IT – Portable Parallel Performance

Andrew Grimshaw & Yan Yanhaona
CCDCS
Chateauform
La Maison des Contes
October 3-6, 2016

I come not to bury MPI

but to layer on top of it.

What is IT?

- IT is an language to experiment with PCubeS (multi-space) parallel language constructs and performance.
- IT is designed to address the challenge of writing portable, performant, parallel programs.
- IT is the brain-child of Yan Yanhaona.

Agenda

- The problem – the five P's
- Current Practice
- The PCubeS Type Architecture
- IT – a PCubeS language
- Performance
- Conclusions and Future Work

The Problem

Productive, Portable, Performing, Predictable, Parallel Programs

Parallel programming is hard

- Seitz once said parallel programming is no harder than sequential programming.
- Time spent dealing with parallelization, parallel correctness, performance, and porting is time not spent on the application.
- Optimization is hardware dependent. Memory hierarchies are deep and getting deeper
- Increasingly heterogeneous environments

The problem is not getting any easier

Once solved for one machine you then face the portability problem

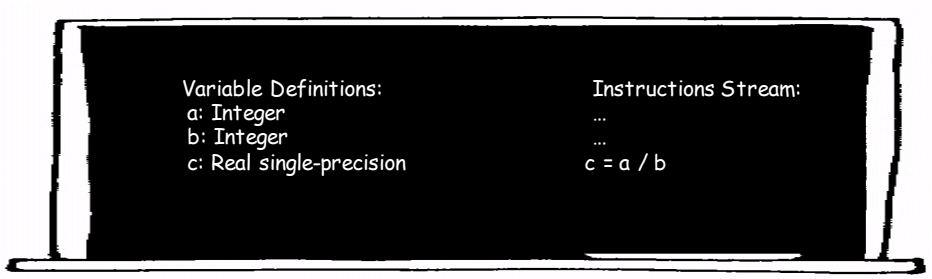
Problem identified by Snyder

- The salient features of an architecture must be reflected in programming languages or the programmer will be misled.
- The language influences algorithms and constrains how the programmer can express the solution.

Lawrence Snyder. Annual review of computer science vol. 1, 1986. chapter Type Architectures, Shared Memory, and the Corollary of Modest Potential, pages 289–317. Annual Reviews Inc., Palo Alto, CA, USA, 1986.

Von Neumann

- Fetch/execute over a flat random access memory



- Very successful – the model provides an abstraction that has been implemented over a wide variety of physical machines.
- Imperative languages map easily to the model.
- The compilers job is relatively simple.

We have not found an analog to the Von Neumann machine

Agenda

- The problem – the five P's
- Current Practice
- The PCubeS Type Architecture
- IT – a PCubeS language
- Performance
- Conclusions and Future Work

- Hundreds of parallel languages from the 80's to today
- Dominant life forms
  - MPI
    - Reflects a type architecture of communicating sequential processes quite well. Clearly separates "local" from "remote" communication and synchronization.
  - Pthreads
  - OpenMP
    - Syntactic sugar for Pthreads. Reflects shared memory type architecture with assumption of uniform access. Works well at small scale, but fails as more and more cores are added.
  - CUDA
- Modern attempts to solve the problem
  - PGAS
  - Fortress, X10 …

Programmer is responsible for

- Deciding where to perform computations, e.g., cores, GPUs, SMs
- Deciding how to decompose and distribute data structures
- Deciding where to place data structures, including managing caches
- Managing the communication and synchronization to ensure that the right data is in the right place at the right time
- All in the face of asynchrony

Our Approach

1. Develop an abstraction to view different hardware architectures in a uniform way.
   - Abstraction must expose salient architectural features of a hardware.
   - Cost of using those features should be apparent.
   - We call this Partitioned Parallel Processing Spaces – **PCubeS**

   Type Architecture: Laurence Snyder, 1986
2. Then develop programming paradigms that work over that abstraction.
   - Paradigms should be easy to understand.
   - IT is the first PCubeS language.

Objective: once you learn the fundamentals, you should be able to write efficient parallel programs for any hardware platform.

Basic idea

- Think of the hardware of consisting of layers of processing and memory.
  - Node layer, socket layer (w/L1, L2, L3), core layer, GPU layer, SM layer, warp layer.
- Define software "spaces" or "planes" that consist of processing done at that layer over data structures defined at that layer.
- Map the software spaces to the hardware layers.
- Sub-divide the spaces into sub-spaces defined by the partitioning of arrays in the spaces. Processing occurs in these spaces called Logical Processing Spaces (LPUs).
  - This can be done recursively to arbitrary depth.
- LPUs are mapped to physical processing units (PPUs) at the corresponding hardware layer.

Programmer Responsibility

- Programmers are responsible for deciding which tasks execute in which space, for partitioning the data within LPSes, and for mapping the LPSes to PPSes

Agenda

- The problem – the five P's
- Current Practice
- The PCubeS Type Architecture
- IT – a PCubeS language
- Performance
- Conclusions and Future Work

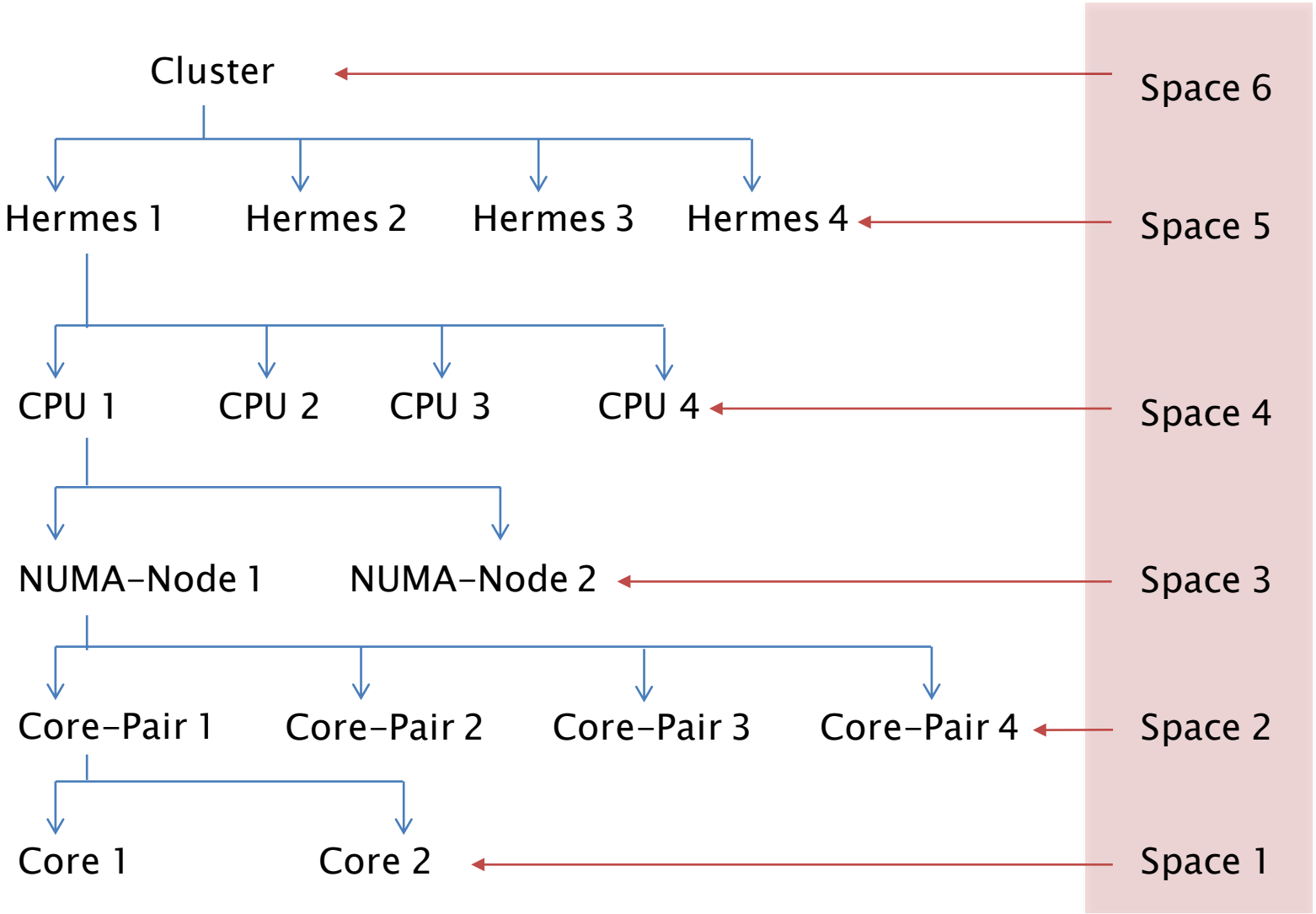
Partitioned Parallel Processing Spaces (PCubeS)

PCubeS is a finite hierarchy of parallel processing spaces (PPS) each having fixed, possibly zero, compute and memory capacities and containing a finite set of uniform, independent sub-spaces (PPU) that can exchange information with one another and move data to and from their parent.

Fundamental Operations of a Space:

- Floating point arithmetic
- Data Transfer

PCubeS Example: Hermes Cluster



PCubeS for Supercomputers

The Mira Supercomputer

- Blue Gene Q System
- 49,152 IBM Power PC A2 nodes
- 18 Cores Per Node
- 5D Torus Node Interconnect Network

Space 6: Rack
Processing Capacity: None
Memory: None
Sub–spaces: 2
Information Exchange: rack crossing latency

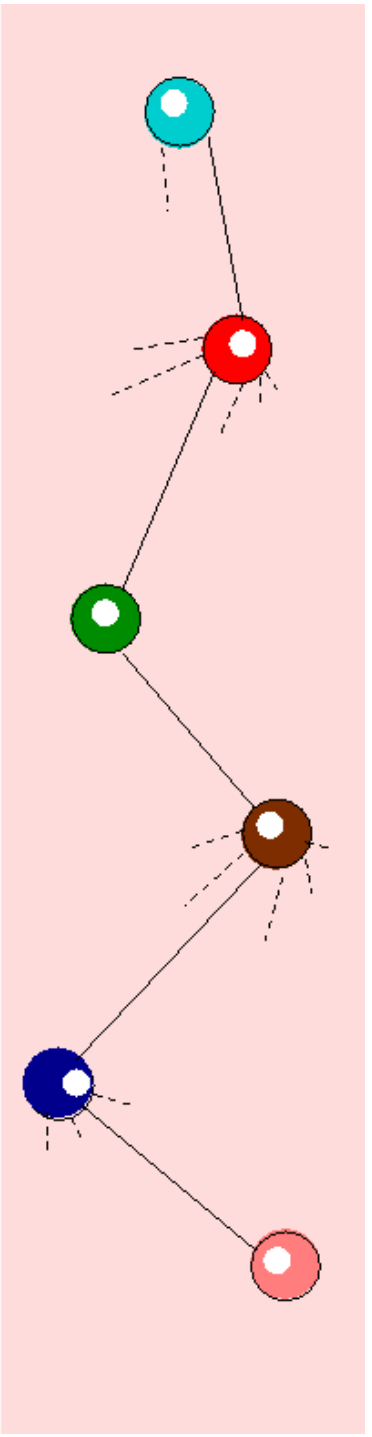
Space 5: Midplane
Processing Capacity: None
Memory: None
Sub–spaces: 512
Information Exchange: midplane crossing latency

Space 4: Node/CPU
Processing Capacity: None
Memory: 16 GB
Sub–spaces: 1
Information Exchange: 9 links traversal delay

Space 3: L2 Cache
Processing Capacity: None
Memory: 32MB
Sub–spaces: 16

Space 2: Core
Processing Capacity: None
Memory: 16 KB L1
Sub–spaces: 4

Space 1: Hyperthread
Processing Capacity: 4 operations
Speed 400 MHz
Memory: None
Sub–spaces: None
Informaction Exchange: register write

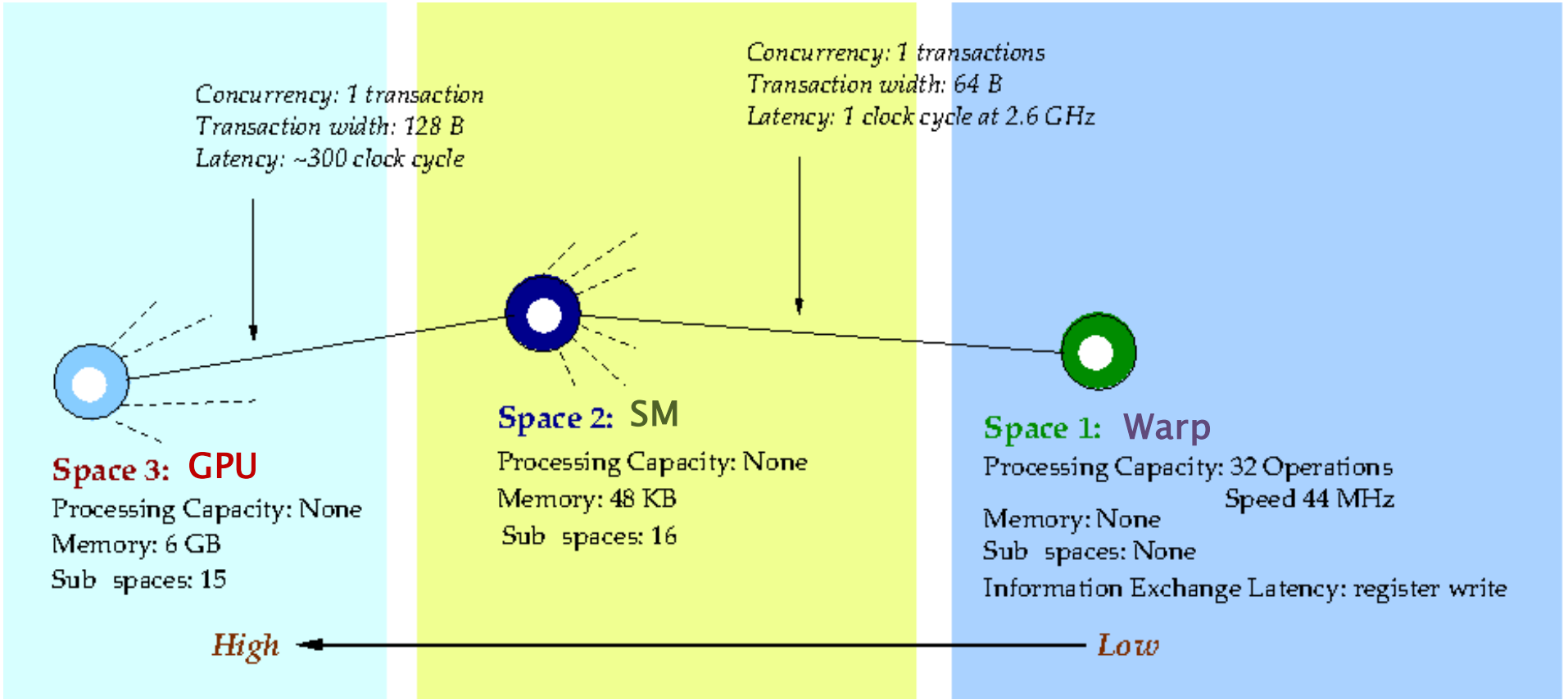


PCubeS Example: NVIDIA Tesla K20

Source: NVIDIA

- Core frequency 706 MHz
- 2496 CUDA cores
- 6GB on board memory
- 64KB shared memory

- 15 SMs
- Ideally 16 Warps Per SM
- 32 threads read/write at once
- 48 KB shared memory accessible



Agenda

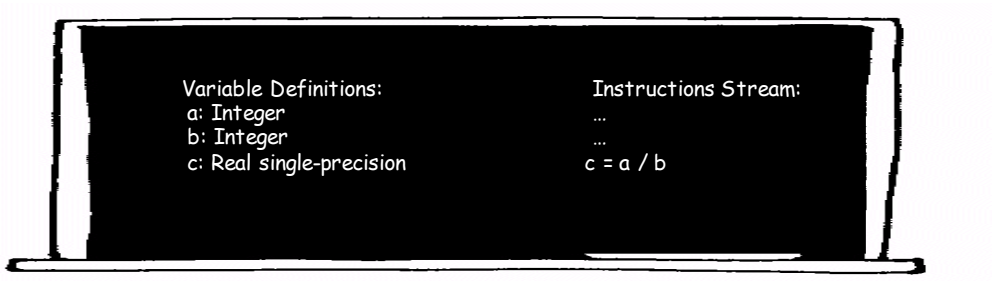
- The problem – the five P's
- Current Practice
- The PCubeS Type Architecture
- IT – a PCubeS language
- Performance
- Conclusions and Future Work

IT Parallel Programming Language

- Has a declarative pseudo-code like syntax.
- Characterized by emphasis on separation of concerns.
- IT is a *PCubeS* language.
- Programs and data structures are defined with respect to one or more possibly nested logical processing spaces (LPSes).
- Data partitioning and mapping are defined separately from the specification of the algorithm, i.e., the code written by the programmer is written in a data partitioning and placement-independent manner.
- Data partitioning and mapping are specified for each target execution environment and code is generated specifically for the target environment without the programmer needing to re-write any code.

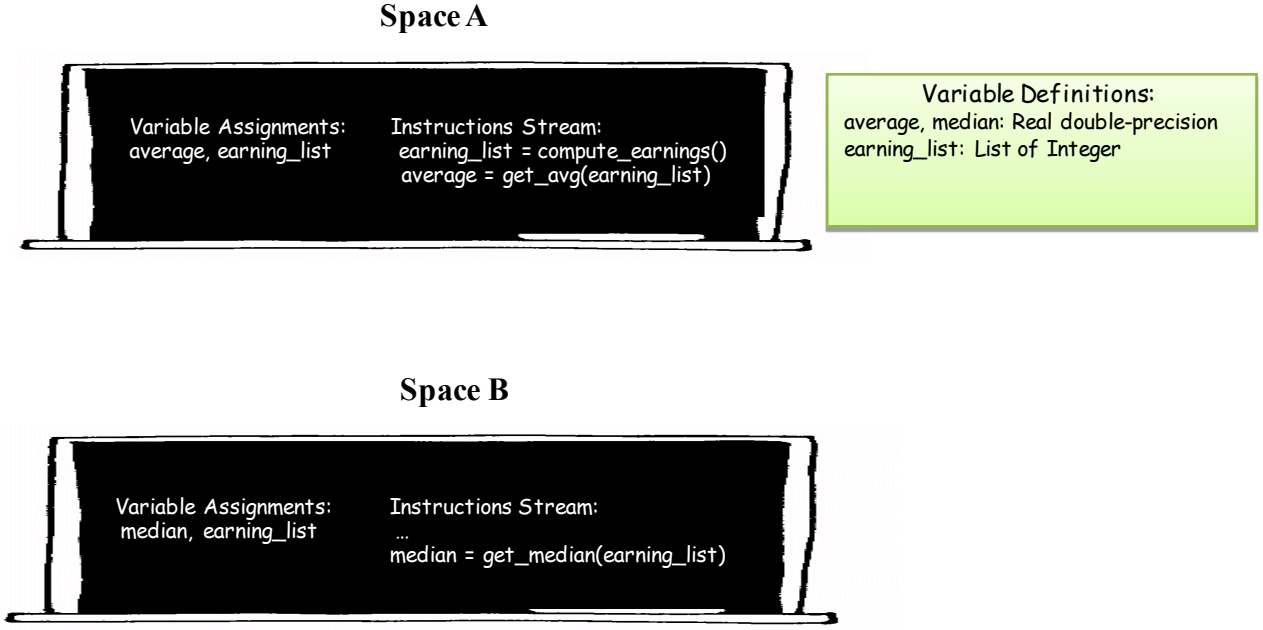
Goal: approximate the performance of low level techniques

Von Neumann single space



Multiple spaces

- Variables and functions exist/operate in one or more LPSes



- A space may sub-divide another space
- One can define a large number of spaces

A program

- Consists of a coordinator (main program) and a set of tasks
  - The coordinator reads/parses command line arguments, manages task execution environments, binds environment data structures to files, and executes tasks
- Tasks may be executed asynchronously when data dependence permits

execute(task: task-name;
environment: environment-reference;
initialize: comma separated initialization-parameters*;*
partition: comma separated integer partition parameters)

Tasks

Task "Name of the Task":
Define:
    // list of variable definitions
Environment:
    // instructions regarding how environmental variables of the task are related to rest of the program
Initialize <(optional initialization parameters)>:
    // variable initialization instructions
Stages:
    // list of parallel procedures needed for the logic of the algorithm the task implements
Computation:
    // a flow of computation stages in LPSes representing the computation
Partition <(optional partition parameters)> :
    // specification of LPSes, their relationship, and distribution of data structures in them

Task: define

```
Task MM {
        Define:
        a, b, c : 2D Array of Real double-precision;

        Compute-Stages:
      …
}
```

Task: Stages

- Declarative, data parallel syntax
- Parameter passing by reference, parameters must be task global or constant
- Types are inferred. Result is simple type polymorhism

Task: stages

```
Task MM {
        Define:
        a, b, c : 2D Array of Real double-precision;

        stages:
           multiplyMatrices(x,y,z){
                    do { x[i][j]=x[i][j]+y[i][k]*z[k][j]}
                             for i, j, in x; k in y
           }

      ...
}
```

Task: Partition

- Defines how the LPS should be divided into LPUs and the parts of the data structures distributed to those LPUs.

```
Task MM {
    Define:
    a, b, c : 2D Array of Real double-precision;
    …
    Partition (l, k ,q):
      Space A <2D> {
        c: block_size(k, l)
        a: block_size(k) replicated
        b: replicated, block_size(l)
      }
}
```

All kinds of partitions

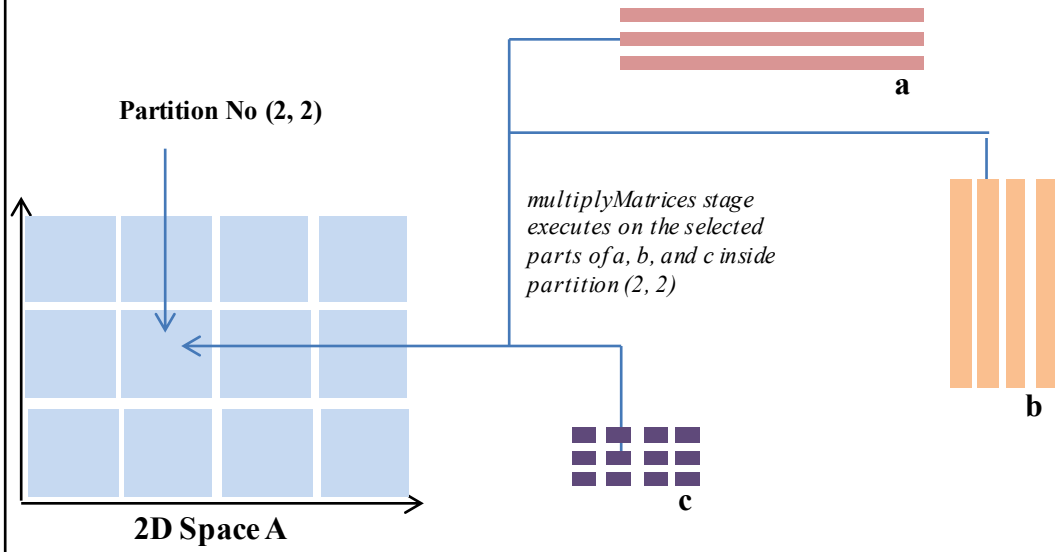
- **block(int i)**
- **stride(int i)**
- **block_stride(int i)**
- **block_count(int i)**
- Recursively sub-partition

Partition (L,K):

```
Space A <un-partitioned>{ a,b,c}
Space B <1D>  divides Space A partitions{
        a:<dim1> block_(L);
        d:<dim1> block(L);
}
Space C <1D> divides Space B partitions{
        a:<dim2> block(K);
        d:<dim2> block(K);
}
```

Variables can be partitioned

```
Task MM {
        Define:
        a, b, c : 2D Array of Real double-precision;

        Environment:
    …
        Initialize:
    …
        Compute-Stages:
    …
        Partition  (l, k ,q):
            Space A <2D> {
              c: block_size(k, l)
              a: block_size(k) replicated
              b: replicated, block_size(l)}
            }
}
```

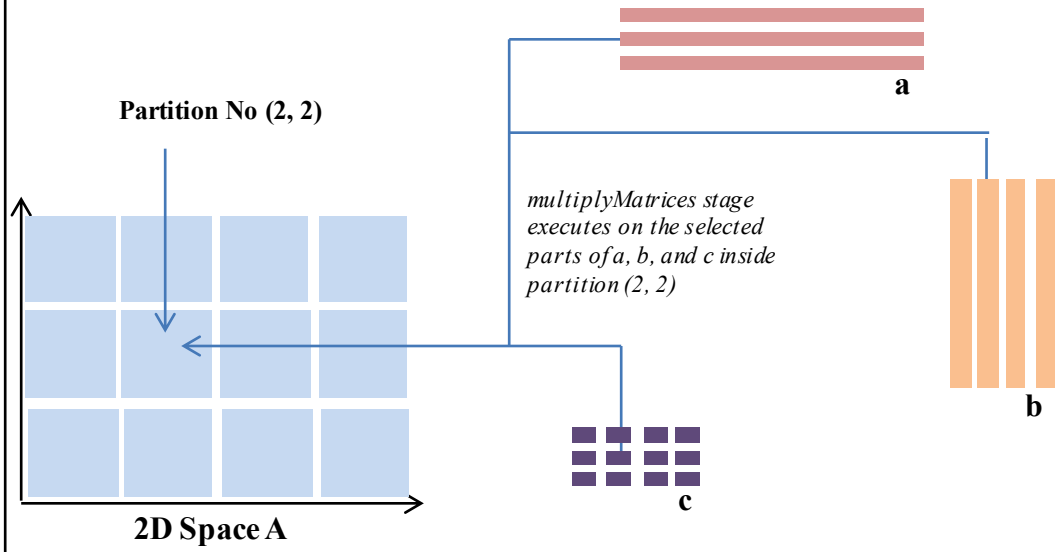


An Illustration of Space Partitioning for a Small Matrix-Matrix Multiply Problem. A block of rows of *a*, a block of columns of *b*, and a block of *c* are contained in the LPU corresponding to partition (2, 2).

Partitions define LPUs

Sub-partition

```
Task MM {
        Define:
        a, b, c : 2D Array of Real double-precision;

        Environment:
    …
        Initialize:
    …
        Compute-Stages:
    …
        Partition  (l, k ,q):
            Space A <2D> {
              c: block_size(k, l)
              a: block_size(k) replicated
              b: replicated, block_size(l)
              sub-partition <1d><unordered>{
                a<dim2>, b<dim1>:block_size(q)
              }
            }
}
```



An Illustration of Space Partitioning for a Small Matrix-Matrix Multiply Problem. A block of rows of *a*, a block of columns of *b*, and a block of *c* are contained in the LPU corresponding to partition (2, 2).

Partitions define LPUs

Effect of sub-partition

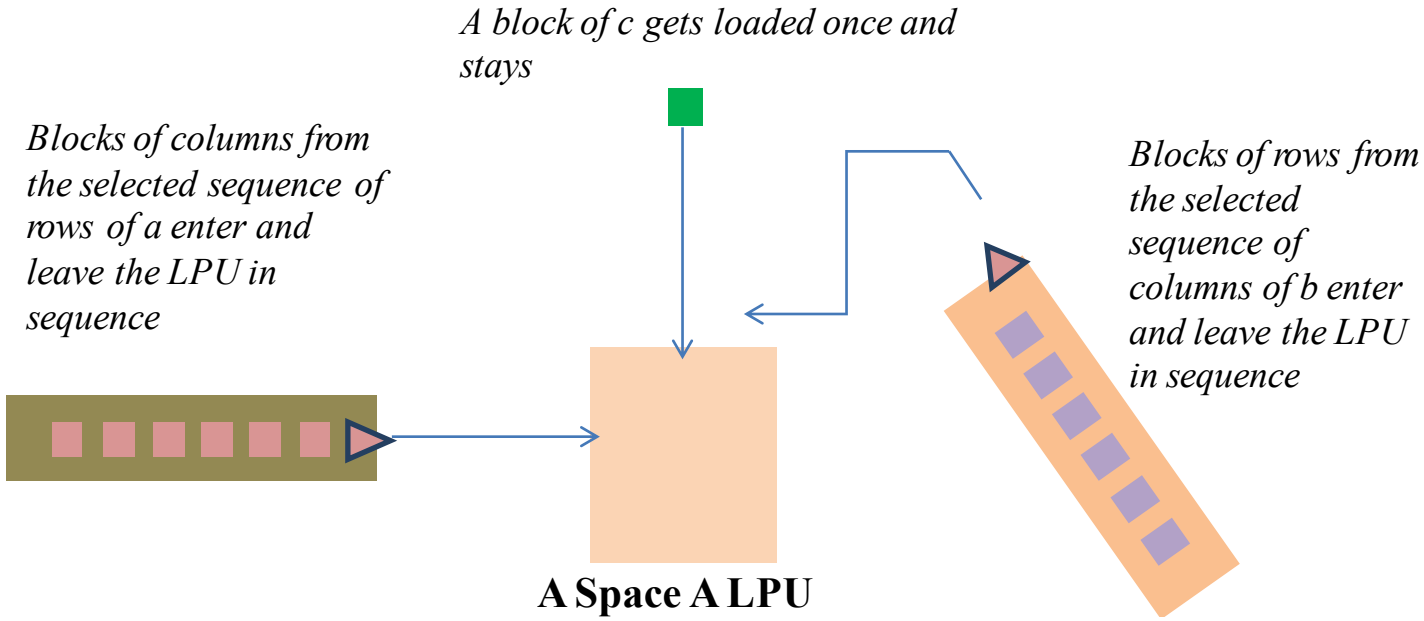


Figure 5: Incremental Data Loading in an LPU

Task: Computation

- "main" program of the tasks

```
Space A {
   stageY(args)
   Space B {
        …
        Stage C { …. }
        Stage D { …}
}
```

- All kinds of control flow constructs supported

Space transitions

- Space transitions may cause communication and/or synchronization
  - E.g., different partitions of data structures in different spaces may cause significant communication
- Space transitions may cause a flow control shift between physical layers of the hardware
  - E.g., execution shifts from cores to the GPU
- All the details are handled by the compiler and run-time

Task: computation

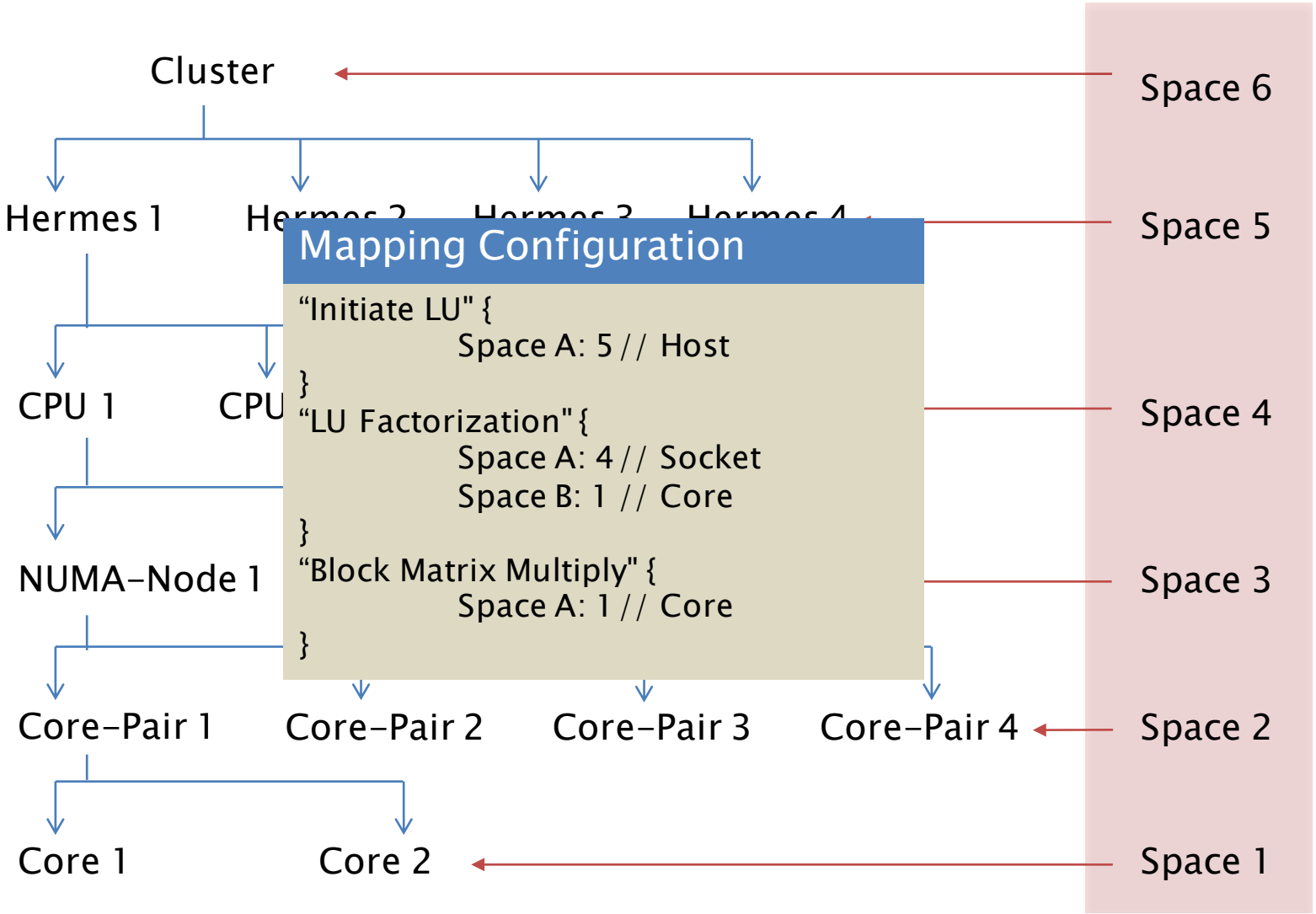
```
Task MM {
        Define:
        a, b, c : 2D Array of Real double-precision;

        stages:
          multiplyMatrices(x,y,z){
                    do { x[i][j]=x[i][j]+y[i][k]*z[k][j]}
                              for i, j, in x; k in y
          }
          computation:
          Space A {
                    multiplyMatrices(c, a, b);
          }
}
```

Block matrix multiply

```
1 Program (args) {
2         // create an environment object for the matrix-matrix multiplication task
3         mmEnv = new TaskEnvironment(name: "Block Matrix-Matrix Multiply")
4         // specify how external input/output files are associated with the environmental objects
5         bind_input(mmEnv, "a", args.input_file_1)
6         bind_input(mmEnv, "b", args.input_file_2)
7         bind_output(mmEnv, "c", args.output_file)
8         // execute the task
9         execute(task: "Block Matrix-Matrix Multiply"; environment: mmEnv; partition: args.k, args.l, args.q)
10 }
11
12 Task "Block Matrix-Matrix Multiply":
13         Define:
14                 a, b, c: 2d Array of Real single-precision
15         Environment:
16                 a, b: link
17                 c: create
18         Initialize:
19                 c.dimension1 = a.dimension1
20                 c.dimension2 = b.dimension2
21         Stages:
22                 // a single computation stage embodying the logic of the matrix-matrix multiplication
23                 multiplyMatrices(x, y, z) {
24                         do { x[i][j] = x[i][j] + y[i][k] * z[k][j]
25                         } for i, j in x; k in y
26                 }
27         Computation:
28                 Space A {
29                         // the stage has to be repeated for each sub-partition of Space A to have a block implementation
30                         // as opposed to a traditional one
31                         Repeat foreach sub-partition {
32                                 multiplyMatrices(c, a, b)
33                         }
34                 }
35         Partition (k, l, q):
36                 // 2D partitioning of space giving a block of c in each partition along with a chunk of rows of a
37                 // and a chunk of columns of b
38                 Space A <2d> {
39                         c: block_size(k, l)
40                         a: block_size(k), replicated
41                         b: replicated, block_size(l)
42                         // block-by-block flow of data inside a PPU is governed by the sub-partition specification
43                         Sub-partition <ld> <unordered> {
44                                 a<dim2>, b<dim1>: block_size(q)
45                         }
46                 }
```

To compile we must first map logical spaces to physical

Mapping


Cluster
Hermes 1
CPU 1
NUMA-Node 1
Core-Pair 1
Core 1
Core 2
Core-Pair 2
Core-Pair 3
Core-Pair 4
Mapping Configuration
"Initiate LU" {
Space A: 5 // Host
}
"LU Factorization" {
Space A: 4 // Socket
Space B: 1 // Core
}
"Block Matrix Multiply" {
Space A: 1 // Core
}
Space 6
Space 5
Space 4
Space 3
Space 2
Space 1

Project Status

Project Status

- Three compilers: multi-core, segmented (distributed memory MPI plus multi-core), and hybrid (distributed memory MPI, multi-core, GPGPU)
- Minimal optimization done. Following get it right then make it fast approach.
- Collecting base-line results for 5 applications: MM, LuF (2 versions), Integer Sort, finite difference, Monte Carlo.
- Hybrid GPU compiler compiled first codes last month
- Language features and syntax will evolve at the same time.

Agenda

- The problem – the five P's
- Current Practice
- The PCubeS Type Architecture
- IT – a PCubeS language
- Performance
- Conclusions and Future Work

Multi-core

- General
  - All results for double precision (64-bit)
  - Compiler: g++ with -O3 -mtune=native -march=native -mfpmath=sse
  - Sequential codes hand optimized and cache blocked
- Multi-core tests run on Hermes.
  - Four 16-core AMD Opteron 6276. 256GB memory total.
  - Core-pairs share a floating point unit. Thus only 32 floating point units.

Matrix Multiply

Time in seconds for sequential, speedup for others vs sequential

	1000	2000	4000	8000	10000
Sequential	2.1	18.1	167.4	1560.0	2302.0
OpenMP-32	7.8	3.5	3.1	4.0	4.3
OpenMP-64	6.6	4.4	3.2	3.4	2.4
IT-1	0.8	0.8	0.9	0.8	0.8
IT-4	3.0	3.2	3.4	3.3	3.3
IT-8	5.8	6.1	6.8	6.5	6.6
IT-32	17.8	19.6	24.2	24.4	24.4
IT-64	24.3	27.0	26.2	40.7	40.0

MPI/Multi-core

- Performance comparison is versus a hand coded/tuned sequential C program.
- Distributed memory tests run on Rivanna.
  - Rivanna is a Cray Cluster Solution connected by FDR (fourteen data rate) Infiniband. Nodes have Intel(R) Xeon(R) CPU E52670 processors. Each node has two processors with ten 2.5GHz cores each and each processor has 32K L1 data cache per core, 32K L1 instruction cache per core, 256K L2 cache per core and a 25MB shared L3 cache. Nodes 128GB memory.
- Compiler: GNU compiler with O3 optimization flag for all the tests.
- One MPI task per node. Internal parallelism using pthreads.

Block Matrix Multiply

Sequential	10K	1769	20K	11751

Block size 32 speedup

Cores	10K	Efficiency	20K	Efficiency
20	11.30	0.57	9.50	0.48
100	57.00	0.57	47.30	0.47
200	117.90	0.59	96.80	0.48
400	231.60	0.58	188.90	0.47

Block size 64 speedup

Cores	10K	Efficiency	20K	Efficiency
20	17.90	0.90	18.50	0.93
100	89.39	0.89	91.70	0.92
200	180.10	0.90	183.70	0.92
400	361.50	0.90	368.30	0.92

Hybrid GPU compiler

- Compiler has generated code for less than a month. Lots of work to be done still on optimization
- Bigred 2 at Indiana
  - Host: 16 core AMD Opteron(TM) Processor 6276
  - GPU: NVIDIA Tesla K20

Performance – MM

Kepler K-20	Time (S)			
	10KX10K	Slowdown	20KX20K	Slowdown
Handwritten	21.4		*171.2*	
IT - one GPU	126.4	5.91	983.6	5.75
IT - four GPUs	32.9	1.54	251.4	1.47

Notes:
1) 20K time is an estimate, 8X 10K time. 20K will not fit on card.
2) IT time is better than 50% of the students in parallel computing class
3) Same code on all platforms!
4) Handwritten is ~100GF double precision

Agenda

- The problem – the five P's
- Current Practice
- The PCubeS Type Architecture
- IT – a PCubeS language
- Performance
- Conclusions and Future Work

Take away messages

- Machine hierarchies are getting deeper
- The type architectures and programming languages must reflect the physical machine structure
- PCubeS/IT models and implements a hierarchically nested machine model

Take away

- IT is a combined task/data parallel language
- IT separates the specification of the computation from
  - The physical layer on which it executes
  - The partitioning and mapping of the data to physical resources
- The IT compiler and run-time manage all communication and synchronization, as well as dealing with the heterogeneity of the layers

Compiler/Run-Time Status

- Compilers available for V0 language
  - Multicore
  - Distributed memory MPI with multicore
  - Now generating code, but not ready for distribution: distributed memory MPI with multicore and CUDA.

Future Work

- Results are promising yet still preliminary
- Need to expand the set of codes (we have five currently) AND
  - Extend scale significantly
  - Examine the tuning parameter space to determine whether PCubeS parameters lead to best performance, e.g., block size
- Compiler/run-time performance bugs need to be worked out

Other control flow constructs

do in sequence {statement+}
for $index in Range-Expression
step Step-Expression
do in sequence {statement+}
while Boolean-expression
If (Boolean-expression) {statement+}
***Repeat** Boolean-expression { nested sub-flow }*
***Where** Boolean-expression { nested sub-flow }*
***Epoch** {nested stages accessing version dependent data structures }*