Dynamic Load Balancing of Parallel Computational Iterative Routines on Platforms with Memory Heterogeneity

David Clarke Alexey Lastovetsky Vladimir Rychkov

Heterogeneous Computing Laboratory
School of Computer Science and Informatics, University College Dublin,
Belfield, Dublin 4, Ireland
http://hcl.ucd.ie

HeteroPar'2010

Outline

Outline

Outline

Outline

- We present an algorithm for load balancing data-intensive parallel iterative routines

- We present an algorithm for load balancing data-intensive parallel iterative routines
- Target platform is a dedicated cluster with heterogeneous processors and heterogeneous distributed memory.

Layout

General Iterative Routine

$$x^{k+1} = f(x^k) \qquad k = 0, 1, ... \tag{1}$$

x^k is an n-dimensional vector
f is some function from $\mathbb{R}^n$ into itself.

General Iterative Routine

$$x^{k+1} = f(x^k) \qquad k = 0, 1, \ldots \tag{1}$$

x^k is an n-dimensional vector
f is some function from $\mathbb{R}^n$ into itself.

Parallel Routine

- Data is partitioned over all processors

General Iterative Routine

$$x^{k+1} = f(x^k) \qquad k = 0, 1, \ldots \tag{1}$$

x^k is an n-dimensional vector
f is some function from $\mathbb{R}^n$ into itself.

Parallel Routine

- Data is partitioned over all processors
- Some independent calculations are carried out in parallel

General Iterative Routine

$$x^{k+1} = f(x^k) \qquad k = 0, 1, ... \tag{1}$$

x^k is an n-dimensional vector
f is some function from $\mathbb{R}^n$ into itself.

Parallel Routine

- Data is partitioned over all processors
- Some independent calculations are carried out in parallel
- Some data synchronisation takes place

General Iterative Routine

$$x^{k+1} = f(x^k) \qquad k = 0, 1, \ldots \tag{1}$$

x^k is an n-dimensional vector
f is some function from $\mathbb{R}^n$ into itself.

Parallel Routine

- Data is partitioned over all processors
- Some independent calculations are carried out in parallel
- Some data synchronisation takes place

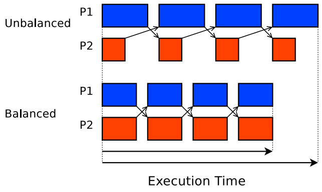
Typically computational workload is directly proportional to the size of data

Layout

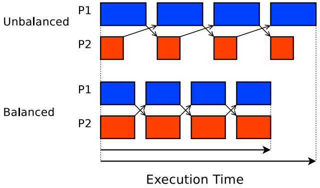
- Load balancing minimises overall computation time.

- Load balancing minimises overall computation time.
- All processors should complete an iteration in the same time.

- Load balancing minimises overall computation time.
- All processors should complete an iteration in the same time.



- Load balancing minimises overall computation time.
- All processors should complete an iteration in the same time.



- On a heterogeneous cluster this is achieved by partitioning data and calculations in proportion to processor speed.

Layout

Speed as a function of problem size

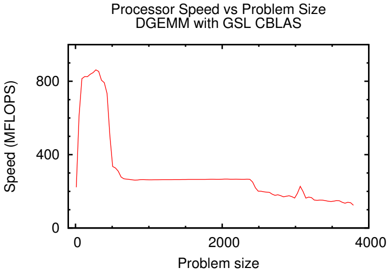
- Traditionally, processor performance is defined by a constant number.

Speed as a function of problem size

- Traditionally, processor performance is defined by a constant number.
- In reality, speed is a function of problem size.

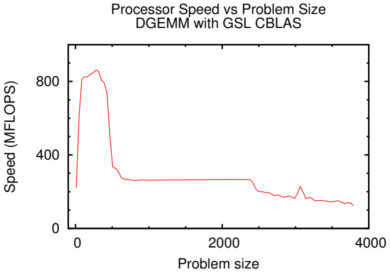
Speed as a function of problem size

- Traditionally, processor performance is defined by a constant number.
- In reality, speed is a function of problem size.



Speed as a function of problem size

- Traditionally, processor performance is defined by a constant number.
- In reality, speed is a function of problem size.
- Algorithms based on constant performance models are only applicable for limited problem sizes.



Layout

Traditional Load Balancing Algorithms

n computational units distributed across p processors.

Traditional Load Balancing Algorithms

n computational units distributed across p processors.
Processor P_i has d_i units such that $n = \sum_{i=1}^{p} d_i$

Traditional Load Balancing Algorithms

n computational units distributed across p processors.
Processor P_i has d_i units such that $n = \sum_{i=1}^{p} d_i$
Initially $d_i^0 = n/p$

Traditional Load Balancing Algorithms

n computational units distributed across p processors.
Processor P_i has d_i units such that $n = \sum_{i=1}^{p} d_i$
Initially $d_i^0 = n/p$
At each iteration

1. Execution times measured and gathered to root

Traditional Load Balancing Algorithms

n computational units distributed across p processors.
Processor P_i has d_i units such that $n = \sum_{i=1}^{p} d_i$
Initially $d_i^0 = n/p$
At each iteration

1. Execution times measured and gathered to root
2. **if** relative difference between times $\leq \epsilon$
   then no balancing needed

Traditional Load Balancing Algorithms

n computational units distributed across p processors.
Processor P_i has d_i units such that $n = \sum_{i=1}^{p} d_i$
Initially $d_i^0 = n/p$
At each iteration

1. Execution times measured and gathered to root
2. **if** relative difference between times $\leq \epsilon$
   then no balancing needed
   else new distribution is calculated as:
   $d_i^{k+1} = n \times \frac{s_i^k}{\sum_{j=1}^{p} s_j^k}$ where speed $s_i^k = \frac{d_i^k}{t_i(d_i^k)}$

Traditional Load Balancing Algorithms

n computational units distributed across p processors.
Processor P_i has d_i units such that $n = \sum_{i=1}^{p} d_i$
Initially $d_i^0 = n/p$
At each iteration

1. Execution times measured and gathered to root
2. **if** relative difference between times $\leq \epsilon$
   then no balancing needed
   else new distribution is calculated as:
   $d_i^{k+1} = n \times \frac{s_i^k}{\sum_{j=1}^{p} s_j^k}$ where speed $s_i^k = \frac{d_i^k}{t_i(d_i^k)}$
3. New distributions d_i^{k+1} broadcast to all processors and where necessary data is redistributed accordingly.

Layout

Traditional Load Balancing Algorithm

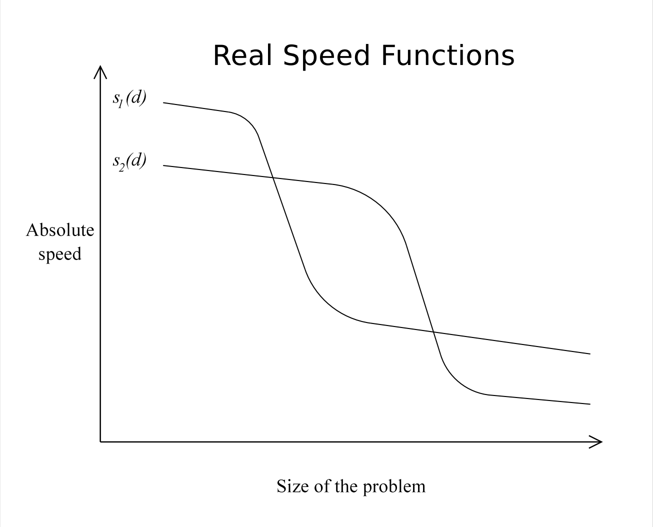
- Speed of each processor is considered as a constant positive number at each iteration.

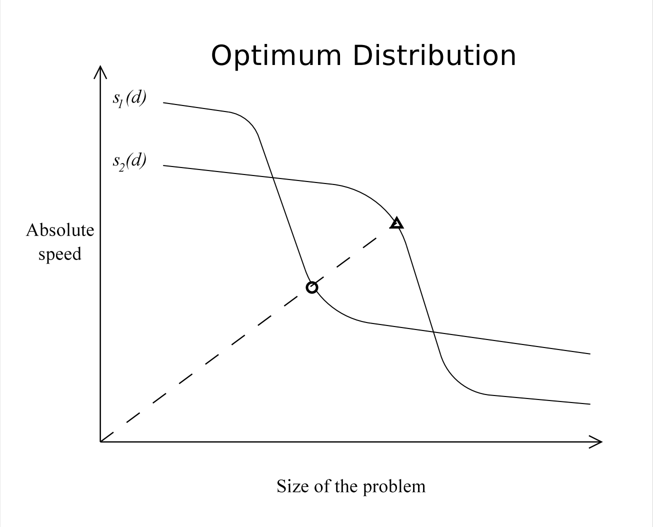
Traditional Load Balancing Algorithm

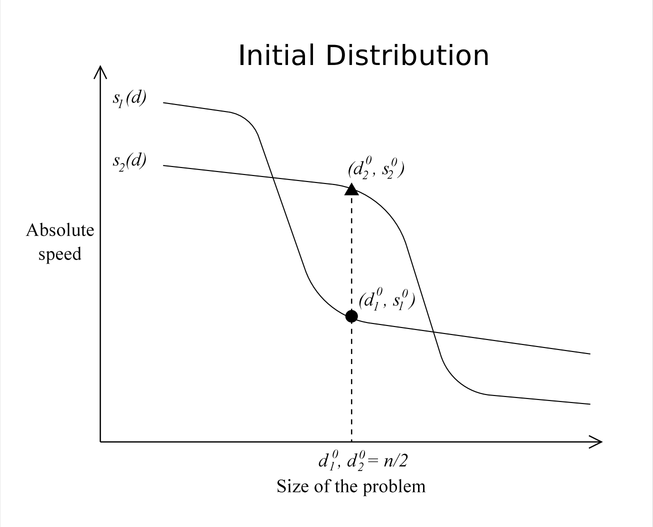
- Speed of each processor is considered as a constant positive number at each iteration.
- Within the range of problem sizes for which this is true, traditional algorithms can successfully load balance.

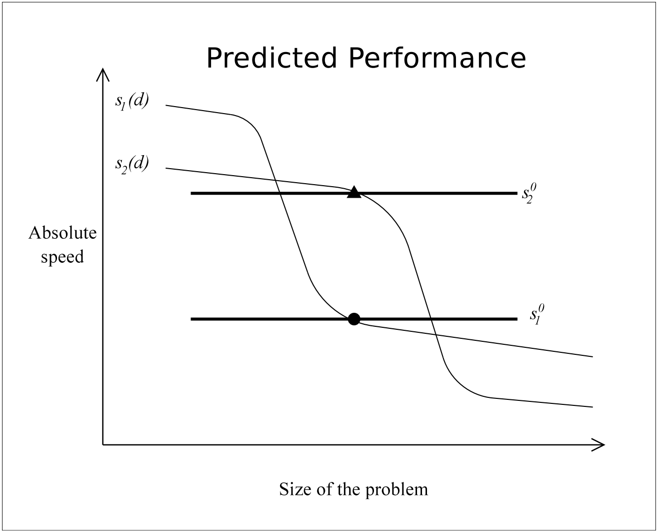
Traditional Load Balancing Algorithm

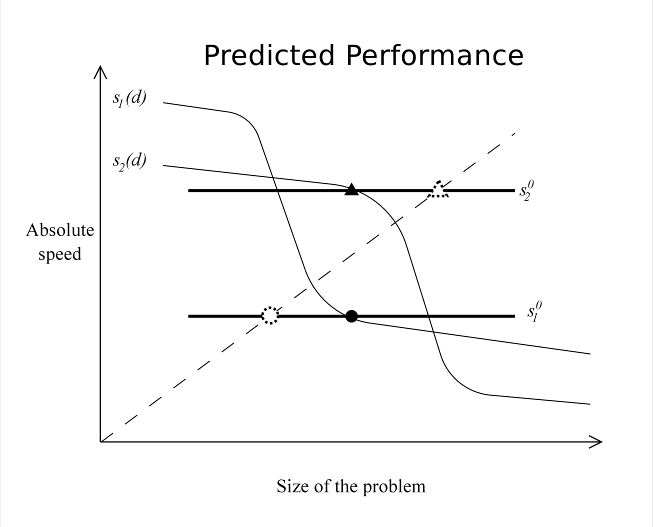
- Speed of each processor is considered as a constant positive number at each iteration.
- Within the range of problem sizes for which this is true, traditional algorithms can successfully load balance.
- Can fail for problem sizes for which the speed is not constant.

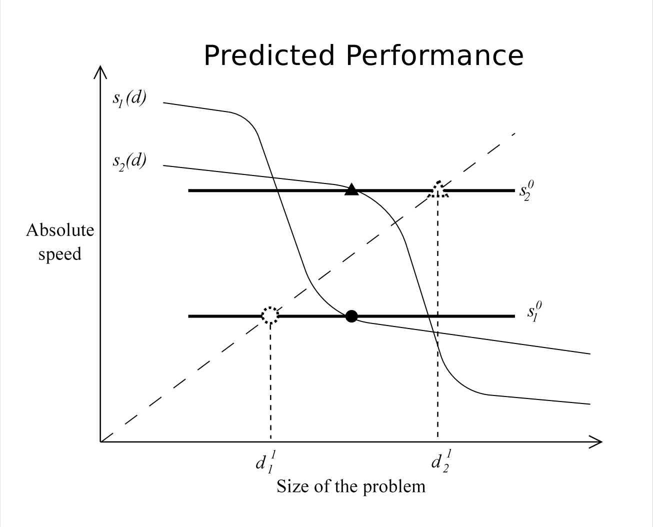

Real Speed Functions
$s_1(d)$
$s_2(d)$
Absolute speed
Size of the problem

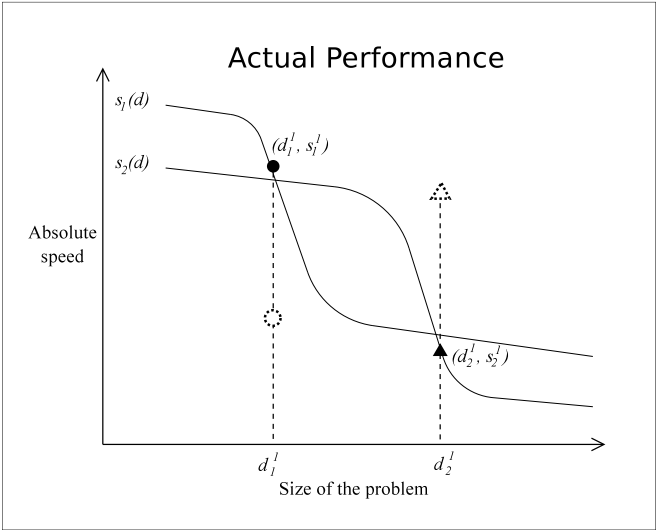

Optimum Distribution
$s_1(d)$
$s_2(d)$
Absolute speed
Size of the problem

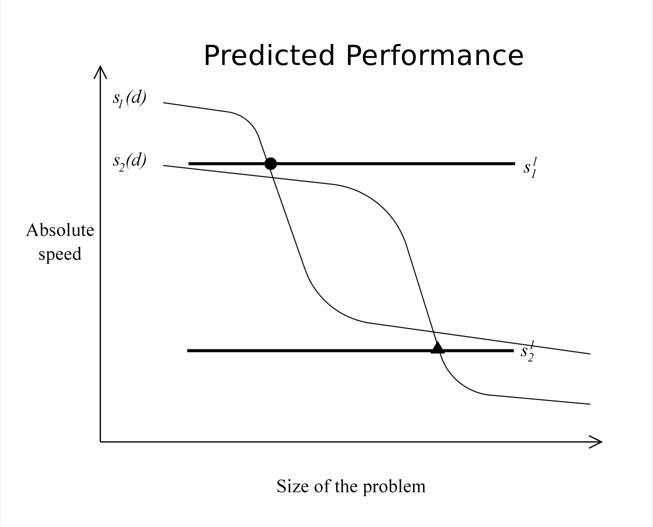

Initial Distribution
$s_1(d)$
$s_2(d)$
(d_2^0, s_2^0)
(d_1^0, s_1^0)
Absolute speed
$d_1^0, d_2^0 = n/2$
Size of the problem

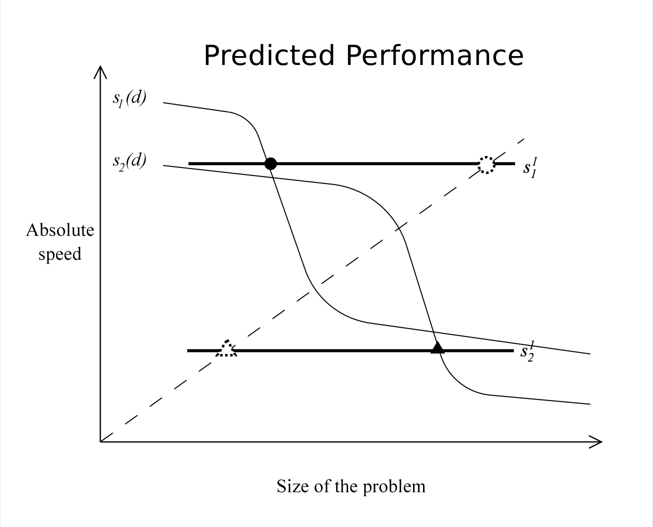

Predicted Performance
$s_1(d)$
$s_2(d)$
s_2^0
s_1^0
Absolute speed
Size of the problem

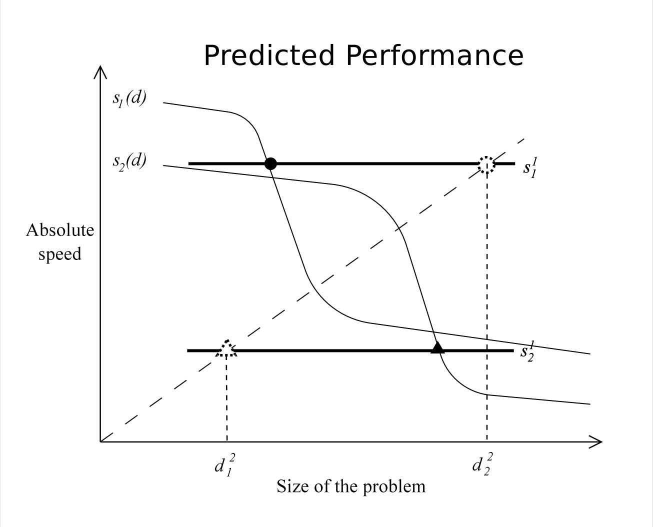

Predicted Performance
$s_1(d)$
$s_2(d)$
s_2^0
s_1^0
Absolute speed
Size of the problem

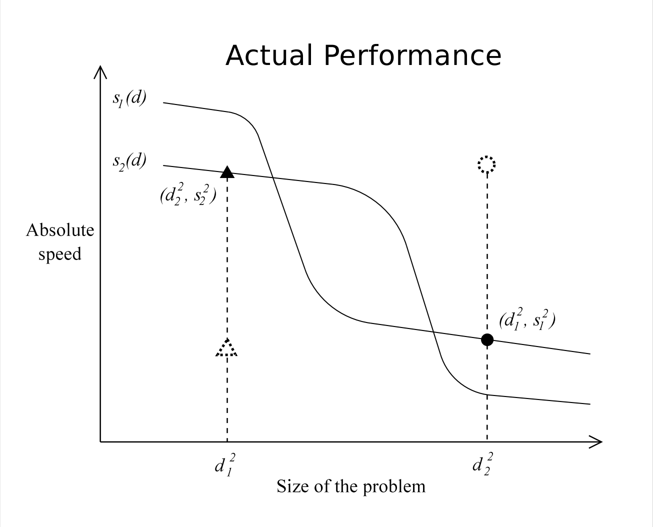

Predicted Performance
$s_1(d)$
$s_2(d)$
s_2^0
s_1^0
Absolute speed
d_1^1
d_2^1
Size of the problem


Actual Performance
$s_1(d)$
$s_2(d)$
(d_1^1, s_1^1)
(d_2^1, s_2^1)
Absolute speed
d_1^1
d_2^1
Size of the problem


Predicted Performance
$s_1(d)$
$s_2(d)$
s_1^1
s_2^1
Absolute speed
Size of the problem


Predicted Performance
$s_1(d)$
$s_2(d)$
s_1^1
s_2^1
Absolute speed
Size of the problem


Predicted Performance
$s_1(d)$
$s_2(d)$
s_1^1
s_2^1
Absolute speed
d_1^2
d_2^2
Size of the problem


Actual Performance
$s_1(d)$
$s_2(d)$
(d_2^2, s_2^2)
(d_1^2, s_1^2)
Absolute speed
d_1^2
d_2^2
Size of the problem

Layout

Iterative Routine

Jacobi method for solving a system of linear equations.

Iterative Routine

Jacobi method for solving a system of linear equations.

Experimental Setup

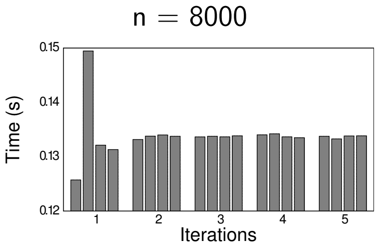
	P_1	P_2	P_3	P_4
Processor	3.6 Xeon	3.0 Xeon	3.4 P4	3.4 Xeon
Ram (MB)	256	256	512	1024

Iterative Routine

Jacobi method for solving a system of linear equations.

Experimental Setup

	P_1	P_2	P_3	P_4
Processor	3.6 Xeon	3.0 Xeon	3.4 P4	3.4 Xeon
Ram (MB)	256	256	512	1024

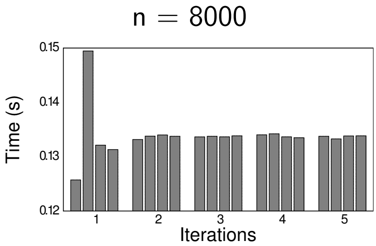


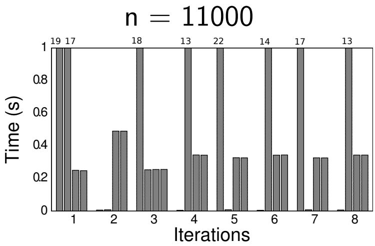
Iterative Routine

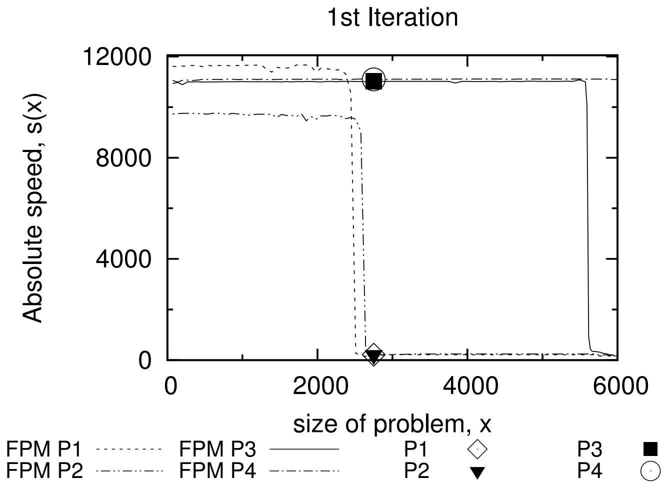
Jacobi method for solving a system of linear equations.

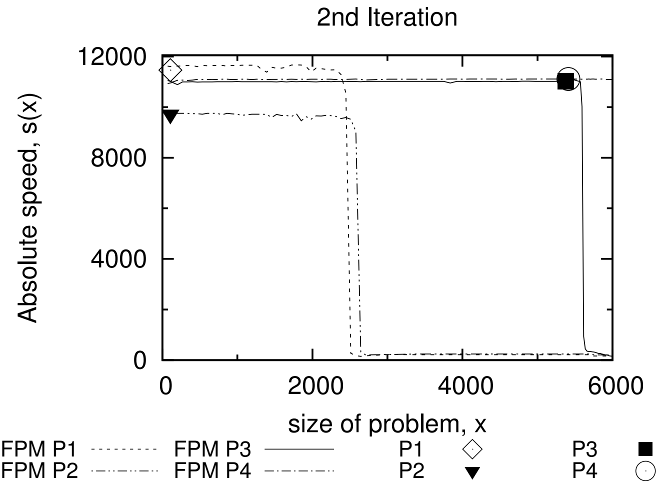
Experimental Setup

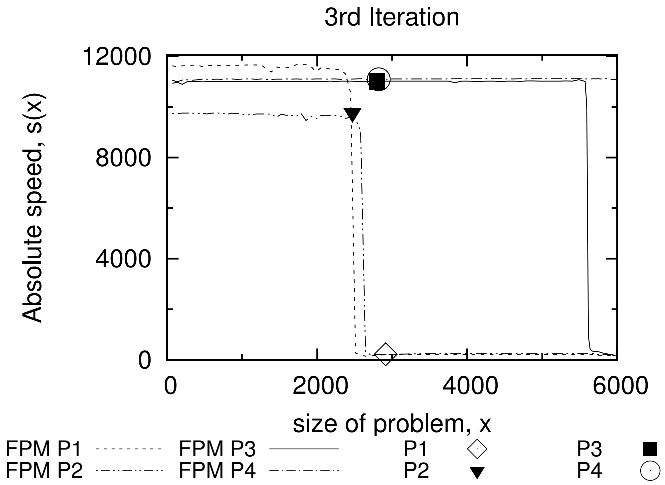
	P_1	P_2	P_3	P_4
Processor	3.6 Xeon	3.0 Xeon	3.4 P4	3.4 Xeon
Ram (MB)	256	256	512	1024

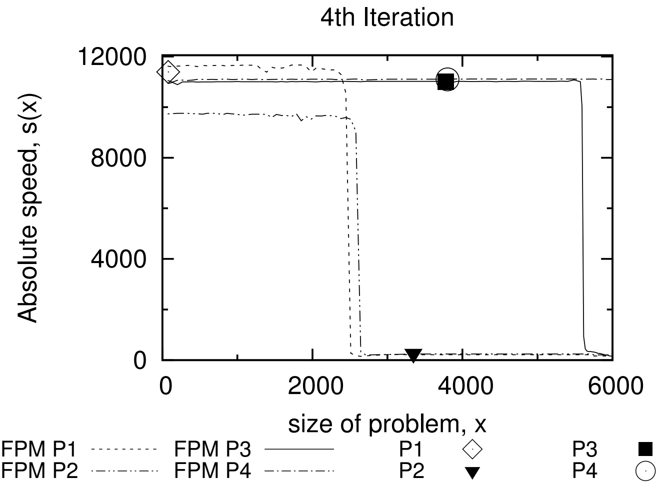


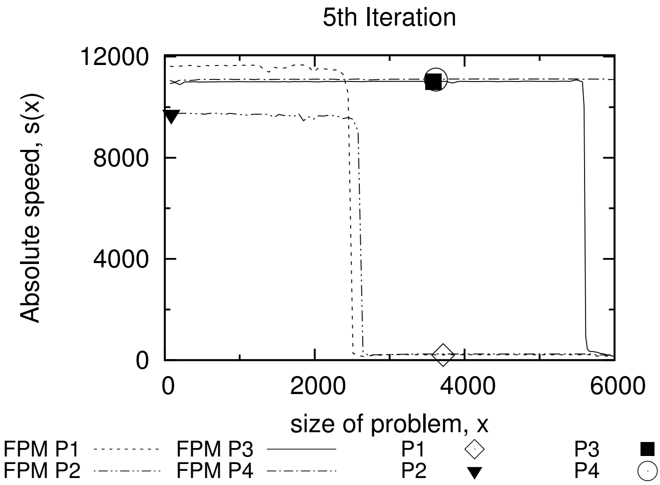



1st Iteration
Absolute speed, s(x)
12000
8000
4000
0
0
2000
4000
6000
size of problem, x
FPM P1
FPM P2
FPM P3
FPM P4
P1
P2
P3
P4


2nd Iteration
Absolute speed, s(x)
12000
8000
4000
0
0
2000
4000
6000
size of problem, x
FPM P1
FPM P2
FPM P3
FPM P4
P1
P2
P3
P4


3rd Iteration
Absolute speed, s(x)
12000
8000
4000
0
0
2000
4000
6000
size of problem, x
FPM P1
FPM P2
FPM P3
FPM P4
P1
P2
P3
P4


4th Iteration
12000
8000
4000
0
0
2000
4000
6000
Absolute speed, s(x)
size of problem, x
FPM P1
FPM P2
FPM P3
FPM P4
P1
P2
P3
P4


5th Iteration
12000
8000
4000
0
0
2000
4000
6000
Absolute speed, s(x)
size of problem, x
FPM P1
FPM P2
FPM P3
FPM P4
P1
P2
P3
P4

Layout

New Dynamic Load Balancing Algorithm

- Our algorithm is based on models for which speed is a function of problem size.

New Dynamic Load Balancing Algorithm

- Our algorithm is based on models for which speed is a function of problem size.
- Load balancing achieved when:

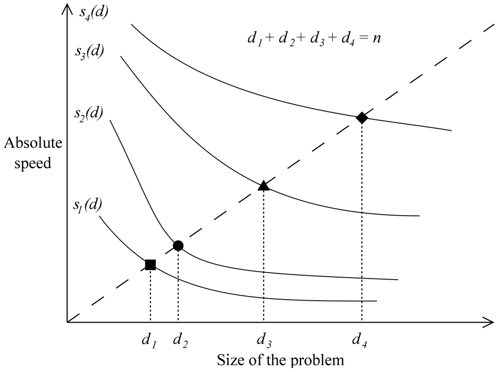
$$t_i \approx t_j, \quad 1 \leq i, j \leq p \tag{2}$$

$$\frac{d_1}{s_1(d_1)} \approx \frac{d_2}{s_2(d_2)} \approx \cdots \approx \frac{d_p}{s_p(d_p)} \tag{3}$$

where $d_1 + d_2 + \cdots + d_p = n$

Solving Distribution Problem

- Problem is solved geometrically by noting that the points $(d_i, s_i(d_i))$ lie on a line passing through the origin when $\frac{d_i}{s_i(d_i)} = constant$.



Our Dynamic Load Balancing Algorithm

- These functional performance models are different for each routine on each processor.

Our Dynamic Load Balancing Algorithm

- These functional performance models are different for each routine on each processor.
- Building these models for all conceivable problem sizes is very computationally expensive.

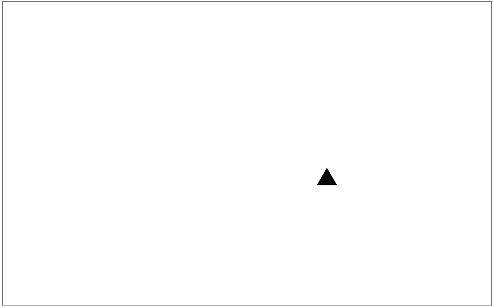
Our Dynamic Load Balancing Algorithm

- These functional performance models are different for each routine on each processor.
- Building these models for all conceivable problem sizes is very computationally expensive.
- Building full models is not an option for a self adaptive algorithm.

Our Dynamic Load Balancing Algorithm

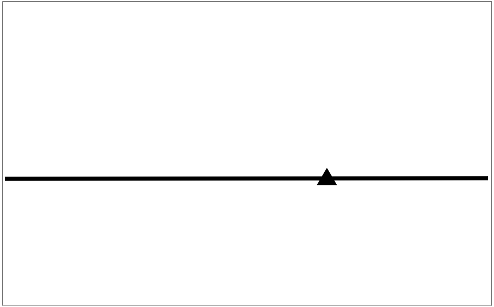
- These functional performance models are different for each routine on each processor.
- Building these models for all conceivable problem sizes is very computationally expensive.
- Building full models is not an option for a self adaptive algorithm.
- Our algorithm dynamically builds the models at relevant problem sizes using piecewise linear approximations.

First iteration Point $(\frac{n}{p}, s_i^0)$ with speed $s_i^0 = \frac{n/p}{t_i(n/p)}$



First iteration Point $(\frac{n}{p}, s_i^0)$ with speed $s_i^0 = \frac{n/p}{t_i(n/p)}$

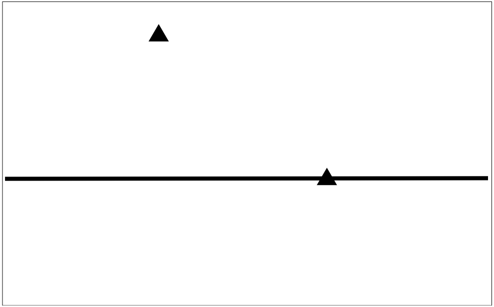
First function approximation $s_i'(d) = s_i^0$



First iteration Point $(\frac{n}{p}, s_i^0)$ with speed $s_i^0 = \frac{n/p}{t_i(n/p)}$

First function approximation $s_i'(d) = s_i^0$

Subsequent iterations Point (d_i^k, s_i^k) with speed $s_i^k = \frac{d_i^k}{t_i(d_i^k)}$

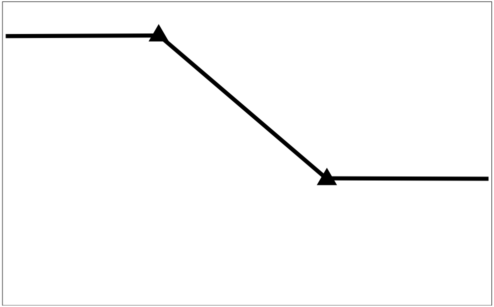


First iteration Point $(\frac{n}{p}, s_i^0)$ with speed $s_i^0 = \frac{n/p}{t_i(n/p)}$

First function approximation $s_i'(d) = s_i^0$

Subsequent iterations Point (d_i^k, s_i^k) with speed $s_i^k = \frac{d_i^k}{t_i(d_i^k)}$

Function approximation updated by adding the point

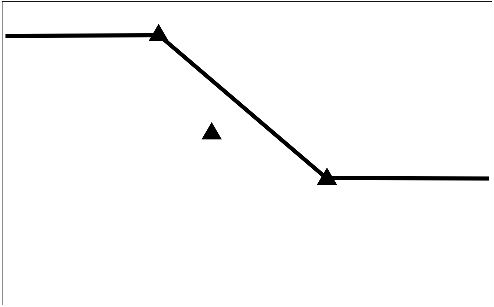


First iteration Point $(\frac{n}{p}, s_i^0)$ with speed $s_i^0 = \frac{n/p}{t_i(n/p)}$

First function approximation $s_i'(d) = s_i^0$

Subsequent iterations Point (d_i^k, s_i^k) with speed $s_i^k = \frac{d_i^k}{t_i(d_i^k)}$

Function approximation updated by adding the point

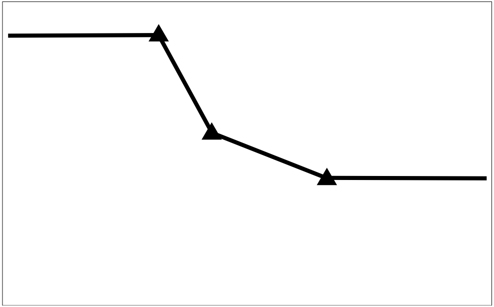


First iteration Point $(\frac{n}{p}, s_i^0)$ with speed $s_i^0 = \frac{n/p}{t_i(n/p)}$

First function approximation $s_i'(d) = s_i^0$

Subsequent iterations Point (d_i^k, s_i^k) with speed $s_i^k = \frac{d_i^k}{t_i(d_i^k)}$

Function approximation updated by adding the point

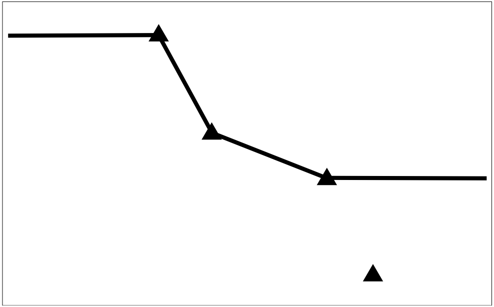


First iteration Point $(\frac{n}{p}, s_i^0)$ with speed $s_i^0 = \frac{n/p}{t_i(n/p)}$

First function approximation $s_i'(d) = s_i^0$

Subsequent iterations Point (d_i^k, s_i^k) with speed $s_i^k = \frac{d_i^k}{t_i(d_i^k)}$

Function approximation updated by adding the point

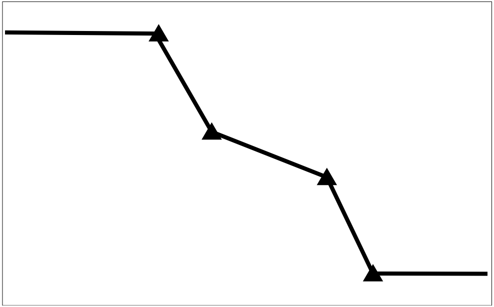


First iteration Point $(\frac{n}{p}, s_i^0)$ with speed $s_i^0 = \frac{n/p}{t_i(n/p)}$

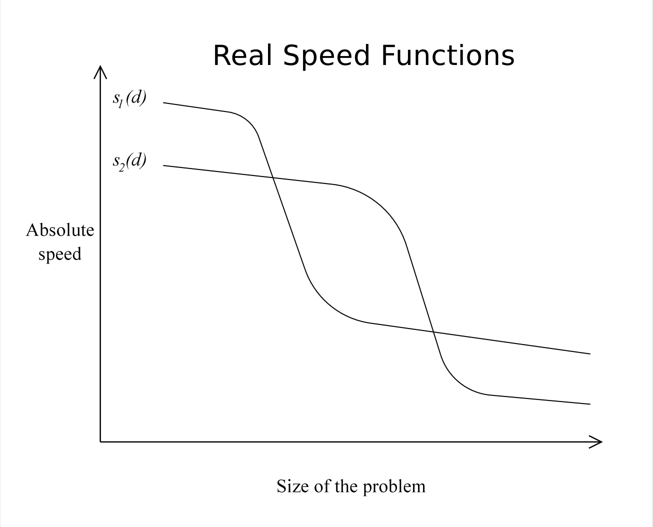
First function approximation $s_i'(d) = s_i^0$

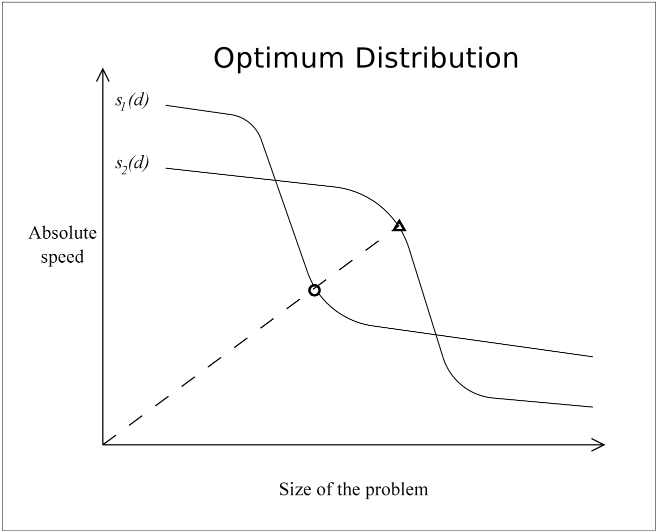
Subsequent iterations Point (d_i^k, s_i^k) with speed $s_i^k = \frac{d_i^k}{t_i(d_i^k)}$

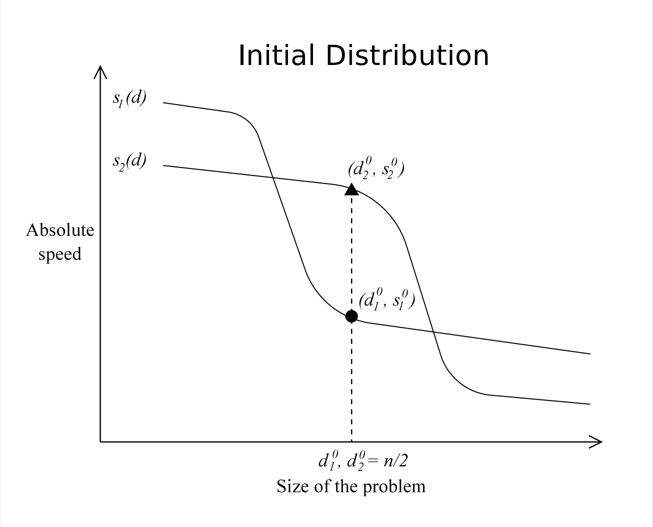
Function approximation updated by adding the point

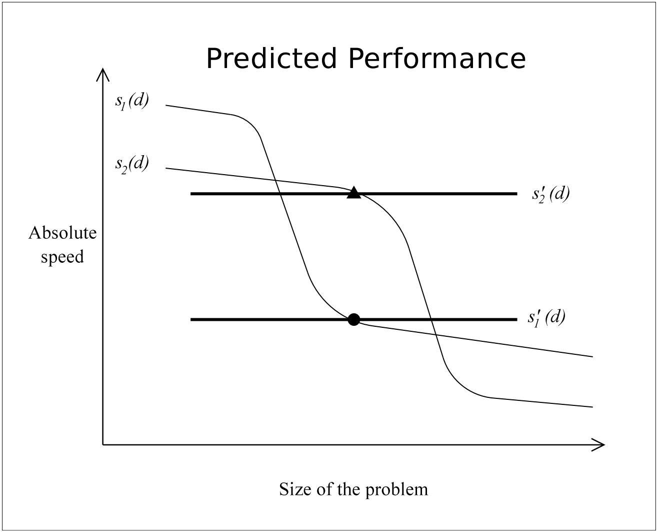


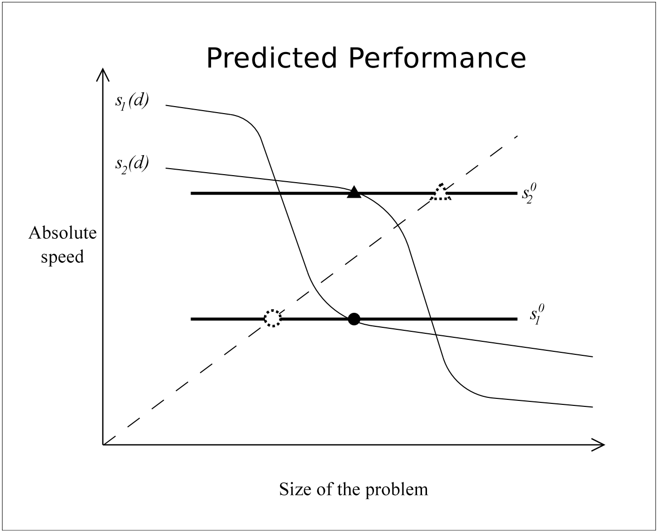
Layout

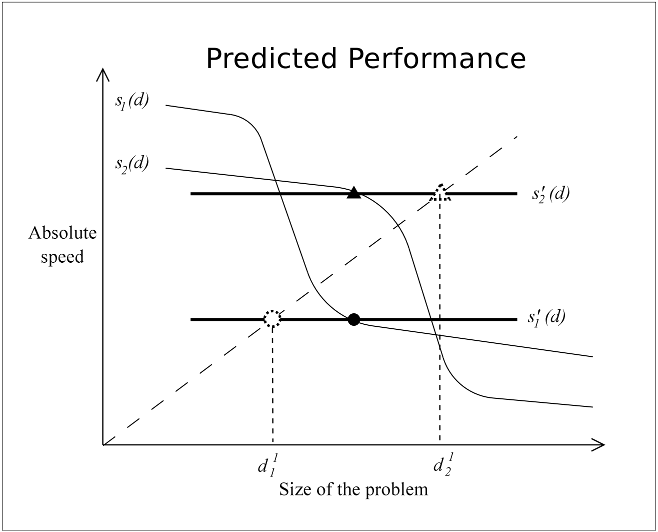

Real Speed Functions
$s_1(d)$
$s_2(d)$
Absolute speed
Size of the problem

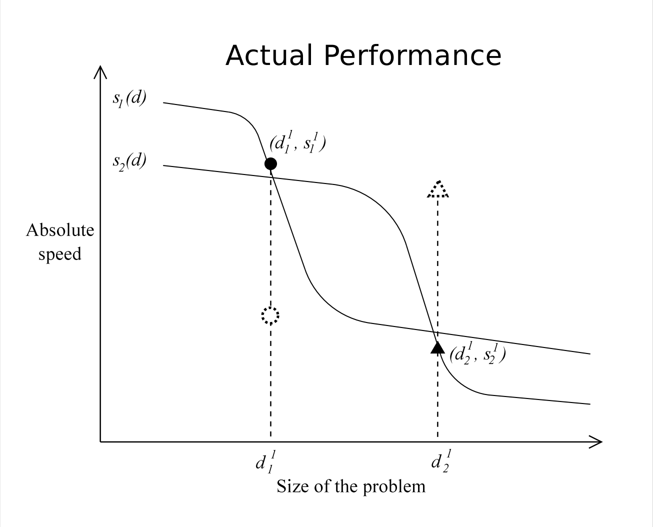

Optimum Distribution
$s_1(d)$
$s_2(d)$
Absolute speed
Size of the problem

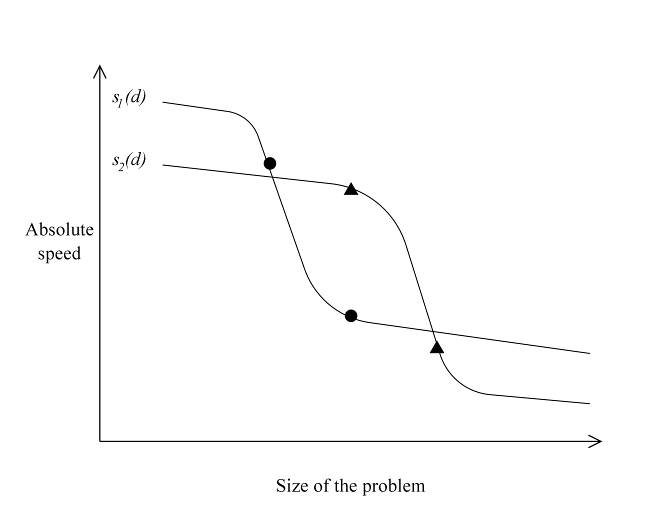

Initial Distribution
$s_1(d)$
$s_2(d)$
(d_2^0, s_2^0)
(d_1^0, s_1^0)
Absolute speed
$d_1^0, d_2^0 = n/2$
Size of the problem

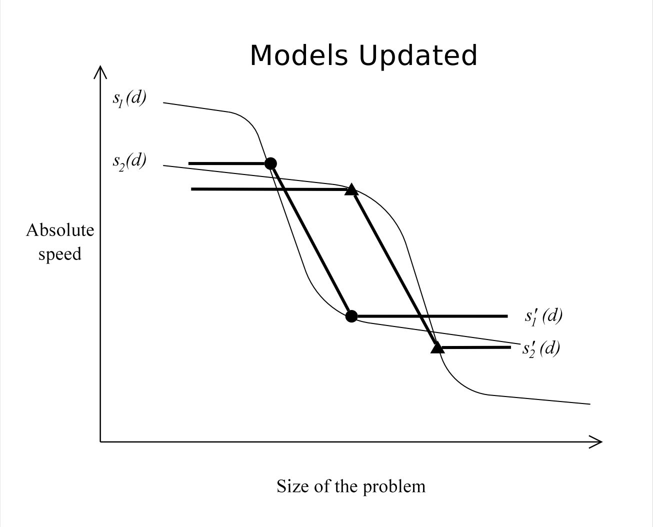

Predicted Performance
s1(d)
s2(d)
s2'(d)
s1'(d)
Absolute speed
Size of the problem

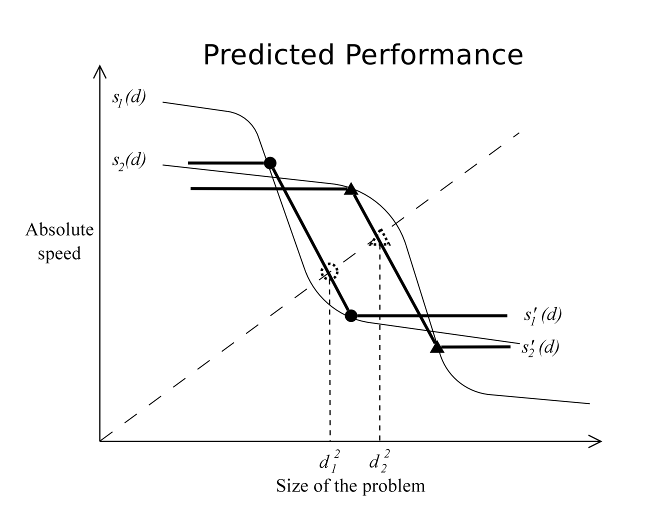

Predicted Performance
s1(d)
s2(d)
s2 0
s1 0
Absolute speed
Size of the problem

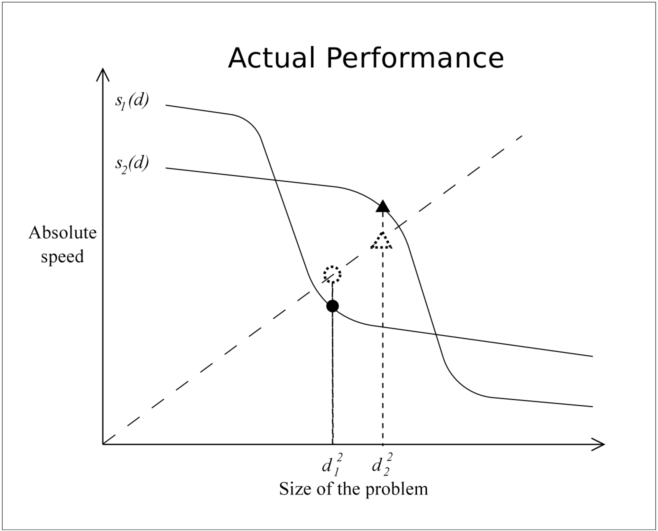

Predicted Performance
$s_1(d)$
$s_2(d)$
$s'_2(d)$
$s'_1(d)$
Absolute speed
d_1^1
d_2^1
Size of the problem

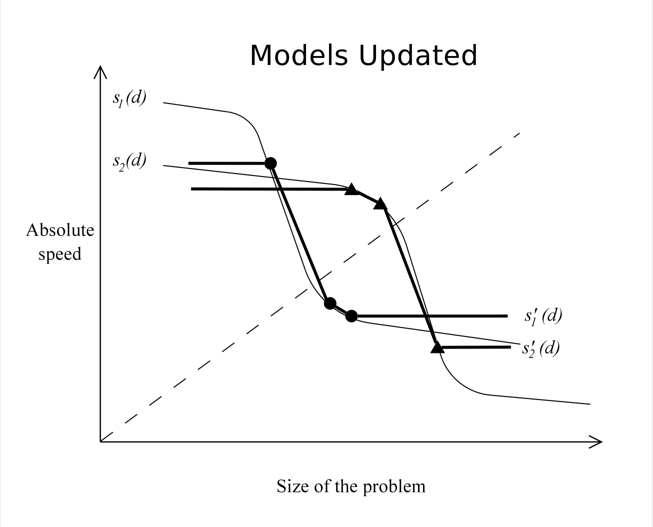

Actual Performance
$s_1(d)$
$s_2(d)$
(d_1^1, s_1^1)
(d_2^1, s_2^1)
Absolute speed
d_1^1
d_2^1
Size of the problem

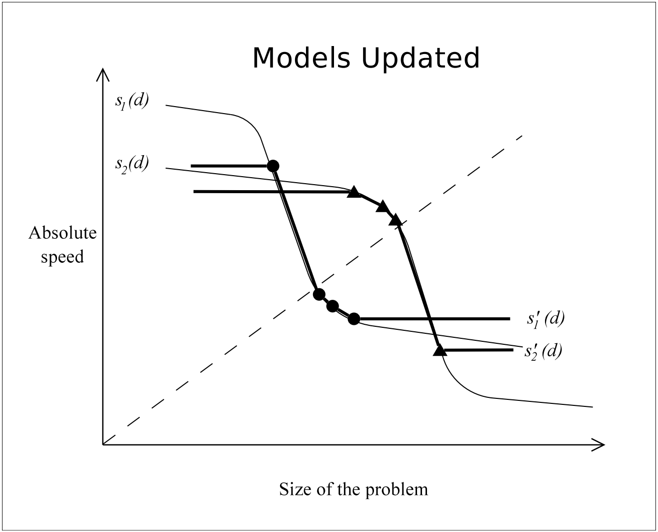

s1(d)
s2(d)
Absolute speed
Size of the problem


Models Updated
$s_1(d)$
$s_2(d)$
Absolute speed
$s_1'(d)$
$s_2'(d)$
Size of the problem


Predicted Performance
$s_1(d)$
$s_2(d)$
Absolute speed
$s_1'(d)$
$s_2'(d)$
d_1^2
d_2^2
Size of the problem

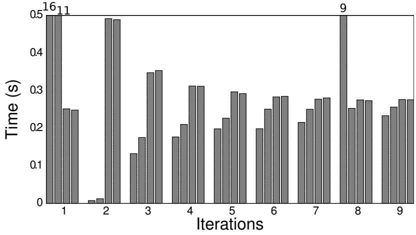

Actual Performance
s1(d)
s2(d)
Absolute speed
d1 2
d2 2
Size of the problem

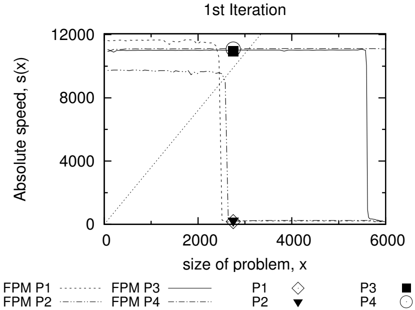

Models Updated
$s_1(d)$
$s_2(d)$
Absolute speed
$s_1'(d)$
$s_2'(d)$
Size of the problem


Models Updated
$s_1(d)$
$s_2(d)$
Absolute speed
$s_1'(d)$
$s_2'(d)$
Size of the problem

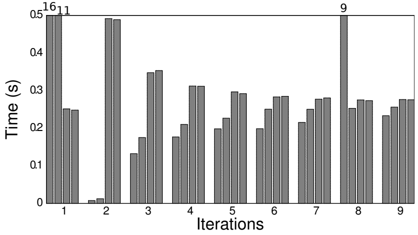
Layout

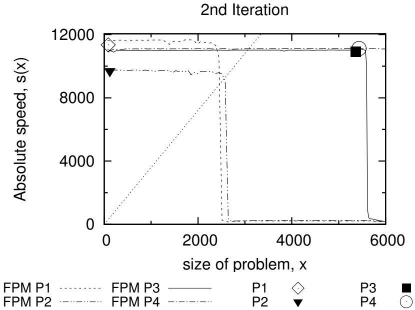
Experimental Results



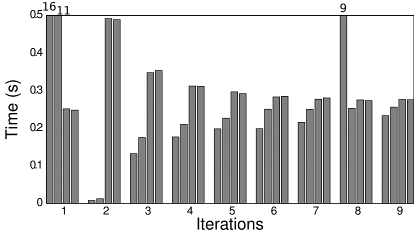


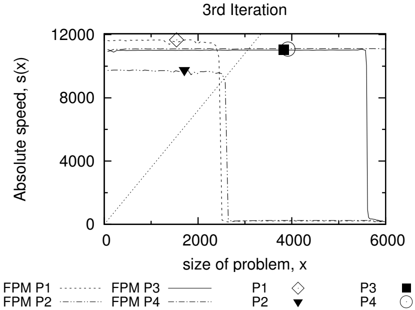
Experimental Results



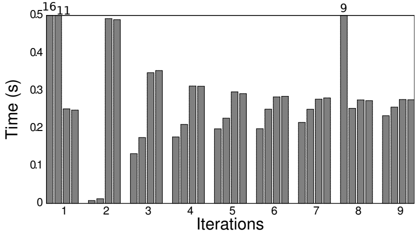


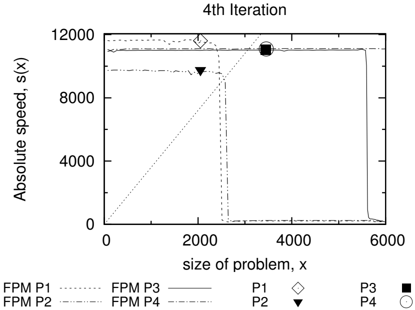
Experimental Results



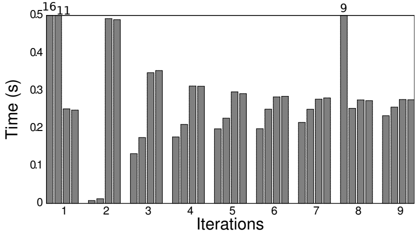


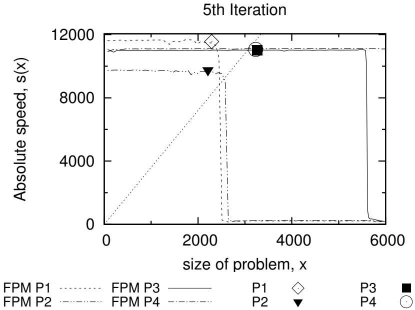
Experimental Results



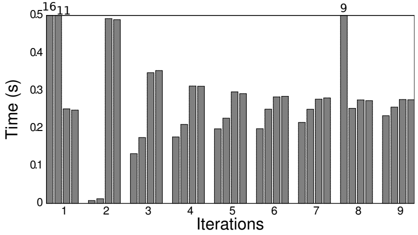


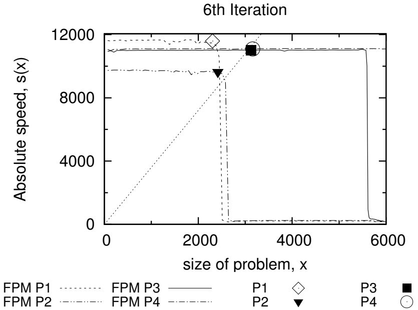
Experimental Results



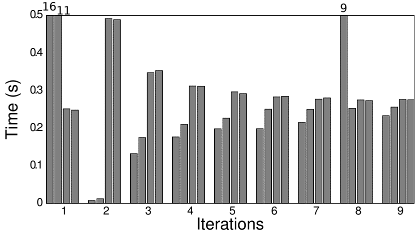


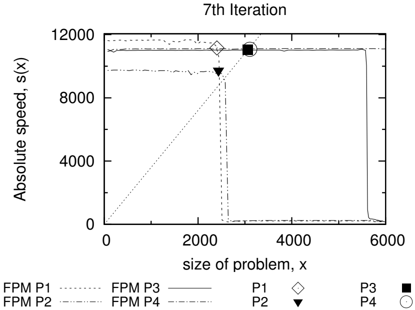
Experimental Results



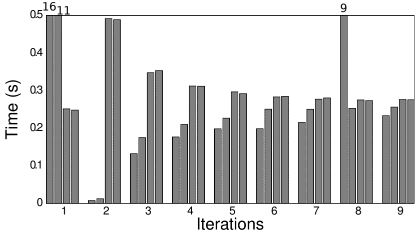


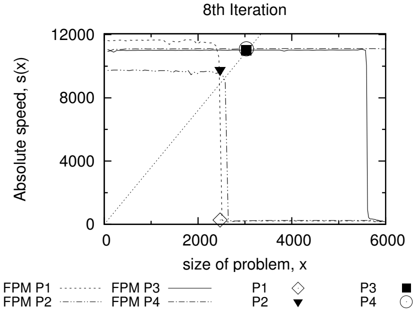
Experimental Results



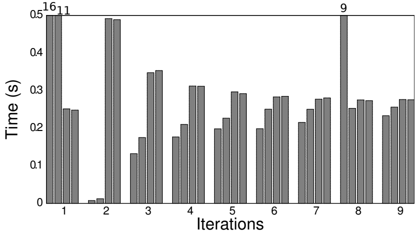


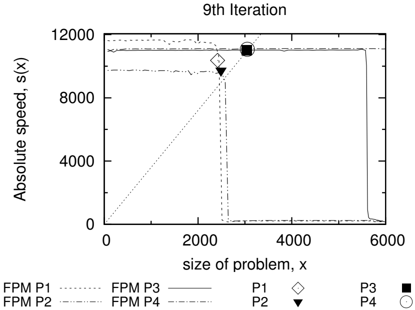
Experimental Results





Experimental Results





Conclusions

- Traditional algorithms only work for problems which fit into the main memory of all processors.

Conclusions

- Traditional algorithms only work for problems which fit into the main memory of all processors.
- Our algorithm, based on functional performance models, can balance for all problem sizes.

Conclusions

- Traditional algorithms only work for problems which fit into the main memory of all processors.
- Our algorithm, based on functional performance models, can balance for all problem sizes.
- No prior information about the heterogeneity and memory hierarchy of the platform needed as inputs into the algorithm.

Conclusions

- Traditional algorithms only work for problems which fit into the main memory of all processors.
- Our algorithm, based on functional performance models, can balance for all problem sizes.
- No prior information about the heterogeneity and memory hierarchy of the platform needed as inputs into the algorithm.
- Can be deployed self adaptively on any dedicated platform.

Questions?