Data Parallelism

Kathy Yelick
yelick@cs.berkeley.edu
www.cs.berkeley.edu/~yelick/cs194f07
Data Parallelism

• Parallelization through data decomposition is very common
• There have been languages and machines designed around the idea that all parallelism comes from data
• “Pure” dataparallel languages have serial semantics, i.e., the parallelism is *implicit* (just a performance hint)

• Example with A, B, and C as arrays:
  \[ C = 2 \times A + B \]
  \[ C = C + A \]

• Evaluation rule: for each statement, evaluate the right-hand-side entirely, then assign to left-hand
Programming Model: Data Parallel

- Single thread of control consisting of parallel operations.
- Parallel operations applied to all (or a defined subset) of a data structure, usually an array
  - Communication is implicit in parallel operators
  - Elegant and easy to understand and reason about
  - Coordination is implicit – statements executed synchronously
  - Similar to Matlab language for array operations

- Drawbacks:
  - Not all problems fit this model
  - Difficult to map onto coarse-grained machines

A = array of all data
fA = f(A)
s = sum(fA)
program Jacobi; /* Nearest neighbor by L. Snyder, 1994 */
config var n : integer = 512; -- Declarations
delta : float = 0.000001;

region R = [1..n, 1..n];
var A, Temp: [R] float;
err : float;

direction north = [-1, 0];
  east = [ 0, 1];
  west = [ 0,-1];
  south = [ 1, 0];

procedure Jacobi();
begin
  [R] A := 0.0; -- Initialization
  [north of R] A := 0.0;
  [east of R] A := 0.0;
  [west of R] A := 0.0;
  [south of R] A := 1.0;

  [R] repeat -- Body
    Temp := (A@north + A@east + A@west + A@south)/4.0;
    err := max<< abs(A - Temp);
    A := Temp;
    until err < delta;
  end;

ZPL is a data parallel language

Example is a nearest neighbor relaxation (Jacobi) code, similar to Game of Life computation.


Constants will be used to “shift” arrays
Initialization

\[ \text{A} \]

[north of R] \( A := 0; \)  \[ \text{east of R} \] \( A := 0; \)

[west of R] \( A := 0; \)  \[ \text{south of R} \] \( A := 1; \)

0's

1's
Stencil Computation in ZPL

Temp := (A@north + A@east + A@west + A@south)/4.0;
Consider HW1 in a Dataparallel Style

How might we do HW1 with vector ops instead of Pthreads?

- \( \vec{x} \) := vector of pseudorandom numbers
- \( \vec{y} \) := black_scholes_value(\( \vec{x} \))
- \( M \) := length(\( \vec{y} \))
- \( \mu := \sum(\vec{y}) / M \)
- \( \sigma := \sqrt{\frac{\sum((\vec{y} - \mu)^2)}{M}} \)

How does summation work?

What might be the problem with the first line?
Sequential Radix Sort: Counting Sort

- Idea: build a histogram of the keys and compute position in answer array for each element

\[ A = [3, 5, 4, 1, 3, 4, 1, 4] \]

- Make temp array B for answer
- Calculate the final position for each element of A
  - A histogram tells you how much space to leave for each
  - Add up left-to-right to find starting location for each value
- Write (in parallel) all the elements of A to B
Counting Sort Pseudo Code

• Counting Sort

static void countingSort(int[] A) {
    int N = A.length;
    int L = min(A), U = max(A);
    int[] count = new int[U-L+2];
    /* these loops are dataparallel */
    for (int i = 0; i < N; i += 1)

    for (int j = 1; j < count.length; j++)
        count[j] += count[j-1];

    …

A = [3, 5, 4, 1, 3, 4, 1, 4]
N = 8
L=1, U=5
count = [0,0,0,0,0,0]
count = [0,2,0,2,3,1]
count = [0,2,2,4,7,8]
Distribution Sort Continued

```java
static void countingSort (int[] A) {
    ...

    int[] B = new int[N];
    for (int i = 0; i < N; i += 1) {
        B[count[A[i]-L]] = A[i];
        count[A[i]-L] += 1;
    }

    // copy back into A
    for (int i = 0; i < N; i += 1)
        A[i] = B[i];
}
```

```
A = [3, 5, 4, 1, 3, 4, 1, 4]
count = [0,2,2,4,7,8]

B = [0, 0, 0, 0, 0, 0, 0, 0]
count = [0,2,3,4,7,8]

B = [0, 0, 3, 0, 0, 0, 0, 0]
count = [0,2,3,4,8,8]

B = [0, 0, 3, 0, 0, 0, 0, 5]
count = [0,2,3,5,8,8]

B = [1, 0, 3, 0, 4, 0, 0, 5]
count = [1,2,3,5,8,8]

B = [1, 0, 3, 3, 4, 0, 0, 5]
count = [1,2,4,5,8,8]

...```

```
B = [1, 1, 3, 3, 4, 4, 4, 5]
```
Radix Sort: Separate Key Into Parts

- Divide keys into parts, e.g., by digits (radix)
- Using counting sort on these each radix:
  - Start with least-significant

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>sat</td>
<td>run</td>
<td>sat</td>
<td>pin</td>
</tr>
<tr>
<td>saw</td>
<td>pin</td>
<td>saw</td>
<td>run</td>
</tr>
<tr>
<td>tip</td>
<td>tip</td>
<td>pin</td>
<td>sat</td>
</tr>
<tr>
<td>run</td>
<td>sat</td>
<td>tip</td>
<td>saw</td>
</tr>
<tr>
<td>pin</td>
<td>saw</td>
<td>run</td>
<td>tip</td>
</tr>
</tbody>
</table>

sort on 3^{rd} character

sort on 2^{nd} character

sort on 1^{st} character
Machine Model 1: SIMD System

- A large number of (usually) small processors.
  - A single “control processor” issues each instruction.
  - Each processor executes the same instruction.
  - Some processors may be turned off on some instructions.
- Machines are very specialized to scientific computing, so they are not popular with vendors (CM2, Maspar)
- Programming model can be implemented in the compiler
  - mapping n-fold parallelism to p processors, n >> p, but it’s hard (e.g., HPF)
Model 2: Vector Machines

- Vector architectures are based on a single processor
  - Multiple functional units
  - All performing the same operation
  - Instructions may specify large amounts of parallelism (e.g., 64-way) but hardware executes only a subset in parallel

- Historically important
  - Overtaken by MPPs in the 90s

- Re-emerging in recent years
  - At a large scale in the Earth Simulator (NEC SX6) and Cray X1

- Fast memory systems, including strided (read a column of a C array) and indexed (write into histogram buckets)

- Key idea: Compiler does some of the difficult work of finding parallelism, so the hardware doesn’t have to
Vector Processors

- Vector instructions operate on a vector of elements
  - These are specified as operations on vector registers

- A supercomputer vector register holds ~32-64 elts
  - The number of elements is larger than the amount of parallel hardware, called vector pipes or lanes, say 2-4
  - The hardware performs a full vector operation in
    - \(\frac{\text{#elements-per-vector-register}}{\text{#pipes}}\)

(logically, performs # elts adds in parallel)

(actually, performs # pipes adds in parallel)
Machine Model 3: SIMD: Single Instruction, Multiple Data

- **Scalar processing**
  - traditional mode
  - one operation produces one result

- **SIMD processing**
  - with SSE / SSE2
  - one operation produces multiple results

- The memory operations are more limited than with vectors. Need to load/store contiguous (aligned) values.
Patterns of Parallel Computation

• Consider a task dependence graph as the representation of a parallel program execution
• It is useful to identify some special cases as a “pattern”

- General task graph with static or dynamic weights & structure
- If all the functions are the same, this is data parallelism
- Divide-and-conquer tree with dynamic weights/structure
Implementing Data Parallelism

- On SIMD machines these languages are “easy”
- On multicore, SPMD, you *could* create threads for each statement, but…

Static parallelism with barriers
Data Parallelism

• Flavors of data parallelism
  • “Pure” data parallel with serial semantics: execute as if each statement is executed serially
  • As functions on right hand-side become more general (larger), this loses the nice serial semantics but is easier to implement
  • At the extreme the entire program is the “operator” to parallelize, and this becomes “Single Program Multiple Data” parallelism

• A key piece of these languages is combining values across arrays into a scalar (e.g., max, sum, etc.)