Overpartitioning with the Rice dHPF Compiler

Strategies for Achieving High Performance in High Performance Fortran

Ken Kennedy
Center for High Performance Software
Rice University

http://www.cs.rice.edu/~ken/Presentations/HUG00Overpartitioning.pdf
Collaborators

Bradley Broom
Daniel Chavarria-Miranda
Guohua Jin
Rob Fowler
Ken Kennedy
John Mellor-Crummey
Qing Yi
Outline

• Overview of dHPF Framework
  — Introduction to overpartitioning
  — Integer set framework
  — Virtual processors

• Multipartitioning
  — Strategy for load balance in series of dimensional sweeps

• Out-of-Core Distributions
  — Use of overpartitioning to generalize

• Automatic Conversion to Recursive Blocking
  — Use of iteration-space slicing and transitive dependence
Motivation and Goals

• **Goal:** Compile-time techniques that provide highest performance,
  — Competitive with hand-coded parallel programs
  — With little or no recoding to suit compiler

• To achieve this goal, a data-parallel compiler must:
  — Support complex choices for partitioning computation
  — Facilitate analysis and code generation for sophisticated optimizations

An abstract integer-set approach in dHPF makes sophisticated analysis and code generation practical
Overpartitioning Framework in dHPF

Generalization of BLOCK partitionings

• Partition computation into tiles
  — each processor will compute one or more tiles
  — symbolic variables specify tile extents

• Treat each tile as a virtual processor when analyzing data movement (i.e. communication, I/O)

• Generate code to orchestrate tiled computations
  — iterate over a processor’s multiple tiles in the right order
    - must preserve a computation’s data dependences
  — aggregate data movement for a processor’s tiles when appropriate
    - yes: shift communication for all tiles
    - no: data movement for an out-of-core tile
### Analysis Support: Integer Set Framework

<table>
<thead>
<tr>
<th>3 types of Sets</th>
<th>3 types of Mappings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>Layout: data processors</td>
</tr>
<tr>
<td>Iterations</td>
<td>Reference: iterations data</td>
</tr>
<tr>
<td>Processors</td>
<td>Comp Part: iterations processors</td>
</tr>
</tbody>
</table>

- **Representation:** Omega Library [Pugh et al]
  - relations: sets and mappings of integer tuples
  - relation constraints expressed using Presburger Arithmetic

- **Analysis using set equations to compute**
  - iterations allocated to a tile
  - non-local data needed by a tile
  - tiles that contain non-local data of interest, ...

- **Code generation from sets**
  - use iteration sets to generate SPMD loops and statement guards
  - use data movement sets to generate messages or I/O
Innovations Using the Set Framework

• Generalized computation partitionings
  – non-owning processors can compute values for data elements
    - select partitionings to reduce data movement frequency
    - partially replicate computation to reduce data movement
  – data movement analysis support for non-local writes

• Data movement optimization
  – e.g. aggregation of communication for multiple tiles

• Sophisticated computation restructuring
  – e.g. local/non-local index set splitting for loops

• Scalar code optimization
  – logical constraint propagation to reduce guards
Symbolic Product Constraints in HPF

- **HPF BLOCK** distribution on unknown $P$; block size $B = \lceil N/P \rceil$:

  \[
  \begin{array}{cccc}
  0 & \ldots & p & P-1 \\
  1 & p*B & p*B+B & N \\
  \end{array}
  \]

- **HPF CYCLIC** distribution on unknown $P$:

  \[
  p \ p+P \ \ldots \ p+k*P \ \ldots
  \]

- **Challenge**: no symbolic products in Presburger Arithmetic

- **Approach**:
  - analysis: use virtual processors instead of physical
  - code generation: augment generated code with extra loops to perform virtual-to-physical mapping
For BLOCK distributed dimension

- Any block is a VP
  - \(v\) owns \([v, v+B-1]\), \(1 \leq v \leq N\) (yields fictitious VPs)
  - \(p \rightarrow v : v = p \times B + 1\)

- **Computation:** No extra code needed
- **Communication:** Avoid messages to fictitious processors
Speedups for BT on IBM SP-2 *

Class A speedups relative to 4 processor MPI
Class B speedups relative to 16 processor MPI

*Coarse grain pipelining does pretty well for BT.
But ...
Multipartitioning

• **Motivation**
  
  — Data and Computation partitioning strategies
    - critical for parallel performance
  
  — Standard HPF and OpenMP partitionings
    - good performance for loosely synchronous computations
  
  — Algorithms such as ADI are difficult to parallelize
    - They require tighter synchronization
    - A dimensional sweep along a partitioned dimension will induce serialization
Parallelizing Line Sweeps via Block Partitionings

- Local Sweeps along $x$ and $z$
- Local Sweep along $y$

- Fully parallel computation
- High communication volume: transpose ALL data
- Performance lags behind coarse-grained pipelining [SC98]
Coarse-Grain Pipelining

- Partial wavefront-type parallelism
- Processors are idle part of the time
Multipartitioning

- Type of skewed-cyclic distribution
- Each processor owns a section in each of the distributed dimensions
Multipartitioning in dHPF

- Extend HPF language -

  ➔ New keyword, MULTI, for specifying multipartitioned data distributions

  ➔ The handling of multiple dimensions in multipartitioning, requires a 1D processor array

  ➔ !HPF$ PROCESSORS p(number_of_processors())

  ➔ !HPF$ TEMPLATE gridx(SX,SY,SZ)

  ➔ !HPF$ DISTRIBUTE gridx(MULTI,*,MULTI) ONTO p
Evaluating Parallelizations

Compiler-generated coarse-grain pipelining

Hand-coded multipartitioning
Experimental Results

Execution Trace for NAS SP Class ’A’ - 16 processors (September 2000)

- dHPF-generated Multipartitioning
- Hand-coded Multipartitioning
Speedups

Measurements taken in September 2000

Speedups

Hand MPI

dHPF MPI

Number of Processors

Speedup

0
2
4
6
8
10
12
14
16
18
20
1
4
9
16
Compilation for Out-of-Core Execution

• dHPF compiler recognizes new HPF-like directives for distributing data over virtual out-of-core (OOC) processors:
  
  CSIO$  processors pio(100)
  CSIO$  template tio(10000,10000)
  CSIO$  align b(i,j) with tio(i,j)
  CSIO$  distribute tio(*,block) onto pio

• dHPF compiler generates code to

  —iterate over virtual processors,
  —copy tiles to and from disk,
  —communicate between tiles.
Divide and Conquer by Compiler

• Recursive algorithms are valuable
  — Locality in multi-level memory hierarchy
    - Comparable or better performance than blocking
    - Less tuning for cache configurations
  — Parallelism in multi-processor system
    - Dynamic load balancing

• Automatic recursion by compiler
  — Transforming loops into recursion automatically
  — Should working across different loop nests
  — Should be practical for real-world applications

• Techniques
  — Iteration space slicing
  — Transitive dependence analysis
Recursive Partition of Computation

\[
\begin{align*}
\text{K:} & \quad \text{do } k = 1, N-1 \\
& \quad \text{do } i = k+1, N \\
& \text{\quad s1: } \quad a(i,k) = a(i,k) / a(k,k) \\
& \quad \text{\quad enddo} \\
\text{J:} & \quad \text{do } j = k + 1, N \\
\text{I:} & \quad \text{do } i = k+1, N \\
& \text{\quad s2: } \quad a(i,j) = a(i,j) - a(i,k) \times a(k,j) \\
& \quad \text{\quad enddo} \\
& \quad \text{enddo} \\
& \text{enddo} \\
\text{(LU with no pivoting)}
\end{align*}
\]
Recursive Partition of Computation

K: do k = 1, N-1
    do i = k+1, N
    s1: a(i,k) = a(i,k) / a(k,k)
        enddo
J: do j = k + 1, N
I: do i = k+1, N
    s2: a(i,j) = a(i,j) - a(i,k) * a(k,j)
        enddo
    enddo
enddo

(LU with no pivoting)

K: 1, min(Ubi, Ubj)-1
J: max(k+1, Lbj), Ubj
I: max(k+1, Lbi), Ubi
Recursive Partition of Computation

\[ K: \text{do } k = 1, N-1 \]
\[ \text{do } i = k+1, N \]
\[ s1: \quad a(i,k) = a(i,k) / a(k,k) \]
\[ \text{enddo} \]
\[ J: \text{do } j = k + 1, N \]
\[ I: \text{do } i = k+1, N \]
\[ s2: \quad a(i,j) = a(i,j) - a(i,k) \ast a(k,j) \]
\[ \text{enddo} \]
\[ \text{enddo} \]
\[ \text{enddo} \]

(LU with no pivoting)

Key statement

- \( K: 1, \min(Ubi,Ubj)-1, \min((Ubj,Ubi)-1 \)
- \( I: \max(k+1,Lbi), Ubi \)
- \( J: \max(k+1,Lbj), Ubj \)
- \( K: \max(Lbi,Lbj)-1, \min((Ubj,Ubi)-1 \)
Recursive Partition of Computation

K: \( \text{do } k = 1, N-1 \)
   \( \text{do } i = k+1, N \)
   s1: \( a(i,k) = a(i,k) / a(k,k) \)
   enddo
J: \( \text{do } j = k + 1, N \)
I: \( \text{do } i = k+1, N \)
   s2: \( a(i,j) = a(i,j) - a(i,k) * a(k,j) \)
   enddo
   enddo
   enddo

(\( \text{LU with no pivoting} \))
Iteration Space Slicing for Recursion

1

Previous(s2)

Before(s1,s2) Previous(s1) Current(s1)

Current(s2)

N

N

Center for High Performance Software
Transformed non-pivoting LU

LU_recur (Lbj, Ubj, Lbi, Ubi)

if data fit in primary cache

\[
\begin{align*}
d & \quad do \quad k = 1, \min(N-1, Ubi-1, Ubj-1) \\
& \quad \quad if \quad (k >= \max(Lbj-1, Lbi-1)) \quad then \\
& \quad \quad \quad do \quad i = \max(k+1, Lbi), \min(Ubi, N) \\
& s1: \quad a(i,k) = a(i,k) / a(k,k) \\
& J: \quad do \quad j = \max(k + 1, Lbj), \min(Ubj, N) \\
& I: \quad do \quad i = \max(k+1, Lbi), \min(Ubi, N) \\
& s2: \quad a(i,j) = a(i,j) - a(i,k) * a(k,j)
\end{align*}
\]

else

\[
\begin{align*}
call \quad & LU-recur(Lbj, (Lbj+Ubj)/2, \quad Lbi, (Lbi+Ubi)/2) \quad (left-top) \\
call \quad & LU-recur(Lbj, (Lbj+Ubj)/2, (Lbi+Ubi)/2+1, Ubi) \quad (right-top) \\
call \quad & LU-recur((Lbj+Ubj)/2 +1, Ubj, Lbi, (Lbi+Ubi)/2) \quad (left-bottom) \\
call \quad & LU-recur((Lbj+Ubj)/2 +1, Ubj, (Lbi+Ubi)/2 +1, Ubi) \quad (right-bottom)
\end{align*}
\]
Experimental Evaluation

• Benchmarks
  - MM, LU (non-pivoting), Cholesky, Erlebacher

• Experimental environment
  - 195MHz SGI R10000 workstation
  - L1 cache: 32KB 2-way associative
  - L2 cache: unified 1MB 2-way associative

• Benchmark performance
  - Significant improvement over unblocked original code
  - Comparable or better performance than blocking
Performance of Matrix Multiply

Cycles in billions

<table>
<thead>
<tr>
<th></th>
<th>513*513</th>
<th>1025*1025</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.9</td>
<td></td>
<td>22.8</td>
</tr>
</tbody>
</table>

L1 misses in millions

<table>
<thead>
<tr>
<th></th>
<th>513*513</th>
<th>1025*1025</th>
</tr>
</thead>
<tbody>
<tr>
<td>36.0</td>
<td></td>
<td>290.1</td>
</tr>
</tbody>
</table>

L2 misses in millions

<table>
<thead>
<tr>
<th></th>
<th>513*513</th>
<th>1025*1025</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;3.6</td>
<td></td>
<td>&gt;36.7</td>
</tr>
</tbody>
</table>
Performance of non-pivoting LU

Cycles in billions

Size: 1024*1024

L1 misses in millions

L2 misses in millions
Summary

• dHPF Compiler Designed to Achieve Highest Possible Performance
  — Integer set framework
  — Overpartitioning
  — Virtual processors

• Multipartitioning
  — Strategy for load balance in series of dimensional sweeps

• Out-of-Core Distributions
  — Can be generalized through use of overpartitioning

• Automatic Conversion to Recursive Blocking
  — Powerful framework for transitive dependence and iteration space slicing
  — Direct conversion to recursive blocking for loop nests