Parallel programming with hierarchically tiled arrays

David Padua

University of Illinois at Urbana-Champaign
Objectives

• To develop, implement and evaluate a new parallel programming paradigm that is a generalization of the SIMD programming paradigm.
  – Main features of the paradigm:
    • Single thread of control: simplifies understanding and analysis of the code and the transformation into parallel form of sequential codes
    • Use of aggregates to represent parallelism: using operations on aggregates tends to produce shorter and more readable codes than explicit MPI programs
  – Therefore programs in the proposed paradigm are easy to develop and maintain

• To develop compiler techniques for the proposed paradigm.
  – Compiler techniques needed by the proposed paradigm are significantly simpler than those needed to compile High-Performance Fortran and similar languages.
Characteristics of the proposed programming paradigm

• Programs written in the proposed paradigm can be conceived as executed by a workstation attached to a multiprocessor.

• The workstation executes all operation of the program except operations on *hierarchically tiled arrays (HTA)* whose top-level tiles are distributed across the multiprocessor. Operation on these HTAs are carried out by the multiprocessor.

• However, execution is SPMD.
Hierarchically tiled arrays

- Hierarchically tiled arrays are arrays partitioned into tiles. The tiles could be conventional arrays or could recursively be tiled.
- Tiles are first class objects. They can be accessed explicitly and operations are defined on tiles.
- The tiles in a hierarchically tiled array can be ignored and the scalar elements of the HTAs could be accessed directly.
Accessing the elements of an HTA

• Above we depict a three-level HTA. The top level is a vector with two tiles. Each one of these tiles is a 4 by 3 array of tiles. The second-level tiles are 3-element vectors.
• The red element can be referenced as \( A\{2\}\{3,1\}(2) \) or as \( A(3,11) \). In the first case the HTA is accessed hierarchically and in the second case as a “flat” two-dimensional array.
Two Ways of Referencing the Elements of an 8 x 8 Array.

<table>
<thead>
<tr>
<th>C{1,1}(1,1) C(1,1)</th>
<th>C{1,1}(1,2) C(1,2)</th>
<th>C{1,1}(1,3) C(1,3)</th>
<th>C{1,1}(1,4) C(1,4)</th>
<th>C{1,2}(1,1) C(1,5)</th>
<th>C{1,2}(1,2) C(1,6)</th>
<th>C{1,2}(1,3) C(1,7)</th>
<th>C{1,2}(1,4) C(1,8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C{1,1}(2,1) C(2,1)</td>
<td>C{1,1}(2,2) C(2,2)</td>
<td>C{1,1}(2,3) C(2,3)</td>
<td>C{1,1}(2,4) C(2,4)</td>
<td>C{1,2}(2,1) C(2,5)</td>
<td>C{1,2}(2,2) C(2,6)</td>
<td>C{1,2}(2,3) C(2,7)</td>
<td>C{1,2}(2,4) C(2,8)</td>
</tr>
<tr>
<td>C{1,1}(3,1) C(3,1)</td>
<td>C{1,1}(3,2) C(3,2)</td>
<td>C{1,1}(3,3) C(3,3)</td>
<td>C{1,1}(3,4) C(3,4)</td>
<td>C{1,2}(3,1) C(3,5)</td>
<td>C{1,2}(3,2) C(3,6)</td>
<td>C{1,2}(3,3) C(3,7)</td>
<td>C{1,2}(3,4) C(3,8)</td>
</tr>
<tr>
<td>C{1,1}(4,1) C(4,1)</td>
<td>C{1,1}(4,2) C(4,2)</td>
<td>C{1,1}(4,3) C(4,3)</td>
<td>C{1,1}(4,4) C(4,4)</td>
<td>C{1,2}(4,1) C(4,5)</td>
<td>C{1,2}(4,2) C(4,6)</td>
<td>C{1,2}(4,3) C(4,7)</td>
<td>C{1,2}(4,4) C(4,8)</td>
</tr>
<tr>
<td>C{2,1}(1,1) C(5,1)</td>
<td>C{2,1}(1,2) C(5,2)</td>
<td>C{2,1}(1,3) C(5,3)</td>
<td>C{2,1}(1,4) C(5,4)</td>
<td>C{2,2}(1,1) C(5,5)</td>
<td>C{2,2}(1,2) C(5,6)</td>
<td>C{2,2}(1,3) C(5,7)</td>
<td>C{2,2}(1,4) C(5,8)</td>
</tr>
<tr>
<td>C{2,1}(2,1) C(6,1)</td>
<td>C{2,1}(2,2) C(6,2)</td>
<td>C{2,1}(2,3) C(6,3)</td>
<td>C{2,1}(2,4) C(6,4)</td>
<td>C{2,2}(2,1) C(6,5)</td>
<td>C{2,2}(2,2) C(6,6)</td>
<td>C{2,2}(2,3) C(6,7)</td>
<td>C{2,2}(2,4) C(6,8)</td>
</tr>
<tr>
<td>C{2,1}(3,1) C(7,1)</td>
<td>C{2,1}(3,2) C(7,2)</td>
<td>C{2,1}(3,3) C(7,3)</td>
<td>C{2,1}(3,4) C(7,4)</td>
<td>C{2,2}(3,1) C(7,5)</td>
<td>C{2,2}(3,2) C(7,6)</td>
<td>C{2,2}(3,3) C(7,7)</td>
<td>C{2,2}(3,4) C(7,8)</td>
</tr>
<tr>
<td>C{2,1}(4,1) C(8,1)</td>
<td>C{2,1}(4,2) C(8,2)</td>
<td>C{2,1}(4,3) C(8,3)</td>
<td>C{2,1}(4,4) C(8,4)</td>
<td>C{2,2}(4,1) C(8,5)</td>
<td>C{2,2}(4,2) C(8,6)</td>
<td>C{2,2}(4,3) C(8,7)</td>
<td>C{2,2}(4,4) C(8,8)</td>
</tr>
</tbody>
</table>
Uses of HTAs

• The topmost levels of an HTA can be distributed across the multiprocessor. This enables the distribution of data and the explicit representation of communication and parallel computation.

• Lower level tiles can be used to conveniently represent algorithms with a high degree of locality. This is of great importance for machines with a deep memory hierarchy.

• The “flattened” representation enables the gradual transformation of sequential programs to parallel form. The part of the sequential program that has not been parallelized will reference the array in its original (untiled) form. This referencing will be meaningful because of flattening.
HTA Operations:
F90 Conformability

\[
\begin{align*}
\text{size}(\text{rhs}) &= 1 \\
\dim(\text{lhs}) &= \dim(\text{rhs}) \\
\text{and} \\
\text{size}(\text{lhs}) &= \text{size}(\text{rhs})
\end{align*}
\]
All conformable HTAs can be operated using the primitive operations (add, subtract, etc)
HTA Assignment

\[ h(:, :) = t(:, :) \]

\[ h(1,:) = t(2,:) \]

\[ h(1,:)(2,:) = t(2,:)(1,:); \]

implicit communication
Data parallel functions in F90

\[
\begin{array}{c}
\text{sin} \\
\begin{array}{cccc}
\cdot & \cdot & \cdot & \\
\cdot & \cdot & \cdot & \\
\cdot & \cdot & \cdot & \\
\end{array}
\end{array}
\rightarrow
\begin{array}{c}
\text{sin(\(\cdot\)) sin(\(\cdot\)) sin(\(\cdot\))} \\
\text{sin(\(\cdot\)) sin(\(\cdot\)) sin(\(\cdot\))} \\
\text{sin(\(\cdot\)) sin(\(\cdot\)) sin(\(\cdot\))}
\end{array}
\]
Map

\[ r = \text{map} (@\sin, h) \]
Map

\[ r = \text{map}(\text{@sin}, h) \]
Map

\[ r = \text{map} (@\sin, h) \]
Reduce

\[ r = \text{reduce}\left(\@\text{max}, h\right) \]
Reduce

\[ r = \text{reduce} \left( @\text{max}, h \right) \]
Reduce

\( r = \text{reduce} (\@max, h) \)
Higher level operations

- repmat(h, [1, 3])
- circshift(h, [0, -1])
- transpose(h)
Matrix Multiplication

1. Tiled Matrix Multiplication in a Conventional Language

for I=1:q:n
    for J=1:q:n
        for K=1:q:n
            for i=I:I+q-1
                for j=J:J+q-1
                    for k=K:K+q-1
                        c(i,j)=c(i,j)+a(i,k)*b(k,j);
                    end
                end
            end
        end
    end
end
2. Tiled Matrix Multiplication Using HTAs

Matrix Multiplication

Here $c_{i,j}$, $a_{i,k}$, $b_{k,j}$, and $T$ represent submatrices.

The * operator represents matrix multiplication in MATLAB.

```matlab
for i=1:m
    for j=1:m
        T=0;
        for k=1:m
            T=T+a{i,k}*b{k,j};
        end
        c{i,j}=T;
    end
end
```
Blocked Recursive Matrix Multiplication

Blocked-Recursive implementation

\[ A\{i, k\}, B\{k, j\} \text{ and } C\{i, j\} \text{ are sub-matrices of } A, B \text{ and } C. \]

```plaintext
function c = matmul (A, B, C)
    if (level(A) == 0)
        C = matmul_leaf (A, B, C)
    else
        for i = 1:size(A, 1)
            for k = 1:size(A, 2)
                for j = 1:size(B, 2)
                    C\{i, j\} = matmul(A\{i, k\}, B\{k, j\}, C\{i,j\});
                end
            end
        end
    end
end
```
Matrix Multiplication

matmul \( (A, B, C) \)

matmul \( (A_{i,k}, B_{k,j}, C_{i,j}) \)

matmul \( (A A_{i,k}, B B_{k,j}, C C_{i,j}) \)

matmul_leaf \( (A A A_{i,k}, B B B_{k,j}, C C C_{i,j}) \)
Matrix Multiplication

```
for i = 1:size(A, 1)
    for k = 1:size(A, 2)
        for j = 1:size(B, 2)
            C(i,j) = C(i,j) + A(i, k) * B(k, j);
        end
    end
end
```

matmul \((A, B, C)\)

matmul \((A[i, k], B[k, j], C[i, j])\)

matmul \((AA[i, k], BB[k, j], CC[i, j])\)

matmul_leaf \((AAA[i, k], BBB[k, j], CCC[i, j])\)

```
for i = 1:size(A, 1)
    for k = 1:size(A, 2)
        for j = 1:size(B, 2)
            C(i,j) = C(i,j) + A(i, k) * B(k, j);
        end
    end
end
```
Parallel operations and communication with HTAs

• *Array operations* on HTAs can represent communication or computation.
  – Assignment statements where all HTA indices are identical are computations executed in the home of each of the HTA elements involved.
  – Assignment statements where this is not the case represent communication operations.
Advantages

• Aggregate operations on HTAs representing communication are implemented using a communication library (MPI or native library). This model imposes structure on the use of the communication library routines in the same way that looping constructs impose structure on branch instructions.

• Tiles represent algorithms with a high degrees of locality naturally. Explicit access to tiles make the algorithms easy to read and understand.
Using HTAs to Represent Data Distribution and Parallelism

Cannon’s Algorithm
Implementation of Cannon’s algorithm

c{1:n,1:n}(1:p,1:p) = 0

!Communication
!Zero is broadcast to all
!processors

do i=2,n
    a{1:n,:} = cshift(a(i:n,:),dim=2,shift=1);
    b{:,i:n} = cshift(b{:,i:n},dim=1,shift=1)
end do

do k=1,n
    c{::} = c{::} + a{::}*b{::};
    a{::} = cshift(a{::},dim=2);
    b{::} = cshift(b{::},dim=1);
end do
The SUMMA Algorithm

Use now the outer-product method (n²-parallelism)
Interchanging the loop headers of the loop in Example 1 produce:

\[
\begin{align*}
\text{for } k &= 1:n \\
& \text{for } i = 1:n \\
& \quad \text{for } j = 1:n \\
& \quad \quad C_{i,j} = C_{i,j} + A_{i,k} B_{k,j} \\
& \quad \text{end} \\
& \text{end} \\
& \text{end}
\end{align*}
\]

To obtain n² parallelism, the inner two loops should take the form of a block operations:

\[
\begin{align*}
\text{for } k &= 1:n \\
& \quad C_{:,} = C_{:,} + A_{:,k} \otimes B_{k,:} \\
& \text{end}
\end{align*}
\]

Where the operator \( \otimes \) represents the outer product operations
The SUMMA Algorithm

Switch Orientation -- By using a column of A and a row of B broadcast to all, compute the "next" terms of the dot product.
The SUMMA Algorithm

c{1:n,1:n} = zeros(p,p); % communication
for i=1:n
  t1(:,:,)=spread(a(:,i),dim=2,ncopies=N); % communication
  t2(:,:,)=spread(b(i,:),dim=1,ncopies=N); % communication
  c(:,:,)=c(:,:,)+t1(:,:,)*t2(:,:,); % computation
end
while dif > epsilon
    v{2:,:}(0,:) = v{:n-1,:}(p,:);
    v{:n-1,:} (p+1,:) = v{2,:} (1,:);
    v{:,:} (:,0) = v{:,:} (:,p);
    v{:,:} (:,p+1) = v{2,:} (:,1);

    u{:,:}(1:p,1:p) = a * (v{:,:} (1:p,0:p-1) + v{:,:} (0:p-1,1:p)+
                               v{:,:} (1:p,2:p+1) + v{:,:} (2:p+1,1:p)) ; %computation

    dif=max(max( abs ( v - u ) ));
    v = u ;
end
Sparse Matrix Vector Product

\[ A \mathbf{b} \]

(Distributed) (Replicated)

P1

P2

P3

P4
Sparse Matrix Vector Product
HTA Implementation

c = hta(a, {dist, [1]}, [4 1]);

v = hta(4,1,[4 1]);

v{:} = b;

r = t(:);
Implementations

• MATLAB Extension with HTA
  – Global view & single threaded
  – Front-end is pure MATLAB
  – MPI calls using MEX interface
  – MATLAB library routines at the leaf level

• X10 Extension
  – Emulation only (no experimental results)

• C++ extension with HTA (Under-progress)
  – Communication using MPI (& UPC)
Evaluation

• Implemented the full set of NAS benchmarks (except SP)
  – Pure MATLAB & MATLAB + HTA
  – Performance metrics: running time & relative speedup on 1 - 128 processors
  – Productivity metric: source lines of code
  – Compared with hand-optimized F77+MPI version
  – Tiles only used for parallelism
3D Stencil convolution:
primitive HTA operations and assignments to implement communication

restrict

interpolate
\[ r_{1,:}(2,:) = r_{2,:}(1,:) \]

communication

\[ r_{:, :}(2:n-1, 2:n-1) = 0.5 \times u_{:, :}(2:n-1, 2:n-1) + 0.25 \times (u_{:, :}(1:n-2, 2:n-1) + u_{:, :}(3:n, 2:n-1) + u_{:, :}(2:n-1, 1:n-2) + u_{:, :}(2:n-1, 3:n)) \]

computation

\( n = 3 \)
Sparse matrix-vector multiplication ($A^\ast p$)
2D decomposition of $A$
replication and 2D decomposition of $p$
sum reduction across columns
\[ A = \text{hta}(M \times, \{\text{partition}_A\}, [M \ N]); \] 
\[ p = \text{repmat}(\text{hta}(\text{vector}', \{\text{partition}_p\}, [1 \ N]), [M,1]); \]
\[ A = \text{hta}(MX, \{\text{partition}_A\}, [M N]); \]
\[ p = \text{repmat}(\text{hta}(\text{vector'}, \{\text{partition}_p\} [1 N]), [M,1]); \]
\[ p = \text{map}(\text{@transpose}, p) \]
\[
A = \text{hta}(M X, \{\text{partition}_A\}, [M N]);
\]
\[
p = \text{repmat}( \text{hta}( \text{vector}' , \{\text{partition}_p\} [1 N] ), [M, 1]);
\]
\[
p = \text{map} (@\text{transpose}, p)
\]
\[
R = \text{reduce} ( '\text{sum}' , A * p, 2, \text{true});
\]
**FT**

*3D Fourier transform*

Corner turn using dpermute
Fourier transform applied to each tile

\[
\begin{align*}
  h &= \text{ft}(h, 1); \\
  h &= \text{dpermute}(h, [2 1]); \\
  h &= \text{ft}(h, 1);
\end{align*}
\]

A 2D illustration
Bucket sort of integers
Bucket exchange using assignments

1. Distribute keys
2. Local bucket sort
3. Aggregate
4. Bucket exchange
5. Local sort
**Bucket sort of integers**

Bucket exchange assignments

\[ h \text{ and } r \text{ top level distributed} \]

\[
\begin{align*}
8 & 2 & 1 & 4 & 7 & 9 & 11 & 16 & 18 & 3 & 6 & 10 & 12 & 13 & 15 & 5 & 14 & 17 \\
1 & 2 & 4 & 8 & 7 & 9 & 3 & 6 & 11 & 10 & 16 & 18 & 5 & 12 & 13 & 15 & 14 & 17 \\
\end{align*}
\]

for \( i = 1:n \)
for \( j = 1:n \)
\[ r\{i\}\{j\} = h\{j\}\{i\} \]
end
end

@sort  @sort  @sort
Experimental Results

- NAS benchmarks (MATLAB)
  - (Results in the paper)
  - CG & FT – acceptable performance and speedup
  - MG, LU, BT – poor performance, but fair speedup

C++ results yet to be obtained.
Lines of Code