Reusable Work Seeking Parallel Framework for Ada 2005 (*and Beyond)

By Brad Moore
Presentation Outline

• Describe generic classification
  – Iterative vs Recursive
  – Work Sharing vs Work Seeking
  – Reducing vs Non-Reducing
• Describe Work Sharing, Work Stealing, Work Seeking
• Iterative & Recursive Parallelism Examples
• Pragma ideas for further simplification
• Lessons Learned, Affinity, Worker Count, Work Budget
• Briefly discuss how generics could be applied to Battlefield Spectrum Management
• Performance Results
# Parallel Generics Implemented

<table>
<thead>
<tr>
<th></th>
<th>Iterative Parallelism</th>
<th>Recursive Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Work Sharing</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(without load balancing)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-Reducing</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Reducing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Elementary</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Composite</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Work Seeking</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(load balancing)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-Reducing</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Reducing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Elementary</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Composite</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Iterative usage

• Speeding up loops
  - Best applied to ”for” loops, where number of iterations known before starting parallelism

• Example usage
  - Solving matrices, partial differential equations
  - Determining if a number is prime
  - Processing a large number of objects
  - Processing a small number of ”big” objects
Recursive usage

- Processing recursive (tree) data structures
  - Binary trees, Red/Black Trees
  - N-way trees
- Recursive algorithms (e.g. Fibonacci)
  
  \[
  \text{Fibonacci (X)} \\
  = \text{Fibonacci (X – 1) + Fibonacci (X - 2)};
  \]
Workers, Work defined

• In scheduling world,
  – workers are processors,
  – work is threads/processes.

• For these generics in the application domain,
  – workers are tasks
  – work is subprograms
    • or sequential fragments of code that can be wrapped in a subprogram
Work Sharing

- When scheduling new work attempt to give to under-utilized worker.
- Conceptually, a centralized work queue shared between workers.
Work Sharing Optimizations used in Parallelism Generics

- Simple Divide and Conquer
- Define work such that;
  Work Item Count = Worker Count
  - i.e., no load-balancing takes place
  - Well suited if load balancing not needed
- Centralized queue ”optimized” out
- Optimal performance for evenly distributed loads
Work Stealing

- Idle workers try to "steal" work from busy workers.
- Idle worker typically search for work randomly from busy workers.
- Load balancing managed by idle workers.
- Ruled out as an approach for various reasons
  - Work Seeking seen as better choice
Work Sharing Issues

• Pro
  - Optimal for evenly distributed loads, with minimal overhead

• Con
  - Unevenly distributed work can lead to poor processor utilization. (Idle processors waiting for other processors with larger work that could be further broken up)
Work Stealing Issues

• Pro
  – Optimal processor utilization assuming uneven work load distribution.

• Con
  – Compartmentalization structure likely introduces overhead
  – More overhead than work sharing for evenly distributed loads
A Work Stealing Approach (Ruled out)

- Benchmark: Sequential code running on single processor.
- Ideally algorithm should show single worker executes as fast as sequential code.
- An approach with minimal interference on busy workers has idle task suspend busy worker, steal work, then resume worker.
  - Most general purpose OS's don't allow one thread to suspend/resume another.
  - RT OS may allow.
Work Stealing Approaches (Cont)

• Another approach using deques. Idle tasks steal work from the tail of deque, busy workers extract work from the head of deque.
  
  - Approach used by Cilk++

• Compartmentalizing work to insert on deque introduces overhead to process deque.
Load Balancing Approach Taken: Work Seeking

- Compromise between Work Sharing and Work Stealing models.
- Idle tasks request (seek) work.
- Busy tasks check for existence of work seekers, and offer work.
- Low distributed overhead involves simple check of an atomic Boolean variable.
- Direct handoff eliminates need for random searching for work.
Work Seeking (cont)

- No need to randomly search for busy worker
  - Busy worker hands off work directly to idle worker requesting work.
- Minimal contention, can outperform barrier approach using POSIX barrier calls.
- Generic implementation does not use heap allocation. Everything is stack based.
Work Sharing vs Work Seeking

- Choice depends on whether load balancing is needed.

<table>
<thead>
<tr>
<th></th>
<th>Evenly distributed loads</th>
<th>Unevenly distributed loads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Work Sharing</td>
<td>Good</td>
<td>Poor processor utilization, high idle times</td>
</tr>
<tr>
<td>Work Seeking</td>
<td>Load balancing overhead not needed</td>
<td>Good</td>
</tr>
</tbody>
</table>
Work Seeking
Example Problem: Sum of integers

```pascal
Sum : Integer := 0;
for I in 1 .. 1_000_000_000 loop
  Sum := Sum + I;
end loop
```

- Divide and Conquer between available processors.
- Assuming two processors mapped to two tasks,
  - T1 gets 1 .. 500_000_000
  - T2 gets 500_000_001 .. 1_000_000_000
- Issue: Race condition updating Sum
- Each task gets own copy of global Sum
  - Final result involves reducing copies of Sum
Sum of Integers: (cont)

- Generally, we can add parallelism to process globals if reducing operation is associative.
  - e.g. Addition, Appending to list, Min/Max, multiplication?

- Order of operations is preserved.
  - e.g. Appending integers to list results in sorted list from 1 .. 1_000_000_000,
  - same result as sequential code
One can write custom solution in Ada but...

- Too much effort, unless absolutely needed.

(Even worse if generalized for any number of processors).

- More likely to have bugs than simple sequential solution

- Programmers likely wouldn't bother

- Lost Parallelism
Goal

- To facilitate parallelism in loops and recursion.
- Ada's strong nesting shines (Insertion at original loop site).

```ada
Sum : Integer;
declare
    procedure Iteration (Start, Finish : Positive; Sum : in out Integer) is
    begin
        for I in Start .. Finish loop -- Based on original sequential code
            Sum := Sum + I;
        end loop;
    end Iteration;
begin
    Integer_Addition_Reducer -- Work Sharing Generic Instantiation
        (From  => 1,
         To    => 1_000_000_000,
         Process => Iteration'Access,
         Item   => Sum);
end;
```
Work Sharing Generic Instantiation

- Common Reducers may be pre-instantiated and reused/shared

```vhdl
with Parallel.Iterate_And_Reduce;
procedure Integer_Addition_Reducer is new
  Parallel.Iterate_And_Reduce
  (Iteration_Index_Type => Positive,
   Element_Type => Integer,
   Reducer => "+",
   Identity_Value => 0);
```
Ultimate Goal

• Even better if we can provide syntactic sugar
• The pragma would expand to the code as shown previously

Sum : Integer := 0;
for I in 1 .. 1_000_000_000 loop
    Sum := Sum + I;
end loop
pragma Parallel_Loop – Idea for a new pragma
    (Load_Balancing => False, – = Work Sharing, not Work Seeking
    Reducer => "+", – Monoid Reducing function
    Identity => 0, – Monoid Identity Value
    Result => Sum); – Global State
Work Seeking Version

Sum : Integer;
declare
procedure Iteration
(Start : Integer;
 Finish : in out Integer;
 Others_Seeing_Work : not null access Parallel.Work_Seeing;
 Sum : in out Integer) is
begin
for I in Start .. Finish loop  -- Based on original sequential code
  Sum := Sum + I;
  if Others_Seeing_Work.all then  -- Atomic Boolean check
    Others_Seeing_Work.all := False;  -- Stop other workers from checking
    Finish := I;  -- Tell generic how far we got
    exit;  -- Generic will re-invoke us with less work
  end if;
end loop;
end Iteration;
begin
Work_Seeing_Integer_Addition Reducer  -- Pre-instantiated generic
(From => 1,
 To => 1_000_000_000,
 Process => Iteration'Access,
 Item => Sum);
end;
Ultimate Work Seeking Version

- Note almost identical to work sharing version

```
Sum : Integer := 0;
for I in 1 .. 1_000_000_000 loop
    Sum := Sum + I;
end loop

pragma Parallel_Loop  -- Idea for a new pragma
    (Load_Balancing => True,  -- Work Seeking, not Work Sharing
     Reducer => "+",  -- Monoid Reducing function
     Identity => 0,  -- Monoid Identity Value
     Result => Sum);  -- Global State
```
Parallel Recursion

• Idea is to allow workers to recurse independently of each other.
  - While one worker is recursing upwards, others may still be recursing down the tree.

• Unlike loop iteration, total iteration count not typically known.

• Number of ”splits” at given node likely is known however.
Possible Recursion Syntax Example

- Similarly for parallel recursion...

```plaintext
procedure Iterate (Container : Tree;
     Process   : not null access procedure (Position : Cursor))
is
     procedure Span_Tree (Node : Node_Access) is
     begin
         if Node = null then
             return;
         end if;

         Span_Tree (Node.Left);
         Process (Cursor'(Container'Unrestricted_Access, Node));
         Span_Tree (Node.Right);
     end Span_Tree;
pragma Parallel_Procedure (Load_Balancing => True, Splits => 2);
begin  -- Iterate
     Span_Tree (Container.Root);
end Iterate;
```
Lessons Learned: Affinity

- Affinity: locking tasks to specific processors
- Thought extra control would improve performance
- Seldom provided benefit, and only if:
  - iterations mod processors = 0
  or
  - processor count insignificant compared to iteration count
- Otherwise, better left to scheduler to decide
  - Could consider sophisticated dynamic algorithm
Affinity

• Assume 2 processors, 3 iterations

• With workers = 2. W1 <= I1  W2 <= I2-I3
  - W1 Finishes I1 when W2 starts I3
    • Total time = 2 * Iteration time
    • Idle time = 1 * iterator time

• With workers = 3. p1 <= W1, p2 <= W2-W3
  - P1 finishes W1 when p2 is half-way through W2 & W3
    • Total time = (1 + (0.5 + 0.5)) * Iteration time
    • Idle time = 1 * iterator time
Without Affinity

- 3 workers, 3 iterations
- OS scheduler migrates workers between processors as needed to provide fair sharing of processors
- All 3 workers complete at the same time.
  - Total time = 3 * iteration time / processor count
  - Idle time = 0
  - 1.5t beats 2t
Lessons Learned: Choosing Worker Count

- If iterations count significant relative to processor count...
  - If iteration count \( \geq \) processor count
    Select worker count that is the smallest factor of the iteration count that is greater or equal to the number of processors
  - else
    Use Iteration count
- else use processor count
Iterative Worker Count Example

- e.g. for 4 processor target

<table>
<thead>
<tr>
<th>Iteration Count</th>
<th>Recommended Worker Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
</tr>
</tbody>
</table>
Work Budget

- Number of times a worker task may seek work
  - 1 approximates work sharing
  - -1 (unlimited)
- Thought diminishing returns would mean need to tune value for optimum performance
- Generally found that unlimited work budget provides optimum results for work seeking.
Subcontractor count

• For recursion, since iteration count is unknown
• = Number of sub workers (subcontractors) a worker is allowed to ”hire”
• Used for initial loading of workers.
• Attempts to evenly distribute workers among available processors. Better to assign as soon as possible in the recursion
Possibility for industrial usage

Battlefield Spectrum Management

- Algorithm to assign radio frequencies to emitters.
- Used by signal planners in military to plan communications deployment
- Limited Frequencies
- Interference
- Numerical analysis can take time
- Looping through emitters suggest these generics could improve performance.
To Do

• Port to RTOS
  – MaRTE specifically
  – Add work stealing generics with suspend/resume semantics
  – Compare work stealing against work seeking, work sharing.

• Follow up on interest for syntactic sugar
  – AI for post Ada 2012?
Performance Results

- Single worker performs comparably to sequential code
- Ada generics significantly outperform similar examples written in Cilk++
- Ada generics significantly outperform non-generic Ada code using POSIX barriers to manage splits and joins for matrix solving, partial diff. equations.
Conclusions

- Parallel Generics encourage increased use of parallelism in applications.
- Further simplification possible
  - Syntactic sugar pragmas
  - Extra compiler checks to validate parallel usage
- Default affinity may be good enough here
- Programmer needs to indicate preference for load balancing. Compiler likely can't make decision.
Questions?