Estimating Parallel Performance, A Skeleton-Based Approach

Oleg Lobachev and Rita Loogen

Philipps Universität Marburg

HLPP 2010
September 25, 2010

Parallel Performance Measures

- Amdahl's law
- speedup = seq. time / parallel time
- efficiency
- serial fraction
- isoefficiency, scaled speedup, etc.

$\Rightarrow$ yet another one. Why?

. . . a Skeleton-Based Approach

- algorithmic skeletons = parallel algorithm abstractions
- in FP: higher-order functions
- skeletons as algorithm classification
- e.g., `map`-like, divide and conquer, iteration

⇒ use skeletons to designate types of parallel computation

The Essential Idea I

- different types of parallel programs run differently
- $\Rightarrow$ classification using skeletons

The Essential Idea I

- different types of parallel programs run differently
- $\Rightarrow$ classification using skeletons
- parallel programs do various things
  $\Rightarrow$ computation part + additional parallel overhead
- additional parallel overhead is harmful
- name it "parallel penalty"

The Essential Idea II

- n = input size, p = number of processors
- $T(n)$ = seq. runtime
- *assumption:* $T(n)$ = total amount of work
- $T(n, p)$ = parallel runtime
- traditional view:

$$T(n,p) = T(n)/\text{speedup}(n,p)$$

- our approach: $\bar{A}(n,p)$ = parallel penalty

$$T(n,p) = T(n)/p + \bar{A}(n,p)$$

The Essential Idea III

- sequential time $T(n)$ and parallel penalty $\bar{A}(n, p)$ are of different nature
  $\Rightarrow$ predict them separately!
- use statistical methods

The Essential Idea III

- sequential time $T(n)$ and parallel penalty $\bar{A}(n, p)$ are of different nature
  $\Rightarrow$ predict them separately!
- use statistical methods
- be aware: $\bar{A}(n, p)$ has two dimensions: input size n and number of processors p
  $\Rightarrow$ fix one, predict the other

Estimating Parallel Penalty and Runtime

"learning"

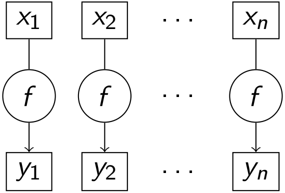
- measure $T(n)$ for several values of n
- measure $T(n, p)$ for several values of n or p keeping the other parameter fixed
- compute

$$\bar{A}(n, p) = T(n, p) - T(n)/p$$

prediction

- predict $T(n)$ for non-measured values of n
- predict $\bar{A}(n, p)$ for non-measured values of n or p
- compute $T(n, p)$ from above estimations

Parallel Map Skeletons I

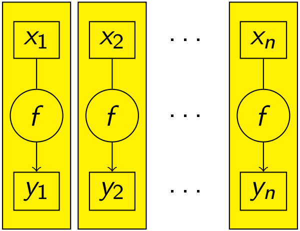


- `parMap` creates a process for each function application (task)
- *we assume:* same time needed for each list element

$$T(n) = nT(1)$$

$$\texttt{parMap}: \quad T(n,p) = n/p\ T(1) + \bar{A}(n,p)$$

Parallel Map Skeletons I

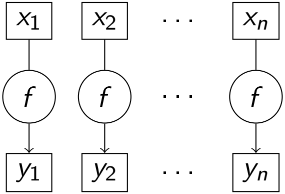


- `parMap` creates a process for each function application (task)
- *we assume:* same time needed for each list element

$$T(n) = nT(1)$$

`parMap`: $T(n, p) = n/p\ T(1) + \bar{A}(n, p)$

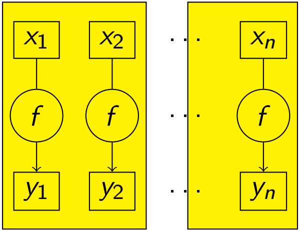
Parallel Map Skeletons II



- `farm`: static task distribution
  divide task list into blocks before the computation
- same *assumption:* same task size for all tasks

$$\texttt{farm}: \quad T(n,p) = T(n/p) + \bar{A}(n,p)$$

Parallel Map Skeletons II



- `farm`: static task distribution
  divide task list into blocks before the computation
- same *assumption:* same task size for all tasks

$$\texttt{farm}: \quad T(n,p) = T(n/p) + \bar{A}(n,p)$$

Parallel Iteration

$$\underbrace{\left.\begin{matrix} 1 & 2 & \dots & p \\ p+1 & p+2 & \dots & 2p \\ \vdots & \vdots & \ddots & \vdots \\ (l-1)p+1 & (l-1)p+2 & \dots & lp \end{matrix}\right\} l \text{ times}}_{p \text{ speculative tasks}}$$

- parallel `do-while`
- lp iterations, p speculative tasks, l “rounds”
- $s(n)$ cost for a single iteration

$$T(n) = lp\, s(n)$$
$$T(n,p,l) = l\, s(n) + \bar{A}(n,p)$$

How do we proceed?

- measure $T(n)$
- measure $T(n, p)$
- compute $\bar{A}(n, p)$ from them

scale w.r.t. input size

- $\hat{n}$ = non-measured input size
- estimate $T(\hat{n})$
- estimate $\bar{A}(\hat{n}, p)$
- compute $T(\hat{n}, p)$ from them

scale w.r.t. number of processors

- $\hat{p}$ = non-measured number of PEs
- $T(n)$ is known!
- estimate $\bar{A}(n, \hat{p})$
- compute $T(n, \hat{p})$ from them

How do we proceed?

- measure $T(n)$
- measure $T(n, p)$
- compute $\bar{A}(n, p)$ from them

scale w.r.t. input size

- $\hat{n}$ = non-measured input size
- estimate $T(\hat{n})$
- estimate $\bar{A}(\hat{n}, p)$
- compute $T(\hat{n}, p)$ from them

scale w.r.t. number of processors

- $\hat{p}$ = non-measured number of PEs
- $T(n)$ is known!
- estimate $\bar{A}(n, \hat{p})$
- compute $T(n, \hat{p})$ from them

How do we proceed?

- measure $T(n)$
- measure $T(n, p)$
- compute $\bar{A}(n, p)$ from them

scale w.r.t. input size

- $\hat{n}$ = non-measured input size
- estimate $T(\hat{n})$
- estimate $\bar{A}(\hat{n}, p)$
- compute $T(\hat{n}, p)$ from them

scale w.r.t. number of processors

- $\hat{p}$ = non-measured number of PEs
- $T(n)$ is known!
- estimate $\bar{A}(n, \hat{p})$
- compute $T(n, \hat{p})$ from them

Statistical methods

`spline`	interpolation method exact in specified points stepwise polynomials weighted extrapolation
`loess`	"local polynomial regression fitting" inexact: regression fitting stepwise polynomials special feature for extrapolation
`lm`	linear model fitting inexact: regression fitting fits a single line
`lm(poly)`	linear model fitting with orthogonal polynomials inexact: regression fitting relaxed with polynomials
`mean`	take average of the best two methods

A speedup analogy

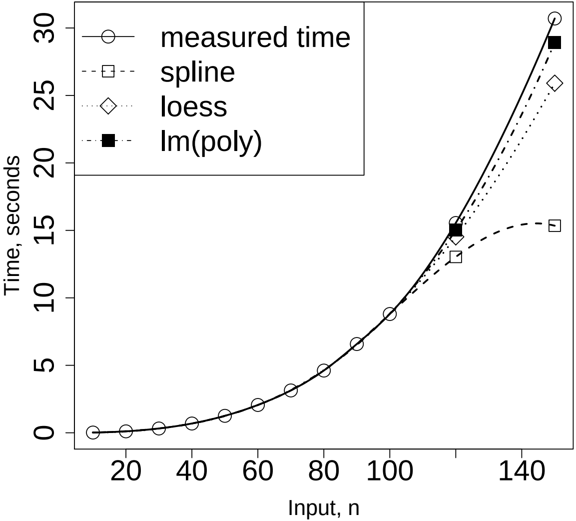
absolute speedup		relative speedup
	vs.	
$T(n, p)/T(n)$		$T(n, p)/T(n, 1)$

need sequential time to compute $\bar{A}(n, p)$

absolute reference point	vs.	relative reference point
$\bar{A}(n, p) = T(n, p) - T(n)/p$		$\bar{A}(n, p) = T(n, p) - T(n, 1)/p$

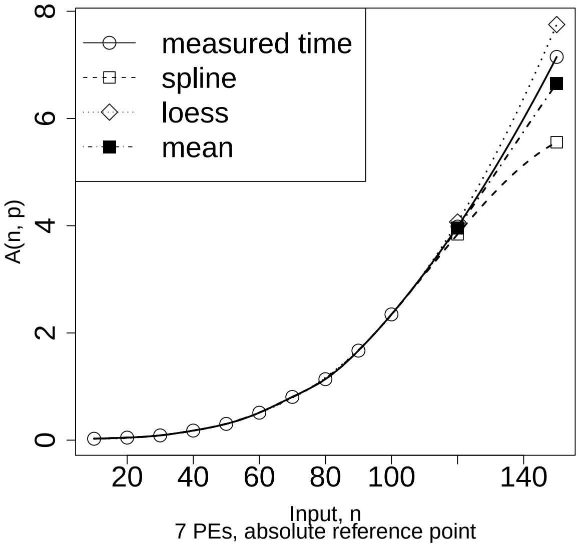
Gauß Elimination: Predicting Sequential Time

- matrix computations
- a `farm` instance
- predict $T(n)$ for $n = 120, 150$
- best: `lm(poly)`
- -3.41% rel. err for $n = 120$



Gauß Elimination: Predicting Parallel Penalty

- predict $\bar{A}(n, p)$ w.r.t. n with fixed $p = 7$ for $n = 120, 150$
- best: `mean`
- -0.73% rel. err for $n = 120$



Gauß Elimination: Estimation results

- want parallel runtime $T(\hat{n}, p)$ for $\hat{n} = 120$ and fixed $p = 7$
- combine estimations of $T(\hat{n})$ and $\bar{A}(\hat{n}, 7)$ w.r.t. $\hat{n}$ $\Rightarrow$ $T(\hat{n}, p)$ with an relative error -1.69%
- estimate $T(n, \hat{p})$ for non-measured $\hat{p}$: also possible $\Rightarrow$ see the paper, relative error -1.1%

Gauß Elimination: Estimation results

- want parallel runtime $T(\hat{n}, p)$ for $\hat{n} = 120$ and fixed $p = 7$
- combine estimations of $T(\hat{n})$ and $\bar{A}(\hat{n}, 7)$ w.r.t. $\hat{n}$
  $\Rightarrow$ $T(\hat{n}, p)$ with an relative error -1.69%
- estimate $T(n, \hat{p})$ for non-measured $\hat{p}$: also possible
  $\Rightarrow$ see the paper, relative error -1.1%

- considered implementation of Gauß Elimination always spawns 8 tasks
- our method does not know this!

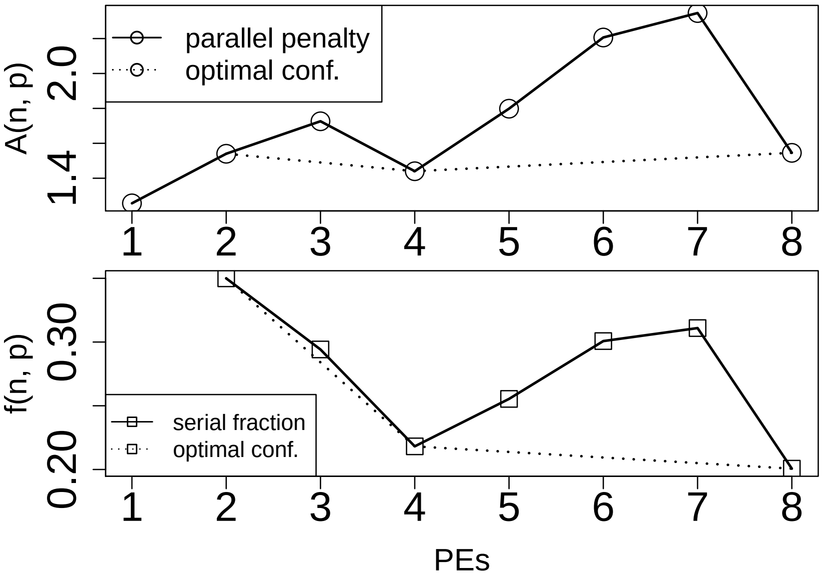
Serial Fraction

- serial fraction = measure for sequential part of the program

$$f(n,p) = \frac{T(n,p)/T(n) - 1/p}{1 - 1/p}$$

- parallel program quality measure
- should be constant

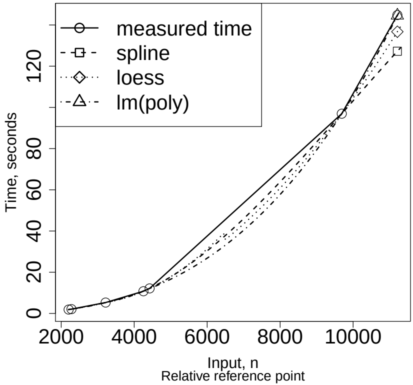
Parallel penalty vs. serial fraction for Gauß Elimination

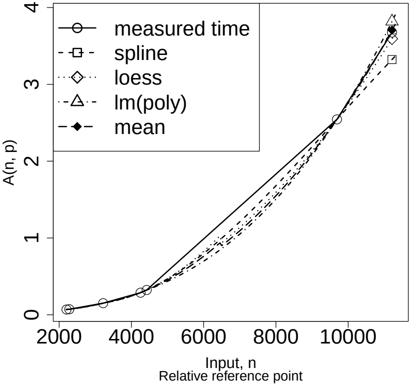


Rabin-Miller Test

- Rabin-Miller test = instance of iteration skeleton
- checks for primality
- definite answer in negative case
- is never sure in positive case
- *our implementation:* speculative iteration
  has load-balancing issues
  always starts 20 tasks
- predict for an input size $n = 11213$

Results for Rabin-Miller Test





predicting $T(n)$

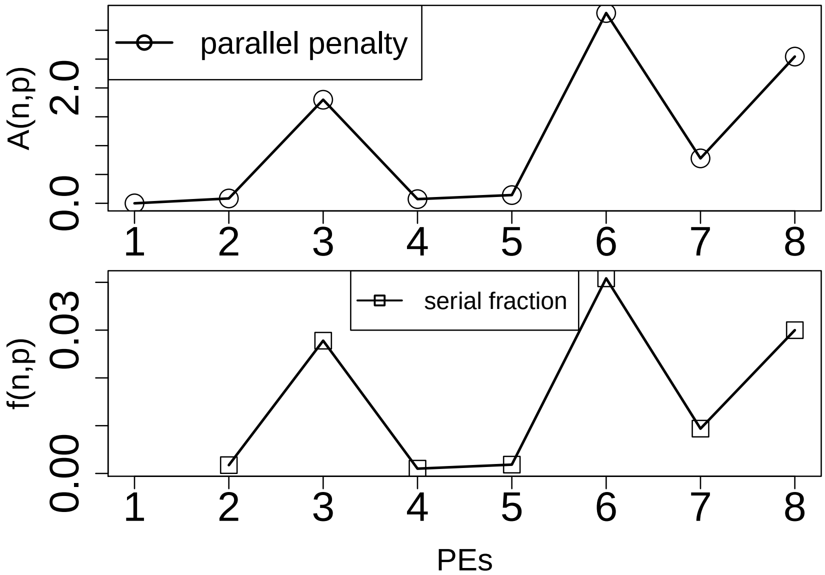
best: `lm(poly)`

predicting $\bar{A}(n, p)$

best: `mean`

combined: 0.01% rel. err

Parallel penalty vs. serial fraction for Rabin-Miller test



Conclusions and Future Work

Conclusions

- a method for parallel runtime estimation,
  better than a direct prediction
  for vast class of programs
  (more examples and experiments in the paper!)
- parallel penalty term as quality measure

Future work

- investigate relation to serial fraction further
- extend the formalism to further skeletons
- try other statistical methods
- try other prediction techniques — automated learning?

Karatsuba multiplication: outline

- fast integer multiplication
- instance of divide and conquer skeleton
- used with a fine-granular divide and conquer skeleton
- implemented in Eden — parallel Haskell extension
- distributed memory setting
- tested on a multicore

Karatsuba multiplication: results

$n \cdot 1000$	16	...	56	**60**	**64**
$T(n, 8)$	1.29	...	9.95	**11.0**	**11.86**
$T(n)$	9.88	...	74.39	**82.02**	**88.94**
$\bar{A}(n, 8) \cdot 100$	5.39	...	64.66	**74.22**	**74.65**
predict $T(n)$, rel. err, %				-0.014	1.9
predict $\bar{A}$, w.r.t. n, rel. err, %				2.3	2.08
predict $T(n, p)$, rel. err, %				0.14	1.78

Karatsuba multiplication: estimating $T(n)$