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

#### 3 types of Sets
- Data
- Iterations
- Processors

#### 3 types of Mappings
- Layout: Data ⇔ Processors
- Reference: Iterations ⇔ Data
- Comp Part: Iterations ⇔ Processors

- **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

---

*Center for High Performance Software*
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 $B = \lceil N/P \rceil$:

$$
\begin{array}{cccc}
0 & \cdots & p & P-1 \\
1 & p*B & p*B+B & N
\end{array}
$$

• HPF CYCLIC distribution on unknown $P$:

$$
p \quad p+P \quad \cdots \quad p+k*P \quad \cdots
$$

• 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
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 $y$

Transpose

Transpose back

- 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
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
Measurements taken in September 2000

**Speedups**

- **Hand MPI**
- **dHPF MPI**

![Graph showing speedups](image-url)
Multipartitioning Summary

- dHPF: First compiler support for multipartitioned data distributions
  - Automatic code generation for complex multipartitioned distributions
  - Scalar performance is within 13% of sequential version
  - Competitive parallel performance
    - Message coalescing across multiple arrays will improve it
  - Scalable computation partitioning: balanced parallelism
  - Coexists with advanced optimizations in dHPF
    - non-owner computes computation partitionings,
    - partial computation replication, ...
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.
Out-of-Core a special case of Overpartitioning

- Out-of-core partitioning similar to multipartitioning.
  - Both must generate code to iterate over virtual processors.
  - Both must manage communication between tiles.

- OOC implemented in dHPF using ad hoc code before general overpartitioning framework available.
  - OOC and in-core distributions restricted to disjoint dimensions.

- OOC partitioning could be better implemented as a client of the general overpartitioning framework
  - simpler,
  - allow more flexible combinations of OOC and in-core distributions.
Specific Requirements of OOC Partitioning

• Traversal order of OOC virtual processors significantly affects performance.

• Communication must be kept within the loop iterating over virtual processors.

• Communication involving a tile can occur only when the tile is in memory.

• Communication between OOC tiles allocated to the same processor is very expensive.
  — Requires extra I/O, retaining data in memory, or revisiting a tile.
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

K: \hspace{1cm} \text{do } k = 1, N-1
    \hspace{1cm} \text{do } i = k+1, N
        a(i,k) = a(i,k) / a(k,k)
    \hspace{1cm} \text{enddo}

s1: \hspace{1cm} \text{enddo}

J: \hspace{1cm} \text{do } j = k + 1, N

I: \hspace{1cm} \text{do } i = k+1, N
    a(i,j) = a(i,j) - a(i,k) * a(k,j)
    \hspace{1cm} \text{enddo}

s2: \hspace{1cm} \text{enddo}
    \hspace{1cm} \text{enddo}
    \hspace{1cm} \text{enddo}

(LU with no pivoting)

Recursive Loops
(Lbj : Ubj, Lbi : Ubi)

Key statement
K: 1, \text{min(Ubi, Ubj)}-1
J: max(k+1, Lbj), Ubj
I: max(k+1, Lbi), Ubi
Iteration Space Slicing for Recursion

- $\text{Previous}(s1) = \text{Before}(s1,s2)[\text{Previous}(s2)]$
- $\text{Current}(s1) = \text{Before}(s1,s2)[\text{Current}(s2)] - \text{Previous}(s1)$
Transformed non-pivoting LU

\textbf{LU\_recur (Lbj, Ubj, Lbi, Ubi)}

if data fit in primary cache

\begin{verbatim}
    do k = 1, min(N-1, Ubi-1, Ubj-1)
        if (k >= max(Lbj-1, Lbi-1) ) then
            do i = max(k+1, Lbi), min(Ubi, N)
                a(i,k) = a(i,k) / a(k,k)
        \end{verbatim}

\begin{verbatim}
    J:     do j = max (k + 1, Lbj), min(Ubj, N)
    I:     do i = max (k+1, Lbi), min(Ubi, N)
    s2:     a(i,j) = a(i,j) - a(i,k) * a(k,j)
\end{verbatim}

else

\begin{verbatim}
    call LU-recur(Lbj, (Lbj+Ubj)/2, Lbi, (Lbi+Ubi)/2)  \hspace{1cm} (left-top)
    call LU-recur(Lbj, (Lbj+Ubj)/2, (Lbi+Ubi)/2+1, Ubi) \hspace{1cm} (right-top)
    call LU-recur((Lbj+Ubj)/2 +1, Ubj, Lbi, (Lbi+Ubi)/2) \hspace{1cm} (left-bottom)
    call LU-recur((Lbj+Ubj)/2 +1, Ubj, (Lbi+Ubi)/2 +1, Ubi) \hspace{1cm} (right-bottom)
\end{verbatim}
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>

Center for High Performance Software
Performance of non-pivoting LU

Cycles in billions

<table>
<thead>
<tr>
<th></th>
<th>original</th>
<th>block</th>
<th>recursion</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>11.8</td>
<td>5.2</td>
<td>4.2</td>
</tr>
</tbody>
</table>

Size: 1024*1024

L1 misses in millions

<table>
<thead>
<tr>
<th></th>
<th>original</th>
<th>block</th>
<th>recursion</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>93.1</td>
<td>359.4</td>
<td>103.4</td>
</tr>
</tbody>
</table>

L2 misses in millions

<table>
<thead>
<tr>
<th></th>
<th>original</th>
<th>block</th>
<th>recursion</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>45.3</td>
<td>1.7</td>
<td>1.3</td>
</tr>
</tbody>
</table>
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