PARALLEL MACHINE SIMULATOR

Manual

June 2015

Content

1. Introduction

Parallel Machine Simulator (*PMS*) is an interpreter for execution of programs made for Parallel Random Access Machine (*PRAM*). PMS is intended for execution on Random Access Machine (RAM) for purposes of testing and emulation. Interpreter language created for interpreter has a Python – like syntax.

1.1. Random Access Machine (RAM) Model

Traditionally, software has been written for serial computation. A problem is broken into a discrete series of instructions. Instructions are executed sequentially one after another.

Random Access Machine (RAM) is a favorite model of a sequential computer. RAM model has unbounded number of local memory cells. Instruction set includes operations for moving data between memory cells, comparisons and conditional branches, and simple arithmetic operations. Execution of program starts with the first instruction and ends when a HALT instruction is executed. All operations take unit time regardless of the lengths of operands. Time complexity is equal to the number of instructions executed. Space complexity is equal to the number of memory cells accessed.

Under this model of computation, we are confronted with a computer where:

• Each simple operation (+, *, –, =, if, call) takes exactly one time step.

• Loops and subroutines are not considered simple operations.

• Each memory access takes exactly one time step. Further, we have as much memory as we need. The RAM model takes no notice of whether an item is in cache or on the disk.

Under the RAM model, we measure run time by counting up the number of steps an algorithm takes on a given problem instance. If we assume that our RAM executes a given number of steps per second, this operation count converts naturally to the actual running time.

The RAM is a simple model of how computers perform. Perhaps it sounds too simple. After all, multiplying two numbers takes more time than adding two numbers on most processors, which violates the first assumption of the model. It strikes a fine balance by capturing the essential behavior of computers while being simple to work with. RAM model is useful in practice.

1.2. Abstract computer for designing parallel algorithms

PRAM is a straightforward and natural generalization of RAM. PRAM model has an unbounded collection of numbered RAM processors P0, P1, P2,... and an unbounded collection of shared memory cells M[0], M[1], M[2],... . Each Pi has its own (unbounded) local memory (registers) and knows its index i. Each processor can access any shared memory cell (unless there is an access conflict, see further) in unit time. Input at a PRAM algorithm consists of n items stored in (usually the first) n shared memory cells. Output of a PRAM algorithm consists of n' items stored in n' shared memory cells.

PRAM instructions execute in 3-phase cycles: 1) Read (if any) from a shared memory cell, 2) Local computation (if any) and 3) Write (if any) to a shared memory cell. Processors execute these 3-phase PRAM instructions synchronously. The only way processors can exchange data is by writing into and reading from memory cells.

Special assumptions have to be made about shared memory access conflicts. P0 has a special activation register specifying the maximum index of an active processor. Initially, only P0 is active, it computes the number of required active processors and loads this register, and then the other corresponding processors start executing their programs.

Computation proceeds until P0 halts, at which time all other active processors are halted. Parallel time complexity is equal to the time elapsed for P0's computation. Space complexity is equal the number of shared memory cells accessed.

PRAM is an attractive and important model for designers of parallel algorithms. It is natural: the number of operations executed per one cycle on p processors is at most p. It is strong: any processor can read or write any shared memory cell in unit time. It is simple: it abstracts from any communication or synchronization overhead, which makes the complexity and correctness analysis of PRAM algorithms easier. Therefore, it can be used as a benchmark: If a problem has no feasible/efficient solution on PRAM, it has no feasible/efficient solution on any parallel machine. It is useful: it is an idealization of existing (and nowadays more and more abundant) shared memory parallel machines.

The PRAM corresponds intuitively to the programmers' view of a parallel computer: it ignores lower level architectural constraints, and details, such as memory access contention and overhead, synchronization overhead, interconnection network throughput, connectivity, speed limits and link bandwidths, etc.

2. PRAM programming model

As its name indicates, the PRAM was intended as the parallel-computing analogy to the random-access machine (RAM). In the same way that the RAM is used by sequential-algorithm designers to model algorithmic performance (such as time complexity), the PRAM is used by parallel-algorithm designers to model parallel algorithmic performance (such as time complexity, where the number of processors assumed is typically also stated). Similar to the way in which the RAM model neglects practical issues, such as access time to cache memory versus main memory, the PRAM model neglects such issues as synchronization and communication, but provides any (problem-size-dependent) number of processors.

PRAM corresponds intuitively to the programmers' view of a parallel computer: it ignores lower level architectural constraints, and details, such as memory access contention and overhead, synchronization overhead, interconnection network throughput, connectivity, speed limits and link bandwidths, etc.

2.1. Handling shared memory access conflicts

To make the PRAM model realistic and useful, some mechanism has to be defined to resolve read and write access conflicts to the same shared memory cell.

Concurrent Read Concurrent Write (CRCW) PRAM: Both simultaneous reads and simultaneous writes of the same memory cell are allowed.

Concurrent Read Exclusive Write (CREW) PRAM: Simultaneous reads of the same memory cell are allowed, but only one processor may attempt to write to an individual cell.

Exclusive Read Concurrent Write (ERCW) PRAM: Simultaneous writes to the same memory cell are allowed, but only one processor may attempt to read from an individual cell.

Exclusive Read Exclusive Write (EREW) PRAM: No two processors are allowed to read or write the same shared memory cell simultaneously.

Assume p-processor PRAM, p<n. Assume that shared memory contains n distinct items and P0 owns value x. The task is to let P0 know whether x occurs within the input array.

EREW PRAM algorithm:

1) P0 broadcasts x to P1,...,Pp in log p steps using binary broadcast tree.
2) All processors perform local searches, each on [n/p] items in [n/p] steps.
3) Every processor defines a flag Found and all processors perform a parallel reduction.

T(n,p)=log p + n/p

ERCW PRAM algorithm: is rarely considered, due in part to a general belief that concurrent writing does not add much power to a model without concurrent reading. There are some algorithms that solve problems on the ERCW PRAM much faster than they could be solved on the EREW PRAM. Here the algorithm and the complexity would be the same as EREW.

CREW PRAM algorithm: A similar approach, but P1,...,Pp can read x simultaneously in O(1) time. But the final reduction takes O(log p) time anyway, so

T(n,p)=log p + n/p

CRCW PRAM algorithm: The final step takes now also O(1) time, those processors with the flag Found set can write simultaneously into P0's cell

T(n,p)=n/p.

PMS interpreter is designed to test algorithms on these models. Breaking constraints of memory model in interpreter results with no execution of block and an error message.

3. Interpreter

We wanted to create a tool that will interpret code written for PRAM computer and simulate its behavior. PMS is interpreter. Interpreter is a computer program that directly executes instructions written in a programming or scripting language, without previously compiling them into a machine language program.

Focus of the PMS is functionality not performance. Having this in mind and that Python is one of the most popular languages we have decided to adopt Python – like syntax for the PMS.

There were two possible approaches to build such a tool. First approach was to create entirely new language and interpreter for it. Second approach was to create pre – interpreter that would be on top of another component. We have decided for the second approach, pre – interpreter on top of the Python interpreter.

Python is a widely used general-purpose, high – level programming language. Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than would be possible in languages such as C++ or Java. The language provides constructs intended to enable clear programs on both a small and large scale.

3.1. Interpreter Language

An interpreted language is a programming language for which most of its implementations execute instructions directly, without previously compiling a program into machine-language instructions. The interpreter executes the program directly, translating each statement into a sequence of one or more subroutines already compiled into machine code.

As already mentioned PMS interpreter is inspired by existing Python interpreter. Syntax of PMS is intended to expand on the syntax of the Python 2.7 version. Minimum executable unit for interpreter is block.

Block is a section of code which is grouped together. Blocks consist of one or more declarations and statements.

PMS follows Python interpreter behavior when interacting with users. Interpreter will prefix beginning of every user's line input with '... ' if there is a current block that is not finished, otherwise it will put '>>> '.

PMS forbids usage of tabs. This was done because mixing whitespace and tabs bring mess into the code and possibly can lead to unwanted outcomes. Using whitespace is strongly encouraged.

Statements interpreter understands are divided into pre – interpreter statements and interpreter statements. Interpreter statements are instructions to be executed on the interpreter. While pre – interpreter statements manipulate interpreter itself.

3.2. Pre – interpreter statements

Pre - interpreter statements and their short description can be found at Table 1 Pre – interpreter statements and short description.

Statement	**Short Description**
Comment	Comments in the code
Load	Loads external source of code
PRAM	Sets memory model (RAM, CRCW, CREW, ERCW, EREW)
Reset	Resets PMS
Verbose	Sets additional information during execution

Table 1 Pre – interpreter statements and short description

Comment is a programming language construct used to embed programmer – readable annotations in the source code of a computer program. Those annotations are potentially significant to programmers but are generally ignored by compilers and interpreters. Comments start with '#' and continue to the end of the line. Interpreter will disregard anything that is commented. Inputting a line that is commented at its beginning will start a block, this behavior was adopted from Python interpreter. Example of line that is commented at its beginning:

```
# This is a comment
```

Instead of making interpreter relying heavily on user's line by line manual input we have decided that a good option of input would be from the file. There are algorithms and procedures that are common, loading those common parts of the code may be useful. Also writing code in the file and loading it may also be very useful. *Load* statement will calculate its whitespace offset and when inserting loaded content it will prefix with the same whitespace offset as calculated. *Load* statement is used to input code to interpreter from the file. *Load* pre – interpreter statement form is:

:load <file>	where file is relative or absolute path to the file that user wants to load from.

We have discussed how there are different memory models for making PRAM more real. *PRAM* pre – interpreter statement changes memory model of the interpreter. There are five memory models RAM, CRCW, CREW, ERCW and EREW. Memory model cannot be changed inside of the block, it must be standalone statement. Changing memory model does not change variables in the local namespace or any other state of the interpreter. Default memory model when PMS starts is RAM. Form of the *PRAM* pre – interpreter statement is:

:pram <MODEL>	where MODEL is one of the five available memory models.

After each execution of block, variables that interpreter has stored will persist. Sometimes in between blocks it is useful to remove any changes done to the memory. *Reset* statement will reset the local namespace of the interpreter to the state it was when it started. *Reset* will clean all the memory locations that user has manipulated. *Reset* takes no arguments. Reset cannot be executed inside block, it must be a standalone command. Form of the reset pre – interpreter statement is:

:reset	reset takes no arguments

During the execution some information is available regarding what is happening in the interpreter. In this current version PMS can inform the client about what are the memory locations that were accessed by corresponding nodes during the execution of code in parallel environment. Client can decide to be fully informed or not be informed at all with *Verbose* statement. *Verbose* command can be executed within a block if user desires. When PMS starts verbosity is set to false. Form of the verbose pre – statement is:

:verbose <Boolean>	where Boolean can be replaced with Boolean value true or false. It is case insensitive.

3.3. Interpreter statements

Depending on the memory model interpreter available statements are different. If the memory model is RAM all the statements that are in Python 2.7 interpreter will be able to translate. If the memory model is any of the PRAM models then only *Assignment, For, If, Parallel, Pass, Print* and *While* statements are available and no function can be called within parallel block. Statements and their short descriptions can be found at Table 2 Interpreter statements and short description for each.

Additionally, *scan* function is defined for all the models. If there is any error with any of the statements in the block such as syntax error or memory violation, PMS will return the state of the local variable namespace to the state before the execution of block that contains such an error.

Assignment statement is used for binding a value to a memory location with assignment operation. It is done with assignment operator '='. *Assignment* operation for PRAM model has additional constraint that it can contain only one assignment operator in that line. *Assignment* operation can be standalone statement or it can be included in the block. It cannot start a block. Form of the *Assignment* statement is:

<lhs> = <rhs>	where lhs is the left hand side marking memory location and rhs is right hand side defining value that is being assigned.

For statement is used to represent *for* loop. *For* loops are traditionally used when you have a piece of code which you want to repeat n number of times. *For* statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. It can iterate over number generator or over generated numbers. *For* loop can be standalone statement or it can be included in the block. It must start a block. Form of the *For* statement is:

for <it> in <range>: where it is iterator variable that iterates over assigned

#statement(s) range

Interpreter statement	Short description
Assignment	Assign value to memory location
For	Loop for iteration over sequence
If	Branching statement
Parallel	Loop for parallel iteration over sequence
Pass	Does nothing. Placeholder statement.
Print	Printing values on standard output.
While	Loop with iteration under condition.

Table 2 Interpreter statements and short description for each

If statement is probably the most well-known statement type for flow control. There is no support for 'else' and 'elif' statements. *If* statement takes Boolean expression that we will name condition. If the condition is true, then do the indented statements. If the condition is not true, then skip the indented statements. *If* statement can be standalone statement or it can be included in the block. It must start a block. Form of the *If* statement:

if (<condition>): parenthesis is optional. Condition is Boolean expression.

#statement(s)

Parallel statement can be only executed if any of PRAM memory models is active (EREW, ERCW, CREW, CRCW). *Parallel* statement cannot be inside sub block of another *Parallel* statement. *Parallel* statement can be standalone statement or it can be included in the block. It must start a block. *Parallel* statement syntax resembles for loop statement syntax. *Parallel* statement distributes one variable, which we will call inherited node property variable, over desired number of nodes. Each inherited node property variable's value is determined by the value of element with the same intent in assignment list as the index of the node receiving the value. Values can be the same or different, it depends on the user's desires. Statement also declares that entire block of the *Parallel statement* will be instantly executed over desired number of nodes. Form of the parallel statement is:

```
Parallel <imp> in <al>:          where imp is inherited property node variable
    #statement(s)                and al is assignment list
```

The *Pass* statement does nothing. It can be used when a statement is required syntactically but the program requires no action. *Pass* statement can be standalone statement or it can be included in the block. It can not a block. Form of the *Pass* statement is:

```
    pass                         pass takes no arguments
```

Print evaluates one expression and writes resulting object to standard output. If an object is not a string, it is first converted to a string using the rules for string conversions. The (resulting or original) string is then written. *Print* statement can be standalone statement or it can be included in the block. It can not a block. Form of the *Print* statement is:

```
print <rhs>              where rhs is right hand side expression
```

A *While* loop statement repeatedly executes a target statement as long as a given condition is true. There is no support for ‘else’ statement. *While* loop can be standalone statement or it can be included in the block. It must start a block. There is no support for ‘else’ statement. Form of the *While* statement is:

while (<conditional>): parenthesis is optional. Condition is Boolean expression.

```
    #statement(s)
```

Scan function has signature scan(function, iterable[, state]), where function is the associative function, iterable is the iterable container and state is the value to be added to each element of the result. For example list(scan(operator.add, [1,2,3])) returns [1, 3, 6].

Reduce function has signature Reduce(function, iterable[, initializer]). Reduce function applies function of two arguments cumulatively to the items of iterable, from left to right, so as to reduce the iterable to a single value. For example, reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) calculates ((((1+2)+3)+4)+5). The left argument, x, is the accumulated value and the right argument, y, is the update value from the iterable. If the optional initializer is present, it is placed before the items of the iterable in the calculation, and serves as a default when the iterable is empty. If initializer is not given and iterable contains only one item, the first item is returned.

4. Architecture

PMS is written as Java 8 project. Maven is used as build automation tool. It uses Spring Context framework for dependency injection and inversion of control. Jython interpreter is used as Python interpreter.

4.1. Packages overview

Let's make overview of the most important packages that contain crucial classes and mechanics behind them.

4.1.1. hr.fer.zemris.parallelmachinesimulator

Parallel machine simulator package contains classes Main and Parallel Machine Simulator. It also contains sub packages: constants, exception, expression, interpreter, memory, model, output, pram processor, pre interpreter and utils.

Main class provides for Spring framework context. From that context Parallel Machine Simulator class is requested.

When Parallel Machine Simulator is started, banner message is printed, trace back is invoked and infinite loop is started. In the infinite loop input from user is obtained. If the input contains tabs exception is raised. If input contains no tabs its execution is forwarded to the Active Interpreter. Parallel Machine Simulator takes responsibility for receiving input and not allowing exception to leave infinite loop.

4.1.2. Interpreter

Interpreter package contains interfaces Interpreter and Python interpreter; and also classes Active Interpreter, Interpreter Factory Jython Interpreter and PRAM Interpreter.

Interpreter interface has only one method named push that takes String and returns Boolean value. Method takes line to push into the interpreter and form a block. Return value of the function gives an answer if the pushed line has finished the block or not.

Python interpreter interface contains set of methods that are needed (expected) from a Python interpreter. As we have said PMS is designed to be on top of the Python interpreter. This interface makes a contract for desired functionalities.

Active Interpreter when called by Parallel Machine Simulator informs it if current block is completed or not so Parallel Machine Simulator can prompt user with corresponding start of line. Active Interpreter component takes lines of user input and forwards their execution to the Pre Interpreter, PRAM Interpreter or Python Interpreter. Pre Interpreter is first to be offered with line. If Pre Interpreter cannot process the line Active Interpreter will offer line to one of the interpreters. Until block is finished there can be no change of Interpreter. After block is finished reference to current Interpreter is removed. When there is no active block Active Interpreter will ask Interpreter Factory to provide him with an appropriate instance of Interpreter.

Interpreter Factory returns reference to the appropriate instance based on the current Active Memory Model. Both of the interpreters are singleton and factory just asks for the appropriate object from the Spring context.

Jython Interpreter implements Python interface. It is an adapter for the Interactive Console defined in the Jython library. This is the active interpreter if there is a line to be pushed and active memory model is RAM.

PRAM Interpreter is interpreter to be used when active memory model is not RAM. It organizes lines pushed to it into the blocks. Blocks are represented as graph of components that execute statements. Those components are named PRAM Processor (name may be misleading) objects. It decides upon indentation of lines to which block does that line belong to. It delegates creation of PRAM Processor objects to the PRAM Processor Factory. If the execution of block fails, PRAM Interpreter rolls back local variable state to the one previous to the execution of current block.

4.1.3. Pre interpreter

Pre Interpreter package contains interface Pre Interpreter Directive and class Pre Interpreter. Within it package directive is contained. Package directive contains classes Comment, Load, PRAM, Reset and Verbose.

Pre Interpreter is hardcoded to take references to the classes Comment, Load, PRAM, Reset and Verbose when created. This is done because some pre – interpreter statements have higher precedence in front of others like Comment. More elegant solution can be made but for the current version of PMS we are satisfied with this solution. Upon offered with line Pre Interpreter will check if any of the Pre Interpreter Directives he has reference to can process given line in order. First Pre Interpreter Directive that can process is assigned for processing.

Pre Interpreter Directive is interface for declaring pre – interpreter statements. For every class that implements Pre Interpreter Directive it is important to define when it can process given line and how to process it.

Comment class implements Pre Interpreter Directive. It can process line if it contains character '#'. It removes commented part of the line and pushes the rest to the Parallel Machine Simulator.

Load class implements Pre Interpreter Directive. It takes responsibility for all the lines containing ':load' sequence. If in correct format it will read content of provided file and every line it has read will additionally prefix with the indentation that is equal to the indentation of load command itself. That content will be pushed to the Parallel Machine Simulator.

Reset class implements Pre Interpreter Directive. It can process all lines that are equal to the sequence ':reset' if there is no active interpreter at the moment (meaning there is no block being interpreted). Reset will delegate request to Parallel Machine Simulator. Parallel Machine Simulator will proceed with the resetting.

Verbose class implements Pre Interpreter Directive. It takes responsibility for all the lines containing ':verbose' sequence. If in correct format it will set Verbose Component to be verbose or not to be.

4.1.4. Output

Package output contains only one class named Verbose Component. Verbose Component is designed to be entry point for all the output of the Parallel Machine Simulator. It is not fully developed. It should take responsibility for different levels of information and taking orders on which information should be printed on the output for the user.

4.1.5. Model

Model package contains enum Memory Model and class Active Memory Model. Enum Memory Model enumerates memory models RAM, EREW, ERCW, CREW and CRCW and has static method for creating memory model object from String.

Active Memory Model class is singleton that is intended to contain information about current memory model. Every class that has to be have access this information has reference to this object.

4.1.6. PRAM Processor

PRAM Processor package contains annotation PRAM Processor Statement, enum Block Property, interface PRAM Processor, abstract class PRAM Processor, class PRAM Processor Factory and package statement. Statement package contains classes Assignment, For, If, Parallel, Pass, Print and While.

PRAM Processor Statement has runtime retention and has method keyword that returns String. Its purpose is to mark statements that PRAM Interpreter can interpret.

Block Property is enumerator that enumerates CREATOR and BODY. These enumerations are used to mark PRAM Processors property for being able to create block or not to create block.

PRAM Processor is interface that defines that classes implementing it should have Memory Model, Body Property and indentation information. Public methods for assigning a line of interpreter language to it, pushing some other PRAM Processor to its sub block and execute method to execute assigned line and all the sub block that are pushed to it.

Abstract PRAM Processor defines common behavior for all the PRAM Processors. Abstract PRAM Processor defines how all the blocks define their memory model from their parent’s memory model and their default memory model. Each Abstract PRAM Processor defines its default memory model. Abstract PRAM Processor defines behavior for pushing itself to its parent Abstract PRAM Processor depending on indentation. It takes care that it is not able to receive any Abstract PRAM Processor as its child if it does not have appropriate Block Property. It also takes care of reporting it’s ignore location, left hand side expression and right hand expression to the Expression receiver.

PRAM Processor Factory is assigned to create PRAM Processor based on the line that is supposed to assign, parent to which that line belongs and indentation (which is passed just for faster calculations). When created it acquires list of all the classes with PRAM Processor Statement annotation. When requested to create an instance of PRAM Processor it select from the catalogue it had acquired based on the keyword of that can be found in that line.

Assignment class extends Abstract PRAM Processor and is annotated with PRAM Processor statement. When requested Block Property it will return BODY. Its keyword is '='. Its lhs is equal to expression on the left of the equation operator and its rhs is equal to the expression on the right side of the equation operator. If the current memory model is not RAM and variable to which is being assigned does not already belong to the list it is added to the ignore list. Default memory model of assignment statement is RAM.

For class extends Abstract PRAM Processor and is annotated with PRAM Processor statement. When requested Block Property it will return CREATOR. Its keyword is 'for'. Its rhs is equal to the sequence being iterated over. It updates the list of ignored memory locations with variable that is iterating trough sequence if its memory model is not RAM. This is done because those for loops are being executed on every node and that variable is disjoint for all the nodes. Default memory model of for statement is RAM.

If class extends Abstract PRAM Processor and is annotated with PRAM Processor statement. When requested Block Property it will return CREATOR. Its keyword is 'if'. Its rhs is equal to the condition being examined. Default memory model of if statement is RAM.

Parallel class extends Abstract PRAM Processor and is annotated with PRAM Processor statement. When requested Block Property it will return CREATOR. Its keyword is 'parallel'. Its rhs is equal to the sequence being iterated over. Variable that is being divided among processors is added to the ignore list for every node. Default memory model of Assignment statement is equal to the current memory model of the Active Memory Model. Parallel interacts with Joint Memory and Expression Receiver to coordinate simulation of execution on nodes.

Pass class extends Abstract PRAM Processor and is annotated with PRAM Processor statement. When requested Block Property it will return BODY. Its keyword is 'pass'. Pass statement does not interact with any memory locations. Default memory model of pass statement is RAM. Execution of pass statement does not change anything.

Print class extends Abstract PRAM Processor and is annotated with PRAM Processor statement. When requested Block Property it will return CREATOR. Its keyword is 'print'. . Its rhs is equal to the expression being printed. Default memory model of print statement is RAM.

While class extends Abstract PRAM Processor and is annotated with PRAM Processor statement. When requested Block Property it will return CREATOR. Its keyword is 'while'. Its rhs is equal to the condition being examined. Default memory model of while statement is RAM.

4.1.7. Exception

Exception package contains classes Syntax Exception and Memory Violation. Both classes provide static factory methods for some standard occasions when they are raised. These two exceptions are the main exceptions that occur in the system. Syntax exception represents user's mistakes and Memory violation represents violation of the shared memory constraint.

4.1.8. Expression

Expression package contains class Expression Receiver. This singleton class is the main point for all the expressions that PRAM Processors have to report. Expression that need to be reported are the left side expressions, right side expressions and expressions depicting variable locations that should be ignored by shared memory constraints. After being notified by Parallel, Expression Receiver will create Memory objects from all the locations and forward them to the Joint Memory.

4.1.9. Memory

Memory package has three classes Joint Memory, Memory and Memory Factory.

Joint Memory class represents shared memory of the PRAM model. It takes responsibility to store all memory access per one node and for all nodes. If there has been violation of current active memory model Joint Memory will raise an exception.

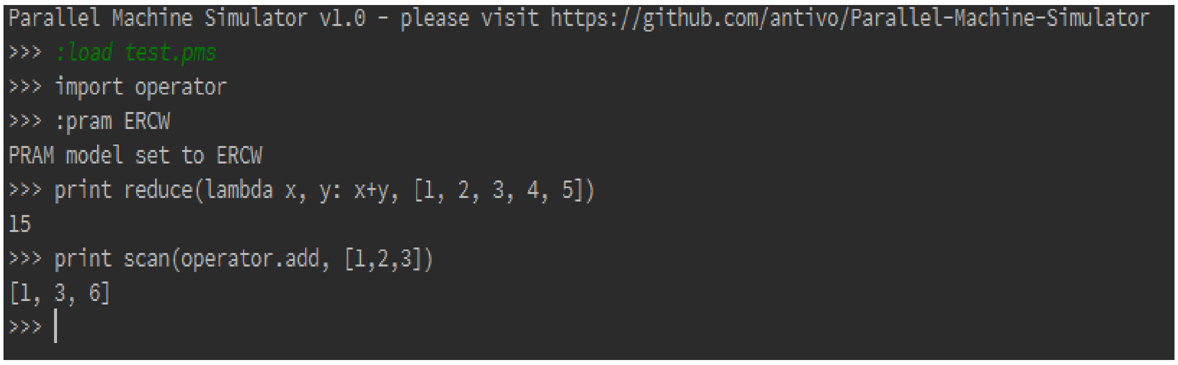
Memory class represents memory location being use. Each Memory object is instructed how to check if it is equal location to another memory object. Two memory objects are considered equal if they both represent same memory location or if one memory object represents memory location that is subset of memory location that is represented by that other object. When two objects are determined to be the same transformation can be done. Transformation is when one of the objects is changed (the one that points to the smaller memory location) to contain the same data as the other object.

Memory Factory class creates Memory objects. Memory objects are created from appropriate String object.

5. Examples and demos

5.1. Scan and reduce

Example of executing scan and reduce function can be seen on Figure 1 Scan and reduce example. We can also see that PMS will greet user and direct him to check out web page where source code of PMS can be found. In this example we see that importing module works since we are in memory model RAM. User is using load statement to load code from file *'test.pms'*. On the picture user's input is in green. PMS output is in white. PRAM model is changed to ERCW. In the next line we have print statement that requires reduction over list of numbers with addition operator. In the line after that we can see scan function over list of natural numbers with addition operator.

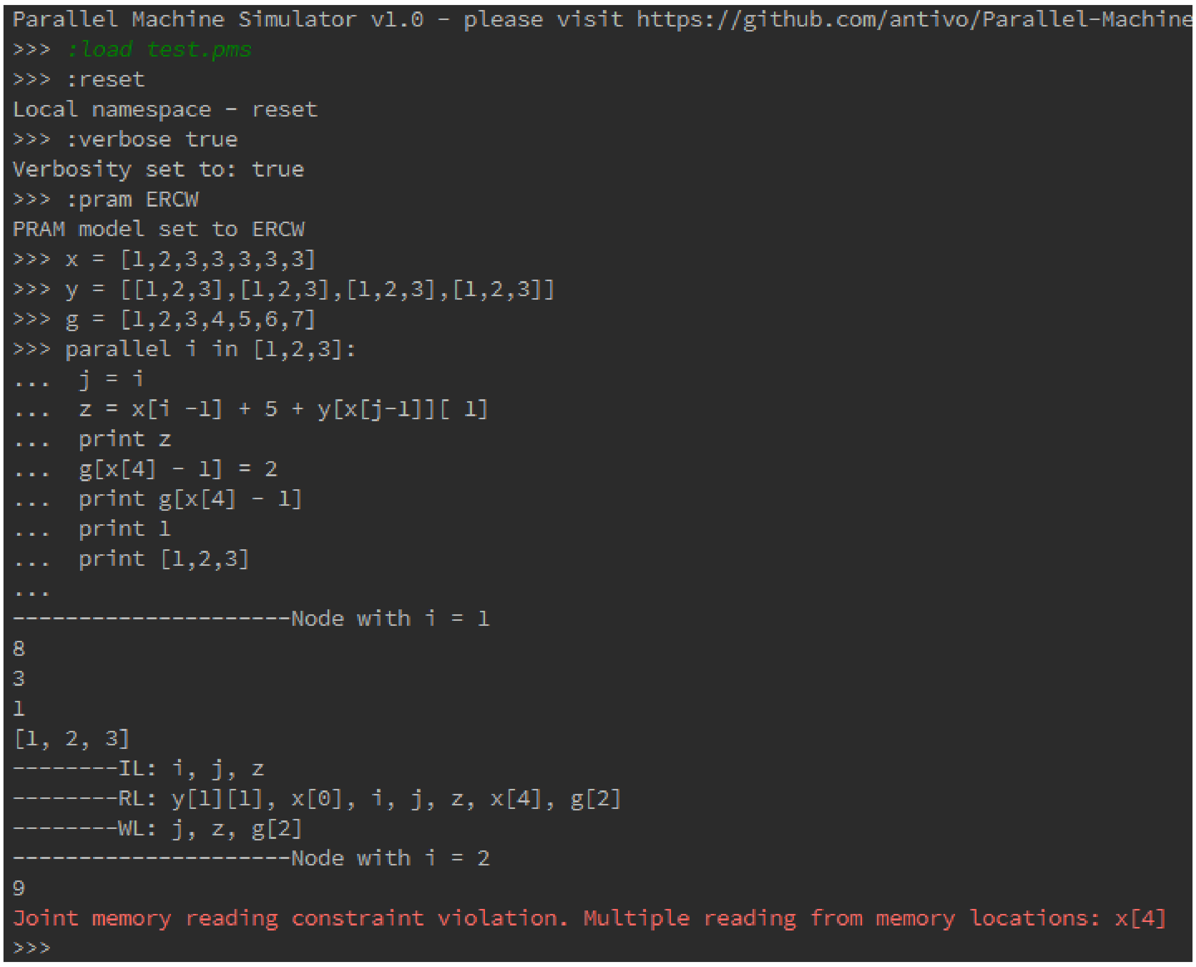


```
Parallel Machine Simulator v1.0 - please visit https://github.com/antivo/Parallel-Machine-Simulator
>>> :load test.pms
>>> import operator
>>> :pram ERCW
PRAM model set to ERCW
>>> print reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])
15
>>> print scan(operator.add, [1,2,3])
[1, 3, 6]
>>> 
```

Figure 1 Scan and reduce example

5.2. Shared memory model violation

In the Figure 2 Shared memory violation we can observe shared memory violation report from PMS. User's input is colored green, PMS output is colored white and PMS errors are colored red. User requested reset of the variables in the local namespace. Verbosity is set to true which means user will receive additional info that is available during execution. Memory model is set to ERCW. In the next few lines we have initialization of one dimensional arrays *x* and *g* and also initialization of two dimensional array y. Statement that follows is parallel statement. In the parallel statement it is defined that there will be three nodes each assigned with different value of variable *i.* In the other lines contained in that block we have assignments and reading from various memory locations. During the execution of second node memory violation has occurred. As we can see in this example node with value *i = 1* is reading from the memory location *x[4]*, the same memory location that node with the value *i = 2* is reading from.

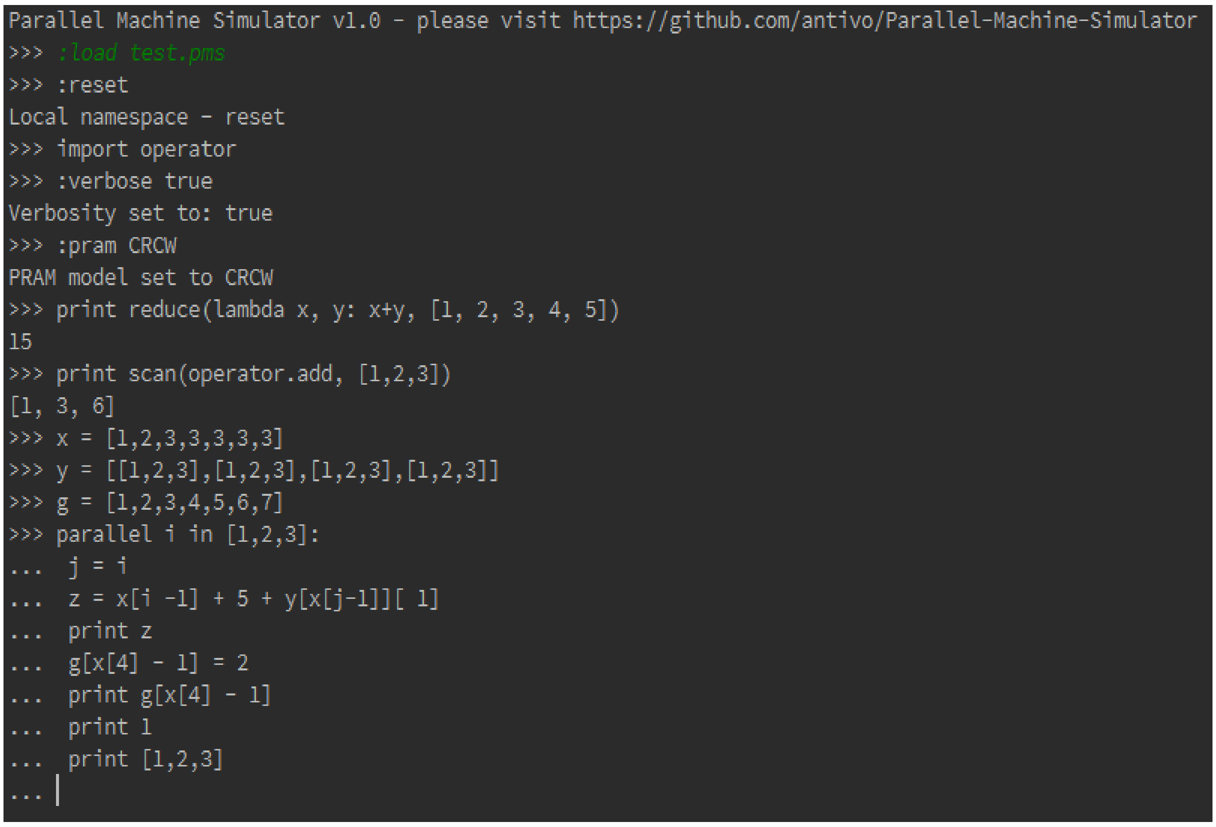


```
Parallel Machine Simulator v1.0 - please visit https://github.com/antivo/Parallel-Machine
>>> :load test.pms
>>> :reset
Local namespace - reset
>>> :verbose true
Verbosity set to: true
>>> :pram ERCW
PRAM model set to ERCW
>>> x = [1,2,3,3,3,3,3]
>>> y = [[1,2,3],[1,2,3],[1,2,3],[1,2,3]]
>>> g = [1,2,3,4,5,6,7]
>>> parallel i in [1,2,3]:
...   j = i
...   z = x[i -1] + 5 + y[x[j-1]][ 1]
...   print z
...   g[x[4] - 1] = 2
...   print g[x[4] - 1]
...   print 1
...   print [1,2,3]
...
---------------------Node with i = 1
8
3
1
[1, 2, 3]
--------IL: i, j, z
--------RL: y[1][1], x[0], i, j, z, x[4], g[2]
--------WL: j, z, g[2]
---------------------Node with i = 2
9
Joint memory reading constraint violation. Multiple reading from memory locations: x[4]
>>>
```

Figure 2 Shared memory violation

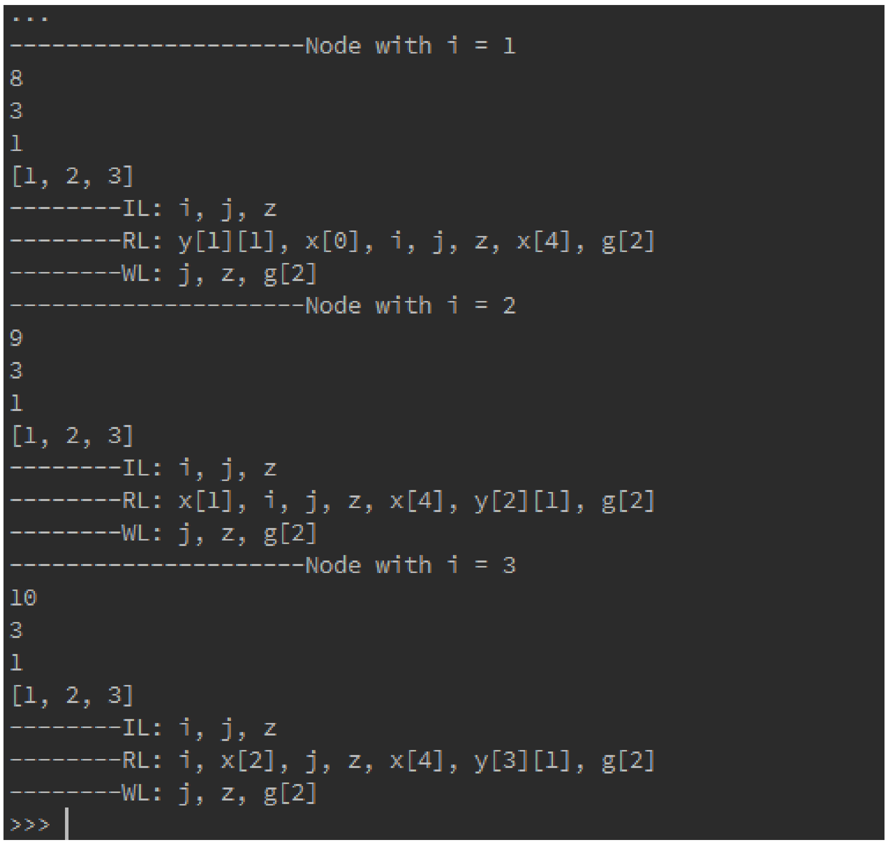
5.3. CRCW example

In the Figure 3 CRCW algorithm we can observe the same problem with major difference, model is set to CRCW. In this model no shared model violation should occur. In the Figure 4 CRCW algorithm execution we can see the results of this algorithm on a CRCW model, due to verbosity command that has informed PMS to show additional information that is available. Output from the processors is divided by information of which values were assigned to nodes, from which locations those nodes were reading and to which location were writing as well as memory locations that would be ignored due to being associated to that node.



```
Parallel Machine Simulator v1.0 - please visit https://github.com/antivo/Parallel-Machine-Simulator
>>> :load test.pms
>>> :reset
Local namespace - reset
>>> import operator
>>> :verbose true
Verbosity set to: true
>>> :pram CRCW
PRAM model set to CRCW
>>> print reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])
15
>>> print scan(operator.add, [1,2,3])
[1, 3, 6]
>>> x = [1,2,3,3,3,3,3]
>>> y = [[1,2,3],[1,2,3],[1,2,3],[1,2,3]]
>>> g = [1,2,3,4,5,6,7]
>>> parallel i in [1,2,3]:
...  j = i
...  z = x[i -1] + 5 + y[x[j-1]][ 1]
...  print z
...  g[x[4] - 1] = 2
...  print g[x[4] - 1]
...  print 1
...  print [1,2,3]
...
```

Figure 3 CRCW algorithm



```
...
----------------------Node with i = 1
8
3
1
[1, 2, 3]
--------IL: i, j, z
--------RL: y[1][1], x[0], i, j, z, x[4], g[2]
--------WL: j, z, g[2]
----------------------Node with i = 2
9
3
1
[1, 2, 3]
--------IL: i, j, z
--------RL: x[1], i, j, z, x[4], y[2][1], g[2]
--------WL: j, z, g[2]
----------------------Node with i = 3
10
3
1
[1, 2, 3]
--------IL: i, j, z
--------RL: i, x[2], j, z, x[4], y[3][1], g[2]
--------WL: j, z, g[2]
>>>
```

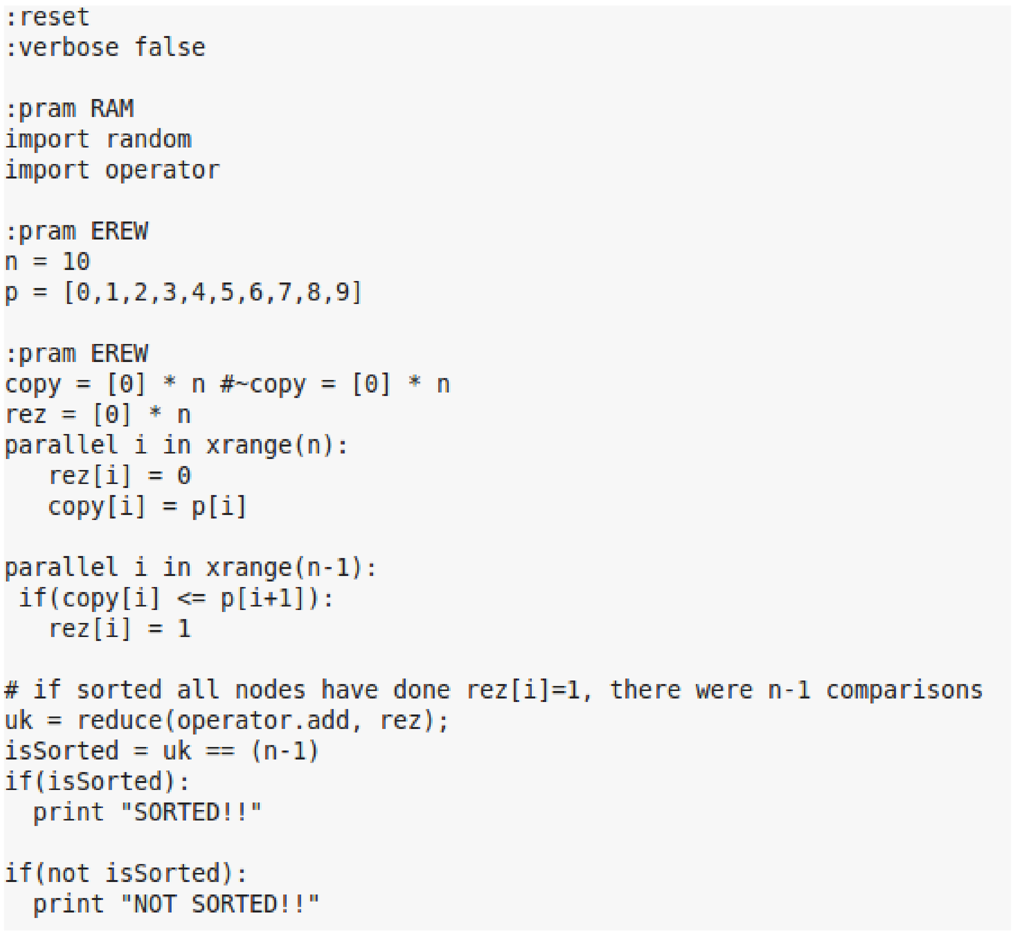
Figure 4 CRCW algorithm execution

5.4. EREW algorithm example

Let’s write an EREW algorithm that answers the question is the given array sorted in ascending order. If for each two adjacent elements in the input array left element is lesser than the right element of the pair we can say that array is sorted in ascending order.

Each PRAM processor can take care of one such pair and compare them. Because each node is making comparison between two nodes of the original array we will make a new array that is copy of original array. With this approach we will not violate any shared memory constrains. The result of comparisons we will store in separate array. If the total number of comparisons where left element was lesser than the right was equal to the number of total comparisons array is sorted in ascending order, otherwise it is not.

In Figure 5 EREW algorithm for determining if array is sorted in ascending order is presented this algorithm written for Parallel Machine Simulator. Code that can be seen in Figure 5 EREW algorithm for determining if array is sorted in ascending order written for Parallel Machine Simulatorstarts with statements that reset local namespace and decides not to take additional information that is available to the user during execution. When importing modules memory model must be changed to RAM because include statement can not be interpreted in any of the PRAM models in this current version of PMS. After all the preparations have been made it is usual to set desired memory model. In this case it is EREW. In this example we have hardcoded array of 10 elements sorted in ascending order from 0 to 9. Using the parallel loop we are creating copy of input array and initializing container for results of comparisons in constant time. Again using parallel block we are doing comparisons in constant time. When comparisons are done all that is left is using reduce function to count all the correctly ordered pairs. If that number is equal to the n-1 (because there are n-1 comparisons) we say the array is sorted, otherwise we say it is not.



```
:reset
:verbose false

:pram RAM
import random
import operator

:pram EREW
n = 10
p = [0,1,2,3,4,5,6,7,8,9]

:pram EREW
copy = [0] * n #~copy = [0] * n
rez = [0] * n
parallel i in xrange(n):
   rez[i] = 0
   copy[i] = p[i]

parallel i in xrange(n-1):
 if(copy[i] <= p[i+1]):
   rez[i] = 1

# if sorted all nodes have done rez[i]=1, there were n-1 comparisons
uk = reduce(operator.add, rez);
isSorted = uk == (n-1)
if(isSorted):
  print "SORTED!!"

if(not isSorted):
  print "NOT SORTED!!"
```

Figure 5 EREW algorithm for determining if array is sorted in ascending order written for Parallel Machine Simulator

6. Conclusion

We believe that PMS will prove to be useful tool in designing algorithms for PRAM. Currently there are no similar open source solutions that we have encountered.

7. Literature

1. Jakobovic, Domagoj: “Predavanja iz kolegija Paralelno Programiranje“, 2015
2. Blelloch, Guy: “Prefix Sums and Their Applications“, School of Computer Science, Carnegie Mellon University, Pittsburgh
3. Skiena, Steven: “The Algorithm Design Manual (2nd ed.)“, Springer Science+Business Media, 2010
4. Tvrdik, Pavel: “PRAM model“,

   http://pages.cs.wisc.edu/~tvrdik/2/html/Section2.html, 1999
5. “Python – Tutorial“, http://www.tutorialspoint.com/python/
6. Juneau, Josh; Baker, Jim; Ng, Victor; Soto, Leo; Wierzbicki, Frank: “The Definitive Guide to Jython“,

   http://www.jython.org/jythonbook/en/1.0/index.html, 2010
7. “The Python Language Reference“, https://docs.python.org/2/reference/, 2015
8. Keller, Jörg; Kessler, Christoph W.; Träff, Jesper L.: “Practical PRAM Programming“, http://www.ida.liu.se/~chrke55/ppp.html, 2001
9. “Parallel random-access machine“, http://en.wikipedia.org/wiki/Parallel_random-access_machine, 2015
10. “Random-access machine“, http://en.wikipedia.org/wiki/Random-access_machine, 2014
11. “Interpreter (computing)“, http://en.wikipedia.org/wiki/Interpreter_%28computing%29, 2105
12. “Interpreted language“, http://en.wikipedia.org/wiki/Interpreted_language, 2015
13. “Block (programming)“, http://en.wikipedia.org/wiki/Block_%28programming%29, 2015
14. “Apache Maven“, http://en.wikipedia.org/wiki/Apache_Maven, 2015
15. “Python (programming language)“, http://en.wikipedia.org/wiki/Python_%28programming_language%29, 2015
16. “Spring Framework“, http://en.wikipedia.org/wiki/Spring_Framework, 2015