Compiler Transformations for High-Performance Computing (2)

Presented by
Jason Pazis and Yi Zhang

March 25, 2010

Procedure call transformations (1)

- Leaf procedure optimization
  - Won't call other procedures
- Cross-call register allocation
- Parameter promotion
  - Advantage:

Procedure call transformations (1)

- Leaf procedure optimization
  - Won't call other procedures
  - No need to save/restore return address register
  - No need to allocate stack frame
- Cross-call register allocation
- Parameter promotion
  - Advantage:

Procedure call transformations (1)

- Leaf procedure optimization
  - Won't call other procedures
  - No need to save/restore return address register
  - No need to allocate stack frame
- Cross-call register allocation
  - Caller can use registers which callee won't use
- Parameter promotion
  - Advantage:

Procedure call transformations (1)

- Leaf procedure optimization
  - Won't call other procedures
  - No need to save/restore return address register
  - No need to allocate stack frame
- Cross-call register allocation
  - Caller can use registers which callee won't use
- Parameter promotion
  - Used when parameter is passed by reference
  - Unmodified $\rightarrow$ pass by value; modified $\rightarrow$ pass by value-return
  - Advantage:

Procedure call transformations (1)

- Leaf procedure optimization
  - Won't call other procedures
  - No need to save/restore return address register
  - No need to allocate stack frame
- Cross-call register allocation
  - Caller can use registers which callee won't use
- Parameter promotion
  - Used when parameter is passed by reference
  - Unmodified $\rightarrow$ pass by value; modified $\rightarrow$ pass by value-return
  - Advantage: can use registers instead of LOAD/STORE

Procedure call transformations (2)

- Procedure inlining
  - What about recursive procedures?
  - Advantages:
  - Disadvantages:
- Procedure cloning (grouped into specialized versions)
- Loop pushing
- Tail recursion elimination
  - When is it not applicable?
- Function Memoization
  - When is this useful?

Procedure call transformations (2)

- Procedure inlining
  - What about recursive procedures?
  - Advantages:
    - No separate stack frame allocation
    - No transfer of control (better cache behavior)
    - Improves compiler analysis and optimization
    - Cheaper than interprocedural analysis
  - Disadvantages:
- Procedure cloning (grouped into specialized versions)
- Loop pushing
- Tail recursion elimination
  - When is it not applicable?
- Function Memoization
  - When is this useful?

Procedure call transformations (2)

- Procedure inlining
  - What about recursive procedures?
  - Advantages:
    - No separate stack frame allocation
    - No transfer of control (better cache behavior)
    - Improves compiler analysis and optimization
    - Cheaper than interprocedural analysis
  - Disadvantages:
    - Increases code size
    - May increase cache misses
- Procedure cloning (grouped into specialized versions)
- Loop pushing
- Tail recursion elimination
  - When is it not applicable?
- Function Memoization
  - When is this useful?

Procedure call transformations (2)

- Procedure inlining
  - What about recursive procedures?
  - Advantages:
    - No separate stack frame allocation
    - No transfer of control (better cache behavior)
    - Improves compiler analysis and optimization
    - Cheaper than interprocedural analysis
  - Disadvantages:
    - Increases code size
    - May increase cache misses
- Procedure cloning (grouped into specialized versions)
- Loop pushing
- Tail recursion elimination
  - When is it not applicable? C++
- Function Memoization
  - When is this useful?

Procedure call transformations (2)

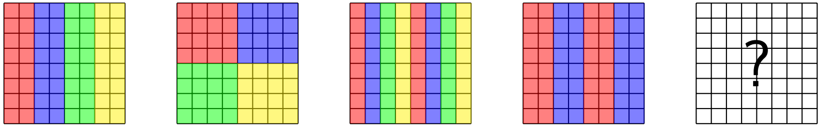
- Procedure inlining
  - What about recursive procedures?
  - Advantages:
    - No separate stack frame allocation
    - No transfer of control (better cache behavior)
    - Improves compiler analysis and optimization
    - Cheaper than interprocedural analysis
  - Disadvantages:
    - Increases code size
    - May increase cache misses
- Procedure cloning (grouped into specialized versions)
- Loop pushing
- Tail recursion elimination
  - When is it not applicable? C++
- Function Memoization
  - When is this useful? side-effect free callee, expensive computation, limited parameter configuration

Transformations for parallel machines

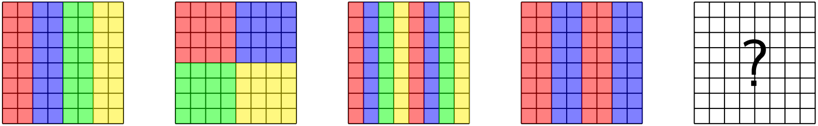
- Automatic parallelization of sequential code is hard
- Some compilers support explicit directives
  - Examples: HPF, OpenMP, ...

```
#pragma omp parallel for
for (int i=0; i<n; i++)
    c[i] = a[i] + b[i];
```

Regular array decomposition



Regular array decomposition



Serial Block Cyclic-serial Block-cyclic

- Decomposition based on load balancing
- Needs to consider together with locality/communication

Other parallelization techniques based on data layout

- Scalar privatization
- Array privatization
- Cache alignment

Automatic decomposition and alignment

- Decomposition: how array elements are distributed across a set of processors
- Alignment: which elements go onto each processor
- Goals: maximize parallelism and minimize communication
- Approaches
  - Manual: e.g., BLOCK and CYCLIC in HPF
  - Automatic: represent program behavior (e.g., communication) so that it can be reasoned and computed

Automatic global optimization for parallelism and locality [Anderson & Lam 1993]

- Trade-off between parallelism and locality
- Target machines: both distributed and shared address space
- Domain
  - Dense matrix code: loop bounds and array subscripts are affine functions of loop indices and symbolic constants
  - Across multiple loop nests
  - #iterations $\gg$ #processors
- Objective: find first-order, or "shape" of data and computation decomposition

Example

```
forall i=0 to N do
  forall j=0 to N do
    Y[i,N-j] += X[i,j];
forall i=1 to N do
  for j=1 to N do
    Z[i,j] = Z[i,j-1]+Y[j,i-1];
```

Problem formulation

- Given a loop nest of depth l, with loop bounds being affine functions of the loop indices, an iteration space $\mathcal{I}$ is defined
- Given an m-dimensional array, an array space $\mathcal{A}$ is defined
- Given an n-dimensional processor array, a processor space $\mathcal{P}$ is defined

Problem formulation

- Given a loop nest of depth l, with loop bounds being affine functions of the loop indices, an iteration space $\mathcal{I}$ is defined
- Given an m-dimensional array, an array space $\mathcal{A}$ is defined
- Given an n-dimensional processor array, a processor space $\mathcal{P}$ is defined
- Data decomposition modeled as a function $d(\mathbf{a}) : \mathcal{A} \rightarrow \mathcal{P}$, where $d(\mathbf{a}) = D\mathbf{a} + \delta$
- Computation decomposition modeled as a function $c(\mathbf{i}) : \mathcal{I} \rightarrow \mathcal{P}$, where $c(\mathbf{i}) = D\mathbf{i} + \gamma$

Problem formulation

- Given a loop nest of depth l, with loop bounds being affine functions of the loop indices, an iteration space $\mathcal{I}$ is defined
- Given an m-dimensional array, an array space $\mathcal{A}$ is defined
- Given an n-dimensional processor array, a processor space $\mathcal{P}$ is defined
- Data decomposition modeled as a function $d(\mathbf{a}) : \mathcal{A} \rightarrow \mathcal{P}$, where $d(\mathbf{a}) = D\mathbf{a} + \delta$
- Computation decomposition modeled as a function $c(\mathbf{i}) : \mathcal{I} \rightarrow \mathcal{P}$, where $c(\mathbf{i}) = D\mathbf{i} + \gamma$
- Objective: find $c(\mathbf{i})$ for each loop nest and $d(\mathbf{a})$ for each array in each loop nest, s.t. parallelism is maximized and communication is minimized

Solution to example

```
forall i=0 to N do
  forall j=0 to N do
    Y[i,N-j] += X[i,j];
forall i=1 to N do
  for j=1 to N do
    Z[i,j] = Z[i,j-1]+Y[j,i-1];
```

- $d_X(\mathbf{a}) = [0\ 1]\mathbf{a} + [0]$
- $d_Y(\mathbf{a}) = [0\ \ -1]\mathbf{a} + [N]$
- $d_Z(\mathbf{a}) = [-1\ 0]\mathbf{a} + [N+1]$
- $c_1(\mathbf{i}) = [0\ 1]\mathbf{i} + [0]$
- $c_2(\mathbf{i}) = [-1\ 0]\mathbf{i} + [N+1]$

Basic concepts

Solving the problem in three steps

1. Partition
   - Collocate data and computation
   - Described by the null spaces of D and C
2. Orientation
   - Determine the orientation of axes of each space
   - Described by D and C
3. Displacement
   - Determine the offsets of starting position of data and computation
   - Described by δ and γ

What about communication?

- Condition of no communication: $D_x(f_{xj}(\mathbf{i})) + \delta = C_j(\mathbf{i}) + \gamma$
  - Maximizing parallelism means minimizing the nullspace of C
- Allowing pipelined communication (with single loop nest)
  - Solvable by an extension to the no communication case
- Allowing data reorganization communication (due to mismatch of decompositions for multiple loop nests)
  - Modeled using communication graphs
  - Dynamic decomposition is NP-hard

Exposing coarse-grained parallelism

Identify big chunks of computation with no or little communication

- Procedure call parallelization
  - Perform a call as an independent, parallel task
- Split
  - Split some iterations of one loop off so that they are independent of some other loop
- Graph partitioning
  - Data-flow graphs: computation as nodes, communication as edges
  - Individual nodes often too small as unit of scheduling
  - Common approach: dynamic scheduling + task merging (e.g., Dryad)

Computation partitioning

Original code

```
for i=1,n
  do a[i]
  do b[i]
end for
```

Guard introduction

```
for i=1,n
  if i in my range
    do a[i]
  if i in my range
    do b[i]
end for
```

Redundant guard elimination

```
for i=1,n
  if i in my range
    do a[i]
    do b[i]
  end if
end for
```

Bounds reduction

```
for i in my range
  do a[i]
  do b[i]
end for
```

Communication optimization

- Cost model: startup time $+$ per-element cost $\times$ #elements
  - Implication: prefer one large message than multiple small ones
- Techniques
  - Message vectorization
  - Message coalescing
  - Message aggregation
  - Collective communication
  - Message pipelining
  - Redundant communication elimination

Transformations for specific architectures

- VLIW
  - Requires more parallelism than in basic blocks
  - Trace scheduling can be helpful
- SIMD
  - Has regular interconnection network
  - Optimization based on *multistencils*
  - Optimization based on alignment preference graphs

Automatic data allocation to minimize communication on SIMD machines [Knobe & Natarajan 1993]

- SIMD computers
  - Many processing elements (PEs), each with its own local memory
  - Each instruction executed by a subset of the PEs
  - Two kinds of communication between PEs: *router* and *grid*

Automatic data allocation to minimize communication on SIMD machines [Knobe & Natarajan 1993]

- SIMD computers
  - Many processing elements (PEs), each with its own local memory
  - Each instruction executed by a subset of the PEs
  - Two kinds of communication between PEs: *router* and *grid*
- Q: how to allocate arrays to maximize parallelism and minimize communication?

Automatic data allocation to minimize communication on SIMD machines [Knobe & Natarajan 1993]

- SIMD computers
  - Many processing elements (PEs), each with its own local memory
  - Each instruction executed by a subset of the PEs
  - Two kinds of communication between PEs: *router* and *grid*
- Q: how to allocate arrays to maximize parallelism and minimize communication?
- Key concepts: alignment by usage, layout preferences, dynamic alignment

Canonical allocation

- Allocation function: maps a array element to a PE where it is stored
- Canonical allocation
  - Fixed mapping for each array for its entire lifetime
  - Each read/write goes to the PE where it is stored
  - Why not go dynamic?

Allocation driven by usage—preferences

- Example:

```
temp(1:N) = A(J,1:N)
A(J,1:N)  = B(J,1:N)
B(J,1:N)  = temp(1:N)
```

Allocation driven by usage—preferences

- Example:

```
temp(1:N) = A(J,1:N)
A(J,1:N)  = B(J,1:N)
B(J,1:N)  = temp(1:N)
```

- Identity preference
  - True dependency; identical alignment of array elements for the two references
  - Honored by identical allocation functions

Allocation driven by usage—preferences

- Example:

```
temp(1:N) = A(J,1:N)
A(J,1:N)  = B(J,1:N)
B(J,1:N)  = temp(1:N)
```

- Identity preference
  - True dependency; identical alignment of array elements for the two references
  - Honored by identical allocation functions
- Conformance preference
  - Relates two array sections in the same operation
  - Honored by choosing alignment functions that produce the same results

Allocation driven by usage—preferences

- Example:

```
temp(1:N) = A(J,1:N)
A(J,1:N)  = B(J,1:N)
B(J,1:N)  = temp(1:N)
```

- Identity preference
  - True dependency; identical alignment of array elements for the two references
  - Honored by identical allocation functions
- Conformance preference
  - Relates two array sections in the same operation
  - Honored by choosing alignment functions that produce the same results
- Handling conflicts
  - Unhonor some preferences and compensate with data motion
  - Decrease parallelism

Unhonored preferences and data motion

- Unhonored conformance preferences
- Unhonored identity preferences

Unhonored preferences and data motion

- Unhonored conformance preferences
  - Addressed by inserting data motion local to the operation
- Unhonored identity preferences

Unhonored preferences and data motion

- Unhonored conformance preferences
  - Addressed by inserting data motion local to the operation
- Unhonored identity preferences
  - Complicated by branches and loops
  - A single definition may reach multiple uses; a single use may be reached by multiple definitions

Unhonored preferences and data motion

- Unhonored conformance preferences
  - Addressed by inserting data motion local to the operation
- Unhonored identity preferences
  - Complicated by branches and loops
  - A single definition may reach multiple uses; a single use may be reached by multiple definitions
- Using canonical allocation?
  - Easy to implement
  - Compiler has the freedom to choose the location of single allocation
  - But may incur a lot of communication

Unhonored preferences and data motion

- Unhonored conformance preferences
  - Addressed by inserting data motion local to the operation
- Unhonored identity preferences
  - Complicated by branches and loops
  - A single definition may reach multiple uses; a single use may be reached by multiple definitions
- Using canonical allocation?
  - Easy to implement
  - Compiler has the freedom to choose the location of single allocation
  - But may incur a lot of communication
- A better approach: partition program into "regions"
  - Combines single allocation and allocation by usage
  - Can have different allocations for an array in different regions

High-level algorithm

- Lifetime analysis of arrays
- Construction of preference graph
  - Nodes: array occurrences
  - Edges: preferences labeled with costs
  - Costs: Cost of motion resulting from not honoring the preference
- Processing of the preference graph
  - Greedy, in non-increasing cost order
  - If edges have the same cost, identity edges are processed first
- Computed alignments lead to division of regions

Transformation frameworks

- Unified transformation
  - Unimodular matrix theory: applicable to loop interchange, reversal, and skew
  - Template-based: applicable to unimodular, tiling, coalescing, and parallel loop execution of perfect loop nests
  - More ambitious (and expensive) techniques available
- Searching the transformation space
  - Target machine represented as a set of features
  - Search based on hierarchical heuristics

Compiler evaluation

- How can we compare one compiler to another?
- Remains an unsolved problem
- No universally agreed upon metrics
- Results are architecture specific
- Results are application specific

Benchmarks

- Benchmarks have received much attention

Benchmarks

- Benchmarks have received much attention
- Popular benchmarks:
  - SPEC
  - SPLASH
  - NAS
  - The Perfect Club

Code characteristics

- A number of studies focus on the applications
- Early studies by Knuth in 1971

Code characteristics

- A number of studies focus on the applications
- Early studies by Knuth in 1971
- Classical results:
  - Most of the time is spent in a small fraction of the code
  - 95% of all do loops incremented their index by 1
  - 40% of all do loops contained only one statement

Compiler effectiveness

- How can we evaluate the effectiveness of a compiler?

Compiler effectiveness

- How can we evaluate the effectiveness of a compiler?
  - Examine compiler's output by hand
  - Compare its performance to other compilers
  - Compare full- with no- optimization
  - Compare parallel and uniprocessor versions of an application

Instruction-level parallelism

- What is the potential gain of instruction-level parallelism?
- A number of studies have tried to find an upper bound.

Instruction-level parallelism

- What is the potential gain of instruction-level parallelism?
- A number of studies have tried to find an upper bound.
- Some where very pessimistic (Flynn bottleneck)
  - Looked only within the scope of a basic block

Instruction-level parallelism

- What is the potential gain of instruction-level parallelism?
- A number of studies have tried to find an upper bound.
- Some where very pessimistic (Flynn bottleneck)
  - Looked only within the scope of a basic block
- Parallelism can be exploited across block boundaries
  - However some approaches required huge amounts of hardware
  - Many suggested that general applications are much harder to parallelize than scientific applications
  - VLIW architectures showed promise

Conclusion

- Where are the optimizations with the biggest impact?

Conclusion

- Where are the optimizations with the biggest impact?
- Loops:
  - Expose parallelism over a loop
  - Reduce the number of instructions in a loop body
  - Improve memory locality over the loop
  - Reduce loop overhead

Summing it all up

- We have described a large number of transformations
- Many of them promise large performance gains

Summing it all up

- We have described a large number of transformations
- Many of them promise large performance gains
- However:
  - Current optimizing compilers lack an organizing principle
  - Their tuning is more of a black-art than a science
  - Often a transformation is performed only to be undone by a subsequent one
  - Object oriented paradigms present significant challenges for optimization

Current issues

- Scalability
- Automatic parallelization has yet to yield any concrete results

Current issues

- Scalability
- Automatic parallelization has yet to yield any concrete results
- How about learning which optimizations to perform?
  - There seems to be no work on this approach
  - It would be very complicated
  - One of its prerequisites would be the ability to measure a solution's success
  - However that is an unsolved problem as well

Discussion

- What do you think of compiler technology?
- Do they have a really hard problem?
- Are they approaching it correctly?

Discussion

- What do you think of compiler technology?
- Do they have a really hard problem?
- Are they approaching it correctly?
- Should languages become more constrained to aid optimization?

Discussion

- What do you think of compiler technology?
- Do they have a really hard problem?
- Are they approaching it correctly?
- Should languages become more constrained to aid optimization?
- What do the failures of parallel compilers mean for frameworks such as map-reduce?
- How does the content of this paper translate to your research?

Thank you for your attention

- Thank you for your attention
- Questions?