Parallel Algorithms

Lecture 8: Data Management

Ronald Veldema
Data Management

• What partition of the data goes where
• What granularity do we choose
  – Can’t answer this question in general
• When do we do what
  – Can’t answer this question in general
What goes Where?

• Object X on Machine Y?
  – Compute where the data is?
    • Dynamically? Statically?

• How do I get from here to there?

• Use data structures matching the network?
  • Hierarchical data layout if network is a tree
  – Note: we do not require a physical network of some topology
    • Virtual networks
Data mapping (1)

- Map data to CPUs
- In SPMD programs, map partition X to cpu Y
- Block distributing arrays/matrixes such that accesses will be
  - Sequential in memory
  - Require as few network accesses as possible
    - NP hard when trying to also optimize load balancing
Data mapping (2)

• example: given 1D array and 10 cpus:

```c
int j = cpu_number;
parfor (int i=0;i<1000;i++) {
    A[j] = i;
    j += number_of_cpus;
}
```

// Put a[i] on cpu i.
Dynamically mapping work to network

• Divide and conquer
  – Work stealing: steal work from overloaded nodes
    • But with data management:
      – Steal selectively: steal tasks only where you already have some data of

• Master-slave
  – Master pushes work to idle nodes
    • Where master knows the client already has some data
  – Idle nodes pull work from master
    • Master selects a good task for client
Statically mapping work to network (1)

• HPF
  – Programmer specifies data layout for each array
  – For best performance programmer needs to know network layout
    • Fortran programs often ‘tuned’ for a specific network

• MPI
  – Send message to cpu #5, because #5 has the data
Statically mapping work to network (2)

// map ‘a’ to put every 3\textsuperscript{rd} element on a cpu
parfor I=0;I<N;I+=3)
    a[I] *= 10;

// insert code here to remap ‘a’
// to put every 4\textsuperscript{th} element on a cpu
parfor I=0;I<N;I+= 4)
    a[I] *=4;
Statically mapping work to network (3)

• Good HPF compilers try and
  – Minimize the number of remappings
    • Rewrite loops
  – Remap a[I] to a’[I] while still changing the rest of the array
    • Overlap communication with communication
Statically mapping work to network (4)

- **Block cyclic distribution**
  - Divide input in blocks, \# blocks >> tasks
  - Round robin distribute blocks over machines

- **Semi-random block distribution**
  - Divide input in blocks, \#blocks >> tasks
  - Machine X has block HashFunction(I)

- **Random block distribution**
  - Distribute blocks using a *real* random function
  - Maintain directory somewhere of where block I was mapped
Replication (1)

- Normally, in parallel programming
  - All code is replicated
  - Data exists only once
- Data replication is difficult
  - Where to place replicas
  - Where to find closest replica
  - How to keep replicas consistent
Replication (2)

- Replica consistency protocols
  - Invalidation protocols
    - Upon modification, other replicas are deleted
  - Update protocols
    - After modification, broadcast your copy
  - Merge protocols
    - Allow multiple modifying machines, replicas merged in future
      - Highest priority
      - First come, first serve
  - Lazy update
    - Only after a while update other replicas
Replication (2a)

• Case study: globedoc
  – It makes sense that for different objects (i.e. Web pages), different replication strategies are best
    • Some pages/objects should be consistent at all times
      – A ‘current price & product’ page of a company
    • Some pages/objects can be relaxed consistent and replicated aggressively
      – The generic company home-page
Replication (2b)

• Globus
  – Millions of users, hundreds of replicas, files can be VERY large
  – Each file has a logical name
  – Each replica has a logical+physical name
  – File open
    • takes a logical name
    • Asks replica location service for a physical name given logical name
    • We can then use local name to open file
    • Users manually create replicas (using globus_copy_file)
Replication (3)

- Example: lazy partial data replication for dynamically scheduled independent loop iterations

```c
parfor (int I=0; I<N; I++)
    a[I] = b[ f() ] * c[ g() ]
```

Compiler/programmer notices that:
- iterations are independent;
- single iteration depends on \{ I, f(), g() \};
- b, c are readonly. **If large:** replicating b and c everywhere would cost a lot of memory/bandwidth.
Replication (4)

• Maintain a ‘work-queue’ of ‘I’ still to be done
  – At program startup queue contains [0..N]
  – Each processor asks ‘master’ and ‘steals’ an entry

  • Master should give entry where there is a large chance that that machine already has some of the data.
    – Master records where it has sent b[x] and c[x]
    – Master guesses/executes f(), g()
## Replication (5)

<table>
<thead>
<tr>
<th>Master</th>
<th>Request history</th>
<th>Machine 1</th>
<th>Machine 2</th>
<th>Machine 3</th>
<th>Machine 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>f()</td>
<td>g()</td>
<td>target</td>
<td>b[0],c[0]</td>
<td>b[3],c[2]</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>b[1],c[2]</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>b[1],c[2]</td>
<td>b[2],c[3]</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Machine 3 finishes and asks master for work, master should give an iteration where machine 3 is able to reuse one of the replicated b, c arrays.
Prefetching (1)

• Fetch data before its needed
  – Works both with message passing and with shared memory models
• How much to prefetch
• Prefetch data even if you might not need it?
Prefetching (2)

- Most modern processors have some form of prefetch instruction/mechanism
  - Pentium4: prefetch <cache-level> <address>
- Message passing:
  - When requesting datum X, also fetch X’s referred-to data
  - Stride detection for data accesses
Prefetching (3)

- Data consistency issues

\[
\begin{array}{c|c}
\text{Cpu 0} & \text{Cpu 1} \\
\hline
\text{Prefetch a[I]} & \text{a[I] = 34; flush} \\
\text{……} & \\
\text{Use a[I]} & \\
\end{array}
\]

Should cpu 0 see the change to a[I]?
- depends on consistency model, programming model, prefetch semantics…
Prefetching (4)

- Data consistency issues

Request_data_msg (a[i], prefetch = b[i])
  
  /* request_data_msg(b[I]) */
  
  print a[I], b[I]

... Send_data(a[I],b[I])
  b[I] = 323;
...  

(sending prefetch request for ‘b’ with concurrent modifications to ‘b’)
Thread Migration (0)

Machine 0

Machine 1
Thread Migration (1)

• Policy
  – When to migrate

• Mechanism
  – How to migrate

• Different from remote procedure call
  – Thread is interrupted & moved, not a single function as with RPC
  – Execution on another CPU can last arbitrarily long
Thread Migration (2)

• Policy
  – Move a thread to where the most data is to eliminate network latencies (reduce network traffic)
  – Move to less loaded machine (load balancing)
  – Move a thread back after N seconds ?
  – Move only if local state smaller than data in messages ?
  – When to move ?
    • After N times sending a message to a specific remote machine ?
Thread Migration (3)

• Mechanism
  – How to handle open files
  – How to handle pointers
    • Pointers to local objects
    • Pointers on the stack
    • Pointers to objects on the stack
Thread Migration (4)

• Case study: “Harmony”
  – DSM system (shared memory simulation on top of a message passing: distributed memory machine)
  – Minimize both #messages & load imbalance
    • NP hard problem…
  – See paper (handed out)
Pointer jumping (1)

- Quickly find data in set of rooted directed trees
- Initially, the pointers are all pointing on wrong direction
Pointer jumping (2)

• Determine root $S(j)$ of the tree containing node $j$ for each $j$ in the forest of directed trees

• Sequential:
  – Identify roots of forest
  – Reverse links
  – Depth first traversal
Pointer jumping (3)

Step 1: identify roots of forest
Pointer jumping (4)

Step 2: reverse links
Pointer jumping (5)

Step 2: reverse links
Pointer jumping (6)

• Input: forest of rooted directed trees
  – Root has circular reference to itself so $S(I) = I$
• Output: $S(I)$ is root of node $I$

// for every node try and find root in parallel
parfor 1 <= I <= n
    $S(I) := P(I)$ // am I root ?
    while $S(I) != S(S(I))$ // already at root ?
        $S(I) := S(S(I))$ // one level deeper

Jump Pointer Prefetching (1)

- When using a linked data structure, add extra links to double, tripple etc indirections.

Before…

```java
Class LinkedListNode {
    Data d;
    LinkedListNode Next;
}
```

After…

```java
Class LinkedListNode {
    Data d;
    LinkedListNode Next;
    LinkedListNode NextNext;
    LinkedListNode NextNextNext;
}
```
Jump Pointer Prefetching (2)

```java
While (node != null) {
    process(node);
    node = node.Next;
}

While (node != null) {
    prefetch node.Next.Next.Next;
    process(node);
    node = node.Next;
}
```
Network Properties

• static routing / switched
• bisection width
  – how many links can I remove before I divide network in two disconnected networks
• blocking network
  – switch can't be concurrently used by two packets to different destinations
• fully connected
  – Everybody has a connection to everybody else
  – Use virtual networks to create…
• neighbor connected
Omega network (1)

- how to route, crossbar switch
Omega Network (2)

(shift left source address to get to destination address)
Latice (mesh) networks (1)

- Everyone is connected to a number of neighbours
- Many spanning trees possible…
- Everyone is connected to a number of neighbours
- Many spanning trees possible…
Lattice (mesh) networks (3)

- Everyone is connected to a number of neighbours
- Many spanning trees possible…
Map tree to mesh:
break all cycles by not using some links
Latice (mesh) networks (5)

• Systolic matrix multiplication
  – NxN matrix
  – NxN mesh
• Systolic = matrixes are slowly ‘absorbed/consumed’ by the network
Latice (mesh) networks (6)

• Each time step
  – Cpu x,y computes
    • c[x,y] += a[x,m]*b[m,y]
  – Sends a[x,m] to east neighbour
  – Sends b[m,y] to north neighbour
• In n steps, everybody will have seen all ‘m’
  – Log(n) complexity
Star networks
Star networks

Master:
   JobDistributor = new JobDistributor();

Each slave machine:
   Master m = get_master();
   while not done:
       job = master.jobDistributor.get_job();
       job.compute();
Hypercube networks (1)

*node I and J are connected if address is 1 bit less or more
Hypercube networks (2)

- Sum array elements on a hypercube
  - Element $a[I]$ is on cpu $I$
  - Array size = $N$ then $\log(n)$ cpus ($n=2^d, d=\text{dimension}$)
  - Store result on cpu 0

// $d = \text{dimension}$, $x$ iterates over subdimensions
// I am cpu ‘I’
for $x=d-1$ to 0
  if $0 \leq I \leq 2^x-1$ then // should I participate this iteration ?
    $a[I] = a[I] + a[I^x]$
Hypercube networks

• Given a 8 node cluster: n=8, then $8=2^3$, we thus have a 3D hypercube

• First iter:

• Second iter:

• Third iter:
  – $a[0] = a[0…7]$

• Question: did we use static or dynamic routing ?
Bus based networks (1)
Bus based networks (2)

- Fast broadcast
  - Everybody receives a packet at the same time

- Bus snooping
  - Everybody listens to all packets, even the packets not destined for you
Shared Memory Broadcast...

class A {
    volatile int value;

    synchronized void bcast(int x) {
        value = x;
    }
}

On a SMP machine, with a snooping bus:
- the ’value’ assign is seen by all processors
- all processors evict the old value from the cache
Matrix Transposition

• Diagonally mirror a 2D matrix

Matrix = NxM

\[
\begin{array}{cc}
(1,1) & (1,N) \\
...............
\end{array}
\]

then transpose =

\[
\begin{array}{cc}
(1,1) & (M,1) \\
...............
\end{array}
\]

now map \((i,j)\) to processor \(P(i)\)

1) Matrix transposition = data remapping problem
2) Won’t it be better to allocate matrix in transposed form?
Matrix Multiplication (1)

- Let A and B be n*n matrices
  - compute $C = A \times B$

$$C = \begin{bmatrix}
  a_{11} & a_{12} & a_{13} & a_{14} \\
  a_{21} & a_{22} & a_{23} & a_{24} \\
  a_{31} & a_{32} & a_{33} & a_{34} \\
  a_{41} & a_{42} & a_{43} & a_{44}
\end{bmatrix} \begin{bmatrix}
  b_{11} & b_{12} & b_{13} & b_{14} \\
  b_{21} & b_{22} & b_{23} & b_{24} \\
  b_{31} & b_{32} & b_{33} & b_{34} \\
  b_{41} & b_{42} & b_{43} & b_{44}
\end{bmatrix}$$

$$c_{ik} = \sum_{j=1}^{n} a_{ij} \times b_{jk}$$

This requires $n$ multiplications and $n-1$ additions per element of $C$
So it takes $n^3$ multiplications and $n^3 - n^2$ additions to compute $C$
Matrix Multiplication (2)

- Replicate A, B (read-only after all !)
- Each cpu computes row J of C[I, J]
- Afterwards, merge rows to create complete C matrix
Improving Matrix Memory Layout

(1)

Access element $A[I][j]$ of $N \times N$ matrix using:

$$\text{ptr} + (N \times I) + J$$
Improving Matrix Memory Layout (2)

Access element $A[I][j]$ of $N \times N$ matrix using:

$$\text{ptr} + (n \times I) + J$$

Access element $A[I][j]$ of $N \times N$ matrix using?
Mapping Matrix to Network (1)

- Map:
  
  For (int I=0; I<N; I++)
  
  For (int I=0; I<N; I++)
  
  matrix[I][I] = matrix[I*3][I*3];

- To a 2D mesh with N cpus
  
  • Put every 3rd element on a cpu
# Mapping Matrix to Network (2)

<table>
<thead>
<tr>
<th></th>
<th>0,0</th>
<th>0,3</th>
<th>0,6</th>
<th>0,9</th>
<th>0,12</th>
<th>0,15</th>
<th>0,18</th>
</tr>
</thead>
<tbody>
<tr>
<td>3,0</td>
<td>3,3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6,0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9,0</td>
<td>9,18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Mapping Matrix to Network (3)

• Trade-offs:
  • How to map array with different concurrent access patterns?
  • Example:
    \[ A[I] = A[j*2] \times N \]
    \[ A[I] = A[j*3] \times M \]
    Choose between every 2\textsuperscript{nd} or every 3\textsuperscript{rd} on a cpu….
Mapping Matrix to Network (4)

- Compile time unknowns can make a-priori data-mapping hard
  - A[ B[I] ]
  - A[ string2int(commandline[1]) ]
Barrier on a hierarchical network (1)

(barrier = point in code where cpus wait until all cpus have reached that point on the program)

Wants to enter barrier.
Barrier on a hierarchical network (2)

(barrier = point in code where cpus wait until all cpus have reached that point on the program)

1) Wants to enter barrier.
2) Sends “enter_barrier” to parent, parent sets “child_block_counter” to ‘1’
Barrier on a hierarchical network (3)

(barrier = point in code where cpus wait until all cpus have reached that point on the program)

1) Wants to enter barrier.
2) Sends “enter_barrier” to parent, parent sets “child_block_counter” to ‘1’
3) Same as 2, parent sends block message to parent
Barrier on a hierarchical network (4)

(barrier = point in code where cpus wait until all cpus have reached that point on the program)

1) Root receives a block counter for each of its Child nodes and sends a ‘barrier_release’ message
Barrier on different networks

- Hypercube
  - Same as tree
- Star
  - Central barrier on center node
- Mesh
  - Create virtual topology
    - spanning tree
Map-Reduce

• Idea: data-parallel computation with reduction
  – 'data-parallel'
    • apply a function to every element of a data-structure (in parallel):
      – “x[i].foo()” or “x[i] = func(data[i]);”
  – 'reduction/reduce':
    • Combine all x[i]'s to a single result

• Implemented in Google's map-reduce / Apache Hadoop
## Map-Reduce

- **Matrix-Multiplication using map-reduce**

```java
// parallel without map-reduce:
parfor (int x = 0 to N)
{
    parfor (int y = 0 to N)
    {
        parfor (int k = 0 to N)
        {
            float f = b[y, k];
            float g = a[k, x];
            c[y, x] += (f * g);
        }
    }
}
```

```java
// parallel with map-reduce:

// results of 'map'
float tmp_result[N, N, N];
void map(int x, y, k) {
    tmp_result[y, k, x] = b[y, k] * a[k, x];
}
void reduce(int x, y, k) {
    c[y, x] += tmp_result[y, k, x];
}
```

```java
// map-reduce library:
foreach (x, y, k) {
    map(x, y, k)
}
foreach (x, y, k) {
    reduce(x, y, k)
}
```
Map-Reduce

• Create frequency table of words in file:

```cpp
list<Word> map(File f)
{
    list<Word> out = new list<Word>();
    while (! eof_of_file(f))
    {
        Word w = get_next_word(f);
        out.add(w);
    }
    return out;
}

class Result {
    int frequency[MAX_WORD];
};

Result reduce(Word p1) {
    Result r = new Result();
    int index = p1.get_index();
    r.frequency[index] ++;
    return r;
}

Result reduce(Result r, Pair p1) {
    int index = p1.get_index();
    r.frequency[index] ++;
    return r;
}

Result reduce(Result r1, Result r2) {
    for i in 0 to MAX_WORD
        r1.frequency[i] += r2.frequency[i];
    return r1;
}
```
Agents

• An agent is
  – An object (data + operations on that data)
  – Active
    • Each agent runs inside its own thread on some machine

• Agents interact by exchanging messages
  – Each agent has a 'mail-box'
  – Messages are dropped in the mailbox
  – Agent can, when its convenient, see if mailbox non-empty
  – Location independent! (agents can move around the network, messages are forwarded)
Agent: example:

- Microsoft “Axum”, is an agent based language (also: Occam, Erlang, Salsa)

```csharp
channel Adder
{
    input int Num1;
    input int Num2;
    output int Sum;
}

agent AdderAgent : channel Adder
{
    public AdderAgent()
    {
        int result = receive(PrimaryChannel::Num1) +
                      receive(PrimaryChannel::Num2);
        PrimaryChannel::Sum <-- result;
    }
}

agent MainAgent : channel Microsoft.Axum.Application
{
    public MainAgent()
    {
        var adder = AdderAgent.CreateInNewDomain();
        adder::Num1 <-- 10;
        adder::Num2 <-- 20;
        // do something useful ...
        var sum = receive(adder::Sum);
        Console.WriteLine(sum);
        PrimaryChannel::ExitCode <-- 0;
    }
}
```