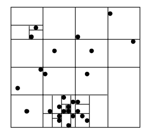
Parallel Algorithmic Techniques

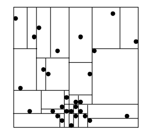
The goal in (practical) parallel algorithm design is to express parameterized parallelism (so it can be scaled to the actual number of processors available) that minimizes communication and synchronization, and has good load balance

Algorithms For N-body Computations

Allocating bodies spatially eases communication load by placing interacting bodies near one another



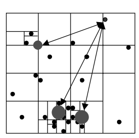
Partitioning Space

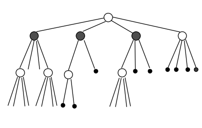


Partitioning Bodies

N-body Representation

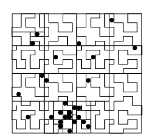
To exploit the fact that only nearby attractions need to be explicitly calculated, partition space, inducing an oct-tree, traverse the oct tree computing the attractions, update positions





Salmon's Space Filling Curve

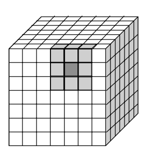
John Salmon of Caltech observed that arranging bodies in the order specfied by a space filling curve places near elements near one another ... the basis of an "out of core" solution



Molecular Dynamics

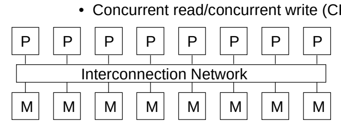
- Uses N-body techniques, but unlike gravitation there is a greater uniformity of elements
- Typically 3D, all pairs of interactions up to a cut-off distance

Protein folding is an important application ... stick protein inside region, flood with water molecules, simulate using basic physical laws.



Algorithms For The PRAM

- The Parallel Random Access Machine was an early and obvious generalization of the RAM:
  - P processors, where typically P <= n
  - Any processor can reference any memory location in "unit" time
  - Handling collisions at locations varies:
    - Exclusive read/exclusive write (EREW)
    - Concurrent read/exclusive write (CREW)
    - Concurrent read/concurrent write (CRCW)



PRAM As A Model

This model is unrealistic ... why?

- Nonlocal memory cannot be referenced in constant time independent of P ... a matter of physics
- Contention and competiton for resources matters in all practical computations

But is it useful?

- "By idealizing the machine it is easier for a programmer to formulate an intial parallel algorithm, and from that transform it to something practical"
- Hardware can be built to make it come true sometimes ... e.g. multithreading; Need slack
- Theoretically interesting

Finding The Maximum On PRAM

Finding the maximum of n values in V is best done with `max<<  V`, but it can be computed using Valiant's algorithm

- In round i use all P (= n) processors to compare for all pairs of 2¦n numbers, finding the Ri max

```
-- Round i
x[j] := 1                        -- initialize
if V[j] <= V[k] then x[j]:=0  -- each k < j
if x[j] = 1 then Ri := V[j]   -- find winner
```

- Repeat same process on the maxes found
- The algorithm finds the global max in time O(*loglog n*)

Results Of Valiant's Algorithm

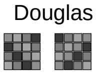
- Very clever
- Theoretically interesting ... the maximum function is extremely easy to compute
- Could this be used in practice? Many cases where max is used, e.g. for loop control Jacobi iteration, there is little slack; would certainly be worse than Ladner/Fischer
- Does this really lead one to develop a practically efficient parallel computation?

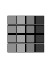
Shared Memory ...

- The CRCW PRAM model may be overly general, but the claim persists that "shared memory is a good simplification for creating initial solutions"
- Abstracts away exactly what is critical -- cost of data motion
- Notice the complications in "orchestrating for performance" Chapter 3 ... tied to shared model, and programmer does compiler's work
- Perhaps the best way to discover initial solutions is to have a "repertoire of techniques" and an accurate cost model in mind

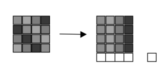
Determinant

Douglas Low's determinant solutions ...


Sum the product reductions of the diagonals, both ways

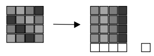
- In ZPL where there is no diagonal *<<
  - Program a diagonal reduction
  - Rotate all Cannon's, reduce down cols, reduce across bottom row 
  - Use gather (<##) to permute as with Cannon
  - Use a new feature for pipelining called mscan

Shift Solution



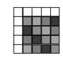
```
[R]             begin
                   ldsum := 0.0;
                   rdsum := 0.0;
                   temp := A;
                   temp2 := A;

                   for row := 2 to n do
[east of R]          wrap temp;
[row..n, 1..n]       temp := temp@east;     -- rotate row i west by i-1 elements

[west of R]          wrap temp2;
[row..n, 1..n]       temp2 := temp2@west;   -- rotate row i east by i-1 elements
                   end;

                   /* Process the forward diagonals */

[south of R]       temp := *<<[R] temp;     -- calculate column products
[south of R]       ldsum := +<< temp;       -- sum the products

                   /* Process the reverse diagonals */

[south of R]       temp2 := *<<[R] temp2;   -- calculate column products
[south of R]       rdsum := +<< temp2;      -- sum the products

                   return ldsum - rdsum;    -- Return the determinant
                end;
```

Permute Solution



```
[R]             begin
                  ldsum := 0.0;
                  rdsum := 0.0;

                  permute := mymod(Index2+(Index1-1), n);
                  temp := <##[Index1, permute] A;

                  permute := mymod(Index2-(Index1-1)+n, n);
                  temp2 := <##[Index1, permute] A;

                  /* Process the forward diagonals */

[south of R]      temp := *<<[R] temp;      -- calculate column products
[south of R]      ldsum := +<< temp;        -- sum the products

                  /* Process the reverse diagonals */

[south of R]      temp2 := *<<[R] temp2;    -- calculate column products
[south of R]      rdsum := +<< temp2;       -- sum the products

                  return ldsum - rdsum;     -- Return the determinant
                end;
```


MSCAN solution



```
[R] begin
             ldpsum := A;
[north of R] ldpsum := 1.0;
[west of R]  ldpsum := 1.0;
[nw of R]    ldpsum := 1.0;

             rdpsum := A;
[north of R] rdpsum := 1.0;
[east of R]  rdpsum := 1.0;
[ne of R]    rdpsum := 1.0;

             scan
               ldpsum *= ldpsum'@nw;
             end;

             scan
               rdpsum *= rdpsum'@ne;
             end;

             for i := 1 to n-1 do
[n-i,n]          temp := +<<ldpsum;
[n,i]            ldpsum *= temp;

[i,1]            temp := +<<rdpsum;
[n,i+1]          rdpsum *= temp;
             end;

[n, 1..n]    return +<<ldpsum - +<<rdpsum;    -- Return determinant
end;
```


Parallel Sorting

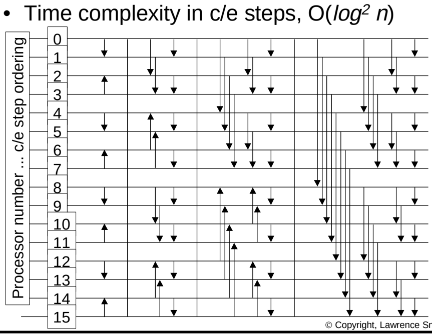
- For small n the counting sort -- compute the final position of each element by n^2 comparisons -- probably works well enough
  - A ZPL solution (based on the Problem Space Promotion idea) was given in Lecture #3
- For huge n -- greater than will fit in all memory -- a merging algorithm is probably best
  - Order local elements independently on each processor -- perfect parallelism
  - Merge the P lists
  - Probably constrained by disk speed
- Any algorithm using compare/exchange can use merge/split to handle larger n

Batcher's Bitonic Sort

- Batcher's bitonic sort was derived from his sorting network, and remains an effective parallel sort -- uses compare/exchange
- An easy way to think of the algorithm is as if the processors formed a hypercube
  - Compare/exchanges are performed between processors differing in the i^{th} bit position with the higher element going to a specific processor
  - Start with LSB position
  - After compare/exchanging i^{th} bit, sort lower bit positions

Batcher's Sort

- Time complexity in c/e steps, O(log^2 *n*)



Sample Sort

- In Sample Sort a random subset of t elements from each processor are forwarded to a designated processor, the sample, and sorted
- The sample is assumed to be distributed like the whole set -- so every t^{th} element is a pivot
- Send pivots to all processors, and they will know where to send the elements they own
- This isn't precise, so final (local) adjustments might be necessary to balance the load

Review of Parallel Processing

CSE596 has sought to put parallel computation in perspective

- Historical antecedents and progress
- Models of parallel computation
- Contemporary machine architectures
- Programming approaches, specifically ZPL
- Parallel algorithms and techniques

- Parallel computation is extremely challenging, but it is the only way to dramatically increase performance

General Conclusions

- Parallel computing is not “standardized” like sequential computing is
  - Some are still debating what model to use, though CTA is working fine for ZPL
  - Using the SMP is straightforward ... moving beyond that is intellectually difficult
- Scientific computation has driven parallel processing research ...could other areas benefit, such as data mining and compute intensive database applications
- TeraFLOPS performance has been achieved

Architectural Conclusions

- Parallel computers differ dramatically
- Architectural diversity creates a portability “gotcha”
  - Whenever a program exploits features not common to all parallel computers, there is a likelihood (certainty?) that performance on other platforms will suffer
  - Never program to the machine
  - A CTA-like model is therefore essential
- Shared memory is expensive and complicated when shared bus is no longer viable

Architectural Conclusions, Continued

What is best machine design?

- Latency hiding Tera is most customized, expensive in many dimensions
  - Wave of the future?
  - Overkill on supporting parallelism?
- Beowulf is most primitive, cheapest
  - Underperforms -- more engineering would help
  - Cheap enough to waste
  - Other applications
- Cray T3D & T3E
  - Global addressing + 1 sided comm is fast/handy
  - Low latency network design effective

Programming Conclusions

- Other than ZPL, the only “portable” approach is message passing with PVM or MPI libraries
  - In message passing the programmer does all of the parallelization, from process spawning to shadow-buffer definition and management
  - Interface, but maybe not the semantics portable
- Many attempts at new languages, libraries for C++, etc. ... why not more successful?
  - Languages ... recognizing when sequential semantics can be parallel is tough
  - Libraries ... poor interface, tough to optimize and customize
- Parallel algorithms is a rich area of study

Prospects

What would it take for parallel computation to be mainstream?

- Make it completely transparent to programmers?
- Teach every programmer from day 1 to write parallel computations
- Evolve from some threaded form like Java