Microsoft Research

Each year Microsoft Research hosts hundreds of influential speakers from around the world including leading scientists, renowned experts in technology, book authors, and leading academics, and makes videos of these lectures freely available.

2016 © Microsoft Corporation. All rights reserved.
Collaborative Computing Science Research: From Algorithms to Application Impact

Sriram Krishnamoorthy

Pacific Northwest National Laboratory
Compile-time and Runtime Transformations for HPC

Funding: DOE ASCR, DOE BES, DOE ECP, DARPA, IARPA, PNNL LDRD, ...
Focus Areas

- Runtime optimization and automation
- Compile-time transformation/optimization
- Resilience (error detection and recovery)
- Correctness checking
- Application impact
Runtime Systems

- Global Arrays
- Tensor Algebra for Many-body Methods (TAMM)
- Task-parallel runtime systems
- Shared-memory and distributed memory runtimes
- Load balancing, communication, and memory management
- Automation with potential user control
Resilience

► Compile-time techniques
  ▪ Vulnerability analysis
  ▪ Error propagation analysis
  ▪ Error detection

► Runtime techniques
  ▪ Error localization
  ▪ Localized rollback
  ▪ Scalable recovery
Correctness

- Builds on fault tolerance research and work on compile-time analysis
- Equivalence checking
- Static analysis to identify program invariants
Interdisciplinary Research: Hammering the Nails among the Screws
Polyhedral Analysis ➔ Resilience and Correctness

- Typically used for loop optimization
  - We employed it in different contexts.

- Detecting soft errors impacting the memory subsystem [PLDI’14]

- Checking correctness of an optimized code [POPL’16]

- Cache behavior modeling for affine programs [POPL’18]
  - Toward accurate compile-time cache vulnerability analysis
Combinatorial Algorithms
Scalable Error Recovery

- Resilient data distributions using graph matching [CF’11]

- Approximation strategies to optimize memory access for page rank and graph matching [HiPC’17]
Numerical Error Analysis →
Efficient and Sound Error Detection

▶ Reason about accuracy of a sequence of operations

▶ Numerically sound error detectors for specific program classes

▶ Optimize overhead to account for error propagation
End Goal: Application Impact
Application Areas

- Computational chemistry
  - Long history of collaboration
  - NWChem, NWChemEx, SPEC, etc.

- Computational biology (genome assembly)

- Power grid

- ...

- Software releases, joint publications, joint projects in CS/ECP and applied offices, etc.
NWChem Overview

NWChem is suite of methodologies for computational chemistry

- High Accuracy Methods → MP, CC, EOMCC, MRCC
  - Ground & Excited States
- Gaussian-based DFT/TDDFT
  - Ground & Excited States, Optimization, Properties (NMR, EFG, EPR, linear response, UV/Vis, XAS...)
- Plane wave based DFT
  - Car-Parinello MD (CPMD), Band Structure, Optimization, Free Energy...
- Molecular Dynamics, Molecular Mechanics
- Integrated Methodologies → QM/MM
- Workstations → Supercomputers

Open source since Oct 2010
Educational Community License 2.0

www.nwchem-sw.org
Examples of peta-scale simulations with NWChem

- Gordon-Bell finalist 2009
- Cray XT5 CCSD(T) run

Processor Groups (Schematic)

- Group A
- Group B
- Group C

- Time per iteration (sec)
- Number of cores

- SC11
- Pentacene C_{22}H_{14}

Simulations performed on NERSC’s Kasper Cray-XT5 system

- SC14

- Number of Xeon Phi and CPU threads

- Intel Xeon Phi Core

- Calculating disposal on silicon implementation
Reasoning About Program Properties Using Polyhedral Analysis

Joint work with:
Wenlei Bao, Sanket Tavarageri, Louis-Noel Pouchet, Fabrice Rastello, and P. Sadayappan
Correctness Checking

- How do we check if an optimized code is correct?
  - Compare against a reference implementation

- Typical approaches:
  - Run both versions
    - How to choose the input data set?
  - Compare semantics of memory reference traces
    - Need to deal with large trace sizes
  - Statically prove equivalence
    - Restricted to some transformations

Our approach: compare with specification in the form of an affine loop nest
Transformation Examples

for (i=0; i<N; i++)
/*S:*/ A[i] = B[i];

(a) Original

for (i=0; i<N/32; i++)
for (j=0; j<32; j++)
A[i*32+j] = B[i*32+j];
for (j=(N/32)*32; j<N; j++)
A[j] = B[j];

(b) Constant sectioning

i=0; do { //N and T are coprime
A[i] = B[i];
i = (i+T)%N;
} while (i!=0);

(c) Round-robin

for (i=0; i<N; i++)
for (j=idx[i]; j<idx[i+1]; j++)
A[j] = B[j];

//0=idx[0]<idx[1]<...<idx[M]=N

(e) Irregular sectioning

for (i=0; i<N; i++)
for (j=idx[i]; j<idx[i+1]; j++)
A[j] = B[j];

(d) Parametric sectioning

do fn(lo, hi)
if (lo>hi) return;
if (lo==hi) A[lo] = B[lo];
else {
  fn(lo, (lo+hi)/2);
  fn((lo+hi)/2+1, hi);
}
/*Call fn*/ fn(0, N-1);

(f) Recursion
Transformation Examples

(a) Original

(b) Constant sectioning

(c) Round-robin

(d) Parametric sectioning

(e) Irregular sectioning

All above examples “move” operations around. They are iteration-reordering transformations
Observation: iteration-reordering transformations do not change the order in which array elements are updated.

Data accesses in the transformed program match those in the input affine program.

Transformed program operation writing the $i$-th version of an array element = Input program statement instance writing the $i$-th version of the same element
Modeling LRU Cache Behavior of Affine Loops

```
for (i=0; i<2; i++)
  for (j=0; j<3; j++)

for (i=0; i<2; i++)
  for (j=0; j<3; j++)
    Register r1, r2, r3;
    r1 = A[i][j+1];
    r2 = A[i+1][j];
    r3 = r1 + r2;
    A[i][j] = r3;
```

- Typical approach: simulation and trace analysis
- Exact modeling of caches
  - Reads/Writes to cache lines
  - Cold misses when a address line is first accessed
  - Other misses (capacity/conflict)
  - Dirty evictions of cache lines to next level of cache or to memory
  - Cache state at any point in time (including recency count, etc.)
Challenges

- Cache parameters
  \[ \text{Line}[\text{addr}] = \text{addr} / \text{line}\_size \]

- Data layout
  \[ \text{addr}(A[i][j]) = \text{addr}(A) + i \times N + j \]
Challenges

- Cache parameters
  \[ \text{Line}[\text{addr}] = \text{addr} / \text{line}\_size \]

- Data layout
  \[ \text{addr}(A[i][j]) = \text{addr}(A) + i \times j \]
Challenges

- Cache parameters
  \[ \text{Line}[\text{addr}] = \text{addr} / \text{line size} \]

- Data layout
  \[ \text{addr}(A[i][j]) = \text{addr}(A) + i \times \text{dim} + j \]

We have an affine polyhedral problem!
Modeling Set-Associative Caches

- A cache line $c_l$ is evicted from a k-way cache before next use if:
  - There are $\geq k$ distinct accesses to the same cache set as $c_l$ after the time point at which $c_l$ is accessed but before the next access to $c_l$.

- Avoid cardinality computation.

- Incrementally compute miss sets:
  - Misses in k-way cache from calculation for (k-1)-way cache.
Tracking and Constraining Work-stealing Schedulers

Joint work with:
Jonathan Lifflander, Laxmikant Kale
Task-Parallel Abstractions

- Finer specification of concurrency, data locality, and dependences
  - Convey more application information to compiler and runtime

- Adaptive runtime system to manage tasks

- Application writer specifies the computation
  - Writes optimizable code

- Tools to transform code to generate an efficient implementation
Random Work Stealing

- Simple model
  - Idle worker steals oldest continuation from a victim worker
  - Asymptotic parallel efficiency

- Automatically adapt to changes in the execution environment

- Widespread adoption

- Limitations: Lack of memory
  - Oblivious to locality across program phases
  - Significant time distributing work in iterative programs
Objective

- Trace execution
- Exploit trace to perform various optimizations
- Constrain the scheduler to obtain desired behavior
Space Overhead: Shared Memory

Small trace sizes, less affected by core count or problem size
Space Overhead: Shared Memory

Small trace sizes, less affected by core count or problem size
Space Overhead: Distributed Memory

Still less than 160MB in total on 32000 cores
Time Overhead: Shared Memory

Time overhead within variation in execution time
Time Overhead: Distributed Memory

Time overhead within variation in execution time
Replay Schedulers

- **Strict, ordered replay (StOWS)**
  - Exactly reproduce the template schedule
  - Donation of continuations to be stolen

- **Strict, unordered replay (StUWS)**
  - Reproduce the template schedule, but allow the order to deviate (respecting the application’s dependencies)

- **Relaxed work-stealing replay (RelWS)**
  - Reproduce the template schedule as much as possible, but allow workers to deviate when they are idle, by further stealing work
How good are the schedulers?

Relaxed work stealing incurs some overhead because it combines replay and work stealing.
Relaxed Work Stealing: Adaptability I

Slow down one out of 80 workers 4 times

![Graph showing execution time comparison between fib(48) StOWS and fib(48) RelWS with slow worker]
Relaxed Work Stealing: Adaptability II

Relaxed replay of schedule from \((p-10)\) workers on \(p\) workers

![Graph showing execution time for different values of \(p\)]
Relaxed Work Stealing: Adaptability III

Relaxed work stealing of \( \text{fib}(54) \) with a schedule from \( \text{fib}(48) \)
What can we do with a steal tree?
Visualization

- Core utilization plot over time
- Cilk LU benchmark on 24 cores
- Trace size <100KB
Retentive stealing stabilizes stealing costs
Similar trends on all systems
Constrained Schedules in OpenMP

```c
#pragma omp parallel for schedule(static)
for (i = 0; i < size; i++)
    A[i] = B[i] = 0; //init
#pragma omp parallel for schedule(static)
for (i = 0; i < size; i++)
    B[i] = A[i]; //memcpy
```

<table>
<thead>
<tr>
<th>A</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>4</th>
<th>4</th>
<th>4</th>
<th>5</th>
<th>5</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Loops are naturally matched, leading to good performance
Constrained Schedules in OpenMP

```c
#pragma omp parallel for schedule(static)
for (i = 0; i < size; i++)
    A[i] = B[i] = 0; //init

#pragma omp parallel for schedule(static)
for (i = 0; i < size; i++)
    B[i] = A[i]; //memcpy
```

| A | 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5 |
| B | 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5 |

Empirical study
- Parallel memory copy of 8GB of data, using OpenMP schedule static
- 80-core system with eight NUMA domains, first-touch policy
- Execution time: 169ms
Cilk Scheduling

```
cilk_for (i = 0; i < size; i++)
    A[i] = B[i] = 0;  //init

cilk_for (i = 0; i < size; i++)
    B[i] = A[i];  //memcpy
```

A
---
5 2 3 1 2 4 3 4 5 1 5 2 3 1 4

B
---
5 2 3 1 2 4 3 4 5 1 5 2 3 1 4

memcpy, thread
---
3 4 5 2 1 3 1 4 2 5 1 3 5 2 4

Random work stealing mismatches the initialization and subsequent use, causing performance degradation.
Cilk Scheduling

cilk_for (i = 0; i < size; i++)
    A[i] = B[i] = 0; //init
    B[i] = A[i]; //memcpy

<table>
<thead>
<tr>
<th>A</th>
<th>5 2 3 1 2 4 3 4 5 1 5 2 3 1 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>5 2 3 1 2 4 3 4 5 1 5 2 3 1 4</td>
</tr>
<tr>
<td>thread</td>
<td>3 4 5 2 1 3 1 4 2 5 1 3 5 2 4</td>
</tr>
</tbody>
</table>

Empirical study

- Parallel memory copy of 8GB of data, using MIT Cilk or OpenMP 3.0 tasks
- Execution time: 436ms (Cilk/OMP task) vs 169ms (OMP static)
Can we constrain the work stealing scheduler to improve NUMA locality?
Evolving a Schedule

Capture an application phase's steal tree

Is load balanced?

Yes → Strict ordered replay

No → Data localization

Load imbalance observed?

No → Adapt schedule using relaxed work stealing

Yes → Strict ordered replay
Observations

- Work stealing can be adapted to construct locality-aware schedules
- Data reorganization cost needs to be amortized across multiple iterations
- Support to effectively specify schedules for Cilk programs can be useful