

PNRSim: A Parallel Network RAM Simulator

John Oleszkiewicz Li Xiao

Department of Computer Science and Engineering
Michigan State University
East Lansing MI 48824 USA

{oleszkie, lxiao}@cse.msu.edu

Technical Report MSU-CSE-04-13

April 14, 2004

Abstract

PNRSim ("Parallel Network RAM Simulator") is a simulator written to test the performance characteristics of proposed parallel network RAM algorithms. PNRSim models a computing cluster system, parallel jobs running on that system and parallel network RAM servers coordinating the use of parallel network RAM. This technical report is an introduction to PNRSim and is specifically written for those interested in using PNRSim as a tool to test new parallel network RAM strategies. It outlines the usage of PNRSim (e.g. input and output), the models PNRSim uses, and the implementation of these models.

1 Introduction

PNRSim ("Parallel Network RAM Simulator") is a discrete event driven simulator created to model a cluster environment running a variety of parallel jobs for the purpose of measuring performance gains yielded by proposed Parallel Network RAM algorithms. This technical report is intended to serve as an introduction to PNRSim for those interested in using or altering PNRSim.

This paper is organized as follows. Section 2 describes how to use PNRSim, going over PNRSim input and output. Section 3 describes PNRSim implementation details in an effort to provide a general overview to the programmer who wishes to understand and change PNRSim. Section 4 describes the models used to simulate clustering systems in PNRSim. Section 5 describes the models used to simulate proposed Parallel Network RAM schemes.

2 Using PNRSim

This section describes how to use PNRSim, including the input and output format PNRSim uses. This section does not describe how to obtain, compile, or install PNRSim.

2.1 Input

This section describes PNRSim input. This includes command-line options, configuration file parameters, and workload (trace) files.

2.1.1 Command-line Options

PNRSim has several command-line options. To get a listing of the available options and their meaning, invoke PNRSim with the "--help" option.

The following is a list of the available command-line arguments.

- tracefile** Sets the path of the workload file PNRSim will be reading. The default is "default.swf".
- configfile** Sets the path of the configuration file PNRSim will be reading. The default is "sim.cfg".
- simpack-report** Causes PNRSim to produce more simulation output. This output comes directly from the Simpack simulation library.
- hardware-report** Output information about the hardware of the simulated network. This is primarily useful for debugging.
- verbose** Currently has no effect.
- help** Displays a help message describing how to use PNRSim.

2.1.2 Configuration File

PNRSim reads a text-based configuration file that defines simulation parameters. Each parameter is listed on a line. Each parameter starts with the parameter name, followed by whitespace (tabs or spaces), followed by the parameter value. The type of parameter value will depend on the parameter. The configuration file is not case-sensitive. Hashes (or semicolons) mark the beginning of comments.

The following is a list of accepted parameters:

System The system name. Used for output purposes only. The value be any string without whitespace.

MetricsPeriod Controls how often PNRSim produces output that indicates simulator progress and how often samples are taken for statistical purposes. The value may be any floating point number.

Note that the current version of PNRSim cannot guarantee output or sampling will occur at absolute times. PNRSim will print output as close as it can to the period described. For instance, if no event happens for 1000 time units, and the period is set for 100, no output will be produced until update 1000.

Seed The random number seed.

The value may be any positive integer or zero. If the value is zero, the seed will be set using the current system time (in seconds).

Process The name of each process. The value may be any string (no spaces).

MemAccessRate The average amount of time (in seconds) between memory accesses for processes in the system. The value may be any positive floating point number. This value is used as a parameter in a formula that determines the frequency of page faults.

CommRate The average amount of CPU time (in seconds) between local communications among threads. The value may be any positive floating point number.

SynchRate The average amount of CPU time between global synchronizations of all threads in a job. The value may be any positive floating point number.

Scheduler Sets the global system scheduler to be one of the pre-defined scheduler types. The only scheduler currently available is a gang scheduler (GANG) - a scheduler that implements both time and space sharing. However, a simple space-sharing scheduler can be implemented by running the gang scheduler with a MPL of 1.

Priority Sets the policy on the priority queue the scheduler uses. Currently available are:

- FCFS - first-come, first-serve
- BESTFIT - the jobs that require the most PE's are scheduled first
- WORSTFIT - the jobs that require the least PE's are scheduled first

Packing Packing controls how the gang scheduler puts new jobs into the available time slices. That is, multiple time slices could possibly accommodate a new job, and some method must be used to choose among the available time slices.

Currently available packing schemes are:

- **FIRSTFIT** - packs new job in the first time slot found with adequate PE's available.
- **BESTFIT** - packs new job in the time slot with the least amount of adequate PE's available.
- **WORSTFIT** - packs new job in the time slot with the most PE's available.

MPL Defines the Multi-Programming Level for the gang scheduler. MPL defines the maximum number of time slices the gang scheduler may use. A value of zero indicates an infinite number of time slices may be used. A positive integer value sets the maximum.

PCS Sets the amount of time overhead incurred during each Parallel Context Switch. This is for use in the gang scheduler only. During this overhead time, no useful work is done. The value may be set to any positive floating point number.

MemAware Determines if the scheduler is "memory aware" or not. If the scheduler is memory aware, it will queue jobs when there is not enough physical memory to accommodate their memory needs. The scheduler does this even if enough processors are available. If a scheduler is memory aware, it is assumed the scheduler not only keeps track of current memory allocations on each PE, but also is perfectly aware of the memory requirements of each incoming job. This is unrealistic.

An implication of memory awareness is that no thread may ask for more memory than is physically installed in system RAM. If any thread does this, it is queued forever.

The value of this parameter may be "Yes" or "No".

Quantum This parameter sets the time quantum between parallel context switches. This is the length of a single time slice. This parameter is used only by gang schedulers. The value may be set to any positive floating point number.

NetRamPacking This parameter is currently unused.

Computer The base name of each computer (PE) in the system. A computer number will be appended to the base name in system output. The value may be any string with no spaces. This parameter is only used for output purposes.

Number The number of computers (PE's) installed in the system. The value may be any positive integer.

It is assumed that all computers are homogenous.

Netram Signals the system whether Parallel Network RAM will be used or not. The values for this parameter may be "Yes" or "No".

PNRStrategy If PNR is being used, this parameter sets which variety of PNR will be used. The different PNR algorithms may yield different results.

Currently available strategies include:

- **CENTRAL** - a single central manager receives all client requests and does all of the allocation and deallocation server requests.
- **CLIENT** - each client seeks network RAM on its own without coordination with other clients or managers.
- **MANAGER** - clients must contact local managers to allocate and deallocate network RAM. The servers acting as local managers are elected and which particular servers act as managers will vary.
- **BACKBONE** - a constant number of servers will act as managers. Clients may choose one manager among the multiple available managers. If only one manager exists, this strategy is equivalent to the **CENTRAL** strategy.

GlobalKnowledge If PNR is used, this parameter turns PNR servers' access to global information on and off. Specifically, if this parameter is turned on, servers have access to the current memory load of each computer in the system. Servers can use this information to more accurately select PNR servers for network RAM allocation. This information is, of course, impossible to obtain in the real world. However, this parameter can help determine how big a factor outdated information is playing in PNR strategies.

The values for this parameter may be "Yes" or "No".

Managers This parameter is specific to the "backbone" PNR strategy. The value may be any positive integer equal to or less than the number of PE's available in the system. This number will determine how many PNR managers are available on the system.

ServerThreshold This parameter determines the lowest acceptable level of idle RAM on a PNR server. If idle RAM goes below this number, the PNR server will not allow any more network RAM to be allocated. If idle RAM is above this number, managers may allocate network RAM as long as the allocation does not put the server under this threshold.

The threshold is a positive floating-point number representing Megabytes.

CPU This parameter is currently unused.

RAM Amount of RAM on each computer. The number is in megabytes. The value may be any positive floating-point number.

RPM The revolutions per minute each PE's disk rotates at. This number is used to calculate disk read/write times. This value may be any positive floating-point number.

Seek The average seek time of each PE's disk. This number is used to calculate disk read/write times. This value may be any positive floating-point number.

Transfer The rate at which data is transferred from disk to CPU on each PE. This number is used to calculate disk read/write times and is in MB/sec. This value may be any positive floating-point number.

Network Defines the network topology. The following are available topologies.

- Star - N links and 1 switch
- Wheel - $N + N/2$ links and 1 switch

- Bus - 1 link
- Connected - $N(N - 1)$ links

Bandwidth The bandwidth of each link in the system. The value is in Megabytes per second and may be set to any positive floating point number.

Latency The amount of time it takes data to traverse links in the system. The value may be set to any positive floating point number.

Delay The fixed amount of time it takes for switches to process packets on the system. The value may be set to any positive floating point number.

2.1.3 Workload Data

PNRSim reads trace files in Standard Workload Format (SWF). This format was defined by Dror Feitelson [1]. Generally speaking, a SWF trace defines a series of parallel jobs and their characteristics using a space delimited list. Semicolons denote comments.

Only some of the information in the SWF file is actually used. The following is a list of SWF fields in order of appearance in SWF files and a description of how they are used in the simulator.

Job Number A unique ascending number assigned to each job. This number does not have anything to do with the specific system's PID. This number is used in the simulator to differentiate processes.

Submit Time The time a job is submitted to the system. This is the time the scheduler becomes aware of the job. After this time, it is up to the scheduler to decide what to do with the job. For example, it could decide to immediately run the job or queue it. This number is used by PNRSim to create an arrival schedule.

Wait Time How much time the process had to wait on the traced system before running. This value is not used by PNRSim.

Run Time The time at which the job started running. This value is not used by PNRSim.

Allocated Processors The number of PE's allocated to the job. PNRSim treats this number as the absolute number of PE's that are required for the processes' successful execution. That is, it is assumed that the number of PE's required are given at job submittal time - a variable partitioning scheme [2]. The number of allocated processors is very important because it will determine the total number of threads used by the job.

Average CPU Time This value records the average amount of CPU time (in seconds) used by each thread of a job. PNRSim uses this value as the absolute amount of CPU time required for each thread for successful completion. This value is very important to the simulation.

RAM memory accesses are considered as part of CPU time. Note that CPU time does not include other events such as disk accesses or communication time. If a process is busy handling these events, it will not use CPU time.

Used Memory This value records the average amount of memory used by each thread (in kilobytes). PNRSim uses a static memory allocation model. It is assumed that this amount of memory is exactly the amount of memory required for use by the thread for its entire execution. At thread start time, the PE's operating system is informed of this value and memory allocation may take place normally.

Other Fields All subsequent values listed in the SWF file are read by PNRSim but ignored. Defaults should be left in as placeholders if creating a new SWF file from scratch.

2.2 Output

This section describes the output PNRSim produces. Currently, all output is put to STDOUT or to STDERR.

2.2.1 Simulator Progress Output

PNRSim periodically prints the current event occurring in the simulator to STDOUT. This is used as an indication of simulator progress. PNRSim uses the "MetricsPeriod" configuration parameter to determine how often this event should occur. Note that metrics period is also used to determine how often to sample some metrics.

2.2.2 Metrics Output

At the end of a simulation run, human-readable metrics are printed to STDOUT. If PNRSim terminates unexpectedly, it will attempt to print the currently recorded metrics to STDERR. No single metric is an authoritative measure of system performance [3]. So, PNRSim produces several metrics as output.

A file called "metrics.dat" is created by PNRSim during each execution but PNRSim does not write to it. Eventually, metrics.dat will be the home of a space-delimited line of metrics output suitable for direct import into a spreadsheet.

The following is a list of metrics recorded in the order they are written:

- Total simulation time
- Number of processes finished
- Throughput
- Average CPU utilization
- Average number of page faults per PE.
- Average amount of time spent servicing page faults on each PE
- Cumulative amount of time communication units (e.g. links, switches) were in use
- Average amount of memory allocated on each PE

- Standard deviation of memory allocated on each PE
- Average amount of disk space allocated on each PE
- Standard deviation of disk space allocated on each PE.
- Average process response time
- Average process waiting time
- Average process execution time
- Average slowdown
- Average bounded slowdown
- Highest observed multiprogramming level
- Average multiprogramming level
- Total number of Parallel Context Switches (PCS's)

3 PNRSim Implementation

PNRSim is written exclusively in C++ and was developed under Linux. PNRSim should be easily compiled under standard Linux distributions. PNRSim should also be compilable under UNIX. However, the "getopt" library must be installed. Compilation and installation instructions are included in the PNRSim distribution.

3.1 Documentation

The main source of documentation of PNRSim is this technical report. In addition, there is a set of code documentation HTML files included with the PNRSim distribution under "pnrsim/src/docs/prog". These files are automatically generated from code comments using DOC++. Reading these may be helpful as a reference. Since they are generated automatically from the code, looking at the source code may be helpful also.

3.2 Simulation Library

PNRSim relies on Sim++, a C++ version of Simpack. Simpack is a general-purpose set of simulation libraries [4]. We took Sim++'s discrete-event simulation library and modified it to suit our needs. Specifically, a large amount of code that generated compiler warnings was modified, some code speedups and additional output functions were added, and classes that were not used, such as the Calendar class, were removed. Also removed were unimplemented classes included with Sim++.

A helpful manual is included with the Sim++ distribution [5]. This manual is required reading to use or change the Sim++ library.

3.3 Implementation Overview - A Top-Down Approach

The main object in the PNRSim simulator is the System object. This object contains all of the required structures needed to run simulations.

In terms of computer hardware, System stores a single Network object and many Computer objects. The Network object stores Links, NetSwitches, and routing information. The Computer objects store CPUs, RAM, and Disks. Links, NetSwitches, CPUs, and Disks all inherit from the Resource object, which allows simulated processes to use these objects as limited resources. Each Computer also hosts a PNRServer. The PNRServer contains three additional objects: a PNRClient, a PNRManager, and a PNRServer. Each of these objects is a PNRAgent and handles aspects of the various PNR algorithms built-in to PNRSim. The PNRServer can be configured with different behaviors or disabled if parallel network RAM is not to be used in the simulation.

The System stores multiple Processes. Processes are parallel jobs which store multiple Threads. Each Thread is associated with many Tokens in its lifetime. Tokens are the basic unit of the simulation and allow Threads to reserve computer hardware by interacting with the Sim++ library. A Token is typically generated each time a new event occurs. A Thread can potentially generate many events during its lifetime, and hence, a thread can be associated with many Tokens.

The System also stores a centralized Scheduler, which controls the associations among Threads and Computers. Each Computer can host several Threads, but a Thread must belong to a particular Computer at all times. The Scheduler is an abstract class, and, potentially, many schedulers could be developed. However, the only scheduler currently available is the GangSched (gang scheduler) class. The GangSched class uses the TimeSlices object to control the time-sharing aspect of the gang scheduler.

4 System Models

This section describes the models PNRSim uses to simulate the hardware of the cluster system.

Specifically, the environment being modeled is a network of processing elements (PE) comprising a supercomputer or computing cluster. Each PE has a fixed amount of local memory and a disk with infinite capacity. Each PE will typically use its local memory for accessing data, but can access non-local memory if it is willing to use message-passing to do so. To do this, a network RAM scheme will have to be used.

4.1 PE Model

This section describes how a single PE and its attendant hardware is modeled in PNRSim.

4.1.1 CPU Model

Each thread requires a certain amount of CPU service time to complete its work. Only one thread may access a CPU at a time. No formal timeslicing mechanism on the CPU level exists. If one thread is using the CPU, a second thread must wait until the first releases the CPU before it may use the CPU. Timeslicing may exist on a system-wide level, given an appropriate scheduler. For

instance, gang schedulers may preempt threads at regular intervals to allow other threads use of the CPU.

The workload file defines how much CPU time was used, on average, for each thread of each process listed. If a thread never experiences page faults, communicates with other threads or contends for CPU resources, then the thread will run the length of its CPU time and no more. Implicit is the assumption that the computers defined in the environment match the CPU speeds of the computers the processes in the trace file ran on.

4.1.2 Memory Model

Memory (RAM) is treated as a resource that may be accessed at any time simultaneous with any other process. Memory has a finite capacity (set in megabytes) set as a parameter to the simulation. If a process requests more memory than is available, it may allocate as much as possible. The rest of the memory must be allocated via either network RAM or disk.

It is assumed a memory access is instantaneous. That is, if a process requests data, and this data is stored in RAM, it is assumed that this request can be fulfilled in zero time.

4.1.3 Disk Model

The disk is accessed only when paging occurs. No other disk activity (e.g. reading files, loading programs) occurs. Only one thread may access a disk at a time. Disk reads and writes are treated as equivalent (in terms of service time).

Disk access time is determined by three parameters - seek time, RPM, and bandwidth. Seek time and RPM determine the fixed amount of overhead required to reach a particular unit of data. Bandwidth determines how much time is required to access the required information. Many operating systems use techniques to mask disk transfer times. No such mechanisms are assumed for the simulated PE's.

4.1.4 OS Model

No OS is assumed to be running on any PE. No overhead associated with operating system activity (e.g. CPU usage, disk usage) occurs.

4.1.5 Page Fault Detection Algorithm

Each thread has an average memory access rate. The page fault detection algorithm determines how much time will elapse between the current time and the time when the next page fault will occur. For example, if no pages of the currently running thread are loaded into memory, then the first memory access will be a page fault and the time until the next page fault will be the same as the average memory access rate. If all pages of the thread are loaded into memory, then the amount of time until the next page fault will be infinite.

If some, but not all, pages are loaded into memory, there should be some time between page faults that is greater than the time between memory accesses, since some references will be to pages

already loaded. In fact, this should happen quite often due to locality of memory references. We construct a model to mimic this behavior.

To start, we define the time until the next page fault to be T . We then define the average memory access rate to be R . If we assume no pages are ever loaded into memory, then the next page fault will occur in exactly this amount of time.

$$T = R$$

The operating system will load a page into memory when a page fault occurs. It is very likely that subsequent memory accesses will reference the same page because memory accesses, in general, have the property of locality. If the program accessed memory linearly, then the memory references will traverse the entire loaded page before encountering a new, unloaded page. We assume this is the case and multiply the memory access rate by the page size P . In the case of our simulated system, P has the value 4096.

$$T = R \times P$$

No thread will ever access memory perfectly linearly and it is possible that the thread will return to pages already loaded into memory. The more data that is loaded into memory, the more likely it is that this situation will occur. We define the data brought in from page faults as D . D starts at a zero value and increases by the size of a page at each page fault event. There must be a limit on the amount of memory paged in by a single thread, and that limit is L . L is defined as the minimum of the total amount of memory overallocated on the node and the thread data size. If, for instance, the amount of memory overallocated on the current PE is 32 MB, and the current thread size is 16 MB, L is 16.

$$L = \min(\text{NodeOverallocation}, \text{ThreadSize})$$

We can calculate the amount of overallocated data O of the thread not currently in RAM by subtracting D from L .

$$O = L - D$$

From this amount, we can calculate the percentage of overallocated data not currently in RAM. We call this value U .

$$U = O/L$$

We can then apply U to the calculation of T . Dividing our previous equation by U will cause the average time until the next page fault to go up as more data is loaded into RAM. Note that when U is 0%, T will be defined as infinite.

$$T = (R \times P)/U$$

Finally, the resulting T is fed to an exponential distribution as an average. The distribution will randomize the time between accesses while still following a general trend.

$$T = \text{expntl}((R \times P)/U)$$

One problem with this model is that it does not handle processes that are larger than physical RAM correctly. Specifically, this model will have the processes load a only finite amount of data into RAM. This is not correct for very large processes where, in order to page parts of a program in, other parts will have to be paged out. In this case, it is possible that the process will thrash between localities forever.

We avoid this problem using the following method. When all of the process' data (as determined by the model) is loaded into memory ($U = 0\%$), the simulator will reset the amount of memory unloaded using a triangle distribution. This triangle distribution is given the parameters $A = P$, $B = P$, and $C = (RAMSize - ThreadSize)$. This makes the amount unloaded likely to be small - generating a low level of page faults for future execution. This model mimics locality changes in the program and will produce the desired behavior of continued page fault activity for very large processes.

4.2 Network Model

This section describes the models used to simulate the network equipment of the simulated cluster system.

4.2.1 Links

A link connects either one computer to another computer (or set of computers) or a computer to a switch. Only one message may be transmitted on a link at a time. A link has a fixed latency time and a transmission time based on the bandwidth and the size of the message transmitted. In our experiments, we set the latency of each link to be that of the speed of light through 10 meters of copper wire (50 nanoseconds). No collisions occur on the simulated links. Incoming messages are queued if the link is currently in use.

4.2.2 Switches

A simulated network switch stores communications and forwards them to the appropriate link. Switches may only forward one message at a time and all other messages are queued. Switches are assumed to have a single infinite length queue sorted in FIFO order. Switches have a fixed length processing delay that can be set as a parameter.

4.2.3 Topologies

The network topology of the simulated supercomputer must be one of the predefined topologies. We define N as the number of PE's in the system.

The following topologies are available:

- Star - N links and 1 switch
- Wheel - $N + N/2$ links and 1 switch
- Bus - 1 link
- Connected - $N(N - 1)$ links

4.2.4 Sending Messages

To send a message, a thread must query the network for a path from its local PE to the remote PE. What path is returned depends on the topology. Point-to-point and broadcast messages are supported and used in the simulator. Broadcast messages are implemented as N-1 simultaneous point-to-point messages.

4.3 Process Model

This section describes the models used to simulate the behavior of parallel jobs running on the cluster system.

4.3.1 Memory Allocation Model

It is assumed that the trace file provides accurate information on the total (or average) amount of memory used. This amount is statically allocated at process start time. The amount of memory used by the process does not grow or shrink over the course of execution. Memory is deallocated at process termination.

4.3.2 Local Communication Model

At process start time, each thread of a parallel job is assigned a set of neighbors. Messages are sent to these neighbors periodically to simulate local data updates. These messages are not synchronous. That is, one thread sends a message and the neighboring thread will instantaneously send a response back. Neither the initiating thread nor the receiving thread will wait (synchronize) for local communications. The primary use for this model as it is currently implemented is to increase the general level of network utilization.

In the current implementation of PNRSim, each thread has exactly one neighbor. The neighbor network is arranged in a circular, unidirectional ring. How often (amount of CPU time) local communications occur is a parameter of the simulation.

4.3.3 Global Synchronization Model

Each thread is instructed to synchronize with the "master" thread regularly. How often this occurs is a parameter of the simulation.

After the set amount of CPU time, a thread will signal its intent to synchronize with the controlling master thread. After every thread has passed the set amount of CPU time and signaled its intent, synchronization will occur. Note that a single slow thread (e.g. slowed down by excessive paging) can cause the entire parallel job to slow down at synchronization time. Every thread must wait for the slowest thread. This behavior is an important observed phenomena in parallel systems.

When all threads are ready, synchronization is simulated by each thread simultaneously sending a message to one thread deigned the "master" thread. That a thread is the "master" thread holds no special significance to the simulator.

This model is unrealistic in that threads will wait for every other thread before sending their synchronization message. This model will have to be changed in the future.

4.4 Scheduler Model

Only one scheduler is currently implemented for PNRSim. That is the gang scheduler. However, a space-sharing scheduler can be implemented simply by setting the MPL of the gang scheduler to one.

It is assumed that jobs arrive at the scheduler and the scheduler is able to immediately assign them to PE's. No communication or processing overhead is taken into account.

4.4.1 Space sharing Scheduler

The space sharing scheduler runs as many processes as possible, based on what processes are currently running and what the PE needs of incoming processes are. Once a parallel process has been assigned PE's, the threads of that parallel process run until completion. Preemption is not permitted.

4.4.2 Gang Scheduler

A gang scheduler has been modeled for this simulator. Each job is assigned to a specific time slot. If all submitted jobs can run simultaneously on the set of provided PE's, then only space-sharing is used. If all submitted jobs can not run on the set of provided computers simultaneously, then time-sharing is used in addition to space sharing.

The MPL of the gang scheduler may be capped to a certain number or may be infinite. This is a parameter to the simulation.

The gang scheduler has slot unification and alternative scheduling implemented [6]. Whenever a process terminates or the scheduler progresses to the next time slot, the unification algorithm is invoked. This algorithm identifies idle CPUs in the current time slice and attempts to find jobs in other time slices that can make use of the idle CPUs. If another job can do so, it is scheduled in the current time slice by moving it from its previous time slice to the current one. When its original time slice comes around, the job will be moved again - unless another job has started execution in its place.

If a time slice becomes empty because of the alternative scheduling activity, slot unification occurs. That is, the empty time slot is destroyed and the relocated job permanently stays at its new assigned slot.

The overhead incurred by the system with a parallel context switch is a parameter that is passed to the simulator. In addition to the fixed overhead, simulated parallel context switches kill all currently active message-passing. When processes are resumed, then each message will have to be resumed from its starting point.

Sophisticated supercomputing systems are able to stop communication in mid-completion at one PCS and resume communication from where it left off at another PCS. This sort of behavior could be built into PNRSim in the future.

5 Parallel Network RAM Models

Several Parallel Network RAM approaches are provided. Each approach is a peer-to-peer solution implemented by software loaded on the system's PE's. No solution coordinates with or receives information from the assumed centralized scheduler of the system.

All computers host PNR servents. Servents are composed of three components: a client, a server, and a manager. Each servent may act as one of its components at different times. How these components interact with other servent components is determined by the PNR strategy.

PNR clients allocate network RAM on the behalf of the PE operating system. PNR servers make unallocated local RAM available for allocation by PNR clients . PNR managers coordinate the requests of clients and act as proxies between clients and servers. Managers exist to balance memory requests from multiple threads that belong to one parallel process.

Migration of network RAM is prohibited. "Eviction" of network RAM is prohibited. Overallocation of network RAM and local memory on one PE will result in paging activity on that PE. Each thread will use its allocated network RAM until its termination.

5.1 Network RAM Clients

A PNR client attempts to allocate and deallocate network RAM on the behalf of its hosting node. The node uses allocated network RAM as additional virtual memory just as it would with disk space.

5.1.1 Thread Start

When a process starts execution on a node, it allocates the amount of memory it will use during its execution. If the node's PNR client determines that this allocation will lead to disk usage, the client contacts a manager and requests network RAM. The thread will block until the client notifies it of the network RAM (or lack thereof) received.

If enough network RAM was allocated for the node hosting the thread, network RAM will be used for paging. If some network RAM was allocated, but not enough to cover the need, the client will signal the computer to use the disk for paging.

5.1.2 Thread Execution

During thread execution, the thread will generate memory requests. If the memory request is determined to be a page fault that references network RAM, then the appropriate PNR server is contacted to retrieve a page of memory. It is assumed that a page of memory is sent to the network RAM server for storage for each request for information on a network RAM page.

5.1.3 Thread Termination

At thread termination time, the PNR client decides if it can give up allocated network RAM. Currently, the PNR client simply calculates the total memory demand and compares it to the

amount of network RAM allocated and RAM available. If the demand is less, the network RAM is marked as being able to be deallocated. If some network RAM can be deallocated, but some is still needed, clients deallocate chunks of memory first from servers it believes have the highest memory load. Clients will use the information in their remote memory load tables (typically used to determine which network RAM server to contact for allocation) to make this decision.

5.2 Network RAM Managers

The manager maintains a table of available PNR servers. It is assumed that the manager knows how much RAM is installed on each server machine. This availability table initializes an entry for each server to the amount of RAM installed on the server. This entry indicates how much RAM should be available for allocation as network RAM.

5.2.1 Network RAM Request

Managers listen for network RAM requests from clients (including the client residing on the same node as the manager). When a request is received, a new entry for the parallel process represented by the thread is created in a request table. Depending on the PNR strategy, the manager may immediately act on this request or wait for other requests to come in before acting. Most strategies require that all threads within a parallel job must contact the same manager before the manager is allowed to act on any one of the threads' requests. For this to work, it is assumed that the PNR manager can discover the total number of threads belonging to the parallel job. For instance, the threads can send this information along with their requests. In any case, as each new request comes in, information on the aggregate memory request is stored in a request table.

5.2.2 Server Request and Response

When all requests are received from relevant clients, the manager attempts to select a PNR server from its availability list. Multiple selection schemes are possible. Currently, a randomized worst-fit scheme is used. That is, the server with the most RAM available for remote allocation is always selected. If more than one server has the most amount of RAM available, one server is randomly chosen.

The server is contacted and may grant, partially grant, or deny the request. It is important to note that a server that was listed as having RAM available for allocation may not have the same amount available by the time the request message reaches it.

The server's response is received by the PNR manager. If too little memory was allocated, the PNR manager will attempt to contact another server. The process is repeated until enough network RAM is granted by multiple servers or until no servers are left to query.

5.2.3 Network RAM Request Response

When either enough network RAM is allocated or when all possible network RAM servers are queried, the manager will calculate the total amount granted. Based on this, the manager will divide network RAM up evenly among the requesting clients. If a client requests less network RAM

than it would get in its fair share, exactly as much as it requests is granted and the remainder is reserved for other clients. If a client requests more network RAM than it would get in its fair share, it gets its share and only receives more if other clients do not need all of their shares. Each client is informed of which servers are hosting the granted network RAM.

The even distribution of network RAM is done to ensure that each process in a parallel job runs at roughly the same speed. If network RAM were granted haphazardly, each process would run at different relative speeds. The job as a whole would run at the speed of the slowest process because of synchronization and the fast processes will be held back.

5.2.4 Network RAM Deallocation Notification

It is assumed that deallocation notifications are only sent at job termination time. The manager will listen for deallocation notifications from clients. The deallocation amounts indicated in the messages may have nothing to do with how much was previously allocated to those clients. In some cases, no network RAM may be deallocated because it is still needed by the client. In most strategies, the manager will wait for each client associated with a parallel job to send a deallocation notification.

5.2.5 Server Notification and Response

When all deallocation notifications are received by the manager, the manager determines how much needs to be deallocated and on which servers. Each server is then notified that it may deallocate the specified amount of RAM. The server will respond to this message with an acknowledgment.

5.2.6 Broadcasting Availability Information

During each message-passing interaction, network RAM clients and network RAM servers piggyback current memory load information onto their messages. The network RAM manager receives this information in addition to the relevant payload. It uses this up-to-date memory load information to update its own network RAM availability table. It uses this table for server selection.

Some PNR strategies may prefer share this information by broadcasting it to all servers. Other strategies may only broadcast to selected servers. Currently, broadcasts are only executed after network RAM has been allocated or deallocated.

It should be noted that information may already be out-of-date before it is even broadcast. There is no way around this problem, since it is possible for memory loads to change as messages are being passed on the network. We believe this broadcasting solution provides an economical way to keep availability information reasonably up-to-date.

5.3 Network RAM Server

Servers receive requests from managers for network RAM. If the server has more unallocated RAM than a certain threshold, it will grant the network RAM request and allocate memory to the manager up to that threshold. Currently, the unallocated memory threshold is set to a low value - a tenth of a megabyte. This value can be adjusted and may be useful for future work.

After the memory is allocated, servers receive requests to read and write the allocated network RAM directly from clients. Servers grant all valid deallocation attempts.

5.4 Network RAM Access Model

The paging model is used as the foundation of the network RAM access model. When a page fault occurs, PNRSim will check what percentage of memory is stored as network RAM and what percentage is stored on disk. A random floating-point number between 0 and 1 is chosen to decide if network RAM or disk has just been referenced.

If the disk is referenced, then the activity is no different than normal paging. If network RAM is used, a round-trip message occurs between the host and the remote PE storing its pages. It is assumed that the page is read from remote RAM instantaneously and the only time penalty is that caused by communication overhead.

5.5 PNR Strategies

We propose four different PNR designs. Each design has a slightly different architecture and has different amounts of communication overhead associated with it. This section describes these designs.

5.5.1 Centralized

In this strategy, only one manager exists, and it receives all client requests. All servents know the identity of this manager. All clients will contact this manager and it will coordinate network RAM allocation. The server uses memory load information sent in by clients and servers to make allocation decisions. Since only one active manager exists, the centralized manager does not broadcast any memory load information it receives.

This scheme has the advantage of one agent having all of the information and making all of the decisions. No time or network bandwidth is wasted in sending coordination messages. The disadvantage of this strategy is that it is not scalable. As the system size grows, the network connections leading to the node hosting the manager will become a bottleneck and limit the performance of the system. In the real world, the computational and memory overhead of hosting this central manager would also become major factors. However, in our simulator, these are not taken into account.

5.5.2 Client

In this strategy, each client uses its servent's manager to send allocation requests. The local manager does not wait for messages from other clients - it acts immediately upon the client's request. The manager attempts to allocate as much network RAM as possible for its client. When the client receives network RAM, it begins execution immediately. It does not wait for the other threads in the parallel job. The local manager does not share memory load information with other servents.

This strategy allows clients to allocate network RAM quickly and eliminates all synchronization overhead. It is scalable, since each client is responsible only for itself. It is also simple to implement, since there is no need for a manager at all.

However, this solution has major drawbacks. First, memory load information is not shared. Each client must discover memory load information for itself and this may lead to a large amount of ineffective network RAM allocation requests.

Second, and more serious, the clients do not coordinate memory allocation with each other. Some clients may receive large amounts of network RAM if they get their messages to the servers first while other clients get very little network RAM. This will not improve the performance of parallel jobs as a whole, since each job will only execute at the speed of its slowest thread.

In fact, this scheme may worsen overall performance, since much of the network RAM allocated is can be wasted. Thus, no benefit is gained from network RAM and higher memory loads on the servers may induce disk paging, the very problem we are trying to avoid!

5.5.3 Local Managers

In this strategy, each client contacts a "local" manager. Specifically, whenever a job starts or stops, one of the servents running on a node associated with that job will randomly volunteer to act as the manager in addition to acting as a client. Each servent involved must agree on which servent will act as the manager. All clients from the involved servents will contact this manager. The manager will take their requests, allocate network RAM (if possible), and divide up the received network RAM evenly among the requesting clients. At the end of each allocation and deallocation, the manager will broadcast memory load information to each servent in the system. This is necessary since each servent on the system can potentially act as a manager.

Since no single node is loaded with all requests, this solution is scalable. It makes good use of allocated resources via the coordination of client requests. Broadcasting the memory load information should keep the tables of the servents relatively up-to-date.

However, the major drawback of this approach is the broadcasting step. Sending a message to each servent on the system can introduce congestion into the system, especially if no real broadcasting facility exists and broadcasting must be implemented via multiple point-to-point messages (as it is in our simulator). The larger the system, the bigger this issue will become.

Also, since there is no central authority on memory allocation information, servents may be more likely to act on outdated information. Like the centralized strategy, waiting time is incurred in the coordination step by waiting for all clients to send in their requests.

5.5.4 Backbone

This strategy is similar to the local manager strategy. The key difference is that only a subset of servents will act as managers. This subset of servents will be well known and all clients will contact these servents for their network RAM requests. Clients will randomly select a manager for service. As in the local managers design, all clients associated with a job must agree on who to contact.

The backbone of managers will coordinate among themselves by broadcasting memory load information only to managers. If the backbone of managers consists of only one member, then this strategy is equivalent to the centralized strategy. If the backbone of managers consists of all servents, then this strategy is equivalent to the local managers strategy.

This scheme can potentially be a "best of both worlds" solution, compared to the centralized and

local managers solutions. It is more scalable than the centralized solution, since load is shared among many servants, and it uses fewer messages for synchronization than the local managers solution, since broadcast messages only need to be sent to a subset of nodes.

The backbone strategy also has the advantage of being customizable to the cluster setup. If the network is small, then a small backbone (perhaps a backbone of one) may be all that is required. As the network gets larger, then the number of servants in the backbone can be increased appropriately to manage scalability.

6 Acknowledgments

This research was made possible in by grants from the Michigan Space Grant Consortium and the National Science Foundation

APPENDIX

A Sample Configuration File

```
#####
# Header comments may be placed here.
#####

System YOUR_SYSTEM_NAME_GOES_HERE

## Metrics Recording ##
Maxtime          10000000      #Max amt. of simulated time PNRSim should run
MetricsPeriod    50000        #How often a sampling should take place
Seed             1            #The random number seed

## Process Variables ##
Process          Process      #The process name
memaccessrate    0.00000024    #Memory access rate suggested by the literature
commrate        10           #How often to communicate locally
synchrates       1           #Ballpark figure from CFD app

## Scheduler Variables ##
Scheduler        GANG         #Gang scheduler
priority         FCFS         #Scheduler's priority queue scheme
packing          BESTFIT      #How the scheduler will pack PE's
MPL              2           #Limited time slices to avoid thrashing
PCS              0.004        #Parallel context switch
memaware         No          #Scheduler aware of memory req. ahead of time?
quantum          60.0         #Time slice size
netrampacking    FIRSTFIT     #Ignored

## Computer Configuration variables ##
Computer         Node        #Name of the PE's
Number           64          #The number of PE's in the system

## Network RAM Configuration Variables ##
Netram           Yes         #Whether PNR will be used
PNRStrategy      Backbone    #Which strategy will be used
GlobalKnowledge  No          #Does the strategy know global mem. load info?
Managers         4           #How many managers will be used? (backbone only)
ServerThreshold  0.1         #Amt. of RAM required before granting net. RAM

## CPU Configuration variables ##
CPU              1000        #This parameter is (essentially) ignored
```

```
## RAM Configuration variables ##
RAM          32          # Amount original CM-5 had

## Disk configuration variables ##
RPM          7200
Seek         0.009      #Typical seek time of commodity hard drive
Transfer     50         #Typical transfer time of commodity hard drive

## Network Configuration Variables ##
Network      star       #A common topology for simple clusters

## Link Configuration Variables ##
Bandwidth    12.5       #12.5 MB/s = 100 Mb/s
Latency      0.00000005 #Speed of light for 10 meters of copper wire

## Switch Configuration Variables
Delay        0.00008    #Reported netgear latency
```

B Sample Workload

; Comments beginning with a semi-colon are accepted.
; This workload fragment is taken directly from Dror Feitelson's LANL workload.

```
1 0 -1 3691 8 3436 6676 -1 3600 5904 1 2 2 2 -1 -1 -1 -1
2 465 5 3362 2 489 988 32 900 1024 1 3 3 3 1 -1 -1 -1
3 1906 12 626 32 516 26828 512 900 32768 1 1 1 1 3 -1 -1 -1
4 3571 11 647 32 484 26828 512 900 32768 1 1 1 1 3 -1 -1 -1
5 4009 7 3039 8 2605 1812 128 3600 3200 1 9 8 6 1 -1 -1 -1
6 4031 5 6253 2 476 1004 32 300 1024 0 3 3 3 1 -1 -1 -1
7 4346 15 2412 2 64 6408 32 300 6400 1 6 5 3 1 -1 -1 -1
8 4393 5 986 2 534 2476 32 3300 3840 1 5 4 3 1 -1 -1 -1
9 4684 6 25 32 0 0 512 1800 32768 1 4 1 4 3 -1 -1 -1
10 5705 3 1105 8 53 1732 128 3600 4000 1 7 6 3 1 -1 -1 -1
```

References

- [1] Dror G. Feitelson. <http://www.cs.huji.ac.il/labs/parallel/workload/>.
- [2] Dror G. Feitelson and Larry Rudolph. Parallel job scheduling: Issues and approaches. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 1–18. Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.
- [3] Dror G. Feitelson. Metrics for parallel job scheduling and their convergence. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 188–205. Springer Verlag, 2001. Lect. Notes Comput. Sci. vol. 2221.
- [4] Paul A. Fishwick. *Simulation Model Design and Execution*. Prentice-Hall Inc., 1995.
- [5] R.M. Cubert and P. Fishwick. Sim++, version 1.0. Technical report, Department of Computer and Information Science and Engineering, University of Florida, 28 July 1995.
- [6] Dror G. Feitelson. Packing schemes for gang scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 89–110. Springer-Verlag, 1996. Lect. Notes Comput. Sci. vol. 1162.