PySy: A Python Package for Enhanced Concurrent Programming

By

TODD WILLIAMSON
B.S (University of California at Davis) 2007

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Professor Ronald A. Olsson, Chair

Professor Norman S. Matloff

Professor Aaron W. Keen

Committee in charge

2012

To my family...

Contents

List of Figures

List of Tables

Abstract

Over the last decade, the popularity of Python has increased considerably. Python is widely used and has been demonstrated to be effective over many problem domains including scripting, prototyping, and simulation. Python's easy to use and concise syntax is highly expressive and allows a developer to create considerably shorter and easier to understand programs than semantically equivalent programs written in languages like C, C++, or Java. An important aspect of any language's flexibility is a highly parallelizable environment that allows its users to write concurrent programs. However, Python is still lacking a high-level, expressive concurrent and distributed programming environment. This thesis presents our experience creating PySy, a Python package (based on the SR and JR concurrent programming languages), which provides an easy to use and expressive concurrent environment, and allows for the distributed sharing of resources. Throughout this thesis, we will discuss our design decisions, implementation, show qualitative and quantitative analyses of well-known concurrent programs written in PySy, and evaluate our methodology.

Chapter 1

Introduction

Over the last decade, the popularity of Python [22] has increased considerably. Now, Python is widely and effectively used over many problem domains including scripting, prototyping, web development, and simulation. Python's easy to use and concise syntax is highly expressive and allows a developer to create considerably shorter and easier to understand programs than functionally equivalent programs written in languages like C, C++, or Java. An important aspect of any language's flexibility is a highly parallelizable environment that allows its users to write concurrent programs. Many programs achieve increased performance and are more easily modeled using concurrency, e.g., web servers. Over the years, many Python concurrency packages have been developed, including thread-based and process-based implementations. The traditional threading solutions, e.g., **threading**[1] [22], do not allow for true concurrency in CPython because the interpreter provides mutual exclusion using a global resource, the Global Interpreter Lock (GIL), to serialize access to its internals. The GIL effectively bottlenecks competing threads trying to obtain access to the interpreter's internal data structures. **multiprocessing** [22], an example of a process-based implementation, was developed to bypass the GIL limitation by executing each thread in its own interpreter process. However, **multiprocessing** lacks expressiveness. Python is widely used and adored because it provides libraries with high-level abstractions that allow rapid prototyping, but the most widely used concurrency package, **multiprocessing**, only

provides low-level features that the user must build upon to model complex concurrent programs. We introduce PySy as a way to achieve true concurrency in Python and to provide an interface that easily expresses high-level concurrency and distributed programming mechanisms.

This thesis presents our experience creating PySy, a Python package, which provides many of the JR [17] language features. Chapter 2 reviews relevant Python and JR concepts. Chapter 3 discusses PySy's feature design and demonstrates how to utilize these features. Chapter 4 presents PySy's implementation. Chapter 5 presents and analyzes PySy's performance results for several benchmarks and well-known concurrent programs. Chapter 6 discusses ideas for future work on PySy and presents our conclusions.

[1]Python packages and modules are displayed in bold.

Chapter 2

Background

This chapter reviews relevant background material. Section 2.1 reviews relevant Python concepts. Section 2.2 reviews the JR concurrent programming language.

2.1 Python

This section gives an overview of advanced features and internal nuances of the Python[22][5] language (version 2.x). We assume the reader is moderately familiar with Python's basic syntax and has object oriented programming experience in some language. Most of PySy's design uses basic, fundamental Python concepts, but it also uses some advanced techniques and subtle nuances of the language to enhance usability, which we will discuss in this section. In some situations, we will directly refer to a specific Python implementation because PySy uses CPython, in which the Python interpreter is implemented in C. Sections 2.1.1, 2.1.2, and 2.1.3 review relevant Python concepts. Section 2.1.4 reviews Python's scoping rules. Section 2.1.5 discusses the implications of the Global Interpreter Lock in CPython. Section 2.1.6 reviews existing Python concurrency packages.

2.1.1 Callables

A callable is a special type of Python object that refers to the user's ability to invoke a method using the traditional method invocation syntax. For example, if we have function f[1] with three parameters, it may be invoked by `f(p1,p2,p3)`[2]. Python also supports default parameters and variable number of parameters as arguments. So, f(p1, p2) is also valid if p3 has a default value or if f's definition takes a variable number of arguments, e.g., `def f(*args)`. Figure 2.1 shows the C source code that checks if a Python object is callable. It has two cases: x is an object instance that defines the method *__call__* or *tp_call* is not NULL. The variable *tp_call* [22] is an optional pointer to a function, which is NULL if the object is not callable.

```
int PyCallable_Check(PyObject *x)
{
    if (x == NULL)
        return 0;
    if (PyInstance_Check(x)) {
        PyObject *call = PyObject_GetAttrString(x, "__call__");
        if (call == NULL) {
            PyErr_Clear();
            return 0;
        }
        Py_DECREF(call);
        return 1;
    }
    else {
        return x->ob_type->tp_call != NULL;
    }
}
```

Figure 2.1: C API call for checking if a Python object is callable.

2.1.2 Python Methods

Python has three method types that its users interact with (whether the user is aware of it or not): *functions*, *unbound methods*, and *bound methods*. Figure 2.2 highlights

[2]Variable and method names are italicized.
[2]Source code displayed directly in the text is in verbatim.

the differences. When discussing the different method types, we may also refer to functions as user-defined functions and the two method types as user-defined methods, where appropriate, to avoid confusion with other usages of these terms.

```
def f(): pass
print f

class Foo:
  def g(): pass

print Foo.g

foo1 = Foo()
print foo1.g

#example output:
#<function f at 0x1004b5ed8>
#<unbound method Foo.g>
#<bound method Foo.g of <__main__.Foo instance at 0x1004d3050>>
```

Figure 2.2: Examples of the three main types of Python methods.

A user-defined function is any function created by using the Python keyword **def**. Table 2.1 [22] describes the special attributes belonging to functions. One of the interesting special function attributes, from a developer's perspective, is the dictionary *func_dict*. Given function *f*, it is legal in Python to create attributes for *f* and assign these attributes values. So, the statement `f.x=True` is legal. A newly defined function is stored in the current scope's dictionary. If the function *f* is declared within a class C, then the function object will be available through C's dictionary `C.__dict__['f']`. `C.f`, however, returns an unbound method and not the function *f*. This will be discussed later in this section.

A user-defined method is a special type of function with three extra read-only attributes: *im_class*, *im_self*, and *im_func*. *im_class* is a reference to the class object that encloses the function definition. *im_self* is a reference to the class instance or None if the method is unbound. *im_func* is a reference to the callable object associated with the method (usually of type function). The attributes associated with the callable *im_func* are now read-only. The statement `f.x=True` is now illegal and will throw an AttributeError

Attribute	Meaning	Access
func_doc	The function's documentation string, or None if unavailable	Writable
func_name	The function's name	Writable
__module__	The name of the module in which the function was defined, or None if unavailable	Writable
func_defaults	A tuple containing default argument values for those arguments that have defaults, or None if no arguments have a default value	Writable
func_code	The code object representing the compiled function body	Writable
func_globals	A reference to the dictionary that holds the function's global variables – the global namespace of the module in which the function was defined	Read-only
func_dict	The namespace supporting arbitrary function attributes	Writable
func_closure	None or a tuple of cells that contain bindings for the function's free variables	Read-only

Table 2.1: Special attributes for Python functions

exception, but the reference `f.x` is still legal. The main difference between unbound and bound methods is the value inside of the variable *im_self*.

Previously in this section, we mentioned that if class C has a function f, then `C.f` returns an unbound method, but `C.__dict__['f']` returns the function object. The expression `C.f` is translated to `C.__dict__['f'].__get__(None, C)`. Let the variable *c1* be an instance of class C. The expression `c1.f` is translated to `C.__dict__['f'].__get__(c1, C)`. It is Python's descriptor [22] protocol that transforms a function object to an unbound or bound method object. This transformation happens on *every* attribute access. This is why the simple optimization of assigning a heavily used user-defined method to a local variable will improve performance (see Figure 2.3).

```
import time

class Foo(object):
  def __init__(self):
    self.x = 0
  def increment(self):
    self.x += 1
niters = 1000000

f = Foo()
start = time.time()
for i in xrange(niters):
  f.increment()
end = time.time()
origTime=  end - start
print "Original timing: %f" % (origTime)

f = Foo()
increment = f.increment
start = time.time()
for i in xrange(niters):
  increment()
end = time.time()
optimizedTime =  end - start
print "Optimized timing: %f" % (optimizedTime)

print "Overall speedup: %f percent" % (
  (1.0 - (optimizedTime / origTime)) * 100)
"""output:
Original timing: 0.445786
Optimized timing: 0.365727
Overall speedup: 17.959024 percent
"""
```

Figure 2.3: Example showing the descriptor protocol overhead

2.1.3 Decorators

Python supports the altering of methods or classes via decorators [20] [19] (as in Aspect Oriented Programming). The decorator must be a callable object (see Section 2.1.1) and must return a callable object. Python began using decorators in version 2.2 with the usage of **classmethod** and **staticmethod**, but user-defined decorators were not supported until version 2.4. Figure 2.4 shows two semantically equivalent ways to decorate a function in Python. The first uses function composition, and the second utilizes the syntax of the language, an @ symbol followed by a callable object. The user may decorate a function

any number of times. To additionally add a decorator to the first method, the user simply must add another function to the function composition. For the second method, the user must use the decorator syntax on each line preceding the function. The order in which the modifications are applied is similar to the evaluation of functional compositions in mathematics. For example, $(f \circ g \circ h)(x)$ is evaluated as $f(g(h(x)))$.

The *logFunctionCall* method in Figure 2.4 takes m as a parameter, which corresponds to the method we wish to decorate. Next, we create a new function *wrapped* that will act as the new decorated method. *wrapped* gets its own read-only copy of m because of Python closures. *wrapped* simply prints before and after we invoke m. The *logFunctionCall* method returns a reference to *wrapped*. During the class creation of *Foo*, method x is replaced with *wrapped*. Inspecting *F.x* yields an unbound method object associated with *wrapped*, not with x, and not with *logFunctionCall*. The same applies to method y except it will have its own separate *wrapped* function object and the creation of y uses the old, but semantically equivalent, style for decorator creation.

```
def logFunctionCall(m):
  def wrapped(*args):
    print "entering %s" % m.__name__
    m(*args)
    print "exiting %s" % m.__name__
  return wrapped

class Foo:
  def y(self, val):
    print "in y %d" % val
  y = logFunctionCall(y)

  @logFunctionCall
  def x(self, val):
    print "in x %d" % val


f = Foo()
f.x(1)
f.y(2)
```

Figure 2.4: Two equivalent ways to decorate a function in Python

```
class LogFunctionCall(object):
    def __init__(self, m):
        self.m = m

    def __call__(self, *args):
        print "entering %s" % self.m.__name__
        self.m(*args)
        print "exiting %s" % self.m.__name__

@LogFunctionCall
def f():
  print "in f"
f()
#outputs:
#entering f
#in f
#exiting f
```

Figure 2.5: Example showing how to use a Python class as a decorator.

2.1.4 Scoping

Python's scoping is slightly simpler than that in languages like C, C++, and Java. One of the main differences is that code blocks introduced by conditionals and loops do not create a new scope. In Python, each file has its own global scope and only the method construct (method definition or lambda definition) adds a new scope. When trying to resolve a variable, the Python interpreter first inspects the current scope, next it iteratively checks the parent scopes (ending with the global scope), and finally checks the Python built-ins. An *AttributeError* is thrown when the interpreter cannot resolve a variable.

Python also provides lexical closures. When a function is declared, the function inherits its parent's local variables and is given read-only access to these variables referred to as closed-over variables. The subsequent code examples demonstrate the behavior of Python closures. In the program from Figure 2.6, Python is able to resolve x to the closed-over variable x from *main*. In the program from Figure 2.7, Python complains that *foo* is trying to read and write to a closed-over variable (**print** reads the value of x and the assignment is trying to write to x). The method *foo*, in the program from Figure 2.8, does not use the closed-over variable defined in *main*. Instead, it creates a new variable x in its

own scope and prints it out.

```
def main():
  x = 2
  def foo():
    print x
  foo()
  #expected output:
  #2
```

Figure 2.6: Program showing a legal access under Python's closure scoping.

```
def main():
  x = 2
  def foo():
    print x
    x = 2
  foo()
  #expected output:
  #exception
```

Figure 2.7: Program showing illegal access under Python's closure scoping.

```
def main():
  x = 2
  def foo():
    x = 3
    print x
  foo()
  print x
  #expected output:
  #3
  #2
if __name__ == "__main__":
  main()
```

Figure 2.8: Program showing a legal access under Python's closure scoping, but behaves in a counterintuitive way.

2.1.5 GIL Limitations

The Python (CPython only) interpreter requires a global lock (Global Interpreter Lock or GIL) to ensure thread-safety. The GIL is responsible for guaranteeing that only a single thread has access to Python objects or is invoking Python/C API functions at any given time. With this restriction, true thread-level parallelism is impossible. The GIL, for each Python interpreter, will always be a bottleneck for competing threads. The Python interpreter, however, tries to emulate parallelism by regularly switching threads and attempting to release the GIL on potentially blocking operations (mostly I/O) or CPU intensive computations (e.g., hashing).

2.1.6 Multithreading Alternatives

2.1.6.1 The Python threading Module

The Python **threading** module [22] is a higher-level interface for Python's **thread** module. **threading** provides a nice interface for the user to create simple concurrent programs. With **threading**, the user can create an object that derives from *threading.Thread* or define a function to run in its own thread. Figure 2.9 shows how to execute the code of a function in its own thread. The threading module allows the user to take advantage of writing concurrent programs, however, because of the GIL limitations in CPython, **threading** will not generally provide improved performance in CPU-bound programs.

2.1.6.2 multiprocessing

The **multiprocessing** package overcomes the GIL limitations by spawning a new interpreter process for each process declaration. This implementation allows for Python programs to achieve true concurrency and reap potential performance improvements on CPU-bound tasks. The **multiprocessing** interface is very similar to the **threading** interface. Figure 2.10 shows how to execute the code of a function in its own process. **multiprocessing** also provides several distributed programming abstractions. See Appendix F

```
import threading

lock = threading.Lock()
x = 0

def work():
  global lock, x
  for i in xrange(100):
    with lock:
      x+=1


t1 = threading.Thread(target=work)
t2 = threading.Thread(target=work)
t1.start()
t2.start()
t1.join()
t2.join()
print x
```

Figure 2.9: Example using Python's threading module.

for a distributed **multiprocessing** implementation of a Fast Fourier Transform (FFT) algorithm.

```
from multiprocessing import Process, Lock, Value

def work(x):
  for i in xrange(100):
    x.value+=1

if __name__ == "__main__":
  x = Value('i', 0, lock=True)

  t1 = Process(target=work, args=(x,))
  t2 = Process(target=work, args=(x,))
  t1.start()
  t2.start()
  t1.join()
  t2.join()
  print x.value
```

Figure 2.10: Example using Python's multiprocessing module.

2.1.6.3 River/Trickle

River [9] [26] is a Python framework for distributed computing developed at the University of California, San Francisico. It borrows some of its concepts from the SR [4] [28] language, the predecessor to JR [17] [27], on which PySy is based. While there are a lot of similarities between PySy and River, PySy employs a slightly more complex model and, because of this, is capable of expressing higher-level concurrency mechanisms, e.g., an input statement and dynamically scheduling the servicing of invocations. River only explicitly provides asynchronous message passing, but can simulate synchronous message passing. The River messaging model sends messages to specific processes, while PySy sends messages to a specific object within a process.

A River program consists of at least one Virtual Machine (VM), which is a Python interpreter process. River users manually launch VM processes on the local and remote machines used in the distributed computation. The River run-time system coordinates the allocation and deployment of the VMs through a user-interface in the main program. The River framework is highly extensible, which has allowed its developers to implement a task-based parallelism extension Trickle and a standard MPI concurrency extension with rMPI [26]. Trickle hides a lot of the necessary River start-up code from the user and provides an easy to use interface to inject code and distribute work onto many machines. We provide qualitative and quantitative analysis of several River/Trickle programs in Chapter 5.

2.1.6.4 Other Alternatives

Over the years, many third-party Python concurrency packages have been developed, implementing various concurrency models. **PyCSP** [23] implements the Communicating Sequential Processes [11] (CSP) model. In CSP, programs are written with a fixed number of sequential processes and communicate solely through synchronous message passing. This differs from the PySy model because PySy provides both synchronous and asynchronous message passing and dynamic process creation. Also, several Python packages have implemented the standard MPI [2] concurrency model. The MPI packages include

mpi4py [21], **pypar** [24], and **rMPI**. Finally, much recent work has focused on task-based parallelism in Python, e.g., Celery [18]. Celery allows the user to mark specific functions as tasks and push these tasks to a work queue that is shared by all of the nodes connected to the system. The features provided by task-based systems are a subset of PySy's features.

2.2 JR

JR [17] [27] is an extended Java language that provides a concurrency model based on the SR [4] [28] language. It is used at several universities worldwide as both a research and an educational tool in undergraduate and graduate courses.

2.2.1 Virtual Machines

JR provides users the ability to write distributed programs. This is possible through the concept of Virtual Machines (VMs). A JR VM is an address space on a physical machine (possibly remote). The JR VM is a Java VM with an additional layer to provide the functionality for JR's concurrency model. Each JR program contains at least one VM called the main VM. The execution of JR programs begins inside of the main VM, and resides on the physical machine on which the user initiated the program. The user may explicitly create new VMs by using the **vm** keyword and calling the VM constructor (see Figure 2.11). By default, VMs are created on localhost, but the user may also specify the hostname of the system to create the new VM or a reference to an existing VM.

JR also provides parameterized VMs. JR provides the user the ability to derive its own classes from the *vm* class, define new operations for the VM, and pass user-specified parameters to the constructor. Figure 2.12 shows an example of parameterized VMs in JR.

2.2.2 Remote Object Creation

In JR, the user creates remote objects utilizing the **remote** keyword. Figure 2.13 shows how the user may specify which VM to create the new object on or, by default, create

```
public class jrVMInstantiation{
  public static void main(String[] args)
  {
    vm c = new vm();    //created a new vm on localhost
    vm d = new vm() on "pc12";  //created a new vm on pc12
    vm e = new vm() on d; //created another new vm on pc12
  }
}
```

Figure 2.11: Example showing how to create VMs in JR.

```
public class Main {
  public static op void myOp(int x);
  public static cap void (int) myCap;
  public static void main(String [] args) {
      myCap = myOp;
      // test capability
      vm vm1 = new Myvm(10, "vm1", myCap);
      System.out.println(vm1.GetID());
  }
}

public class Myvm extends vm  {
  private int vm_id;  // test primitive type
  private String vm_name; // test string
  private cap void (int) vm_cap;  // test capability and operation

  public Myvm(int vm_id, String vm_name, cap void (int) vm_cap) {
    this.vm_id = vm_id;
    this.vm_name = vm_name;
    this.vm_cap = vm_cap;
  }

  public op int GetID() {
    return vm_id;
  }

  public op String GetName()  {
    return vm_name;
  }

  public op cap void (int) GetCap() {
    return vm_cap;
  }

}
```

Figure 2.12: JR example showing the creation of a parameterized VM.

the new object on the main VM.

```
public class RemoteInstantiation{
  public static void main(String[] args){
    vm lh = new vm(); //vm on localhost
    vm rem = new vm() on "pc11";  //vm on host pc11
    remote Foo foo = new remote Foo();  //created on the current VM
    remote Foo fooRem = new remote Foo() on rem; //created on vm rem
  }
}

public class Foo{
  public Foo(){}
  public process p1{
    System.out.println("p1");
  }
}
```

Figure 2.13: How to create one remote object in JR on localhost and another on a remote machine.

2.2.3 Operations

When writing a concurrent program, there must be a way for each parallel code segment to communicate (e.g., shared memory or message passing) with the others. JR provides process-level shared memory (all threads on a particular VM share static variables), as well as several other message passing communication mechanisms through the operation abstraction. JR defines two types of operations: OpMethods (or ProcOps) and InniOps. The name of each of these operations is derived from how the operation is serviced. That is, OpMethods are serviced by methods (procedures) and InniOps are serviced by an input statement (called an Inni statement in JR). The act of sending a message via an operation is called an invocation. Operations are invoked using the **call** statement or the **send** statement. A synchronous invocation is accomplished through the **call** statement, while an asynchronous invocation is accomplished through the **send** statement. Synchronously invoking an operation named *op* has the form:

`call op(param_1, param_2,..., param_n)` or

`op(param_1, param_2,..., param_n)`

The second option is used when the user wants to use the operation's return value. Asynchronously invoking an operation named *op* has the form:

`send op(param_1, param_2,..., param_n)`

The overall effect of an invocation is determined by how the operation is invoked and how the invocation is serviced. Table 2.2 outlines the various possible behaviors using operations. Synchronously invoking an operation forces the invoker to wait until the servicer contacts the invoker, allowing it to proceed. Asynchronously invoking an operation immediately transmits the message and the invoker does not wait for a response. Instead, the invoker continues execution with the next statement.

Invocation	Service	Effect
call	OpMethod	procedure call (possibly remote)
call	Inni	rendezvous
send	OpMethod	dynamic process creation
send	Inni	asynchronous message passing

Table 2.2: Possible ways an operation is invoked and serviced.

2.2.3.1 OpMethods

OpMethods are very similar to Java function calls. From the user's perspective, the main difference from the Java method declaration syntax is that an OpMethod declaration adds an extra keyword **op**. The OpMethod declaration is followed by a block of code that will be executed when the OpMethod services an invocation. When servicing a **send** invocation, the OpMethod will execute the OpMethod's code block in a different thread of control (dynamic process creation). If the OpMethod servicing the **send** invocation contains

a return statement, then the overall effect of the return is a no-op because the invoker is not waiting for a response. When servicing a **call**, again the OpMethod's code block is executed, but after the block is finished executing, the servicer transmits a message to the invoker containing a return value, if any, and the invoker continues execution (Remote Method Invocation). Figure 2.14 shows how to define and invoke OpMethods.

```
public class jrOpMethod{
  public static void main(String[] args){
    send twotimes(12);
    System.out.println(twotimes(13));
    call twotimes(14);
  }
  public static op int twotimes(int x){
    System.out.println("in twotimes, got " + x);
    return 2*x;
  }
  /*
  possible output:
  in twotimes, got 12
  in twotimes, got 13
  26
  in twotimes, got 14
  */
}
```

Figure 2.14: How to define and invoke an OpMethod in JR.

JR also allows the user to designate a single thread or multiple threads to start up with the construction of an object. This is done through the **process** keyword. JR processes are different from the OS definition of a process. In JR, a process is a separate thread of execution within the address space of a virtual machine. The process construct is declared in two possible ways:

`process process_id block` or

`process process_id( (quantifier), (quantifier), ... ) block`

See Section 2.2.8 for a review of the quantifier syntax and its uses. The first declaration allows for a single process to be created per object, while the second declaration may potentially create many processes. The user may also modify the process declaration with

the static keyword, which only creates a new process (or processes if the process is defined with a quantifier) once per VM.

```
public class jrProcesses{
  public static void main(String[] args){
    new Test(); new Test();
  }
}
class Test{
  public Test(){}
  public process p1{
    System.out.println("printed once per Test instantiation");
  }
  public static process p2{
    System.out.println("printed once per VM");
  }
  public process p3((int i = 0; i < 6; i++; i % 2 == 0)){
    System.out.println("p3: " + i);
  }
}
  /*possible output:
  printed once per VM
  p3: 0
  p3: 2
  printed once per Test instantiation
  p3: 4
  p3: 2
  p3: 4
  printed once per Test instantiation
  p3: 0
  */
```

Figure 2.15: JR example that demonstrates how to use the process construct.

2.2.3.2 InniOps

InniOps are shared queues used to pass messages between processes. By invoking an InniOp, synchronously or asynchronously, an invocation is placed into a queue, called the invocation list. During synchronous invocations, the invoker will wait until the invocation is serviced by an input statement or a receive (equivalent to a one arm input statement). Otherwise, in an asynchronous invocation, the invocation is immediately added to the queue, and the invoker continues execution with the next statement.

2.2.3.3 Semaphores

Semaphores [8] are a well-studied concurrent programming mechanism to restrict access to resources. JR provides the semaphore abstraction to its users in the guise of an InniOp. In its most simple form, a semaphore is a counter for the number of available resources. Underneath, an InniOp is a synchronized shared queue (the invocation list). We can simulate a semaphore by allowing the number of invocations in the queue to represent the number of available resources. Traditionally, semaphores define two operations: P and V. The P operation tests the availability of a resource and decrements the counter if a resource is available and blocks otherwise. The V operation releases a resource and, likewise, increments the number of available resources. Figure 2.16 shows a simple JR program where a semaphore is required to achieve the desired behavior.

```
public class Sem{
  public static sem s = 1;
  public static int count = 0;
  public static op void done();
  public static void main(String[] args){
    receive done();receive done();
    System.out.println(count);
  }
  public static process pq((int pi = 0; pi < 2; pi++)){
    for(int i = 0; i < 50; i++){
      P(s);
      count += 1;
      V(s);
    }
    send done();
  }
}
```

Figure 2.16: JR example that demonstrates the semaphore construct.

2.2.4 Input Statement

The input (Inni) statement is JR's most powerful statement and gives the user great flexibility. The Inni statement services InniOp invocations. Syntactically, the Inni statement contains one or more op commands separated by a box token with either an

optional else or else after arm:

```
inni op_command1  [] op_command2 [] ...
```

Each *op_command* is referred to as an arm and consists of an operation and an optional such that clause and by expression. The such that clause is a synchronization expression that only allows an invocation to be serviced when the user-defined expression evaluates to true. The by expression modifies the scheduling of invocations. When a by expression is present, the invocation with the lowest evaluation is serviced first. Section 2.2.5 discusses invocation selection further.

When an Inni statement is used to service an invocation, the selected arm's code block is executed. For synchronous invocations, the the Inni transmits a message to the invoker after a reply or return statement is executed and the invoker continues execution. The overall effect is synchronization between the invoker and the servicer (rendezvous). Otherwise, no message is sent to the invoker and the effect is asynchronous message passing.

Figure 2.17 shows a JR program with an input statement with two arms. The main program adds two invocations to f's invocation list and one invocation to g's invocation list. Process P1's Inni statement services all of the invocations for the operations in its arms. When no serviceable invocations exist, the program has reached a quiescent state (see Section 2.2.9), and JR terminates the program. Figure 2.18 shows a JR program with an input statement that has a by clause and a such that expression. The invocation selection for Inni statements with by expressions and such that clauses changes. The by expression orders the serviceable invocations by the lowest evaluation, while the such that clause selects the first serviceable invocation. The next section presents JR's invocation selection semantics.

2.2.5 Invocation Selection

Recall from Section 2.2.3 that operations are invoked either asynchronously or synchronously, and serviced by methods or an input statement. The way a language (or package) chooses to select invocations for servicing will change its semantics, notably, in-

```
public class jrInni2Arms
{
  public static op void f(int);
  public static op void g(int);
  public static void main(String[] args)
  {
    send f(2);send f(3);send g(1);
    call g(6);
  }
  public static process P1{
    while(true){
      inni void f(int x){
        System.out.println(x);}
      [] void g(int x){
        System.out.println(x);}
    }
  }
  /*
  expected output:
  2
  3
  1
  6
  */
}
```

Figure 2.17: JR Inni statement with two arms.

```
public class jrInniByST
{
  public static op void f(int);
  public static void main(String[] args){
    send f(-2);send f(3);send f(1);send f(-1); send f(2);
    while(true){
      inni void f(int x) st x > 0 by -x {
        System.out.println(x);
      }
    }
  }
  /*
  expected output:
  3
  2
  1
  */
}
```

Figure 2.18: JR Inni statement with by clause and such that expression.

vocation fairness and servicing fairness (called process fairness in [16]). Invocation fairness describes the fairness associated with how the servicing algorithm selects the next invocation to be processed. Servicing fairness describes the fairness associated with selecting which Inni statement services an invocation. Both of these concepts are key to understanding how a concurrent program will behave. They describe the guarantees given to the programmer concerning expected program behavior. Consider the program in Figure 2.19. Imagine the user was given no guarantees about which process will service invocations. In this case, it would be possible, however unlikely, for P3 (or, similarly, P2) to be completely starved. The type of guarantees the user wants may vary from program to program, but it is important to provide an environment that encompasses the general desires of the intended user. JR, in particular, selects the next invocation loosely based on first-come, first-served (FCFS). Like Java, JR does not add any new thread scheduling semantics. So, JR provides servicing fairness if and only if the underlying operating system provides a fair thread scheduler.

Before we discuss the invocation selection algorithm, we must define what FCFS means with respect to invocations. In JR, when an operation is invoked, the resulting invocation is assigned a timestamp based on its VM's estimation of the current time – called distributed time. The receiving InniOp appends each invocation to its invocation list. The queued ordering of an InniOp's invocation list describes the explicit order in which its invocations will be serviced (assuming no such that clauses or by expressions). For each attempt by an Inni statement to service an invocation, JR sorts the Inni statement's arm(s) based on the distributed timestamp of the earliest invocation. We will refer to this sorting procedure as "arm sorting." If there is no such that clause or by expression, then the general servicing is extremely fast. We simply take the first element from the earliest arm's invocation list (from the ordering of the arm sorting). The servicing algorithm is now O(a log a), where a is the number of arms. Through empirical study of JR's verification suite [27], JR's book [17], and the experience obtained from taking and being a teaching assistant for ECS140B at UC Davis, I have noticed that the number of arms is generally less than four (not including the usages of quantifiers; see Section 2.2.8). So, we may safely

```
public class jrInvFairness{
  public static op void f(int);
  public static op int g(int) ;
  public static void main(String[] args){
    send f(1); send f(2); send f(6);
  }
  public static process P1{
    int count = 0;
    while(count < 10){
      inni void f(int x){
        int y = g(x);
        System.out.println("x: " + x + " y: " + y);
        send f(x+1);

      }
      [] else{break;}
          count += 1;
    }
  }
  public static process P2{
    while(true){
      inni int g(int x){
        return x+1;
     }
    }
  }
  public static process P3{
    while(true){
      inni int g(int x){
        return x*x;
      }
    }
  }
}
```

Figure 2.19: JR program with multiple processes competing for invocations.

treat the arm sorting as a constant operation in invocation selection, and only consider the number of invocations we try to service in our time bound.

We have just discussed how JR handles invocation selection in the most basic case. However, the input statement may have a by expression, a such that clause, or both. We will now discuss how the invocation selection process changes for such that clauses and by expressions, and describe the effect of these changes on our run-time bounds. First, let us examine the such that clause. The selection algorithm still performs the arm sorting. In the best case, the earliest invocation satisfies the such that clause and we are done.

Otherwise, the selection algorithm must iterate over all invocations in the selected arm until a serviceable invocation is found or all of the selected arm's invocations are not serviceable. If the invocation is not found in the first arm, then the algorithm must iterate over the remaining arms until a serviceable invocation is found or, in the worst case, all invocations are not serviceable. The invocation selection with a such that clause has a worst case running time of O(n), but may, in the best case, run in constant time.

Next, let us examine the by expression. After the arm sorting, the by expression must be evaluated for each invocation in the selected arm's invocation list, and, select the invocation with the lowest value. With a by expression, the invocation selection algorithm is guaranteed to find a serviceable invocation in the first arm. So, our running time, in this case, would be $\theta(n)$.

Finally, consider the case where we have a such that clause and a by expression. After the arm sorting, the selection algorithm iterates over all invocations in the earliest arm testing both the such that clause and evaluating the by expression. If any invocations in the first arm satisfy the such that clause, then the selection algorithm will choose the invocation with the lowest by expression evaluation. Otherwise, it will iterate over the other sorted arms and return the serviceable invocation, if any, with the lowest by evaluation in the arm. The overall time complexity for this process is O(n).

Some may wonder why JR does not provide true FCFS. This is, undoubtedly, for performance reasons. If JR was to provide true FCFS, then the invocation lists of each arm must be merged and sorted by distributed time on each servicing attempt. Let n denote the number of invocations, the selection process will take O(n log n) time or approximately the time to sort all of the invocations by distributed time. This is just for selecting the first invocation to attempt to process. There would be additional overhead with the more complicated Inni statements. Implementing the servicing in this way would severely hinder performance.

See Appendix A for more explanation of invocation selection and a walkthrough of an example.

2.2.6 Reply

The **reply** statement is used on the servicing side of an invocation. It has the form:

```
reply          or
reply expr
```

The **reply** statement returns a value to a synchronous invoker and allows the invoker to proceed. Unlike with the **return** statement, after a **reply** the servicing thread continues executing the next statement. The **reply** statement is defined for any invocation, but the invoker is only contacted in the case of a synchronous invocation. For asynchronous invocations, the overall effect of a reply is a no-op since the invoker is not waiting for a response. The **reply** statement is useful in many situations. One example is if the servicing side needs to perform some expensive post-processing code after the return value has already been computed. Making the invoker wait until after the post-processing code completes would hurt performance. With the **reply** statement, the servicing side can contact the invoker with the return value and, only then, execute the post-processing code. The user may use the **reply** statement any number of times per invocation, but the effect of the **reply** statement described above occurs only for the first **reply** statement executed for the invocation; subsequent executions of **reply** are effectively no-ops.

2.2.7 Forward

The **forward** statement passes the responsibility for servicing a synchronous invocation to another operation. It has the form:

```
forward operation(param_1, param_2, param_n)
```

The servicing side is unaware that the invocation it is servicing is the product of a forward or call. A useful example for forwarding invocations can be seen in the typical client-server model. Each client has a reference to an operation on the server. The server simply acts as an intermediary and dispatches all invocations to a threadpool or some other resource to handle the computation. To do this, the server can use the **forward** statement to push

the responsibility for servicing the incoming invocations to another resource. The **forward** statement may be used any number of times per invocation, but the effect of the **forward** described above only occurs for the first **forward**. Each subsequent execution is treated as a new send invocation.

2.2.8 Quantifiers

The quantifier syntax is nearly identical to Java's for loop syntax except for the quantifier's optional such that expression:

`(initialization_expr; termination_expr; increment_expr)` or

`(initialization_expr; termination_expr; increment_expr; st_expr)`

The such that expression defines an expression that must be satisified for the quantifier to create a new process. For example the process quantifier, `(int i = 0; i < 10; i++; i%2 == 0)` will create five total processes, one for each i in the set {0, 2, 4, 6, 8}. Quantifiers are very useful for creating families of operations.

2.2.9 Quiescence

Terminating programs is much more difficult in a concurrent environment than in a sequential program. In a sequential program, like in Java, when the main function completes, the program is terminated. In a concurrent program, the completion of one thread does not necessarily imply the program is completed. In most concurrent environments, the user is forced to explicitly write code to terminate the program. Quiescence detection [14] [15] takes this responsibility away from the user and gives it to the run-time system. A program is said to be quiescent if and only if no threads are able to run[3] and all network messages that have been sent have also been received. In addition, quiescence detection can also determine if a program is deadlocked. Quiescence detection lessens the amount of the code that the user must write to synchronize the program for termination, but is traded off with a significant performance hit. The JR book [17] discusses the quies-

cence detection's performance in its appendix and we will discuss it further in Chapter 5. By default, quiescence detection is enabled in JR programs.

JR also provides its users the ability to register a quiescence action to be executed when quiescence is detected. The quiescence action is the last code executed before beginning shutdown procedures and can be used to execute post-processing or clean up procedures for a program.

2.2.10 Concurrent Invocation Statement

The concurrent invocation statement [6] [4] or Co statement starts up a group of invocations. The Co statement allows the user to group operation invocations (also allows for quantifiers) and normally terminates when all invocations have been serviced. The Co statement also allows for the user to specify post-processing code and a mechanism for terminating the Co statement early. Figure 2.20 shows an example of JR's Co statement.

[3]Threads that are napping via JR.nap() still count as runnable threads. A thread is not runnable if it is waiting on a JR resource.

```
import edu.ucdavis.jr.JR;
public class CoExample{
  public static void main(String[] args){}
  public static process p{
    System.out.println("before co");
    co call f(2);
    [] call g(3);
    System.out.println("after co");
  }
  public static op void f(int x){
    System.out.println(x);
    JR.nap(1000);
    System.out.println("leaving f");
  }
  public static op void g(int x){
    System.out.println(x);
    JR.nap(200);
    System.out.println("leaving g");
  }
  /*expected output:
  before co
  2
  3
  leaving g
  leaving f
  after co
  */
}
```

Figure 2.20: Example showing a JR program with a co statement.

Chapter 3

Design

PySy is based on the JR concurrent programming language [17][13]. However, there are some interesting differences between the two. The most obvious difference is that PySy is in the form of a package; JR is its own language. The JR language is able to leverage the Java compiler to do a lot of its work, but PySy does not have this luxury. Instead, PySy propagates this work to the user and the developer. Comparatively, an input statement in PySy (Section 3.6) is much more complicated to use from the user's perspective than in JR (Section 2.2.4). In JR, the user simply uses the syntax of the language to utilize the input statement's functionality. In PySy, however, the user must break the input statement into a list of InniArm objects to achieve equivalent functionality. The InniArms are subsequently sent as parameters to PySy's Inni object constructor. Next, the user invokes the Inni's *service()* method to attempt to service an invocation for one of the Inni's arms. This is a recurring pattern throughout PySy's design. In other words, PySy incrementally builds JR constructs by having the user create a construct's lowest level subparts first. Next, it passes these subparts to the constructor of the next higher level abstraction. This procedure iterates until the highest level abstraction, which matches the functionality of the JR mechanism, is completely created. RJ, a Java package that provides JR functionality, is also currently being developed at UC Davis and implements this methodology. Currently, RJ does not have its own webpage, but it will be referenced

shortly from the JR webpage [27].

This chapter outlines PySy's design for each adapted JR feature. In the following sections, we will present a written description of each feature, give code examples, and describe the user interface. Initially, we will show the necessary package import statements, but to save space, subsequent feature uses will omit the necessary imports. The full source code for all of the examples in this thesis can be found at `http://csiflabs.cs.ucdavis.edu/~twilliam/PySy/thesis/PySyCodeExtract.tar.gz`. The structure of this chapter roughly parallels the structure of the JR section in the previous chapter.

Before we discuss PySy's features, we would like to point out some of the noticeable differences. Currently, PySy has adopted almost all of JR's features and has made a concerted effort to maintain JR's semantics. The following are the main differences between PySy and JR.

- Quantifiers are not yet implemented (see Section 6.2.1.1).
- Parameterized VMs are not yet implemented (see Section 6.2.1.2).
- The return value from a synchronous invocation allows for multiple values of any pickleable data type (see Section 3.5).
- Creating remote objects requires invoking a special PySy API function *createInstance* (see Section 3.3).

The end goal of PySy is to provide its users with a useful and expressive environment for implementing concurrent and distributed programs. Like JR, fault tolerance is not a priority. Currently, the user is responsible for handling remote exceptions and responding accordingly. That is, if a portion of a PySy program crashes the system will not attempt to recover from the crash and will likely terminate abruptly and leave a zombie process running on the system.

```
from PySy import *

PySy.init()
def main():
  print "hello world"
```

Figure 3.1: This example shows a "Hello World" PySy program.

3.1 Running PySy Programs

Figure 3.1 demonstrates the most basic PySy program. From the user's perspective, PySy programs appear nearly identical to regular Python programs. The most noticeable differences occur in the global scope. The user is required to import the PySy package, invoke the method PySy.init(), and to declare a main method. PySy.init() ensures that a Pyro (see Section 4.1) nameserver is currently running on the local subnet and initializes PySy's internal data structures, most notably, the Virtual Machine Manager (see Section 4.2.1) and the main VM. It is recommended that the user uses the "from" import syntax to avoid numerous imports. Appendix B presents a list and a brief description of the imported functions and objects. Once the system is started, program execution begins on the main VM in *main()*. To pass in command line arguments, the user simply needs to add the expected incoming parameters (or *args) to the formal parameter list of *main()*. PySy automatically handles the forwarding of command line parameters.

3.2 Virtual Machines

PySy, like JR (see Section 2.2.1), provides its users the ability to write distributed programs through the VM construct. Each PySy VM corresponds to a separate Python interpreter process, which gives each VM its own address space. This relationship is equivalent to the relationship JR VMs have with Java VMs. Also, like JR, every PySy program has at least one VM – the main VM. The main VM is located on the same physical machine where the user launched the PySy program. Each PySy program begins execution inside

the global scope's *main* method. Additionally, the user may create new VMs, either locally or remotely, by calling PySy.createVM() (see Figure 3.2 for the function signature).

```
def createVM(host="localhost", username=None, sshPort=22): pass
```

Figure 3.2: Method signature for creating new VMs in PySy.

3.3 Object Creation

PySy treats local object creation similar to JR. The user simply calls the object's constructor and, afterwards, may use the object normally. However, the PySy API provides *PySy.createInstance* to provide remote object creation. The user must invoke *PySy.createInstance* (see Figure 3.3) to create remote objects. The user should also use *PySy.createInstance* for objects intended to be used both locally and remotely. *PySy.createInstance* takes two parameters: a fully qualified class name and an optional VM reference where the user wants to create the object. A fully qualified class name is the path to the class definition delimited by '.'. This is similar to the Python syntax for importing modules except enclosed in quotation marks. For example, the fully qualified class name "foo.Foo" will search for the file foo.py in *sys.path* and ensure that class Foo is defined within. An exception is thrown if Python cannot find the module or if the class does not exist inside the module. The source code for foo.py does not need to be located on the remote machine. PySy handles transferring the source code, importing the module, the object creation, and the returning of a remote reference. If no VM reference is specified to *PySy.createInstance*, then the remote objects are created on the main VM. Figure 3.4 shows how to create remote objects in PySy.

```
def createInstance(fqcn, *args, **kwargs):
        """
        @summary: Creates a new instance of object $fqcn
        @param fqcn: Fully qualified class name of object to create
        @type fqcn: string
        @param args: arguments to the constructor of the object denoted
    by $fqcn
        @type args: tuple
        @keyword vm: the vm to create the new object on.  By default
    if vm is not specified objects are created on localhost
    in the main VM.  Otherwise the vm specified will be used.
        @return: <RemoteRef>
        """
  pass
```

Figure 3.3: PySy API method signature for creating remote objects

```
def main():
  #localhost is default host
  lhVM = PySy.createVM()
  remVM = PySy.createVM("localhost")
  #created on main VM
  f1 = PySy.createInstance("foo.Foo")
  f2 = PySy.createInstance("foo.Foo", vm=remVM)
  print "lh: %d" % f1.mult.call(3, 2)
  print "lh2: %d" % f2.mult.call(3, 2)

class Foo(PSObject):
  def __init__(self): pass
  @OpMethod
  def mult(self, inv):
    x,y = inv.getParameters()
    return x*y
```

Figure 3.4: How to instantiate a remote object in PySy on localhost and another on a remote VM.

3.4 Invocations

Section 2.2.3 defines the JR term invocation as the act of sending a message via the operation interface. Internally, an invocation contains contextual information including the invocation parameters and, if the invocation is synchronous, a reference to the invoker.

In JR, this context is managed internally, but PySy users must interact with invocations on the servicing side. The servicing side takes *inv* as its first actual parameter. *inv* is an instance of Invocation (see Figure 3.5 for the Invocation interface). Figure 3.6 demonstrates how to retrieve parameters from an invocation object. The reply construct is discussed in Section 3.7 and the forward construct is discussed in Section 3.8.

```
def getParameter(self, index):
  """
  @summary: Returns the parameter at index <index> in the
    invocation
  @precondition: index >= 0, index < len(self.params)
  @raise AssertionError: Raises an assertion error if the
    expression index < len(self.params) == False
  @param index: index of a requested parameter in the list
    self.params
  @type index: int
  @return: the object at self.params[index]
  """
def getParameters(self):
  """
  @summary: Returns all the parameters in the invocation
  @return: Tuple consisting of all the paramters in self.params
  """

def forward(self, fwdOp, inv=None):
  """
  @summary: Forwards an invocation to $fwdOp.  If inv is None
    then self is forwarded to $fwdOp, otherwise, inv is
    forwarded
  @param fwdOp: The operation to service the forwarding of this
    invocation.
  @type fwdOp: <Operation>
  @param inv: The invocation to forward to $fwdOp.  None, if
    we wish to forward self as the invocation
  """
def reply(self, *args):
  """
  @summary: Sends reply message to the invoking object
  @param args: Variable number of parameters to be used in the
    reply invocation
  """
```

Figure 3.5: The user interface for PySy's invocation object.

```
def main():
  inv1 = Invocation(1)
  inv2 = Invocation(4,5,6)
  print inv1.getParameter(0)  #returns the 1st parameter
  print inv1.getParameters()  #returns a tuple of all of the parameters
  x,y,z = inv2.getParameters() #assigns 1st parameter to 1st item on lhs
            #and so on
  print x,y,z
  w = inv2.getParameters()      #returns a tuple of all parameters
  print w
  #expected output:
  #1
  #(1,)
  #4 5 6
  #(4, 5, 6)
```

Figure 3.6: Example showing how to retrieve parameters from an invocation object.

3.5 Operations

Section 2.2.3 discussed OpMethods and InniOps and their usage. This section will present PySy's realization of the operation features. Appendix C shows the user interface for OpMethods and InniOps.

The return value of synchronous invocations in PySy is different than in JR. In JR, synchronously invoking an OpMethod allows the user to return any value, including primitives, back to the caller. Since JR is based on Java, an operation can only return a single value. Python is not as restrictive in both the types of return values and the number of return values. PySy embraces this flexibility and allows for the return of multiple values. Figure 3.7 shows the syntax for returning multiple values from a synchronous invocation follows normal Python conventions.

3.5.1 OpMethod

An OpMethod is a JR construct (see Section 2.2.3.1) for interprocess-communication via method calls (possibly remote). Figure 3.8 demonstrates the creation and the usage of OpMethods. The **OpMethod** decorator injects the *IM_OP* attribute into *p1*. At creation

```
class Foo(PSObject):
  @OpMethod
  def multNSum(self, inv):
    x = inv.getParameter(0)
    y = inv.getParameter(1)
    return x*y, x+y

def main():
  f = Foo()
  product, sum_ =  f.multNSum.call(3, 2)
  print product
  print sum_
```

Figure 3.7: Returning multiple values from an Operation

time, the PSObject replaces *p1* with an OpProxy object, which exposes the Operation interface to the user. With this approach, we are able to keep *p1* as an instance method of **Maine** and give the OpMethod access to all instance variables through *self*. PySy also provides the decorator **StaticOpMethod**, which provides an OpMethod for a class and not a specific instance.

```
class Maine(PSObject):
  def __init__(self):
    self.y = self.p1.call(4)
    print "p1 returned: %d" % self.y


  @OpMethod
  def p1(self, inv):
    x = inv.getParameter(0)
    print "p1 got: %d" % x
    return x+1

def main():
  m = Maine()
  m.p1.send(m.y+1)

  #expected output:
  #p1 got: 4
  #p1 returned: 5
  #p1 got: 6
```

Figure 3.8: Example showing the usage of OpMethods in PySy.

3.5.2 Processes

PySy processes are modeled after JR processes (Section 2.2.3). In the implementation of JR and PySy, processes are a special kind of OpMethod. PySy processes, however, cannot be static. Figure 3.9 and Figure 3.10 show how to define processes in JR and PySy, respectively.

```
public class Processes{
  public static void main(String[] args){new Test(); new Test();}
}
class Test{
  public Test(){}
  public process p1{
    System.out.println("printed once per Test instantiation");
  }
  public static process p2{
    System.out.println("printed once per obj");
  }
}
```

Figure 3.9: Creating processes in JR.

```
class Foo(PSObject):
  def __init__(self):
    self.startProcesses()
  @Process
  def x(self, inv):
    print "in Foo process"
def main():
  f = Foo()
```

Figure 3.10: Creating processes in PySy.

As seen in Figure 3.10, a PySy process is created by using the **Process** decorator. This decorator adds a new attribute, *IM_PROCESS*, to the function x. The creation of a process is similar to the description given above for OpMethods. However, at some point after the instantiation of the object, the user must explicitly call the object's *startProcesses* method to start the declared processes in the object. This is unfortunate, but necessary

because Python does not allow the user to run arbitrary code after a class has been fully created. This is also why PySy cannot have static processes. See Section 4.2.2 for further explanation.

3.5.3 InniOps

Recall from Section 2.2.3.2, InniOps are a shared queue implementation used for inter-process communication. Figure 3.11 demonstrates the creation and the usage of an InniOp.

```
class Maine(PSObject):
  def __init__(self):
    self.foo = InniOp.create()
    self.p1.send()

  @OpMethod
  def p1(self, inv):
    i = 0
    print "+loop"
    while True:
      if i == 4:
        break

      x = self.foo.receive()
      print x
      i += 1
    print "-loop"

def main():
  m = Maine()
  m.foo.send(4)
  m.foo.send(6)
  m.foo.send(12)
  m.foo.call(1)

  #expected output:
  #+loop
  #4
  #6
  #12
  #1
  #-loop
```

Figure 3.11: Example showing the usage of an InniOp in PySy.

3.5.4 Semaphores

Section 2.2.3.3 discusses JR's implementation of the semaphore abstraction. Figure 3.12 shows how to create and use a semaphore in PySy.

PySy's semaphore abstraction also interfaces with the Python **with** statement, which is normally used with Python's locks to execute a critical section of code. Entering and exiting a **with** block executes pre-processing and post-processing code. The user provides the pre-processing and post-processing code by overloading the *__enter__* and *__exit__* methods, respectively. The **with** statement has an optional **as** command that binds the **with** object to a separate identifier and deletes the identifier from the current scope upon leaving the **with** block. A common Python locking idiom is seen in Figure 3.13. Entering the **with** acquires the lock and exiting the **with** releases the lock. The benefit of using the lock in the with statement is that *__exit__* is *guaranteed* to run regardless of a thrown exception inside the **with** code block. This idiom prevents a very common mistake by novice concurrent programmers: surrounding a critical section with the acquiring and releasing of a lock without enclosing the release in a **try...finally** block. When an exception is thrown without the **try...finally** block, the lock may still be acquired after leaving the critical section. PySy stays in the spirit of Python and provides this functionality with the semaphore abstraction. Entering the **with** block for a semaphore is the equivalent to a P operation and exiting the **with** block is equivalent to a V operation. Figure 3.14 shows a semaphore used in a **with** statement and shows the **with** statement's optional **as** command.

3.6 Inni

Recall from Section 2.2.4, the Inni statement has the form:

```
inni op_command1  [] op_command2 [] ...
```

We will refer to each *op_command* as an InniArm. PySy models an Inni statement as a list of InniArms with an optional else arm or else after arm. The else arm is executed

```
niters = 10000
class Foo(PSObject):
  x = 0
  def __init__(self):
    self.mutex = InniOp.createSem(1)
    self.done = InniOp.create()
    self.startProcesses()
  @Process
  def p1(self,inv):
    for i in xrange(niters):
      self.mutex.P()
      Foo.x += 1
      self.mutex.V()
    self.done.send()
  @Process
  def p2(self,inv):
    for i in xrange(niters):
      self.mutex.P()
      Foo.x -= 1
      self.mutex.V()
    self.done.send()

def main():
  f = Foo()
  for i in xrange(2):
    f.done.receive()
  print Foo.x


#expected output:
#0
```

Figure 3.12: How to create and use a semaphore in PySy.

when there are no serviceable invocations. The else after arm is executed when there are no invocations after waiting for a user-specified amount of time (similar to a timeout). The Inni object only provides one user method – *service*. *service* attempts to service an invocation for one of the arms. PySy uses the same invocation selection algorithm as JR (described in Section 2.2.5). *service* returns a control status value that reflects the occurrence of a **break** or a **continue** statement inside of the Inni arm. We discuss this topic more in Section 3.10.

```
import threading
lock = threading.Lock()
sharedX = 0
niters = 10
def t1():
  global sharedX, niters, lock
  for i in xrange(niters):
    with lock:
      sharedX += 1


tOne = threading.Thread(target=t1)
tTwo = threading.Thread(target=t1)
tOne.start()
tTwo.start()
tOne.join()
tTwo.join()
print sharedX
```

Figure 3.13: Example usage of the Python with statement.

```
class Foo(PSObject):
  def __init__(self):
    self.sem1 = InniOp.createSem(1)
    self.done = InniOp.create()
    self.x = 0
    self.p1.send()
    self.p1.send()

  @OpMethod
  def p1(self, inv):
    for i in xrange(10):
      with self.sem1 as sem:
        sem.V()   #unnecessary, but used to show the
            #usage of sem after the 'as' command
        sem.P()
        self.x += 1
    self.done.send()

def main():
  f = Foo()
  f.done.receive();f.done.receive()
  print f.x
```

Figure 3.14: Example usage of the semaphore abstract used in the **with** statement.

3.6.1 InniArm

In JR, the InniArm has the form:

```
r_type op_name(<param_list>)[st st_expr][by by_expr]code_block
```

PySy only requires the InniArm constructor to take one parameter for the InniOp. The code block, such that clause, and by expression are all optional parameters. By default, the InniArm is given a function with an empty code block. Figure 3.15 and Figure 3.16 show an equivalent JR program with an Inni statement written in PySy.

```
public class Inni2Arms
{
  public static op void f(int);
  public static op void g(int);
  public static void main(String[] args)
  {
    send f(2);send f(3);send g(1);
    call g(6);
  }
  public static process P1{
    while(true){
      inni void f(int x){
        System.out.println(x);}
      [] void g(int x){
        System.out.println(x);}
    }
  }
  /*
  expected output:
  2
  3
  1
  6
  */
}
```

Figure 3.15: JR Inni statement with two arms.

3.6.2 InniElse

An else clause inside of an Inni statement behaves exactly like the else clause in a list of conditional statements. When an Inni statement is servicing an invocation, the else

```
class Foo(PSObject):
  def __init__(self):
    self.f = InniOp.create()
    self.g = InniOp.create()
    self.startProcesses()

  @Process
  def p1(self, inv):
    #the code block for each arm
    @ArmCode
    def f_ArmCode(inv):
        print inv.getParameter(0)
    @ArmCode
    def g_ArmCode(inv):
        print inv.getParameter(0)

    f_arm = InniArm(self.f, f_ArmCode)
    g_arm = InniArm(self.g, g_ArmCode)
    inni = Inni(f_arm, g_arm)
    while True:
      inni.service()
def main():
  foo = Foo()
  foo.f.send(2)
  foo.f.send(3)
  foo.g.send(1)
  foo.g.call(6)
```

Figure 3.16: PySy Inni statement with two arms.

branch is executed if there are no serviceable invocations. Figure 3.17 and Figure 3.18 show an equivalent JR and PySy program with an Inni statement and an Inni else arm.

3.6.3 ElseAfter

The else after clause acts as a timeout for an Inni statement. Figure 3.19 and Figure 3.20 show an equivalent JR and PySy program with an Inni statement an an else after arm.

3.6.4 SuchThat/Synchronization Expression

Figure 3.21 and Figure 3.22 show an equivalent JR and PySy program with an Inni statement and a synchronization expression.

```
public class InniElse{
  public static op void f(int);
  public static void main(String[] args){
    send f(3); send f(6);
  }
  public static process P1{
    while(true){
      inni void f(int x){
        System.out.println(x);}
      [] else{
        System.out.println("else"); break;}
    }
  }
}
```

Figure 3.17: JR Inni statement with else arm.

```
def main():
  #create the operation
  fOp = InniOp.create()
  fOp.send(3)
  fOp.send(6)

  #the code blocks for each arm
  @ArmCode
  def f_ArmCode(inv):
    print inv.getParameter(0)
  @ArmCode
  def else_ArmCode(inv, control):
    print "else"
    control.status = Control.BREAK

  f_arm = InniArm(fOp, f_ArmCode)
  else_arm = InniElse(else_ArmCode)
  inni = Inni(f_arm, inni_else=else_arm)
  while True:
    controlStatus = inni.service()
    if controlStatus == Control.BREAK:
      break
```

Figure 3.18: Equivalent PySy Inni statement with else arm.

3.6.5 By/Scheduling Expression

Figure 3.23 and Figure 3.24 show an equivalent JR and PySy program with an Inni statement and a scheduling expression.

```
public class InniElseAfter{
  public static op void f(int);
  public static void main(String[] args){
    send f(3); send f(6);
  }
  public static process P1{
    while(true){
      inni void f(int x){
        System.out.println(x);}
      [] elseafter(1000){//wait 1 second
        System.out.println("else after"); break;}
    }
  }
}
```

Figure 3.19: JR Inni statement with else after arm.

3.7 Reply

Recall from Section 2.2.6 that the reply statement is a mechanism for the servicer to communicate the return value to a synchronous invoker without terminating the servicing thread. Figure 3.25 shows a program that opens a file, reads the files contents into a string, and uses the reply mechanism to return the string to the invoker. The file handle is not appropriately closed until after the reply occurs and sends the return value back to the caller. PySy preserves the JR semantics for multiple usages of reply.

3.8 Forward

Recall from Section 2.2.7 that the forward statement passes the responsibility of servicing to another operation. Figure 3.26 shows an example of PySy's forward mechanism. Also, PySy preserves the JR semantics for multiple usages of a forward.

3.9 Concurrent Invocation Statement

Section 2.2.10 described the JR Co statement. PySy provides the user a similar construct, but without quantifiers. The Co statement, like the Inni, is broken into a list of

```
def main():
  #create the operation
  fOp = InniOp.create()
  fOp.send(3)
  fOp.send(6)

  #the code blocks for each arm
  @ArmCode
  def f_ArmCode(inv):
    print inv.getParameter(0)
  @ArmCode
  def else_ArmCode(inv, control):
    print "else after"
    control.status = Control.BREAK

  f_arm = InniArm(fOp, f_ArmCode)
  else_arm = InniArmElseAfter(1.0, else_ArmCode)
  inni = Inni(f_arm, else_after=else_arm)
  while True:
    controlStatus = inni.service()
    if controlStatus == Control.BREAK:
      break
```

Figure 3.20: Equivalent PySy Inni statement with else after arm.

```
public class InniST{
  public static op void f(int);
  public static void main(String[] args){send f(2); send f(-1);}
  public static process P1{
    while(true){
      inni void f(int x) st x > 0{
        System.out.println(x);}
    }
  }
}
```

Figure 3.21: JR Inni statement with such that clause.

arms called CoArms. Each CoArm contains an operation, an invocation, and the type of Co invocation (CoCall or CoSend). The type of invocation defaults to a CoCall. Subsequently, the CoArms are passed to the Co constructor. The Co constructor may be instantiated with two different sets of parameters. The first way passes the constructor any number of CoArms (similar to the Inni). The second way passes the constructor a list of CoArms. After

```
def main():
  #create the operation
  fOp = InniOp.create()
  fOp.send(2)
  fOp.send(-1)

  #the code blocks for each arm
  @ArmCode
  def f_ArmCode(inv):
    print inv.getParameter(0)
  @SuchThat
  def f_greaterThanZero(inv):
    return inv.getParameter(0) > 0
  f_arm = InniArm(fOp, f_ArmCode, st=f_greaterThanZero)
  inni = Inni(f_arm)
  while True:
    inni.service()
```

Figure 3.22: Equivalent PySy Inni statement with such that clause.

```
public class InniBy{
  public static op void start();
  public static op void f(int);
  public static void main(String[] args){
    send f(6); send f(12); send f(8);
    send start();
  }
  public static process P1{
    receive start();
    while(true){
      inni void f(int x) by -x{
        System.out.println(x);}
      [] else{break;}
    }
  }
  /*expected output:
  12
  8
  6
  */
}
```

Figure 3.23: JR Inni statement with by expression.

instantiation, the user invokes the Co's *go* method to perform the computation. Figure 3.27 shows how to use PySy's Co statement to perform matrix multiplication.

```
def main():
  fOp = InniOp.create()
  fOp.send(6)
  fOp.send(12)
  fOp.send(8)
  #write the code blocks for each arm
  @ArmCode
  def f_ArmCode(inv):
    print inv.getParameter(0)
  @By
  def f_byDecreasing(inv):
    return -inv.getParameter(0)
  f_arm = InniArm(fOp, f_ArmCode, by=f_byDecreasing)
  inni = Inni(f_arm)
  while True:
    inni.service()
```

Figure 3.24: Equivalent PySy Inni statement with by clause.

```
class Foo(PSObject):
  @OpMethod
  def readFile(self, inv):
    filename = inv.getParameter(0)
    f = open(filename, 'r')
    result = ''.join([lines for lines in f.readlines()])
    inv.reply(result)
    f.close()

def main(inFilename):
  try:
    f = Foo()
    contentsAsStr =  f.readFile.call(inFilename)
    print contentsAsStr
  except Exception as e:
    PySy.traceback(e)
```

Figure 3.25: Example showing the usage of the reply mechanism inside of an OpMethod.

3.10 Control Flow

In this section, we discuss how PySy preserves the semantics of the JR language for executing **break**, **continue**, or **return** statements inside of an ArmCode. In JR, the behavior of **break** and **continue** is consistent with the expected Java behavior. However, a

```
class Tester(PSObject):
    @OpMethod
    def p1(self, inv):
        inv.forward(self.p2)

    @OpMethod
    def p2(self, inv):
        return inv.getParameter(0) * 2


def main():
    try:
        f = Tester()
        retVal = f.p1(2)
        print retVal
    except Exception as e:
        PySy.traceback(e)
    #expected output:
    #4
```

Figure 3.26: Example showing the usage of the forward mechanism inside of an OpMethod.

return statement inside of an Inni arm returns a value to the calling invocation and breaks out of the Inni statement and continues execution with the next statement after the Inni. The control flow semantics are highlighted in Figure 3.28.

Recall from 3.6, ArmCode is implemented as a method. In Python, it is a syntax error to use **break** and **continue** statements outside of a loop. Because of this, the user must propagate control flow manually to the caller. This is achieved by using the ArmCode method's optional formal parameter *control* to store any desired control flow changes and pass the changes back to the Inni statement. After invoking the Inni's *service* method, it is the user's responsibility to use the returned control status to provide the desired control flow, as shown in Figure 3.29.

```
r = random.Random()

class MatrixMult(PSObject):
  MAX_COEF = 50
  N = 25
  def __init__(self):
    self.solution = 0
    self.A = self.genMatrix(MatrixMult.N)
    self.B = self.genMatrix(MatrixMult.N)
    self.C = self.genMatrix(MatrixMult.N, isZero=True)
    arms = []
    for r in xrange(MatrixMult.N):
      for c in xrange(MatrixMult.N):
        inv = Invocation(r,c)
        arms.append(CoArm(self.compute, inv, coKind=CoArm.CoKind.COCALL))

    co1 = Co(arms)
    co1.go()
    #print self.C


  @OpMethod
  def compute(self, inv):
    r,c = inv.getParameters()
    for k in xrange(MatrixMult.N):
      self.C[r][c] += self.A[r][k] * self.B[k][c]

  def genMatrix(self, n, isZero=False):
    result = []
    for i in xrange(n):
      result.append([])
      for j in xrange(n):
        val = 0 if isZero else r.randint(0, MatrixMult.MAX_COEF)
        result[i].append(val)
    return result

def main():
    MatrixMult()
```

Figure 3.27: Example showing the usage of the co statement to perform matrix multiplication.

```
public op void f(int);
public process P1{
  while(true){
    //continue statement inside inni continues here
    inni void f(int x){
      System.out.println(x);
      ... //code that may contain a break, continue,
          //or return statement
    }
    //return statement inside inni continues here
  }
  //break statement inside inni continues here
}
```

Figure 3.28: A simple Inni statement with a single arm that does a **break**, **continue**, or **return**.

```
def main():
  f = InniOp.create()
  f.send(0)
  f.send(6)
  @ArmCode
  def f_ArmCode(inv, control):
    if inv.getParameter(0) > 0:
      control.status = Control.BREAK
    else:
      control.status = Control.CONTINUE
  f_arm = InniArm(f, f_ArmCode)
  inni = Inni(f_arm)
  while True:
    cfStatus = inni.service()
    if cfStatus == Control.BREAK:
      print "breaking"
      break
    elif cfStatus == Control.CONTINUE:
      print "continuing"
      continue
```

Figure 3.29: Example showing how to use the control parameter returned from Inni to enforce the desired control flow behavior changes.

Chapter 4

Implementation

This chapter presents the inner workings of PySy. Section 4.1 presents Pyro and discusses its role in providing network communication for PySy VMs and RMI-like capabilities for remote objects. Section 4.2 introduces the implementation of PySy remote objects. Section 4.3 discusses PySy's locking mechanisms for local and remote objects. Section 4.4 demonstrates how PySy provides Process-safe output. Section 4.5 discusses PySy's quiescence detection algorithm. Section 4.6 presents the challenges with providing PySy users similar scoping rules as Python, while still providing a familiar user experience.

4.1 Pyro

Pyro [25], Python Remote Objects (version 3.x), is a Python package that provides the necessary functionality to share objects between processes and across different machines. To run Pyro, the user must have a Python interpreter (version 2.5-2.7) and a system that supports TCP/IP networking.

PySy uses Pyro to provide RMI-like capabilities for all remote objects (see Section 4.2.1). Each Pyro program that uses remote objects must have at least one Pyro daemon running. The Pyro daemon is a server-side Pyro object that accepts and dispatches RMI calls to registered objects. The user registers an object with the daemon and is given

back a Pyro Unique Resource Identifier (URI). The URI encodes the IP address, port number, and object name of the daemon and has the form PYRO://hostname:port/objName. The URI is used to create proxies for remote resources. The Pyro proxy is the interface between the user and the daemon. Each Pyro proxy represents a separate network connection to a Pyro daemon. Proxies are not reclaimed unless the user invokes the *_release()* method or specifies a timeout. When invoking a method through a proxy, a message is sent to the daemon containing the object's URI, the method name, and the arguments for the remote method call. The daemon forwards the call to the appropriate object, executes the method, and returns the result back to the caller. The RMI is completely transparent to the user. In the case of an unhandled exception occurring during an RMI call, Pyro will propagate the error back to the client. During a PySy program, if the user does not handle an exception, it is possible for zombie processes to remain.

A typical Pyro program follows the client-server model. The server creates and publishes remote objects and the client obtains proxies to the remote objects and uses them accordingly. The client is required to know the URI at run-time. This means that the URI is hardcoded into the program or the server communicates the URI to the client. It is obvious that hardcoding names for objects does not scale and would require changes if the hostname or port number of the daemon changed. To make this process easier, Pyro provides a nameserver that maps a user-defined name to a Pyro URI. The nameserver is a special Pyro object that acts as a central location for storing how to acquire a proxy for each registered Pyro object. The user registers an object with a name (*PYRONAME://:Test.MyObject*) or location (*PYROLOC://hostname:port/objectname*) and this name is used by the nameserver as a key in an internal dictionary to locate and communicate with the daemon responsible for handling the RMI requests.

Pyro provides PySy with an easy to use interface to handle the network communication for the distributed computation. Our overall experience is a positive one. Over the course of this research, we have found several bugs in the Pyro implementation and the author has always been available and quick to find fixes.

4.2 PySy

4.2.1 Remote Objects

To provide support for distributed computing, a package or language must define precise semantics for how each component interacts and communicates with other components. PySy's implementation uses three kinds of remote objects: Virtual Machine Manager(VMM), Virtual Machine (VM), and a remote receiver. This section discusses the role of each of these remote objects.

The Virtual Machine Manager (VMM) is created when the user calls PySy.init(). The VMM is synonymous to JR's JRX. The VMM has several responsibilities. The most obvious is the managing of PySy VMs (see Section 3.2). Its other responsibilities include the management of remote locks, monitoring the health of all VMs, and being the central node for program-level quiescence.

After the VMM is created, it automatically creates the main VM. Recall from Section 3.2, the main VM is the default location for PySy object creation and executes the code provided by the user in *main()*. As the program executes, if there are requests to create additional VMs, then the VMM processes these requests, starts up the new VM processes via ssh (even for localhost VMs), and returns a reference to the new VM.

Another role for the VMM is managing remote locks. Initially, the invocation list lock for an InniOp is local. However, as InniOps are used in conjunction with other InniOps in an Inni statement their locks may change from being a local lock to a remote lock (see Section 4.3). If this happens, there must be a way for other VMs to retrieve the remote locks. Since VMs are oblivious to the existence of other VMs, the VMM is an obvious choice to manage the remote locks.

If a VM crashes unexpectedly, it may be difficult or even impossible for a program to recover. The VMM keeps a watchful eye on all VMs and will attempt to gracefully shutdown the program in the case a VM unexpectedly crashes. The VMM periodically pings each of the VMs (every five seconds) to ensure the VM is still running. If a VM

misses three pings, then the VMM will gracefully exit the program.

Terminating concurrent programs is slightly more involved than sequential programs. Generally, in concurrent programs, the user must explicitly program logic to join all program segments before termination. The VMM acts as the central node for providing automatic program termination and deadlock detection. This is discussed further in Section 2.2.9.

Each VM and the VMM has a remote receiver object. The receiver acts as a server for dispatching RMI calls to the correct object. When a user creates a remote object using `foo = PySy.createInstance("foo.Foo")`, the returned object contains a reference to each operation enclosed in *foo*. All subsequent invocations of *foo*'s operations must somehow be forwarded to the correct operation. Before the remote reference is returned to the caller, it is added into a dictionary in the VM's receiver, which maps an operation's unique identifier to an operation. When a remote reference's operation is invoked, an RMI call is made to the receiver. The RMI call contains the UID of the operation that the user wants to invoke, which allows the receiver to correctly forward the invocation. The receiver asynchronously processes each RMI request to avoid bottlenecking.

4.2.2 PySy Objects

Python's object instantiation behaves differently from its counterparts in languages like Java or C++. When the constructor is called for a Python object, a two part process begins. First, the parent object's *__new__* method is invoked, which, by default, details how the object is created and returns a memory reference for the new object. Next, the object's *__init__* method is called to initialize its instance variables.

Consider the Python class and its instantiation in Figure 4.1. *opm1()* is intended to be an OpMethod. After *o* is instantiated, *opm1()* may be invoked by `o.opm1()` (same as when calling normal Python methods). However, this is not the interface that we want to provide the user. As discussed in Section 2.2.3, OpMethods may be invoked synchronously and asynchronously via call and send, respectively. Under the default behavior for class

creation, the interface we want to expose to the user is not possible. That is, we would like to be able to invoke call and send this way: *o.opm1.send()* or *o.opm1.call()*. To do this, we need to do modify how objects, which contain OpMethods (and its variants, e.g., Process) are created.

```
class Obj(object):
  def __init__(self, val):
    self.val = val
  def opm1(self):
    print self.val
o = Obj(6)
```

Figure 4.1: Simple Python class declaration and instantiation of an object.

Python allows its users to modify the creation of a user-defined object by overriding the object's *__new__* method. In PySy's case, we wanted to hide the custom creation implementation details from our users, so we created our own base class, **PSObject**. Figure 4.2 shows how to create a PySy object. The two main differences from Figure 4.1 are Foo inheriting from **PSObject** and the OpMethod decorator. The OpMethod decorator adds a new attribute *IM_OP* to *mult*. During the instantiation process of the PSObject in *__new__*, we iterate over all methods of the class and check for the existence of the *IM_OP* or the *IM_PROCESS* attribute. If these attributes exists, then we replace the original method with an OpProxy object. All invocations of *mult* are now sent to the proxy first. Thus, exposing the proper interface to the user. Since the OpProxy class derives from the Operation base class (see Appendix C) *mult* is now invoked using `mult.send()` or `mult.call()`.

Section 3.5.2 mentions that PySy cannot provide static processes and the user must invoke the PSObject method *startProcesses* to begin each process. Before we discuss this, let us discuss how Python creates new-style objects. The first step involves calling the static method *__new__* to create the object. The return value of *__new__* is an instance of the class and is passed on to *__init__* to initialize its attributes. The *im_class* and *im_self* attributes do not have a reference in memory until *__new__* returns. So, if we were to attempt to start an object's processes in *__new__*, then we would have a race condition. Unfortunately, Python

does not provide any procedures for adding post-processing code after object creation, so we were forced to have the user invoke the *startProcesses* method.

```
class Foo(PSObject):
  @OpMethod
  def mult(self, inv):
    x = inv.getParameter(0)
    y = inv.getParameter(1)
    return x*y

def main():
  f = Foo()
  print f.mult.call(2,4)
```

Figure 4.2: PySy Object with OpMethod as an instance method.

4.2.3 OpMethods

PySy uses the following decorators to add attributes at run-time to a **PSObject**: OpMethod, StaticOpMethod, and Process. Each decorator adds a specific attribute to the function that allows the PSObject to replace the method with the appropriate OpProxy.

4.3 Locking and Equivalence Classes

The utmost concern for concurrent programs is data safety. An increase in performance means nothing if we cannot guarantee the intended behavior. When users are running distributed programs over heterogeneous systems or simply just running a multi-threaded program on a single system, there must be a way of ensuring changes to shared data are performed atomically. This section discusses the concept of equivalence classes in regards to the locking mechanisms of InniOps.

The InniOp (refer to Section 2.2.3.2) represents potentially shared data between multiple processes. It is possible, maybe even likely, that an InniOp is being shared between multiple threads on different machines and being accessed by multiple input statements. It

is imperative to ensure the integrity of the contents of an InniOp's invocation list. We use the concept of equivalence classes to accomplish this task. The equivalence class of an operation is the set of operations that share the same lock. Initially, each operation has its own lock and is, by default, in its own equivalence class. As operations are used together inside of an Inni statement, their equivalence classes are merged. After the merge, all of the operations belong to the same equivalence class. This section discusses some methodologies for computing equivalence classes. We first begin with JR's predecessor, SR and discuss SR's static equivalence class implementation. Finally, we will discuss the JR's dynamic solution (also implemented in PySy).

The SR language computes the equivalence classes[3] of its operations statically. At compile time, the compiler computes which operations appear together in input statements. The set of operations that appear together in any input statement is called the operation's equivalence class. An operation is in the singleton equivalence class if and only if it appears in single arm input statements throughout the entire program.

The algorithm for computing equivalence classes changed in SR version 2.2 (and later versions). The equivalence classes are still computed statically, but SR now allowed for dynamic operations and the servicing of capabilities (reference to an operation) inside of an input statement. A dynamic operation is defined as an operation created at run-time via the new expression. Throughout the lifetime of a program, a capability may refer to different operations. These additions warrant the algorithm change and require the following restrictions:

- The input statement has one arm, no synchronization expression, and no scheduling expression.
- The capability references an operation in the current VM.

JR allows for a more flexible environment than the previous SR implementations. JR completely eliminates the restrictions imposed by SR version 2.2+. However, the equivalence class computation process (referred to as merging equivalence classes or, simply,

merging) is much more complex, and must be computed dynamically. Each InniOp initially belongs to its own equivalence class. When multiple InniOps appear in the same input statement, their equivalence classes are merged. The resulting equivalence class is the union of all the InniOps of the input statement. The input statement must determine if a merge is necessary on each service attempt because the equivalence class of one of the operations may have been changed by another input statement since the previous service attempt. In the general case, however, JR can optimize this computation considerably by caching the equivalence class representation on each service attempt. On the next service attempt, JR checks to see if the current equivalence class differs from the previous one. If not, the equivalence classes do not need to be merged. Otherwise, JR must recompute the equivalence classes. PySy adopts JR's dynamic equivalence computation algorithm. The rest of this section will discuss this algorithm in-depth.

Recall (Section 3.5.3) the role of InniOps, a shared queue implementation, in interprocess and remote communication. Figure 4.3 shows a program with multiple processes on the same machine, which share an operation, and service invocations via an Inni statement. To ensure correct behavior, only one of the processes should be able to access f's invocation list at any given time. This exclusion is accomplished by protecting the InniOp's invocation list with a lock. Initially, each InniOp has its own individual lock. Before either Inni statement in Figure 4.3 can attempt to service any pending invocations, it must first grab f's lock. If pending invocations exist, then the first serviceable invocation is removed from the invocation list (see Section 2.2.5 for more on the invocation selection process), the lock is released, and the invocation is serviced by the code block associated with f's arm.

Now, consider the JR program with two processes, two shared operations, and two Inni statements (one per process) in Figure 4.4. Figure 4.5 depicts a potential execution trace that results in deadlock for the program in Figure 4.4.

Figure 4.5 illustrates the inherent problem with acquiring multiple locks. However, we will discuss a few ways to solve this problem. One solution is to ensure that locks are acquired in a predictable, deterministic fashion by providing a strict ordering of locks. We

```
public class Inni1ArmRival{
  public static void main(String[] args){}
  public static op void f(int);
  public static process P1{
    send f(1);
    while(true){
      inni void f(int x){
        System.out.println("P1 serviced: " + x);
        send f(1);
      }
    }
  }
  public static process P2{
    send f(2);
    while(true){
      inni void f(int x){
        System.out.println("P2 serviced: " + x);
        send f(2);
      }
    }
  }
}
```

Figure 4.3: Two separate processes with Inni statements sharing an operation.

could assign each lock a unique ID (UID) and enforce that the associated locks for each Inni arm's operation are acquired according to this strict ordering. For example, f and g in Figure 4.4 may be given UIDs of 0 and 1, respectively. Assume we have added logic to the Inni to sort the Inni arms by their operation's UID of the operation. Now, f's lock will always be acquired before g's lock. The problematic situation described in Figure 4.5 will no longer happen and, consequently, this program will no longer deadlock. While this solution works, it does seem to be a bit inefficient. How much would the performance of the program suffer if both f and g were remote operations instead of local? In that case, we would have to invoke one remote method to acquire the lock for each distinct operation in the Inni statement. A good rule to achieve maximum performance in distributed programming is to minimize the number of messages sent over the network. Acquiring each lock separately in a strict order sends out n messages for each invocation serviced, where n is the number of distinct operations in the Inni arms.

It turns out there is a way to significantly decrease the number of messages sent

```
public class Inni2ArmRival{
  public static op void f(int);
  public static op void g(int);
  public static void main(String[] args){}
  public static process P1{
    send f(1);
    while(true){
      inni void f(int x){
        System.out.println("P1 serviced: " + x);
        send g(1);
      }
      [] void g(int x){
        System.out.println("P1 serviced: " + x);
        send f(1);
      }
    }

  }
  public static process P2{
    send f(2);
    while(true){
      inni void g(int x){
        System.out.println("P2 serviced: " + x);
        send f(2);
      }
      [] void f(int x){
        System.out.println("P2 serviced: " + x);
        send g(2);
      }
    }
  }
}
```

Figure 4.4: Two separate processes with Inni statements sharing two operations.

- *P1* executes until it gets to its Inni statement and grabs f's lock.
- A context switch occurs, and *P2* is now allowed to run.
- *P2* grabs g's lock and attempts to acquire f's lock, but f's lock has been acquired by *P1*, so *P2* waits until f's lock is released.
- *P1* wakes up and tries to get g's lock, but g's lock is also unavailable.
- *P1* is waiting for *P2* and vice versa. Deadlock.

Figure 4.5: Execution trace showing deadlock in a JR program.

over the network for remote operations. JR and PySy achieve this by utilizing an approach that uses equivalence classes [16]. As mentioned above, the equivalence class of an operation is the set of operations that share the same lock. Figure 4.6 illustrates many of the details involved with merging equivalence classes. Again, initially all InniOps have their own individual locks, and, thus, are in their own equivalence class. We define a remote operation as an operation that is not local to the current VM. Also, a remote lock is a lock associated with at least one remote operation. Consider the program in Figure 4.6 after all the processes have started, *P1* is executing, and reaches its Inni statement. f is the only arm in this Inni statement, so we do not need to merge equivalence classes. *P1* grabs f's lock and services an invocation. Next, *P2* is executing and reaches its Inni statement. *P2* notices that it has two arms that have different equivalence classes and must perform a merge. The equivalence class for f and g are merged and now belong to a new equivalence class denoted {f,g}. The merging process does not create a new local lock for f and g. The algorithm chooses the equivalence class with the largest cardinality. In this case, there is a tie. The merge algorithm breaks the tie by choosing the first equivalence class it checked (g). So, g's lock is chosen and f's lock is forwarded to g's lock.

Continuing with the example, *P3* is executing and it arrives at its Inni statement. It has two arms, one with a remote operation (f is remote since it is local to the main vm) and one local operation, h. f has an equivalence class of {f,g} and h is in its own equivalence class. We merge the equivalence class for f and h to yield {f,g,h}. In the case, where at least one of the operations is remote, but all locks are local we need to create a new remote lock and share it with all elements of the new equivalence class. The implementation uses the VMManager to control the management of all remote locks. The VMManager serializes multiple merge requests to ensure that only one merge operation is occuring globally. At this point, the VMManager is contacted by the merging operation to create a new remote lock and f, g, and h will have their locks forwarded to this new remote lock.

4.4 Thread/Process-safe Output

Concurrent programming adds additional challenges because of its non-deterministic behavior. Reliable output is important for development, regression testing, and usability. Obviously, we cannot guarantee the ordering of output in a concurrent program, but it is important, if possible, to provide mechanisms that will ensure thread-safe output. The Python interpreter, however, interfaces with the C write function. *write()* does not provide thread-safe output or process-safe output and relies on the underlying operating system to handle the file I/O. Internally, `write()` [1] outputs n bytes of a buffer to the current offset of a file descriptor. This is where a race condition occurs. A simple Python print statement executes two buffer writes. The first is the string to be printed and the second is a newline character. Consider the Python program in Figure 4.7, below illustrates a possible execution trace that will result in interleaved output:

1. *t1* writes to the buffer (context switch)
2. *t2* writes to the buffer (context switch)
3. *t1* prints its newline character (context switch)
4. *t2* writes its newline character to the buffer.

To provide thread-safe printing, we must force the trailing newline character to be written to the write buffer directly following the specified string. PySy solves the thread-safe problem by creating a wrapper for the file stream for each process that uses a recursive lock to protect the data buffer. Thus, ensuring that only one thread per process writes to the buffer. The solution, however, risks deadlock. Figure 4.8 shows a sample program where the deadlock will occur. Figure 4.9 gives an execution trace for the deadlock. Practically, we do not believe this complication to be too troublesome or restrictive for the user. If the user is cognizant of this restriction, then they should be able to avoid it entirely.

The above thread-safe output solution only describes how to ensure safety within a single process, but not necessarily within multiple processes. Consider a multiprocess

PySy program where each process has its own file descriptor to the same standard output file and each process writes to stdout using the OS write method. On Linux, `write()`, by default, is not process-safe. First, `write()` writes n bytes to stdout beginning at the current file offset, and,if successful, increments the current file position. However, these two steps are not atomic. It is possible for another process to corrupt the output file by writing to the same current file offset multiple times before the offset is updated. This is remedied by opening the standard output file with the append only flag, which forces the current file position to be incremented before any data is written, and making PySy programs provide process-safe output.

4.5 Quiescence

PySy provides quiescence detection [14] [15] to its users. PySy uses the same quiescence detection algorithm as JR. Each VM keeps a local count of the number of active threads and the number of messages sent. Recall that we are defining an active thread as a thread that is not waiting on a PySy resource. When no active threads are running inside of a VM, it communicates its idle status to the VMManager. Next, the VMManager will query the status of each of the VMs. If all VMs are idle and the total number of messages sent is equivalent to the number of messages received, then program is quiescent. If not, the program continues its execution.

4.6 Scoping

Chapter 3 described how most JR features are modeled nicely as objects in PySy. Intuitively, it makes a lot of sense to use a design that follows basic object oriented principles. Unfortunately, it has the negative side effect of making data sharing difficult between PySy object constructs and the outer scopes. For a review of Python scoping rules and lexical closure behavior, see Section 2.1.4.

Consider the program in Figure 4.10. The user intends to share variables from

main's scope with the class f_ArmCode, but, alas, cannot. Instead, with the armCode implemented as a class, the only way to share data between *main* and the armCode is to pass the necessary shared data to the armCode's constructor (see Figure 4.11). Additionally, the user must also wrap any shared primitives with a wrapper class for the modified value to be reflected in the outer scopes (see Figure 4.11). This solution has significant drawbacks in terms of usability and scalability and would make the modeling of complex programs with many shared variables extremely unpleasant.

Initially, PySy implemented ArmCodes as discussed above. However, the lack of usability forced us to look for a better solution. Currently, PySy uses lexical closures to allow a more user friendly environment. Closures provide a partial solution to the scoping problem introduced by using classes to represent abstractions like ArmCode. Unfortunately, however, closures do not allow write access to the closed over variables. The user may circumvent the no write restriction by wrapping the primitive and modifying the wrapped value via the wrapper's interface, as in Figure 4.12. PySy also provides a synchronized version of the primitive wrapper. The primitive wrappers will no longer be necessary when PySy is ported to Python 3.x. Python 3.x provides a new keyword, **nonlocal**, that allows the user to specify the names of variables declared in an enclosing scope to be used as if they were local. Any subsequent changes to a nonlocal variable will be reflected outside of the local scope.

```
public class InniMergeEC{
  public static void main(String[] args)
  {
    LocalClient lc = new LocalClient();
    vm rvm = new vm() on "pc12";
    remote RemoteClient rc = new remote RemoteClient(lc.f) on rvm;
  }
}
class LocalClient{
  public static op void f(int);
  public static op void g(int);
  public LocalClient(){}
  public static process P1{
    send f(1);
    while(true){
      inni void f(int x){
        System.out.println("P1 serviced: " + x);
        send g(1);
      }
    }
  }
  public static process P2{
    send f(2);
    while(true){
      inni void g(int x){
        System.out.println("P2 serviced: " + x);
        send f(2);
      }
      [] void f(int x){
        System.out.println("P2 serviced: " + x);
        send g(2);
      }
    }
  }
}

public class RemoteClient{
        public static op void h(int);
        public static cap void (int) f;
        public RemoteClient(cap void (int) f){this.f = f;}
        public static process P3{
                while(true){
                        inni void h(int x){
                                System.out.println("P3 serviced: " + x);
                                send f(3);
                        }
                        [] void f(int x){
                                System.out.println("P3 serviced: " + x);
                                send h(3);
                        }
                }
        }
}
```

Figure 4.6: JR program with local and remote operations within Inni statements.

```
import threading
def t1():
  for i in xrange(20):
    print "t1"
def t2():
  for i in xrange(20):
    print "t2"

tOne = threading.Thread(target=t1)
tOne.start()
tTwo = threading.Thread(target=t2)
tTwo.start()
tOne.join()
tTwo.join()
```

Figure 4.7: Simple multithreaded program that may result in interleaved output.

```
class Foo(PSObject):
  @OpMethod
  def op1(self, inv):
    print "hello", self.op2.call()
  @OpMethod
  def op2(self, inv):
    print "world"

def main():
  f = Foo()
  f.op1.send()
```

Figure 4.8: PySy program that deadlocks because of output thread-safety.

1. *op1()* begins a **print** statement and grabs the output buffer lock for its thread. The string "hello" gets written to the buffer, but the lock is not released because the newline character has not been read. (context switch)

2. `self.op2.call()` is executed.

3. *op2()* tries to begin its print statement.

4. *op2()* fails to get the output lock because calls and sends are handled in a different thread of control, and *op1* still holds the lock. *op2()* waits.

5. *op1()* and *op2* are waiting for each other to finish. Deadlock.

Figure 4.9: Execution trace for thread-safe output that results in deadlock.

```
def main():
  x = 2
  class f_ArmCode(ArmCode):
    def codeBlock(self, inv):
      #x is not visible here
      return x * 2
```

Figure 4.10: Program illustrating the scoping problem.

```
class IntWrapper(object):
  def __init__(self, val):
    self.val = val
def main():
  x = IntWrapper(2)
  f = InniOp.create()
  f.send()
  class f_ArmCode(ArmCode):
    def __init__(self,x):
      self.x = x
    def codeBlock(self, inv):
      self.x.val = self.x.val  * 2
  f_arm = InniArm(f, f_ArmCode(x))
  inni = Inni(f_arm)
    inni.service()
```

Figure 4.11: How to share data from outer scopes with ArmCode as a class implementation.

```
class IntWrapper:
  def __init__(self, val):
    self.val = val
  def __str__(self):
    return str(self.val)

def main():
  x = IntWrapper(2)
  def foo():
    print x
    x.val = 3
  foo()
  print x

#outputs:
#2
#3
```

Figure 4.12: Program showing how to circumvent the no write restriction in a closure by using a wrapper.

Chapter 5

Performance

Section 5.1 presents PySy's performance results from several micro-benchmarks that measure the performance for individual PySy operations and macro-benchmarks that compare the performance of PySy to River/Trickle and Python's **threading** and **multiprocessing** modules for well-known parallel and distributed algorithms. We will give an in-depth analysis of these results and, where appropriate, we will indicate potential bottlenecks and offer solutions to alleviate these bottlenecks. Section 5.2 presents a qualitative analysis of PySy with respect to **multiprocessing** and River/Trickle.

5.1 Benchmarks

Each of the experiments discussed in this section were executed on UC Davis's Computer Science Instructional Facility (CSIF). The CSIF is a network of computers used by UC Davis's computer science students for classwork. For single process experiments, the tests were completed on pc25, which is a 3.0GHz dual core Intel Xeon processor with 3GB of RAM running Fedora Core 15 with Linux kernel 2.6.42.12-1, using NFS on a gigabit network, and Python 2.7.2. For the distributed tests, we used several other identical CSIF systems. Besides the CSIF, each benchmark was also run on different hardware configurations, where the general pattern of timing results matches the results from the CSIF. However, in this

chapter, we will only display the CSIF results. Each experiment was performed late in the evening to minimize the system load. The timing results for each experiment were averaged to attain the mean time per operation (for micro-benchmarks) or program (for macro-benchmarks). Each repeated experiment was consistent with the original experiment and had low variance.

5.1.1 Micro-benchmarks

The micro-benchmarks perform experiments on invocations and invocation servicing. For each experiment, we monitor the total lapsed time until completion. We found the average time per experiment and display those results in Table 5.1. The results also show PySy's performances with and without quiescence detection. In many cases, the difference is staggering. We will explain this phenomenon throughout the discussion of the results. Table 5.2 shows the overhead for basic Python operations, such as thread creation, lock acquisition, and method calls.

The inOpSend(local) test repeatedly invokes a local InniOp's *send()* method. The *send()* invocation is relatively fast compared to *call()* because of asynchrony. This is seen in the experiments with and without quiescence detection. Since *send()* never blocks, the thread executing the invocation will never have to wait on a PySy resource. Because of this, the *send()* implementation does not require any code to interface with quiescence detection. The inOpSend(remote) test's timing is consistent with the above description except for a .35 ms difference, which is due to the cost of sending an asynchronous message on the local subnet.

The inOpRecv(local) test repeatedly invokes a local InniOp's *receive()* method and monitors the time to retrieve the first element of the invocation list. Before each experiment starts, we populate the InniOp's invocation list with asynchronous invocations. The results show a huge disparity between the experiments with quiescence detection enabled and disabled. With quiescence detection enabled, the *receive()* implementation contacts the VMManager on at least three occasions. QD comprises the vast majority of the timing

results. In the future, we would like to investigate decreasing the number of messages we send to the VMManager (see Section 6.2.2.2). Also, we plan on reimplementing quiescence detection's message passing from a synchronous message to an asynchronous message and serialize the message processing on the VMManager. This should roughly cut the cost of these messages by 50%.

The Inni(local) experiment repeatedly services an InniOp with an input statement with one arm and is pre-populated with invocations. The experiment monitors the time to completely service an invocation with the Inni statement. This experiment shows that an Inni is six times slower than a receive without quiescence detection and roughly the same cost with quiescence detection. The Inni is slower than a receive because on every service attempt the Inni must acquire the global merge lock from the VMManager to merge equivalence classes (see Section 4.3). This requires a network message. Also, the Inni performs a merge of equivalence classes on every *service()* operation. We should be able to significantly improve performance by monitoring equivalence class changes or, more precisely, lack thereof. If an operation's equivalence class has not changed since the last *service()*, then the merge operation is unnecessary. It is important to note that the performance of the Inni, however, is not constant. The Inni statement's performance will decline as the number of arms increases (especially in the remote case). The optimization discussed above will be very effective in the remote case, where the cost of acquiring a lock is significantly higher.

The inOpCall(local)[Receive] experiment repeatedly makes synchronous invocations to a local InniOp and services the invocation with a receive from a local process. This experiment shows the significant cost of a PySy *call()* to an InniOp.

The inOpCall(local)[Inni] experiment repeatedly makes synchronous invocations to a local InniOp and services the invocation with an Inni statement in a local process. This experiment reaffirms the results from inOpCall(local).

The opMethodSend(local) experiment repeatedly performs a *send()* to a local OpMethod with an empty code block. A *send()* invocation to an OpMethod spawns a

Test Name	Invocation	Service	Time with QD (ms)	Time w/o QD (ms)
inOpSend(local)	send to local InniOp	None	.05	.05
inOpRecv(local)	None	receive	2.49	.07
Inni(Local)	None	Inni	2.82	.33
inOpCall(local)[Inni]	call to local InniOp	Inni	2.94	.79
inOpCall(local)[Receive]	call to local InniOp	receive	3.52	.79
inOpSend(remote)	send to remote InniOp	None	.40	.39
inOpRecv(remote)	None	receive	3.44	.44
inOpCall(remote)[Inni]	call to remote InniOp	Inni	4.07	1.27
inOpCall(remote)[Receive]	call to remote InniOp	receive	4.55	1.17
opMethodSend(local)	send to local OpMethod	OpMethod	.14	.13
opMethodCall(local)	call to local OpMethod	OpMethod	3.40	.40
opMethodSend(remote)	send to remote OpMethod	OpMethod	.48	.47
opMethodCall(remote)	call to remote OpMethod	OpMethod	4.41	.79
InniOpCreation	Creating a local InniOp	N/A	.05	.05

Table 5.1: PySy's performance for micro-benchmarks.

new thread (using the **threading** module) to service the invocation and executes the OpMethod's code block. We may be able to slightly improve performance by implementing a threadpool for each VM to avoid the continual cost of creating threads for each asynchronous invocation.

The opMethodCall(local) experiment repeatedly performs a synchronous invocation on a local OpMethod with an empty code block. This experiment shows that a *call()* is three times slower than a *send()* to an OpMethod.

Appendix D presents the same experiments from this section, but implemented using JR. In most cases, PySy is about 10x slower than JR. We understand that a direct comparison between PySy and JR is not completely fair because of language specific issues, i.e., interpreted vs. compiled. With that said, we believe these performance numbers are quite reasonable.

Test Name	Description	Time (ms)
PyMethodCall	Time to invoke a method with empty code block in Python.	.002
PyAcquireLock	Time to acquire a lock from Python's threading module.	.001
threadCreation(thr)	Time to start up a new thread using Python's threading module.	.11
threadCreation(mp)	Time to start up a new process using Python's multiprocessing module.	1.30

Table 5.2: Timing results for basic Python functionality.

5.1.2 Macro-benchmarks

This section compares the performances of PySy, **multiprocessing**, and River/Trickle on three standard concurrent programming benchmarks: Readers/Writers, Fast Fourier Transform (FFT), and Matrix Multiplication. As an overview of the performance results, the results varied between the aforementioned packages. For embarrassingly parallel programs (FFT and Matrix Multiplication), PySy tends to be two times slower than **multiprocessing**, but for programs that require significant synchronization (Readers/Writers), PySy is significantly faster (5-6 times faster). The performance comparison between PySy and River/Trickle also depends on the amount of required synchronization and the problem. The rest of this section provides greater detail and analysis of our results.

5.1.2.1 Readers/Writers

The first macro-benchmark we will present is the well-known Readers/Writers problem [7]. The program simulates multiple clients reading from and writing to a resource with the restriction that a reader can only access a resource if and only if there are currently no writers accessing it. Also, a writer can only access a resource if and only if no readers and no writers are currently accessing it. We measure the time it takes for n readers and m writers spread across s servers to perform o operations. Appendix E provides the PySy, River, and **multiprocessing** implementations for the Readers/Writers problem. The timing results for PySy with and without quiescence detection, River, and **multiprocessing** are presented in Table 5.3. We were not able to implement a Readers/Writers River

program that uses threading to create multiple readers and writers on a single machine (as in our implementations for **multiprocessing** and PySy). Our attempts to implement Readers/Writers in this fashion ran into inexplicable deadlocks. For this reason, Table 5.3 does not include River performance numbers for some of the experiments. Our River implementation, instead, creates one reader or writer per River VM. Besides this, the River implementation is semantically equivalent to **multiprocessing** and PySy implementations.

To help aid the comparison, we added an experiment to measure the start-up cost for each package. River programs differ from PySy and **multiprocessing** programs in their start-up. River VMs are started prior to launching the program by the user, so they do not incur as much start up cost as PySy or **multiprocessing** (processes are created dynamically). More specifically, PySy forks a new process that uses ssh to create a new PySy VM process on the specified machine. The columns in Table 5.3 with the number of operations set to zero show the approximate start up cost for fifteen servers using **multiprocessing**, River, and PySy. This shows how truly expensive it is to start up a PySy program with a large number of VMs. However, PySy does provide more dynamism than River and an easier interface to create new VMs than **multiprocessing**. PySy users are able to dynamically create new processes on different hosts, while River users must manually start up VMs from the command line or a shell script. This makes starting River programs with large amounts of VMs cumbersome and less dynamic. The user must predict the number of required VMs apriori. This may be difficult to predict, especially if the user desires to execute multiple independent River programs. After removing the start-up costs for PySy, the River Readers/Writers implementation is about three times faster. The PySy Readers/Writers implementation uses an input statement to synchronize the reader or writer with the resource allocator. This method is more expensive than a synchronous message (as in the River implementation), but also provides additional flexibility, i.e., the ability to prioritize readers or writers or different scheduling algorithms.

Package	Readers	Writers	Servers	Operations	Time (s)
multiprocessing	10	5	1	100	33.412
PySy with QD	10	5	1	100	22.763
PySy without QD	10	5	1	100	4.051
multiprocessing	10	5	2	100	33.410
PySy with QD	10	5	2	100	32.072
PySy without QD	10	5	2	100	4.580
multiprocessing	10	5	1	500	164.896
PySy with QD	10	5	1	500	129.913
PySy without QD	10	5	1	500	24.410
multiprocessing	10	5	2	500	164.742
PySy with QD	10	5	2	500	157.289
PySy without QD	10	5	2	500	24.780
multiprocessing	10	5	15	0	0.754
River	10	5	15	0	0.007
PySy with QD	10	5	15	0	5.004
PySy without QD	10	5	15	0	5.001
multiprocessing	10	5	15	100	35.286
River	10	5	15	100	1.322
PySy with QD	10	5	15	100	92.809
PySy without QD	10	5	15	100	8.537
River	10	5	15	500	6.666
PySy with QD	10	5	15	500	432.569
PySy without QD	10	5	15	500	22.720

Table 5.3: Performance results for PySy and multiprocessing for the Readers/Writers program.

5.1.2.2 Fast Fourier Transform

This section compares PySy to the Python packages **threading**, **multiprocessing**, and Trickle for calculating the first N coefficients of the function $(x+1)^x$ defined on the interval $[0, 2]$ using a Fast Fourier Transform (FFT) algorithm modified from the Java Grande Forum's benchmarks [12]. Appendix F contains the source code for this algorithm implemented using PySy, Trickle, and **multiprocessing**. Each program is parameterized with the number of coefficients to calculate and the number of server threads/processes we

would use to divide up the work. We divided the work using the pre-calculated strips idiom. This experiment measures the total elapsed time for each program to run to completion. Table 5.4 displays these results.

This experiment clearly shows the cost of quiescence detection (QD). With QD, PySy is 10-15 times slower than the **threading** and **multiprocessing** implementations and 4-5 times slower than Trickle. However, without QD, PySy is 1.5-2 times slower (for one server) than **multiprocessing** and **threading**, but 10-15% faster than Trickle. As the number of servers increases, **multiprocessing**, Trickle, and PySy see an improvement in performance (all approach maximum linear speedup), while **threading** becomes slightly slower. This shows the benefit of using multiple processes in CPython over a multithreaded implementation.

5.1.2.3 Matrix Multiplication

This section compares PySy to **multiprocessing** and Trickle for a naive matrix multiplication algorithm for an NxN matrix. Appendix G provides the Matrix Multiplication implementation for PySy, Trickle, and **multiprocessing**. Each program divides the work using pre-calculated strips to divide the work amongst a user-specified number of servers. This experiment measures the total elapsed time to complete the NxN matrix multiplication.

This experiment shows the cost for starting up new VMs in PySy. In the case of N=100, the cost of doing a 100x100 matrix multiplication increases as the number of servers increases. This is because the parallelization is overcompensated by the start up costs for the PySy VM. As the size of the array increases, this is no longer true. Also, this experiment shows that Trickle has considerably less start up costs than PySy because the user is required to manually launch River processes on the local network prior to executing a Trickle program with multiple VMs.

Another interesting phenomenon illustrated by this experiment is the equality between PySy programs with quiescence detection enabled and disabled. This was not true

Package	N	Number of servers	Time(s)
threading	10000	1	22.838
multiprocessing	10000	1	20.984
Trickle	10000	1	41.724
PySy with QD	10000	1	195.405
PySy without QD	10000	1	36.684
threading	10000	2	28.575
multiprocessing	10000	2	11.007
Trickle	10000	2	20.897
PySy with QD	10000	2	138.308
PySy without QD	10000	2	18.785
threading	10000	5	27.964
multiprocessing	10000	5	4.940
Trickle	10000	5	14.118
PySy with QD	10000	5	44.678
PySy without QD	10000	5	9.049
threading	100000	1	215.351
multiprocessing	100000	1	200.850
Trickle	100000	1	405.988
PySy with QD	100000	1	1101.736
PySy without QD	100000	1	336.778
threading	100000	2	277.705
multiprocessing	100000	2	102.011
Trickle	100000	2	209.059
PySy with QD	100000	2	961.051
PySy without QD	100000	2	169.028
threading	100000	5	280.548
multiprocessing	100000	5	41.917
Trickle	100000	5	83.387
PySy with QD	100000	5	444.323
PySy without QD	100000	5	69.404

Table 5.4: Performance results for threading and multiprocessing for calculating the first N coefficients of the function $(x+1)^x$ defined on the interval $[0, 2]$.

in the FFT example in Section 5.1.2.2 or in the micro-benchmarks shown in Table 5.1. This stems from matrix multiplication being embarrassingly parallel and not requiring any synchronization. As the number of necessary synchronization mechanisms increases (e.g., bag of tasks instead of pre-calculated strips), the worse PySy with quiescence detection will perform.

Table 5.5 shows the results for **multiprocessing**, Trickle, and PySy. Overall, PySy is shown to be about 1.5-2 times slower than **multiprocessing** and Trickle.

5.2 Qualitative

The quantitative analysis in Section 5.1 has shown PySy (without quiescence detection) to be a factor of two slower than Python's **multiprocessing** module and mixed results with Trickle. Despite this, PySy provides a more expressive interface than both **multiprocessing** and River (and Trickle), more control over invocation selection, an easier and more dynamic interface for user interaction with creating and interfacing with processes, and quiescence detection. This section presents a qualitative analysis of PySy and compares the implementation of concurrent programs using PySy, River, and **multiprocessing**. This section does not directly make comparisons between PySy and **threading**, however, because the interfaces for **threading** and **multiprocessing** are extremely similar, so most of the analysis is applicable to both.

The most glaring qualitative difference between PySy and **multiprocessing** is the creation and synchronization of remote processes. In PySy, the user invokes the API method *createVM* to start a new remote process. Next, the user creates objects on the VM by using *createInstance*. The object's operations control the synchronization between the main VM and the remote VM processes. When creating remote processes using **multiprocessing**, the user must create a Manager object (see Figures G.3, G.4, and G.5) and register synchronization mechanisms (in the Matrix Multiplication example, shared queues). Next, the user creates the new processes. The final step is to create the client to connect with the

Package	N	# of servers	Time (s)
multiprocessing	100	1	1.160
Trickle	100	1	.326
PySy with QD	100	1	1.441
PySy without QD	100	1	1.435
multiprocessing	100	2	1.026
Trickle	100	2	.187
PySy with QD	100	2	1.257
PySy without QD	100	2	1.361
multiprocessing	100	5	.969
Trickle	100	5	.152
PySy with QD	100	5	2.236
PySy without QD	100	5	2.140
multiprocessing	500	1	38.423
Trickle	500	1	40.225
PySy with QD	500	1	78.326
PySy without QD	500	1	77.125
multiprocessing	500	2	20.617
Trickle	500	2	20.325
PySy with QD	500	2	39.483
PySy without QD	500	2	39.168
multiprocessing	500	5	9.393
Trickle	500	5	15.826
PySy with QD	500	5	17.233
PySy without QD	500	5	17.505
multiprocessing	1000	1	318.634
Trickle	1000	1	327.474
PySy with QD	1000	1	558.001
PySy without QD	1000	1	552.607
multiprocessing	1000	2	159.294
Trickle	1000	2	166.502
PySy with QD	1000	2	281.746
PySy without QD	1000	2	279.971
multiprocessing	1000	5	66.554
Trickle	1000	5	66.963
PySy with QD	1000	5	115.753
PySy without QD	1000	5	115.255

Table 5.5: Performance results for PySy and multiprocessing for multiplying NxN matrices.

manager. Overall, the PySy Matrix Multiplication implementation (see Figures G.1 and G.2) is 97 lines of code, the Trickle implementation is 58 lines of code, and the **multiprocessing implementation** requires 162 lines of code. The Trickle implementation requires considerably less code partly because the user does not have to write any code to dynamically create remote processes. The Trickle (and River) remote processes are created before running the program, which makes the Trickle start up costs considerably smaller, but, on the other hand, provides little dynamism and, in the case of a large number of processes, makes its usage cumbersome. Also, problems using the pre-calculated strips idiom like our Matrix Multiplication and FFT implementations are modeled nicely in Trickle because it provides a very simple user interface that easily distributes jobs across remote machines.

PySy, River, and **multiprocessing** provide shared queue implementations, but with different terminology. PySy provides a shared queue in the guise of an InniOp, River has a message queue for each VM, and **multiprocessing** uses the aptly named Queue. Fundamentally, they all provide a first-come, first-serve mechanism for synchronization between concurrently executing program segments. However, PySy is able to combine its shared queue implementation with its input statement to allow a process to provide synchronization over multiple shared queues, synchronize the servicing of invocations (River provides this feature, too), and dynamically schedule the servicing of invocations. **multiprocessing** and River do not provide multi-way receive functionality.

We mentioned above that the shared queue implementations from PySy, River, and **multiprocessing** provide invocation selection using a first-come, first-serve guarantee. PySy, with the use of an Inni, is able to modify the invocation selection semantics using a by expression or a such that clause. River, however, does not provide an input statement, but does provide an equivalent to PySy's such that clause. **multiprocessing**'s Queue does not provide any of these features because it does not allow arbitrary access into the queue (the user *must* remove invocations from the head of the queue) and cannot easily provide these additional invocation selection semantics.

PySy also provides automatic program termination, which is not available (to

the author's knowledge) in any other Python concurrency package. Terminating concurrent programs with PySy's quiescence detection requires no user effort. In very simple programs, this may not be too useful to the user, but in complicated concurrent programs, ones with many interacting processes, quiescence detection trivially simplifies the termination process.

Chapter 6

Conclusion and Future Work

This chapter reflects upon our methodology and provides a roadmap for future development. Section 6.1 presents our conclusions and final thoughts on the topics discussed throughout this thesis. Section 6.2 gives an overview of our ideas for future work.

6.1 Conclusion

This thesis has discussed the design and implementation of the PySy programming package for Python 2.x. We have discussed our methodology for adapting the features of the extended Java language JR to PySy, demonstrated the usage of each PySy feature, discussed our implementation, and shown PySy's performance across several micro-benchmarks and well-known concurrent programs.

The development process was a balancing act. The ideologies of Python and Java are quite opposing. In the imperative programming world, Python and Java appear to be polar opposites. Java provides an environment that focuses on statically enforcing type safety and general object oriented programming structure, while Python is dynamically typed (yet still type safe) and puts the onus on its users to correctly manage the interaction between objects. So, adapting a product of Java to Python was bound to have interesting issues.

With that said, PySy is not a direct mapping of JR to Python. Throughout the development process, we attempted to merge the functionality of JR into Python without losing the spirit of JR and without deviating from conventional Python programming practices. This is evident in our solutions for the scoping problem discussed in Section 4.6, the interface for the semaphore abstraction (see Section 3.5.4), and the return values of synchronous invocations to OpMethods (see Section 3.5.1).

PySy provides a reasonable alternative to other Python concurrency programming packages. The quantitative results in Chapter 5 show that PySy (without quiescence detection) is 1.5-2 times slower than Python's **multiprocessing** package for distributed programs that utilize a Fast Fourier Transform algorithm and perform matrix multiplication. We believe that we can drastically improve PySy's performance by implementing a few optimizations. We discuss several of these optimizations in Section 6.2.2.1.

While PySy is admittedly slower than **multiprocessing**, we feel that PySy provides an easier and more expressive environment than **multiprocessing** for many distributed programs. For example, PySy provides a simple and seamless interface to create new processes on remote machines, while the **multiprocessing** package requires a more coordinated effort by the user to get the same functionality. Appendix F and Appendix G show the user code for two distributed programs using PySy and **multiprocessing** and illustrate how significantly easier it is to implement these programs using PySy than **multiprocessing**.

As a developer, one of the utmost concerns with any product is its usefulness and its ease of use. Most of PySy is intuitive, expressive, and well conceived, while other parts are quite confusing, e.g., Inni. PySy's source code contains significant documentation that will be distributed with an HTML Pydoc [22] along with numerous examples highlighting the usage of each feature. Also, this thesis and the JR book [17] are excellent supplementary resources. Through the latter stages of PySy's development, we had two UC Davis undergraduate students helping out with the project. This gave us an excellent opportunity to observe the difficulty in learning PySy. Neither student had previous experience with

Python, JR, or concurrent programming. While working on PySy, they learned all three concurrently. They mainly used the JR book as a guide to learn about the provided concurrent programming mechanisms and the PySy source and examples to see the realization of the JR features. The feedback was positive with some criticisms surrounding the complexity of some constructs, e.g., Inni and the required user interaction with the Invocation structure (see Section 3.4) for OpMethods. Unfortunately, most of the criticisms just cannot be completely solved because of the functionality's complexity. A lot of the struggles endured during PySy's interface design have provided a new appreciation for the absolute power of an elegant language design and the potential drawbacks and pitfalls, especially from the user's perspective, for providing complex language features in package form.

6.2 Future Work

The development of PySy has concentrated on providing JR-like functionality and establishing a friendly user-interface. Chapter 3 describes both of these topics. However, there is still more work to be done. This section discusses several of our ideas for PySy's future development.

6.2.1 Features

6.2.1.1 Quantifiers

Chapter 3 discussed the design and user interface for each adapted JR feature. Currently, PySy does not implement the JR quantifier construct. The development of quantifiers is currently being planned for the inni statement, the co statement, and, possibly, for the process construct. The actual implementation for adding quantifiers should not be terribly difficult, but it does present some challenges from a user interface perspective. This is especially true for the process construct. Recall from Section 3.5.2, the process is a Python method decorated with the **Process** decorator. If we were to implement the quantifier feature for the **Process** decorator, then we would likely create a quantifier object

and pass it as a parameter to the **Process** decorator. Remember the quantifier syntax for JR looks like:

`(initialization_expr; termination_expr; increment_expr)` or

`(initialization_expr; termination_expr; increment_expr; st_expr)`

For example, a quantifier in JR that looks like: `(int i = 0; i < 10; i++; i%2 == 0)` would result in the creation of a new process for each i that satisfies the such that expression and adds i into each process's scope with the appropriate value. Achieving this in Python may be messy. The most glaring aspect is the creation of the quantifier. Is the quantifier just a reference to a lambda expression for each of the components of the quantifier? Also, how does a quantified process construct change the current method signature for processes? How do we handle the injection of new attributes into local scope? These are just a few questions we will have to answer in order to successfully provide quantifiers in PySy.

6.2.1.2 Parameterized VMs

Section 2.2.1 shows the usage of JR's parameterized VM. PySy does not currently support parameterized VMs. PySy users must use our generic VM construct, which is a Pyro proxy, and cannot add user-defined attributes or operations to the VM. This slightly limits the VM's flexibility. The user is forced to create a VM and, subsequently, instantiate objects through *PySy.createInstance()*. With the current implementation, objects created on a VM are only accessible through the remote reference returned from *PySy.createInstance()*.

6.2.2 Performance

In Chapter 5, we presented PySy's performance results for micro-benchmarks and several macro-benchmarks. Our results showed that PySy (without quiescence detection) was generally 1.5-2 times slower than **multiprocessing**. This section discusses several ideas for improving our performance and closing the gap with other Python concurrent programming alternatives.

6.2.2.1 Optimizations

Table 5.1 shows the cost for the PySy's Inni construct. The example used in the micro-benchmarks was an Inni with one arm with a local InniOp. The cost of the merge increases, however, with remote operations. Each equivalence class lock must be acquired, a new lock must be selected, and the old locks must be forwarded to the new lock. This procedure is executed on every *service()* operation, even in the situation where the equivalence class has not changed. We would like to implement an optimization to monitor the changing of equivalence classes, which will allow us to determine if a merge is necessary in one message (even in the remote case).

PySy currently creates a new thread for each call and send invocation. This is not only expensive (and unnecessary), but also may cause issues on systems that severely limit the number of user-created threads. We would like to create a threadpool (or even a Process pool using **multiprocessing**) for each VM to process invocations and only take the performance hit for thread creation at the VM creation time.

6.2.2.2 Quiescence

The performance results described in Chapter 5 give solid evidence that the current quiescence detection implementation is incredibly expensive. The JR book [17] also arrived at the same conclusion. The actual quiescence detection feature is extremely useful in concurrent programs, but, currently in PySy, it may be too expensive for real world applications that require significant synchronization. We propose searching for a better implementation for quiescence detection, which requires fewer messages, and, thus, reduces overhead.

6.2.3 Networking

At the beginning of PySy's development, we decided to use Pyro (version 3) for the distributed sharing of resources. Pyro3 was mature, stable, and easy to use. At that point, Pyro4 had just been released in beta, but it was too unproven to utilize for this

work. However, during the development of PySy, Pyro4 has matured considerably. The Pyro4 package is a completely revised and simpler version of Pyro3. The main mechanisms remain the same, but many of Pyro3's extra features have been removed. We have recently run several experiments that show using Pyro4 will improve performance by 5-10%. Before we adopt Pyro4 as our distributed resource component, however, we would like to do some more analysis.

With that said, our implementation is not completely committed to Pyro. We would like to experiment with other libraries that provide similar functionality. For example, the Super Flexible Messaging (SFM) protocol in River [10]. Finally, if these other alternatives are are not sufficient, then we can develop our own networking package that is tailored specifically to PySy's needs.

6.2.4 Python 3

Python has discontinued its feature development of Python 2.x with version 2.7.2. All future development is being applied to Python 3. It only makes sense to port PySy to the future of the Python language. The Python 3 distribution provides an automatic conversion tool 2to3.py [22] that automatically converts Python 2.x programs to Python 3. However, Python 3 is not compatible with Pyro3. So, to convert PySy to Python 3, we would have to replace Pyro3 with another distributed programming package, such as Pyro4. This transition is currently being explored by UC Davis undergraduate students Allyson Cauble-Chantrenne and David Kavaler.

6.2.5 Profiling and Debugging

A major part of the software life cycle depends on discovering bugs and improving performance. However, Python has limited options for debugging and profiling distributed and concurrent programs. The development of such tools would greatly increase development efficiency and provide insight into potential performance bottlenecks.

Appendices

Appendix A

Invocation Selection

Understanding the semantics of JR's invocation selection is key to understanding JR program behavior. This section is intended to expand on the invocation selection discussion in Section 2.2.5. It walks through the JR program example in Figure A.1 and discusses the invocation selection procedure in detail.

The Main class in Figure A.1 defines four operations. For simplicity, the program uses an operation as a synchronization mechanism inside of *p1* to guarantee deterministic output. The main function queues up all invocations that need processing and then sends a message to *p1* to allow it to start servicing the invocations. See the first row in Table A.1 for a current view of each operation's invocation list. Each element in operations x, y, and z is a tuple consisting of the invocation parameter and the distributed timestamp, respectively.

Each time an Inni statement is executed, the Inni statement must sort all of its arms by the earliest distributed timestamp. This arm ordering describes the order that the Inni statement will iterate through the arms to find a serviceable invocation. The sorted arms column shows this ordering at each iteration of the while loop. During iteration one, the Inni statement will try to service an invocation in y first. y's arm contains a by clause that will select the invocation with the lowest value. Currently, y has four pending invocations with values 8, 6, -2, and 9, so, -2 is chosen. The selected invocation and the resulting invocation lists are also described in Table A.1. On the second and third

iteration, y will be selected first, again, and invocations 6 and 8 will be selected, respectively. On the fourth iteration, x's arm is selected first. A such that clause appears in x's arm that is only satisfied when the invocation parameter is even. The invocation 6 is the first invocation that satisfies this criteria and is selected. x's arm is also selected first in the fifth iteration. However, this time there is no invocation in x's invocation list that contains an even invocation parameter. Since there are no serviceable invocations in x's arm, the input statement tries the next arm in the sorted arms list, z. z's arm contains a such that clause and a by clause that is looking for the largest odd invocation parameter. The Inni statement will select 11 during this iteration and 7 and 1 during the subsequent iterations, respectively. On the eighth iteration, none of the arms contain any serviceable values, so the else arm is selected and it breaks from the while loop.

```
public class Main
{
  static op void x(int);
  static op void y(int);
  static op void z(int);
  static op void start();
  public static void main(String[] args)
  {
    send y(8);send y(6);
    send x(1);
    send z(11);
    send y(-2);
    send x(6);
    send z(12);send z(1);send z(0);send z(7);
    send y(9);
    send start();

  }

  static process p1
  {
    receive start();
    while(true){
      inni void x(int i) st i%2==0{
        System.out.println("x=" + i);
      }
      [] void y(int j) by j{
        System.out.println("y=" + j);
      }
      [] void z(int k) st k%2==1 by -k{
        System.out.println("z=" + k);
      }
      [] else{
        break;
      }
    }

  }
}
```

Figure A.1: Four arm Inni statement with a variety of different synchronization and scheduling expressions

Iteration	Sorted arms	Operation selected	Invocation selected	Resulting invocation lists
0	N/A	N/A	N/A	x=[(1,2),(6,5)] y=[(8,0),(6,1),(-2,4),(9,10)] z=[(11,3),(12,6),(1,7),(0,8),(7,9)]
1	[y,x,z]	y	(-2,4)	x=[(1,2),(6,5)] y=[(8,0),(6,1),(9,10)] z=[(11,3),(12,6),(1,7),(0,8),(7,9)]
2	[y,x,z]	y	(6,1)	x=[(1,2),(6,5)] y=[(8,0),(9,10)] z=[(11,3),(12,6),(1,7),(0,8),(7,9)]
3	[y,x,z]	y	(8,0)	x=[(1,2),(6,5)] y=[(9,10)] z=[(11,3),(12,6),(1,7),(0,8),(7,9)]
4	[x,z,y]	x	(6,5)	x=[(1,2)] y=[(9,10)] z=[(11,3),(12,6),(1,7),(0,8),(7,9)]
5	[x,z,y]	z	(11,3)	x=[(1,2)] y=[(9,10)] z=[(12,6),(1,7),(0,8),(7,9)]
6	[x,z,y]	z	(7,9)	x=[(1,2)] y=[(9,10)] z=[(12,6),(1,7),(0,8)]
7	[x,z,y]	z	(1,7)	x=[(1,2)] y=[(9,10)] z=[(12,6),(0,8)]
8	[x,z,y]	y	(9,10)	x=[(1,2)] y=[(9,10)] z=[(12,6),(0,8)]

Table A.1: Invocation selection table for Figure A.1

Appendix B

Imported Objects and Functions

Tables B.1 and B.2 briefly describe PySy's user-interface. For more information on each object, see the referenced section.

Object	Description	Reference
ArmCode	Code block executed for an InniArm (decorator).	Section 3.6
By	Code block executed for an InniArm's by expression (decorator).	Section 3.6
Co	Object representing the JR Co statement.	Section 3.9
CoArm	A single arm of a JR Co statement.	Section 3.9
Control	Enumeration of possible control status changes in an InniArm and CoArm.	Section 3.10
Inni	Object representing the JR input statement.	Section 3.6
InniArm	A single arm of a JR input statement.	Section 3.6
InniArmElseAfter	The arm for the JR input statement's optional else after.	Section 3.6
InniElse	The arm for the JR input statement's else.	Section 3.6
InniOp	JR operation serviced by an input statement.	Section 3.5.3
Invocation	Contains contextual information about the invocation of an operation.	Section 3.4
OpMethod	JR operation serviced by a method (decorator).	Section 3.5.1
PSObject	Base class for all PySy objects.	Section 4.2.2
Process	JR Process construct (decorator).	Section 3.5.1
StaticOpMethod	JR OpMethod with static modifier (decorator).	Section 3.5.1
SuchThat	Code block executed for an InniArm's such that clause (decorator).	Section 3.6

Table B.1: The objects imported by the PySy package.

Function	Description
PySy.createInstance	Creates a new remote instance of an object.
PySy.createVM	Creates a new virtual machine.
PySy.exit	Begins shutdown procedures and sets the return status of the program to a user-specified value.
PySy.init	Required method call invoked at global scope to begin PySy program.
PySy.nap	Sleeps for a user specified amount of time (in seconds).
PySy.shutdown	Terminates the PySy program. This is only needed if quiescence detection is disabled.
PySy.traceback	Prints out the traceback for an exception.
PySy.yield_	Yields control of the current thread.

Table B.2: The functions provided by the PySy API.

Appendix C

PySy's Operation Interface

```
class Operation(Remote):
    """
    @summary: Abstract base class of an operation.
    """
    def send(self, inv=None):
        """
        @summary: Asynchronous invocation of an operation
        """
        raise NeedToOverrideError
    def call(self, inv=None):
        """
        @summary: Synchronous invocation of an operation
        """
        raise NeedToOverrideError
    def receive(self):
        """
        @summary: Service the first invocation (for InniOps).
        This method throws an error for OpMethods.
        """
        raise NeedToOverrideError
    def length(self):
        """
        @summary: Gets the length of an operation's invocation list
        """
        return 0
    def isNoop(self):
        raise NeedToOverrideError
```

Figure C.1: User-interface for the Operation construct.

Appendix D

JR Micro-Benchmark Performance

Table D.1 presents JR's performance using the same micro-benchmarks depicted in Table 5.1.

Test Name	Invocation	Service	Time with QD (ms)	Time without QD (ms)
inOpSend(local)	send to local InniOp	None	.002	.002
inOpRecv(local)	None	receive	.86	.001
Inni(Local)	None	Inni	2.82	.33
inOpCall(local)[Inni]	call to local InniOp	Inni	1.28	.09
inOpCall(local)[Receive]	call to local InniOp	receive	1.38	.07
inOpSend(remote)	send to remote InniOp	None	.16	.16
inOpRecv(remote)	None	receive	1.30	.16
inOpCall(remote)[Inni]	call to remote InniOp	Inni	2.07	.34
inOpCall(remote)[Receive]	call to remote InniOp	receive	1.13	.31
opMethodSend(local)	send to local OpMethod	OpMethod	.14	.13
opMethodCall(local)	call to local OpMethod	OpMethod	.22	.0002
opMethodSend(remote)	send to remote OpMethod	OpMethod	.04	.04
opMethodCall(remote)	call to remote OpMethod	OpMethod	.62	.08

Table D.1: JR's performance for micro-benchmarks.

Appendix E

Readers/Writers Examples

Figures E.1, E.2, E.3, and E.4 show the implementation of distributed Readers/Writers program using PySy. Figures E.5, E.6, E.7, E.8, and E.9 show an implementation of a distributed Readers/Writers program using **multiprocessing**. Figures E.10, E.11, and E.12, show an implementation of a distributed Readers/Writers program using River.

```
rand = random.Random()

class RWController(object):
  def __init__(self, performance, nreaders, nwriters, nservers, niters):
    self.performance = performance
    self.done = InniOp.createSem()
    self.nreaders = nreaders
    self.nwriters = nwriters
    self.nservers = nservers
    self.niters = niters
    self.test()


  def test(self):
    self.performance.run([Work(self.work)])


  @timing
  def work(self,*args):
    global rand
    rwa = PySy.createInstance("rwAllocator.RWAllocator", self.nwriters \
      + self.nreaders, self.done, self.niters)
    rwa.start.send()
    vms = []
    for i in xrange(self.nservers):
      hostname = "pc%d" % (i+27) if os.getenv("IS_CSIF") else "localhost"
      vms.append(PySy.createVM(hostname))
    for i in xrange(self.nreaders):
      r = PySy.createInstance("reader.Reader", rwa.startReadOp, \
        rwa.endReadOp, self.niters, vm=vms[rand.randint(0, len(vms)-1)])
      r.start.send()
    for i in xrange(self.nwriters):
      w = PySy.createInstance("writer.Writer", rwa.startReadOp, \
        rwa.endReadOp, self.niters, vm=vms[rand.randint(0, len(vms)-1)])
      w.start.send()


    self.done.receive()
```

Figure E.1: PySy's Readers/Writers driver program.

```
class RWAllocator(PSObject):
  def __init__(self, nclients, done, niters):
    self.startReadOp = InniOp.create()
    self.endReadOp = InniOp.create()
    self.startWriteOp = InniOp.create()
    self.endWriteOp = InniOp.create()
    self.done = done
    self.nclients = nclients
    self.nr = 0
    self.nw = 0
    self.niters = niters
  @OpMethod
  def start(self, inv):
    rwSem = InniOp.createSem(1)
    @ArmCode
    def SR(inv):
      uid = inv.getParameter(0)
      naccess = inv.getParameter(1)
      with rwSem:
        self.nr += 1
    @SuchThat
    def SR_ST(inv):
      with rwSem:
        return self.nw == 0
    @ArmCode
    def ER(inv):
      uid = inv.getParameter(0)
      naccess = inv.getParameter(1)
      with rwSem:
        self.nr -= 1
    @ArmCode
    def SW(inv):
      uid = inv.getParameter(0)
      naccess = inv.getParameter(1)
      with rwSem:
        self.nw += 1
    @SuchThat
    def SW_ST(inv):
      with rwSem:
        return self.nr == 0 and self.nw == 0
    @ArmCode
    def EW(inv):
      uid = inv.getParameter(0)
      naccess = inv.getParameter(1)
      with rwSem:
        self.nw -= 1
    inniArm_sr = InniArm(self.startReadOp, SR, st=SR_ST)
    inniArm_er = InniArm(self.endReadOp, ER)
    inniArm_sw = InniArm(self.startWriteOp, SW, st=SW_ST)
    inniArm_ew = InniArm(self.endWriteOp, EW)
    inni1 = Inni(inniArm_sr, inniArm_er, inniArm_sw, inniArm_ew)
    for i in xrange(self.niters * self.nclients * 2):
      inni1.service()
    self.done.send()
```

Figure E.2: PySy's Readers/Writers resource allocation object.

```
class Writer(PSObject):
  id = 0
  idLock = threading.Condition()

  def __init__(self, startWriteOp, endWriteOp, niters):
    with Writer.idLock:
      self.uid = Writer.id
      Writer.id += 1
    self.niters = niters
    self.startWriteOp = startWriteOp
    self.endWriteOp = endWriteOp

  @OpMethod
  def start(self, inv):
    rand = random.Random()
    for i in xrange(self.niters):
      self.startWriteOp.call(Invocation(self.uid, i))
      self.endWriteOp.call(Invocation(self.uid, i))
```

Figure E.3: PySy's Readers/Writers writer object.

```
class Reader(PSObject):
  id = 0
  idLock = threading.Condition()

  def __init__(self, startReadOp, endReadOp, niters):
    with Reader.idLock:
      self.uid = Reader.id
      Reader.id += 1
    self.niters = niters
    self.startReadOp = startReadOp
    self.endReadOp = endReadOp

  @OpMethod
  def start(self, inv):
    rand = random.Random()
    for i in xrange(self.niters):
      self.startReadOp.call(Invocation(self.uid, i))
      self.endReadOp.call(Invocation(self.uid, i))
```

Figure E.4: PySy's Readers/Writers reader object.

```
def runserver():
  manager = make_server_manager()
  shared_result_q = manager.get_result_q()
  filename = 'rwClient.py'
  serverHost, serverPort = manager.address
  nreaders = RWController.NREADERS / RWController.NSERVERS
  nwriters = RWController.NWRITERS / RWController.NSERVERS
  for i in xrange(RWController.NSERVERS):
    if i == RWController.NSERVERS - 1:
      if RWController.NREADERS % RWController.NSERVERS != 0  \
            and RWController.NREADERS > RWController.NSERVERS:
        nreaders += RWController.NREADERS % RWController.NSERVERS
      if RWController.NWRITERS % RWController.NSERVERS != 0 \
            and  RWController.NWRITERS > RWController.NSERVERS:
        nwriters += RWController.NWRITERS % RWController.NSERVERS
    hostname = "pc%d" % (i+27) if os.getenv("IS_CSIF") else "localhost"
    cmd = 'ssh %s python %s %s %d %d %d %d' % \
            (hostname, filename, serverHost, serverPort, nreaders, \
            nwriters, RWController.NITERS)
    client = subprocess.Popen(cmd.split(), shell=False, stdout=None,\
            stderr=None)
  numresults = 0
  while numresults < RWController.NSERVERS:
    shared_result_q.get()
    numresults += 1
  manager.shutdown()
def make_server_manager():
  class RWManager(SyncManager): pass
  result_q = Queue()
  def startRead(*args):
    t = threading.Thread(target=RWAllocator.startRead, args=args)
    t.start()
  def endRead(*args):
    t = threading.Thread(target=RWAllocator.endRead, args=args)
    t.start()
  def startWrite(*args):
    t = threading.Thread(target=RWAllocator.startWrite, args=args)
    t.start()
  def endWrite(*args):
    t = threading.Thread(target=RWAllocator.endWrite, args=args)
    t.start()
  RWManager.register('get_result_q', callable=lambda: result_q)
  RWManager.register('startRead', \
              callable=lambda *args: startRead(*args))
  RWManager.register('endRead', callable=lambda *args: endRead(*args))
  RWManager.register('startWrite', \
              callable=lambda *args: startWrite(*args))
  RWManager.register('endWrite', callable=lambda *args: endWrite(*args))
  manager = RWManager(address=(socket.gethostname(), 0), authkey='abc')
  manager.start()
  return manager
```

Figure E.5: multiprocessing's Readers/Writers driver program (part 1 of 2).

```
class RWController(object):
  NSERVERS = 1
  NREADERS = 15
  NWRITERS = 10
  NITERS = 100
  def __init__(self, performance, nreaders=15, nwriters=10, \
                 nservers=1, niters=100):
    self.performance = performance
    RWController.NSERVERS = nservers
    RWController.NREADERS = nreaders
    RWController.NWRITERS = nwriters
    RWController.NITERS = niters
    self.test()

  def test(self):
    self.performance.run([Work(self.work)])

  @timing
  def work(self,*args):
    runserver()
```

Figure E.6: multiprocessing's Readers/Writers driver program (part 2 of 2).

```
from multiprocessing import Condition, Lock
class RWAllocator(object):
  sync = Condition(lock=Lock())
  nr = 0
  nw = 0
  @staticmethod
  def startRead( uid, naccess):
    with RWAllocator.sync:
      while RWAllocator.nw > 0:
        RWAllocator.sync.wait()
      RWAllocator.nr += 1
  @staticmethod
  def endRead( uid, naccess):
    with RWAllocator.sync:
      RWAllocator.nr -= 1
      RWAllocator.sync.notify_all()
  @staticmethod
  def startWrite(uid, naccess):
    with RWAllocator.sync:
      while RWAllocator.nw > 0 or RWAllocator.nr > 0:
        RWAllocator.sync.wait()
      RWAllocator.nw += 1
  @staticmethod
  def endWrite(uid, naccess):
    with RWAllocator.sync:
      RWAllocator.nw -= 1
      RWAllocator.sync.notify_all()
```

Figure E.7: multiprocessing's Readers/Writers resource allocation object.

```
class Writer(Process):
    id = 0
    idLock = Condition()
    def __init__(self, rwa, niters):
        Process.__init__(self)
        with Writer.idLock:
            self.uid = Writer.id
            Writer.id += 1
        self.rwa = rwa
        self.niters = niters

    def run(self):
        rand = random.Random()
        for i in xrange(self.niters):
            self.rwa.startWrite(self.uid, i)
            self.rwa.endWrite(self.uid, i)
class Reader(Process):
    id = 0
    idLock = Condition()
    def __init__(self, rwa, niters):
        Process.__init__(self)
        with Reader.idLock:
            self.uid = Reader.id
            Reader.id += 1
        self.rwa = rwa
        self.niters = niters
    def run(self):
        rand = random.Random()
        for i in xrange(self.niters):
            self.rwa.startRead(self.uid, i)
            self.rwa.endRead(self.uid, i)
```

Figure E.8: multiprocessing's Readers/Writers client source (part 1 of 2).

```
def doRW(mgr, nreaders, nwriters, niters):
    workers = []
    for i in xrange(nreaders):
        r = Reader(mgr, niters)
        r.start()
        workers.append(r)
    for i in xrange(nwriters):
        w = Writer(mgr, niters)
        w.start()
        workers.append(w)
    for w in workers:
        w.join()
    q = mgr.get_result_q()
    q.put("done")

def runclient(host, port, nreaders, nwriters, niters):
    manager = make_client_manager(host, port, 'abc')
    doRW(manager, nreaders, nwriters, niters)

def make_client_manager(ip, port, authkey):
    class RWManager(SyncManager): pass
    RWManager.register('startRead')
    RWManager.register('endRead')
    RWManager.register('startWrite')
    RWManager.register('endWrite')
    RWManager.register('get_result_q')
    manager = RWManager(address=(ip, port), authkey=authkey)
    manager.connect()
    return manager

def main(host, port, nreaders, nwriters, niters):
    runclient(host, port, nreaders, nwriters, niters)

if __name__ == "__main__":
    if len(sys.argv) < 6:
        print "not enough args: %s" % args
        sys.exit()
    main(sys.argv[1],int(sys.argv[2]), int(sys.argv[3]), \
            int(sys.argv[4]), int(sys.argv[5]))
```

Figure E.9: multiprocessing's Readers/Writers client source (part 2 of 2).

```
class RWAllocator(object):
  def __init__(self, vr):
    self.nr = 0
    self.nw = 0
    self.vr = vr
    self.lock = threading.Condition(lock=threading.Lock())

  def startRead(self, src, uid, naccess):
    with self.lock:
      while self.nw != 0:
        self.lock.wait()
      try:
        assert self.nw == 0 and self.nr >= 0
        self.nr += 1
      except AssertionError:
        raise AssertionError

    self.vr.send(dest=src, uid=uid, func="startRead")

  def endRead(self, src, uid, naccess):
    with self.lock:
      try:
        assert self.nr >= 0
        self.nr -= 1
      except AssertionError:
        raise AssertionError
      self.lock.notifyAll()
    self.vr.send(dest=src, uid=uid, func="endRead")

  def startWrite(self, src, uid, naccess):
    with self.lock:
      while self.nw != 0 or self.nr != 0:
        self.lock.wait()
      try:
        assert self.nw == 0
        self.nw += 1
      except AssertionError:
        raise AssertionError

    self.vr.send(dest=src, uid=uid, func="startWrite")

  def endWrite(self, src, uid, naccess):
    with self.lock:
      try:
        assert self.nw == 1
        self.nw -= 1
      except AssertionError:
        raise AssertionError
      self.lock.notifyAll()
    self.vr.send(dest=src, uid=uid, func="endWrite")
```

Figure E.10: River implementation of the Readers/Writers program (part 1 of 3).

```
class Reader(object):
  def __init__(self, uid):
    self.uid = uid

class Writer(object):
  def __init__(self, uid):
    self.uid = uid

class RWMain(VirtualResource):
  def vr_init(self):
    discovered = self.discover()
    allocated = self.allocate(discovered)
    deployed = self.deploy(allocated, module=self.__module__, \
      func='work')
    self.vms = [vm['uuid'] for vm in deployed]
    return True

  def main(self):
    if len(sys.argv) != 6:
      print usage

    NTRIALS = int(sys.argv[2])
    NREADERS = int(sys.argv[3])
    NWRITERS = int(sys.argv[4])
    NITERS = int(sys.argv[5])

    assert NWRITERS + NREADERS + 1 <= len(self.vms)
    allocator = self.vms[0]
    self.vms = self.vms[1:NWRITERS+NREADERS+1]

    self.send(dest=allocator, init="init", NTRIALS=NTRIALS,uid=0)
    [self.send(dest=vm, init="init", NTRIALS=NTRIALS,uid=i+1) \
      for i, vm in enumerate(self.vms)]
    for i in xrange(NTRIALS):
      self.send(dest=allocator, start="start", NREADERS=NREADERS, \
        NWRITERS=NWRITERS, NITERS=NITERS)

      print "(RW %d %d %d)" % (NREADERS, NWRITERS, NITERS)
      #start up r/w
      for i in xrange(NREADERS):
        self.send(dest=self.vms[i], start="start", type="reader",\
         NTRIALS=NTRIALS, NITERS=NITERS, allocator=allocator, \
        uid=i+1)
      for i in xrange(NWRITERS):
        self.send(dest=self.vms[NREADERS+i], start="start", \
        type="writer", NITERS=NITERS, allocator=allocator, \
        uid=(NREADERS + i+1))

      for i in xrange(NREADERS + NWRITERS + 1):
        self.recv(src=self.ANY, done="done")
```

Figure E.11: River implementation of the Readers/Writers program (part 2 of 3).

```
def work(self):
  m = self.recv(src=self.parent, init="init", uid=self.ANY)
  NTRIALS = m.NTRIALS
  uid = m.uid
  for i in xrange(NTRIALS):
    if uid == 0:
      #this vr is specifically for the allocator
      m = self.recv(start="start")
      NREADERS = m.NREADERS
      NWRITERS = m.NWRITERS
      NITERS = m.NITERS
      rwa = RWAllocator(self)
      nmessages = (NREADERS + NWRITERS) * NITERS * 2
      threadList = []
      for i in xrange(nmessages):
        m = self.recv(src=self.ANY, func=self.ANY)
        method = getattr(rwa, m.func)
        t = threading.Thread(target=method, args=(m.src, m.uid, \
          m.naccess))
        t.daemon = True
        threadList.append(t)
        t.start()

      [t.join() for t in threadList]
    else:
      m = self.recv(start="start")
      NITERS = m.NITERS
      allocator = m.allocator
      if m.type == "reader":
        r = Reader(m.uid)
        for i in xrange(NITERS):
          self.send(dest=allocator, uid=r.uid, naccess = i,\
            func="startRead")
          self.recv(src=allocator, uid=r.uid, func="startRead")

          self.send(dest=allocator, uid=r.uid, naccess = i,\
            func="endRead")
          self.recv(src=allocator, uid=r.uid, func="endRead")

      else:
        w = Writer(uid)
        for i in xrange(NITERS):
          self.send(dest=allocator, uid=w.uid, naccess = i,\
            func="startWrite")
          self.recv(src=allocator, uid=w.uid, func="startWrite")

          self.send(dest=allocator, uid=w.uid, naccess = i, \
            func="endWrite")
          self.recv(src=allocator, uid=w.uid, func="endWrite")
    self.send(dest=self.parent, done="done")
```

Figure E.12: River implementation of the Readers/Writers program (part 3 of 3).

Appendix F

Fast Fourier Transform Examples

Figures F.1 and F.2 show an implementation of a distributed FFT algorithm using PySy. Figures F.3 and F.4 show an implementation of a distributed FFT algorithm using **multiprocessing**. Figures F.5 and F.6 show an implementation of a distributed FFT algorithm using the River extension Trickle.

```
class FFT(PSObject):
  ARRAY_ROWS = 10000
  NSERVERS = 2
  def __init__(self, performance, ARRAY_ROWS, NSERVERS):
    self.performance = performance
    FFT.ARRAY_ROWS = int(ARRAY_ROWS)
    FFT.NSERVERS = int(NSERVERS)
    self.vms = None
    self.done = InniOp.create()
    self.work()

  def work(self,*args):
    omega = math.pi
    servers = []
    currRow = 1
    strip = FFT.ARRAY_ROWS / FFT.NSERVERS
    for i in xrange(FFT.NSERVERS):
      beginRow = currRow
      endRow = currRow + strip if i != (FFT.NSERVERS -1) \
        else FFT.ARRAY_ROWS
      currRow = endRow
      vm = PySy.createVM("pc%d" % (i+27)) if os.getenv("IS_CSIF") \
          else PySy.createVM()
      srvr = PySy.createInstance("server.Server",beginRow, endRow,\
              vm=vm)
      srvr.start.send()
      servers.append(srvr)
      self.vms.append(vm)
      #print beginRow, endRow
    result = []
    self.TestArray = [[0],[0]]
    self.TestArray[0][0] = servers[0].TIIntegrate.call(0.0, 2.0, \
              1000, 0.0, 0, 0, 0) / 2.0

    for server in servers:
      stripResults = server.results.receive()
      self.TestArray[0].extend(stripResults[0])
      self.TestArray[1].extend(stripResults[1])
```

Figure F.1: PySy's FFT driver program.

```
class Server(PSObject):
  def __init__(self, beginRow, endRow):
    self.beginRow = beginRow
    self.endRow = endRow
    self.TestArray = [[0]*((endRow-beginRow)) \
      for x in xrange(0,2)]
    self.results = InniOp.create()

  @OpMethod
  def TIIntegrate(self, inv):
    x0,x1,nsteps,omega,select,row,col = inv.getParameters()
    return self.TrapezoidIntegrate(0.0, 2.0, 1000, omega*row, \
            select)

  def TrapezoidIntegrate(self, x0, x1, nsteps, omegan, select):
    x = x0
    dx = (x1-x0) / float(nsteps)
    rvalue = self.thefunction(x0, omegan, select) / 2.0
    if nsteps != 1:
      nsteps -= 1
      while nsteps > 1:
        x += dx
        rvalue += self.thefunction(x, omegan, select)
        nsteps -= 1
    return (rvalue + self.thefunction(x1, omegan, select) \
         / 2.0) * dx

  def thefunction(self, x, omegan, select):
    if select == 0:
      return (x+1)**x
    elif select == 1:
      return ((x+1)**x) * math.cos(omegan*x)
    else:
       return ((x+1)**x) * math.sin(omegan*x)

  @OpMethod
  def start(self, inv):
    omega = math.pi
    for i in xrange(0, self.endRow - self.beginRow):
      self.TestArray[0][i] = self.TIIntegrate.call(0.0,\
         2.0, 1000, omega, 1, i+self.beginRow, 0)
      self.TestArray[1][i] = self.TIIntegrate.call(0.0,\
         2.0, 1000, omega, 2, i+self.beginRow, 1)
    self.results.send(self.TestArray)
```

Figure F.2: PySy's FFT server object.

```
def runserver():
  manager = make_server_manager()
  shared_job_q = manager.get_job_q()
  shared_result_q = manager.get_result_q()
  class JobQueueManager(SyncManager): pass
  JobQueueManager.register('get_job_q', callable=lambda: job_q)
  JobQueueManager.register('get_result_q', \
    callable=lambda: result_q)
  currRow = 0
  strip = FFT.ARRAY_ROWS / FFT.NSERVERS
  for i in xrange(FFT.NSERVERS):
    beginRow = currRow
    endRow = currRow + strip if i != (FFT.NSERVERS -1) \
       else FFT.ARRAY_ROWS
    currRow = endRow
    shared_job_q.put((beginRow, endRow))
  serverHost, serverPort = manager.address
  for i in xrange(FFT.NSERVERS):
    hostname = "pc%d" % (i+27) if os.getenv("IS_CSIF") \
      else "localhost"
    cmd = 'ssh %s python -u %s %s %s' % (hostname, filename,\
      serverHost, serverPort)
    client = subprocess.Popen(cmd.split(), shell=False, \
      stdout=None, stderr=None)
  numresults = 0
  results = []
  while numresults < FFT.NSERVERS:
    results.append(shared_result_q.get())
    numresults += 1
  results.sort()
  TestArray = []
  for _, result in results:
    TestArray.extend(result)
  TestArray[0][0] /= 2
  manager.shutdown()
def make_server_manager():
  job_q = Queue()
  result_q = Queue()
  class JobQueueManager(SyncManager): pass
  JobQueueManager.register('get_job_q', callable=lambda: job_q)
  JobQueueManager.register('get_result_q', \
    callable=lambda: result_q)
  hostname = "pc14" if os.getenv("IS_CSIF") else "localhost"
  manager = JobQueueManager(address=(hostname, 5025), authkey='abc')
  manager.start()
  return manager
class FFT(object):
  ARRAY_ROWS = 10000
  NSERVERS = 2
runserver()
```

Figure F.3: multiprocessing's FFT server.

```
from multiprocessing import Process, Array, Queue
from multiprocessing.managers import SyncManager
def compute(jobq, resultq):
    beginRow, endRow = jobq.get()
    omega = math.pi
    TestArray = [[0]*(endRow-beginRow) for x in xrange(0,2)]
    for i in xrange(beginRow, endRow):
        TIIntegrate(0.0, 2.0, 1000, omega, 1, i, 0, \
      beginRow, TestArray)
        TIIntegrate(0.0, 2.0, 1000, omega, 2, i, 1, \
      beginRow, TestArray)
    resultq.put((beginRow,TestArray))

def TIIntegrate(x0, x1, nsteps, omega, select, row, col,\
     beginRow, TestArray):
        TestArray[col][row-beginRow] = TrapezoidIntegrate(0.0, \
          2.0, 1000, omega*row,select)
def TrapezoidIntegrate(x0, x1, nsteps, omegan, select):
    x = x0
    dx = (x1-x0) / float(nsteps)
    rvalue = thefunction(x0, omegan, select) / 2.0
    if nsteps != 1:
        nsteps -= 1
        while nsteps > 1:
            x += dx
            rvalue += thefunction(x, omegan, select)
            nsteps -= 1
    rvalue = (rvalue + thefunction(x1, omegan, select) / 2.0) * dx
    return rvalue
def thefunction(x, omegan, select):
    if select == 0:
        return (x+1)**x
    elif select == 1:
        return ((x+1)**x) * math.cos(omegan*x)
    else:
         return ((x+1)**x) * math.sin(omegan*x)
def runclient(host, port):
    manager = make_client_manager(host, port, 'abc')
    job_q = manager.get_job_q()
    result_q = manager.get_result_q()
    compute(job_q, result_q)
def make_client_manager(ip, port, authkey):
    class ServerQueueManager(SyncManager):
        pass
    ServerQueueManager.register('get_job_q')
    ServerQueueManager.register('get_result_q')
    manager = ServerQueueManager(address=(ip,port),authkey=authkey)
    manager.connect()
    return manager
def main(host, port):
    runclient(host, port)
if __name__ == "__main__":
    main(sys.argv[1],int(sys.argv[2]))
```

Figure F.4: multiprocessing's FFT client.

```
def TIIntegrate(x0, x1, nsteps, omega, select, row, col):
    return TrapezoidIntegrate(0.0, 2.0, 1000, omega*row, select)
def TrapezoidIntegrate(x0, x1, nsteps, omegan, select):
  x = x0
  dx = (x1-x0) / float(nsteps)
  rvalue = thefunction(x0, omegan, select) / 2.0
  if nsteps != 1:
    nsteps -= 1
    while nsteps > 1:
      x += dx
      rvalue += thefunction(x, omegan, select)
      nsteps -= 1
  rvalue = (rvalue + thefunction(x1, omegan, select) / 2.0) * dx

  return rvalue
def thefunction( x, omegan, select):
  import math
  if select == 0:
    return (x+1)**x
  elif select == 1:
    return ((x+1)**x) * math.cos(omegan*x)
  else:
     return ((x+1)**x) * math.sin(omegan*x)
def start(beginRow, endRow):
  import math
  omega = math.pi
  TestArray = [[0]*((endRow-beginRow)) for x in xrange(0,2)]
  for i in xrange(0, endRow - beginRow):
    TestArray[0][i] = TIIntegrate(0.0, 2.0, 1000, omega, 1, \
      i+beginRow, 0)
    TestArray[1][i] = TIIntegrate(0.0, 2.0, 1000, omega, 2, \
      i+beginRow, 1)
  return TestArray[0], TestArray[1]
```

Figure F.5: An implementation of FFT using the River extension Trickle (part 1 of 2).

```
if len(sys.argv) != 4:
  print usage

NTRIALS = int(sys.argv[1])
ARRAY_ROWS = int(sys.argv[2])
NSERVERS = int(sys.argv[3])

print "(FFT %d %d %d)" % (NTRIALS, ARRAY_ROWS, NSERVERS)

omega = math.pi
for i in xrange(NTRIALS):
  begin = time.time()
  vrlist = connect(NSERVERS)
  inject(vrlist, TIIntegrate)
  inject(vrlist, TrapezoidIntegrate)
  inject(vrlist, thefunction)
  inject(vrlist, start)

  hlist = []
  strip = ARRAY_ROWS / NSERVERS
  currRow = 1
  for i, vr in enumerate(vrlist):
    beginRow = currRow
    endRow = currRow + strip if i != (NSERVERS -1) else ARRAY_ROWS
    currRow = endRow
    hlist.append(fork(vr, start, beginRow, endRow))

  tempResults = join(hlist)
  zeroth = fork(vrlist[0], TIIntegrate, 0.0, 2.0, 1000, 0.0, 0, 0, 0)
  zerothRes = join(zeroth) / 2.0
  results = []
  results.append(tuple([zerothRes]) + tuple(tempResults[0][0]))
  results.append(tuple([0.0]) + tuple(tempResults[0][1]))
  end = time.time()
  print (end - begin)
```

Figure F.6: An implementation of FFT using the River extension Trickle (part 2 of 2).

Appendix G

Matrix Multiplication Examples

Figures G.1 and G.2 show the implementation of a distributed NxN matrix multiplication algorithm using PySy. Figures G.3, G.4, and G.5 show an implementation of a distributed NxN matrix multiplication algorithm using **multiprocessing**. Figure G.6 shows an implementation of a distributed NxN matrix multiplication algorithm using the River extension Trickle.

```
r = random.Random()
class MatrixMult(PSObject):
    MAX_COEF = 50
    N = 1000
    NSERVERS = 5
    def __init__(self, performance, N, NSERVERS):
        self.performance = performance
        MatrixMult.N = int(N)
        MatrixMult.NSERVERS = int(NSERVERS)
        self.A = self.genMatrix(MatrixMult.N)
        self.B = self.genMatrix(MatrixMult.N)
        self.vms = None
        self.work()

    def genMatrix(self, n, isZero=False):
        result = []
        for i in xrange(n):
            result.append([])
            for j in xrange(n):
                val = 0 if isZero else r.randint(0, \
      MatrixMult.MAX_COEF)
                result[i].append(val)
        return result

    def work(self,*args):
        currRow = 0
        servers = []
        strip = MatrixMult.N / MatrixMult.NSERVERS
        for i in xrange(MatrixMult.NSERVERS):
            beginRow = currRow
            endRow = currRow + strip if i != (MatrixMult.NSERVERS -1) \
    else MatrixMult.N
            currRow = endRow
            vm = PySy.createVM("pc%d" % (i+27)) if os.getenv("IS_CSIF") \
      else PySy.createVM()
            srvr = PySy.createInstance("server.Server",self.A, self.B, \
        beginRow, endRow, MatrixMult.N, vm=vm)
            srvr.start.send()
            servers.append(srvr)
            self.vms.append(vm)
        result = []
        for server in servers:
            stripResults = server.results.receive()
            result.extend(stripResults)
```

Figure G.1: PySy's Matrix Multiplication driver program.

```
class Server(PSObject):
    def __init__(self, A,B, beginRow, endRow, ncols):
        self.A = A
        self.B = B
        self.C = [[0]*ncols for i in xrange(beginRow, endRow)]
        self.beginRow = beginRow
        self.endRow = endRow
        self.ncols =ncols
        self.results = InniOp.create()

    @OpMethod
    def start(self, inv):
        for r in xrange(self.beginRow, self.endRow):
            for c in xrange(self.ncols):
                self.compute(r,c)

        self.results.send(self.C)

    def compute(self, r,c ):
        #r,c = inv.getParameters()
        for k in xrange(self.ncols):
            self.C[r-self.beginRow][c] += self.A[r][k] * \
                    self.B[k][c]
```

Figure G.2: PySy's Matrix Multiplication server object.

```
r = random.Random()
def runserver(A, B):
  # Start a shared manager server and access its queues
  manager = make_server_manager()
  shared_job_q = manager.get_job_q()
  shared_result_q = manager.get_result_q()

  class JobQueueManager(SyncManager):
    pass
  JobQueueManager.register('get_job_q', callable=lambda: job_q)
  JobQueueManager.register('get_result_q', callable=lambda: result_q)

  currRow = 0
  servers = []
  strip = MatrixMult.N / MatrixMult.NSERVERS

  for i in xrange(MatrixMult.NSERVERS):
    beginRow = currRow
    endRow = currRow + strip if i != (MatrixMult.NSERVERS -1) \
           else MatrixMult.N
    currRow = endRow
    shared_job_q.put((beginRow, endRow, MatrixMult.N, A, B))
  result = []
  #start clients
  sshPort = 22
  username = 'dev'
  filename = 'mmClient.py'
  serverHost, serverPort = manager.address
  activeSubprocesses = []
  for i in xrange(MatrixMult.NSERVERS):
    hostname = "pc%d" % (i+27) if os.getenv("IS_CSIF") else "localhost"
    username = "twilliam" if os.getenv("IS_CSIF") else "dev"
    cmd = 'ssh -p %d %s@%s python %s %s %s' % (sshPort, username, \
      hostname, filename, serverHost, serverPort)
    client = subprocess.Popen(cmd.split(), shell=False, \
      stdout=None, stderr=None)
    activeSubprocesses.append(client)

  numresults = 0
  results = []
  while numresults < MatrixMult.NSERVERS:
    results.append(shared_result_q.get())
    numresults += 1

  results.sort()
  C = []
  for _, result in results:
    C.extend(result)
  for client in activeSubprocesses:
    del client
  manager.shutdown()
```

Figure G.3: multiprocessing's Matrix Multiplication driver program (part 1 of 2).

```
def make_server_manager():
  job_q = Queue()
  result_q = Queue()

  class JobQueueManager(SyncManager):
    pass

  JobQueueManager.register('get_job_q', callable=lambda: job_q)
  JobQueueManager.register('get_result_q', callable=lambda: result_q)
  hostname = "pc14" if os.getenv("IS_CSIF") else "localhost"

  manager = JobQueueManager(address=(hostname, 5025), authkey='abc')
  manager.start()
  return manager


class MatrixMult(object):
  N = 10000
  MAX_COEF = 50
  NSERVERS = 2
  def __init__(self, performance, N=10000, NSERVERS=2):
    self.performance = performance
    MatrixMult.N= N
    MatrixMult.NSERVERS = NSERVERS
    self.A = self.genMatrix(MatrixMult.N)
    self.B = self.genMatrix(MatrixMult.N)

    self.test()
  def test(self):
    self.performance.run([Work(self.work)], setup=(self.setup, ()))


  def setup(self, *args):
    #self.TestArray = [[0]*FFT.ARRAY_ROWS for x in xrange(0,2)]
    self.TestArray = []

  def genMatrix(self, n):
    global r
    result = []
    for i in xrange(n):
      result.append([])
      for j in xrange(n):
        result[i].append(r.randint(0, MatrixMult.MAX_COEF))
    return result
  @timing
  def work(self,*args):
    runserver(self.A, self.B)
```

Figure G.4: multiprocessing's Matrix Multiplication driver program (part 2 of 2).

```
def compute(jobq, resultq):
    beginRow, endRow, ncols, A, B = jobq.get()
    C =  [[0]*(endRow-beginRow) for x in xrange(ncols)]
    for row in xrange(beginRow,endRow):
  for col in xrange(ncols):
      for k in xrange(ncols):
    C[col][row-beginRow] += A[row][k] * B[k][col]


    resultq.put((beginRow, C))

def runclient(host, port):
    manager = make_client_manager(host, port, 'abc')
    job_q = manager.get_job_q()
    result_q = manager.get_result_q()
    compute(job_q, result_q)

def make_client_manager(ip, port, authkey):
    class ServerQueueManager(SyncManager):
  pass

    ServerQueueManager.register('get_job_q')
    ServerQueueManager.register('get_result_q')

    manager = ServerQueueManager(address=(ip, port), \
    authkey=authkey)
    manager.connect()

    return manager


def main(host, port):
    runclient(host, port)



if __name__ == "__main__":
    if len(sys.argv) < 3:
  print "not enough args: %s" % args
  sys.exit()
    main(sys.argv[1],int(sys.argv[2]))
```

Figure G.5: multiprocessing's Matrix Multiplication client.

```
def genMatrix(n, isZero=False):
  result = []
  r = random.Random(12345)

  for i in xrange(n):
    result.append([])
    for j in xrange(n):
      val = 0 if isZero else r.randint(0, 50)
      result[i].append(val)
  return result

def start(beginRow, endRow, ncols, A, B):
  C = [[0]*ncols for i in xrange(beginRow, endRow)]
  for r in xrange(beginRow, endRow):
    for c in xrange(ncols):
      for k in xrange(ncols):
        C[r-beginRow][c] += A[r][k] * B[k][c]
  return C


if len(sys.argv) != 4:
  print usage


NTRIALS = int(sys.argv[1])
N = int(sys.argv[2])
NSERVERS = int(sys.argv[3])

print "(MM %d %d %d)" % (NTRIALS, N, NSERVERS)

for i in xrange(NTRIALS):
  A = genMatrix(N)
  B = genMatrix(N)

  begin = time.time()
  vrlist = connect(NSERVERS)
  inject(vrlist, start)

  hlist = []
  strip = N / NSERVERS
  currRow = 0
  for i, vr in enumerate(vrlist):
    beginRow = currRow
    endRow = currRow + strip if i != (NSERVERS -1) else N
    currRow = endRow
    hlist.append(fork(vr, start, beginRow, endRow, N, A, B))

  results = join(hlist)
  end = time.time()

  print (end - begin)
```

Figure G.6: Trickle's Matrix Multiplication implementation.

Bibliography

[1] *Linux Man Pages*. http://linux.die.net/man/2/write.

[2] *MPI website*. http://www.mcs.anl.gov/research/projects/mpi/.

[3] Gregory R. Andrews, Michael Coffin, Irving Elshoff, Kelvin Nilson, Gregg Townsend, Ronald A. Olsson, and Titus Purdin. An overview of the SR language and implementation. *ACM Trans. Program. Lang. Syst.*, 10:51–86, January 1988.

[4] Gregory R. Andrews and Ronald A. Olsson. *The SR Programming Language: Concurrency in Practice*. Benjamin/Cummings Pub. Co., 1993.

[5] David M. Beazley. *Python Essential Reference*. Addison Wesley, 4th edition, 2009.

[6] Hiu Ning Chan, Esteban Pauli, Billy Yan-Kit Man, Aaron W. Keen, and Ronald A. Olsson. An exception handling mechanism for the concurrent invocation statement. *Euro-Par 2005 Parallel Processing*, pages 699–709, August 2005.

[7] P.J. Courtois, F. Heymans, and D.L. Parnas. Concurrent control with "readers" and "writers". *Communications of the ACM*, pages 667–668, October 1971.

[8] Edsger W. Dijkstra. The superfluity of the general semaphore. EWD 734, Neunen, The Netherlands, April 1980.

[9] Alexey S. Fedosov. A Python-based framework for distributed programming and rapid prototyping of distributed programming models. Master's thesis, University of San Francisco, 2009.

[10] A.S. Fedosov and G.D. Benson. Communication with super flexible messaging. pages 379–388. 2007 International Conference on Parallel and Distributed Processing Techniques, August 2007.

[11] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[12] *Java Grande Forum Benchmark website*. http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html.

[13] Aaron W. Keen, Tingjian Ge, Justin T. Maris, and Ronald A. Olsson. JR: Flexible distributed programming in an extended Java. *ACM Transactions on Programming Languages and Systems*, pages 575–584, May 2004.

[14] Friedemann Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2:161–175, 1987.

[15] Friedemann Mattern. Asynchronous distributed termination – parallel and symmetric solutions with echo algorithms. *Algorithmica*, 5:325–340, 1990.

[16] Ronald A. Olsson, Gregory D. Benson, Tingjian Ge, and Aaron W. Keen. Fairness in shared invocation servicing. *Computer Languages, Systems & Structures*, pages 327–351, December 2002.

[17] Ronald A. Olsson and Aaron W. Keen. *The JR Programming Language: Concurrent Programming in an Extended Java*. Kluwer International series in engineering and computer science; SECS 774. Boston : Kluwer Academic, 2004.

[18] *Python Celery package website*. http://celeryproject.org/.

[19] *Python Class Decorators PEP 3129*. http://www.python.org/dev/peps/pep-3129/.

[20] *Python Function Decorators PEP 318*. http://www.python.org/dev/peps/pep-0318/.

[21] *Python mpi4py package website*. http://mpi4py.scipy.org/.

[22] *Python Online Documentation*. http://docs.python.org/.

[23] *Python PyCSP package website*. http://code.google.com/p/pycsp/.

[24] *Python pypar package website*. http://code.google.com/p/pypar/.

[25] *Python Remote Objects (Pyro)*. http://irmen.home.xs4all.nl/pyro3/.

[26] *Python River framework website*. http://river.cs.usfca.edu/.

[27] *The JR Programming Language website*. http://www.cs.ucdavis.edu/~olsson/research/jr/.

[28] *The SR Programming Language website*. http://www.cs.arizona.edu/sr/.