Proposal for Interfacing NPAC PetaSim V0.3 and UMD Emulators

Department of Computer Science
University of Maryland, College Park

Northeast Parallel Architectures Center
Syracuse University

Points of Contact :

Joel Saltz
Alan Sussman
Tahsin M. Kurc
Mustafa Uysal

Computer Science Department
University of Maryland
College Park, MD 20742-3255

{saltz,als,kurc,uysal}@cs.umd.edu

Geoffrey C. Fox
Yuhong Wen

Northeast Parallel Architectures Center (NPAC)
Syracuse University
111 College Place,
Syracuse, NY 13422–4100

{gcf,wen}@npac.syr.edu

1 Introduction

This report describes the plan to interface two tools, PetaSIM (developed in Northeast Parallel Architectures Center, Syracuse University) and Application emulators (developed in Computer Science Department, Univ. of Maryland at College Park), developed in the framework of project *Performance Prediction and Modeling of Compute and Data Intensive Applications on Current and Future High Performance Architectures.*

PetaSIM is a performance estimator aimed at supporting the conceptual and detailed design phases of parallel algorithms, systems software, and hardware architectures. PetaSIM is aimed at a middle ground - half way between detailed instruction level machine simulation and simple "back of the envelope" performance estimates. It takes care of the complexity - memory hierarchy, latencies, adaptivity and multiple program components which make even high level performance estimates hard. It uses a crucial simplification - dealing with data in the natural blocks (called aggregates, which can be a block of grid points or particles depending on the application) suggested by memory systems - which both speeds up the performance simulation and in many cases will lead to greater insight as to the essential issues governing performance. PetaSIM defines a framework in which the user provides description of the computer and problem and the primitive costs of I/O, communication and computation. The basic idea in PetaSIM is to use a sophisticated spreadsheet. Each cell of the spreadsheet represents a memory unit in the computer (disks, main memory, cache, network etc.) and generated from the machine description. User speficies a strategy called execution script to move aggregates through the cells. Some special cells have CPUs attached to them. An aggergate is computed when it lands on a CPU cell. The execution script defines the behaviour of an application on the given parallel machine.

An application emulator is a program which, when run, emulates the data access and computation patterns of the target application. It provides a parameterized model of the application running on the parallel computer. Only the parts of the application relevant to the performance prediction study are modeled, parameterized and emulated. This reduces the simulation time considerably. Since processing of the input data is emulated, synthetic input data, which can be generated in a controlled and parameterized way, can be used. The parameter adjustments make it possible to scale the application for large scale machines. In addition, parameterized model of the application allows generating different application scenarios. In this work, application emulators will be used to provide "execution scripts" for data intensive applications for PetaSIM performance estimator.

2 PetaSIM – A Performance Estimator for Parallel Hierarchical Memory Computers

Performance estimator PetaSIM is aimed at supporting the (conceptual and detailed) design phases of parallel algorithms, systems software and hardware architecture. Originally this was designed as a result of the two week long workshops - PAWS and PetaSoft - aimed at understanding hardware and software architectures ten years from now when Petaflop (10^{15}) scale computing can be expected. It was clear from these meetings that the community needed better aids for performance estimation as the 8 groups (and 8 different machine designs) present found it difficult to compare designs and in particular differed by a factor of million in estimating performance of even a set of extremely simple algorithms - the PetaKernels – on their new designs. The most sophisticated PetaKernel was a regular finite difference problem solved by simple Jacobi iteration which is of course well understood. These workshops emphasized the need for tools to allow one to describe complex memory hierarchies (present in all future and most current machine designs) and the mapping of problems onto them in a way that allows reliable (if high level) performance estimates in the initial design and rapid prototyping stages.

PetaSIM is aimed at a middle ground - half way between detailed instruction level machine simulation and simple "back of the envelope" performance estimates. It takes care of the complexity - memory hierarchy, latencies, adaptivity and multiple program components which make even high level performance estimates hard. It uses a crucial simplification - dealing with data in the natural blocks (called aggregates) suggested by memory systems - which both speeds up the performance simulation and in many cases will lead to greater insight as to the essential issues governing performance. We motivate and illustrate the design of PetaSIM by the well known formulae for parallel performance of simple regular applications on nodes without significant memory hierarchy. Then one finds,

$$SpeedUp = Number\ of\ Nodes/(1 + Overhead) \tag{1}$$

where the $Overhead$ is proportional to $(GrainSize)^{-g}(t_{comm}/t_{calc})$ and in this case natural data block size is the $GrainSize$ or number of data points in each node. The power g measures edge over area effects and is $1/d$ for a system of geometric dimension d. (t_{comm}/t_{calc}) represents a ratio of communication to compute performance of the hardware. Such a formula shows the importance of identifying natural data block and how such high level analysis allows one to understand the relation of performance to memory sizing, I/O and compute capabilities of the hardware. PetaSIM generalizes this "back of the envelope" estimate to more complex problems and machines. It also includes not just primitive messaging performance (node to node as used in above estimate) but also collective (such as multicast) mechanisms which are present in most applications but ignored in many simple analyses.

PetaSIM defines a framework in which the user provides description of the computer and problem and the primitive costs of I/O, communication and computation. The basic idea in PetaSIM is to use a sophisticated spreadsheet. Each cell of the spreadsheet represents a memory unit in the computer (disks, main memory, cache, network etc.) and generated from the machine description. User specifies a strategy called "execution script" to move aggregates through the cells. Some special cells have CPUs attached to them. An aggregate is computed when it lands on a CPU cell. The execution script defines the behavior of the application on the given parallel machine.

2.1 Current Specification of PetaSIM

There are two parts of descriptions in the PetaSIM performance estimation system, architecture description and application description. The most general computer architectures can be specified using the PetaSIM *nodeset, linkset* while the applications can be specified using *dataset, distribution* objects.

A *nodeset* is a collection of entities with current types allowed as:

- memory with cache (with flushing rules) as special case
- disk which is essentially same as a memory.
- CPU where results can be calculated
- pathway such as a bus, switch or network cable which concentrates data

A *linkset* connects *nodesets* together in various ways. *distributions* specify the horizontal (geometrical) connectivity of *nodesets* and *linksets*. Typically these are arranged in a natural default for the classic homogeneous architectures. The default mapping is inferred from sizes of *nodesets* and done in a simple one-dimensional block fashion. The vertical (flow of information) connectivity in the architecture is specified in the execution script with defaults implied in architecture specification.

The application is specified by a *dataset* object whose implementation is controlled by a *distribution* object that specifies classic HPF style geometric decomposition across memories and CPU objects. The computation is specified by the execution script, which also specifies data movement.

nodeset, linkset, dataset, and *distribution* objects are Java classes that are subclassed as necessary to give particular special cases with particular capabilities. They have methods that are defaulted for simple cases but can be overridden for complicated cases.

1. *nodeset* Object Structure in PetaSIM Estimator

   nodeset has the following properties:

   - **name:** one per *nodeset* object
   - **type:** choose from **memory, cache, disk, CPU, pathway**
   - **number:** number of members of this *nodeset* in the computer architecture
   - **grainsize:** size in bytes of each member of this *nodeset* (only relevant for memory cache or disk
   - **bandwidth:** this is maximum bandwidth allowed in any one member of this *nodeset*
   - **floatspeed:** performance on floating point arithmetic specified as a time to do a single operation for entities in cache. Only used by a CPU
   - **calculate():** method for CPU nodesets that performs computation implied by **floatspeed** and other architectural features.
   - **cacherule:** controls persistence of data in a memory or cache
   - **portcount:** number of ports on each member of nodeset
   - **portname[]:** ports connect to linksets and a member of a nodeset has one or more ports – each of which has a name. A port corresponds to a class of connections and depending on number of members involved, a given port can correspond to multiple connections
   - **portlink[]:** name of linkset connecting to this port

2. *linkset* **Object Structure in PetaSIM Estimator**

   A basic *linkset* has the following properties. A derived linkset is gotten by concatenating several basic *linksets* together. Derived linksets could be specified by special scripts or just written directly in Java.

   - **name:** one per *linkset* object
   - **type:** choose from **updown, across**. If **across**, this is a network of given **topology**, linking members of a single *nodeset*. If **updown**, this is a link between two different *nodesets*
   - **nodesetbegin:** name of initial *nodeset* joined by this *linkset*
   - **nodesetend:** name of final *nodeset* joined by this *linkset*. **Nodesetend** and **nodesetbegin** are identical if type is **across**
   - **topology:** used for **across** networks to specify linkage between members of a single *nodeset*

- **duplex:** choose from **full** or **half**. If **half** only allow transmission from **nodesetbegin** to **nodesetend**. If **full** allow either direction with bandwidth limiting sum of both directions.
- **number:** number of members of this *linkset* in the computer architecture
- **latency:** time to send zero length message across any member of this *linkset*
- **bandwidth:** this is maximum bandwidth in bytes per second allowed across any link in this *linkset*. Time T_{lk} to transfer information from nodeset l to nodeset k is expressed as *latency + bandwidth* . number of bytes. Here l refers to **nodesetbegin** and k to **nodesetend**
- **send():** method that calculates implications of sending information through given *linkset*. For a derived *linkset*, this method can include multiple references to properties and methods of basic *linksets* and *nodesets*.
- **distribution:** name of geometric distribution controlling this *linkset*

3. *distribution* **Object Structure in PetaSIM Estimator**

   distribution has the following properties:

   - **name:** one per *distribution* object
   - **type:** choose from **block1dim, block2dim, block3dim** (can obviously add more to this in analogy with HPF) to specify geometrical structure of entity being distributed. Note most computer architectures are implicitly done as one dimensional block *distribution*

4. *dataset* **Object Structure in PetaSIM Estimator**

   dataset has the following properties:

   - **name:** one per *dataset* object
   - **type:** choose from **grid1dim, grid2dim, grid3dim**, specifies type of *dataset*
   - **bytesperunit:** number of bytes in each unit. If 5 field values at each grid point and double precision used, then **bytesperunit** is 40
   - **floatsperunit:** update cost as a floating point arithmetic count. Differences between double or single precision , should be reflected in values of *CPUnodeset.floatspeed* and *dataset.bytesperunit*
   - **operationsperunit:** operations in each unit. A dataset contains *totalsize* of units, *operationsperunit* then reflects the operations in each unit for the *CPUnodeset* to calculate the computing time.
   - **update():** method that updates given *dataset* which is contained in a CPU *nodeset* and with a grainsize controlled by last memory *nodeset* visited.
   - **transmit():** method that calculates cost of transmission of *dataset* between memory levels including either communication (between distributed nodes) or movement up and down hierarchy. Note classic grid problems are assumed to be done using ghost cells so that edges of regions transmitted.

5. **Computation/Communication Instructions (Execution Script)**

   Much of execution is controlled by methods in *nodeset, linkset* and *dataset* objects. Some typical additional commands that implicitly invoke these methods are:

- ◦ **send** DATAFAMILY **from** MEM-LEVEL-L **to** MEM-LEVEL-K
  Here DATAFAMILY is a *dataset* specified by name
  MEM-LEVEL-K, MEM-LEVEL-L are *nodesets* labeled by name which must be linked by a *linkset*.
- ◦ **move** DATAFAMILY **from** MEM-LEVEL-L **to** MEM-LEVEL-K
- ◦ **Use distribution** DISTRIBUTION **on** NODESET1,,LINKSET1,,DATASET1
- ◦ **compute** DATAFAMILY-A,DATAFAMILYFAM-B .. **on** MEM-LEVEL-L
- ◦ **synchronize** synchronizes all processors (loosely synchronous barrier). Pipelining is stopped by this. Otherwise it is assumed

The difference between the send and move command is that send command will send the data from source node to the destination node, and then after the computing, send the data back to the original source. While move command will just move the data from source to destination, which does not have that "loop" operation.

2.2 Current Status of PetaSIM

A prototype of PetaSIM performance estimation system has been implemented in NPAC. It's based on the architecture description (*nodeset* class and *linkset* class) and application description (*dataset* class and *distribution* class, plus *execution script* to describe the operations on the *dataset*) as we discussed above. In the current version,

- PetaSIM has the ability to describe the target machine's specifications and memory hierarchy using *nodeset* and *linkset* objects. Each *nodeset* corresponds a level in the memory hierarchy. However, execution script (computation/communication instructions) does not provide a mechanism to specify specific elements in a nodeset. Currently, PetaSIM provides performance estimations for moving data between levels within the memory hierarchy.

- an application may contain a set of datasets, each containing *totalsize* units of data. In the assumption of PetaSIM, all computation is supposed to be in block-block form with both:

  - "real" blocks of nonzero size and
  - the "virtual zero size blocks formed by for instance ghost cells around the edge of domain

  The blocks are formed from *dataset* objects with a particular distribution on a particular memory *nodeset*. Each statement in the execution script will operate on a specific dataset, for examples: **move** one dataset from one *nodeset* to another *nodeset*, or **compute** one dataset on a CPU *nodeset*.

- five different kinds of statements are supported for execution scripts; *send, move, Use distribution, compute* and *synchronize* commands. These commands are enough to express the abstractions of operations in the applications, data movement and computation.

- application specific dependencies between different operations are not fully supported in execution script. The dependencies between *send* or *move* operations and *compute* operation are implicitly supported. That is, a compute operation on a dataset is performed after move/send operations on this dataset is completed. However, dependencies (due to application characteristics) between different move/send operations on elements of a nodeset, or between nodesets are not fully supported. Only the dependencies between move/send operations between two nodesets that are connected by a linkset are preserved. In the current version, commands are executed in the order they appear in the execution script. Re-ordering of commands (because of dependencies) is not supported.

3 Application Emulators

An application emulator is a program which, when run, emulates the data access and computation patterns of the target application. It provides a parameterized model of the application running on the parallel computer. Only the parts of the application relevant to the performance prediction study are modeled, parameterized and emulated. This reduces the simulation time considerably. Since processing of the input data is emulated, synthetic input data, which can be generated in a controlled and parameterized way, can be used. The parameter adjustments make it possible to scale the application for large scale machines. In addition, parameterized model of the application allows generating different application scenarios.

Following sections describe the application emulators that abstract and mimic the behavior of three data intensive applications, *Titan* [2], *Pathfinder* [1], and *Virtual Microscope* [3].

3.1 Current Data Intensive Applications Suite

3.1.1 Titan

Titan [2] is a parallel shared-nothing database designed to handle satellite data. The input data of Titan are the sensor readings from AVHRR on NOAA-7 series satellites. An NOAA-7 satellite orbits the earth in a polar orbit. The sensor mounted on the satellite sweeps earth surface gathering readings for different bands. Each such reading is associated with a position on earth in terms of longitude and latitude, and time of recording in universal coordinated time. In a typical operation of Titan, the user issues a query to specify the data of interest. A query defines a spatial and temporal bound. The data intersecting the query are retrieved from disks and processed to generate the output. The output is a 2-dimensional (2D) multi-band image generated by compositing the sensor readings.

Prior to the operation of Titan, the input data is partitioned and placed on the disks. In partitioning phase, sensor readings are grouped to form tiles, which are referred to here as *data-blocks*, to perform efficient retrieval of satellite data from disks for processing. Each such data-block contains equal number of elements to be processed. However, the spatial extend of the data-blocks differ. Data-blocks formed by grouping pixels corresponding to readings near poles have larger spatial extend than data-blocks near equator. In addition, there are more spatially overlapping data-blocks near poles than near equator. In the placement phase, data-blocks are distributed to disks by a declustering algorithm to achieve high parallelism during disk accesses.

The output is defined by superimposing a 2D grid on the spatial extend of the query window. This grid is partitioned among processors evenly into disjoint rectangular regions of equal number of pixels. First, processors are conceptually organized into a 2D mesh. Then, this mesh is superimposed on the output image to form rectangular regions. Each processor is assigned a single rectangular region and processes the data blocks whose spatial extend intersects the portion of image assigned to that processor.

When a query is received, a list of data-block requests for each processor is generated. The request list for a processor contains read requests for the data-blocks that are stored on the local disks of the processor and that intersect the query window. Processors perform retrieval, processing

```
while (not done) do
  1. Issue reads to local disks.
  2. Issue receives.
  3. Poll reads.
     if (some reads completed) then
        Map the data-blocks retrieved.
        if (mapped to other processor)
           Send the data-block.
        if (mapped to myself)
           Enqueue data-block for processing.
     endif
  4. Poll receives.
     if (some data-blocks received)
        Enqueue the data-blocks for processing.
  5. Poll sends.
  6. Process one data-block from enqueued data-blocks.
endwhile
```

Figure 1: Processing loop of Titan.

and exchange of data-blocks in a loop [2]. In this loop, each processor issues reads, sends, receives, and processes the retrieved data-blocks. In order to hide the latency of I/O and communication in this loop, non-blocking asynchronous read, send, and receive operations are used to overlap computation with I/O and communication. The basic units of I/O and communication are the data-blocks. That is, even if a data-block partially intersects with the query window, full data-block is retrieved from disk. Similarly, even if the data-block partially intersects the portion of the output image in a processor, full data-block is sent to the corresponding processor. A high-level description of the processing loop is given in Fig. 1.

3.1.2 Pathfinder

Pathfinder [1] performs calibration for instrumental drift, topographic correction, masking of cloudy pixels, registration of sensor readings with locations on the ground and compositing of multiple pixels corresponding to same ground location. Input data of Pathfinder is the same as that of Titan. The output is twelve-band Interrupted Goode Homolosine 2D image of the world at 8km resolution.

The operation of Pathfinder on a parallel machine employs a client/server approach. A subset of nodes of the parallel machine behave as servers (I/O nodes), while the remaining nodes constitute the clients (compute nodes). Servers are responsible only for reading data from disks and sending them to clients. Clients perform operations on the data received from servers.

The input data is partitioned as in Titan and distributed to the local disks of server processors. The 2D output image is partitioned into rectangular regions and each rectangular region is assigned to a client processor. If the portion of the output data assigned to a compute node is too large to fit into memory, it is partitioned into blocks and stored to local disks on the compute node.

Pathfinder processes all of the input blocks (which can be considered as issuing a global query covering whole input space in Titan). During the execution of Pathfinder, each server node reads all blocks from its local disks and sends them to respective clients whose output image partition intersects the spatial extend of the blocks. The server processors issue non-blocking read and send operations to hide their latencies. The basic unit of I/O and communication is a data-block as in Titan. The compute nodes issue non-blocking receives to overlap overhead of communication with computation. In case of out-of-core output data, when a block is received from the server node, the compute node issues reads to local disks to retrieve corresponding portions of the output data.

3.1.3 Virtual Microscope

The Virtual Microscope [3] is a software system to emulate a High Power Light Microscope. The input data for Virtual Microscope are the digitized images of full microscope slides under high power, i.e., highest magnification level. Each slide consists of several focal planes. Output is a multi-band 2D image of an area on a focal plane under a magnification level.

The Virtual Microscope employs a client/server architecture. The server part of the software runs on the parallel machine. In a typical operation, multiple clients queries to the server. A query specifies a slide, a focal plane in the slide, an area of interest on the focal plane, and a magnification level. When a query is received, blocks that intersect with the query are retrieved from the disks. Each retrieved block is subsampled to generate a portion of the image at the required magnification. After processing each block, the corresponding image portion is sent to the client. The client stitches the image parts received from the server to form the full image.

Prior to the operation of the parallel server, input data is partitioned and distributed to the disks on the parallel machine. In the partitioning phase, adjacent pixels on each focal plane are grouped to form rectangular data-blocks of equal number of pixels. The data-blocks are declustered among the disks in the parallel machine.

Since client stitches image parts received from the server processors, output images corresponding to queries submitted by clients are conceptually replicated on each server processor whose local data-blocks intersect the query window. As a result, in Virtual Microscope each processor behaves as a stand alone server. That is, when a query is received each processor retrieves blocks, which intersect with the query, stored on the local disks, processes them and sends the resulting image patches to the client. There is no communication and coordination between server processors. Different processors can operate on different queries at the same time. In order to overlap disk accesses with processing of blocks and sending image patches to the client, non-blocking read operations are employed.

3.2 Emulating Titan, Pathfinder and Virtual Microscope

In order to emulate the data access and processing patterns of these three applications, we need to model and emulate three components; *input*, *output*, and *processing loop*:

3.2.1 Input

Communication, I/O and processing patterns in these applications are all data dependent. Attributes of input data such as number of data-blocks, spatial extend of each data-block, location of data-block in the disk farm affect these patterns. Emulators should generate synthetic input data (i.e., data-blocks) to emulate communication, I/O and processing patterns. A data-block is represented by four elements: *bounding rectangle*, *disk id*, *offset in the disk*, and *length in bytes*. The bounding rectangle represents the spatial extend of the data-block which affects the I/O patterns in Titan and Virtual Microscope and communication patterns in Titan and Pathfinder. For example, given a query, spatial extend determines whether a block is retrieved from local disk or not, hence affects the number of blocks and which blocks to be retrieved from the local disks in Titan and Virtual Microscope. Along with a partitioning of the output, spatial extend determines whether a block is involved in the interprocessor communication and destination processors in Titan and Pathfinder. Disk id, offset in the disk, and length in bytes are used to emulate access patterns to local disks.

The data-blocks should be distributed to the disks of the simulated parallel machine to emulate the parallel execution. In emulator, simple block cyclic assignment strategy has been employed to decluster even large number of blocks in short time. Blocks are distributed to disks in a round-robin fashion so that each processor and each disk attached to a processor have equal number of blocks. When a block is assigned to a disk, its *disk id*, *offset*, and *length* values are updated accordingly. Number of processors and number of disks attached to a processor are given as parameters to the declustering algorithm.

3.2.2 Output

The partitioning of output data affects communication and processing patterns in Titan and Pathfinder. In the emulator, the output data is represented as a rectangle with spatial coordinates in 2D space. Processors are assumed to be organized into a 2D mesh. This mesh is superimposed on the output rectangle to create rectangular subregions of equal area. The number of processors in each dimension of 2D processor mesh is given as parameter. In this way, different partitioning of the output data can be produced. In case of out-of-core output data structure, each portion of output should be further divided into blocks and distributed to local disks as in Pathfinder.

3.2.3 Processing loop

An important characteristic of processing loops of the three applications is that all reads, sends, receives are non-blocking operations and their execution may overlap with each other and processing of individual blocks. In spite of this dynamic nature, there exists dependencies between operations. First of all, send operations depend on the completion of read and the data-block retrieved. In Virtual Microscope, a completed read operation initiates a compute operation, which is followed by a send operation to the client. In Titan and Pathfinder, a completed read operation can initiate send operations to some other nodes if the retrieved data-block intersects with their output regions. Moreover, in Titan, if the data-block intersects with the local output image, then it is enqueued for processing locally. Initiation of receive operations in a processor does not depend directly on the reads or sends in that processor. However, completion of a receive operation in Titan and Pathfinder depends on the send (hence read) operations on other processors destined for this processor. When a data-block is received it is enqueued for processing. The dependency relations between different operations in each application are shown in Fig. 2 for a data-block. Solid lines indicate the dependencies between tasks in a processor while dashed lines indicate dependencies between tasks in different processors. The processing loop of emulator should retain non-blocking characteristic of operations as well as should pay attention to the dependencies between these operations.

3.3 Current Status of Emulators

Three emulators have been implemented to abstract and mimic the behavior of the three applications described above. The current status of the emulators are:

- The algorithms for the applications assume distributed-disk distributed-memory architectures in which each processor has a number of local disks attached. Emulators assume the same architecture class. Number of processors and number of disks per processor can be given as command line parameters to parameterize the machine. The memory hierarchy and CPU on each processor are not modeled.
- Emulators can generate set of events in accordance with the application specific dependencies between operations (Fig. 2).
- Computation of a single block is modeled at a coarse grain. Each block is assumed to take the same amount of processing time.
- Only the server side of the Virtual Microscope is modeled. That is, only read, compute operations are generated as events. Sending to client is not modeled.
- Currently, in-core data structures are assumed for output data both in Titan and Pathfinder applications. Output data model is only used to emulate the message passing patterns in Titan and Pathfinder since its partitioning among processors affect those patterns. As explained in the previous section, a 2D mesh of processors is assumed, and number of processors in each dimension can be given as command line parameters. In addition, in emulator for Pathfinder, number of I/O nodes and compute nodes can be given as command line parameters.
- A set of parameters can be input as command line parameters to parameterize the input. For example, number of days for Titan and Pathfinder for satellite data and number of queries for Virtual Microscope.

4 Interface between PetaSIM and Emulators

Interfacing PetaSIM and emulators requires the following:

- a common target machine description format (nodeset, linkset),
- a common application description format (dataset, execution script).

4.1 Target Machine Description

In PetaSIM, *nodeset* and *linkset* objects are used to define the computer architecture and memory hierarchies within each architecture. PetaSIM reads architecture description from 2 files, which correspond to nodeset and linkset descriptions. The common target machine description will be based on the machine description currently used by PetaSIM.

4.1.1 Changes to PetaSIM

PetaSIM uses a nodeset or linkset as single entity. There is no mechanism to identify individual members of a nodeset or linkset. Data access and computation patterns in the applications (hence, application emulators) in this work require access to individual members of a nodeset or linkset. PetaSIM should extract this information from current machine description format in a consistent manner. For example, assume that two nodesets nodeset0 and nodeset1, which are connected through a linkset (linkset0), are used to represent CPUs and caches. If there are N members in these nodesets, then each member can be numbered between 0 and $N-1$, and members numbered with the same numbers are assumed to be connected to each other directly through linkset0.

4.1.2 Changes to Emulators

Emulators should be modified accordingly to read from the same files as PetaSIM reads and to extract necessary machine information to generate set of events as execution script.

4.2 Application Description

Currently, PetaSIM reads application description from 3 files which correspond to dataset, distribution and execution script. Emulator will provide the dataset descriptions and execution script. For the applications targeted in this work, distribution can be implicitly represented in execution script. The same dataset definitions used by PetaSIM can be used to describe input modeling of the applications. For example, a dataset object can be used to describe a data-block used in Titan, Pathfinder or Virtual Microscope.

4.2.1 Changes and Additions to Execution Script Format

- **Dataset Hierarchy** In some applications, there may be a hierarchy of datasets. An application may contain a set of *dataset*, and a dataset may contain a bunch of *sub-datasets*. A statement in execution script may operate on a specific *sub-dataset* or directly operate on a parent dataset. For example, if an application has *dataset1, dataset2,* and *dataset3*, and *dataset1* contains *sub_data1, sub_data2, sub_data3* and *sub_data4*. Then, the execution script may look like:

  move sub_data1 **from** memory_node **to** cpu_node
  compute sub_data1 **on** cpu_node
  `# which works on sub_dataset`

  move dataset1 **from** disk_node **to** cache_node
  move dataset1 **from** cache_node **to** memory_node
  `# which automatically operates on sub_data1, sub_data2, subdata3 and sub_data4`

- **Extensions to execution script library** Currently, execution script format supports five different types of statements, *send, move, Use distribution, compute* and *synchronize* commands. These operations are basic statements to express data movement between different nodesets and perform computations on datasets. However, for large scale applications and large scale machines, the execution script can be prohibitively long when only these basic set of statements are used. An extension to these statements will be considered to build an execution script library, which contains aggregates of data movement and computation in a function call. As a result, it will make the execution script of applications shorter and easier to read and understand.

  For example, addition of a new statement `calculate` as follows can reduce the size of the execution script:

  calculate dataset on CPU_nodeset

  which will expand into the following operations

move dataset **from** disk_nodeset **to** disk_controller
move dataset **from** disk_controller **to** cache_nodeset
move dataset **from** cache_nodeset **to** memory_nodeset
compute dataset **on** CPU_nodeset

- **Dependencies Between Statements in Execution Script** Some applications, like the applications in this work, may induce dependencies between different operations. A set of implicit dependencies exist in the current definition of PetaSIM. For example, a compute operation on a dataset on a nodeset cannot start before send/move operations on the same dataset finish. Also, due to dynamic nature of some applications the order of execution of operations may depend on the input data and execution of other statements, hence may change under different machine characteristics. For example,

  s1: move data1 from node1 to node2
  s2: mode data2 from node3 to node2
  s3: move data3 from disk2 to node2 # `according to the arrival of data1`
  s4: mode data4 from disk2 to node2 # `according to the arrival of data2`

  In this example, the order of executing statements s3 and s4 may depend on the order in which the statements s1 and 2 finish. If s1 finishes first, s3 may execute first, and if s2 finishes first, s4 may execute before s3.

  Execution script will be extended with a dependency notation to account for the dependencies between operations.

4.2.2 Generation of Execution Scripts

Currently, PetaSIM reads the execution script from a file, assuming that all statements of execution script already exist in the file. In order to interface PetaSIM and emulators, PetaSIM should take execution script input dynamically, and give an estimation. That is, emulators will generate the execution script statements according to the applications and then send to PetaSIM to give an estimation. This is also a way to try to avoid the long execution script of an application.

Emulators will send statements in execution scripts in sets. That is, the emulator will send a bunch of statements with proper indications of dependencies to PetaSIM and wait for request from PetaSIM for another bunch of execution statements. All statements which have inter-dependencies will be included in the same set of statements. There will be no dependencies between operations in different sets.

4.3 Changes to PetaSIM and Emulators

PetaSIM will be modified accordingly to receive execution scripts directly from emulators (through sockets possibly) in addition to reading from files. In addition, additions to internal mechanisms of PetaSIM will be done to take into dependencies given in the execution scripts.

In emulators, event generation format will be adapted to generate execution scripts that can be understood by PetaSIM. Design and implementation of interfaces will be done to write events to files for later use by PetaSIM and to use sockets for direct interaction with PetaSIM. In addition to modifications to emulator for interfacing with PetaSIM, we can anticipate further modifications such as incorporating out-of-core output modeling into emulators, design of emulators (by design of algorithms) for other architectures such as distributed shared-memory architectures in near future.

However, for the initial interfacing of PetaSIM and emulators we can use the application suite and emulators described in this document.

References

[1] A. Acharya, M. Uysal, R. Bennett, A. Mendelson, M. Beynon, J. Hollingsworth, J. Saltz, and A. Sussman. Tuning the performance of i/o intensive parallel applications. In *Proceedings of the Fourth Annual Workshop on I/O in Parallel and Distributed Systems, IOPADS'96*, May 1996.

[2] C. Chang, B. Moon, A. Acharya, C. Shock, A. Sussman, and J. Saltz. Titan: a high-performance remote-sensing database. In *Proceedings of the 13th International Conference on Data Engineering*, April 1997.

[3] R. Ferreira, B. Moon, J. Humphries, A. Sussman, J. Saltz, R. Miller, and A. Demarzo. The Virtual Microscope. In *AMIA97*, pages 449–453. American Medical Informatics Association, Hanley and Belfus, Inc., Oct 1997. Also available as University of Maryland Technical Report CS-TR-3777 and UMIACS-TR-97-35.

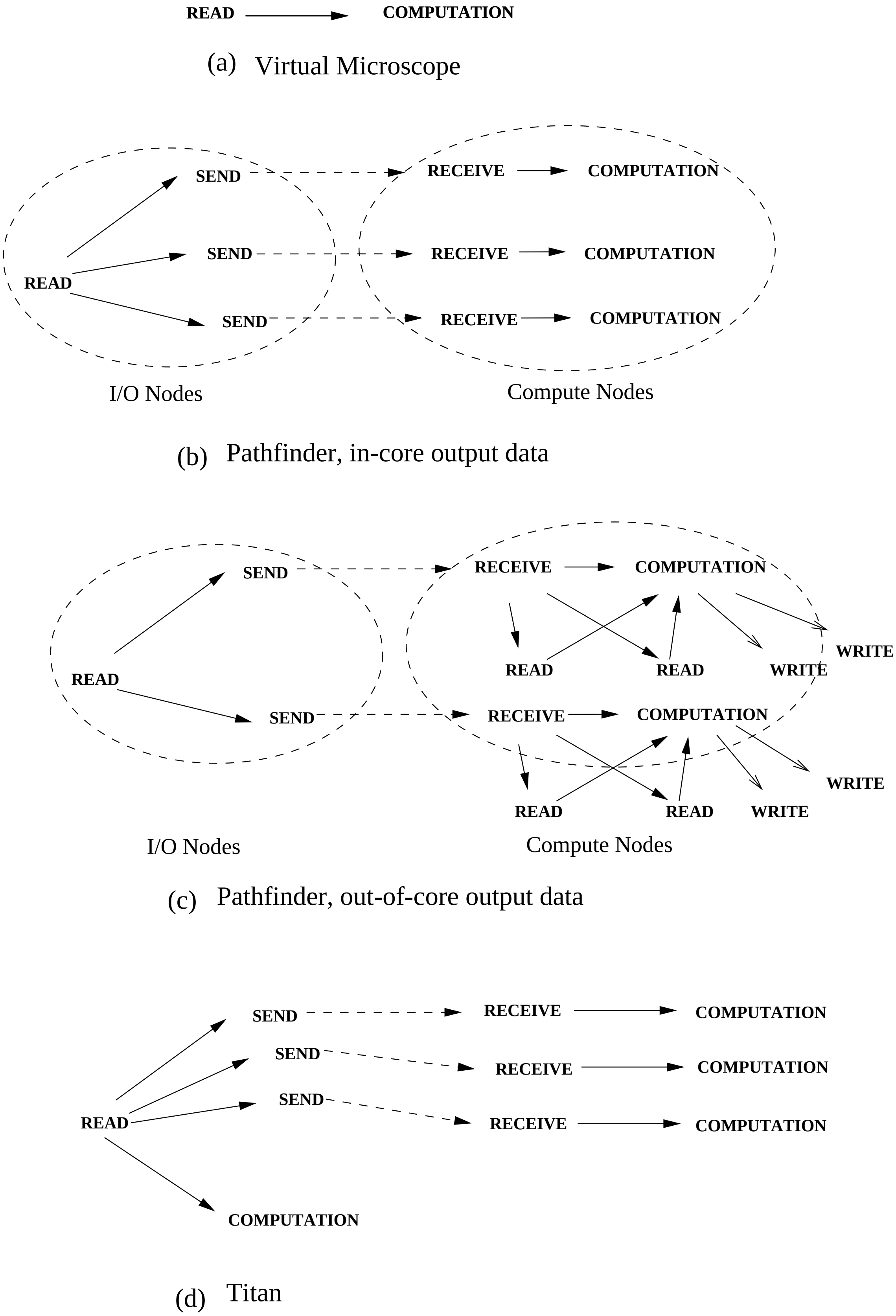

Figure 2: The dependency relation between operations in the processing loop. (a) Virtual Microscope (b) Pathfinder, in-core output data (c) Pathfinder, out-of-core output data (d) Titan.