Enhancing Data Locality for Dynamic Simulations through Asynchronous Data Transformations & Adaptive Control

Bo Wu
Eddy Zheng Zhang
Xipeng Shen

The College of William and Mary
Dynamic Irregular Memory Access

\[ A[i] \quad \text{v.s.} \quad A[p[i]] \]

for\( (i=0; \, i<N; \, ++i) \)
\[ A[i] = \text{compute}(A[i]); \]

for\( (i=0; \, i<n; \, ++i) \)
\[ A[p[i]] = \text{compute}(A[p[i]]); \]

\( A: 0 \quad 1 \quad 2 \quad \ldots \quad N-3 \quad N-2 \quad N-1 \)

\( p: N-1 \quad 1 \quad N/2 \quad \ldots \quad 2 \quad N-2 \quad 0 \)

\( A: 0 \quad 1 \quad 2 \quad \ldots \quad N-3 \quad N-2 \quad N-1 \)
Dynamic Irregular Memory Access


for(i=0; i<N; ++i)
    A[i] = compute(A[i]);

for(i=0; i<n; ++i)
    A[p[i]] = compute(A[p[i]]);

A: 0 1 2 ... N-3 N-2 N-1

p: N-1 1 N/2 ... 2 N-2 0

A: 0 1 2 ... N-3 N-2 N-1

Bo Wu @ William & Mary
Dynamic Irregular Memory Access

\[ A[i] \text{ v.s. } A[p[i]] \]

for (i = 0; i < N; ++i)
\[ A[i] = \text{compute}(A[i]); \]

for (i = 0; i < n; ++i)
\[ A[p[i]] = \text{compute}(A[p[i]]); \]

A:
\[
\begin{array}{cccccccc}
0 & 1 & 2 & \ldots & N-3 & N-2 & N-1 \\
\end{array}
\]

p:
\[
\begin{array}{cccccccc}
N-1 & 1 & N/2 & \ldots & 2 & N-2 & 0 \\
\end{array}
\]

A:
\[
\begin{array}{cccccccc}
0 & 1 & 2 & \ldots & N-3 & N-2 & N-1 \\
\end{array}
\]
Dynamic Irregular Memory Access

A\[i\] v.s. A\[p[i]\]

for(i=0; i<N; ++i)
A\[i\] = compute(A\[i\]);

for(i=0; i<n; ++i)
A\[p[i]\] = compute(A\[p[i]\]);
Dynamic Irregular Memory Access


for(i=0; i<N; ++i)
A[i] = compute(A[i]);

for(i=0; i<n; ++i)
A[p[i]] = compute(A[p[i]]);

Bo Wu @ William & Mary

Monday, March 19, 12
Irregular Memory Access

• However, many dynamic simulation codes have irregular memory access pattern
  ❖ Molecular Dynamics, Computational Fluid Dynamics, Mesh Refinement, ...

• Many interesting papers on this topic
  ❖ [Das+, JPDC’94]
  ❖ [Ding+, PLDI’99]
  ❖ [Han+, TPDS’06]
  ❖ ... ...
for(i=0; i<NITER; ++i)
{
    TransformData();
    IrregularComp();
}

- Transformations can be expensive
  1 RCB transformation > 20 iterations of computation
for(i=0; i<NITER; ++i) 
{
    TransformData();
    IrregularComp();
}

• Transformations can be expensive

1 RCB transformation > 20 iterations of computation
Transformations can be expensive

- 1 RCB transformation > 20 iterations of computation
Key Questions

• Is asynchronous inspector-executor model possible?

• What are the catch and tradeoff?

• With the catch, would the asynchronous still be beneficial? In what scenarios?

• How to get the best of both worlds?
Asynchronous Transformation: The Basic Idea

Core 1  Core 2  Core 3  Core 4

: computation  : transformation
Asynchronous Transformation: The Basic Idea

Core 1  Core 2  Core 3  Core 4

CPU

GPU

: computation  : transformation
Asynchronous Transformation: Challenges

- Data Dependence
- Coordination among heterogeneous threads
- Efficient implementation on a (co-)processor
- Runtime choice between synchronous and asynchronous transformations
Modyn: A benchmark example

for each time step
    if time_to_update()
        IList = update_IList (Location);
    end if

    /* main computation with irreg. references to Location */
    for each (i,j) in IList
        f = calculate_force (Location[i], Location[j]);
        Force[i] += f;
        Force[j] -= f;
    end for

    for each particle i
        Location[i] = update_loc (Location[i], Force[i]);
    end for
end for

Interaction List:
[(1, 20), (62, 70), ..., (9, 88)]
Modyn: A benchmark example

for each time step
    if time_to_update()
        \( IList = \text{update\_IList}(\text{Location}) \);
    end if

/* main computation with irreg. references to Location */

for each \((i, j)\) in \( IList \)
    \( f = \text{calculate\_force}(\text{Location}[i], \text{Location}[j]) \);
    \( \text{Force}[i] += f; \)
    \( \text{Force}[j] -= f; \)
end for

for each particle \( i \)
    \( \text{Location}[i] = \text{update\_loc}(\text{Location}[i], \text{Force}[i]); \)
end for
end for
Modyn: A benchmark example

for each time step
    if time_to_update()
        $IList = update_IList (Location);$  
    end if

/* main computation with irreg. references to Location */

for each (i,j) in $IList$
    $f = calculate_force (Location[i], Location[j]);$
    $Force[i] += f;$
    $Force[j] -= f;$
end for

for each particle i
    $Location[i] = update_loc (Location[i], Force[i]);$
end for
end for
Synchronous Transformation:
Work Flow

for each time step
  if time_to_update()
    \( \text{IList} = \text{update}_{-}\text{ IList} (\text{Location}); \)
    dataTrans(\text{IList}, \text{Location}, \text{Force});
  end if
  ...
end for
Dependence Graph
Dependence Graph

app. \rightarrow \text{ref} \rightarrow \text{clue} \rightarrow \text{ref} \rightarrow \text{target} \rightarrow \text{ref} \rightarrow \text{clue} \rightarrow \text{trans.}
Dependence Graph

\[ \text{app.} \rightarrow \text{trans.} \rightarrow \text{target} \rightarrow \text{ref} \rightarrow \text{clue} \]

\[ \text{app.} \rightarrow \text{analysis} \rightarrow \text{reloc.} \rightarrow \text{ref} \rightarrow \text{clue} \]

\[ \text{ref} \rightarrow \text{target} \rightarrow \text{ref} \rightarrow \text{clue} \]

: strict dependence

: relaxable dependence

Monday, March 19, 12
Dependence Graph

- app.
- trans.
- ref target & ref clue
- ref clue

Decomposition

- app.
- analysis
- reloc.
- ref target
- ref clue
- ref target & ref clue
- obsolete ref clue
- new order

↑: strict dependence
↓: relaxable dependence

Bo Wu @ William & Mary
Dependence Graph

![Dependence Graph Diagram]

- **app.**
- **trans.**
- **ref**
- **target**
- **clue**

### Decomposition

- **app.**
- **analysis**
- **reloc.**
- **ref**
- **target**
- **new order**
- **obsolete ref clue**

**RCB:**

- 8
- 2

: strict dependence

: relaxable dependence
The Framework

Master Thread

IList needs update?

YES

update IList

NO

new order comes?

YES

NO

update particles

data reposition

Helper Thread
The Framework

Master Thread

- **IList needs update?**
  - YES: update IList
  - NO: Helper Thread

Helper Thread

- new order comes?
  - YES: data reposition
  - NO: update particles

---

Bo Wu @ William & Mary

Monday, March 19, 12
The Framework

Master Thread

- **IList needs update?**
  - YES: update IList
  - NO: new order comes?
    - YES: update particles
    - NO: data reposition

Helper Thread

- transformation analysis

Bo Wu @ William & Mary
The Framework

Master Thread

- **IList needs update?**
  - YES: update IList
  - NO: new order comes?
    - YES: transformation analysis
    - NO: update particles

Helper Thread

- **new order comes?**
  - YES: new order
  - NO: data reposition
Thread Coordination

- **Init** to **Busy**: prog start
- **Busy** to **Ready**: analysis finished
- **Ready** to **Done**: state check
- **Done** to **Init**: ref. clue updated

**State Changes**:
- **DReady** to **Dirty**: analysis finished
- **Dirty** to **DReady**: ref. clue updated

**Thread Roles**:
- **<M>**: Master thread changes the state
- **<H>**: Helper thread changes the state

---

Monday, March 19, 12
Tradeoff of Asynchronous Transformation

• Pros:
  ❖ moves most of transformation overhead off critical path
  ❖ enables more frequent transformations

• Cons:
  ❖ May uses obsolete reference clue
Adaptive Control

• Asynchronous data transformation may not be always better than synchronous transformation

• Need runtime selection

• Collect timing information at the beginning

• Emulate both methods
  ❖ iteration time is stable
  ❖ transformation time is stable
  ❖ transformation benefit is stable
Offloading to GPU

• A new way for CPU-GPU collaboration
  • GPU’s high throughput enables faster transformation
  • Avoiding difficulty of porting irregular code to GPU

• Challenge
  • Previous data-layout algorithms are ill-fit to GPU
TLayout: The Algorithm
TLLayout: The Algorithm

1. Seed planting
   
   - Randomly select K nodes as seeds
TLayout: The Algorithm

1. Seed planting
   - Randomly select K nodes as seeds

2. Membership Propagation
   - Assign neighbors’ membership field if NULL
TLlayout: The Algorithm

1. Seed planting
   ▶ Randomly select K nodes as seeds

2. Membership Propagation
   ▶ Assign neighbors’ membership field if NULL

3. Loop
   ▶ Repeat step 2 until some requirement is satisfied
TLayout: The Algorithm

1. Seed planting
   - Randomly select K nodes as seeds

2. Membership Propagation
   - Assign neighbors’ membership field if NULL

3. Loop
   - Repeat step 2 until some requirement is satisfied

4. Layout
   - Reposition data according to resulting clusters
TLayout: The Algorithm

1. Seed planting
   • Randomly select K nodes as seeds

2. Membership Propagation
   • Assign neighbors’ membership field if NULL

3. Loop
   • Repeat step 2 until some requirement is satisfied

4. Layout
   • reposition data according to resulting clusters

• In essence, it is a clustering algorithm.
Experiment Setting

• Platform
  ❖ AMD Opteron 2216
  ❖ NVIDIA Tesla S1070
  ❖ GCC 4.3.2 (-O3) and CUDA 3.0

• Benchmarks
  ❖ MOLDYN, IRREG, NBF, CFD and MESH

• Data transformation algorithm on CPU
  ❖ RCB (Recursive Coordinate Bisection)

• Data transformation algorithm on GPU
  ❖ TLayout
Optimal Transformation Frequency

• Recall the dilemma between overhead and benefit

Benchmark: IRREG

Method: runtime profiling
Optimal Transformation Frequency

• Recall the dilemma between overhead and benefit

Benchmark: IRREG

Method: runtime profiling

Marin+ : SciDAC’08
Results: Sequential Run

- Speedup
- Iterations Per Update
- synch cpu
- asynch cpu
- asynch gpu
- adaptive selection

IRREG, NBF, MOLDYN, CFD, MESH
Results: Sequential Run

![Graph showing speedup comparison for different applications and runs]

- IRREG
- NBF
- MOLDYN
- CFD
- MESH

Legend:
- synch cpu
- async cpu
- async gpu
- adaptive selection

Iterations Per Update

Monday, March 19, 2012
Results: Sequential Run

![Graph showing speedup comparison for different iterations and applications]

- **Speedup**
  - synch cpu
  - async cpu
  - async gpu
  - adaptive selection

- **Applications**
  - IRREG
  - NBF
  - MOLDYN
  - CFD
  - MESH

- **Iterations Per Update**
  - 10
  - 20
  - 30
TLayout Performance

Normalized Average Iteration Time

IRREG  NBF  MOLDYN  CFD  MESH

RCB
TLayout
TLayout Performance

TLayout is 2.8 to 3.3 times faster than RCB.
TLayout: Propagation Speed

- 5000 nodes randomly selected from more than 440,000 nodes (map 3D to 2D)
TLayout: Propagation Speed

• 5000 nodes randomly selected from more than 440'000 nodes (map 3D to 2D)

First propagation
TLayout: Propagation Speed

- 5000 nodes randomly selected from more than 440,000 nodes (map 3D to 2D)

First propagation    Second propagation
TLayout: Propagation Speed

- 5000 nodes randomly selected from more than 440,000 nodes (map 3D to 2D)
TLayout: Propagation Speed

• The reason
Results: Parallel Run

Interesting result: 3 working threads + 1 transformation thread is faster than 4 working threads.
Contributions

• Is asynchronous inspector-executor model possible?
  ❖ Data dependence circumvention

• What are the catch and tradeoff?
  ❖ Obsolete reference clue
  ❖ Offloading transformation cost from critical path

• With the catch, would the asynchronous still be beneficial?
  In what scenarios?
  ❖ Heavy transformation comparable to kernel execution

• How to get the best of both worlds?
  ❖ Adaptive control
Thanks!