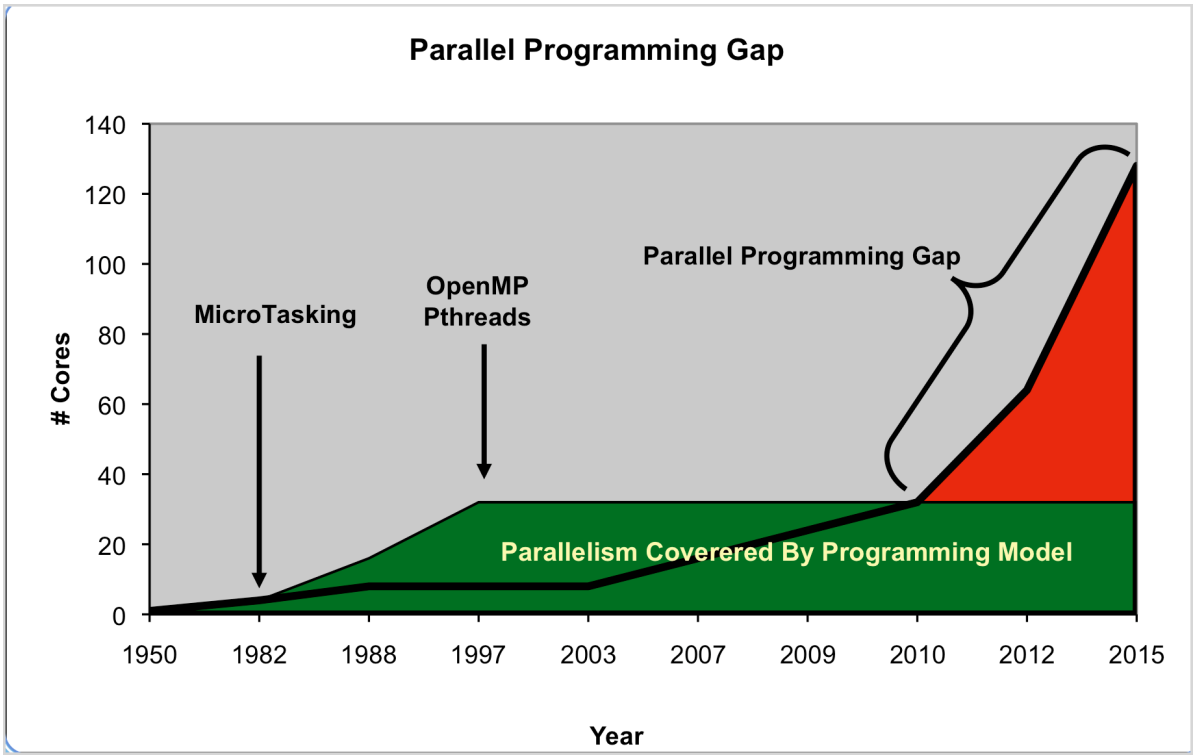
A Comparison Of Shared Memory Parallel Programming Models

Jace A Mogill
David Haglin

Pacific Northwest
NATIONAL LABORATORY

Parallel Programming Gap

- Not many innovations...
  - Memory semantics unchanged for over 50 years
  - 2010 Multi-Core x86 programming model identical to 1982 Symmetric Multi-Processor programming model
- Unwillingness to adopt new languages
  - Users want to leverage existing investments in code
  - Prefer to incrementally migrate to parallelism



- **OpenMP/MicroTasking** – Data parallelism
- **Pthreads** – Task parallelism
- **Wishful Thinking** – Mixed task and data parallelism

Expressing Synchronization and Parallelism

Parallelism and Synchronization are Orthogonal

		Parallelism	
		Explicit	**Implicit**
Synchronization	**Code**	Pthreads/ OpenMP	Vectorization
	Data	MTA	Dataflow

Synchronization and parallelism primitives can be mixed

OpenMP mixed with Atomic Memory Operations
MTA synchronization mixed with OpenMP parallel loops
OpenMP mixed with Pthread Mutex Locks
OpenMP or Pthreads mixed with Vectorization
All of the above mixed with MPI, UPC, CAF, etc.

Thread-Centric versus Data-Centric

TASK: Map millions of degrees of parallelism onto tens of (thousands of) processors

Thread-Centric

Manage Threads

Optimizes for specific machine organization

Requires careful scheduling of moving data to/from thread

Difficult to load balance dynamic and nested parallel regions

Data-Centric

Manage Data Dependencies

Compiler is already doing this for ILP, loop optimization, and vectorization

Optimizes for concurrency, which is performance portable

Moving task to data is a natural option for load balancing

Lock-Free and Wait-Free Algorithms

- Lock Free and Wait Free Algorithms...
  - Don't really exist
  - Only embarrassingly Parallel algorithms don't use synchronization
- Compare And Swap...
  - is not lock free or wait-free
  - has no concurrency
  - is a synchronization primitive which corresponds to mutex try-lock in the Pthreads programming model

Compare And Swap

- **LOCK# CMPXCHG** – x86 Locked Compare and Exchange
- Programming Idioms
  - Similar to mutex try-lock
  - Mutex locks can spin try-lock or yield to the OS/runtime
  - So Called Lock-Free Algorithms
    - Manually coded secondary lock handler
      - Manually coded tertiary lock handler...
        - All this try-lock handler work is not algorithmically efficient...
          - It's Lock-Free Turtles all the way down...
- Implementation
  - Instruction in all i386 and later processors
  - Efficient for processors sharing caches and memory controllers
  - Not efficient or fair for non-uniform machine organizations

Atomic Memory Operations Do Not Scale

It is not possible to go 10,000 way parallel on one piece of data.

Thread-Centric Parallel Regions

OpenMP

Implied fork/join scaffolding

- Parallel regions are separate from loops
  - Unannotated loops: Every thread executes all iterations
  - Annotated loops: Loops are decomposed among existing threads
- Joining Threads
  - Exit parallel region
  - Barriers

Pthreads

Fully explicit scaffolding

- Forking Threads
  - One new thread per PthreadCreate()
  - Loops or trees required to start multiple threads
  - Flow control
    - PthreadBarrier
    - Mutex Lock
- Joining Threads
  - PthreadJoin
  - return()

Data-Centric Parallel Regions on XMT

Multiple loops, with different trip counts, after restructuring a reduction, all in a single parallel region:

```
Parallel region   1 in foo
       Multiple processor implementation
       Requesting at least 50 streams

Loop   2 in foo in region 1
       In parallel phase 1
       Dynamically scheduled, variable chunks, min size = 26
       Compiler generated

Loop   3 in foo at line 7 in loop 2
       Loop summary: 1 loads, 0 stores, 1 floating point operat
                1 instructions, needs 50 streams for full utili
                pipelined

Loop   4 in foo in region 1
       In parallel phase 2
       Dynamically scheduled, variable chunks, min size = 8
       Compiler generated

Loop   5 in foo at line 10 in loop 4
       Loop summary: 2 loads, 1 stores, 2 floating point operations
                3 instructions, needs 44 streams for full utilization
                pipelined
```

```
      | void foo(int n, double* restrict a,
      |          double* restrict b,
      |          double* restrict c,
      |          double* restrict d)
      | {
      |   int i, j;
      |   double sum = 0.0;
      |
      |   for (i = 0; i < n; i++)
3 P:$ |     sum += a[i];
** reduction moved out of 1 loop
      |
      |   for (j = 0; j < n/2; j++)
5 P   |     b[j] = c[j] + d[j] * sum;
      | }
```

Parallel Histogram

Thread Centric *Time = $N_{elements}$*

```
PARALLEL-DO  i = 0 .. Nelements-1
   j = 0
   while(j < Nbins &&
         elem[i] < binmax[j])
       j++
   BEGIN CRITICAL-REGION
     counts[j]++ ← Only 1 thread at a time
   END CRITICAL-REGION
```

- Critical region around update to counts array
- Serial bottleneck in critical region
- Wastes potential concurrency

Data Centric *Time = $N_{elements}/N_{bins}$*

```
PARALLEL-DO  i = 0 .. Nelements-1
  j = 0
  while(j < Nbins &&
        elem[i] < binmax[j])
      j++
  INT_FETCH_ADD(counts[j], 1) ← Updates
                                are atomic
```

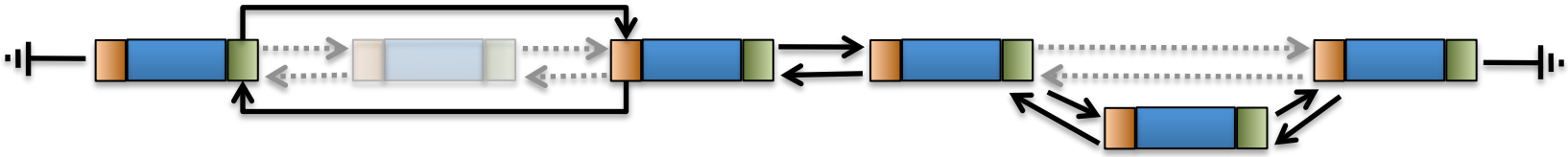
Updates to count table are atomic:

- Requires abundant fine grained synchronization

All concurrency can be exploited:

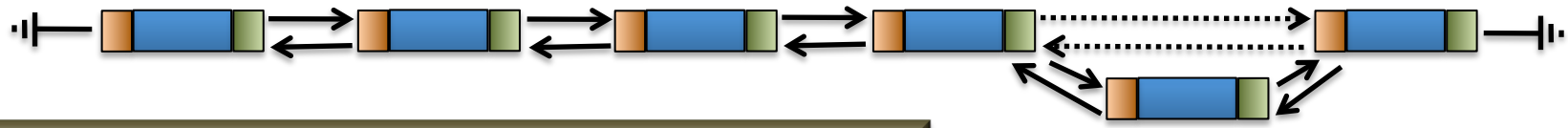
- Maximum concurrency limited to number of bins
- Every bin updated simultaneously

Linked List Manipulation



- Insertion Sort
  - **Time = N*N/2** – Inserting from same side every time
  - **Time = N*N/4** – Insert at head or tail, whichever is nearer
- Unlimited concurrency during search
- Concurrency during manipulations
  - Thread Parallelism: One update to list at a time
  - Data Parallelism: Between each pair of elements
    - Grow list length by 50% on every step
- Two phase insertion (search, then lock and modify)

Two-Phase List Insertion (OpenMP)



1. Find site to insert at
   - Do not lock list
   - Traverse list serially
2. Perform List Update
   - Enter Critical Region
   - Re-Confirm site is unchanged
   - Update list pointers
   - End Critical Region

- Only One Insertion at a time
- Unlimited number of search threads
  - More threads means...
    - more failed inter-node locks
    - more repeated (wasted) searches
  - Wallclock Time = N
    - Parallel Search – 1
    - Serial updates – N

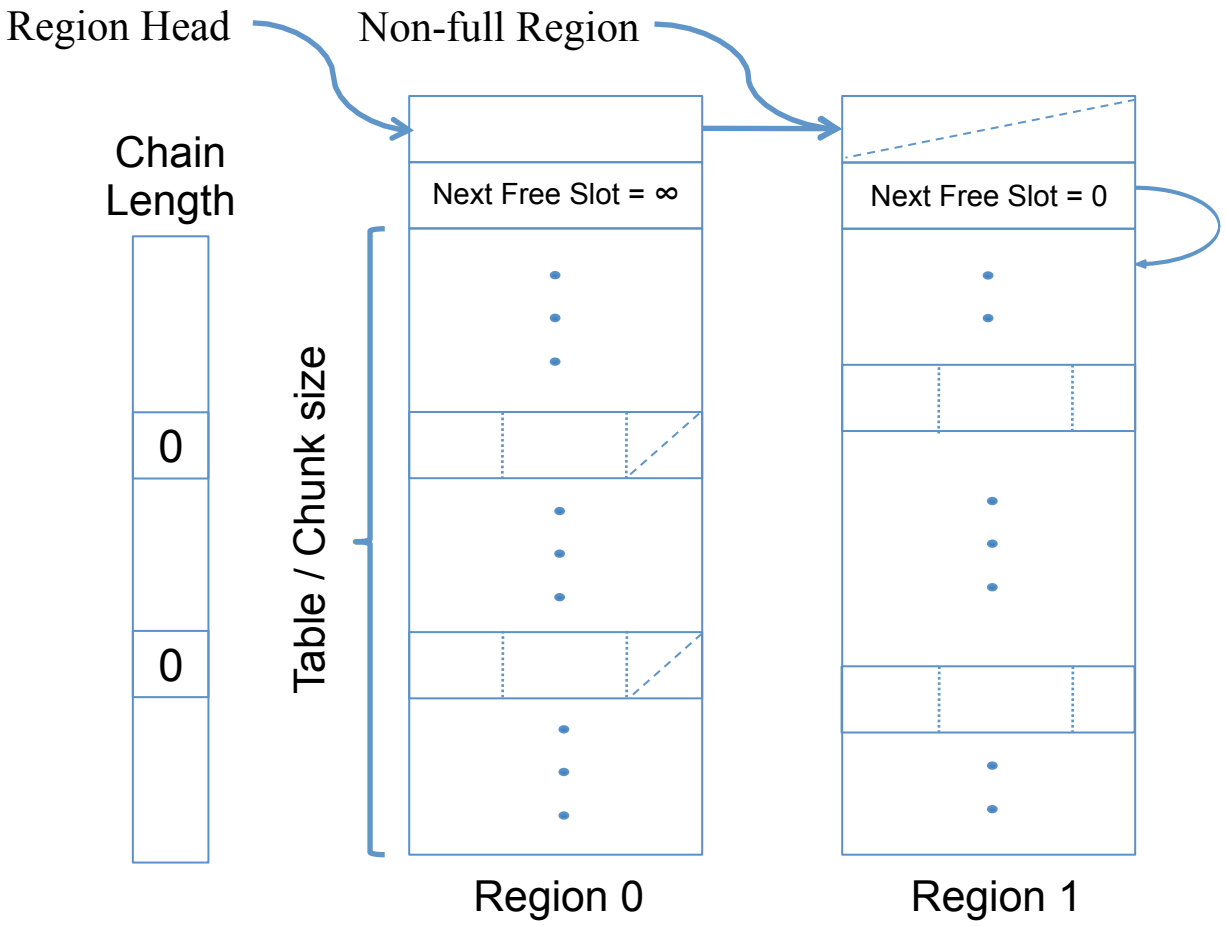
Pacific Northwest
NATIONAL LABORATORY

Two-Phase List Insertion (MTA)

1. Find site to insert at
   - Do not lock list
   - Traverse list serially
2. Perform List Update
   - Lock two elements inserted between
     - Acquire locks in lexicographical order to avoid deadlocks
   - Confirm link between nodes is unchanged
   - Update link pointers of nodes
   - Unlock two elements in reverse order of locking

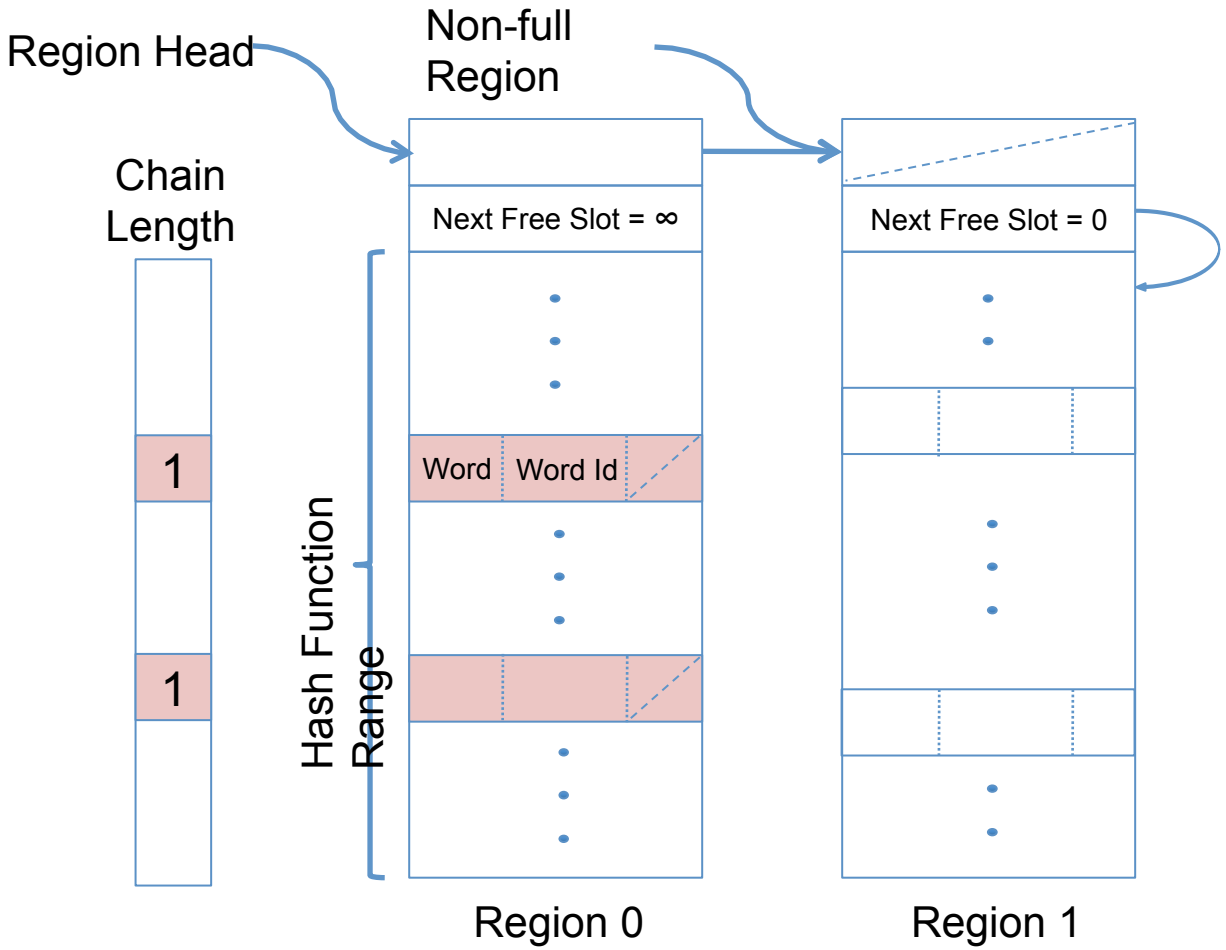
- N/4 Maximum concurrent insertions
  - Insert between every pair of nodes
- More insertion points means fewer failed inter-node locks
- Total Time = logN

Global HACHAR – Initial Data Structure



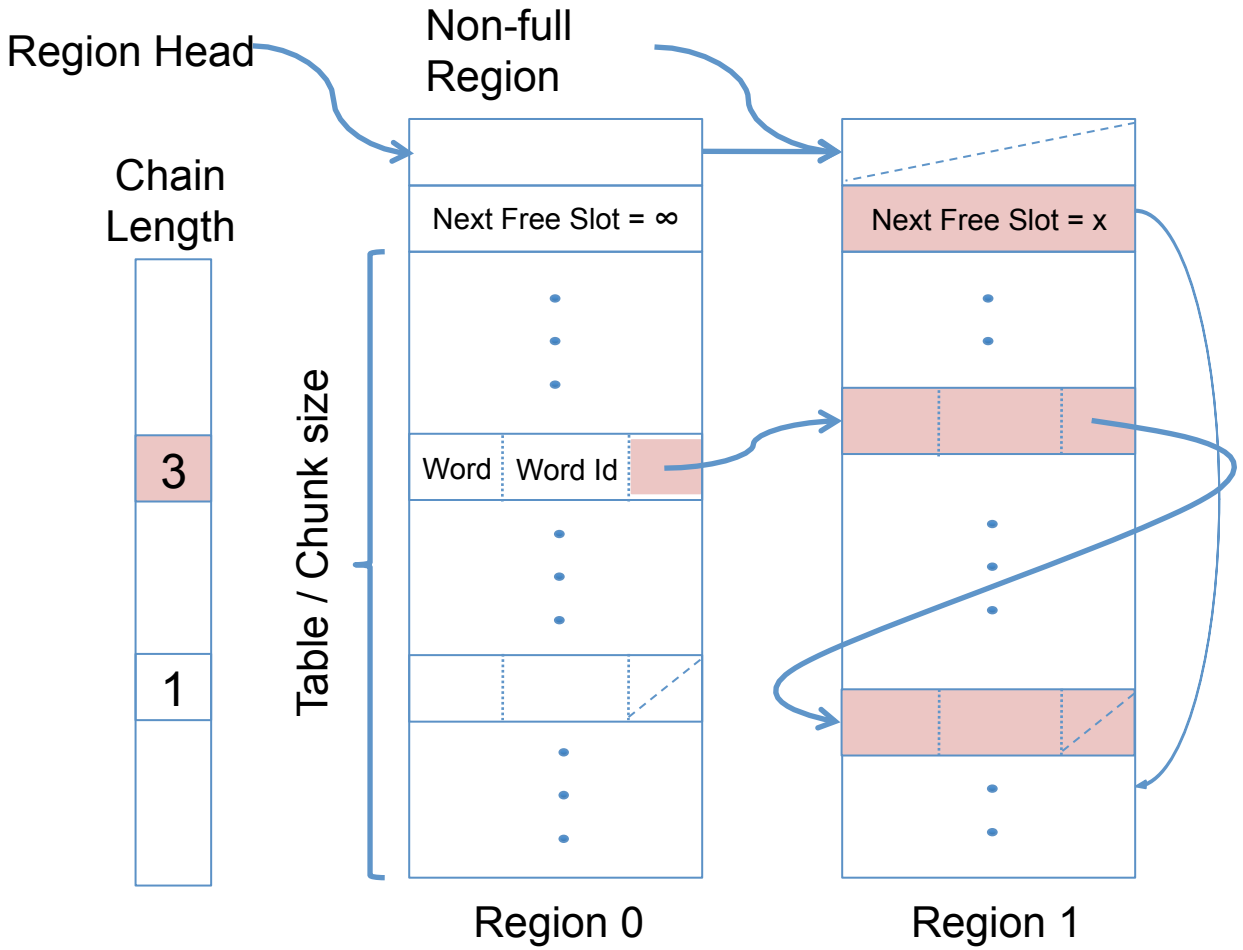
- Use “two-step acquire” on length, region linked list pointers, chain pointers.
- Use int_fetch_add on “next free slot” to allocate list node.

Global HACHAR – Two items inserted



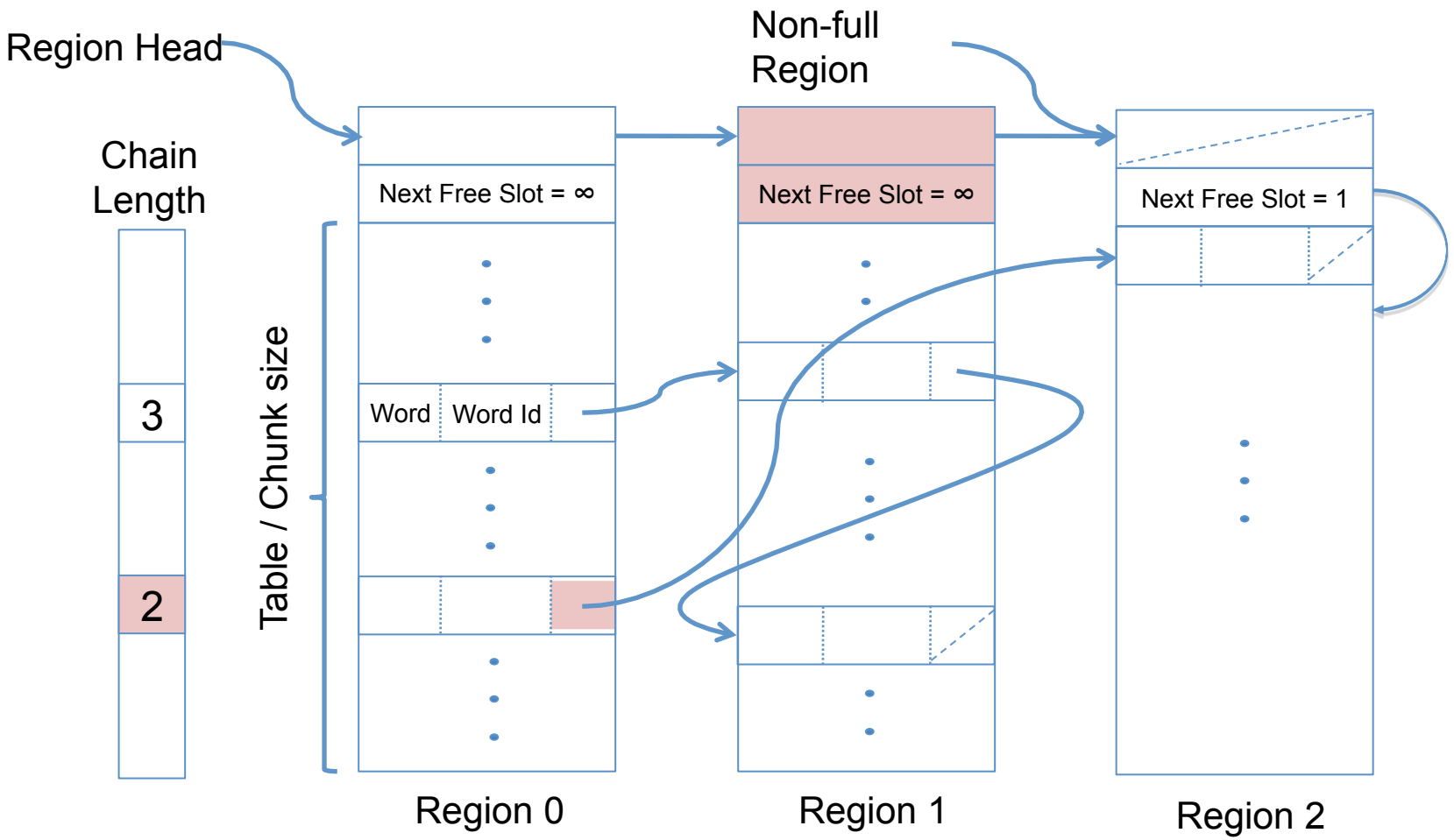
- "locked" length and inserted into "head of list"
- Potential contention only on length
- List node shows example for Bag Of Words

Global HACHAR – Collisions



- Lookup: walk chain, no locking
- Malloc and free limited to the few region buffer
- Growing a chain requires lock of only last pointer (int_fetch_add length)

Global HACHAR – Region Overflow



Once Per Thread vs. Once Per Iteration

Allocating Storage Once Per Thread

```
PARALLEL-DO  i = 0 .. Nthreads-1
  float *p = malloc(...)
  int  n_iters = Nelements / Nthreads
  DO j = i*n_iters .. max(N,(i+1)*n_iters)
    p[j] = ... x[j] ...
    x[j] = ... p[j] ...
  ENDDO
  free(p);
END-PARALLEL-DO
```

Parallel loop, one iteration per thread

`malloc` hoisted out of inner loop, or fused into outer loop

Block of serial iterations

- OpenMP
  - Parallel regions are separate from loops
  - Loop decomposition idiom already captured in conventional pragma syntax
- XMT
  - Abominable Kludge
  - Non-portable pragma semantics
  - Requires separate loops, possibly parallel region

Pacific Northwest
NATIONAL LABORATORY

Synchronization is a Natural Part of Parallelism

- Synchronization cannot be avoided, it must be made efficient and easy to use
  - "Lock Free" algorithms aren't...
    - Sequential execution of atomic ops (compare and swap, fetch and add)
    - Hidden lock semantics in compiler or hardware (change either and you have a bug)
  - Communication-free loop level data parallelism (ie: vectorization) is a limited kind of parallelism
- Synchronization needed for many purposes
  - Atomicity
  - Enforce order dependencies
  - Manage threads
- Must be abundant
  - Don't want to worry about allocating or rationing synchronization variables

Conclusions

- Synchronization is a natural part of parallel programs
  - Synchronization must be abundant, efficient, and easy to use
- Some latencies can be minimized, others must be tolerated
  - Parallelism can mask latency
  - Enough parallelism makes latency irrelevant
- Fine-grained synchronization improves utilization
  - More opportunities for exploiting parallelism
  - Proactively share resources
- Parallelism is performance portable
  - Same programming model from desktop to supercomputer
  - Quality of Service determined by amount of concurrency in hardware, not optimizations in software
- AMOs versus Tag Bits
  - Tags make fine-grained parallelism possible

    More Data == More Concurrency
  - AMOs do not scale