FuPerMod: a Framework for Optimal Data Partitioning for Parallel Scientific Applications on Dedicated Heterogeneous HPC Platforms

David Clarke, Ziming Zhong, Vladimir Rychkov, Alexey Lastovetsky

Heterogeneous Computing Laboratory
University College Dublin, Ireland

Parallel Computing Technologies
St. Petersburg, Russia
30 September - 4 October, 2013

INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa


Introduction

- Modern HPC platform = system of heterogeneous devices and links
- How to execute data-parallel applications efficiently?
  - balance the load between processors
  - optimize communications
- **Data partitioning** = load balancing for data-parallel applications
  - performance models of processors

Introduction

- Modern HPC platform = system of heterogeneous devices and links
- How to execute data-parallel applications efficiently?
  - balance the load between processors
  - optimize communications
- **Data partitioning** = load balancing for data-parallel applications
  - performance models of processors
- **Data-parallel scientific applications:** sparse matrices and meshes
  - sparse matrices and meshes can be represented as graphs
  - graph partitioning = optimization of comm. & load balancing
- **Graph partitioning software**
  - ParMETIS, SCOTCH, JOSTLE
    minimize the communication cost with account for heterogeneity
  - Zoltan, PaGrid
    minimize the execution time using a cost function

Introduction

- **Computational heterogeneity in graph partitioning software**
  weights = constants representing the relative speeds of processors
  - must be provided as input - how to find them?
  - const speed is not accurate, especially for dynamic partitioning, self-adaptive applications, hybrid platforms

Introduction

- **Computational heterogeneity in graph partitioning software**
  weights = constants representing the relative speeds of processors
  - must be provided as input - how to find them?
  - const speed is not accurate, especially for dynamic partitioning, self-adaptive applications, hybrid platforms
- **Solution:** a general-purpose framework for data partitioning based on accurate and efficient performance modeling

  benchmarks $\rightarrow$ *models* $\rightarrow$ *partitioning algorithm* $\rightarrow$ *partitioning (weights)*
- **Main challenges**
  - performance modeling of interconnected devices and optimized kernels
  - complexity of model-based data partitioning algorithms

Introduction

FuPerMod framework

- Support of wide range of platforms and applications
- Static data partitioning

  exhaustive benchmarks $\rightarrow$ *detailed models* $\rightarrow$ *static algorithm* $\rightarrow$ *partitioning*
- Dynamic data partitioning and load balancing

  $\{$*benchmark* $\rightarrow$ *approx. models* $\rightarrow$ *dynamic algorithm* $\rightarrow$ *partitioning*$\}+$

In this talk

- Use case scenarios
  - Static data partitioning on a hybrid node
  - Dynamic load balancing on a heterogeneous cluster
- FuPerMod programming interface

Outline

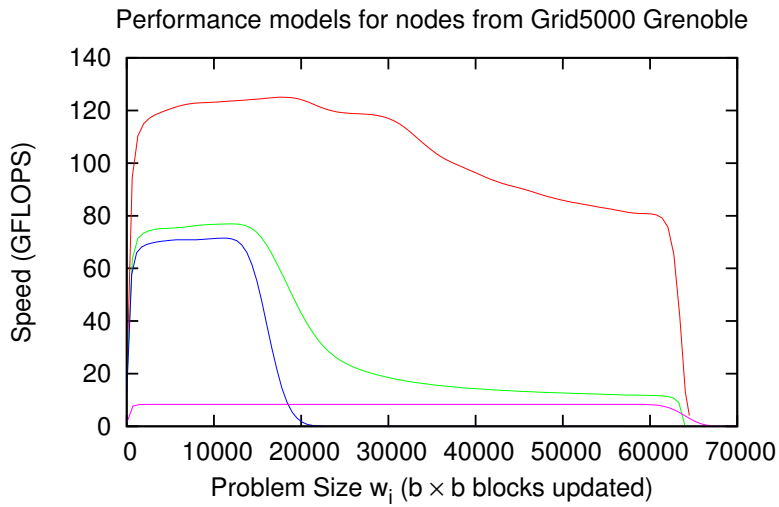
Data partitioning on heterogeneous platforms

Traditionally, performance is defined by Constant Performance Model (CPM):

- Computed from clock speed or by performing a benchmark
- Computations partitioned proportionally to the speed of processors
- Efficient but simplistic: algorithms may fail to balance the load [1]

Functional Performance Model (FPM):

- Speed as a function of problem size [2]: $s(x) = x/t(x)$
- Realistic performance of hardware and application
- Complex data partitioning algorithms (geometrical, numerical) [3]



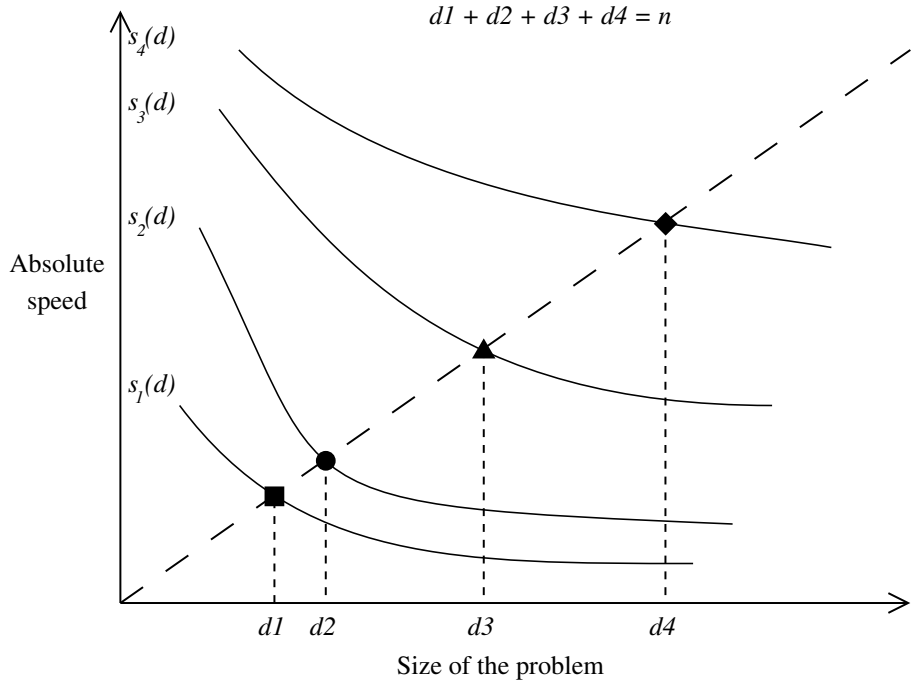
[1] D. Clarke et al: Dynamic load balancing of parallel iterative routines on platforms with memory heterogeneity (2010)
[2] A. Lastovetsky et al: Data partitioning with a functional performance model of heterogeneous processors (2007)
[3] V. Rychkov et al: Using multidimensional solvers for optimal data partitioning on dedicated heterogeneous platforms (2011)

Data partitioning with functional performance models

Load is balanced when:

all processors complete work within the same time:

$$\begin{cases} t_1(d_1) \approx t_2(d_2) \approx \ldots \approx t_p(d_p) \\ d_1 + d_2 + \ldots + d_p = N \end{cases}$$



solution lies on a line passing through the origin:

$$d_i / s_i(d_i) = const$$

- Originally designed for heterogeneous uniprocessor clusters
- Used for static and dynamic data partitioning
- Extended for hybrid multicore multi-GPU nodes [1] and clusters [2]

[1] Z. Zhong et al: Data Partitioning on Heterogeneous Multicore and Multi-GPU Systems Using Functional Performance Models of Data-Parallel Applications (2012)
[2] D. Clarke et al: Hierarchical Partitioning Algorithm for Scientific Computing on Highly Heterogeneous CPU + GPU Clusters (2012)

Outline

Static data partitioning on a hybrid node

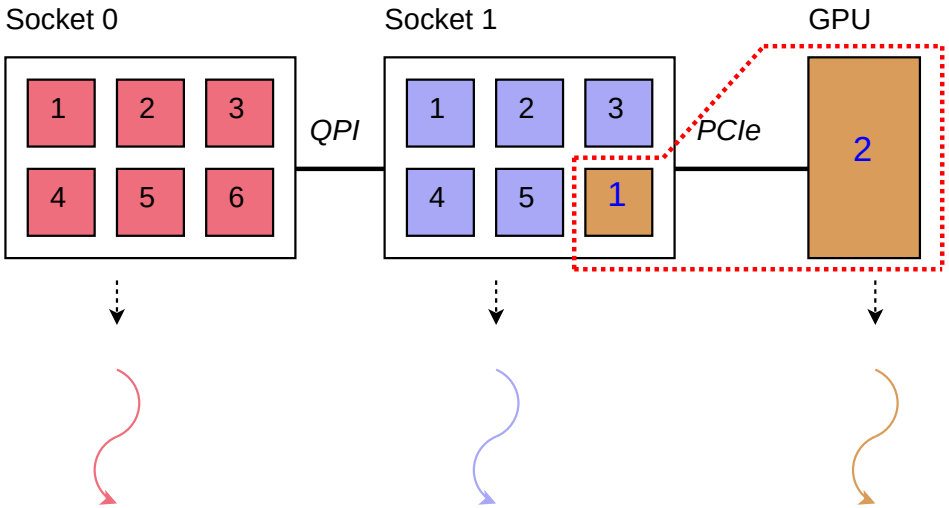
Hardware	CPU (AMD)	GPUs (NVIDIA)	
Architecture	Opteron 8439SE	GF GTX680	Tesla C870
Core Clock	2.8 GHz	1006 MHz	600 MHz
Number of Cores	4×6 cores	1536 cores	128 cores
Memory Size	4×16 GB	2048 MB	1536 MB
Memory Bandwidth	NUMA	192.3 GB/s	76.8 GB/s
Software	multi-threaded	CUDA + out-of-core	
Processes	1 per socket		1 per device

exhaustive benchmarks $\rightarrow$ *detailed models* $\rightarrow$ *static algorithm* $\rightarrow$ *partitioning*

Configuration of data-parallel application

- No idle compute devices
  - may not be the optimal configuration (out of scope of this study)
- Even load of identical compute devices
  - no evidence that uneven load will improve performance
- One-to-one mapping of processes/threads to compute devices
  - no evidence that many-to-one will improve performance
- Same one-to-one mapping for all runs of the program
  - mapping is not delegated to the operating environment

Configuration of data-parallel application

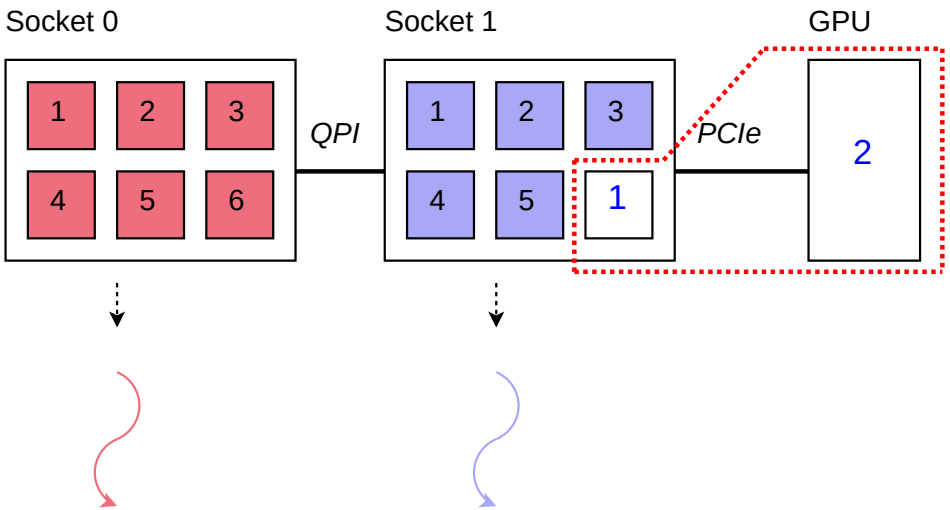


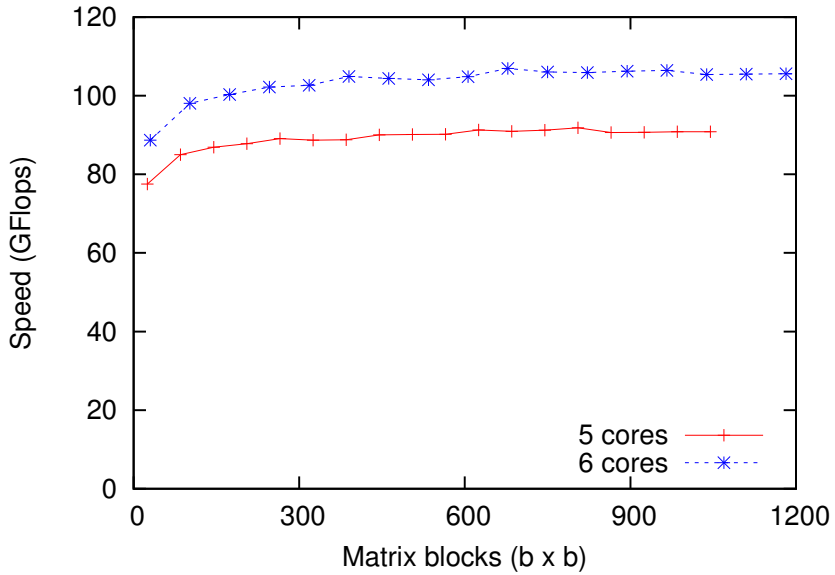
Relatively independent groups of devices: : 6 cores, 5 cores, 1 core + GPU
each group = abstract processor (uni- or multi-processor)

- Cores in a group interfere with each other due to resource contention
- All cores in a group execute the same amount of workload in parallel
- GPU kernel computation time and data transfer time are both included
- Host core for GPU is chosen to maximize data throughput between GPU and NUMA memory

Computation performance models of multicore

$S_c(x)$: speed of c cores that execute a multi-threaded kernel and share system resources, x units distributed between cores

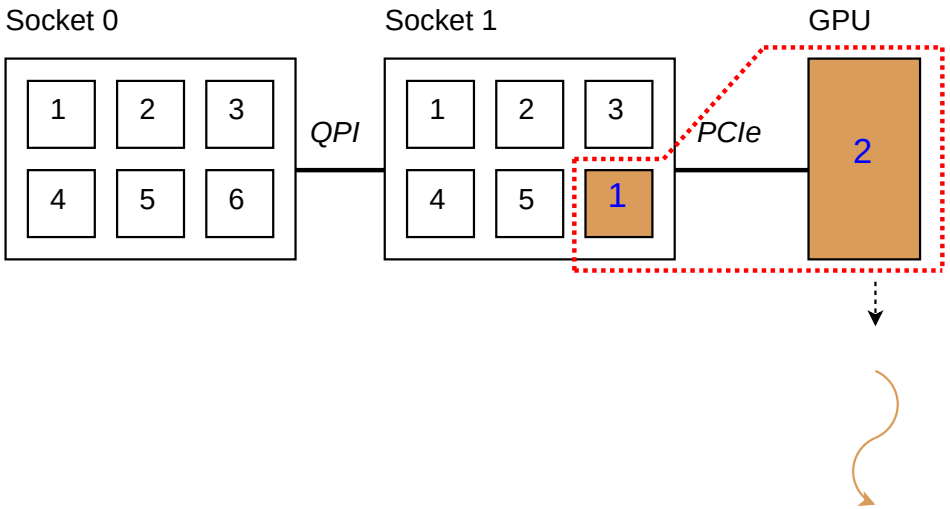


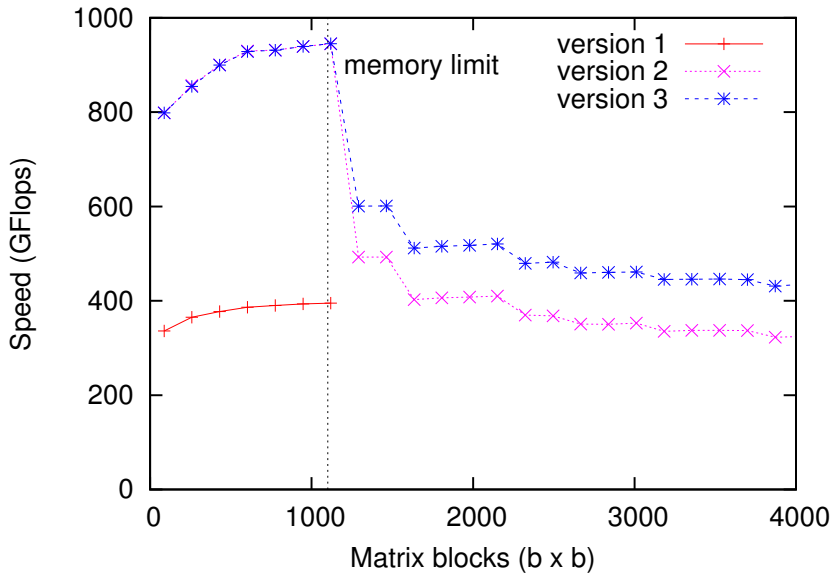


- $S_5(x)$: 5-threaded kernel on a socket, 1 core idle
- $S_6(x)$: 6-threaded kernel on a socket

Computation performance models of GPU

$g(x)$: combined speed of a GPU and its dedicated core, including data transfers



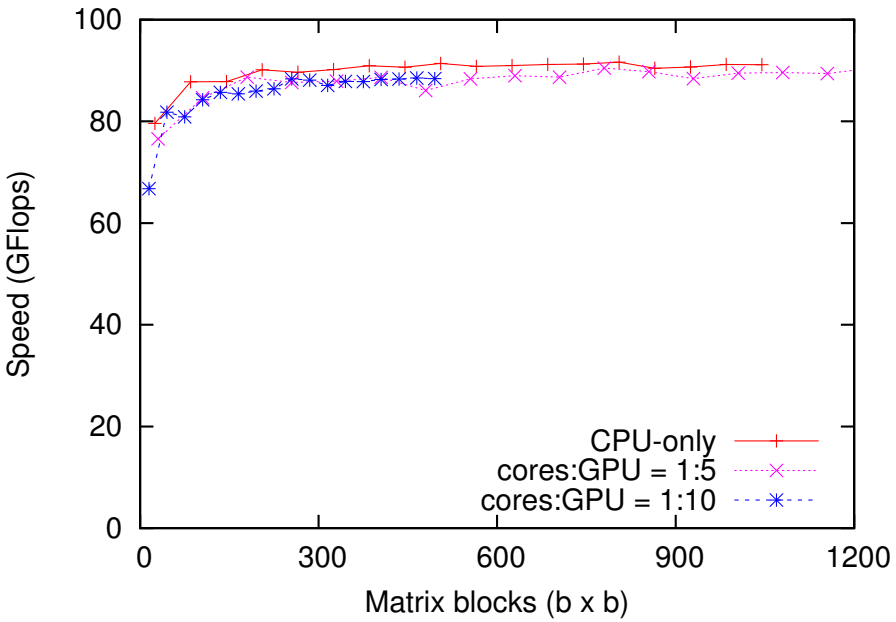


- $g(x)$ (version 1): native CUDA kernel
- $g(x)$ (version 2): accumulate intermediate results + out-of-core
- $g(x)$ (version 3): version 2 + overlap data transfers and kernel executions

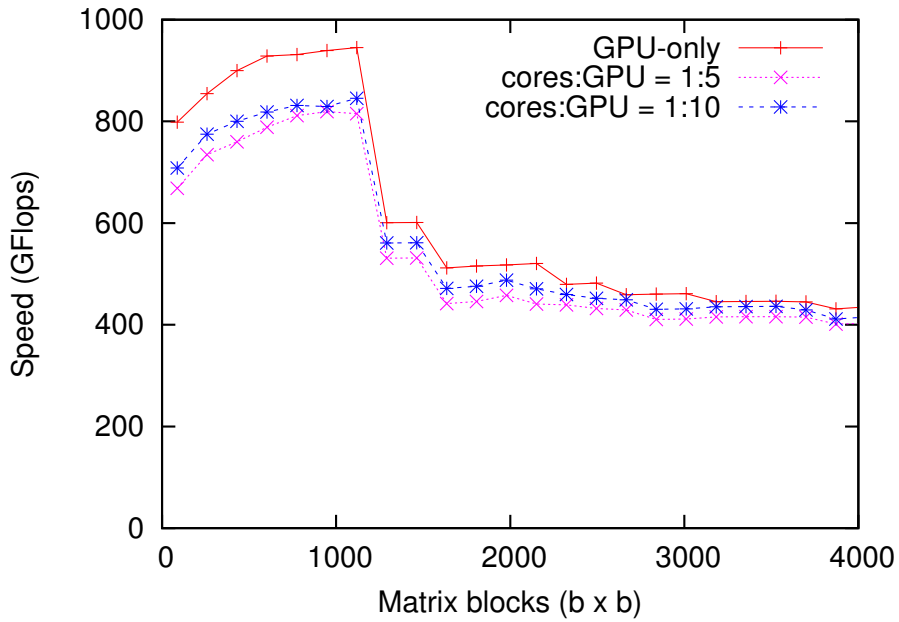
Impact of Resource Contention to Performance Modeling

- CPU and GPU kernel benchmarking simultaneously in a socket
- FPM of multiple cores $S_5(x)$ is barely affected
- FPM of GPU $g(x)$ gets 85% accuracy (speed drops by 7 - 15%)

$s_5(x)$, speed of multiple cores



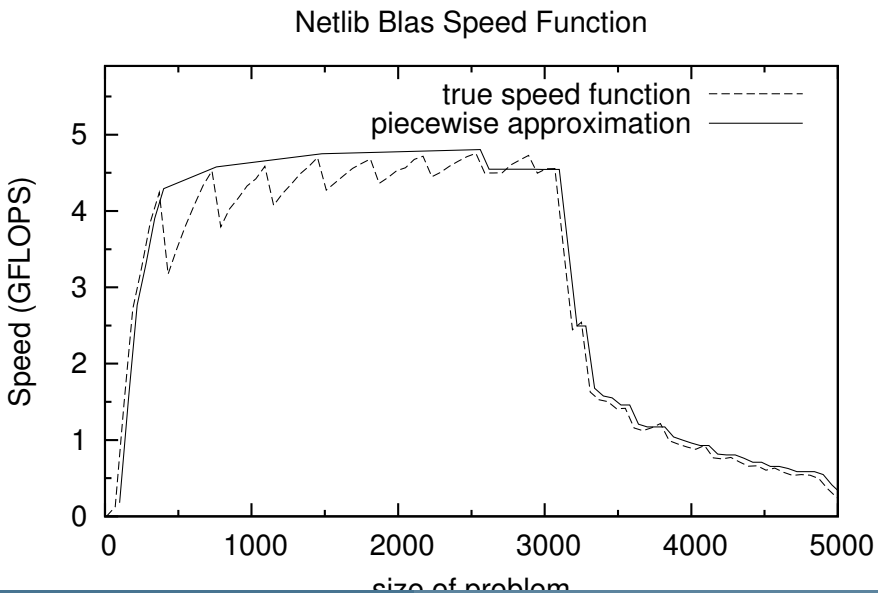
$g(x)$, speed of a GPU

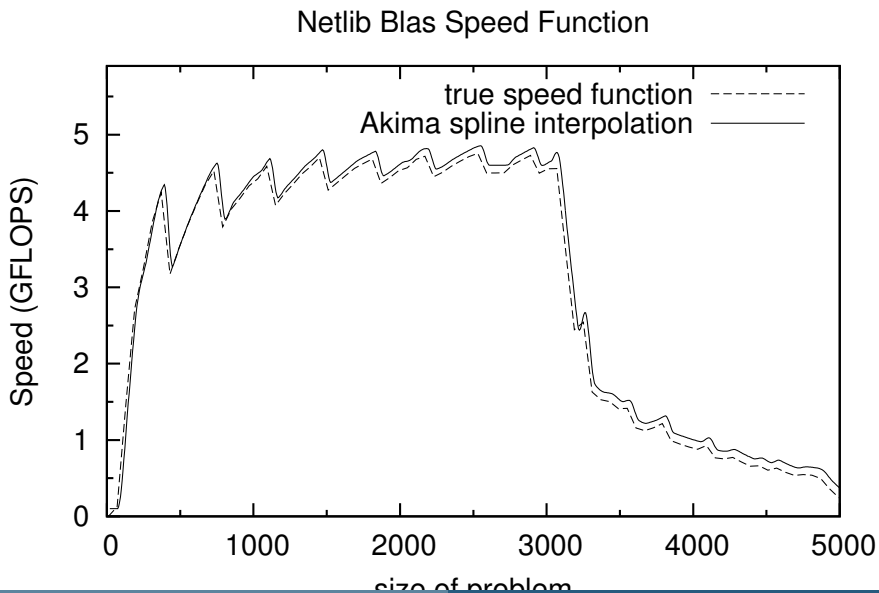


FuPerMod: implemented models

- FPM based on the piecewise linear interpolation of the time function
- FPM based on the Akima spline interpolation of the time function
- CPM (requires only one experimental point)

```
// X = piecewise, interp, constant
fupermod_model* model =
  fupermod_model_X_alloc(int count, fupermod_point* points);
int d = 1000;
double t =  model->t(model, d);
double s = d / model->t(model, d);
```





API for computation performance modeling

Computation performance model

```
struct fupermod_model {
  int count;                // number of data points
  fupermod_point* points; // data points
  // approximation of the time function
  double (*t)(fupermod_model* model, double x);
  // add a point and update the approximation
  int (*update)(fupermod_model* model, fupermod_point point);
};
```

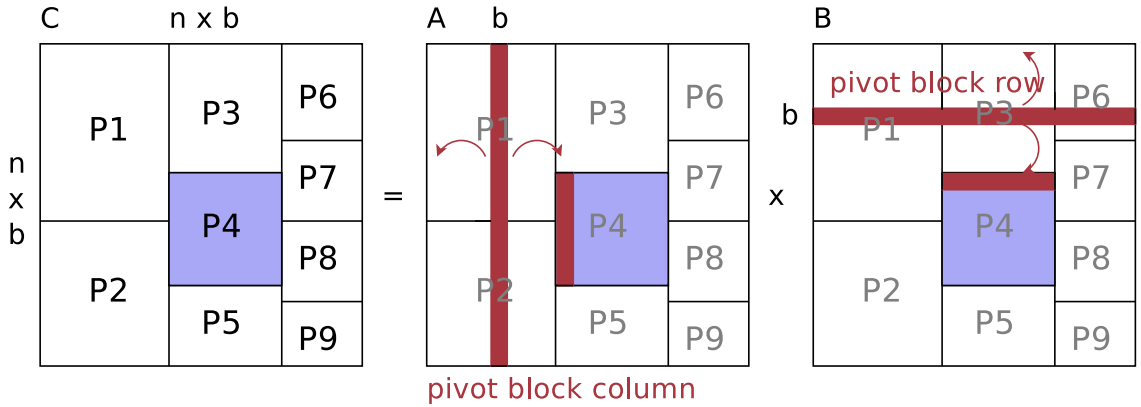
Data point

```
struct fupermod_point {
  int d;     // problem size
  double t; // execution time
  int reps; // actual repetitions
  double ci;// confidence interval
};
```

How to obtain data points? $\longrightarrow$ profiling, benchmarking

Computation performance measurement

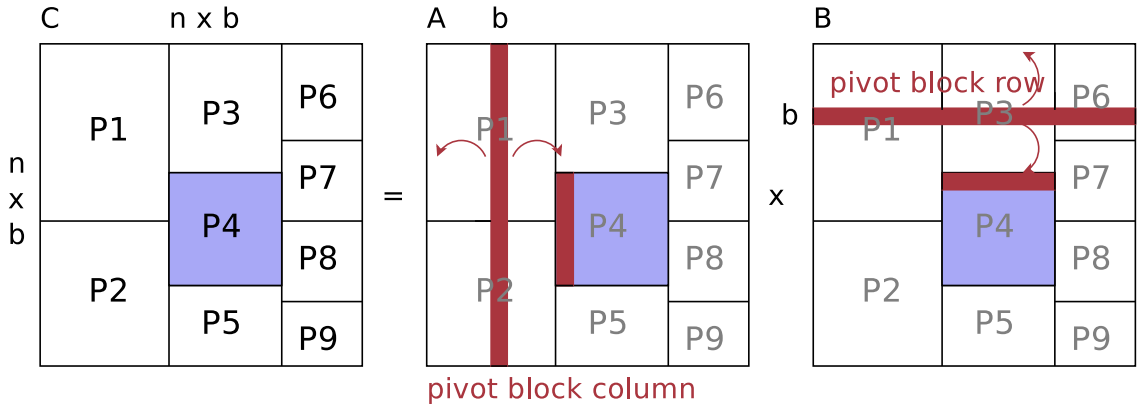
- Parallel matrix multiply: heterogeneous algorithm with minimized volume of communications*
  based on SUMMA



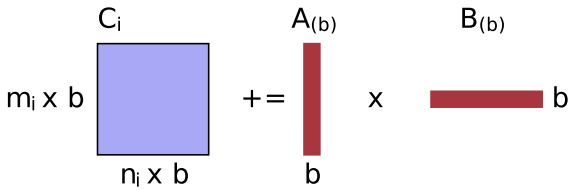
* O. Beaumont et al: Matrix Multiplication on Heterogeneous Platforms (2001)

Computation performance measurement

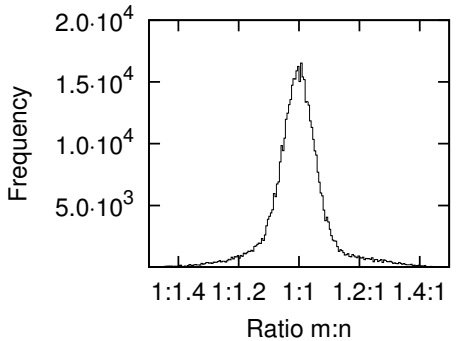
- Parallel matrix multiply: heterogeneous algorithm with minimized volume of communications*
  based on SUMMA

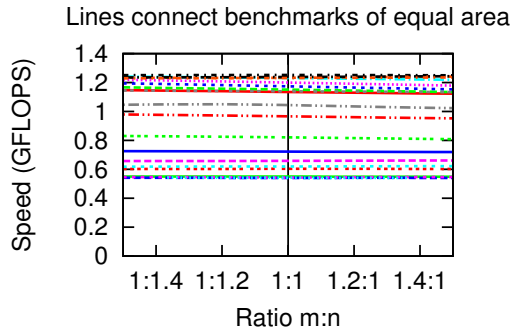


- Computation kernel: panel update
  representative for the entire application



- Speed: function of area
  can be measured for square matrices





* O. Beaumont et al: Matrix Multiplication on Heterogeneous Platforms (2001)

Computation kernel

```
// allocate blocks of matrix, pivot row and column
int initialize(int d, void* params) {
  int m = n = sqrt(d);
  int b = params->b;
  params->C  = malloc(m * b * n * b);
  params->WA = malloc(m * b * b);
  params->WB = malloc(n * b * b);
}

// characteristic computation
// dgemm = optimized implementation from ACML (AMD) or CUBLAS (NVIDIA)
int execute(pthread_mutex_t* mutex, void* params) {
  dgemm(params->WA, params->WB, params->C);
}

// deallocate data
int finalize(void* params) {
  free(params->C);
  free(params->WA);
  free(params->WB);
}
```

API for computation performance measurement

Computation kernel

```
// data allocation/deallocation excluded from measurement
// params - application-specific parameters
// mutex - to terminate benchmarking if it takes too long
struct fupermod_kernel {
  int (*initialize)(int d, void* params);
  int (*execute)(pthread_mutex_t* mutex, void* params);
  int (*finalize)(void* params);
};
```

Benchmark for computation kernel

```
int fupermod_benchmark(
  // kernel and problem size
  fupermod_kernel* kernel, int d,
  // statistical parameters
  fupermod_precision precision,
  // to sync some procs if needed
  MPI_Comm comm_sync,
  // returns a data point
  fupermod_point* point
);
```

$\{benchmark(kernel, problem\ size) \rightarrow point\}+ \rightarrow model$

FuPerMod: implemented static partitioning algorithms

- geometrical algorithm based on the piecewise-linear FPMs
- numerical algorithm based on the Akima-spline FPMs
- basic algorithm based on CPMs

```
// X = geometric, multiroot, basic
int fupermod_partition_X(
  int size, fupermod_model** models, fupermod_dist* dist);
```

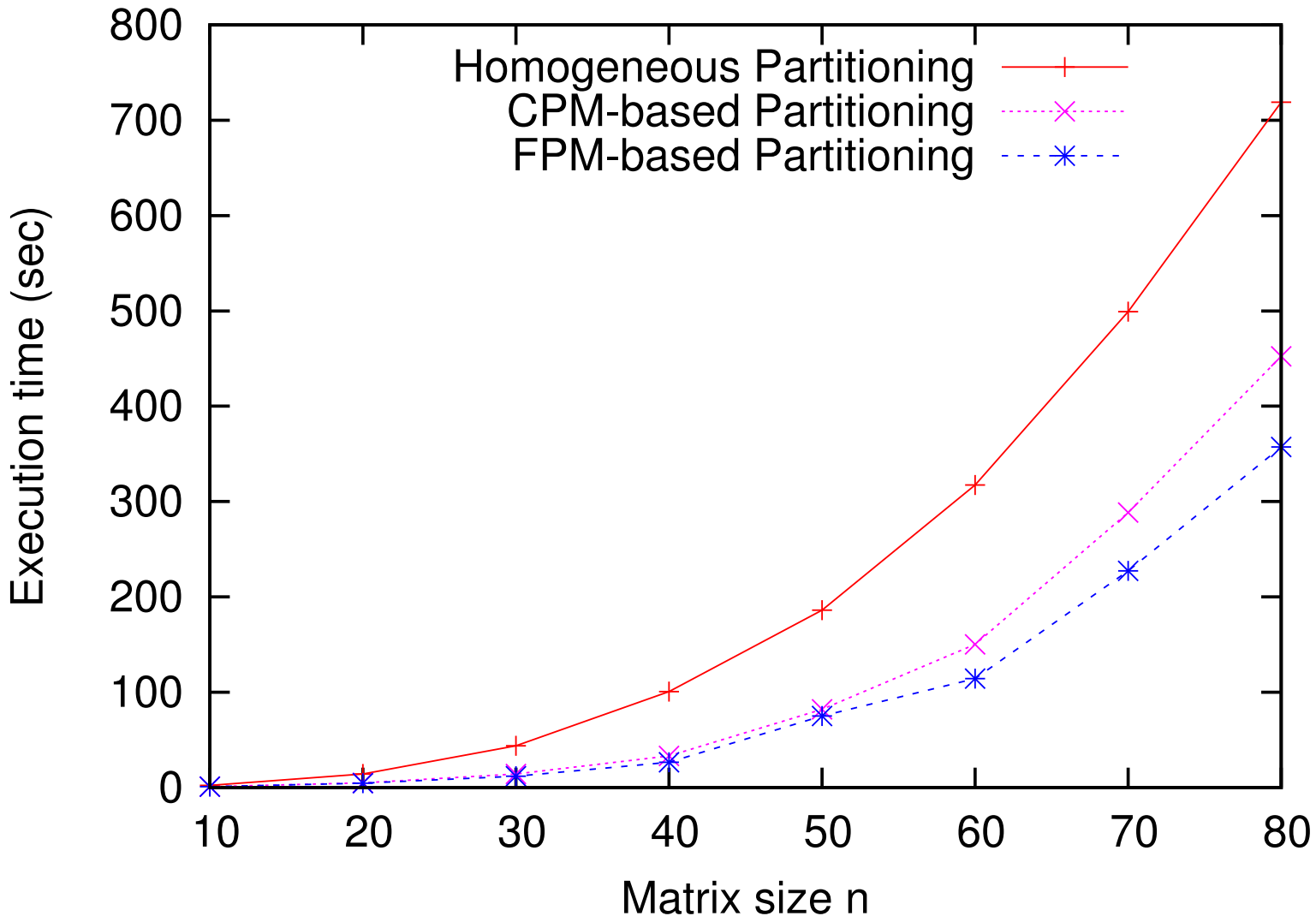
Partitioning

```
struct fupermod_dist {
  int D;     // total problem size
  int size; // number of procs
  fupermod_part* parts;
};
```

```
struct fupermod_part {
  int d;     // data chunk
  double t; // execution time
};
```

exhaustive benchmarks $\rightarrow$ *detailed models* $\rightarrow$ *static algorithm* $\rightarrow$ *partitioning*

Static data partitioning: experiments



Outline

Dynamic load balancing on a heterogeneous cluster

Iterative routine (Jacobi method): $x^{k+1} = f(x^k),\ k = 0, 1, \ldots$

```
MPI_Comm comm;
// Jacobi data: partitioned matrix/vectors
double *A = ..., *b = ..., *x = ...;
// main loop
while (error > eps) {
  // iteration of Jacobi method
  jacobi_iterate(comm, A, b, x, &error);
}
```

Dynamic load balancing on a heterogeneous cluster

Iterative routine (Jacobi method): $x^{k+1} = f(x^k),\ k = 0, 1, \ldots$

```
MPI_Comm comm;
// Jacobi data: partitioned matrix/vectors
double *A = ..., *b = ..., *x = ...;
// main loop
while (error > eps) {
  // redistribution of Jacobi data
  jacobi_redistribute(comm, A, b, x, end - start);
  // iteration of Jacobi method
  gettimeofday(&start);
  jacobi_iterate(comm, A, b, x, &error);
  gettimeofday(&end);
}
```

Dynamic load balancing on a heterogeneous cluster

Iterative routine (Jacobi method): $x^{k+1} = f(x^k),\ k = 0, 1, \ldots$

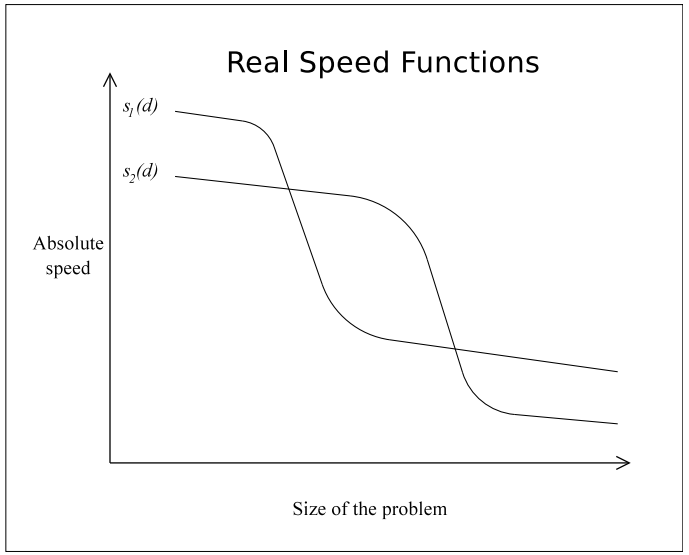
```
MPI_Comm comm;
// Jacobi data: partitioned matrix/vectors
double *A = ..., *b = ..., *x = ...;
// main loop
while (error > eps) {
  // redistribution of Jacobi data
  jacobi_redistribute(comm, A, b, x, end - start);
  // iteration of Jacobi method
  gettimeofday(&start);
  jacobi_iterate(comm, A, b, x, &error);
  gettimeofday(&end);
}
```

$\{benchmark \rightarrow approx.\ models \rightarrow dynamic\ algorithm \rightarrow partitioning\}+$

Dynamic FPM-based data partitioning

Functional performance models can be built:

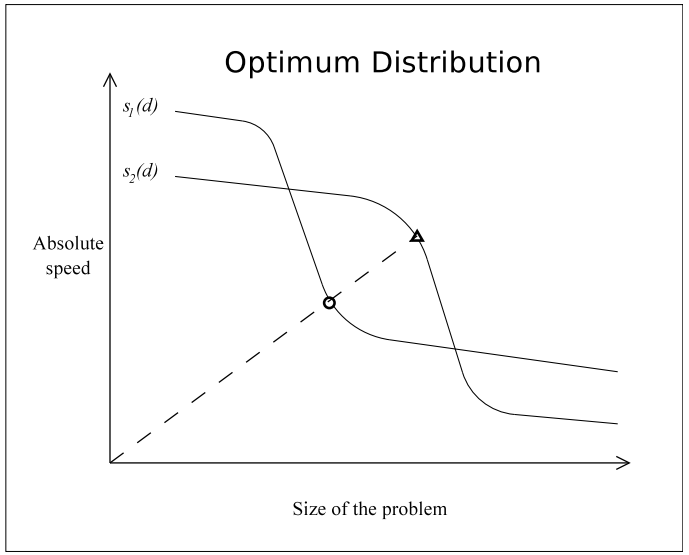
- exhaustively in advance
- dynamically at run time



Dynamic FPM-based data partitioning

Functional performance models can be built:

- exhaustively in advance
- dynamically at run time



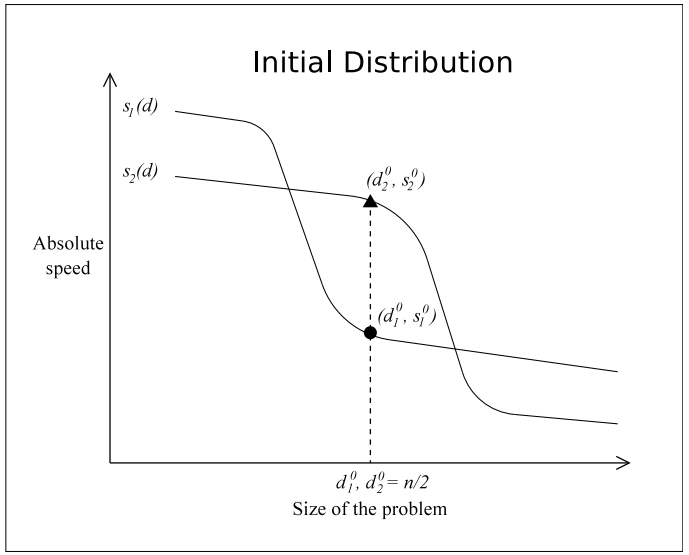
Dynamic FPM-based data partitioning

Functional performance models can be built:

- exhaustively in advance
- dynamically at run time

Initial: point $(n/p, s_i^0)$ with speed $s_i^0 = \dfrac{n/p}{t_i(n/p)}$

first function approximation $s_i'(x) \equiv s_i^0$



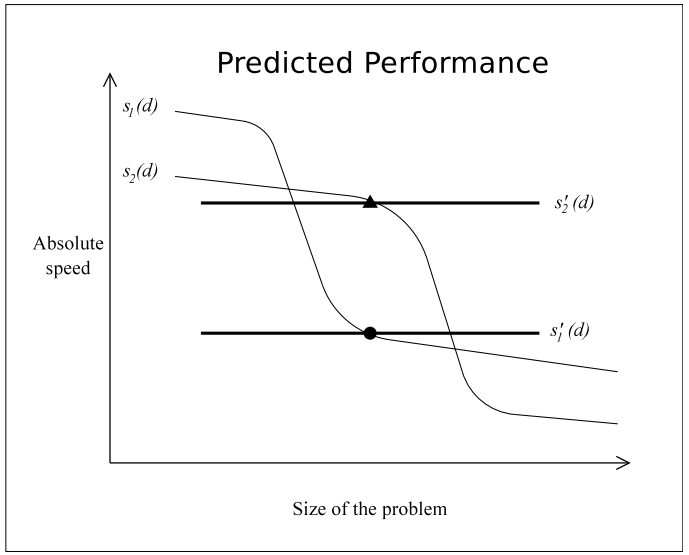
Dynamic FPM-based data partitioning

Functional performance models can be built:

- exhaustively in advance
- dynamically at run time

Initial: point $(n/p, s_i^0)$ with speed $s_i^0 = \dfrac{n/p}{t_i(n/p)}$

first function approximation $s_i'(x) \equiv s_i^0$



Dynamic FPM-based data partitioning

Functional performance models can be built:

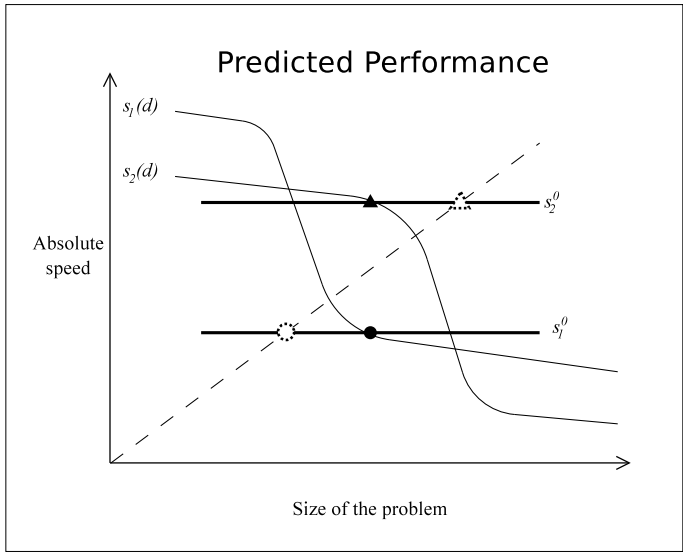
- exhaustively in advance
- dynamically at run time

Initial: point $(n/p, s_i^0)$ with speed $s_i^0 = \dfrac{n/p}{t_i(n/p)}$

first function approximation $s_i'(x) \equiv s_i^0$

Iterations: point (d_i^k, s_i^k) with speed $s_i^k = \dfrac{d_i^k}{t_i(d_i^k)}$

approximation $s_i'(x)$ updated by adding the point



Dynamic FPM-based data partitioning

Functional performance models can be built:

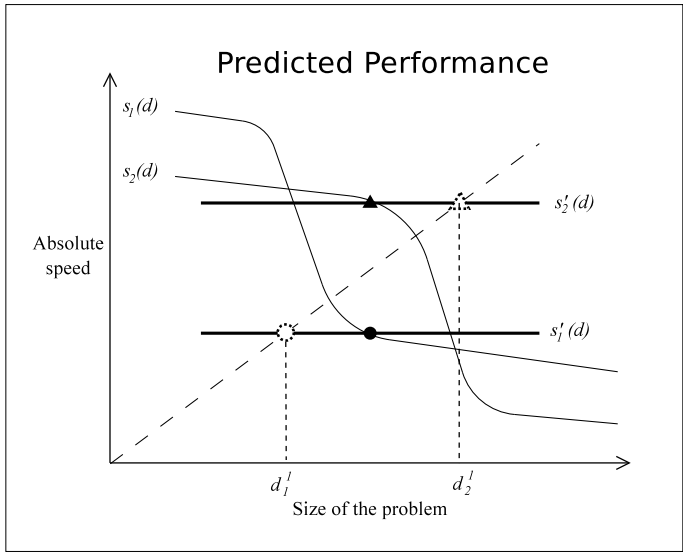
- exhaustively in advance
- dynamically at run time

Initial: point $(n/p, s_i^0)$ with speed $s_i^0 = \dfrac{n/p}{t_i(n/p)}$

first function approximation $s_i'(x) \equiv s_i^0$

Iterations: point (d_i^k, s_i^k) with speed $s_i^k = \dfrac{d_i^k}{t_i(d_i^k)}$

approximation $s_i'(x)$ updated by adding the point



Dynamic FPM-based data partitioning

Functional performance models can be built:

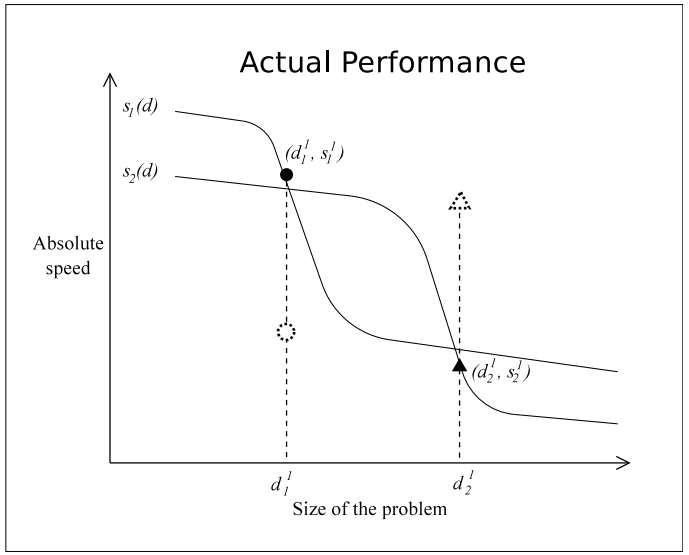
- exhaustively in advance
- dynamically at run time

Initial: point $(n/p, s_i^0)$ with speed $s_i^0 = \dfrac{n/p}{t_i(n/p)}$

first function approximation $s_i'(x) \equiv s_i^0$

Iterations: point (d_i^k, s_i^k) with speed $s_i^k = \dfrac{d_i^k}{t_i(d_i^k)}$

approximation $s_i'(x)$ updated by adding the point



Dynamic FPM-based data partitioning

Functional performance models can be built:

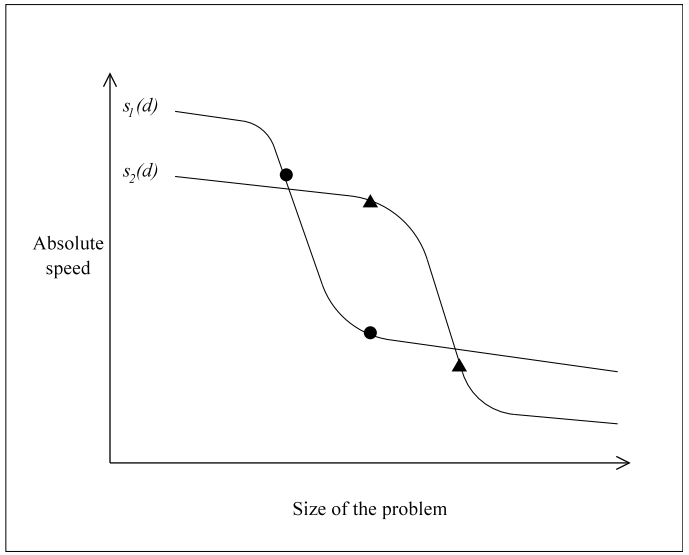
- exhaustively in advance
- dynamically at run time

Initial: point $(n/p, s_i^0)$ with speed $s_i^0 = \dfrac{n/p}{t_i(n/p)}$

first function approximation $s_i'(x) \equiv s_i^0$

Iterations: point (d_i^k, s_i^k) with speed $s_i^k = \dfrac{d_i^k}{t_i(d_i^k)}$

approximation $s_i'(x)$ updated by adding the point



Dynamic FPM-based data partitioning

Functional performance models can be built:

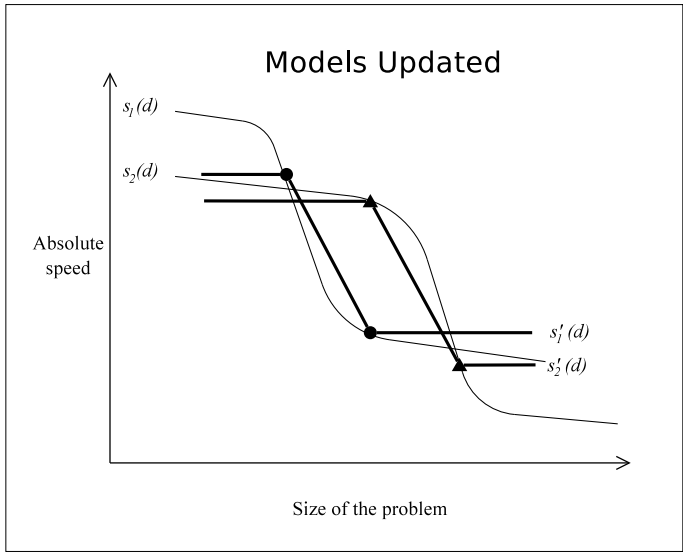
- exhaustively in advance
- dynamically at run time

Initial: point $(n/p, s_i^0)$ with speed $s_i^0 = \dfrac{n/p}{t_i(n/p)}$

first function approximation $s_i'(x) \equiv s_i^0$

Iterations: point (d_i^k, s_i^k) with speed $s_i^k = \dfrac{d_i^k}{t_i(d_i^k)}$

approximation $s_i'(x)$ updated by adding the point



Dynamic FPM-based data partitioning

Functional performance models can be built:

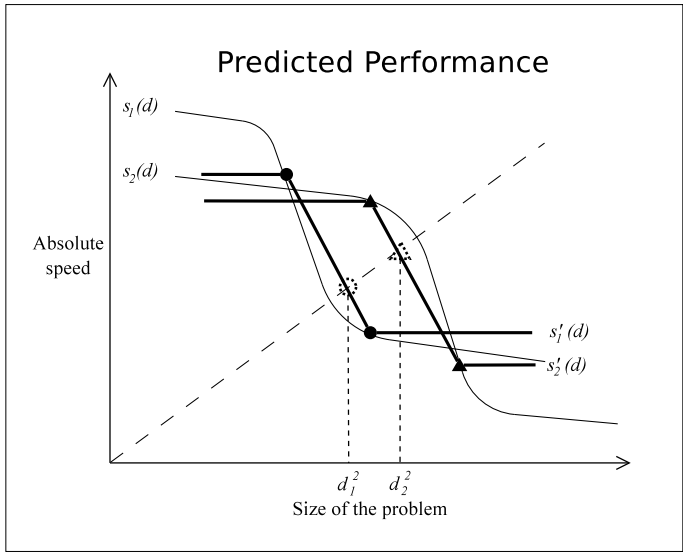
- exhaustively in advance
- dynamically at run time

Initial: point $(n/p, s_i^0)$ with speed $s_i^0 = \dfrac{n/p}{t_i(n/p)}$

first function approximation $s_i'(x) \equiv s_i^0$

Iterations: point (d_i^k, s_i^k) with speed $s_i^k = \dfrac{d_i^k}{t_i(d_i^k)}$

approximation $s_i'(x)$ updated by adding the point



Dynamic FPM-based data partitioning

Functional performance models can be built:

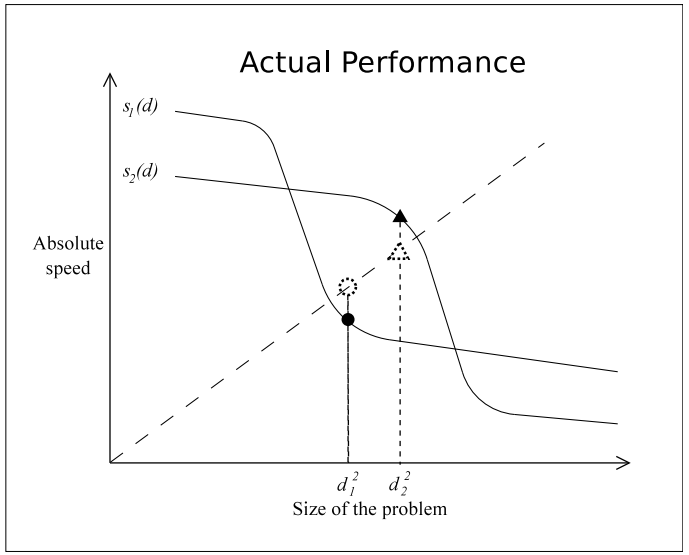
- exhaustively in advance
- dynamically at run time

Initial: point $(n/p, s_i^0)$ with speed $s_i^0 = \dfrac{n/p}{t_i(n/p)}$

first function approximation $s_i'(x) \equiv s_i^0$

Iterations: point (d_i^k, s_i^k) with speed $s_i^k = \dfrac{d_i^k}{t_i(d_i^k)}$

approximation $s_i'(x)$ updated by adding the point



Dynamic FPM-based data partitioning

Functional performance models can be built:

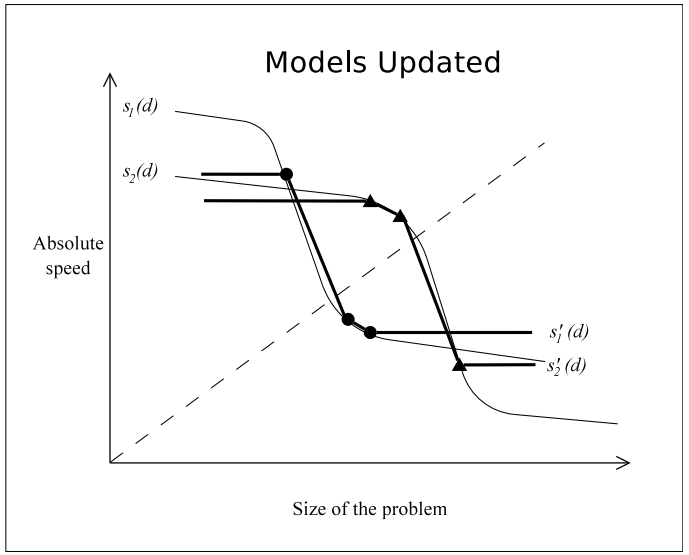
- exhaustively in advance
- dynamically at run time

Initial: point $(n/p, s_i^0)$ with speed $s_i^0 = \dfrac{n/p}{t_i(n/p)}$

first function approximation $s_i'(x) \equiv s_i^0$

Iterations: point (d_i^k, s_i^k) with speed $s_i^k = \dfrac{d_i^k}{t_i(d_i^k)}$

approximation $s_i'(x)$ updated by adding the point



Dynamic FPM-based data partitioning

Functional performance models can be built:

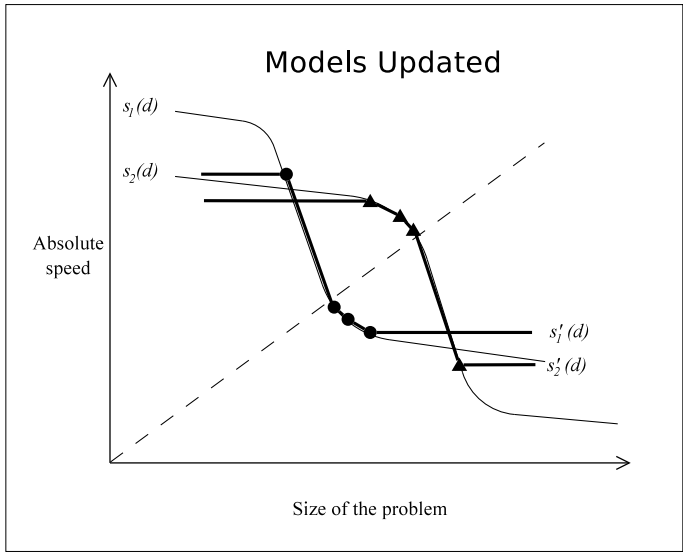
- exhaustively in advance
- dynamically at run time

Initial: point $(n/p, s_i^0)$ with speed $s_i^0 = \dfrac{n/p}{t_i(n/p)}$

first function approximation $s_i'(x) \equiv s_i^0$

Iterations: point (d_i^k, s_i^k) with speed $s_i^k = \dfrac{d_i^k}{t_i(d_i^k)}$

approximation $s_i'(x)$ updated by adding the point



Dynamic load balancing: Jacobi method

```
MPI_Comm_size(comm, &size);
// piecewise linear FPMs
fupermod_model** models = malloc(sizeof(fupermod_model*) * size);
for (i = 0; i < size; i++)
  models[i] = fupermod_model_piecewise_alloc();
// current distribution, initially even
fupermod_dist* dist = fupermod_dist_alloc(D, size);
// context for dynamic load balancing
fupermod_dynamic balancer = { fupermod_partition_geometric,
  size, models, fupermod_dist_alloc(D, size) };
// Jacobi data: dist->parts[i].d rows/els allocated for matrix/vectors
double *A = ..., *b = ..., *x = ...;
// main loop
while (error > eps) {
  // redistribution of Jacobi data: dist --> balancer.dist
  jacobi_redistribute(comm, dist, A, b, x, balancer.dist);
  // store the current distribution
  fupermod_dist_copy(dist, balancer.dist);
  // iteration of Jacobi method
  gettimeofday(&start);
  jacobi_iterate(comm, dist, A, b, x, &error);
  // add new points (now-start) and balance the load
  fupermod_balance_iterate(&balancer, comm, start);
}
```

API for dynamic partitioning and load balancing

Context of the dynamic algorithms

```
struct fupermod_dynamic {
  fupermod_partition partition; // partitioning algorithm
  int size;                     // number of procs
  fupermod_model** models;      // current approximations
  fupermod_dist* dist;          // current partitioning
}
```

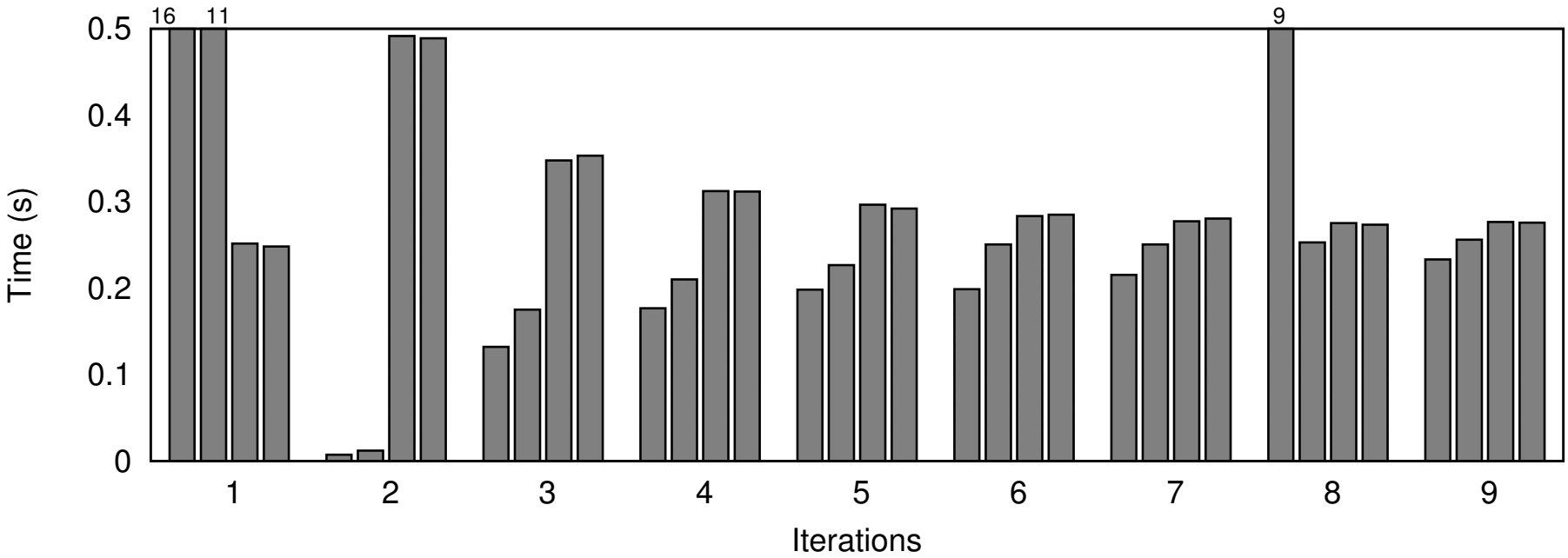
Dynamic data partitioning

```
// performs a benchmark
int fupermod_partition_iterate(fupermod_dynamic*, MPI_Comm comm,
  fupermod_precision precision, fupermod_benchmark* benchmark,
  double eps);
```

Dynamic load balancing

```
// requires the start time as input
int fupermod_balance_iterate(fupermod_dynamic*, MPI_Comm comm,
  struct timespec start);
```

Dynamic load balancing on a heterogeneous cluster



Conclusion

FuPerMod: general-purpose data partitioning framework

- accurate and cost-effective performance measurement
- construction of computation performance models, using different methods of interpolation of time and speed
- invocation of model-based data partitioning algorithms for static and dynamic load balancing

`http://hcl.ucd.ie/project/fupermod`

Thank You!

University College Dublin

Heterogeneous Computing Laboratory

Science Foundation Ireland

China Scholarship Council

Instituto de Engenharia de Sistemas e Computadores

Instituto Superior Tecnico Universidade de Lisboa

Complex HPC EU COST Action IC0805