Gprof Profiler and Critical Path Analysis: Two Performance's Measurement Tools

Nelson Padua-Perez
nelson@cs.umd.edu

Department of Computer Science,
University of Maryland, College Park

Introduction

- gprof: provides a graphical execution profile.
- Takes advantage of the fact programs are large, structured and hierarchical.
- gprof goal: Helps user evaluate alternative abstractions implementations.
- Easy to Use: add -pg compilation option, execute, gprof "executable"

Call Graphs

- gprof uses call graphs to organize timing data.
- Call Graphs Types
  1. **Static Call Graph** - determined from static program text.
  2. **Dynamic Call Graph** - includes routines and arcs by profiled execution.
  3. **Complete Call Graph** - static + dynamic graph.

Gathering Profile Data

- Which data we want to gather
  1. call counts.
  2. execution time for each profiled routine.
  3. arcs of the dynamic call graph traversed by execution.
- How to gather the data
  1. By having the compiler augment the code.
  2. By calls to a monitoring routine (gprof alternative).

Execution Time

- Can be determined by
  1. Wall clock time - (entry to exit time)
  2. Program counter sampling - exec. time inferred from samples distribution (gprof approach).
  3. Process time - support provided by O.S.
- gprof assumption - all calls to a specific routine take same execution time.

Data Post Processing

- Dynamic call graph built from caller-callee data.
- Time propagation among functions done by using topological sort.
- Time propagation formula:

$$T_r = S_r + \sum_{\text{r calls e}} \left(T_e \times \frac{C_e^r}{C_e} \right) \tag{1}$$

T_r – Total time of routine r

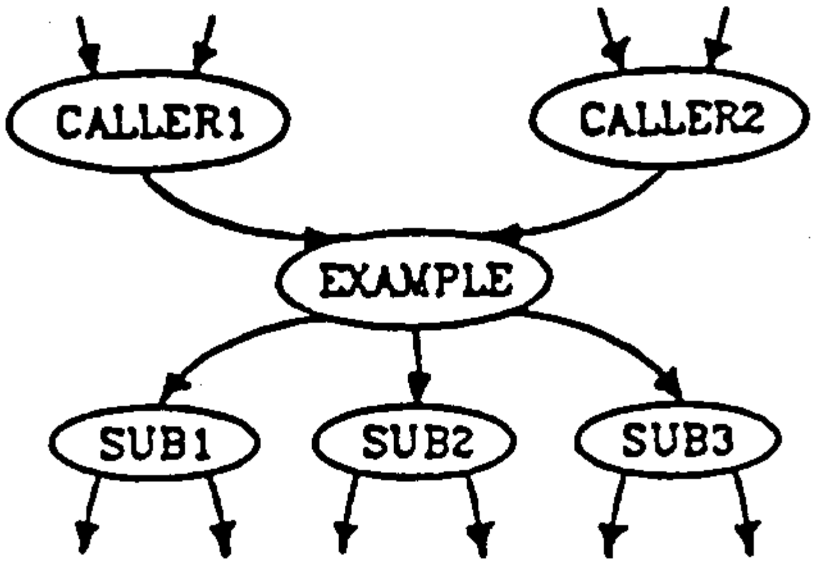
S_r – Self time of routine r

T_e – Total time of routine e

C_e^r – Number of calls from r to e

C_e – Number of calls to routine e

- Recursive calls represent cycles in dynamic graph.
- Detect strongly-connected components (use static graph), then sort.
- Recursive functions: gprof does not provide strong support for them.



index	%time	self	descendants	called/total called+self called/total	parents name children	index
		0.20	1.20	4/10	CALLER1	[7]
		0.30	1.80	6/10	CALLER2	[1]
[2]	41.5	0.50	3.00	10+4	EXAMPLE	[2]
		1.50	1.00	20/40	SUB1 <cycle1>	[4]
		0.00	0.50	1/5	SUB2	[9]
		0.00	0.00	0/5	SUB3	[11]

Profile entry for EXAMPLE.

Conclusions

- Provides simple tool to analyze the performance of sequential program.
- Adds relatively low execution overhead (5 - 30 %).
- Programs with high degree of recursion are not well analyzed.

Introduction

- Simple Performance Techniques not too useful in Parallel/Distributed environments.

- Critical Path Analysis (CPA) as an alternative.

- We will look to the Design, Implementation and Testing of CPA.

Critical Path Analysis

- Turnaround time: important performance measure.

- Turnaround time: determined by event path of longest duration (Critical Path).

- CPA goal: Determine CP and bottleneck(s) along it in order to understand the performance problem.

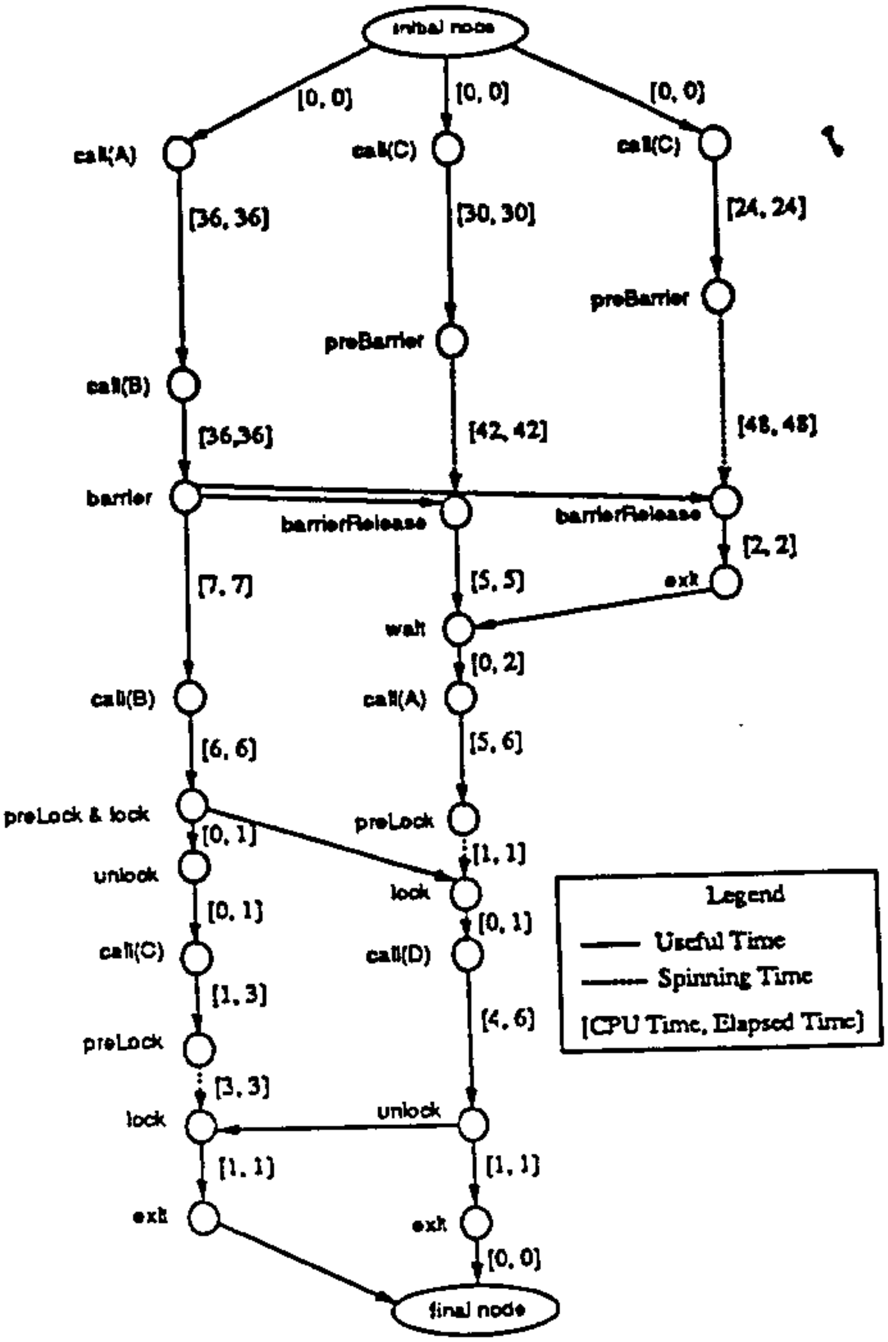
Program Activity Graphs

- Program Activities represented by Program Activity Graphs.

- PAG: weighted, directed multigraph where an edge represents an activity and its weight the activity duration.

- CP = longest weighted path in PAG.

- CPA Approach: Construct PAG from execution data, determine CP and analyze it.



A sample Program Activity Graph.

Nodes represent interesting events in the application (e.g. procedure calls and message passing). The ordered pair indicates the CPU and elapsed times between events.

Algorithms for Critical Path Analysis

- The Problem: How to determine CP from the PAG?

- Acyclic characteristic of PAG's allows the use of shortest-path algorithms.

- Algorithms Used:
  1. Centralized: based on PDM shortest-path algorithm.
  2. Distributed: based on Chandy and Misra's distributed shortest-path algorithm (using a process per sub-graph).

Testing Environment

- Generation of Testing PAG's
  1. **Application 1** - Processes in Master-Slave structure. Non evenly distributed vertices.
  2. **Application 2** - Processes in Pipeline structure. Evenly distributed vertices.
- Number of vertices varied from 1000 - 10000+.
- Number of machines went up to 9.
- Testing hardware/software
  1. All experiments run in VAX-11/750.
  2. Centralized Algorithm used 4.3BSD UNIX, Distributed Charlotte O.S.
- Use of log for information gathering (data recorded are sends, recvs, calls, rets).

Algorithms Performance Metrics

$$\text{Speed-up (S)} = \frac{\text{Execution time Centralized Alg.}}{\text{Execution time Distributed Alg.}}$$

$$\text{Efficiency (E)} = \frac{\text{Speed-up}}{\text{Number of Machines Used}}$$

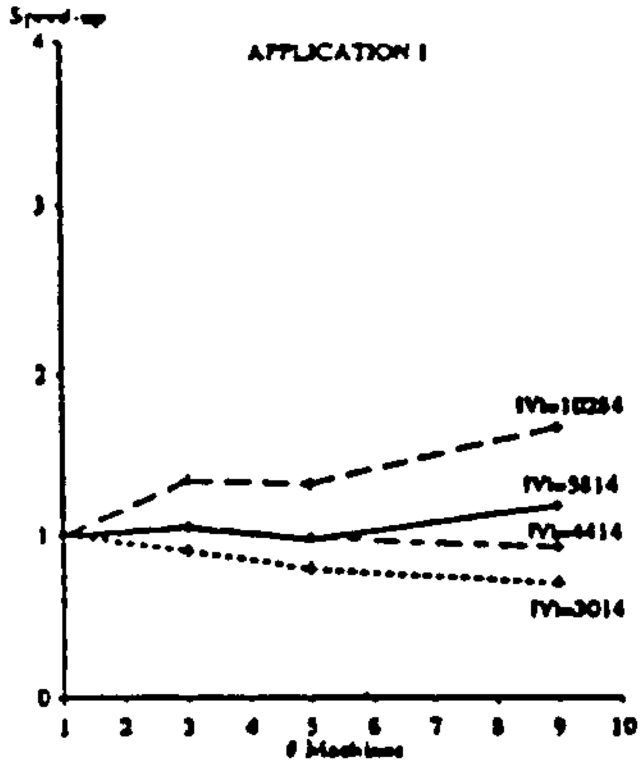


Fig. 3: Speed-up of Distributed Algorithm

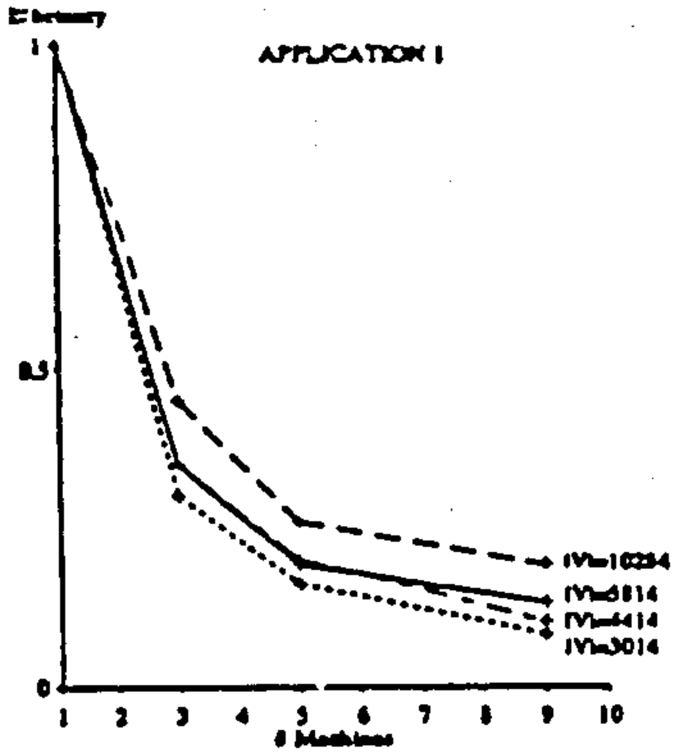


Fig. 4: Efficiency of Distributed Algorithm

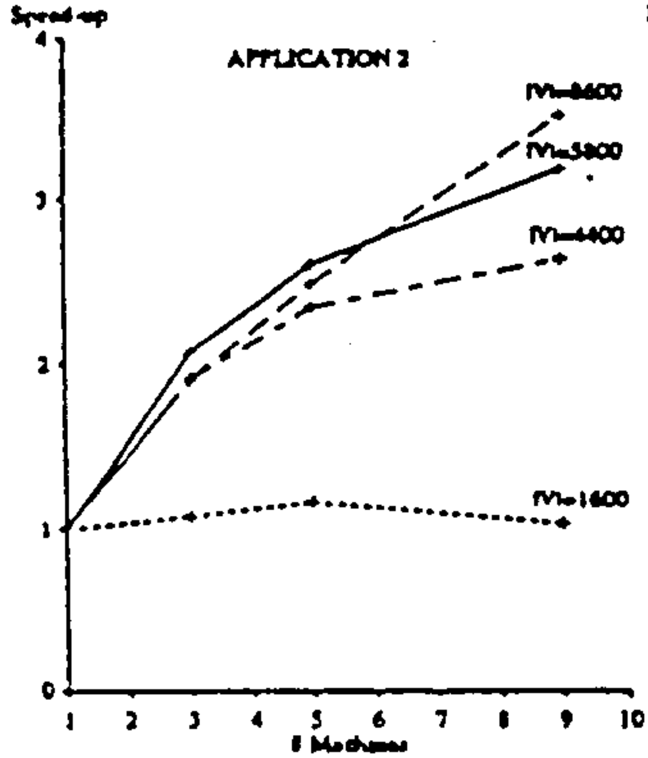


Fig. 5: Speed-up of Distributed Algorithm

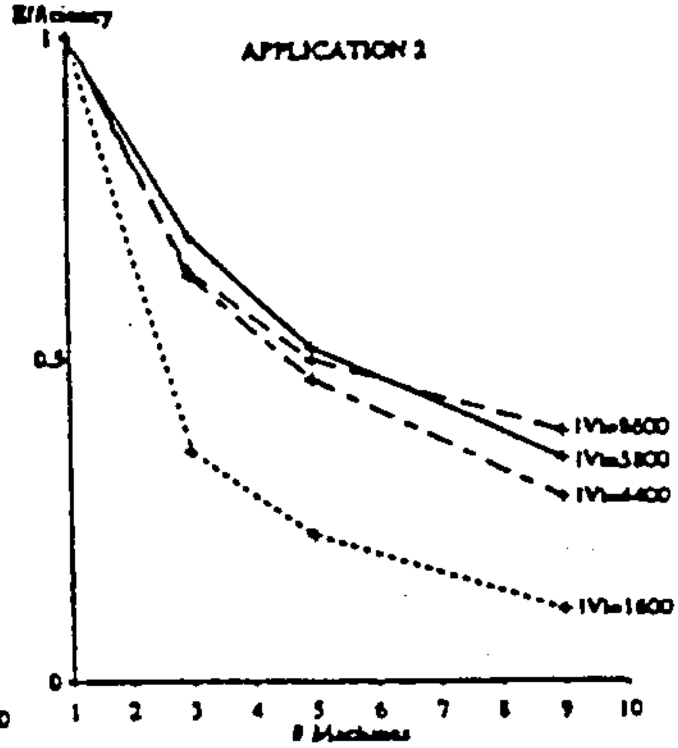


Fig. 6: Efficiency of Distributed Algorithm

Testing of the CPA Technique

- Tests Goals
  1. Show CPA Technique can be integrated in performance measurement tools.
  2. CPA technique can help to understand the performance behavior of a program.

- Test Environment
  1. Performance Measurement Tool: IPS
  2. Program Chosen: Simplex Method for linear Programming.
  3. Configuration: 1 controller process and 8 calculator processes.

Entry Name	Time(ms)	%
CPU	10347	62
Inter-machine Msg	4960	30
Intra-machine Msg	1360	8
Total	16667	100

Critical Path Information at Program Level

Entry Name	Time(ms)	%
P(1,3) CPU	9740	58
P(1,3)->P(3,5) Msg	840	5
P(3,5)->P(1,3) Msg	840	5
P(1,3)->P(2,5) Msg	480	3
P(2,5)->P(1,3) Msg	480	3
P(3,4)->P(1,3) Msg	480	3
P(1,3)->P(3,4) Msg	480	3
P(1,3)->P(2,4) Msg	440	3
P(2,4)->P(1,3) Msg	440	3
P(1,3)->P(1,5) Msg	408	2
P(1,5)->P(1,3) Msg	408	2
P(1,3)->P(1,4) Msg	272	2
P(1,4)->P(1,3) Msg	272	2
P(1,3)->P(3,3) Msg	240	1
P(3,3)->P(1,3) Msg	240	1
P(3,5) CPU	159	1
P(1,5) CPU	108	1
P(2,5) CPU	88	1
P(3,4) CPU	79	*
P(2,4) CPU	67	*
P(1,4) CPU	64	*
P(3,3) CPU	42	*
Total	16667	100

(P(i,j) denotes process j in machine i, * denotes less than 1%)

Critical Path Information at Process Level

Procedure Name	Mach. Process ID	Time(%)
MainLoop	(Mach 1, Proc 3)	21
SendChild	(Mach 1, Proc 3)	17
Init	(Mach 1, Proc 3)	11
read1	(Mach1, Proc 3)	4
CheckWaiting	(Mach 1, Proc 3)	4
MainLoop	(Mach 3, Proc 4)	1
recv	(Mach 1, Proc 3)	1
MainLoop	(Mach 3, Proc 5)	1
MainLoop	(Mach 1, Proc 5)	1
MainLoop	(Mach 2, Proc 4)	1
MainLoop	(Mach 1, Proc 4)	*
MainLoop	(Mach 2, Proc 5)	*
Total CPU		62

(* denotes less than 1%)

Critical Path Information at Procedure Level

Conclusions

- Critical Path Analysis provides guidance to locate performance problems.
- Determining the Critical Path can be done with simple and efficient algorithms.
- Performance tools like gprof work fine for sequential environments but not well in parallel/distributed environments, where CPA represents an excellent tool.