Pacific Northwest
NATIONAL LABORATORY
Proudly Operated by Battelle Since 1965

Palm: Easing the Burden of Analytical Performance Modeling

NATHAN TALLENT, ADOLFY HOISIE

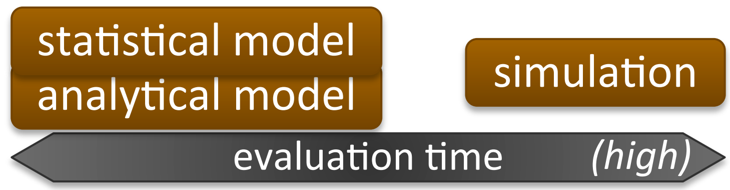
Pacific Northwest National Lab

Petascale Tools Workshop

August 5, 2014

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
  - Reproducing and distributing models is ad hoc
    - 1 modeler, N application variants
    - 1 application, N modelers

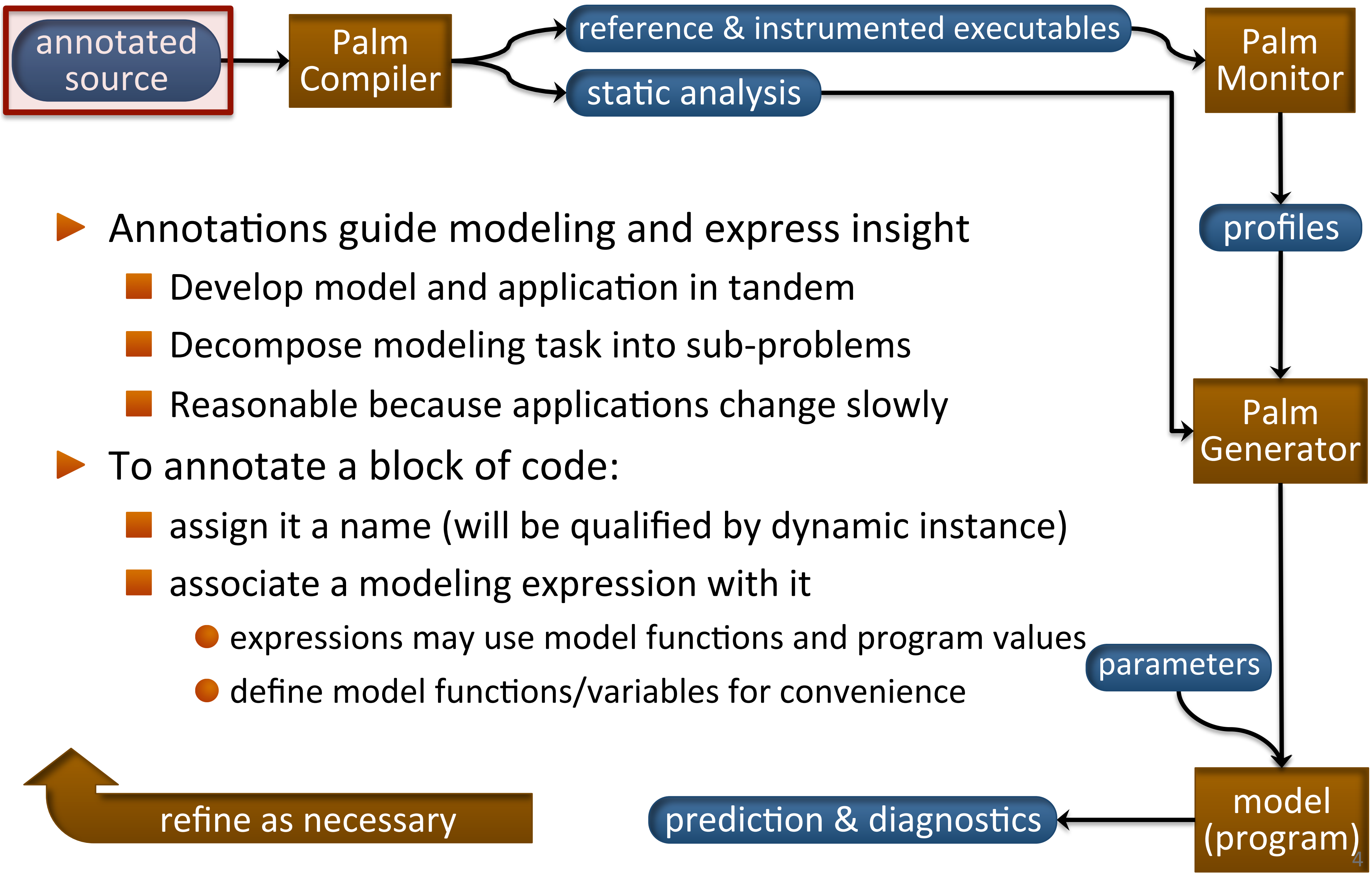
What can a tool automate? Can we pair model and source code?

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

Palm: Annotations Guide Modeling



- Annotations guide modeling and express insight
  - Develop model and application in tandem
  - Decompose modeling task into sub-problems
  - Reasonable because applications change slowly
- To annotate a block of code:
  - assign it a name (will be qualified by dynamic instance)
  - associate a modeling expression with it
    - expressions may use model functions and program values
    - define model functions/variables for convenience

Simple Annotations for Nekbone (CG solver)

```
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

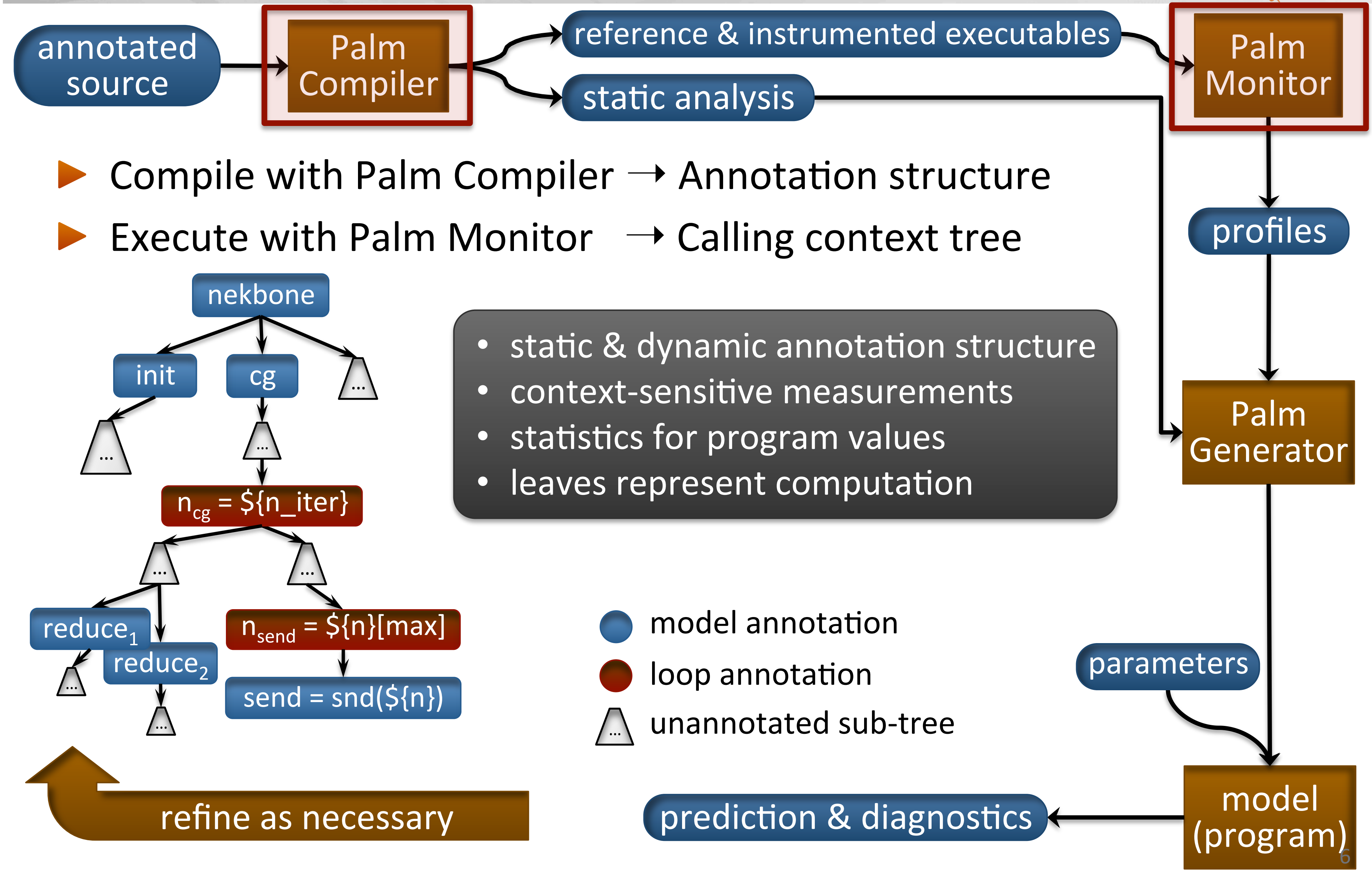
def: define model variable or function

${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: Static & Dynamic Analysis

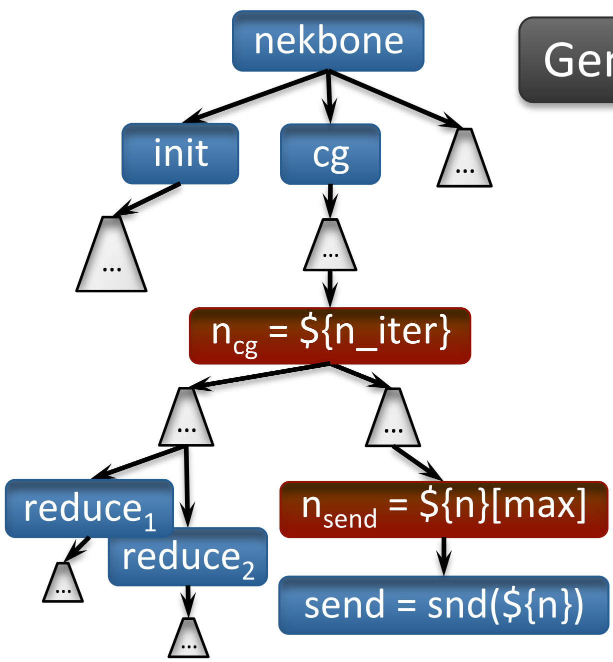


Palm: Generating Models

annotated source → Palm Compiler → reference & instrumented executables → Palm Monitor → profiles → Palm Generator

Palm Compiler → static analysis → Palm Generator

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



Generate model by composing model nodes

$$\text{nekbone}() = \text{init}() + \text{cg}() + k_2$$

$$\text{init}() = k_1$$

$$\text{cg}() = n_{cg} * (\text{reduce}_1() + \text{reduce}_2() + n_{send} * \text{snd}(k))$$

- model annotation
- loop annotation
- unannotated sub-tree

parameters → model (program) → prediction & diagnostics

refine as necessary

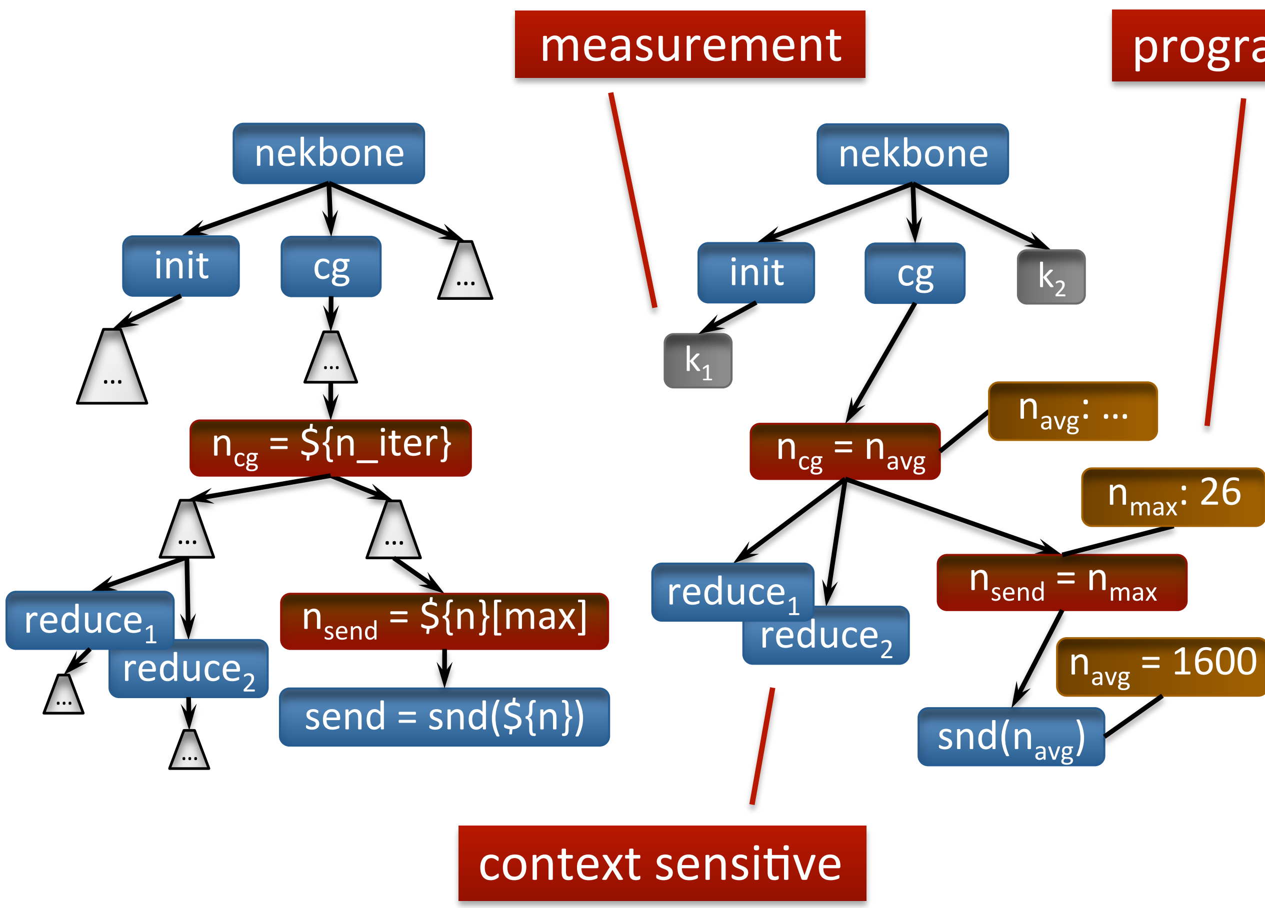
Model Generation for Nekbone

Calling Context Tree → **Model Tree** → **Model**

Calling Context Tree
- Annotation structure
- Performance measurements
- Program values (runtime)

Model Tree
- Model structure
- Convert sub-trees to measurement constants
- Resolve expression references

Model
- Annotation → model function
- Compose model functions
- Combine model expressions and measurements



$\text{nekbone}() = \text{init}() + \text{cg}() + k_2$

$\text{init}() = k_1$

$\text{cg}() = n_{cg} * (\text{reduce}_1() + \text{reduce}_2() + n_{send} * \text{snd}(1600))$

$\text{reduce}_1() = \dots$

$\text{snd}(sz) = \dots$

def-namespace

snd(sz) = ...

Palm: Using Models

annotated source → Palm Compiler → reference & instrumented executables → Palm Monitor → profiles → Palm Generator

Palm Compiler → static analysis → Palm Generator

- 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

parameters

Palm Generator → model (program) → prediction & diagnostics

refine as necessary

Palm's Model Matches Human-Generated Model

```
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)
```

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

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

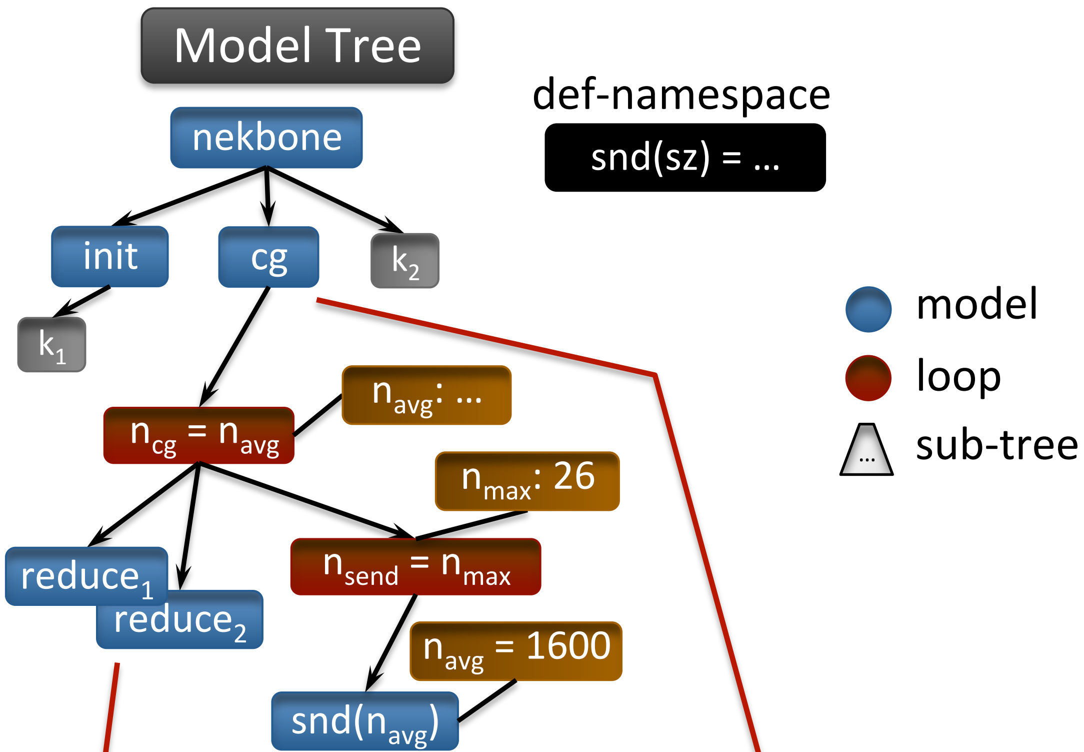
cg() model's form matches a human-generated model: $T_f + 3\ T_{reduce} + 26\ T_{send}$

model function (from def annotation)

machine parameters (from model library)

evaluate to obtain runtime

Models are Hierarchical



Palm Model

nekbone() = init() + cg() + k_2

init() = k_1

cg() = n_{cg} * (reduce$_1$() + reduce$_2$() + n_{send} * snd(1600))

reduce$_1$() = …

snd(sz) = …

model defined in terms of models; preserves annotation structure

a model node's subtree contains other model nodes

model annotations are context sensitive

Non-hierarchical Model

k_1 + n_{cg} * (reduce$_1$+ reduce$_2$ + n_{send} * …) + k_2

Models are First Class Values

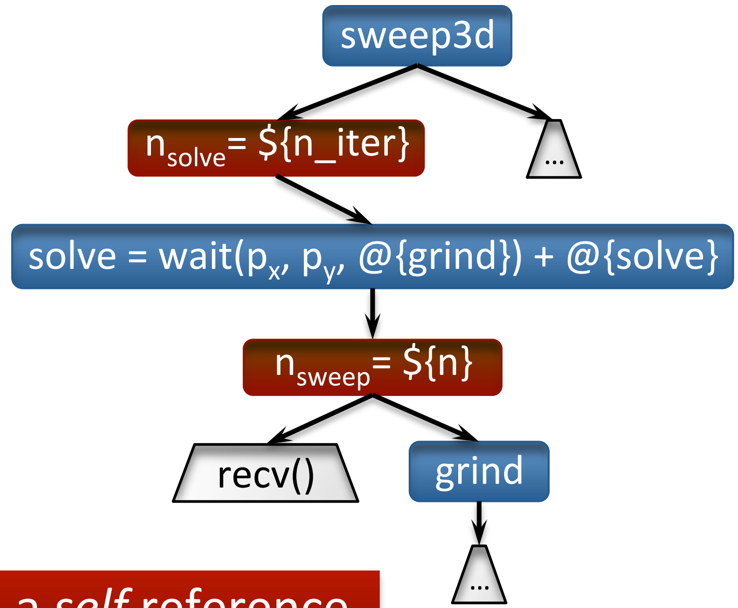
- Sweep3D: 2D pipeline where wait time depends on number ranks & pipeline stage.
- Easier to model aggregate wait time than per iteration wait time
- Use models as *values*

@{x}: model class reference: placeholder for x's (yet to be) *synthesized* model

"wait time plus myself (≈ compute)"

```
!$pal def wait(x, y, g) = (x + y − 1) g
!$pal model solve = wait(p_x, p_y, @{grind}) +@{solve})
call solve(…)
```

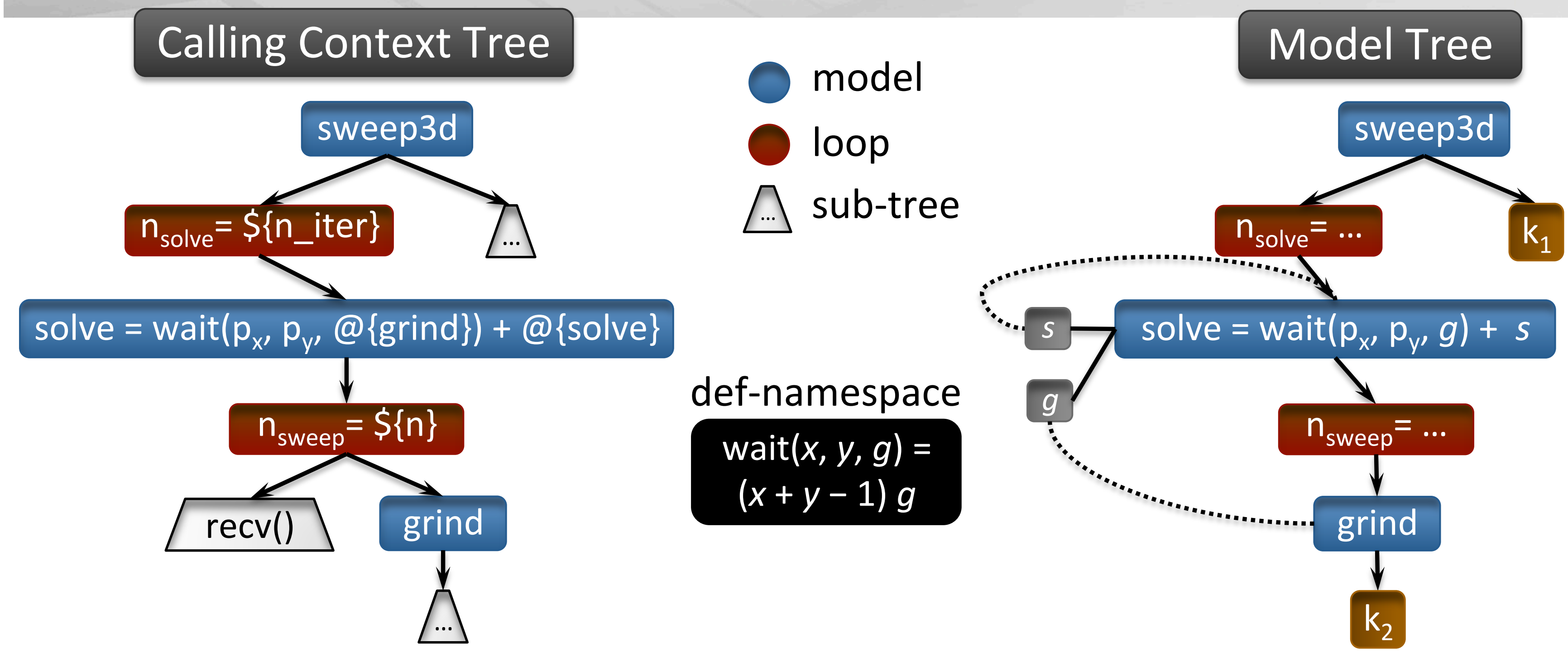
```
!$pal loop n_sweep = |dir| * |z-block|
for each dir and z-block b_z
   recv(pipeline-prev)
  [ #pal model grind
   compute(b_z)
   send(pipeline-next)
```



@{solve} is a *self* reference

@{grind} refers to the model for the tree fragment in this context

First Class Models Unify Models & Measurements



To permit recursive models, define an inductive ordering of model types:

2. annotation expression
1. synthesized expression (includes an annotation expression)
0. synthesized measurement

inductive case

base case

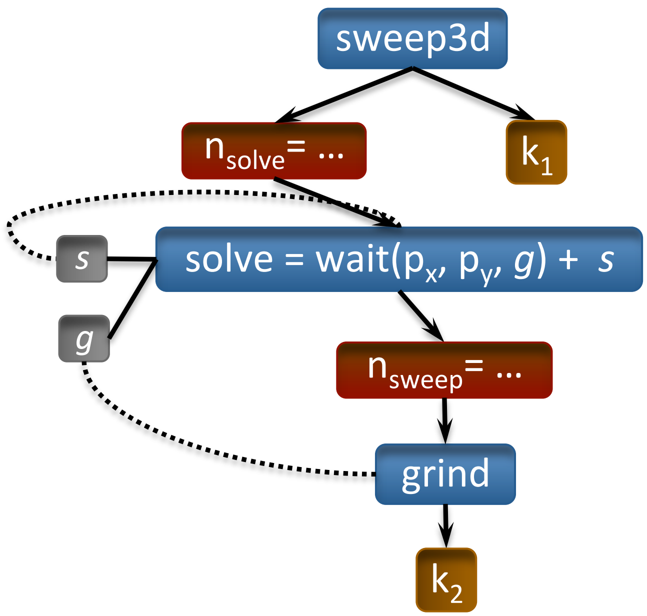
Model

solve(variant) =
  case variant
    ×1 g = grind(×1)
        s = sweep(×1′)
        wait(p_x, p_y, g) + s
    ×1′ n_{sweep} * grind (×1)

grind(variant) = k_2

Models and Accurate Measurements

Model Tree



- model
- loop
- sub-tree

def-namespace

wait(x, y, g) = ($x + y - 1$) g

```
def solve(variant)
    case variant
      ×1     g = grind(×1)
             s = sweep(×1')
             wait(p_x, p_y, g) + s
      ×1'    sweep(×n') / n_solve
      ×n     n_solve * sweep(×1)
      ×n'    grind(×n')
```

Two ways to measure:

1. time each instance & average
2. time many instances & divide

Method (2) is more accurate

Each model has four variants, a combination of

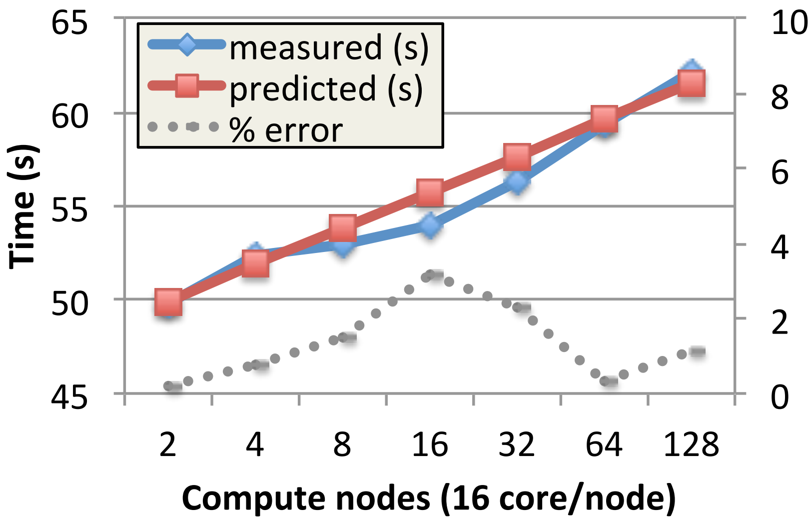
- instance types: per (×1) vs. multi (×*n*)
- model types: inductive vs. base (')

Examples:

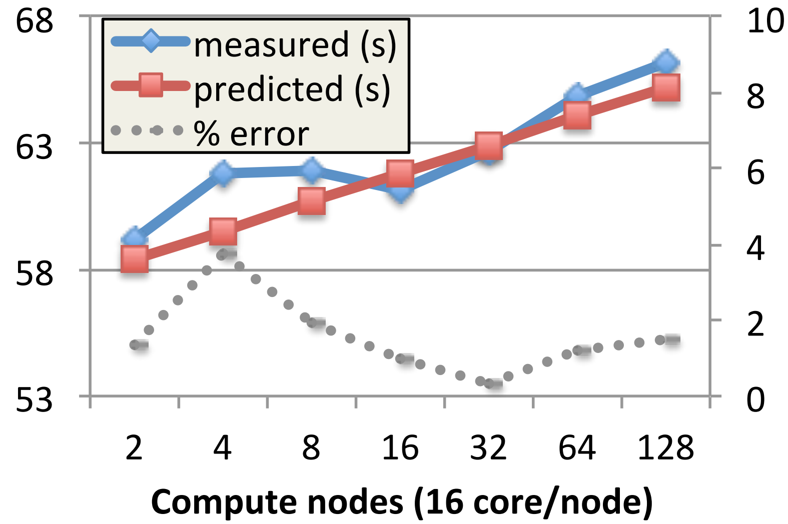
- sweep(×1): one instance
- sweep(×*n*): all instances

Results: Models Match Validated Models

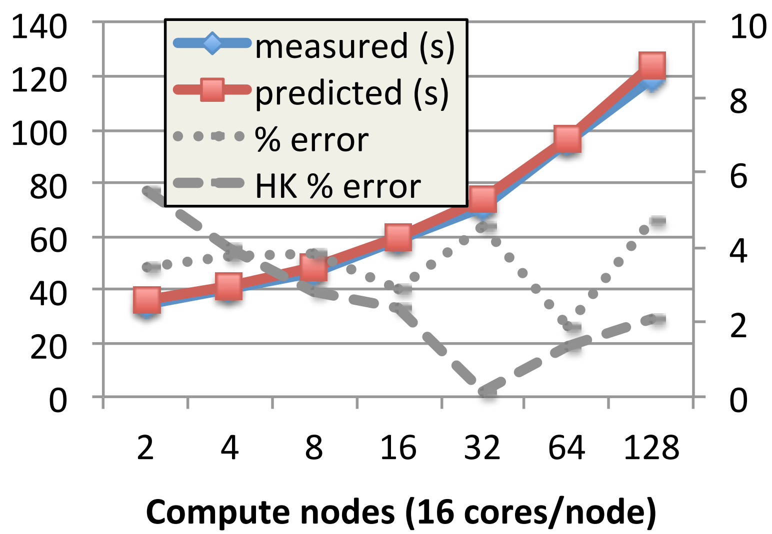
(a) Nekbone, weak scaling



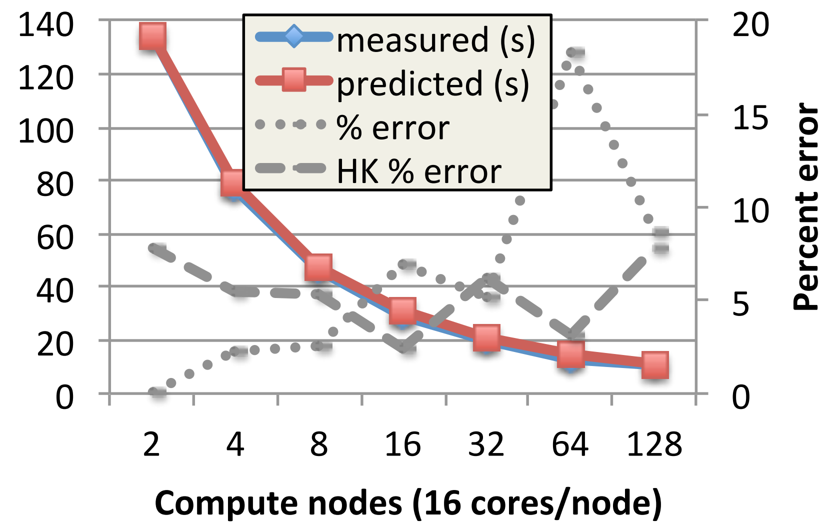
(b) GTC, weak scaling



(c) Sweep3D, weak scaling



(d) Sweep3D, strong scaling



Palm • hpc.pnnl.gov/palm

- 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