Palm: Easing the Burden of Analytical Performance Modeling

NATHAN TALLENT, KEVIN BARKER, DARREN KERBYSON, ADOLFY HOISIE

Pacific Northwest National Lab

ISC ’15: Performance Modeling: Methods & Applications

July 16, 2015
Analytical Modeling of Performance is Hard

Analytical model of performance
- Quantitatively explains and predicts application execution time
- Diagnose performance-limiting resources, design machines, etc.

How is application modeling difficult?
- Modeling requires expertise and labor
  - model critical path: identify parameters for each critical path segment
  - parameter reduction: represent ‘invariant’ code as measurement
  - validate: iterate until model captures all interesting behavior
- Representing, reproducing and distributing models is ad hoc
  - 1 modeler, N application variants
  - 1 application, N modelers

What can a tool automate? Can we pair model and source code?
Palm: How Can Tools Help?

- Identify and formalize best practices
- Make the simple easy and the difficult possible
  - Provide a fully general framework (do not hinder)
  - Automate routine tasks
- Facilitate a divide-and-conquer modeling strategy
  - Construct model by composing sub-models
  - Define model structure from static & dynamic code structure
- Assist reproducibility
  - Generate same model given same input
  - Generate model according to well-defined rules
- Assist validation (feedback loop)
  - Generate contribution and error reports

Palm: Performance & Architecture Lab Modeling Tool
Outline

▶ Overview

▶ Scientific Workflows and Resource Contention

▶ Silicon Photonics’ Potential For Graph Applications
Annotations guide modeling and express insight
- Develop model and application in tandem
- Decompose modeling task into sub-problems
- Reasonable because applications change slowly

Generate model from (static/dynamic) annotation structure
- Combine annotation expressions and measurements

Models are ‘first class’ objects

Generate same model given same input (reproducible)

Generated model is an executable program
- Instantiate with parameters to generate prediction

refine as necessary

prediction & diagnostics

parameters

profiles

model (program)
Simple Annotations for Nekbone (CG solver)

```plaintext
program nekbone
  !$pal model init
  call init_dim, call init_mesh, ...
  !$pal model cg
  call cg(...)
end

subroutine cg(...)
  !$pal loop n\_cg = \{n\_iter\}
  do iter=1,n\_iter
     ...
  enddo

void halo_exchange(buf[n], n...)
  #pragma pal loop n\_send = \{n\}[max]
  for(i = 0; i < n; ++i)
    isend(..., buf[i]...);

model: classify code block and model one instance of its execution; if expression is omitted, automatically synthesize one

loop: model several instances of a code block; name block and model its trip count

def: define model variable or function

$\{x\}$: program value reference: capture x’s value during program execution and compute statistic across instances & ranks

#pal def snd(sz) = ...

void isend(...size_t n, uint dst...)
  #pal model send = snd(\{n\})
  MPI_Isend(... n, dst...)
```
Palm’s Model Matches Human-Generated Model

A model is a program. Here, it is a Ruby script.

A synthesized model function
(from model & loop annotations and measurements)

cg() model’s form matches a human-generated model:
\[ T_f + 3 \ T_{reduce} + 26 \ T_{send} \]

model function
(from def annotation)

machine parameters
(from model library)

evaluate to obtain runtime

---

class Model
  def nekbone() (init() + cg() + k2) end
  def init() k1 end
  def cg()
    ncg * (f() + reduce1() + ... + reduce3() + 26 * send())
  end
  def snd(sz) @machine.send(sz) end
end

require 'machine-pic.rb'
m = Model.new(PAL::ExecutionPIC.new(...))
m.eval(parameter-list)
Palm: Using Models

- Models are (Ruby) programs
  - scripting language is convenient; could use machine code
  - invoke by passing appropriate parameters (e.g., # cores)
  - replace sub-models by re-defining functions
- Refine annotations using model diagnostics
  - show contribution of each sub-model (expression)
    - quantitatively distinguish 1st- and 2nd-order effects
  - show errors of each sub-model w.r.t. measurements
    - understand effects of replacing a sub-model (function)
    - example: new communication model

Refine as necessary
Sweep3D: 2D pipeline

- Wavefronts propagate in phases, yielding active and idle states
- Idle (pipeline) time depends on ranks, phase, & pipeline stage

Need more than static analysis

- Pipeline formed dynamically
  - State variables and guarded code

Palm assists modeling the critical path – before it exists

- Express idle time as function of a pipeline stage’s model
  - Model critical path using a forward reference to a generated model
- Palm assembles model using dynamic analysis & composition rules

\[ M(\text{rank, phase, } M(\text{stage})) \rightarrow M(\text{rank, phase}) \]
Outline

- Overview
- Scientific Workflows and Resource Contention
- Silicon Photonics’ Potential For Graph Applications
High Energy Physics: Belle II

International effort to advance particle physics

Credit: Malachi Schram
Data! 25 PB/year of raw data
- Stored data expected to reach 350 PB

Belle II Workflow: Extensive data analysis
- Normalize data and ‘do physics’

Many analysis pipelines run concurrently
- Goal: Predict (& mitigate) resource contention

Example analysis pipeline:
- Dynamically assembled modules (Python script)

Palm creates workflow model by composing models for each module
Outline

- Overview
- Scientific Workflows and Resource Contention
- Silicon Photonics’ Potential For Graph Applications
Assessing the Impact of Silicon Photonics

Question: What is the impact of silicon photonics on graph-based workloads in the 4–6 year timeframe?

Methodology

- Work with architects; Identify silicon-photonics enabled systems
  - IBM TOPS (64 nodes, fully connected): photonics off node
  - Oracle Macronode (32 nodes, fully connected): photonics on & off node
- Draw workloads from PNNL’s experience with graph applications
- Compare silicon-photonics systems with electrical counterpart
  - fix footprint; fix power
- Large, distributed graphs ("require a rank")
  - Validate at scale 34; Project at scale 40
  - Scale $\log_2(\text{edges})$
- Models explore both performance and power
- Model intra-node and inter-node data movement
Two Workloads To Represent Important Use Cases

**Community Detection**
- Input: Graph with weighted edges
- Output: Disjoint sets of related vertices
- Aggregated personalized all-to-all to send each edge’s target info (~1 GB)
- Iterate until Δ-modularity < threshold
  - Each vertex initially its own community
  - For each vertex, determine whether modularity increases by moving to neighboring community

**Matching (½ approx)**
- Input: Graph with weighted edges
- Output: Maximal weighted matching
- Two phases b/c of multi-step protocol
  - Based on locally dominant neighbor
- Phase 1:
  - Try matching each vertex
  - Aggregate messages between nodes
- Phase 2:
  - Try matching on “matched frontier”
  - Iterate until all vertices are matched
  - Use very small (24 B) messages

**Large, aggregated messages**
- Optimized for cluster networks
- Combine reqs with same target vertex

**More computation**
- Modularity requires collectives
- Denser graph; aggregation cost

**Small messages**

**Scale-40 distributed graphs**
Two Workloads To Represent Important Use Cases

Community Detection

- Input: Graph with weighted edges
- Output: Disjoint sets of related vertices
- Aggregated personalized all-to-all to send each edge’s target info (~1 GB)
- Iterate until $\Delta$-modularity < threshold
  - Each vertex initially its own community
  - For each vertex, determine whether modularity increases by moving to neighboring community

Using Palm...

Annotations convey insight about input graph

Capture important runtime properties. E.g.: probability that communities are formed

Large, aggregated messages

- Optimized for cluster networks
- Combine reqs with same target vertex

More computation

- Modularity requires collectives
- Denser graph; aggregation cost

Swap network models

Convenient representation

Challenge: Help specialize model for graph input class
Conclusions

- Ease burden of modeling
  - Facilitate divide-and-conquer modeling strategy
  - Automatically incorporate measurements
  - Generate contribution and error reports

- Enable first-class models
  - Coordinate models and source code
  - Functions unify annotations, generated models, and measurements

- Expressive: elegantly represent non-trivial critical paths
  - Annotations provide convenience within fully generic framework

- Reproducible: generate same model given same input
  - Generate model according to well-defined rules
  - Define model structure from static & dynamic code structure

- Future: Especially interested in more dynamic assistance