElem,
    Index_t numNode)
{
    Index_t numElem8 = numElem * 8;

    #pragma omp parallel for firstprivate(numElem, hourg)
    for (Index_t i2=0; i2<numElem; ++i2) {
        // 200 lines
    }
}
```
LULESH: Scaling with OpenMP

best single-thread time per iteration: 34.64 ms

parallel speedup vs. number of threads

Mesh size: $30^3$
protected def calcFBHourglassForceForElems(
       domain:Domain,
       determ:Rail[Double],
       x8n:Rail[Double], y8n:Rail[Double], z8n:Rail[Double],
       dvdx:Rail[Double], dvdy:Rail[Double], dvdz:Rail[Double],
       hourg:Double) {

    val numElem8 = numElem * 8;

    finish for (i2 in 0..domain.numElem-1) async {
        // 100 lines
    }
}
LULESH: Scaling with X10 Simple Parallel Loop

best single-thread time per iteration: 34.64 ms
Other Application Kernels: Scaling with X10 Simple Parallel Loop

- **DAXPY**
  - Best single-thread time per iteration: 8.09 ms

- **GEMM**
  - Best single-thread time per iteration: 163.20 ms

- **SpMV**
  - Best single-thread time per iteration: 434.11 ms

- **Hourglass**
  - Best single-thread time per iteration: 18.19 ms
Problems with Simple Parallel Loop

High overhead – one activity per iteration

Poor locality – activities dealt to / stolen by worker threads in random order

Cause: loop ordering dependencies

```scala
val complete = new Rail[Boolean](ITERS);
foreach (i in 0..(ITERS-1)) {
  when(complete(i+1));
  compute();
  atomic complete(i) = true ;
}
```
Parallel Iteration with `foreach`

```plaintext
foreach ( Index in IterationSpace ) Stmt
```

body `Stmt` executed for each value of `Index`, making use of available parallelism.

no dependencies between iterations: any reordering or fusing of iterations must be valid

can transform to an efficient pattern of parallel activities

implied finish: all activities created by `foreach` terminate before progressing to next statement
Code Transformations

Parallel iteration to compute DAXPY:

```java
val x:Rail[Double];
val y:Rail[Double];
val alpha:Double;

foreach (i in lo..hi) {
    x(i) = alpha * x(i) + y(i);
}
```
val x: Rail[Double];
val y: Rail[Double];
val alpha: Double;
val body = (min_i: Long, max_i: Long) => {
  for (i in min_i..max_i) {
    x(i) = alpha * x(i) + y(i);
  }
};
val x:Rail[Double];
val y:Rail[Double];
val alpha:Double;
val body = (min_i:Long, max_i:Long) => {
  for (i in min_i..max_i) {
    x(i) = alpha * x(i) + y(i);
  }
};
Foreach.block(lo, hi, body);
val x:Rail[Double];
val y:Rail[Double];
val alpha:Double;
Foreach.block(lo, hi, (min_i:Long, max_i:Long) => {
  for (i in min_i..max_i) {
    x(i) = alpha * x(i) + y(i);
  }
});
Code Transformations: Block

val numElem = hi - lo + 1;
val blockSize = numElem / Runtime.NTHREADS;
val leftOver = numElem % Runtime.NTHREADS;

finish {
    for (var t:Long = Runtime.NTHREADS-1; t > 0; t--) {
        val tLo = lo + t <= leftOver ?
            t*(blockSize+1) : t*blockSize + leftOver;
        val tHi = tLo + ((t < leftOver) ?
                        (blockSize+1) : blockSize);
        async body(tLo..tHi);
    }
    body(0, blockSize + leftOver ? 1 : 0);
}
Code Transformations: Recursive Bisection

```java
static def doBisect1D(lo:Long, hi:Long,
    grainSize:Long,
    body:(min:Long, max:Long)=>void) {
    if ((hi-lo) > grainSize) {
        async doBisect1D((lo+hi)/2L, hi, grainSize, body);
        doBisect1D(lo, (lo+hi)/2L, grainSize, body);
    } else {
        body(lo, hi-1);
    }
}

finish doBisect1D(lo, hi+1, grainSz, body);
```
Parallel Reduction

\[
\text{result:} U = \text{reduce} \ [T, U] \\
\quad ( \text{reducer:} (a:T, b:U) => U, \text{identity:} U ) \\
\text{foreach ( Index in IterationSpace )} \{ \\
\quad \text{Stmt} \\
\quad \quad \text{offer Exp:T;} \\
\};
\]

arbitrary reduction variable computed using
- provided reducer function and
- identity value such that \( \text{reducer} (\text{identity}, x) = x \)
Worker-Local Data

```scala
foreach ( Index in IterationSpace )
    local (
        val l1 = Initializer1;
        val l2 = Initializer2;
    ) {
        Stmt
    }
```

- a lazy-initialized worker-local store
- created with initializer function
- first time worker thread accesses the store, initializer is called to create local copy
Kernels: Dense Matrix Multiplication

```python
foreach ([j,i] in 0..(N-1) * 0..(M-1)) {
    var temp:Double = 0.0;
    for (k in 0..(K-1)) {
        temp += a(i+k*M) * b(k+j*K);
    }
    c(i+j*M) = temp;
}
```
Kernels: Sparse Matrix Vector Multiplication

```scala
foreach (col in 0..(A.N-1)) {
    val colA = A.getCol(col);
    val v2 = B.d(offsetB+col);
    for (ridx in 0..(colA.size()-1)) {
        val r = colA.getIndex(ridx);
        val v1 = colA.getValue(ridx);
        C.d(r+offsetC) += v1 * v2;
    }
}
```
Kernels: Jacobi

```scala
error = reduce[Double](
    (a:Double, b:Double)=>{return a+b;}, 0.0
) foreach (i in 1..(n-2)) {
    var my_error:Double = 0.0;
    for (j in 1..(m-2)) {
        val resid = (ax*(uold(i-1, j) + uold(i+1, j)) +
            ay * (uold(i, j-1) + uold(i, j+1)) +
            b * uold(i, j) - f(i, j))/b;
        u(i, j) = uold(i, j) - omega * resid;
        my_error += resid*resid;
    }
    offer my_error;
}
```
Kernels: LULESH Hourglass Force

```scala
foreach (i in 0..(numElem-1))
local {
  val hourgam = new Array_2[Double](hourgamStore, 8, 4);
  val xd1 = new Rail[Double](8);

  { 
    val i3 = 8*i2;
    val volinv = 1.0 / detem(i2);
    for (i1 in 0..3) {
      ...
      val setHourgam = (idx:Long) => {
        hourgam(idx,i1) = gamma(i1,idx) - volinv * (dvdx(i3+idx) * hourmodx + dvdy(i3+idx) * hourmody + dvdz(i3+idx) * hourmodz);
      };
      setHourgam(0);
      setHourgam(1);
      ...
      setHourgam(7);
    }
  }
}  
```
Experimental Setup

Intel Xeon E5-4657L v2 @ 2.4 GHz:
4 sockets x 12 cores x 2-way SMT = 96 logical cores

X10 version 2.5.2 plus x10.compiler.Foreach and x10.compiler.WorkerLocal

g++ version 4.8.2 (inc. post-compile)

Intel TBB version 4.3 update 4

run each kernel for large number of iterations (100-5000), min. total runtime > 5 sec

mean time over total of 30 test runs
X10 vs. OpenMP and TBB: DAXPY

best single-thread time per iteration: 74.88 ms

Vector size: 50M (double precision)
X10 vs. OpenMP and TBB: Dense Matrix Multiplication

Matrix size: $1000^2$

best single-thread time per iteration: 163.20 ms
X10 vs. OpenMP: Jacobi

best single-thread time per iteration: 4.90 ms

Grid size: $1000^2$
LULESH (full code): X10 vs OpenMP

Mesh size: 30^3

best single-thread time per iteration: 34.64 ms
X10 vs. OpenMP: LULESH Hourglass Force

best single-thread time per iteration: 14.73 ms

Mesh size: $30^3$
## Differences with OpenMP / TBB Parallel Loops

<table>
<thead>
<tr>
<th>Feature</th>
<th>OpenMP</th>
<th>TBB</th>
<th>X10 / foreach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composable with other loops / tasks</td>
<td>✗ Thread explosion</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Load balancing</td>
<td>✓ Dynamic/guided schedule</td>
<td>✓ Work stealing</td>
<td>✓ Work stealing</td>
</tr>
<tr>
<td>Worker-local data</td>
<td>✓ Private clause</td>
<td>✓ enumerable_thread_specific</td>
<td>✓ at(p) async</td>
</tr>
<tr>
<td>Distribution</td>
<td>✗</td>
<td>✗</td>
<td>✓ at(p) async</td>
</tr>
</tbody>
</table>
LULESH (full code): X10 vs OpenMP

best single-thread time per iteration: 34.64 ms

Mesh size: $30^3$
Summary

`foreach` supports efficient local parallel iteration and reduction, is composable with X10's APGAS model, and achieves comparable performance with OpenMP or TBB for selected applications.

Future Work

Further explore composability with `at` / `atomic`

Support for affinity-based scheduling (per TBB)
Additional Material
Comparing Transformations: DAXPY

best single-thread time per iteration: 8.09 ms

Vector size: 5M (double precision)
Comparing Transformations: Dense Matrix Multiplication

best single-thread time per iteration: 163.20 ms

Matrix size: 1000^2
Comparing Transformations: SpMV

best single-thread time per iteration: 434.11 ms
Comparing Transformations: Jacobi

best single-thread time per iteration: 4.89 ms

parallel speedup vs. number of threads

Grid size: $1000^2$
Comparing Transformations: LULESH Hourglass Force

best single-thread time per iteration: 18.19 ms

Mesh size: $30^3$