Lecture 17

Data parallel programming

Motivation for data parallel programming

• Message passing is difficult: parallelism is explicit, and we need to manage many details
  – Communication
  – Synchronization
  – Data partitioning

• We have to think about new things, or look at things a bit differently
  – The distinction between remote, collective, and local data
  – Losses due to processor idleness can be dramatic

• Today we’ll look at another approach to parallel programming which eliminates some concerns, simplifying the programming process
Data parallel programming

- Recall SIMD parallelism
- Processors execute instructions in lock step under the direction of a control processor
- We may implement a software variant on MIMD architectures: data parallelism

The data parallel model

- A parallel data structure, e.g. an array, list, sequence
- Apply an operation uniformly over all processors in a single step
- Assign each array element to a virtual processor
- Implicit barrier synchronization between each step
- Program executes as if in a global space

\[
\begin{array}{ccc}
2 & 12 & 2 \\
8 & 25 & 3 \\
18 & 42 & 6
\end{array}
\]

\[
\begin{array}{ccc}
1 & 4 & 2 \\
2 & 5 & 4 \\
3 & 6 & 6
\end{array}
\]

\[
\begin{array}{ccc}
12 & 2 & 3 \\
25 & 5 & 5 \\
42 & 6 & 7
\end{array}
\]
Practical data parallel languages

- APL (1962)
- Matlab
- Fortran 90, 95
- HPF (High Performance Fortran) - 1994

The link with SIMD parallelism

- Historically, SIMD machines implemented data parallelism
  - ILIAC IV (1960’s)
  - Connection Machine (1980’s and early 1990’s)
- Intel re-introduced SIMD parallelism to support multimedia and graphics
  - SSE = Streaming SIMD extensions
  - 8 vector registers, each with 4 x 32 bit elements
  - Operations defined on vectors
  - http://x86.ddj.com/articles/sse_pt1/
How do we express parallelism?

- Operations on whole arrays
- Forall, a parallel for loop
- FORALL (triplet, triplet,...) assignment statement
  
  \[
  \text{forall } (i=0:n-1) \ x[i] = (i*2.0/n)-1.0
  \]
  
  \[
  \text{forall } (i=0:n-1, j = 0:m-1) \ X[\ i, j \ ] = 1.0/(i+j)
  \]
- The head of the loop defines an index domain
- We think of each member of the index domain as defining a virtual processor
  - In the first example, we have an index domain of 0 to n-1, with n-1 virtual processors
  - In the second example we have an index domain of n \times m processors

Other kinds of loops loops

- Indirect indexing
  \[
  \text{forall } (i = 0:n-1) \ D[\ \text{Indx}[i]] = C[i]
  \]
- Optional mask or guard
  \[
  \text{forall } (i=1:n, j:1:N, i == j) \ X[i,j] = 0 \ // \ Guard
  \]
- A parallel loop nested inside a serial loop
  \[
  \text{forall } (i=1:n, j:1:N, i == j) \ X[i,j] = 0 \ // \ Guard
  \]
  
  \[
  \text{for } k = 1:n
  \]
  
  \[
  \text{forall } (i=1:n, j=1:n) \ C(i,j) = C(i,j) + A(i,k) * B(k,j)
  \]
- An illegal forall loop
  \[
  \text{forall } (i=1:n, j=1:n, k=1:n) \ C(i,j) = C(i,j) + A(i,k)*B(k,j)
  \]
Forall loop evaluation

- Evaluate entire RHS for all index values (in any order) and assign to a temporary
- Perform all assignments (in any order) using the temporary
- No more than one value for each element on the left hand side
- The following are equivalent

\[
\text{forall } (i=0:n-1) \ x[i] = \left( \frac{i \times 2.0}{n} \right) - 1.0 \\
\text{forall } (i=0:n-1) \ tmp[i] = \left( \frac{i \times 2.0}{n} \right) - 1.0 \\
\text{forall } (i=0:n-1) \ x[i] = tmp[i]
\]

Stencil computations

- Sweep over a 1D mesh using nearest neighbor computation

\[
\text{forall } (i = 1:n) \ u\text{new}[i] = \frac{u\text{old}[i-1] + u\text{old}[i+1]}{2.0}
\]

- Sweep over a 2D mesh using nearest neighbor computation

\[
\text{forall } (i = 1:n, j=1:n) \ u\text{new}[i,j] = \frac{u\text{old}[i,j-1] + u\text{old}[i,j+1] + u\text{old}[i+1,j] + u\text{old}[i-1,j]}{4.0}
\]
A note on semantics

• For our 1D mesh sweep

\[ \text{forall } (i = 1:n) \text{ unew}[i] = (\text{uold}[i-1] + \text{uold}[i+1]) / 2.0 \]

• We may use same array on LHS and RHS

\[ \text{forall } (i = 1:n) \text{ u}[i] = (\text{u}[i-1] + \text{u}[i+1]) / 2.0 \]

• Because all assignments entail a complete evaluation of the RHS before the LHS is assigned

\[ \text{forall } (i = 1:n) \text{ tmp}[i] = (\text{u}[i-1] + \text{u}[i+1]) / 2.0 \]
\[ \text{forall } (i = 1:n) \text{ u}[i] = \text{tmp}[i] \]

Array Operations

Parallel Assignment, equivalent to a forall

```
Double A[N,N,N], Z[N,N], D[N], C[N], T[5]
A = 0  // scalar extension, all elements set
       // to 0
forall(i=0:N-1,j=0:N-1,k=0:N-1) A[i] = 0
Z = 3.7 //forall(i=0:N-1,j=0:N-1) Z[i] = 3.7
D = C  // array copy
T = [1 2 3 4 5]  // An array literal
```

Binary array operators operate pointwise on conforming arrays

• same size and shape
• The arrays could be multidimensional
Extension to array operations

- Scalars can be combined with arrays
- There are also specialized intrinsics

\[ T = \begin{bmatrix} 1 & 4 & 9 & 16 & 25 \end{bmatrix} \]
\[ U = 3 + T \quad // \quad 4 \ 7 \ 12 \ 19 \ 28 \]
\[ Z = \sqrt{T} \quad // \quad \text{Built in intrinsic extended to array} \]
\[ \quad // \quad 1 \ 2 \ 3 \ 4 \ 5 \]
\[ Y = \max(T, 10) \quad // \quad 10 \ 10 \ 10 \ 16 \ 25 \]

Array Sections

- Portion of an array defined by a triplet in each dimension
- May appear wherever an array is used

\[ A[1:5] \quad \! \text{first five elements} \]
\[ A[1:10:2] \quad \! \text{odd elements} \]
\[ A[10:2:-2] \quad \! \text{Even elements in reverse order} \]
\[ B[2:4, 2:4] \quad \! \text{3 x 3 block} \]
\[ B[, 1] \quad \! \text{first column} \]
\[ B[\cdot, \cdot] \quad \! \text{jth row} \]
Subtle semantics of `forall`

Evaluate entire RHS for all index values (in any order) and assign to a temporary array

Assign temporary to LHS of assignment

`Forall` is not like `for`

But it is like array section assignment

\[
\begin{align*}
A, B, C &= \{3 \ 2 \ 0 \ 1\} \\
\text{forall (i=1:3)} &\quad A[i] = A[i-1] \quad \downarrow \{3 \ 3 \ 2 \ 0\} \\
\text{for (i=1:3)} &\quad B[i] = B[i-1] \quad \downarrow \{3 \ 3 \ 3 \ 3\} \\
&\quad C[1:3] = C[0:2] \quad \downarrow \{3 \ 3 \ 2 \ 0\}
\end{align*}
\]

An example

- Compute the following loop using array statements

\[
\text{for i := 0 to N-1 do} \\
\quad B[i] := B[N-i-1];
\]

- If \( B = \{0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7\} \) initially
- Result is \( \{7 \ 6 \ 5 \ 4 \ 4 \ 5 \ 6 \ 7\} \)
What happens with forall?

- We try
  \[
  \text{forall } (i=0:N-1) \ B[i] = B[N-i-1]
  \]
- Which returns \(\{7 \ 6 \ 5 \ 4 \ 3 \ 2 \ 1 \ 0\}\)
  instead of \(\{7 \ 6 \ 5 \ 4 \ 4 \ 5 \ 6 \ 7\}\)

A solution

- The original loop:
  \[
  \text{for } i := 0 \text{ to } N-1 \text{ do } \\
  \quad B[i] := B[N-i-1];
  \]
- For 0 to \(N/2-1\), \(B[N-i-1]\) references initial data, but updated data for remaining values of \(i\)
Preserving semantics

- We split the loop accordingly
  
  ```
  for i := 0 to N/2-1 do
      B[i] := B[N-i-1];
  for i := N/2 to N do
      B[i] := B[N-i-1];
  ```

- Introducing the array statements
  ```
  forall (i=0:N/2-1) B[i] = B[N-i-1]
  ```

- We didn’t need to second loop

Motivating application: the N-body problem

- A classical problem
- Compute trajectories of a system of N bodies moving under mutual influence
- The bodies can be molecules, planets, stars, charged particles…
Applications

- A force law governs the way the particles interact
- No general analytic (exact) solution when $N > 2$
- Numerical simulations needed
- The value of $N$ typically ranges from thousands to millions, depending on the problem
- See animations:
  - http://atoms.org.uk/links/

Discretization

- Particles move continuously through space and time
- Thus, the force is a continuous function of position and time: $F(\mathbf{x},t)$ [$\mathbf{x}$ is a vector]
- Because we cannot solve the problem analytically we must solve it numerically
- On a computer we represent continuous values using a discrete approximation
- This implies that the motion of a particle is discrete rather than continuous
Timesteps

- Evaluate force field at discrete points in time, called timesteps $\Delta t$, $2\Delta t$, $3\Delta t$, ...
  - $\Delta t$ is called the time step (a parameter)
- “Push” the bodies according to Newton’s third law $F = ma = m \frac{du}{dt}$

Solving the N body problem

**while** (current time < end time)
  **forall** bodies $i \in 1:N$
  compute force $F_i$ induced by all bodies $j \in 1:N$
  \[ F_i = \sum_j F_{ij} \]
  update position $x_i$ by $F_i \Delta t$
  current time += $\Delta t$
**end**
Computing the force

- The running time of the computation is dominated by the force computation, so we ignore the rest
- The simplest approach is to use the direct method, with a running time of $O(N^2)$

$$\text{Force on particle } i = \sum_{j=0}^{N-1} F(x_i, x_j)$$

- $F(\cdot)$ is the force law
- One example is the gravitational force law
  $$G \frac{m_i m_j}{r_{ij}^2} \text{ where } r_{ij} = \text{distance}(x_i, x_j)$$
  $G$ is the gravitational constant

A simple parallel algorithm

- Each processor is assigned $N/P$ particles
- Processors circulate a copy of the particles in a processor ring configuration
- After $P-1$ steps every processor has seen every particle
Ring algorithm

copy positions from mine[ ] into array incoming[ ]

Repeat P times

Compute forces from incoming against mine

Transmit incoming particles to processor (myRank+1) MOD p

Receive incoming particles from processor (myRank-1+p) MOD p

End Repeat

Data parallel variant of the particle computation

• The particles are described by
  • the position and mass array xyzm[ ] + shifted in copy mxyzC
  • Let the force law be a given function $F(xyzm,xyzmC)$

$xyzmC = xyzm$

do while ($t < t_{end}$)

for $i = 1 : n$

  force = force + $F(xyzm,xyzmC)$
  $xyzmC = CSHIFT(xyzmC,1)$ ....

end for

end do
Reduction Operators

Reduce an array to a scalar under an associative binary operation

- sum, product
- minval, maxval
- many others

\[
\text{do while (maxdiff < epsilon)} \\
\text{unew[1:N] = (uold[0:N-1] + uold[2:N+1]) / 2.0} \\
diff = unew - uold \\
\text{absdiff} = \text{abs(diff)} \\
\text{maxdiff} = \text{maxval(absdiff)} \\
\text{enddo}
\]

Conditional Operations

- The following statement
  \[
  \text{dist} = \text{(max(abs(fishp),0.01))}
  \]
  Is equivalent to
  \[
  \text{where (abs(fishp) >= 0.01)} \\
  \text{dist} = \text{abs(fishp)} \\
  \text{elsewhere} \\
  \text{dist} = 0.01 \\
  \text{end where}
  \]
- Recall that in an SIMD architecture, processors can individually opt out of executing an operation
- An SIMD machine needs 2 steps to execute the statement
- What about an MIMD machine?
Data Distribution

- We have three kinds of distributions
  - Block
  - Cyclic
  - Block_cyclic
Block cyclic

Implicit Communication
Array section copies may induce communication

\[ A(1:7) = B(2:8) \quad ! \text{A and B are distributed BLOCKwise} \]
Procedures

- Communication may occur if formal and actual parameters have different layouts
  - Consider $A(1:7) = C(2:8)$, where $A$ is BLOCK distributed and $C$ is CYLIC
  - Important to be aware of this activity

![Diagram of procedures]

Global Communication

```plaintext
X = X(n:1:-1)               ! permutation (reverse)
B = A(Indx(:))             ! "gather"
C(Indx(:)) = B             ! C = "scatter:"
                         ! no duplicates on left!
```

![Diagram of global communication]
Specialized Communication

CSHIFT( array, dim, shift)     ! cyclic shift in one dimension
TRANSPOSE( matrix )            ! matrix transpose

Parallel prefix (scan) operations

X = [4 5 6 7 8 9]
SUM_PREFIX(X)

SUM_PREFIX(X,MASK=[T T F T F T])

SUM_PREFIX(X,SEGMENT=[T T F F F T])