Probability Convergence in a Multithreaded Counting Application

Chad Scherrer[1] Nathaniel Beagley[1] Jarek Nieplocha[1]
Andrés Márquez[1] John Feo[2] Daniel Chavarría-Miranda[1]

[1]Pacific Northwest National Laboratory

[2]Cray, Incorporated

Workshop on Multithreaded Architectures and Applications
March 30, 2007

Outline

Outline

The Problem

- We are given data in the form of a sequence of tuples,

$$[(a_1, b_1, c_1), \ldots, (a_n, b_n, c_n)] .$$

- We wish to be able to quickly answer queries of the form

$$\text{count}\,(A = a_2, B = *, C = c_{17}) .$$

- Note that some variables may be unspecified.
- In many modeling contexts, the queries may take on a more restricted form, *e.g.*, at most two fixed values. We wish to take advantage of any such structure.

The Problem

- We are given data in the form of a sequence of tuples,

$$[(a_1, b_1, c_1), \ldots, (a_n, b_n, c_n)] .$$

- We wish to be able to quickly answer queries of the form

$$\text{count}\,(A = a_2, B = *, C = c_{17}) .$$

- Note that some variables may be unspecified.
- In many modeling contexts, the queries may take on a more restricted form, *e.g.*, at most two fixed values. We wish to take advantage of any such structure.

The Problem

- We are given data in the form of a sequence of tuples,

$$[(a_1, b_1, c_1), \ldots, (a_n, b_n, c_n)] .$$

- We wish to be able to quickly answer queries of the form

$$\text{count}\,(A = a_2, B = *, C = c_{17}) .$$

- Note that some variables may be unspecified.
- In many modeling contexts, the queries may take on a more restricted form, *e.g.*, at most two fixed values. We wish to take advantage of any such structure.

The Problem

- We are given data in the form of a sequence of tuples,

$$[(a_1, b_1, c_1), \ldots, (a_n, b_n, c_n)] .$$

- We wish to be able to quickly answer queries of the form

$$\text{count}\,(A = a_2, B = *, C = c_{17}) .$$

- Note that some variables may be unspecified.
- In many modeling contexts, the queries may take on a more restricted form, *e.g.*, at most two fixed values. We wish to take advantage of any such structure.

Our Approach

- Design a tree structure for storing multivariate count data, allowing a user-specified nesting.
- Queries can be answered at any time as the tree is populated. For testing, we assume each new observation has a corresponding set of queries.
- Parallelize by breaking sequence into blocks, possibly introducing a race condition.
- Prove a bound on the effects of the race condition that shrink as data volume grows.

Our Approach

- Design a tree structure for storing multivariate count data, allowing a user-specified nesting.
- Queries can be answered at any time as the tree is populated. For testing, we assume each new observation has a corresponding set of queries.
- Parallelize by breaking sequence into blocks, possibly introducing a race condition.
- Prove a bound on the effects of the race condition that shrink as data volume grows.

Our Approach

- Design a tree structure for storing multivariate count data, allowing a user-specified nesting.
- Queries can be answered at any time as the tree is populated. For testing, we assume each new observation has a corresponding set of queries.
- Parallelize by breaking sequence into blocks, possibly introducing a race condition.
- Prove a bound on the effects of the race condition that shrink as data volume grows.

Our Approach

- Design a tree structure for storing multivariate count data, allowing a user-specified nesting.
- Queries can be answered at any time as the tree is populated. For testing, we assume each new observation has a corresponding set of queries.
- Parallelize by breaking sequence into blocks, possibly introducing a race condition.
- Prove a bound on the effects of the race condition that shrink as data volume grows.

Outline

Introducing PDtrees

- An *ADtree* [Moore and Lee] is a nested data structure that stores "All Dimensions", in that counts are stored for every possible combination of variables.
- Storage costs for an ADtree depend on the number of variables, the number of levels of each variable, and the dependence structure among the variables.
- The time required to populate an ADtree is linear in the number of observations but exponential in the number of variables.
- If this expense is unacceptable, a *PDtree* (for "Partial Dimensions") might be appropriate.
- Nesting structure is specified in an auxilliary data structure called a *guide tree*.
- Nesting structure can be changed without the need to recompile.

Introducing PDtrees

- An *ADtree* [Moore and Lee] is a nested data structure that stores "All Dimensions", in that counts are stored for every possible combination of variables.
- Storage costs for an ADtree depend on the number of variables, the number of levels of each variable, and the dependence structure among the variables.
- The time required to populate an ADtree is linear in the number of observations but exponential in the number of variables.
- If this expense is unacceptable, a *PDtree* (for "Partial Dimensions") might be appropriate.
- Nesting structure is specified in an auxilliary data structure called a *guide tree*.
- Nesting structure can be changed without the need to recompile.

Introducing PDtrees

- An *ADtree* [Moore and Lee] is a nested data structure that stores “All Dimensions”, in that counts are stored for every possible combination of variables.
- Storage costs for an ADtree depend on the number of variables, the number of levels of each variable, and the dependence structure among the variables.
- The time required to populate an ADtree is linear in the number of observations but exponential in the number of variables.
- If this expense is unacceptable, a *PDtree* (for “Partial Dimensions”) might be appropriate.
- Nesting structure is specified in an auxilliary data structure called a *guide tree*.
- Nesting structure can be changed without the need to recompile.

Introducing PDtrees

- An *ADtree* [Moore and Lee] is a nested data structure that stores "All Dimensions", in that counts are stored for every possible combination of variables.
- Storage costs for an ADtree depend on the number of variables, the number of levels of each variable, and the dependence structure among the variables.
- The time required to populate an ADtree is linear in the number of observations but exponential in the number of variables.
- If this expense is unacceptable, a *PDtree* (for "Partial Dimensions") might be appropriate.
- Nesting structure is specified in an auxilliary data structure called a *guide tree*.
- Nesting structure can be changed without the need to recompile.

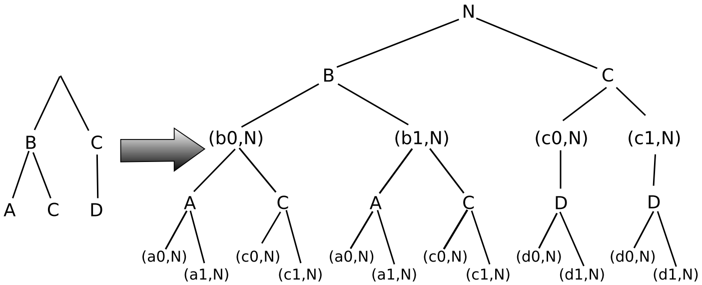
Introducing PDtrees

- An *ADtree* [Moore and Lee] is a nested data structure that stores "All Dimensions", in that counts are stored for every possible combination of variables.
- Storage costs for an ADtree depend on the number of variables, the number of levels of each variable, and the dependence structure among the variables.
- The time required to populate an ADtree is linear in the number of observations but exponential in the number of variables.
- If this expense is unacceptable, a *PDtree* (for "Partial Dimensions") might be appropriate.
- Nesting structure is specified in an auxilliary data structure called a *guide tree*.
- Nesting structure can be changed without the need to recompile.

Introducing PDtrees

- An *ADtree* [Moore and Lee] is a nested data structure that stores “All Dimensions”, in that counts are stored for every possible combination of variables.
- Storage costs for an ADtree depend on the number of variables, the number of levels of each variable, and the dependence structure among the variables.
- The time required to populate an ADtree is linear in the number of observations but exponential in the number of variables.
- If this expense is unacceptable, a *PDtree* (for “Partial Dimensions”) might be appropriate.
- Nesting structure is specified in an auxilliary data structure called a *guide tree*.
- Nesting structure can be changed without the need to recompile.

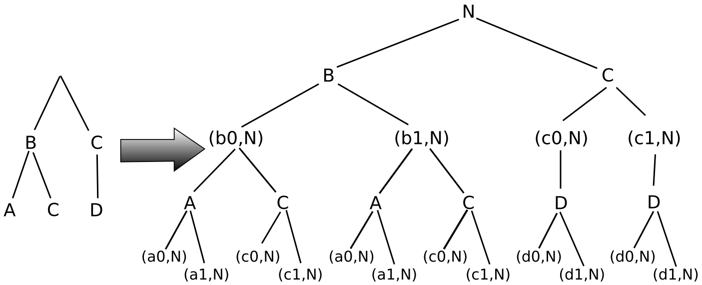
Building a PDtree from a Guide Tree



Efficiently storing data for a Bayesian network

- Start with Bayesian network $A \to B \to C \to D$.
- Only need to store counts for $\{AB, B, BC, C, CD\}$.
- This is equivalent to storing $\{B, C, A|B, C|B, D|C\}$.

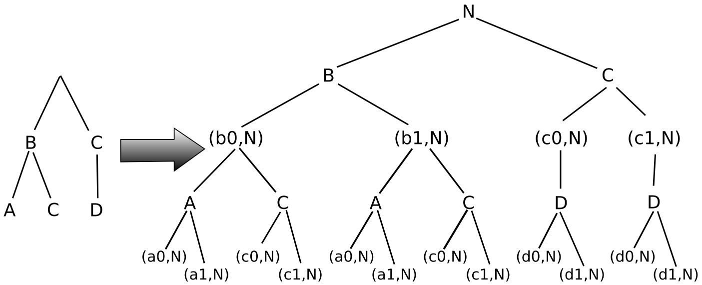
Building a PDtree from a Guide Tree



Efficiently storing data for a Bayesian network

- Start with Bayesian network $A \to B \to C \to D$.
- Only need to store counts for $\{AB, B, BC, C, CD\}$.
- This is equivalent to storing $\{B, C, A|B, C|B, D|C\}$.

Building a PDtree from a Guide Tree



Efficiently storing data for a Bayesian network

- Start with Bayesian network $A \to B \to C \to D$.
- Only need to store counts for $\{AB, B, BC, C, CD\}$.
- This is equivalent to storing $\{B, C, A|B, C|B, D|C\}$.

Outline

An Implementation on the Cray MTA-2

- First node for each variable is implemented as an array, because all possible values will be taken on.
- Lower branches are implemented as linked lists, and values become increasingly sparse.

An Implementation on the Cray MTA-2

- First node for each variable is implemented as an array, because all possible values will be taken on.
- Lower branches are implemented as linked lists, and values become increasingly sparse.

Multithreaded List Insertion, Take 1


```
while true {
  ptr = readfe(node.next)
  if ptr is null
    ptr = memory for new node
    initialize new node
    writeef(node.next, ptr)
    break
  else if next node is the one I want
    increment counter
    writeef(node.next, ptr)
    break
  else
    writeef(node.next, ptr)
  end if
} end while
```

- Branches in a PDtree are currently implemented using a linked list.
- Synchronized read and write implemented with readfe and writeef, resp.
- This version is overly serial.
- Critical section per link rather than only at the end of the list.

Multithreaded List Insertion, Take 1


```
while true {
  ptr = readfe(node.next)
  if ptr is null
    ptr = memory for new node
    initialize new node
    writeef(node.next, ptr)
    break
  else if next node is the one I want
    increment counter
    writeef(node.next, ptr)
    break
  else
    writeef(node.next, ptr)
  end if
} end while
```

- Branches in a PDtree are currently implemented using a linked list.
- Synchronized read and write implemented with readfe and writeef, resp.
- This version is overly serial.
- Critical section per link rather than only at the end of the list.

Multithreaded List Insertion, Take 1

```
while true {
  ptr = readfe(node.next)
  if ptr is null
    ptr = memory for new node
    initialize new node
    writeef(node.next, ptr)
    break
  else if next node is the one I want
    increment counter
    writeef(node.next, ptr)
    break
  else
    writeef(node.next, ptr)
  end if
} end while
```

- Branches in a PDtree are currently implemented using a linked list.
- Synchronized read and write implemented with readfe and writeef, resp.
- This version is overly serial.
- Critical section per link rather than only at the end of the list.

Multithreaded List Insertion, Take 1

```
while true {
  ptr = readfe(node.next)
  if ptr is null
    ptr = memory for new node
    initialize new node
    writeef(node.next, ptr)
    break
  else if next node is the one I want
    increment counter
    writeef(node.next, ptr)
    break
  else
    writeef(node.next, ptr)
  end if
} end while
```

- Branches in a PDtree are currently implemented using a linked list.
- Synchronized read and write implemented with readfe and writeef, resp.
- This version is overly serial.
- Critical section per link rather than only at the end of the list.

Multithreaded List Insertion, Take 1

```
while true {
  ptr = readfe(node.next)
  if ptr is null
    ptr = memory for new node
    initialize new node
    writeef(node.next, ptr)
    break
  else if next node is the one I want
    increment counter
    writeef(node.next, ptr)
    break
  else
    writeef(node.next, ptr)
  end if
} end while
```

- Branches in a PDtree are currently implemented using a linked list.
- Synchronized read and write implemented with readfe and writeef, resp.
- This version is overly serial.
- Critical section per link rather than only at the end of the list.

Multithreaded List Insertion, Take 2


```
while true {
  ptr = node.next
  if ptr is null
    ptr = readfe(node.next)
    if ptr is not null then continue
    ptr = memory for new node
    initialize new node
    writeef(node.next, ptr)
    break
  else if next node is the one I want
    increment counter
    writeef(node.next, ptr)
    break
  else
    writeef(node.next, ptr)
    node = ptr
  end if
} end while
```

- Changes are shown in red.
- Test the pointer before locking it.
- Must retest after readfe in case another threads grabs the lock to insert a new node.
- This version scales linearly up to 32 processors.

Multithreaded List Insertion, Take 2


```
while true {
  ptr = node.next
  if ptr is null
    ptr = readfe(node.next)
    if ptr is not null then continue
    ptr = memory for new node
    initialize new node
    writeef(node.next, ptr)
    break
  else if next node is the one I want
    increment counter
    writeef(node.next, ptr)
    break
  else
    writeef(node.next, ptr)
    node = ptr
  end if
} end while
```

- Changes are shown in red.
- Test the pointer before locking it.
- Must retest after readfe in case another threads grabs the lock to insert a new node.
- This version scales linearly up to 32 processors.

Multithreaded List Insertion, Take 2

```
while true {
  ptr = node.next
  if ptr is null
    ptr = readfe(node.next)
    if ptr is not null then continue
    ptr = memory for new node
    initialize new node
    writeef(node.next, ptr)
    break
  else if next node is the one I want
    increment counter
    writeef(node.next, ptr)
    break
  else
    writeef(node.next, ptr)
    node = ptr
  end if
} end while
```

- Changes are shown in red.
- Test the pointer before locking it.
- Must retest after readfe in case another threads grabs the lock to insert a new node.
- This version scales linearly up to 32 processors.

Multithreaded List Insertion, Take 2

```
while true {
  ptr = node.next
  if ptr is null
    ptr = readfe(node.next)
    if ptr is not null then continue
    ptr = memory for new node
    initialize new node
    writeef(node.next, ptr)
    break
  else if next node is the one I want
    increment counter
    writeef(node.next, ptr)
    break
  else
    writeef(node.next, ptr)
    node = ptr
  end if
} end while
```

- Changes are shown in red.
- Test the pointer before locking it.
- Must retest after readfe in case another threads grabs the lock to insert a new node.
- This version scales linearly up to 32 processors.

Multithreaded List Insertion, Take 2


```
while true {
  ptr = node.next
  if ptr is null
    ptr = readfe(node.next)
    if ptr is not null then continue
    ptr = memory for new node
    initialize new node
    writeef(node.next, ptr)
    break
  else if next node is the one I want
    increment counter
    writeef(node.next, ptr)
    break
  else
    writeef(node.next, ptr)
    node = ptr
  end if
} end while
```

- Changes are shown in red.
- Test the pointer before locking it.
- Must retest after readfe in case another threads grabs the lock to insert a new node.
- This version scales linearly up to 32 processors.

Outline

Sequential Vs. Parallel Counts

n	Sequential	Parallel, 3 threads
0	$-$	$- - -$
1	$+-$	$- - +-$
2	$+ + -$	$+ - + - -$
3	$+ + +-$	$+ - + + --$
4	$+ + + + -$	$+ - + + - + -$

In general, using k threads in the parallel implementation gives a maximal count deviation of $k - 1$.

Sequential Vs. Parallel Counts

n	Sequential	Parallel, 3 threads
0	$-$	$- - -$
1	$+-$	$- - +-$
2	$+ + -$	$+ - + - -$
3	$+ + +-$	$+ - + + --$
4	$+ + + + -$	$+ - + + - + -$

In general, using k threads in the parallel implementation gives a maximal count deviation of $k - 1$.

Maximal Count Deviation

Lemma

Let $c_{seq}(n)$ and $c_{par}(n)$ be the number of times a particular collection of variables takes on a specified configuration, given the number n of observations so far, for a sequential and parallel implementation, respectively. If the parallel implementation uses k threads, then

$$|c_{par}(n) - c_{seq}(n)| < k.$$

Now let $\hat{p}_{par}(n) = \frac{c_{par}(n)}{n}$ and $\hat{p}_{seq}(n) = \frac{c_{seq}(n)}{n}$ be the estimated probabilities of a given value after n observations.

Maximal Count Deviation

Lemma

Let $c_{seq}(n)$ and $c_{par}(n)$ be the number of times a particular collection of variables takes on a specified configuration, given the number n of observations so far, for a sequential and parallel implementation, respectively. If the parallel implementation uses k threads, then

$$\left|c_{par}(n) - c_{seq}(n)\right| < k.$$

Now let $\hat{p}_{par}(n) = \frac{c_{par}(n)}{n}$ and $\hat{p}_{seq}(n) = \frac{c_{seq}(n)}{n}$ be the estimated probabilities of a given value after n observations.

Probability Convergence

Theorem

For a counting application, suppose a sequential implementation is compared to a parallel implementation using k threads, and let n be the number of observations. The estimated probabilities are then related by

$$\hat{p}_{par}(n) = \hat{p}_{seq}(n) + O\left(\frac{k}{n}\right).$$

Proof.

Using the result from the lemma,

$$\left|\hat{p}_{par}(n) - \hat{p}_{seq}(n)\right| = \left|\frac{c_{par}(n)}{n} - \frac{c_{seq}(n)}{n}\right| < \frac{k}{n}.$$

Probability Convergence

Theorem

For a counting application, suppose a sequential implementation is compared to a parallel implementation using k threads, and let n be the number of observations. The estimated probabilities are then related by

$$\hat{p}_{par}(n) = \hat{p}_{seq}(n) + O\left(\frac{k}{n}\right).$$

Proof.

Using the result from the lemma,

$$\left|\hat{p}_{par}(n) - \hat{p}_{seq}(n)\right| = \left|\frac{c_{par}(n)}{n} - \frac{c_{seq}(n)}{n}\right| < \frac{k}{n}.$$

Summary

- A PDtree data structure has similar benefits to an ADtree, but allows specification of the nesting structure, leading to mmory savings and speed improvements.
- Parallelism is easily achieved on a Cray MTA-2, but a race condition is introduced.
- The numeric effect of this race condition decays as $\frac{1}{n}$.

Summary

- A PDtree data structure has similar benefits to an ADtree, but allows specification of the nesting structure, leading to mmory savings and speed improvements.
- Parallelism is easily achieved on a Cray MTA-2, but a race condition is introduced.
- The numeric effect of this race condition decays as $\frac{1}{n}$.

Summary

- A PDtree data structure has similar benefits to an ADtree, but allows specification of the nesting structure, leading to mmory savings and speed improvements.
- Parallelism is easily achieved on a Cray MTA-2, but a race condition is introduced.
- The numeric effect of this race condition decays as $\frac{1}{n}$.