Optimization of data parallel applications for heterogeneous and hierarchical HPC platforms based on multicores and multi-GPUs

Alexey Lastovetsky

Heterogeneous Computing Laboratory
University College Dublin, Ireland

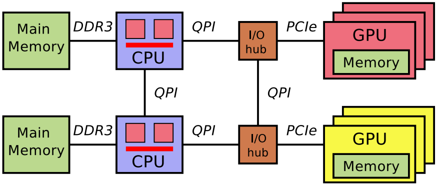
Parallel Processing and Applied Mathematics
Warsaw, Poland – September 8-11, 2013

Acknowledgment

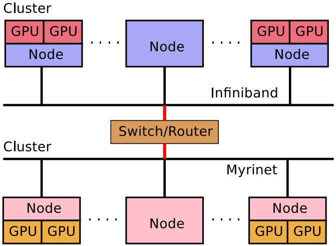
- David Clarke, UCD
- Aleksandar Ilic, TU Lisbon
- Vladimir Rychkov, UCD
- Leonel Sousa, TU Lisbon
- Ziming Zhong, UCD

Introduction

- Modern HPC platform = complex system of highly heterogeneous devices and links
- How to execute data parallel applications efficiently?



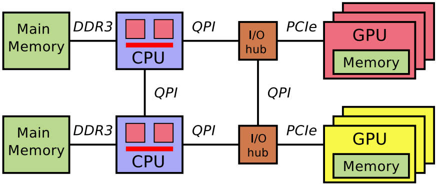
Hybrid Multicore & Multi-GPU Node



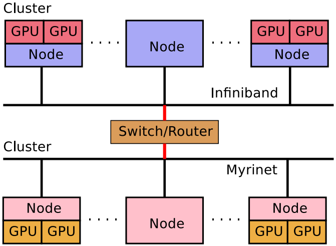
Interconnected Hybrid Clusters

Introduction

- Modern HPC platform = complex system of highly heterogeneous devices and links
- How to execute data parallel applications efficiently?
- Traditional heterogeneous clusters: balance the load of relatively independent processors and optimize communications
- Load balancing for data parallel applications = data partitioning



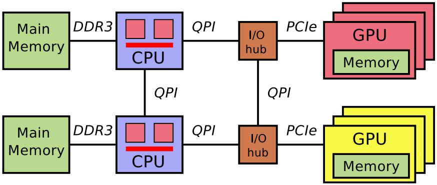
Hybrid Multicore & Multi-GPU Node



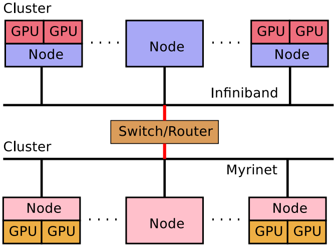
Interconnected Hybrid Clusters

Introduction

- Modern HPC platform = complex system of highly heterogeneous devices and links
- How to execute data parallel applications efficiently?
- Traditional heterogeneous clusters: balance the load of relatively independent processors and optimize communications
- Load balancing for data parallel applications = data partitioning
- How to apply data partitioning to multicore/multi-GPU? Compute devices are more tightly coupled (and less independent), as resources are shared between devices



Hybrid Multicore & Multi-GPU Node



Interconnected Hybrid Clusters

Introduction

Our target:

- Data parallel application
  - Divisible computational workload
  - Workload proportional to data size
- Dedicated hybrid system
- Reuse of optimized software stack

Introduction

Our target:

- Data parallel application
  - Divisible computational workload
  - Workload proportional to data size
- Dedicated hybrid system
- Reuse of optimized software stack

Our approach:

- Partitioning devices into independent groups
  - Each group = abstract processor
    - May be uni- or multi-processor depending on software kernel
- Accurate performance modeling of the abstract processors
- Model-based data partitioning between the heterogeneous abstract processors

Outline

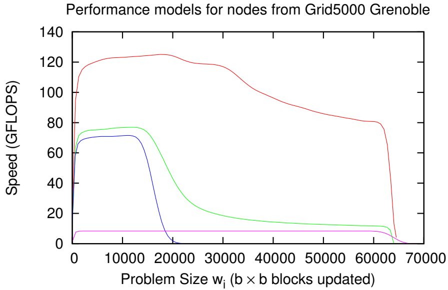
Data Partitioning on Heterogeneous Platform

Traditionally, performance is defined by a single constant number

- Constant Performance Model (CPM)
- Computed from clock speed or by performing a benchmark
- Computational units are partitioned as $d_i = N \times (s_i / \sum_{j=1}^{p} s_j)$
- Simplistic, algorithms may fail to converge to a balanced solution [1]

Functional Performance Model (FPM):

- Represent speed as a function of problem size [2]
- Realistic
- Application centric
- Hardware specific



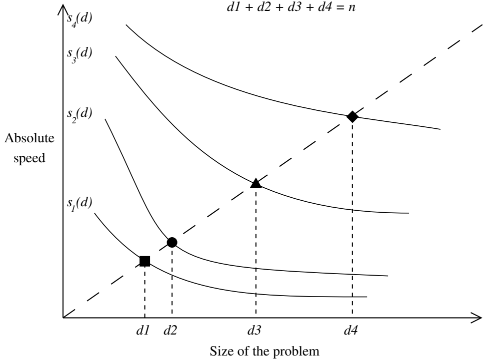
[1] D. Clarke et al: Dynamic Load Balancing of Parallel Iterative Routines on Platforms with Memory Heterogeneity, 2010
[2] A. Lastovetsky et al: Data partitioning with a functional performance model of heterogeneous processors, 2007.

Partitioning with functional performance models*

Load is balanced when:

$$t_1(d_1) \approx t_2(d_2) \approx \ldots \approx t_p(d_p)$$

$$\begin{cases} t_i(d_i) = d_i / s_i(d_i), \\ d_1 + d_2 + \ldots + d_p = N \end{cases}$$

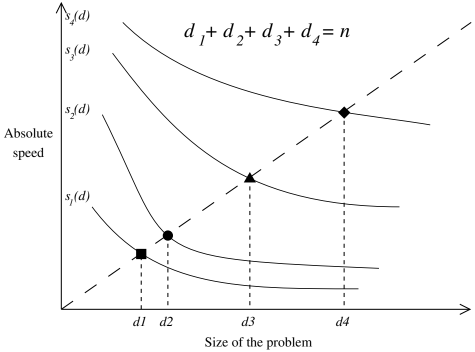


- All processors complete work within the same time
- Solution lies on a line passing through the origin when $d_i / s_i(d_i) = constant$
- However, only designed for heterogeneous uniprocessor cluster

* A. Lastovetsky et al: Data partitioning with a functional performance model of heterogeneous processors, 2007.

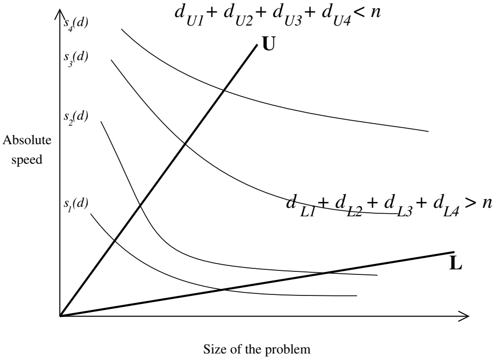
FPM-based data partitioning algorithm

- Total problem size determines the slope
- Algorithm iteratively bisects solution space to find values d_i



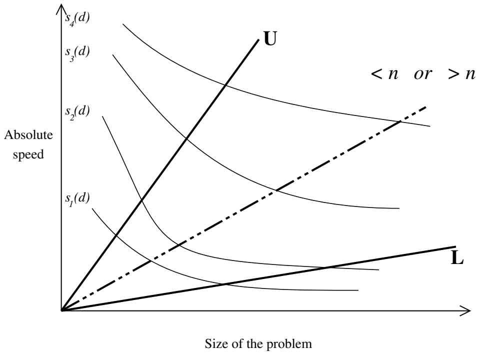
FPM-based data partitioning algorithm

- Total problem size determines the slope
- Algorithm iteratively bisects solution space to find values d_i



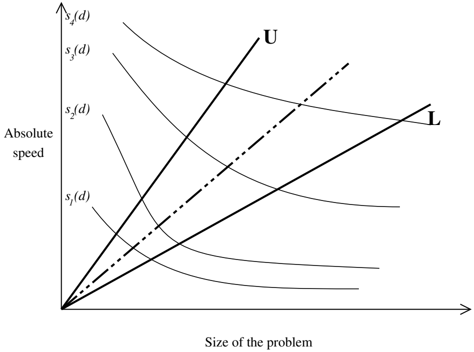
FPM-based data partitioning algorithm

- Total problem size determines the slope
- Algorithm iteratively bisects solution space to find values d_i



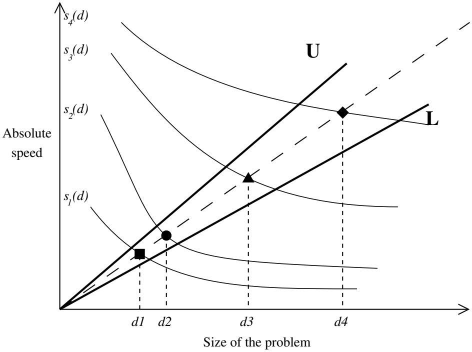
FPM-based data partitioning algorithm

- Total problem size determines the slope
- Algorithm iteratively bisects solution space to find values d_i



FPM-based data partitioning algorithm

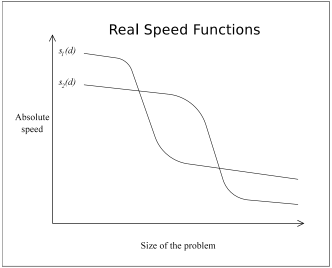
- Total problem size determines the slope
- Algorithm iteratively bisects solution space to find values d_i



Dynamic FPM-based data partitioning

Functional Performance Models may be built:

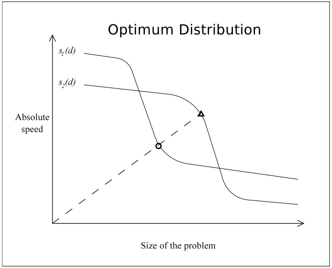
- exhaustively in advance
- dynamically at run time



Dynamic FPM-based data partitioning

Functional Performance Models may be built:

- exhaustively in advance
- dynamically at run time



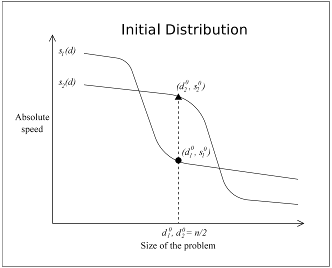
Dynamic FPM-based data partitioning

Functional Performance Models may 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 $s_i'(x) \equiv s_i^0$



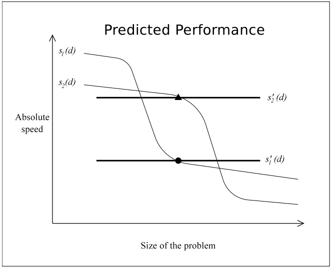
Dynamic FPM-based data partitioning

Functional Performance Models may 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 may be built:

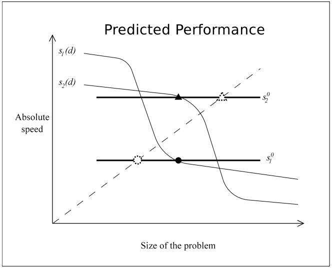
- exhaustively in advance
- dynamically at run time

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 may be built:

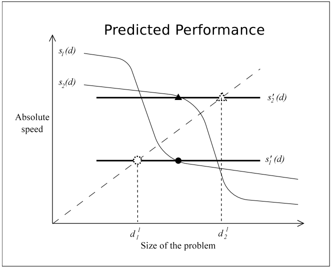
- exhaustively in advance
- dynamically at run time

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 may be built:

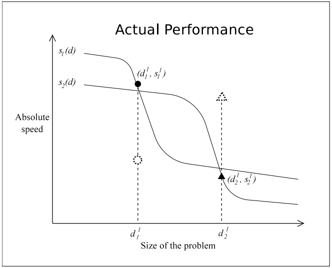
- exhaustively in advance
- dynamically at run time

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



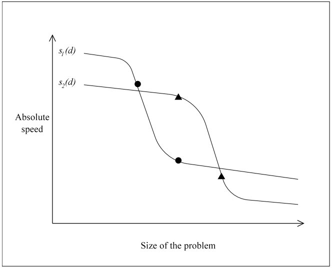
Dynamic FPM-based data partitioning

Functional Performance Models may 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 may be built:

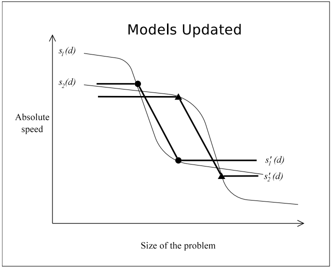
- exhaustively in advance
- dynamically at run time

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



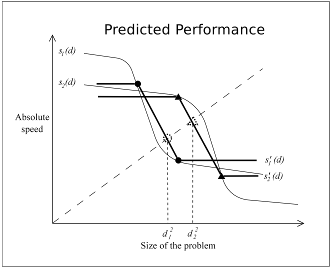
Dynamic FPM-based data partitioning

Functional Performance Models may 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 may be built:

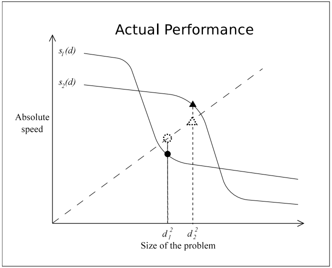
- exhaustively in advance
- dynamically at run time

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



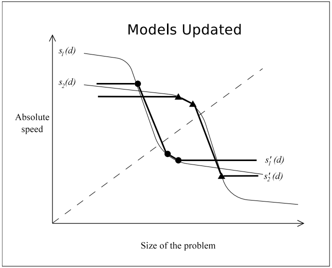
Dynamic FPM-based data partitioning

Functional Performance Models may 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 may be built:

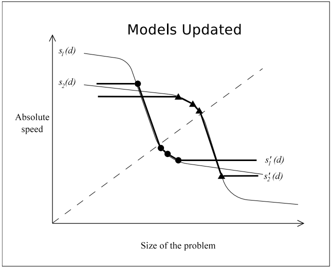
- exhaustively in advance
- dynamically at run time

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



Outline

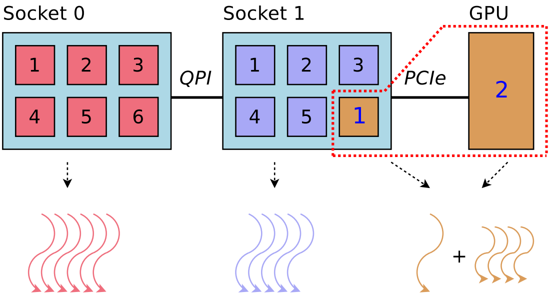
Programming Models for Hybrid Systems

- Data-parallel MPI program with calls to MT and GPGPU kernels
  - Hierarchical or flat execution on the cluster of hybrid nodes
- Partitioning compute devices of the node into independent groups
  - Identical cores
    - Running optimized MT kernel
    - Running multiple single-threaded kernels (one per core)
  - Core + GPU
    - Running native GPGPU kernel
    - Running out-of-core version of native GPGPU kernel
  - Identical core+GPU pairs
    - Running multiple native GPGPU kernels
  - Core + multi-GPU
    - Running multi-GPU kernel

Assumptions about program configuration

- No idle compute devices
  - May not be the optimal configuration (out of scope of this study)
  - May affect the independence of groups
- Even load of identical abstract processors
  - 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
  - The mapping is not delegated to the operating environment

Performance Measurement on Hybrid Node

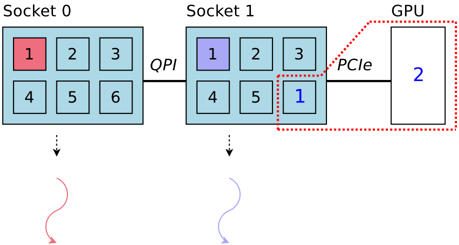


- 3 groups of devices: 6 cores, 5 cores and 1 core + GPU
- Cores in one group interfere with each other due to resource contention
- All cores in the group execute the same amount of workload in parallel
- Kernel computation time and data transfer time are both included
- Host core for GPU is chosen to maximize data throughput between GPU and NUMA memory

Outline

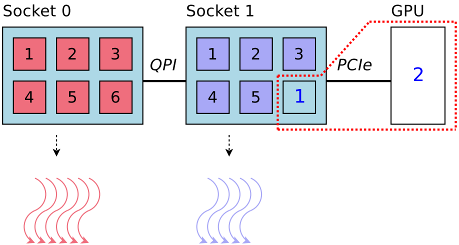
Functional Performance Models of multicore

- $s(x)$ speed of a core executing a single-threaded kernel exclusively
  $s(x) = x/t$



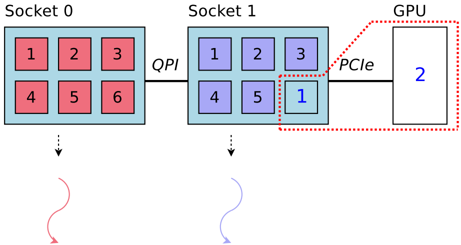
Functional Performance Models of multicore

- $s(x)$ speed of a core executing a single-threaded kernel exclusively
  $s(x) = x/t$
- $s_c(x)$ speed of a core that executes a single-threaded kernel and shares the system resources with identical cores, each core receives x units
  $s_c(x) = x/max_1^c(t_i)$

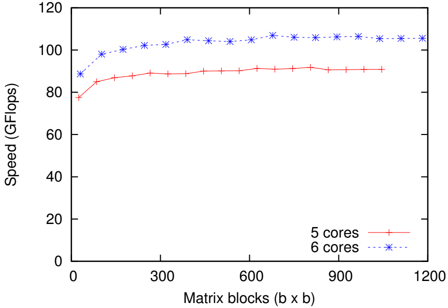


Functional Performance Models of multicore

- $s(x)$ speed of a core executing a single-threaded kernel exclusively
  $s(x) = x/t$
- $s_c(x)$ speed of a core that executes a single-threaded kernel and shares the system resources with identical cores, each core receives x units
  $s_c(x) = x/max_1^c(t_i)$
- $S_c(x)$ speed of c cores that execute a multi-threaded kernel and share system resources, x units distributed between cores
  $S_c(x) = x/t$



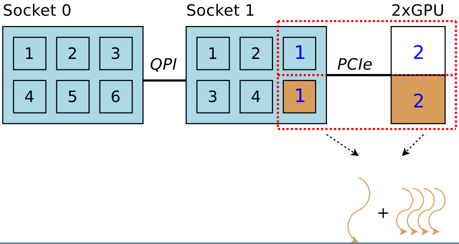
Functional Performance Models of multicore: Example



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

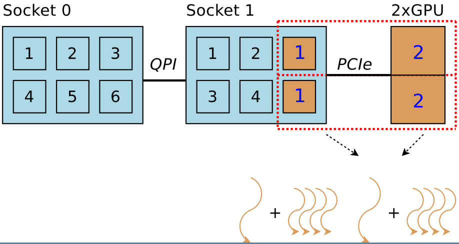
Functional Performance Models of GPU

- $g(x)$: combined speed of a GPU and its dedicated core, exclusive PCIe
  $g(x) = x/t$



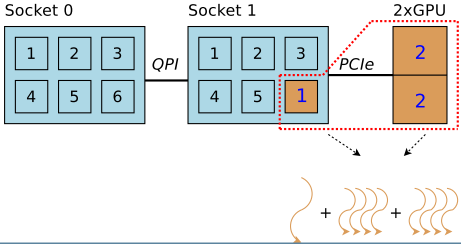
Functional Performance Models of GPU

- $g(x)$: combined speed of a GPU and its dedicated core, exclusive PCIe
  $g(x) = x/t$
- $g_d(x)$ combined speed of a GPU and its dedicated core, that share PCIe with identical pairs of processors, each pair receives x computation units
  $g_d(x) = x/max_1^d(t_i)$

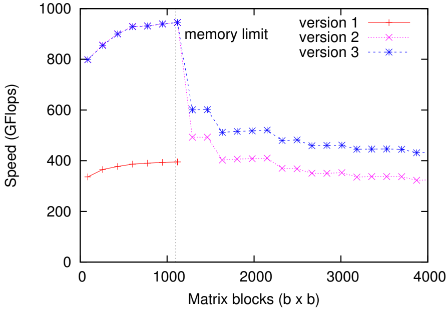


Functional Performance Models of GPU

- $g(x)$: combined speed of a GPU and its dedicated core, exclusive PCIe
  $g(x) = x/t$
- $g_d(x)$ combined speed of a GPU and its dedicated core, that share PCIe with identical pairs of processors, each pair receives x computation units
  $g_d(x) = x/max_1^d(t_i)$
- $G_d(x)$ combined speed of d GPUs and a dedicated CPU core that execute a multi-GPU kernel and share PCIe, x computation units are distributed between GPUs
  $G_d(x) = x/t$



Functional Performance Models of GPU: Example

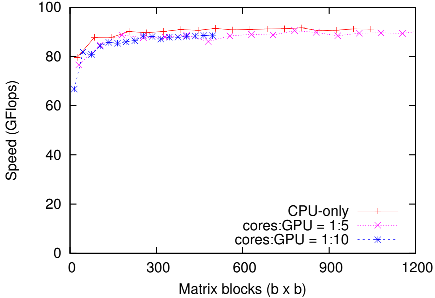


- $g(x)$ (version 1): naive kernel
- $g(x)$ (version 2): accumulate intermediate result + out-of-core
- $g(x)$ (version 3): version 2 + overlap data transfers and kernel executions

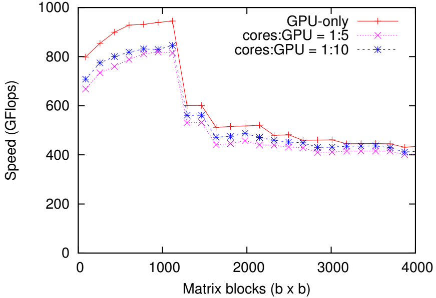
Impact of Resource Contention to Performance Modeling

- CPU and GPU kernels benchmarked simultaneously on 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



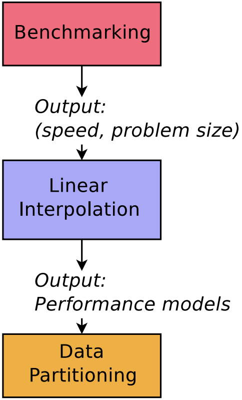
$g(x)$, speed of a GPU



Note: the above two figures have different scales, 1:10

Performance Modeling of Hybrid System

- Multicore/GPUs are modeled independently
  - Separate memory, programming models
  - Represented by speed functions (FPM)
  - Benchmarking with computational kernels
- Performance model of multicore:
  - Approximate the speed of multiple cores
  - e.g. all cores in a processor except the ones dedicated to GPUs
- Performance model of GPU:
  - Approximate combined speed of a GPU and it's dedicated core



Processing Flow

Outline

Applications: Linear Algebra

Linear Algebra applications:

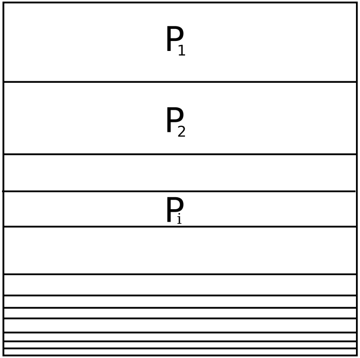
- Matrix multiplication
- LU decomposition
- Jacobi iterative method
- ...

How to optimally partition matrices?

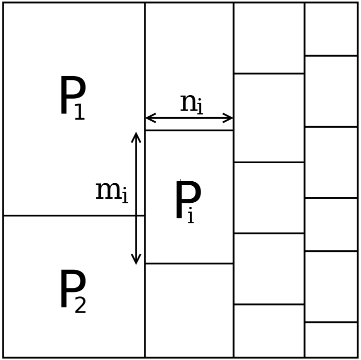
- Partition matrices between nodes
- Sub-partition between devices within a node
- To achieve load balancing, partition with respect to device and node speed
- Minimise total volume of communication

Matrix Partitioning

Simple Partitioning

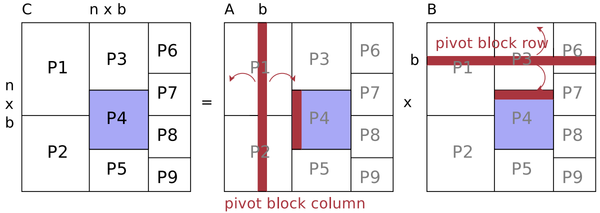


2D Partitioning



Matrix Multiplication on Heterogeneous Platform*

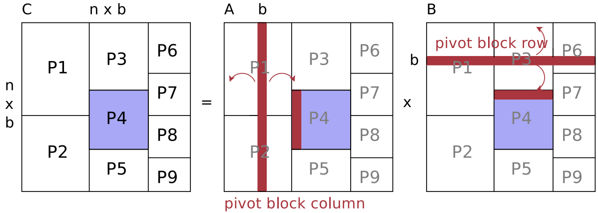
- Input: constant processor speeds
- Matrices partitioned so that
  - Area of the rectangle proportional to the speed
  - Volume of communication minimized



* Beaumont, O. et al: Matrix Multiplication on Heterogeneous Platforms. IEEE Trans. Parallel Distrib. Syst. 2001

Matrix Multiplication on Heterogeneous Platform*

- Input: constant processor speeds
- Matrices partitioned so that
  - Area of the rectangle proportional to the speed
  - Volume of communication minimized



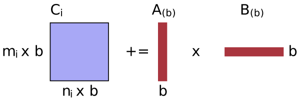
- More accurate solution is based on speed functions as input**

* Beaumont, O. et al: Matrix Multiplication on Heterogeneous Platforms. IEEE Trans. Parallel Distrib. Syst. 2001

** Clarke, D. et al: Column-Based Matrix Partitioning for Parallel Matrix Multiplication on Heterogeneous Processors Based on Functional Performance Models. In: HeteroPar-2011, LNCS 7155, 2012

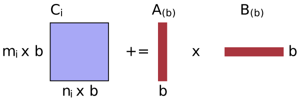
Matrix Multiplication on Heterogeneous Platform

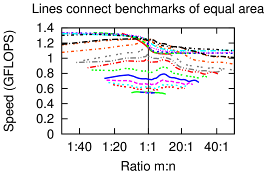
- Computational kernel: panel-panel update

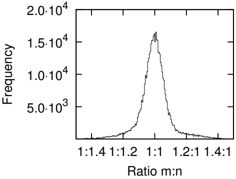


Matrix Multiplication on Heterogeneous Platform

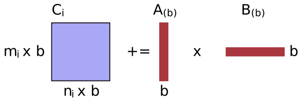
- Computational kernel: panel-panel update
- Processor speed - function of area
  Built by running the kernel for square matrices



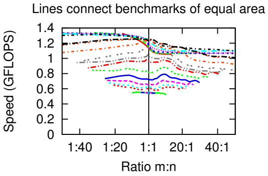


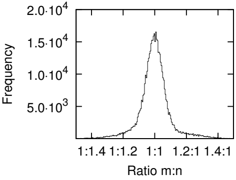


Matrix Multiplication on Heterogeneous Platform

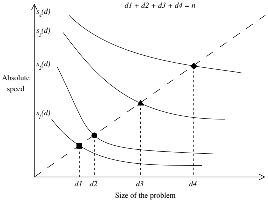


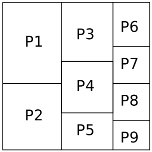
- Computational kernel: panel-panel update





- Processor speed - function of area
  Built by running the kernel for square matrices





- FPM-based partitioning algorithm finds the optimal areas
  The areas are used as input to the matrix partitioning algorithm

Outline

Matrix multiplication on hybrid node

Experimental platform

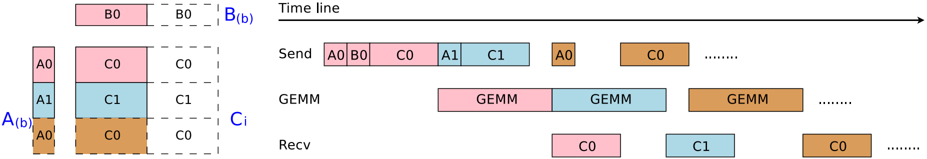
	CPU (AMD)	GPUs (NVIDIA)	
Architecture	Opteron 8439SE	GF GTX680	Tesla C870
Core Clock	2.8 GHz	1006 MHz	600 MHz
Number of Cores	4×6 cores	1536 cores	128 cores
Memory Size	4×16 GB	2048 MB	1536 MB
Memory Bandwidth		192.3 GB/s	76.8 GB/s

Computational Kernels for Hybrid Node

- Multicore CPU:
  - GEMM routine from ACML library
  - Multi-threaded processes (one per socket)
- GPU accelerator:
  - GEMM routine from CUBLAS library
  - Develop out-of-core kernel to overcome memory limitation
  - Overlap data transfers and kernel execution to hide latency

Out-of-core Kernel, Overlap of Data Transfers and Kernel Execution:

- allocated 5 buffers in device memory: A0, A1, B0, C0, C1



Experiments on hybrid multicore multi-GPU node

Execution time of the application under different configurations

Matrix size (blks)	CPUs (sec)	GTX680 (sec)	Hybrid-FPM (sec)
40×40	99.5	74.2	26.6
50×50	195.4	162.7	77.8
60×60	300.1	316.8	114.4
70×70	491.6	554.8	226.1

Column 1: block size is 640×640
Column 2: 4×6 CPU cores, homogeneous data partitioning
Column 3: CPU core + GPU
Column 4: 2×6 CPU cores + 2×5 CPU cores + $2 \times$ (CPU core + GPU), FPM-based data partitioning

Experiments on hybrid multicore multi-GPU node

Execution time of the application under different configurations

Matrix size (blks)	CPUs (sec)	GTX680 (sec)	Hybrid-FPM (sec)
40×40	99.5	74.2	26.6
50×50	195.4	162.7	77.8
60×60	300.1	316.8	114.4
70×70	491.6	554.8	226.1

Column 1: block size is 640×640
Column 2: 4×6 CPU cores, homogeneous data partitioning
Column 3: CPU core + GPU
Column 4: 2×6 CPU cores + 2×5 CPU cores + $2 \times$ (CPU core + GPU), FPM-based data partitioning

Experiments on hybrid multicore multi-GPU node

Execution time of the application under different configurations

Matrix size (blks)	CPUs (sec)	GTX680 (sec)	Hybrid-FPM (sec)
40×40	99.5	74.2	26.6
50×50	195.4	162.7	77.8
60×60	300.1	316.8	114.4
70×70	491.6	554.8	226.1

Column 1: block size is 640×640
Column 2: 4×6 CPU cores, homogeneous data partitioning
Column 3: CPU core + GPU
Column 4: 2×6 CPU cores + 2×5 CPU cores + $2 \times$ (CPU core + GPU), FPM-based data partitioning

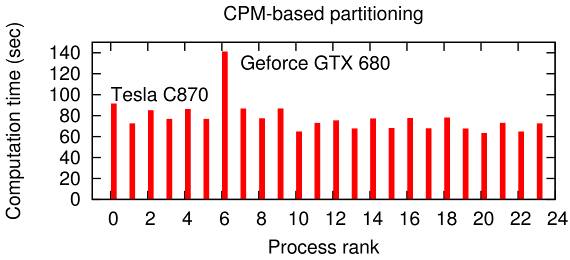
Experiments on hybrid multicore multi-GPU node

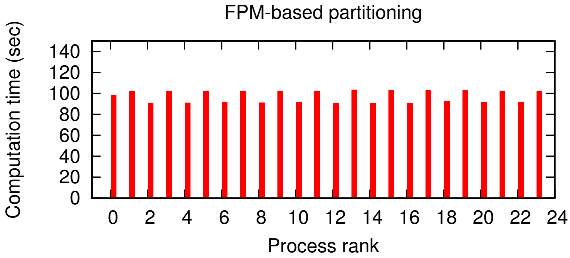
Execution time of the application under different configurations

Matrix size (blks)	CPUs (sec)	GTX680 (sec)	Hybrid-FPM (sec)
40×40	99.5	74.2	26.6
50×50	195.4	162.7	77.8
60×60	300.1	316.8	114.4
70×70	491.6	554.8	226.1

Column 1: block size is 640×640
Column 2: 4×6 CPU cores, homogeneous data partitioning
Column 3: CPU core + GPU
Column 4: 2×6 CPU cores + 2×5 CPU cores + $2 \times$ (CPU core + GPU), FPM-based data partitioning

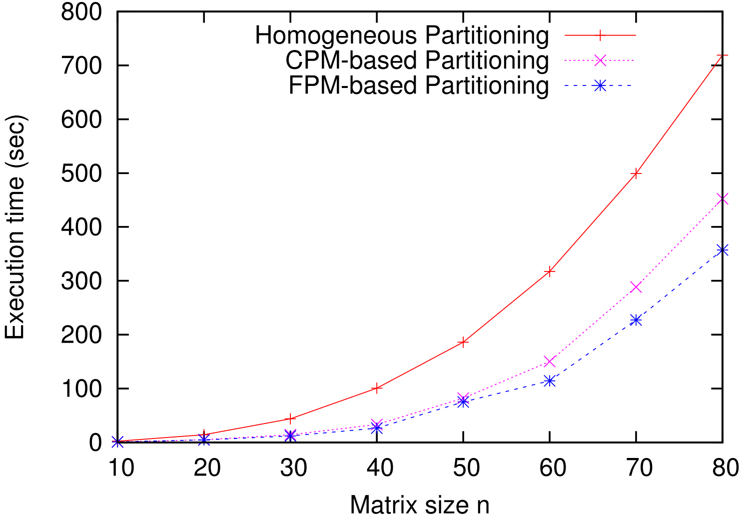
Computation time of each process





Matrix size 60×60, Computation time reduced by 40%

Performance with different partitionings



Execution time reduced by 23% and 45% respectively

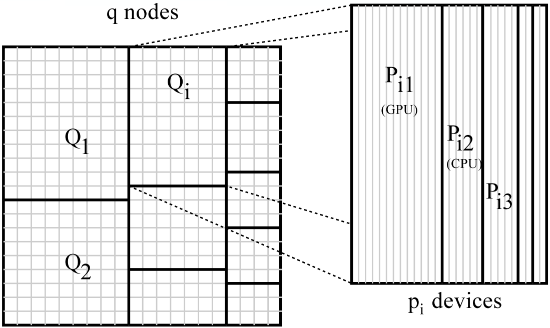
Outline

Data partitioning on heterogeneous cluster of hybrid nodes

- Target platform - dedicated heterogeneous cluster of hybrid nodes
- Hierarchical partitioning algorithm
  - Dynamic algorithm - no a priori information about performance required.
  - Inputs:
    - Problem size
    - Number of nodes
    - Number of devices per node
    - Device type (eg. cpu, gpu, ...).
  - Link computational kernel to be benchmarked for each device.

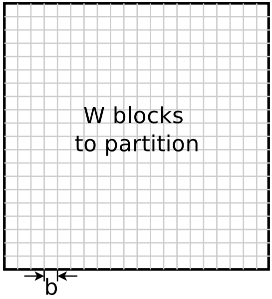
  - Initially distribution is partitioned evenly between nodes and between devices within a node
  - Algorithm converges towards optimum solution

Hierarchical Partitioning Algorithm



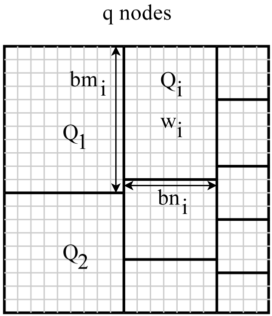
- q nodes, $Q_1, \ldots, Q_q$.
- node Q_i has p_i devices, $P_{i1}, \ldots, P_{ip_i}$
- Hierarchy in platform $\rightarrow$ hierarchy in partitioning
  - Nested parallelism
  - *inter-node partitioning algorithm (INPA)*
  - *inter-device partitioning algorithm (IDPA)*
  - IDPA is nested inside INPA

Hierarchical Partitioning Algorithm



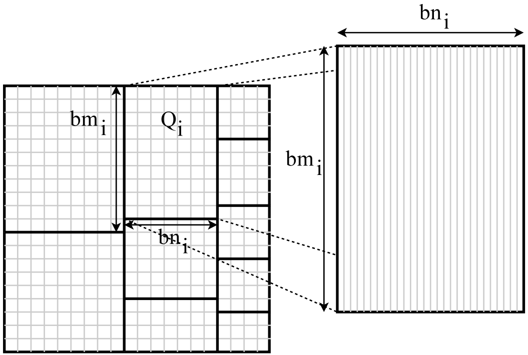
- W computational units to partition between nodes
- inter-node partitioning algorithm (INPA) creates node-FPM's and computes $w_1, \ldots, w_q$
  so that $w_1 + \ldots + w_q = W$.

Hierarchical Partitioning Algorithm



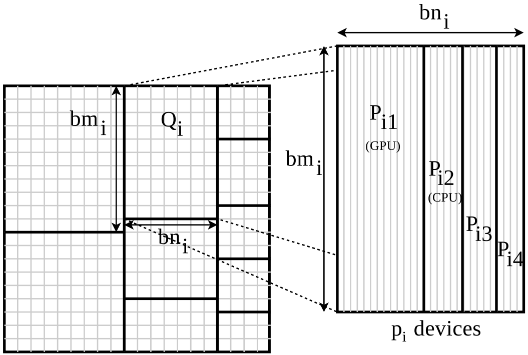
- Communication minimising algorithm has input: $w_1, \ldots, w_q$ and output: $(m_1, n_1), \ldots, (m_q, n_q)$ such that $m_i \times n_i = w_i$ and matrix is completely tiled.

Hierarchical Partitioning Algorithm

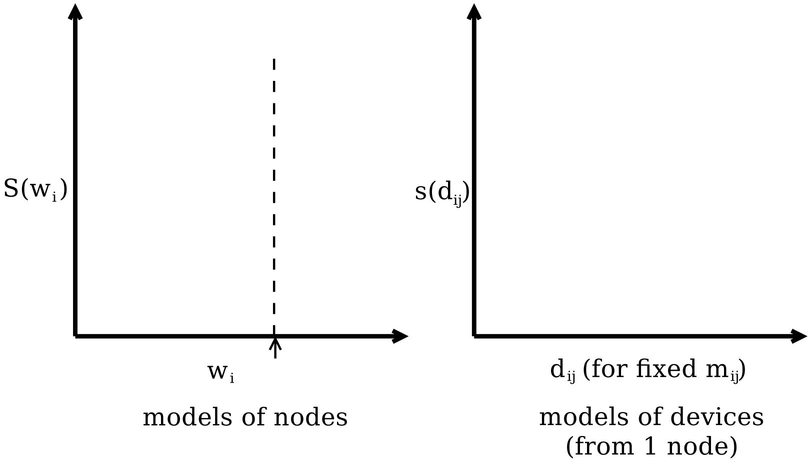


- inter-device partitioning algorithm (IDPA) creates device-FPM's and computes $d_{i1}, \ldots, d_{ip}$,
  such that $d_{i1} + \ldots + d_{ip} = bn_i$

Hierarchical Partitioning Algorithm

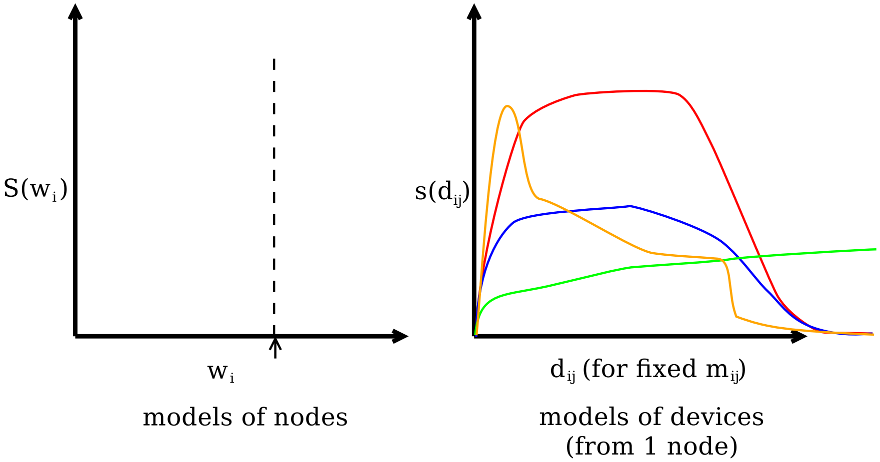


- inter-device partitioning algorithm (IDPA) creates device-FPM's and computes $d_{i1}, \ldots, d_{ip}$, such that $d_{i1} + \ldots + d_{ip} = bn_i$



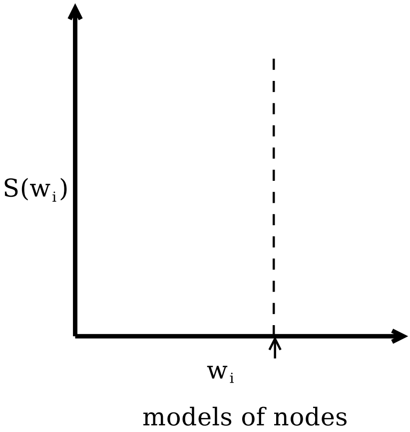
$$w_i = m_i \times n_i$$

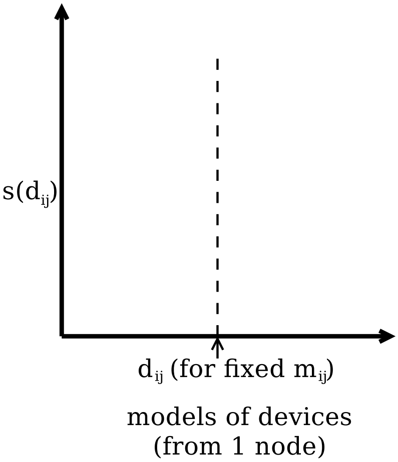
$$\sum_{j=1}^{p} d_{ij} = b \times n_i$$



$$w_i = m_i \times n_i$$

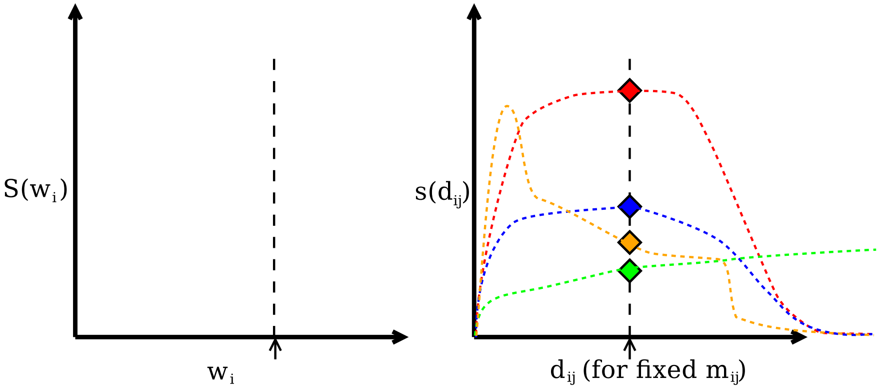
$$\sum_{j=1}^{p} d_{ij} = b \times n_i$$





$$w_i = m_i \times n_i$$

$$\sum_{j=1}^{p} d_{ij} = b \times n_i$$

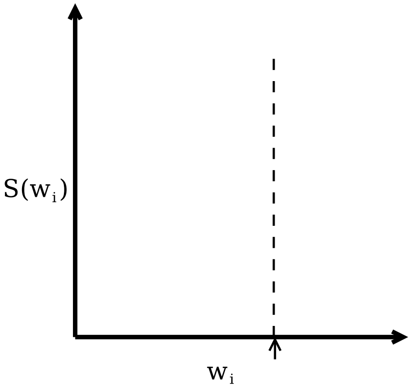


models of nodes

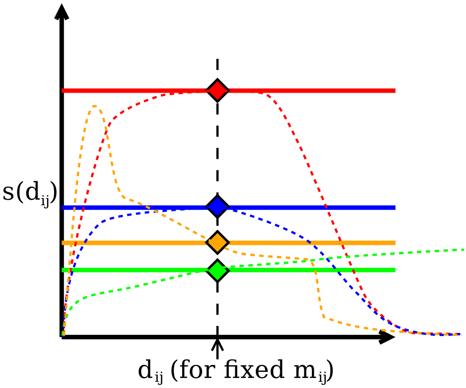
models of devices
(from 1 node)

$$w_i = m_i \times n_i$$

$$\sum_{j=1}^{p} d_{ij} = b \times n_i$$



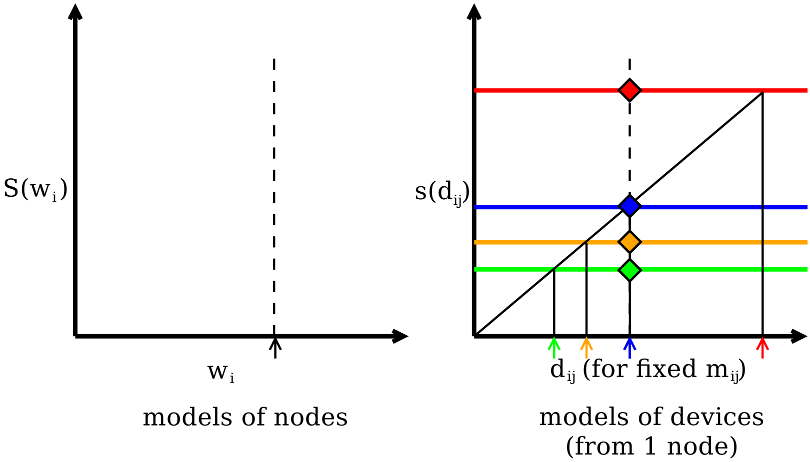
models of nodes



models of devices (from 1 node)

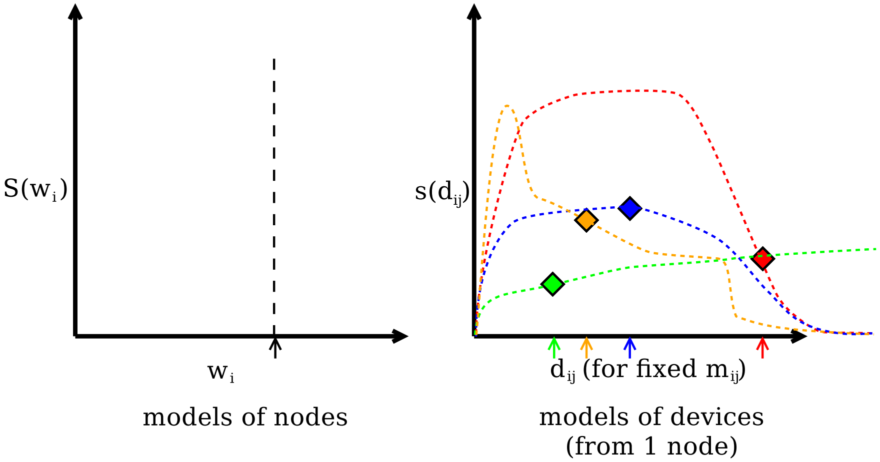
$$w_i = m_i \times n_i$$

$$\sum_{j=1}^{p} d_{ij} = b \times n_i$$



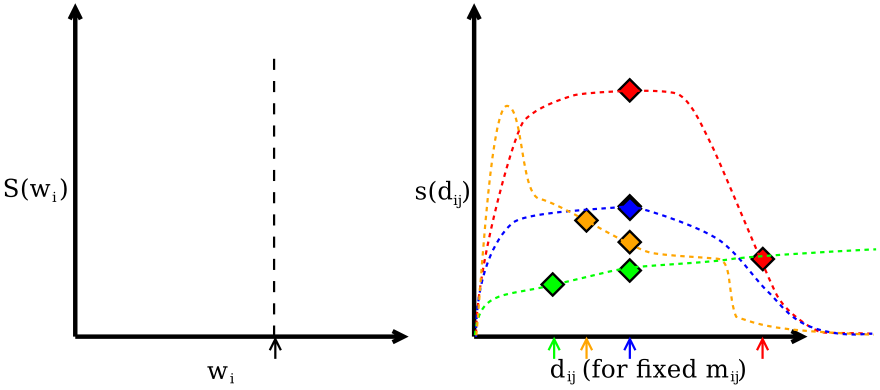
$$w_i = m_i \times n_i$$

$$\sum_{j=1}^{p} d_{ij} = b \times n_i$$



$$w_i = m_i \times n_i$$

$$\sum_{j=1}^{p} d_{ij} = b \times n_i$$

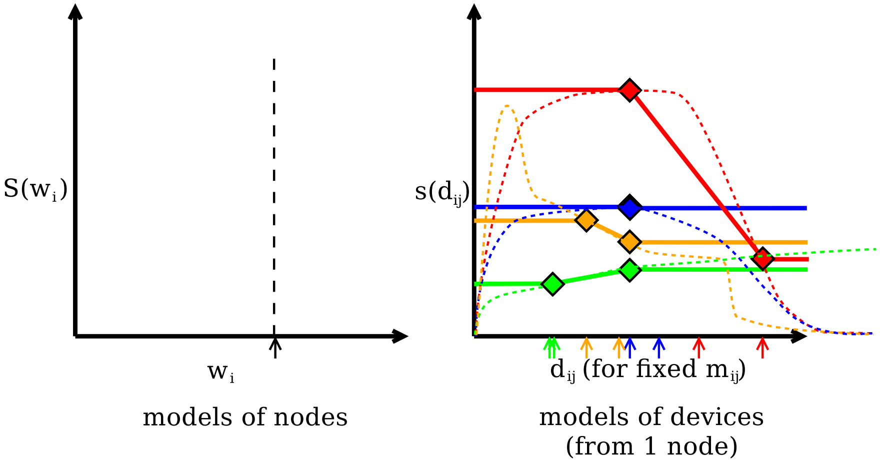


models of nodes

models of devices (from 1 node)

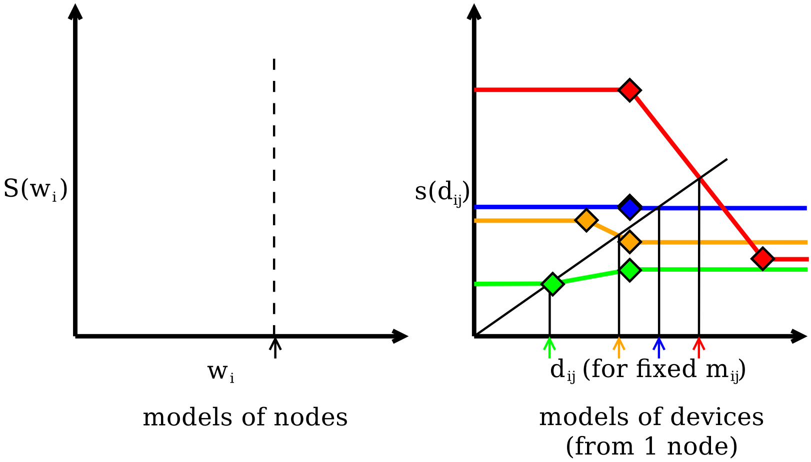
$$w_i = m_i \times n_i$$

$$\sum_{j=1}^{p} d_{ij} = b \times n_i$$



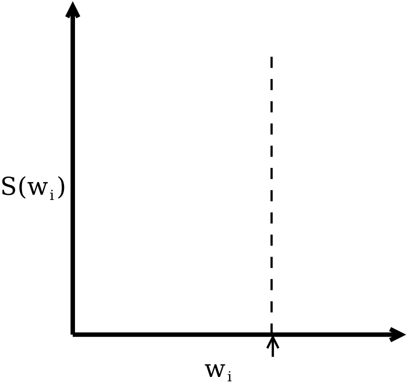
$$w_i = m_i \times n_i$$

$$\sum_{j=1}^{p} d_{ij} = b \times n_i$$

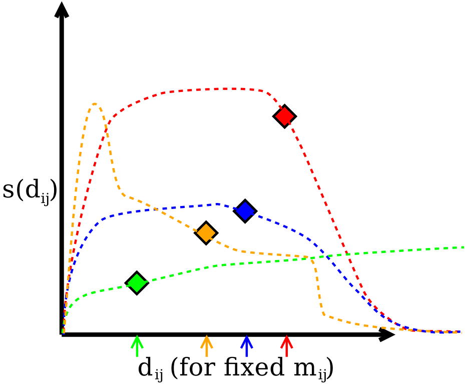


$$w_i = m_i \times n_i$$

$$\sum_{j=1}^{p} d_{ij} = b \times n_i$$



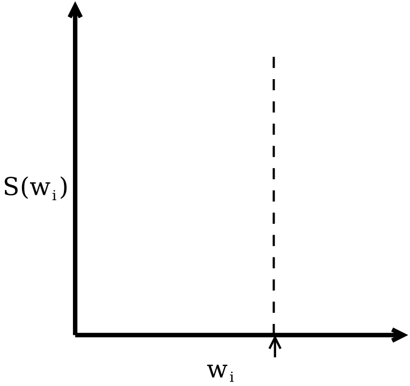
models of nodes



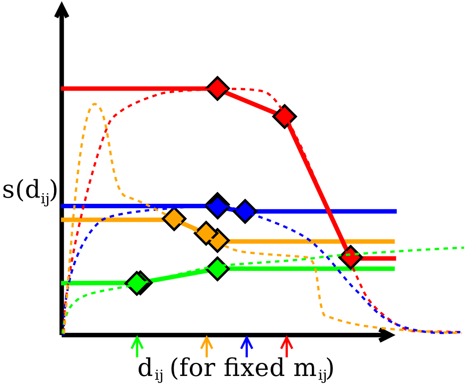
models of devices
(from 1 node)

$$w_i = m_i \times n_i$$

$$\sum_{j=1}^{p} d_{ij} = b \times n_i$$



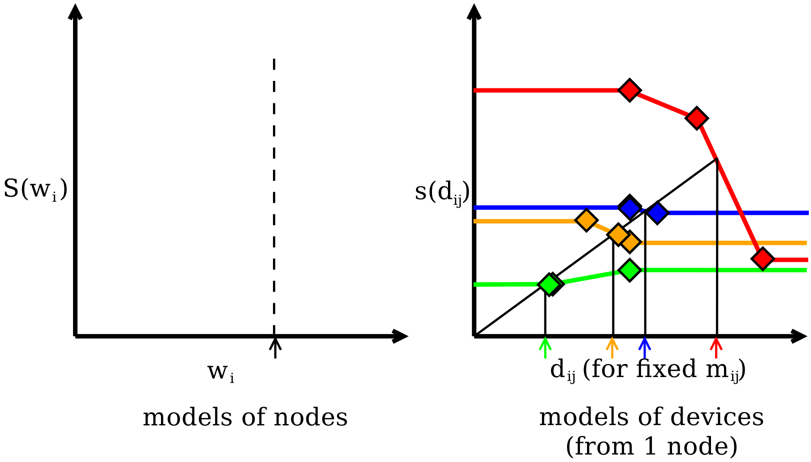
models of nodes



models of devices
(from 1 node)

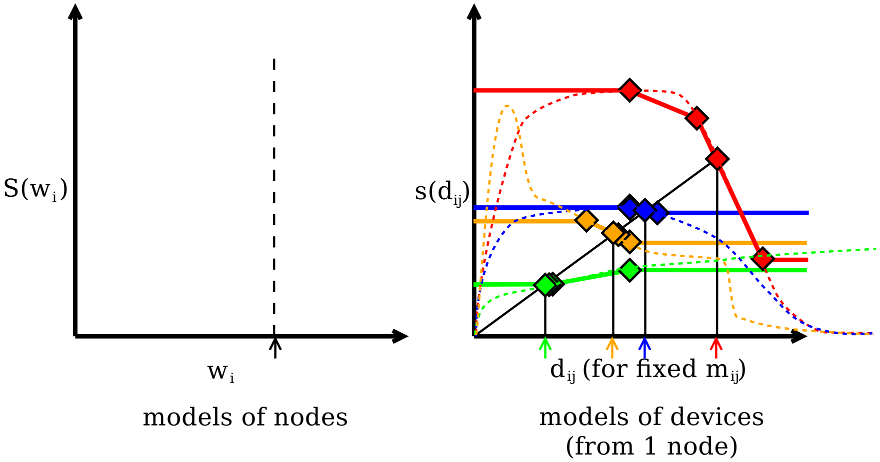
$$w_i = m_i \times n_i$$

$$\sum_{j=1}^{p} d_{ij} = b \times n_i$$



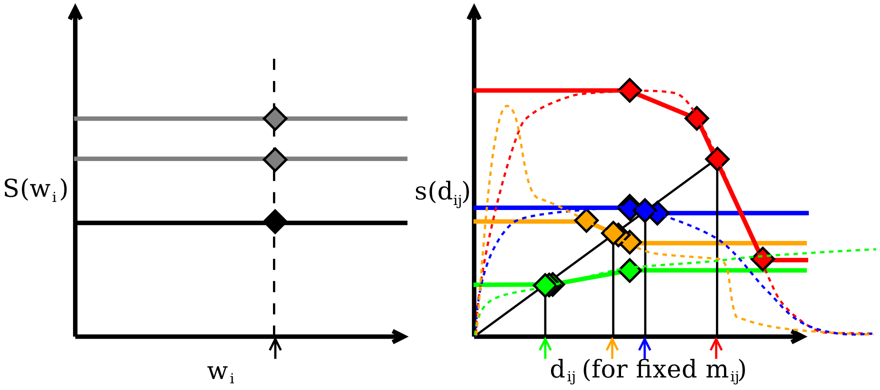
$$w_i = m_i \times n_i$$

$$\sum_{j=1}^{p} d_{ij} = b \times n_i$$



$$w_i = m_i \times n_i$$

$$\sum_{j=1}^{p} d_{ij} = b \times n_i$$

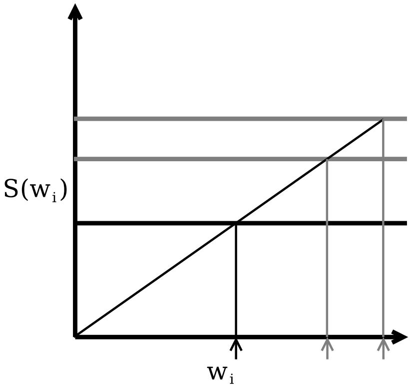


models of nodes

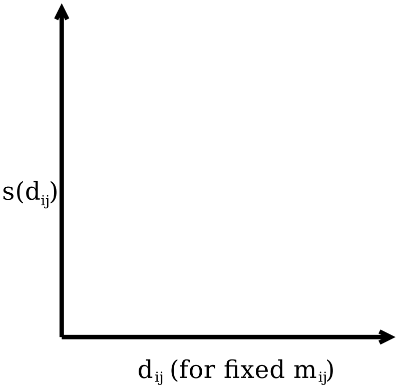
models of devices
(from 1 node)

$$w_i = m_i \times n_i$$

$$\sum_{j=1}^{p} d_{ij} = b \times n_i$$



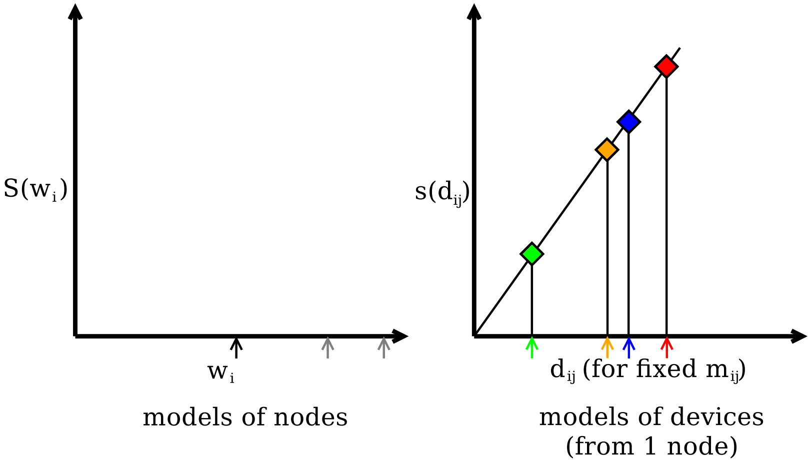
models of nodes



models of devices
(from 1 node)

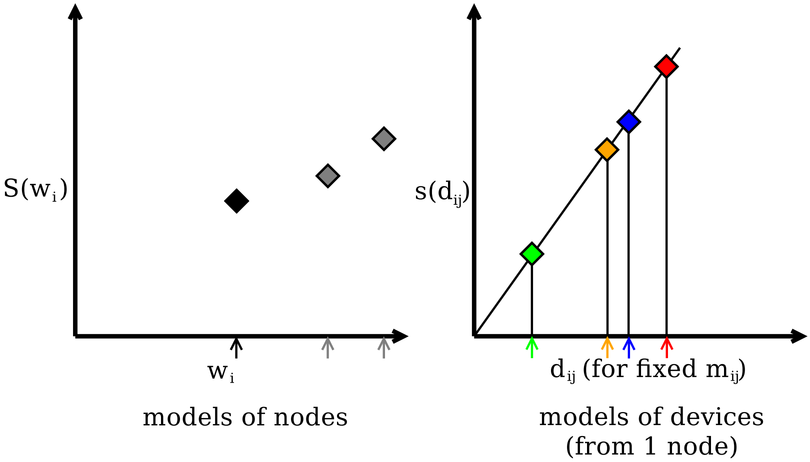
$$w_i = m_i \times n_i$$

$$\sum_{j=1}^{p} d_{ij} = b \times n_i$$



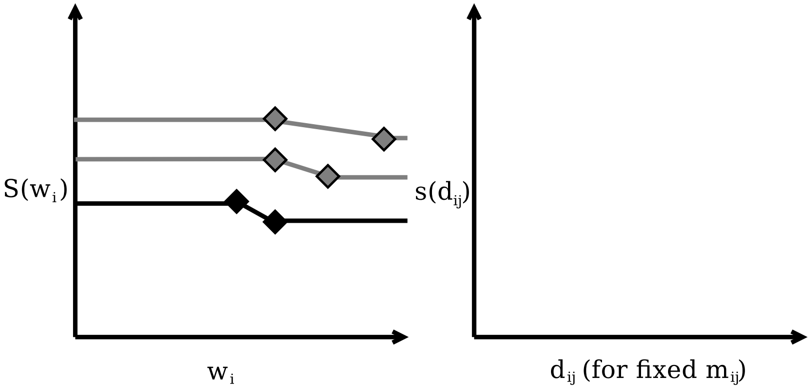
$$w_i = m_i \times n_i$$

$$\sum_{j=1}^{p} d_{ij} = b \times n_i$$



$$w_i = m_i \times n_i$$

$$\sum_{j=1}^{p} d_{ij} = b \times n_i$$

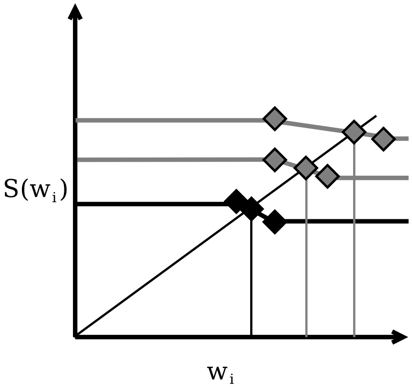


models of nodes

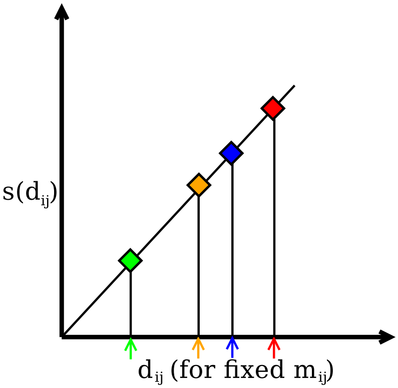
models of devices
(from 1 node)

$$w_i = m_i \times n_i$$

$$\sum_{j=1}^{p} d_{ij} = b \times n_i$$



models of nodes



models of devices
(from 1 node)

$$w_i = m_i \times n_i$$

$$\sum_{j=1}^{p} d_{ij} = b \times n_i$$

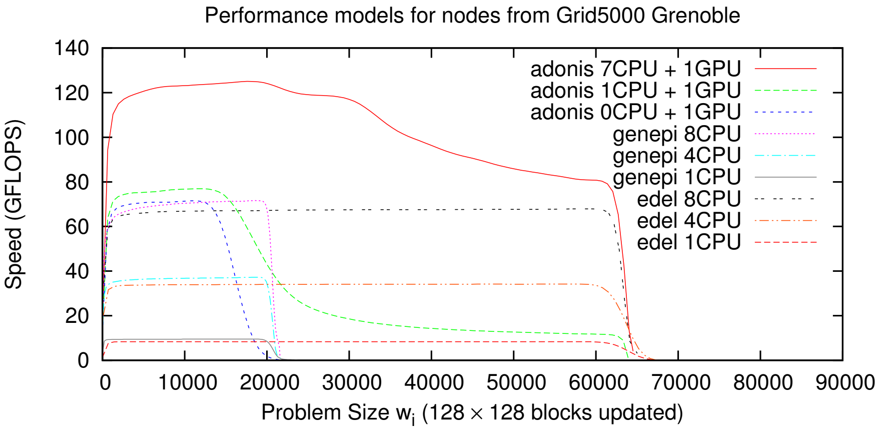
Experimental Setup

90 Nodes from Grid5000 Grenoble site

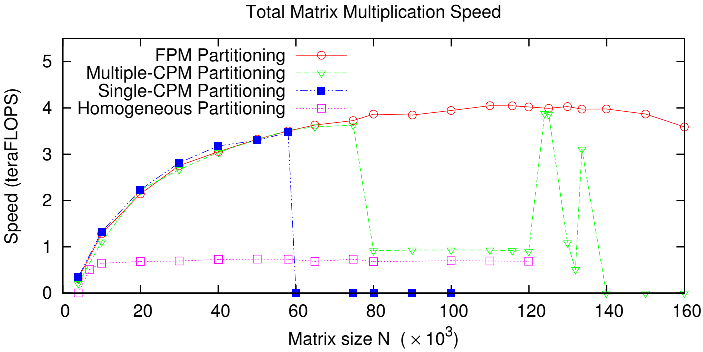
Cores:	0	1	2	3	4	5	6	7	8	Nodes	Cores	GPUs	Hardware
Adonis	2	1	1	1	1	1	2	3	0	12	48	12	2.27/2.4GHz Xeon, 24GB
Edel	0	6	4	4	4	8	8	8	8	50	250	0	2.27GHz Xeon, 24GB
Genepi	0	3	3	3	3	4	4	4	4	28	134	0	2.5GHz Xeon, 8GB
Total										90	432	12	

- All nodes connected with InfiniBand communication network.
- High performance BLAS libraries: Intel MKL for CPU, CUBLAS for GPU devices.
- Open MPI for inter node communication.
- OpenMP for inter-device parallelism.

Experimental Results



Experimental Results



- Functional performance model (FPM): the proposed algorithm
- Multiple constant performance models (CPM): Redistribute based on previous benchmark.
- Single-CPM: One benchmark is preformed.
- Homogeneous distribution: Partitioned evenly between nodes, then evenly between devices within each node.

Conclusion

- Defined and built functional performance models (FPMs) of hybrid multicore and multi-GPU system, considering it as a distributed memory system
- Adapted FPM-based data partitioning to hybrid node, achieved load balancing and delivered good performance
- Adapted dynamic FPM-based data partitioning to hybrid cluster, achieved self-adaptiveness

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