Motivation

- Current methods to understand memory behaviour is to use a execution driven simulation which slow the program by a factor of 10 to 1000
- Insufficient parallelism
- Load imbalance
- Poor memory hierarchy performance

- Taxonomy of sources of performance loss
  - Synchronization overhead
  - Memory Hierarchy losses
  - Extra work in parallel program vs sequential
Basic Idea

- Introduce as little perturbation as possible
- A basic block is a set of contiguous instructions with a unique entry and exit point
- In the first pass the MTOOL instruments blocks with counters
- Using known instruction latencies and counts from the initial run MTOOL builds an execution profile
- MTOOL is instrumented to collect actual time spent in selected loops, procedures, synchronization calls and basic block counts

Continued…

- Compare actual user time profile with a ideal compute time profile
- Memory losses can be isolated by comparing the actual time measurements to estimates made using the basic block counts under the assumption of an ideal memory system.
- Synchronization overheads are identified by the time taken by synchronization calls
- Extra work is identified by either the instruction count of user-tagged instructions or where the MTOOL can identify extra work because they occur in parallel control constructs
Minimizing Timer Perturbation

- Using start/stop timer pairs using local memory can distort synchronization overhead
- Overhead of timings depends strongly on the hardware and software support available
- In another paper the authors describe an implementation of MTOOL using a memory mapped software clock
- Inserting a start/stop timer pair can take ten to fifteen instructions and therefore is a potential source of perturbation if not carefully inserted

Continued…

- MTOOL uses its initial execution profile to estimate the average time per call to a region. If this time is significantly greater than 5 times the start/stop timer overhead then timers are inserted.
- Explicit timer overhead can be removed by pc-sampling, a technique also used in PROF/GPROF
Memory Bottleneck

- A programmer’s view of memory is uniform access whereas in reality there can exit per processor multi-level caches, buffers, interconnection networks and banked memory
- Some memory accesses cost more than others
- MTOOL uses basic block counts to accurately estimate the execution time of a loop or procedure assuming the memory hierarchy performs optimally. The actual time spent in the loop or procedure is also recorded. The two times are then contrasted to identify memory bottlenecks.

Continued…

- The time for a block to execute can be accurately predicted except for
  - Data dependent stalls
  - Stalls between instructions in different basic blocks
- Data dependent stalls occur primarily because of memory hierarchy on the MIPS and CRAY structures. MTOOL can estimate execution time given perfect memory behaviour on these architectures
Procedure Selection

- Contain majority of global memory operations
- Can be timed without introducing perturbation
- Is comprehensible to the user

Minimum Cost Basic Block Counting

- If we consider the flow of control of a procedure, it is only necessary to place counters on independent control paths. Dependent counts can be computed in a post-processing phase.
- Loops include a built-in counter, the loop index, whose value can be recorded on loop exit thus avoiding the overhead of counting the individual loop iterations.
Example

SUBROUTINE FOO(A, B, N)
REAL A(N), B(N)
DO 10 I = 1, N
   IF (A(I) .NE. 0.0) THEN
      B(I) = 1.0 / A(I)
   ELSE
      B(I) = 0.0
   ENDIF
10 CONTINUE
START

(A(I) .NE. 0)

I = I + 1

(I .GT. N)

Return

B(I) = 1/A(I)

B(I) = 0.0
START
(I .GT. N)

(A(I) .NE. 0)

B(I) = 1/A(I)

I = I + 1

(I .GT. N)

Return

B(I) = 0.0

0

START
(I .GT. N)

(A(I) .NE. 0)

B(I) = 1/A(I)

I = I + 1

(I .GT. N)

Return

B(I) = 0.0

0
START

(A(I) .NE. 0)

B(I) = 1/A(I)

I = I + 1

(I .GT. N)

Return

B(I) = 0.0
\[ A(I) \neq 0 \]
\[ I = I + 1 \]
\[ I > N \]
\[ B(I) = 0.0 \]
\[ B(I) = 1/A(I) \]
\[ I = I + 1 \]
\[ I > N \]
\[ F \]
\[ \text{Return} \]
Continued…

- The maximum weight spanning tree is with edges:
  - (b,c) (c,e) (d,e) (e,f) (f,a)
- Remaining edges include:
  - (a,b) (b,d) (e,b) (a,f)
- The total basic block count cost comes out to be 4
- In most cases the overhead lies between 1-5%

Synchronization Bottlenecks

- Synchronization overhead is defined as any time when a processor is idle or waiting.
- These overheads include Load Imbalance, Critical Sections and Starvation.
- MTOOL can detect synchronization overheads by instrumenting locks and barriers to measure waiting times.
- User defined spin waiting can be ignored by MTOOL if the user explicitly mentions in a file.
Extra Work

- MTOOL uses its basic block counting technique to estimate time spent in parallel control operations like spawning tasks and allocating locks
- The user can define extra work in a file

Case Study # 1
Case Study # 2

Figure 3: TRI Procedure Histogram, Source Window, and Info Box

Continued...

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measured Speedup</td>
<td>1.0</td>
<td>1.9</td>
<td>3.3</td>
<td>3.5</td>
<td>3.5</td>
</tr>
</tbody>
</table>

Table 1: Speedup of ForwardSolvePar_Self
Critical Path Profiling of Message Passing and Shared-Memory Programs

Jeffrey K. Hollingsworth
Motivation

- The basic idea is to benchmark or fine-tune parallel programs
- Various metrics & tools already exist such as CPU utilization
- A post-mortem version of critical path profiling but requires space proportional to the number of procedures calls and becomes impractical for long running programs

Basic Idea

- Critical path is the longest time weighted sequence of events from the start of a program to its termination.
- Provides global view of the performance but that is exactly what makes it difficult to compute
Basic Ideas

- **Process** is a thread of execution with its own address space.
- **Event** is any observable operation performed by a process.
- **Program** consists of one or more processes that communicate during execution.
- **Program Trace** is a set of logs, one per process, that records the events that happened during the execution of that process. \( PT[p,i] \) defines the \( i \)th event in process \( p \).
- **CPU Time** doesn’t include the time spent waiting for a message.
- **Program Activity Graph** is a graph of events in a single program trace. Nodes represent events or communication dependencies.

Program Activity Graph (PAG)

![Program Activity Graph](image)
initial node

call(A) 0[0] call(C) 0[0] call(C)
Old longest path value of 8 changed to the new received longest path value of 10.
Old longest path value of 11 changed to the new received longest path value of 15
Selecting Procedures

- Critical Path Profile is calculated for a selected number of procedures.
- These procedures can be selected manually by the programmer.
- A good heuristic to select the procedures for Critical Path Profile is to use CPU time.
- This process can also be automated and the authors provide an algorithm to do so in O(mlog₂n).
Critical Path For Partial Program Execution

- Amounts to dividing PAG into three disjoint pieces
- The selection region is assumed to be terminated by a barrier
- As the PAG isn’t implicitly built, the desired region of the PAG is identified by sampling
- Sampling brings in the issue of having a consistent snapshot of the critical path
Continued…

- Happen Before: an event x happens before event y if event x occurs before event y in the program trace. A send always occurs before a receive.
- A state slice is the last event in a process that is required to happen before an event e.
  \[
  \text{slice}[p,e] = \text{PT}[p,i] : \text{PT}[p,i] \rightarrow e \quad \text{and} \\
  \quad \text{for all } j>i \not\exists (\text{PT}[p,j] \rightarrow e)
  \]
- Timeliness of samples

Continued …

- To ensure timeliness we combine the state slices of the latest sample from each process.
- \( G[p,i] \) is the latest event for process p from all of the state slices for samples 0 to i.
  \[
  G[p,i] = \max( G[p,i-1], \text{slice}[p,i])
  \]
  \[
  G[p,0] = \text{PT}[p,1]
  \]
- Messaging issues
Continued…

- Not all processes will receive the message to start critical path calculation at the same time
- A message without CP data arrives at a process that is not computing the critical path.
- A message with CP data arrives at a process that is already computing the critical path.
- A message with CP data arrives at a process that is not computing the critical path.
- A message without CP data arrives at a process that is already computing the critical path.
- Critical Path must be calculated from a consistent state during the execution of the program
- Chandy and Lamport algorithm

Online Critical Path Zeroing

- Serves as a better guide than CP
- Logical zeroing computes the reduction in the length of the critical path due to tuning specific procedures
- Instrumentation data piggybacked onto application messages
- At the merge nodes the “net” path length is compared
- When CP is sampled only “net” path length is reported
### Overhead Calculations

<table>
<thead>
<tr>
<th>Number of CP Items</th>
<th>~75% Computation</th>
<th></th>
<th></th>
<th></th>
<th>~60% Computation</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Wall Time</td>
<td>Overhead</td>
<td>Wall Time</td>
<td>Overhead</td>
<td>Wall Time</td>
<td>Overhead</td>
<td>Wall Time</td>
<td>Overhead</td>
</tr>
<tr>
<td>Base</td>
<td>154.1</td>
<td>1.9%</td>
<td>91.7</td>
<td>3.4%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>157.0</td>
<td>2.1%</td>
<td>94.9</td>
<td>4.1%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>157.4</td>
<td>2.2%</td>
<td>95.5</td>
<td>4.4%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>157.5</td>
<td>2.8%</td>
<td>95.8</td>
<td>4.4%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>158.4</td>
<td>2.9%</td>
<td>95.9</td>
<td>4.6%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>158.5</td>
<td>3.6%</td>
<td>97.0</td>
<td>5.8%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>159.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of CP Items</th>
<th>5 msgs/sec Time</th>
<th>Percent</th>
<th>50 msgs/sec Time</th>
<th>Percent</th>
<th>150 msgs/sec Time</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>50.3</td>
<td>0.3</td>
<td>51.8</td>
<td>0.0</td>
<td>194.5</td>
<td>0.3</td>
</tr>
<tr>
<td>0</td>
<td>50.4</td>
<td>1.2</td>
<td>52.7</td>
<td>1.7</td>
<td>197.6</td>
<td>1.9</td>
</tr>
<tr>
<td>1</td>
<td>50.8</td>
<td>1.1</td>
<td>53.3</td>
<td>2.7</td>
<td>206.0</td>
<td>6.2</td>
</tr>
<tr>
<td>4</td>
<td>51.4</td>
<td>2.2</td>
<td>55.4</td>
<td>6.9</td>
<td>215.4</td>
<td>11.1</td>
</tr>
<tr>
<td>8</td>
<td>55.1</td>
<td>9.5</td>
<td>64.0</td>
<td>23.5</td>
<td>287.8</td>
<td>48.4</td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Implementation Results

<table>
<thead>
<tr>
<th>Procedure</th>
<th>CP</th>
<th>% CP</th>
<th>CPU</th>
<th>% CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>nas_is_ben</td>
<td>8.8</td>
<td>31.7</td>
<td>103.1</td>
<td>67.8</td>
</tr>
<tr>
<td>do_rank</td>
<td>0.5</td>
<td>1.8</td>
<td>29.9</td>
<td>19.7</td>
</tr>
<tr>
<td>create_seq</td>
<td>18.5</td>
<td>66.5</td>
<td>18.7</td>
<td>12.3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Procedure</th>
<th>CP</th>
<th>% CP</th>
<th>CPU</th>
<th>% CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>BeManager</td>
<td>31</td>
<td>48.0</td>
<td>47</td>
<td>19.3</td>
</tr>
<tr>
<td>MinCircuit</td>
<td>29</td>
<td>44.9</td>
<td>196</td>
<td>80.7</td>
</tr>
</tbody>
</table>
Shared Memory Programs

- The synchronization abstractions between threads of execution are 
  locks and barriers.
- Data structure for each synchronization object is maintained in the 
  memory and updated by the instrumentation code in each thread of 
  control whereas critical path data is exchanged via messages in a 
  message passing system.

Continued…

---

[Diagram of shared memory program with nodes labeled as 'spin Lock', 'call(b)', 'call(c)', 'unlock', 'lock', 'arBar', 'lvBar', and numerical values on the edges representing synchronization operations.]
Non-determinism in parallel programs can create problems for critical path zeroing to work properly.

Critical path zeroing uses PAG for a single program execution and any changes due to tuning can alter the graph structure.

To correctly predict the performance of these cases, the PAG tracks the item leaving and entering the task queue.
Questions ???