

Concurrent

Easy to use python computing environment

Wundke, Moritz

Guim Bernat, Francesc

Universitat oberta de Catalunya

Table of Contents

Introduction.....	1
Python.....	1
Framework.....	1
Motivation.....	2
Objectives.....	3
Related work.....	4
Event based frameworks.....	4
Simple processing frameworks.....	4
Volunteer Computing.....	4
The evolution.....	5
Current parallel processing libraries for python.....	5
dispy.....	5
ParallelPython.....	5
IPython.....	6
Superpy.....	6
Planning.....	7
Methodology.....	7
Project Plan.....	8
XP Phases.....	8
User Stories.....	8
Cost, Scope and Iterations.....	10
Architecture.....	12
Overview.....	12
Components.....	12
Nodes.....	13
MasterNode.....	14
SlaveNode.....	14
ApplicationNode.....	15
Configs.....	15
Transport concerns.....	16
RESTful.....	16
TCP Sockets.....	16
ZeroMQ.....	16
RPC implementation.....	17
Work Distribution.....	21
Sending work versus polling work.....	21
Sending work execution flow.....	22
Polling work execution flow (round-robin).....	23
Simple tasks and batches.....	24
Security concerns.....	24
Monitoring Nodes and Performance.....	24
Next Steps.....	25
Samples.....	26
MD5 Hash Reverse.....	26
Setup.....	26
The application.....	27
Benchmark.....	31

The Application.....	31
ITaskSystem vs plain tasks.....	36
Optimizing number of workers.....	37
Mandlebrot.....	38
The application.....	38
ITaskSystem vs plain tasks.....	48
Net congestion and workloads.....	49
DNA Curve analysis.....	50
Net congestion and workloads.....	50
Consideration on data flow.....	50
Conclusions.....	51
Results.....	52
Appendix A.....	53
Gantt project plan.....	53
Appendix B.....	55
Project Page.....	55
API.....	55
Slides.....	55
References.....	56

Introduction

Concurrency is not a new concept in our ever changing world, but it is still a pending tasks in most scenarios. We have concurrent system running everywhere, a simple app that updates in the background, a game running in multiple threads, even some share their computer for scientific computations through [SETI@Home](#) [1].

But the important point is that all those options come along with a heavy burden, they are for experts. Of course every one can setup a grid in their home and starting to build a concurrent application using OpenMP, MPI, TBB, etc ... but the fact is that we have to reinvent the wheel again an again.

Another important aspect in those environment is the ability to prototype or to follow a rapid development process, right now we are left to use inmost cases low-level C/C++. Which gives us power but is also prone to slow development and cross-platform issues.

This research aims to fill the gap in the current high performance processing world to achieve a flexible, fast, effective, easy to learn and cross-platform framework that first into the grid computing and the multimedia world.

Python

Python seems to be a perfect candidate that gives us the basic keystones of our requirements as mentioned earlier. Python is a high-level interpreted programming language with support for both features and other languages to integrate with.

Developing in python has several gives us the flexibility we need, we do not have to compile/link our programs, they are interpreted on the fly by a virtual machine speeding up our development cycle. Being interpreted is not only agile from a developer perspective, it gives us instant cross-platform behavior. Python's philosophy of code-once executed-everywhere is a perfect match for our approach.

Python itself is already used in scientific computation, it features a wide set of packages encapsulated in the acclaimed *SciPy* stack [2]. *SciPy* includes from numeric analysis tools such as *NumPy* [3] or *Pandas* [4] to data visualization and plotting tools such as *Matplotlib* [5].

Now questions arises of why should we invest our time to create a framework if the selected language itself seems sufficient to get the job done. Python is not perfect, it features already concurrent basics but they are problematic out-of-the-box. The next point will outline why a framework is needed to leverage with the hard work so the user can focus rather on creating high-performance computations and not fighting with the systems.

Framework

Having already discussed some of the good points about Python and its extensive use in scientific computation why do we require a new framework? The answer is fairly easy, to be able to perform efficient computation using Python one has to build low-level primitives and fight with the same issues over and over again.

Python itself is an interpreted language and suffers from a huge design flaw, the interpreter is not thread-safe. This means that we can not access or execute python in parallel, this is actually prevent by the *Global Interpreter Lock* short GIL [6]. The GIL is the chosen mechanism to make Python thread-safe but degrading performance.

There are several solution to the GIL, one is to switch from the basic C implementation of Python – Cpython – and using other implementations such as Jython [7] or IronPython [8]. At first glance it seems reasonable to use a reimplementations but we have to accept compromises. Jython is considered slow (compared to Python 3, while Jython is actually faster then python 2.7), IronPython is a .NET implementation of Python and is actually fast, it's main problem is that is does only run officially on Windows and MAC through mono. Both implementations come with a other issue, they lack python's recent features, both Jython and IronPython are compliant with Python 2.7 officially.

Another way to address the GIL issues is to have those CPU intense tasks releasing the GIL explicitly. To do this safely we have to implement those tasks in low-level C code. Now this is the place where Cython [9] comes into play. Cython is an optimistic static compiler that transforms Python code into C code which then gets compiled on the target platform. Using Cython gives us the felxibility we have using the Python while begin able to address the performance and GIL issues we just outlined.

But we have another issue to solve, even with Cython we have only one interpreter available that will look pure Python code. To come along this issues instead of using threads we will use processes. Using processes will drop shared memory features and introduce other issues that have to be solved but we clearly can now say that we have solved our GIL issues.

Motivation

The motivation to start this endeavour is basically the need for a real concurrent system in Python without losing its essence for flexibility. The framework should not only be applicable to scientific computation, thus it should be seen as a basic for any application that requires performance.

Another important challenge is to build such a framework and to prove it's potential implementing a real world example. We will implement two type of example projects, one focused on the entertainment industry and another focused on scientific computation.

The third motivation is to address one and for all Python infamous GIL. The GIL makes CPU bound computation nearly unpractical in Python and therefor other solutions are used that are proven less flexible then many developer which to. Figure 1 (a) outlines a typical GIL contention using one threads while figure X (b) shows how C extension can get around this issue.

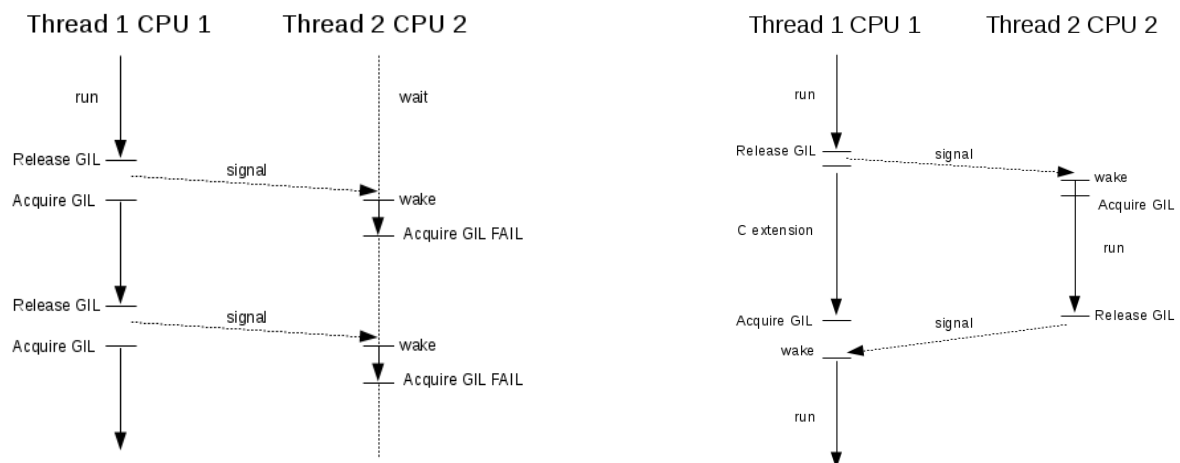


Fig. 1 – (a) left GIL contention in thread 2, (b) right preventing GIL contention using C extension

Objectives

The main objective is to create a easy to use GIL free high-performance computing environment that can be executed from every platform that supports both Python and features a C compiler. Even on platforms where no interpreted an embedded one can be used. On mobile platforms off-line compilation is an option but out of scope of this research though feasible to be achieved.

The main objectives for Concurrent are:

1. Cross-Platform HPC environment
2. Distribute task execution on several worker nodes
3. Usage of all available resources (CPU and GPU friendly [10])
4. Address pythons GIL problem
5. High performance Python through compiled Python using Cython
6. Non-shared memory at general level but shared on job level (OpenMP [11] usage within jobs)
7. Data analysis and scientific stack using the SciPy stack
8. Develop real world examples demonstrating scientific and leisure usages

Related work

There are many concurrency frameworks build using python in the large world of open source. Many of them even implement underlying OpenMP or MPI mechanisms making it easier to create concurrent processing applications. The key point why we consider a new point of view is necessary is that none of those frameworks address the issues or work-flows Concurrent aim to solve. Building Concurrent is a heavy tasks and many ideas have been already development and are considered the basis of our own implementation. Some of those parallel computing frameworks are presented as an overview to outline the key pros and cons of their solution.

Event based frameworks

Implementing async I/O execution in python is a very simple tasks thanks to the integration of libevent. The idea behind such frameworks is to optimize the use of one thread/process to evenly distribute non-blocking work load.

Eventlet and Gevent are only two frameworks that make the usage of aync I/O operation easy. Both are based around the same idea while Gevent has been implemented using low-level C boosting its performance compared to eventlet.

Another possible way to build an async I/O system is to use libevent using a C extension directly or via Cython.

Concurents Web API framework is designed to adjust with such a schema while it is possible to implement the API using regular Python WSGI (Web Server Gateway Interface) module for flexibility.

Simple processing frameworks

Other processing frameworks such as pprocess are based behind the idea to take the most of off the local hardware and to map a given function to a map-able dataset and spread the work load over several cores. Those frameworks are used for simple tasks executing dedicated processes and waiting for its completion.

Parallel Python is another example of a simple processing framework while adding a cluster like behavior to scale over several machines.

Asyncoro is yet another framework to develop local applications using the power of the availble processes of the host machine while conforming with pythons thread API and making development easier.

Volunteer Computing

Volunteer computing such as solution like Folding@Home are very similar in nature to what we aim to achieve. Folding@Home is based on work polling workers machines operated by volunteers. Each of such workers request jobs from central worker servers which they execute on their own hardware. The drawback of some volunteer computing frameworks is its lack of flexibility and the type of applications it can process. Each volunteer node must expose its requirements and some can not execute a given tasks if the hardware of operating system is compatible.

The evolution

All discussed frameworks have a common key-point used to handle program execution flow, they are all based on local execution and use available processes or cluster to help out with the computation. Concurrent is a framework that uses a different point of view by executing the whole application on a distributed computation cluster by changing the programming model completely.

A Concurrent application is not build around the idea of sequential flow, instead it features a main application module that decides what tasks to spawn and when. The execution is totally leveraged, even the application module itself is executed remotely.

The process of creating a Concurrent application changes the programming model completely by adding the step of tasks synchronization into the match. Such a model has already been proven to be very efficient and resulting in exceptional computation power.

The concurrency model the framework pursues is similar to what has been achieved by IMPS SPU processing and it's job system. Many game developers using the PS3 failed to squeeze out the available power of the PS3 when it came out due to the fact they had to change completely how to schedule execution and to think about the proposed solutions. Jobs had to be scheduled and synchronized correctly to not create bottlenecks within the global execution, while difficult to achieve if we think about the possible performance gains when isolating small independent parts we are able to use all of the processing power.

In game development many tasks could be implemented using such a paradigm, each AI agent could execute on their own dedicated processor not interfering with the others.

The problem of synchronization still remains and with a distributed framework it can even lead to severe problems if two dependent tasks are located on different clusters or even different continent. Network Coordinate Systems could be integrated into the system while this is complete out of the research of this project.

Current parallel processing libraries for python

dispy

dispy is a simple python module used to distribute python functions and classes over different processes and even over the network. It does feature all basics to build a concurrent application but it does not impose a programming model.

It differs from concurrent in the way of its architecture, in dispy you have to administrate you cluster and you have to use that cluster for your own development. Concurrent in turn does not force you to create you own cluster you could connect you local application to an existing actor and just send your arbitrary workloads over to it.

Dispy does not handle network congestion very well as many others and is getting unreliable when reaching the MTU (maximum transmission unit) limit.

ParallelPython

ParallelPython is very similar to dispy, it features distributed computation over a set of servers that perform the distributed processing. As in dispy no type of congestion control is made and once the computation is finished the connection is being closed. It mainly uses the map function to execute a given function using a set of arguments. ParallelPython is forked bases to address GIL issues.

IPython

IPython is also known as Interactive Python. It makes using the interactive shell over a set of IPython instances easy. IPython is not library or an actual framework, it is an interactive version of python.

Superpy

Superpy is similar to what concurrent does, is also distributes python applications over a cluster. Superpy uses XMLRPC to distribute the workload which comes with a severe performance impact.

Superpy does feature some nice extras such as a GUI to administrate the clusters and the workload, thus it is only available on windows operating system. Concurrent in turn is doing its administration tasks over a web interface and a console command line application and is available for linux, mac and windows.

Planning

Methodology

The framework itself is a project that is very alive, we do not consider it to have a retirement soon. A framework is an evolving ever changing organism that has to adapt to changes quickly to survive. A traditional development methodology does not fit in such a scenario, neither in a project that aims to be developed further by the Open Source community.

Once it is clear that we will not follow traditional PM work-flows we have to chose an agile methodology. In our we have decided to use Extreme Programming short XP [12].

XP is an agile methodology designed to adapt to changes and to work well with projects that will fight with high risks. In our case the risk of failure is high due to the fact that we are aiming to address and improve on new grounds. We have several issues to address from synchronization to come along Python's GIL.

XP is also focusing on small teams, the project itself will be build by a single developer but with the Open Source community in mind more developer might join the journey. XP has proven to work very well in teams from 2 to 12 developer.

Another important aspect of XP is that it aims to be test-driven. Python comes with an integrated unittest and doctest framework which we will use extensively.

To finally decide to use an agile methodology we have to answer seven simple questions:

1) Is the project size adequate?

The size of the project is neither large nor small. It is complex and we have a small team available for it. In fact it is just one developer and the professor leading this research. So we can answer yes.

2) Are the requirements dynamic?

Our requirements may seem stable but considering this is a research project we may change our focus. Also new discoveries can change the direction totally. We can conclude that our requirements are in fact dynamic.

3) What happens if the system fails?

We will produce a stable project, but while we are developing it we can accept bugs. There is also no direct risk for the framework. It is mainly research oriented and so we have to know that it will have bugs.

4) Does the project owner have limited time to invest?

Being the product owner myself I have to dedicate totally to the project. So we do not have limited dedication.

5) How many junior developer does the project team have

Simply none considering only myself.

6) How is the business culture of the developing party?

We will follow an informal work-flow. Mainly because of the own nature of the project.

7) Do you want to use agile?

Yes. Agile seems to fit perfectly as stated earlier and so it seems to be the right choice.

Analyzing our answers to the given questions we can assure that an agile methodology is the correct way to achieve the objectives and goals of the project.

Project Plan

We follow an agile methodology, in concrete the Extreme Programming (short XP) methodology. Extreme programming is a common used agile methodology for experienced teams and very appropriate for projects that are prone to changes. Our case is no other, the concurrent world changes rapidly and we have to adapt to new architectures and technologies as soon as they arrive.

Extreme Programming features six phases that go from the very beginning until the death of the project. In our case we will only use the production phases due to the fact that our framework is intended for research and so the end of its life-cycle is still unknown.

XP Phases

- Exploration
 - Spike / Prototype
 - User Stories
- Planning
 - Iteration plan
 - Scope
 - Define deliverables
- Iterating (once for each iteration)
 - Estimate cost of stories that are going to be implemented in the iteration

User Stories

Before we are able to create our user stories we prepare our high-level organization in five deliverable packages short WBS. Each WBS will focus on a main feature set required by the framework as shown on table 1.

WBS	Main Feature
WBS 1	Base Framework
WBS 2	Task Scheduling / Execution
WBS 3	SDK – Robot battle arena example
WBS 4	SDK – Concurrent data analysis methods
WBS 5	SDK – Benchmarks examples

Table 1 – Main WBS deliverables

The following User Stories serve as our base planning and required feature set. Each User story will then be transformed into tasks which in turn are implemented in the upcoming iterations. Our framework is intended to be used by expert users, the traditional en user is actually using and programming with our framework. The developer in turn is meant to be developing the framework itself.

WBS	ID	User Story
WBS 1	1.1	Developer must be able to use unit and doc tests for implemented modules
	1.2	Developer must be able to document its code automatically
	1.3	Developer must not setup application stack (scipy stack, Cython)
	1.4	User must be able to create Cython modules and send them to the framework
	1.5	User must be able to test his implementation locally before sending it to the framework
	1.6	User must be able to benchmark its module when executed on the framework
	1.7	Developer must be able to use any of our main platforms to develop (windows, osx, linux)
	1.8	Multiprocessing/Threading back-ends must be transparent to both user and developers
	1.9	OpenMP integration must be transparent to both user and developers
	1.10	OpenCL integration must be transparent to both user and developers
	1.11	User and developer must be able to trace execution along the system in a graphical manner
	1.12	User and developer must be able to choose between lock-free and lock-step execution mode selecting the step resolution in both methods
	1.13	Synchronization mechanisms must be transparent to both user and developers
	1.14	User and developer should not be able to interfere in execution of other tasks environments
	1.15	User and developer should be able to specify a tasks environment for a given set of modules
	1.16	User and developer should be able to add tasks to an existing task environment
WBS 2	2.1	User executes task in dedicated process of systems worker pool
	2.2	User should be able to compile it's tasks module into compiled python/cython code
	2.3	User should be able to define subtasks that are getting executed on the same process or another one (same process is preferred due to less synchronization)
	2.4	User task data changes should get synchronized with other data at the end of a step
	2.5	User should be able to communicate directly with other tasks or in deferred at the end of a step
	2.6	User should be able to send arbitrary objects back and forth the environment
	2.7	User should be able to see tasks splitting and execution after each step
WBS 3	3.1	User should be able to define a robot and its behavior as a System (see architectural overview)
	3.2	User should be able to query environment state based on sensors
	3.3	User should be able to create behavior through a behavior tree

WBS	ID	User Story
	3.4	User should be able to observe how the turns (steps) are developing
	3.5	User should be able to measure execution of his system
	3.6	User should get a final score and stats of his robot implementation
WBS 4	4.1	User should be able to perform LDA, MDS, PCA data analysis
	4.2	User should be able to perform genetic algorithms
	4.3	User should be able to perform DNA Curvature analysis
	4.4	User should be able to measure execution of his system
WBS 5	5.1	User should be able to perform high-performance counters
	5.2	User should be able to benchmark several aspects at once using a stats system
	5.3	User should be able to request the stats of his system
	5.4	User should be able to specify accumulative stats
	5.6	User should be able to use stats that reset every step
	5.5	Highest, average and lowest should be recorded for benchmarking stats

Table 2 – User stories organized per WBS

The weight of each story is defined when assigning tasks the first day a new iteration starts. Basic functionality and framework related stories will weight more then additional functionality that is used only in the examples.

It may come to the case that additional functionality is getting developed. In such a scenario extra WBS packages will get created suffixing them with a upper case A letter (WBSA) to prevent confusion.

Cost, Scope and Iterations

We divide the scope into several packages (see *Objectives > Results*) which go from the framework/SDK through the benchmarks, examples and finally documentation. The scope is summarized by the user stories presented earlier and placed in the following groups:

- Framework / SDK
- SDK Documentation
- Examples
- Benchmarks
- Distributed nodes

The cost of the project is calculated per iteration. As stated early we will follow a total of 7 iterations plus the initial spike/planning phase (for more information view appendix A to check the Gantt project overview).

Apart from the time we have to consider the cost of our main resource to calculate the expected cost of the project in terms of time investment. Table X illustrates the time required for each Iteration and the associated cost. We will expect a 3 h investment per day considering a weekly work as 7 days week. Each iteration lasts about two weeks apart from the initial planning spike which lasted 3 weeks. The cost of each hour is set to €50.

Iteration	Time	Expected Cost
Spike / Planning	15 days – 45 h	2250
Iteration 1	10 days – 30 h	1500
Iteration 2	10 days – 30 h	1500
Iteration 3	10 days – 30 h	1500
Iteration 4	10 days – 30 h	1500
Iteration 5	10 days – 30 h	1500
Iteration 6	10 days – 30 h	1500
Iteration 7	10 days – 30 h	1500
Project Corrections	5 days – 15 h	750
Total	90 days – 270 h	13500

Table 3 – Iteration plan

Architecture

Overview

Concurrent is a framework to achieve results with less effort. It includes job scheduling, a full scientific computing stack, industry standard threading support through Cython and OpenMP.

Concurrent is a node based distributed system executing an application and its tasks within a realm of servers. The idea behind the distributed system is simple: take advantage about common web technology to create a scalable and consistent distributed task execution system. Concurrent is build following a master/slave architecture to create a simple yet powerful distributed computation framework. Concurrent is designed to be both high-fault tolerant and to be deployed heterogeneous low-cost hardware.

The system is different from other concurrent frameworks by the fact that we do not just execute simple functions on it, we rather serialize full featured python applications which are then executed and orchestrated by the master to achieve high concurrency levels. The application developer is then responsible to define how the application should execute on the framework. One drawback of such a framework is that remote debugging can be sometimes painful, some of these issues has been addressed in the framework later on.

Components

The framework follows a plug-able component architecture very similar to Trac [put ref], a very popular python based project management tool. Figure 1. show the basic class hierarchy of the component framework.

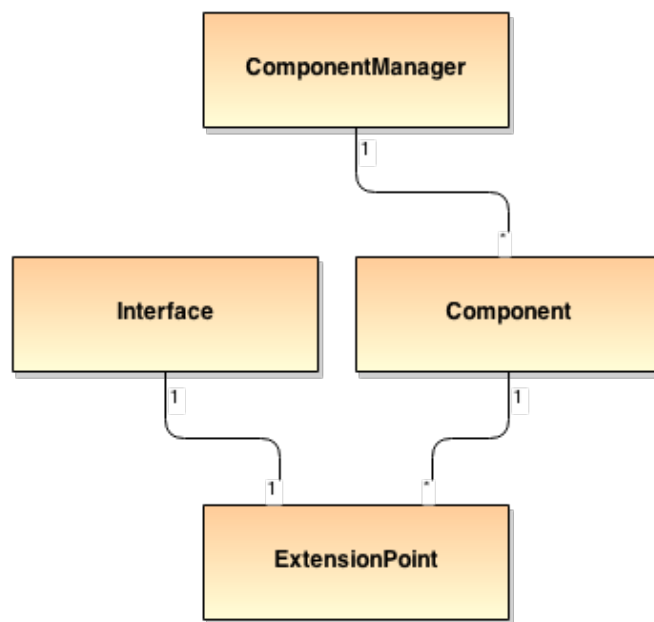


Figure 1 – Concurrent component framework

As you can see the system is based on four main classes, the *ComponentManager*, the *Component* itself and the *ExtensionPoint* using a given *Interface*.

The *ComponentManger* controls the life cycle of a *Component* and holds a list of all active *Components* that has been spawned using the manager. A manager can only handle a single instance of a *Component*. While a multi-manager setup is indeed possible it will introduce more complications then benefits.

The Component is our plug-ins abstract base class. A component can implement one or more interfaces which dictates its behavior.

An ExtensionPoint is used as a link where components can plug in to. Declaring an extension point will automatically give a list of suitable Components at runtime giving us the ability for hot plugin reloading.

An Interface defines the link between an extension and a component. It declares it's access API and assures that all components it contains are conforming to it.

Figure 2 outline a simple example how extension points are organized and how they will act in runtime.

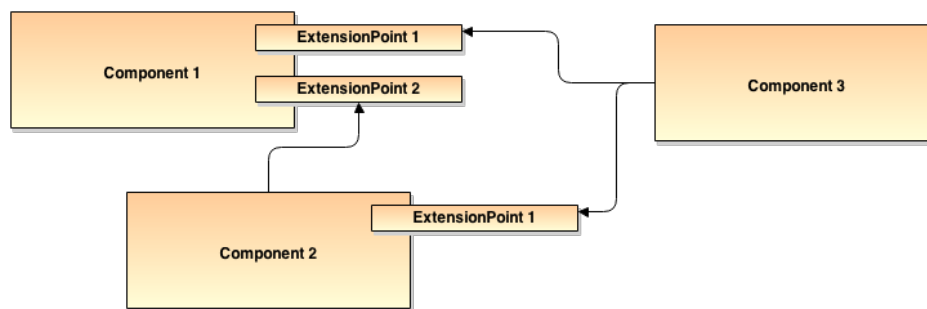


Figure 2 – Sample component setup

Components does come with certain built-in features such as a reference to the global environment (through the environment class) for restricted file system access and global stores, logging functionality using a configured Python logger and a simple configuration access using hot reload of configuration values.

Nodes

As stated before Concurrent used a master/slave like architecture to be able to achieve high-fault tolerance and to be deployed every where on heterogeneous hardware. Figure 3 shows the class hierarchy of our node based framework.

As shown in Figure 1 each leaf node is also a component and implement the IApp interface. The IApp interface is used to launch an autonomous application from a folder set as the run environment. An application environment features a sandbox prevent applications to write or read outside a given file hierarchy and so is essential for the security of the system. The tree principal nodes within the framework are the MasterMode, responsible to control the distributed system and to decide on the workload, the ComputationNode, a slave node used to perform all required computations, and the ApplicationNode, this node it executed by the end user representing the node how requested the computation.

Each node features a simple JSON-RPC 2.0 compatible web API used as a control channel and to process heartbeats and bidirectional message passing. The web interface also exposes the API to the public for maintenance and overall control.

The ServerNode features the addition of a TCP Server to communicate with other nodes using persistent connections for high-performance. Node is a simple node featuring the connection to a configured MasterNode, this is crucial to start the computation process and to handle disconnection and automatical recovery.

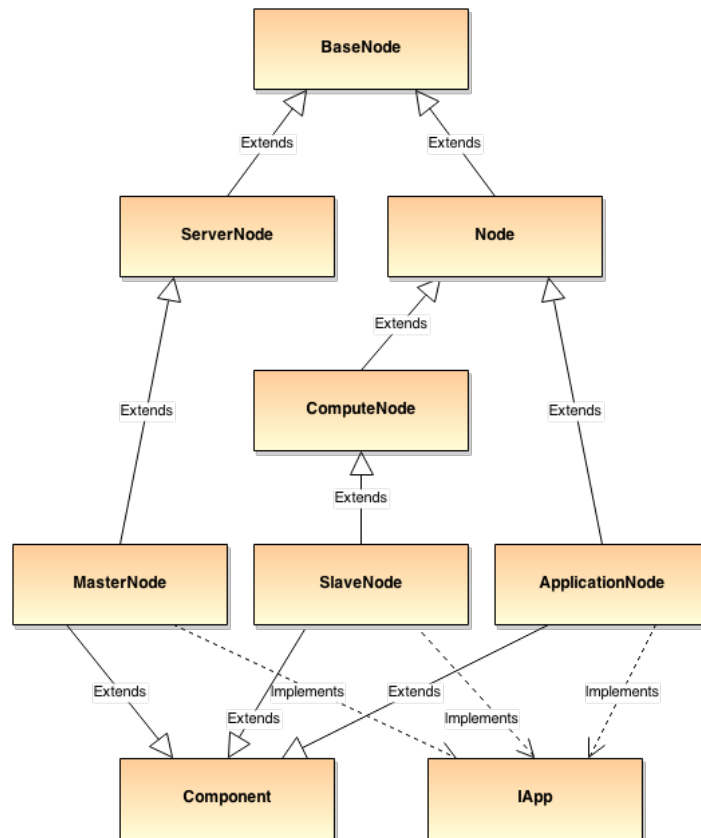


Figure 3 – Node architecture class hierarchy

All nodes except the MasterNode, the SlaveNode and the child classes of an ApplicationNode are abstract node organizing the feature set.

MasterNode

The MasterNode is the entry point of the framework. All slave nodes are connected to a single master, or in the case of a multi-master setup to a set of master using chaining mechanism. Even the application nodes are connected to a given master to monitor the execution of its requested applications.

The master handle a global registry for both connected clients (ApplicationNodes) and slaves, it handle registration and scheduling their work (see work scheduling for more information). The master node expects every connect node to send periodical heartbeats to control the state of the distributed network.

SlaveNode

A SlaveNode is a simple computation node that hosts the processing processes. The SlaveNode is an heterogeneous system featured a wide range of processing hardware, such as general purpose CPUs, co-processing units or GPU units.

A SlaveNode simple received a job from a MasterNode and schedules its execution on to its processing hardware. Concurrent comes with a simple scheduling strategy trying to balance the workload evenly on all its processing units.

The processing policy is yet another ExtensionPoint and so different SlaveNodes are able to feature different scheduling policies which makes a heterogeneous system flexible and better optimized.

ApplicationNode

Last but not least the system features the ApplicationNode. This node is the node that is launched from a client process requesting computation. The node features a simple WEB API used for dimensional communication with the master node without the need to open a dedicated socket with the master. The ability to open an active communication with the master is also possible and very useful for many programming schemes.

Configs

One of the most important parts of any framework is how easy it is to tweak the system. We have implemented a config system that features hot reloading and integrated within our component architecture. Figure X shows the config system in detail outlining the various types of configuration values concurrent supports. The most important ConfigItem is the ExtensionPointItem.

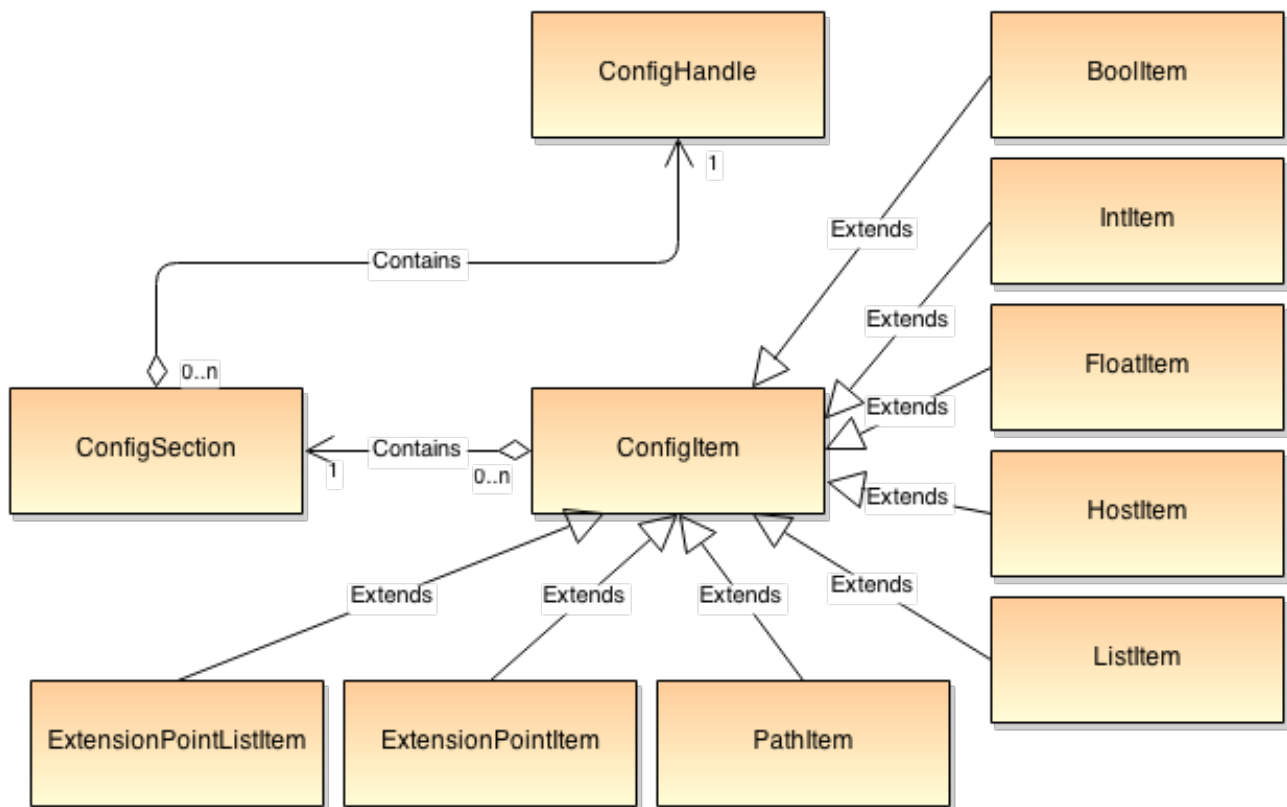


Figure X – Config system in detail

The **ExtensionPointItem** lets us drive a plug-able environment. As shown in the samples we just reference using a string a component that should be accessed via a config value. This way we can change the scheduling strategy, the application that is running, the task manager used and much more.

Transport concerns

RESTful

Communication in Concurrent is based in two main sections. The first one is called the control channel handling registration, maintenance and public access to each of the nodes, in short it represents our RESTful public API. The second communication system is the underlying TCP socket system using a very simple communication protocol similar to a RPC web service.

Our main RESTful API is build around JSON-RPC 2.0, it is a very simple yet powerful standardized way to communicate used by many organizations such as Google or even online gaming companies such as Valve and their Steam distribution service.

The API features registering of nodes, heartbeats, status checks, sending an application to be processed using a specialized web interface, reviewing tasks execution in real-time or even changing the behavior of a given node.

While using an RESTful API is very easy and flexible using proxy objects making the call to RPC methods seamless it comes along with low performance implications. Each call initiates a new HTTP connections, serializes the data over a POST handler and sends the data over to the end node resolving host names and going through a full RTT (Round Trip Time) run. It is not feasible using the API interface for computation or task synchronization. Our tests stated that using the control channel for computation related communicates results in a global throughput of about 50 to 100 requests per second on a local Ethernet installation. While this could seam reasonable taking into account that we expect long lasting computations it will add long waiting times to schedule new tasks and to perform result processing on the MasterNode.

TCP Sockets

The solution to the network performance problems while remaining flexible and transparent to the caller is to implement a low level TCP server and socket subsystem that operates on the same RPC principal then our RESTful API. While the messages send using the computation channel are also JSON messages (smaller in size and simpler then the JSON-RPC 2.0 standard to optimize bandwidth usage).

The computation network layer can be used the same way the control channel is used, we just call a remote method using a proxy system given us the ability to call a method just like it where a local class instance.

While the throughput of the control channel is about 50 to 100 requests per second we are able to achieve nearly 8000 requests per second using the computation channel using the same test machine and setup. The control channel gives us the ability to handle many thousands of different concurrent connections while the computation channel lets us handle a high load of data interaction using persistent sockets while supporting less concurrent connections. Using a mix of both worlds gives us benefits of both worlds while being able to work around their drawbacks.

ZeroMQ

While using a simple TCP server we get a very decent boost in throughput we also start getting congestion issues over TCP. Congestion of the protocol arise when we are sending too much packets over the same socket, for example sending 10k task requests and results for very small workloads. What we end-up is that some packages are mixed with others and we stop being able to read the stream properly, we end up reading messages messages our of order and finally loosing data. While the simplest solution for such cases is to sleep a very small amount of time after sending data through a socket this is still our main bottleneck.

Concurrent has chosen to implement its computation channel over ZeroMQ [15] instead of low level sockets. ZeroMQ, short ZMQ, helps us leveraging congestion control and implementing several patterns we require to distribute work-load. It comes with a full set of patterns that gives us scalability and fault tolerance.

ZMQ basically buffers internally what has to be send and dispatches packages to avoid congestion of the TCP layer. While it also comes along with a small performance impact.

RPC implementation

The main concern about how we implement communication between nodes is to not scarify flexibility in any case. Python is all about fast development and we should not require to spend large amount of time creating new interaction schemata when we just want to call a function on the other side.

The RPC model, used commonly in any modern web page, fits perfectly in our system. Implementing a common layer to connect nodes being it over HTTP or our low level TCP or ZMQ layer should be fully transparent to the user. The following example code shows how we create a callable function for both the JSON-RPC channel and our TCP Channel (ZMQ and TCP servers are equal in their interface).

```
@jsonremote(self.api_service_v1)
def register_slave(request, node_id, port, data):
    self.stats.add_avg('register_slave')
    return self.register_node(node_id, web.ctx['ip'], port, data, NodeType.slave)

@tcpremote(self.zmq_server, name='register_slave')
def register_slave_tcp(handler, request, node_id):
    self.stats.add_avg('register_slave_tcp')
    return self.register_node_tcp(handler, request, node_id, NodeType.slave)
```

Code 1 – Registering a slave node over the control (json-rpc) and the computation channel (tcp)

As you can see we use decorators to transform ordinary functions into callable functions from the outside. We register or expose functionality to be callable from other nodes to build a API for all interactions. This way all interactions are handled the same way and are fully transparent to the transport layer used.

The transport layer will receive a packet of data containing a dictionary defining the function name and its arguments. It then just simply calls the exposed function and returns the result if needed. In case no result is required a special exception must be raised to stop sending data.

Calling those exposed methods is done using proxy mechanisms that transform function calls into conformed dictionaries that the other end can understand. Such a proxy for our control channel is outlined in the following code example.

```
class TCPProxy(object):
    """
    TCP socket proxy using our JSON RPC protocol. The TCP proxy does not handle
    the answer from the given call, the answers are beeing received within the sockets
    own answer thread.
    """
    def __init__(self, ProxyObject, log):
        self._obj=ProxyObject
        self.log = log
```

```

self.stats = Stats.getInstance()

class _Method(object):

    def __init__(self, owner, proxy_obj, method, log):
        self.owner = owner
        self._obj=proxy_obj
        self.method = method
        self.log = log

    def __call__(self, *args, **kwargs):
        # Connect and close are very special
        if self.method == "close":
            self._obj.close()
        elif self.method == "connect":
            self._obj.connect()
        else:
            try:
                start_time = time.time()
                self._obj.send_to(self.method, *args, **kwargs)
            except Exception as e:
                raise e
            finally:
                self.owner.stats.add_avg('TCPProxy', time.time() - start_time)

    def __call__(self, method, *args, **kwargs):
        # Connect and close are very special
        if method == "close":
            self._obj.close()
        elif method == "connect":
            self._obj.connect()
        else:
            try:
                start_time = time.time()
                self._obj.send_to(self.method, *args, **kwargs)
            except Exception as e:
                raise e
            finally:
                self.owner.stats.add_avg('TCPProxy', time.time() - start_time)

    def __getattr__(self, method):
        # Connect and close are very special
        return self._Method(self, self._obj, method = method, log = self.log)

    def dump_stats(self):
        self.log.debug(self.stats.dump('TCPProxy'))

```

Code 2 – TCP proxy using an underlying socket object

Calling such a proxy is as simple as

```
self.master_node_tcp.register_slave(self.node_id_str)
```

Code 3 – Calling a proxy method

All proxy calls are made in a non-blocking fashion, what that means is that we are not getting the result directly and so we have to await it. Calling a RPC method requires us to expose in turn methods the other end will call to inform us with the results, methods that use a fire and forget pattern wont require any additional methods on the client. The following shows the exposed result methods for the call made in Code 3.

```
@tcpremote(self.master_node_tcp_client)
def register_slave_failed(handler, request, result):
    self.register_slave_failed(result)

@tcpremote(self.master_node_tcp_client)
def register_slave_response(handler, request, result):
    self.register_slave_response(result)
```

Code 4 – Response methods for RPC call in Code 3

Figure X shows the transport package with its main classes. The most important classes are the TCPServer and the TCPServerZQM classes. Both implement a the basic servers that the framework use, both are also TCPHandlers which are used to register the @tcpremote decorator shown in Code 4.

Work Distribution

Work distribution is a keystone of every parallel framework and such it is also one of the bigger parts of Concurrent. While on other frameworks such as Folding@Home the worker nodes are the nodes who actively request new work a concurrent SlaveNode is given the work it has to process. The MasterNode knows its hardware and schedules the work using the best possible worker to complete the given task.

Another key point of Concurrent is its flexibility, while using other volunteer systems require the computation nodes to hold a set of prerequisites, Concurrent uses python code serialization and sends actual instances of python's byte code over the framework and to the SlaveNode back and forth. The message used in the framework is called pickling.

Pickling is an object serialization local to Python itself and so we are able to create a Concurrent application from any machine and send it over to all possible workers on the framework creating a fully heterogeneous system.

The process of creating a Concurrent application is straightforward, the client's local ApplicationNode just creates a subclass of our global ITaskSystem base class (note that this is not a component or a component's Interface class because those are not serializable for security reasons). It then connects to a MasterNode, registers itself and sends the instance of ITaskSystem as a Pickle over the network. The MasterNode receives the pickled ITaskSystem and deserializes it, once deserialized it will then ask the system to create the first set of tasks. The tasks are then queued into a global work queue which the TaskScheduler picks up and then sends to all nodes evenly. The SlaveNode receives a task, again as a pickle and deserializing it, and adds the tasks to his internal job queue. One of the SlaveNode's worker processes will then process the task and send the result onto the SlaveNode's global result queue. The SlaveNode will pack together several results to send them as one package over to the MasterNode which then routes the results to the originating ITaskSystem. The ITaskSystem then can decide to spawn new tasks or to consider the calculation finished to send the global application results to the ApplicationNode who requested the work. Figure 4 outlines the architecture and its interaction outlining the previous example.

While using an autonomous ITaskSystem to handle execution flow is in many cases the best way to perform our computation, a complex system can create several systems and send them to be executed with different tasks and so on, we sometimes require a simpler approach.

In Concurrent the client program can specify and send tasks or batches of tasks directly to the framework and wait for its completion. Even if we send a batch of tasks over to the backend once a task is finished we will be informed of the result directly.

Sending work versus polling work

Concurrent implements two types of basic work scheduling, the first one is sending the work to a given slave with the best rating for a task, and the second type is a round-robin implementation pulling work from a common load-balancer.

The first type of scheduling is suitable for a heterogeneous system where we send the tasks based on the slave's hardware and software features. This way we are able to select the best slave to complete a given task. Different scheduling strategies can be implemented and plugged in to change the behavior of how to select the best possible slave for the actual work. Figure 4 shows the execution flow in detail.

The second scheduling type is a simple but very scalable round-robin push/pull pattern. The push/pull pattern basically pushes work onto an accessible socket where the slaves receive their data. The slave pulls the work from it, does the processing and sends the results back to the master.

Figure X shows the execution flow in detail.

Sending work execution flow

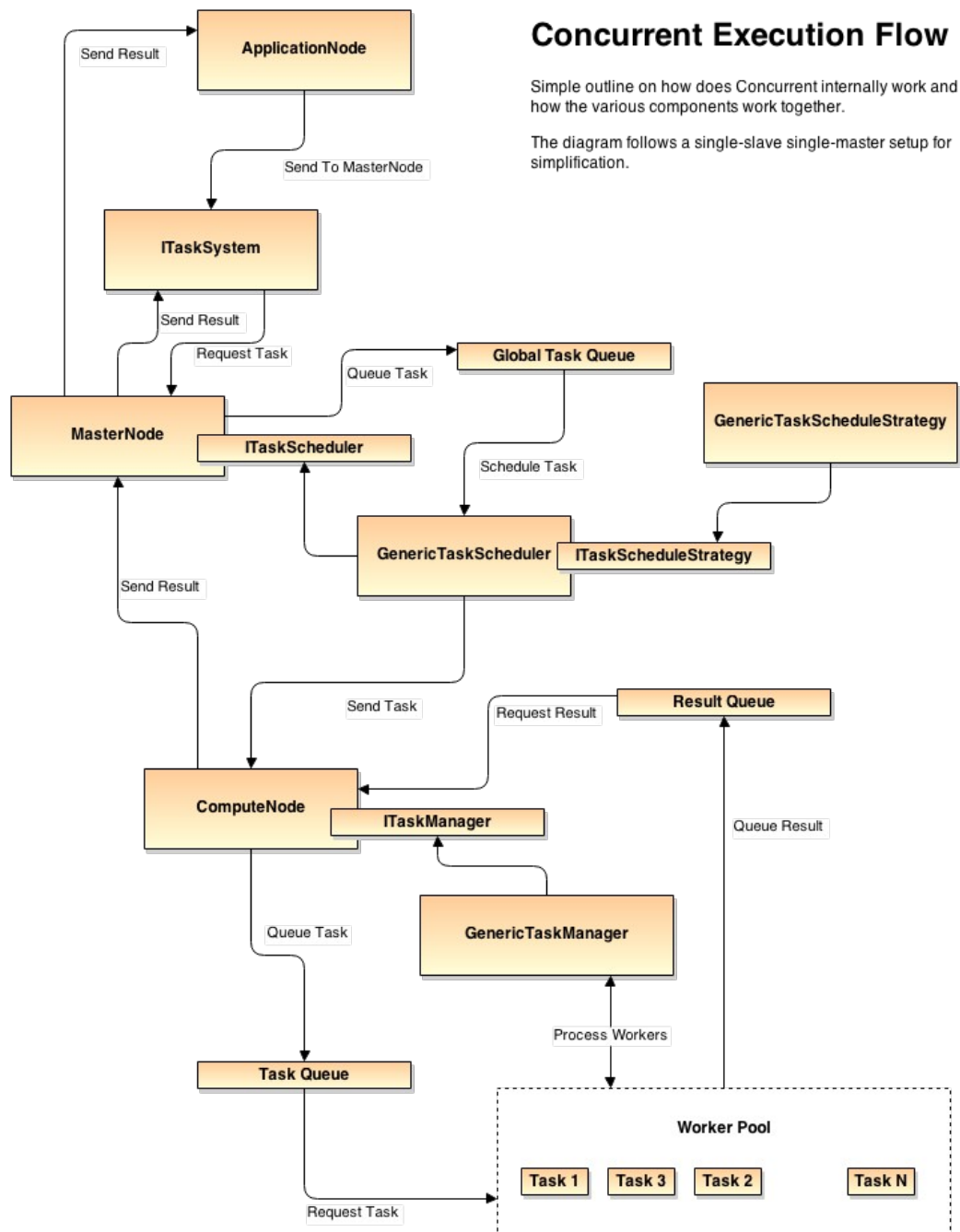


Figure 4 – Concurrents execution flow

Polling work execution flow (round-robin)

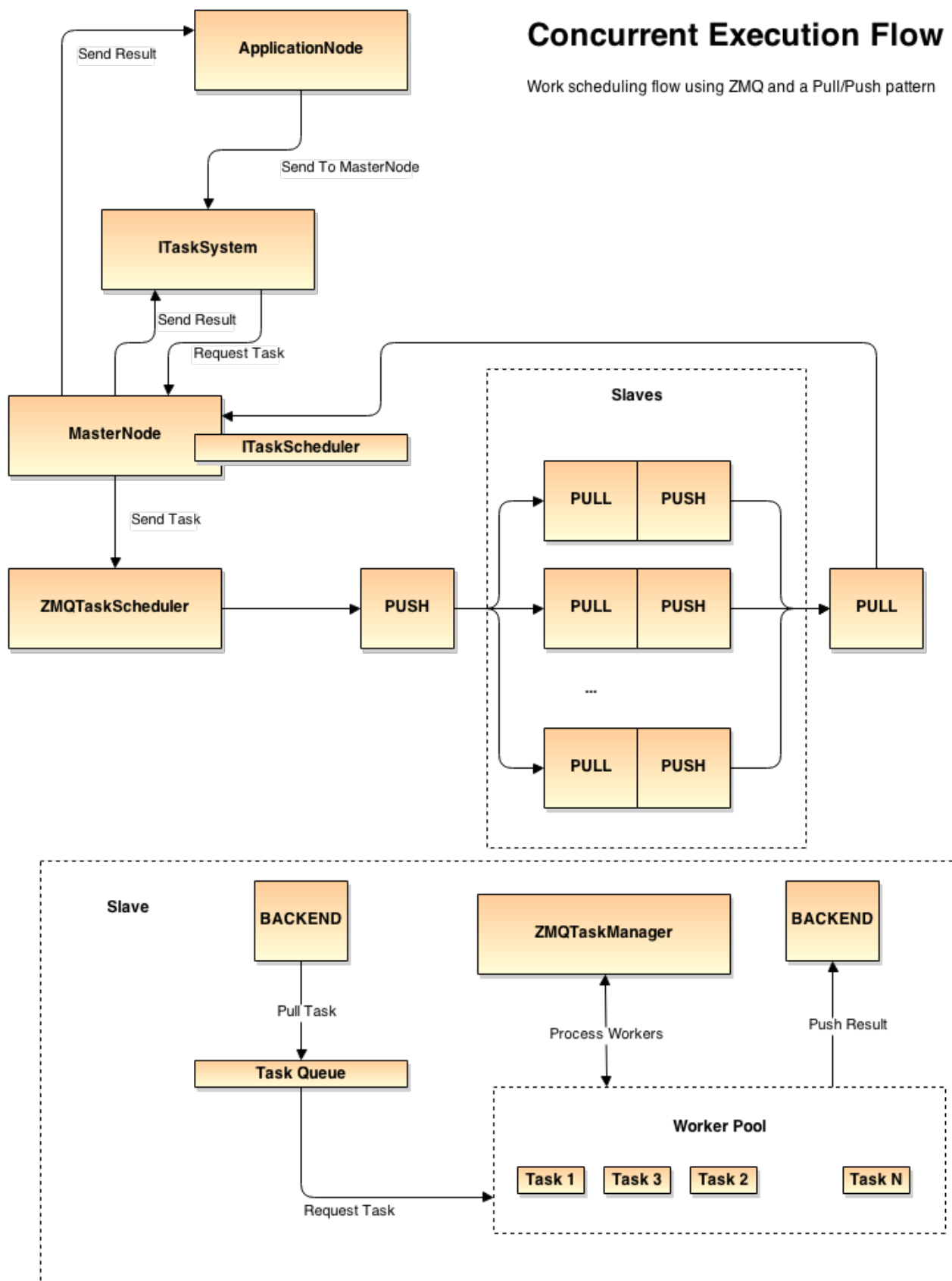


Figure X – Round-Robin scheduling execution flow

Simple tasks and batches

As mentioned earlier concurrent features two ways of sending for to the framework, the first one is to create an `ITaskSystem` which handle task creation and monitoring and the second is sending plain tasks over to be processed. Sending single tasks is in many cases the best solution but we also have to take into account that each time we send a task we have some overhead to pay for it.

Batching tasks is a simple solution for such a scenario, while batching has also its downsides. If the number of tasks is too high the batch also increases and this will result in congestion of the socket we use to communicate with the master.

Security concerns

Security is a major concern in distributed system, both malicious code and non-legitimate users will result in poor performance or even total system failures. Our system features three major security risks that are addressed or avoidable by the end user or system administrator.

The first security risk comes from non-legitimate users, we currently do not feature any login or client authentication mechanisms to start a computation. While this is a first tear risk it can be implemented over the current control channel adding encryption and login. A node that has not been registered over the control channel is not able to use the computation channel and so we only require an extra layer of security. ZMQ features security through encryption and certificates making the framework secure and reliable.

The second security risk comes from the intrinsic fact of serializing executable code over the system. A pickle is deserialized on the system and executed, this can lead to malicious code being executed on the environment. An attacker could change the pickle and send over code to execute their own code such as a shell command or sending the private keys back to its origin or an external service. This risk has been mitigated using both encryption and encoding mechanisms.

The third and last risk is not directly concerned to a direct security risk. An arbitrary application executing its jobs on the framework could produce hangs or crashes and result in system instability, stales or resources consumptions letting other tasks without resources. All jobs are executed within a global exception handler ensuring that a process is not able to crash a worker process. While the danger of infinity loops are still there the python process will launch another exception on such a scenario and the `MasterNode` is informed of such bad behavior.

Monitoring Nodes and Performance

Performance and tasks execution maintenance is an important key part to both optimize tasks and the framework itself. The hardest tasks when creating fully parallel applications is handling the right synchronization techniques. Monitoring execution to find possible locks, bottlenecks or less then optimum tasks execution is a powerful debugging utility to code applications.

Every node features a simple stats system letting the application collect basic stats used for performance and later revision. The stats systems is based on the idea of producing comulative averages, maximum, minimum and call number of the given stat. The stat system is flushed every x seconds to a global backend to be reviewed later for the application and its execution.

The web interface and control channel are also used to access the current state of the system and to monitor real-time execution of tasks. This can be very useful to catch dependency errors such as circular dependencies.

Next Steps

There are several improvements that are suitable for Concurrent. The proposed changes go from stability, security over to performance improvement.

While the system is fault-tolerant and self restarting it requires a MasterNode to track the execution. If the MasterNode fails the execution is truncated and the ApplicationNode is informed to handle actions. A multi-node environment will increase the global fault-tolerance and thanks to the fact that the applications and the tasks are serializable pickles they can be stored in an external database to be recovered if required. A multi-node environment would require both a DHT (Dynamic Hash Table) to locate nodes and a NCS(Network Coordinate System) to choose near SlaveNode.

The control channel is currently using a multi-threaded web server, while this is good for a limited amount of concurrent connections we can reach the resource limit fast (the infamous 10K problem). Switching to an async I/O framework such as Eventlet, Gevent or implementing a low level libevent handler will increase the control channel performance. We have to take into account that the control channel is meant to be non-blocking and so an async I/O framework is very suitable. Using processes on the master node for the compute channel will solve our current GIL bottleneck. Using async I/O through ZMQ will give us the ability to handle even more concurrent connection.

The last short term improvement that could be made to the system is the creation of a sandbox. The nodes itself are already execution using a dedicated environment. The way to make sure that no access outside the environment can be made is to overwrite some python built-ins and to ensure no path can be accessed outside the assigned folder.

Enable slaves to expose their GPU hardware feature to be able to create Cuda / OpenCL tasks and boost vectorized computation on the framework. A task itself could be implemented using different ways. GPU computation will give us a unique flavor of how we use the distributed systems resources and optimize computation. If we use the system for a volunteer computation setup we have to take into account that many volunteers will do have a high-end graphic card that we could make use of.

Including GPU bases computation requires us to include a better scheduling strategy that in turn has to take into account the slaves capabilities and hardware features.

Statistics and real-time monitoring is a key point to create optimized tasks and will unveil the whole power of the system to the application developer. The developer has to see where it system hangs and why to be able to optimize its resource usage to produce optimized applications. Concurrent already come with an integrated stats system but it requires the real-time monitoring part.

Samples

All samples has been executed on a single-machine environment. This means that the network will have a high impact on the final results. This is ok for our purpose demonstrating the usage and the difference between using a traditional approach (plain tasks) and using a task system (ITaskSystem).

MD5 Hash Reverse

The chosen sample application is a simple hash reverse function. The procedure we are about to follow is a simple brute force technique splitting the workload over different processes to try different possible solutions to crash a MD5 hash.

We will outline the process of creating an application in a plug-able fashion which can be reused to create even more applications.

Setup

The first think we have to do is to create a simple setup script. Concurrent applications are loaded in run-time and so it is important to create a python egg file or a source distribution. The system later sends the egg file placed within a special directory of the environment to send it over to the framework and setup the required prerequisites in all nodes.

The application prerequisites are installed on all nodes automatically and uninstalled when the application finishes and stops working. Using Pythons distribution tools and the 'pip' installer, a frameworks own prerequisite, makes installing and uninstalling easy and reliable by just executing simple shell commands.

Code 1 shows a simple setup script used by one of our sample applications. In this case we do not require to send the created python egg onto the framework because the sample ships with the framework itself and is used by the automated test framework.

```
# -*- coding: utf-8 -*-
"""
Sample application using Concurrent to perform a reverse hash

File: reversemd5.setup.py
"""
from setuptools import find_packages, setup

setup(
    name='ConcurrentReverseMD5', version='0.1',
    packages=find_packages(exclude=['*.tests*']),
    entry_points = {
        'concurrent.components': [
            'samples.reversemd5 = concurrent.famework.samples.reversemd5.app',
        ],
    },
)
```

Code X – Setup template script for Concurrent applications

The application

The application itself is very simple, we will just compute all possible hashes between the numbers 1 to 2000000. The workload is distributed over 128 tasks making the computation extremely parallel.

Our main application which we are going to launch is *MD5HashReverseNode*. Our setup file already loaded the file using the setup script and pointing to the python file setting up the entry points to be found in run-time.

Our application node connects to the pre-configured MasterNode using the configuration files located in the applications environment folder. Once the connection has been established we are asked to create our implementation of *ITaskSystem*. The task system is responsible of controlling the applications behavior on the MasterNode on behalf of our ApplicationNode.

Once the task system has reached the MasterNode it is asked to generate the first set of tasks. In our case we create 128 tasks instances of *MD5ReverseTask* and push them to the system.

Each time a tasks has finished we check the tasks result and if we found the hash we mark the system as finished. When marking a system as finished any pending tasks of that given systems are ignored and skipped for further processing.

The MasterNode then requests the final data it has to send over to the originating ApplicationNode and uninstalls the computations prerequisites.

```
# -*- coding: utf-8 -*-
"""
Sample application using Concurrent to perform a reverse hash

File: reversemd5.app.py
"""

from concurrent.framework.nodes.applicationnode import ApplicationNode
from concurrent.core.application.api import IApp
from concurrent.core.components.component import implements
from concurrent.core.async.task import Task
from concurrent.core.async.api import ITaskSystem

import time
import md5

class MD5HashReverseNode(ApplicationNode):
    """
    Reverse hash application
    """
    implements(IApp)

    def app_init(self):
        """
        Called just before the main entry. Used as the initialization point instead of the ctor
        """
        ApplicationNode.app_init(self)
```

```

def app_main(self):
    """
    Applications main entry
    """
    return ApplicationNode.app_main(self)

def get_task_system(self):
    """
    Called from the base class when we are connected to a MasterNode and we are
    able to send computation tasks over
    """
    return MD5HashReverseTaskSystem(128)

def task_system_finsihed(self, result, error, stats):
    """
    Called when our task system has finished. Returns either the result
    of the computation or an error. The stats dictionary provides useful
    information like the execution time.
    """
    if not error:
        if result:
            self.log.info("Hash as been reversed. Initial number was %s" % str(result))
        else:
            self.log.info("Failed to reverse the hash :(")
    else:
        self.log.error("Computation failed: %s" % str(error))

class MD5HashReverseTaskSystem(ITaskSystem):
    """
    The task system that is executed on the MasterNode and controls what jobs are required to be
    performed
    """

    def __init__(self, jobs=128, hash_number=1763965):
        """
        Default constructor used to initialize the base values. The ctor is
        executed on the ApplicationNode and not called on the MasterNode so we can
        use it to initialize values.
        """
        if not hash_number.isdigit():
            raise ValueError("hash_number must be a number!")

        self.start = 1
        self.end = 2000000
        self.target_hash = md5.new(str()).hexdigest()

        # Create a number of jobs that will be processed
        self.jobs = jobs
        self.finished_jobs = 0

```



```

self.step = (self.end - self.start) / jobs + 1
self.result

def init_system(self, master):
    """
    Initialize the system
    """
    self.start_time = time.time()

def generate_tasks(self, master):
    """
    Generate the initial tasks this system requires
    """
    job_list = []
    for i in xrange(self.jobs):
        job_start = self.start + i*self.step
        job_end = min(self.start + (i + 1)*self.step, self.end)
        job_list.append(MD5ReverseTask(target_hash=self.target_hash, start=job_start, end =
job_end))
    return job_list

def task_finished(self, master, task, result, error):
    """
    Called once a task has been performed
    """
    self.finished_jobs += 1
    if result:
        self.result = result

def gather_result(self, master):
    """
    Once the system stated that it has finsihed the MasterNode will request the required results
that
are to be send to the originator. Returns a tuple like (result, Error)
    """
    return (self.result, None)

def is_complete(self, master):
    """
    Ask the system if the computation has finsihed. If not we will go on and generate more tasks.
This
gets performed every time a tasks finishes.
    """
    # Wait until all computation has been finsihed or we have found the hash
    return self.result or self.finished_jobs == self.parts

class MD5ReverseTask(Task):

def __init__(self, name, system_id, **kwargs):
    Task.__init__(self, name, system_id, **kwargs)

```

```

print("Created task: %s" % str(self.task_id))

self.target_hash = kwargs['target_hash']
self.start = kwargs['start']
self.end = kwargs['end']

def __call__(self):
    """
    No try to find the hash
    """
    for i in xrange(self.start, self.end):
        if md5.new(str(i)).hexdigest() == self.target_hash:
            return i
    return None

def finished(self, result, error):
    """
    Once the task is finished. Called on the MasterNode within the main thread once
    the node has recovered the result data.
    """
    pass

```

Code X – Sample application demonstrating a simple reverse hashing algorithm

Benchmark

The benchmarking sample is a very simple application that minimizes data interaction to be able to outline the pure performance of the framework and its configurations.

The benchmark basically

The Application

The benchmark features two ApplicationNodes. The *ExpensiveNode*, which uses a task system to distribute the workload, and the *ExpensiveSimpleNode*, which sends tasks on its own to the distributed framework.

```
class ExpensiveNode(ApplicationNode):
    """
    Application node distributing the computation of an expensive task
    """
    implements(IApp)

    time_per_task = IntItem('expensivesample', 'time_per_task', 1,
        """Time each task will perform on doing nothind (active wait) to simulate an expensive
        computation""")

    num_tasks = IntItem('expensivesample', 'num_tasks', 8,
        """Number of tasks that must be performend""")

    def app_init(self):
        """
        Called just before the main entry. Used as the initialization point instead of the ctor
        """
        super(ExpensiveNode, self).app_init()

    def app_main(self):
        """
        Applications main entry
        """
        return super(ExpensiveNode, self).app_main()

    def get_task_system(self):
        """
        Called from the base class when we are connected to a MasterNode and we are
        able to send computation tasks over
        """
        self.start_time = time.time()
        self.system = ExpensiveNodeTaskSystem(self.time_per_task, self.num_tasks)
        return self.system

    def work_finished(self, result, task_system):
        """
        Called when the work has been done, the results is what our ITaskSystem
        sent back to us. Check resukt for more info
```

```

"""
end_time = time.time() - self.start_time
self.log.info("Total time: {}".format(end_time))

# Print expected single threaded time and improvement
expected_time = self.time_per_task * self.num_tasks
self.log.info("Plain python expected time: {}".format(expected_time))
self.log.info("Concurrent improvememnet: {}".format((expected_time/end_time)*100.0))
self.shutdown_main_loop()

def push_tasksystem_response(self, result):
    """
    We just added a ITaskSystem on the framework. Check result for more info
    """

    self.log.info("Tasks system send to computation framework")

def push_tasksystem_failed(self, result):
    """
    We failed to push a ITaskSystem on the computation framework!
    """

    self.log.error("Tasks system failed to be send to framework!")
    # Check if the resuklt dict contains a traceback
    if "t" in result:
        self.log.error(result["t"])

```

Code X – Application node using a tasks system for its workload

```

class ExpensiveNodeTaskSystem(ITaskSystem):
    """
    The task system that is executed on the MasterNode and controls what jobs are required to be
    performed
    """

    def __init__(self, time_per_task, num_tasks):
        """
        Default constructor used to initialize the base values. The ctor is
        executed on the ApplicationNode and not called on the MasterNode so we can
        use it to initialize values.
        """

        super(ExpensiveNodeTaskSystem, self).__init__()

        # Init task related stuff
        self.time_per_task = time_per_task
        self.num_tasks = num_tasks

    def init_system(self, master):
        """
        Initialize the system
        """

        pass

```

```

def generate_tasks(self, master):
    """
    Create task set
    """
    self.start_time = time.time()
    self.finished_jobs = 0
    return [ExpensiveTask("expensive_{}".format(i), self.system_id, None,
sleep_time=self.time_per_task) for i in range(self.num_tasks)]

def task_finished(self, master, task, result, error):
    """
    Called once a task has been performed
    """
    self.finished_jobs += 1

def gather_result(self, master):
    """
    Once the system stated that it has finsihed the MasterNode will request the required results
that
are to be send to the originator. Returns the total time spend on the master.
    """
    total_time = time.time() - self.start_time
    self.log.info("Calculated in {} seconds!".format(total_time))
    return total_time

def is_complete(self, master):
    """
    Ask the system if the computation has finsihed. If not we will go on and generate more tasks.
This
gets performed every time a tasks finishes.
    """
    self.log.info("%d -> %d" % (self.finished_jobs, self.num_tasks))
    # Wait for all tasks to finish
    return self.finished_jobs == self.num_tasks

```

Code X – Tasks system used by the ExpensiveNode sample

```

class ExpensiveSimpleNode(ApplicationNode):
    """
    Application node distributing the computation of the mandelbrot set using just tasks
    """
    implements(IApp)

    send_task_batch = BoolItem('expensivesample', 'task_batch', True,
        """Should we send all tasks one by one or should we batch them into a hughe list""")

    time_per_task = IntItem('expensivesample', 'time_per_task', 1,
        """Time each task will perform on doing nothind (active wait) to simulate an expensive
computation""")

    num_tasks = IntItem('expensivesample', 'num_tasks', 8,

```

```

"""Number of tasks that must be performend""")

def app_init(self):
    """
    Called just before the main entry. Used as the initialization point instead of the ctor
    """
    super(ExpensiveSimpleNode, self).app_init()

def app_main(self):
    """
    Applications main entry
    """
    return super(ExpensiveSimpleNode, self).app_main()

def get_task_system(self):
    """
    Called from the base class when we are connected to a MasterNode and we are
    able to send computation tasks over
    """
    # Do not create a tasks system, we will handle tasks on our own
    return None

def start_processing(self):
    """
    Called when the app is not using a ITaskSystem and will instead just add tasks and
    will take care of the task flow itself
    """
    self.log.info("Starting computation")
    if self.send_task_batch:
        self.log.info(" Task batching enabled")

    self.start_time = time.time()
    self.finished_jobs = 0
    if self.send_task_batch:
        self.push_tasks( [ExpensiveTask("expensive_{}".format(i), None, self.node_id_str,
sleep_time=self.time_per_task) for i in range(self.num_tasks)])
    else:
        for i in range(self.num_tasks):
            self.push_task( ExpensiveTask("expensive_{}".format(i), None, self.node_id_str,
sleep_time=self.time_per_task) )
            self.check_finished()

def task_finished(self, task, result, error):
    """
    Called when a task has been done
    """
    self.finished_jobs += 1
    self.check_finished()

def check_finished(self):

```

```

"""
Check if we finished all computation or not
"""
self.log.info("%d -> %d" % (self.finished_jobs, self.num_tasks))
if self.finished_jobs == self.num_tasks:
    self.log.info("All tasks finished!!")
    end_time = time.time() - self.start_time
    self.log.info("Total time: {}".format(end_time))

    # Print expected single threaded time and improvement
    expected_time = self.time_per_task * self.num_tasks
    self.log.info("Plain python expected time: {}".format(expected_time))
    self.log.info("Concurrent improvememnet: {}%".format((expected_time/end_time)*100.0))
    self.shutdown_main_loop()

def push_task_response(self, result):
    """
    We just add a Task to the computation framework
    """
    pass
    #self.log.info("Task send to computation framework")

def push_task_failed(self, result):
    """
    We failed to add a Task to the computation framework
    """
    self.log.info("Failed to send task send to computation framework")

def push_tasks_response(self, result):
    """
    We just add a set of Tasks to the computation framework
    """
    self.log.info("Tasks send to computation framework")

def push_tasks_failed(self, result):
    """
    We failed to add a set of Tasks to the computation framework
    """
    self.log.info("Failed to send tasks send to computation framework")

```

Code X – Application node sending tasks on its own

```

class ExpensiveTask(Task):

    def __init__(self, name, system_id, client_id, **kwargs):
        Task.__init__(self, name, system_id, client_id)
        self.sleep_time = kwargs['sleep_time']

    def __call__(self):
        """

```

```

Calculate assigned work
"""

# Simulate an active wait, this is more accurate then sleeping
end_time = time.time() + self.sleep_time
while True:
    if time.time() > end_time:
        break
    return self.sleep_time

def finished(self, result, error):
    """

    Once the task is finished. Called on the MasterNode within the main thread once
    the node has recovered the result data.
    """

    pass

def clean_up(self):
    """

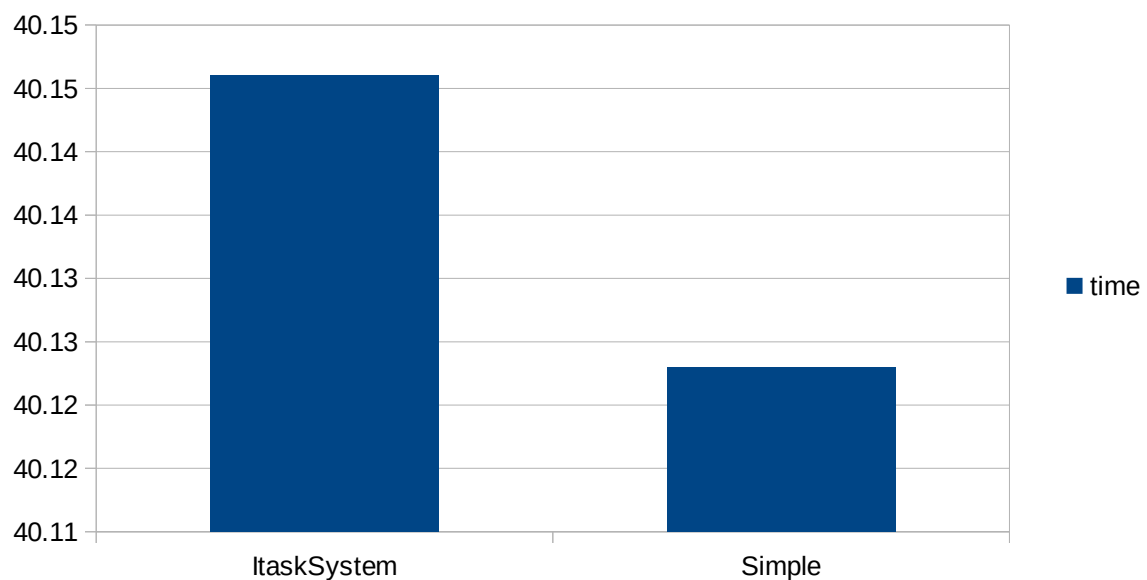
    Called once a task has been performed and its results are about to be sent back. This is used
    to optimize our network and to cleanup the tasks input data
    """

```

Code X – Tasks that just does a dummy computation, in our case an active wait

ITaskSystem vs plain tasks

Both system are nearly identical, the fact that we do not send any additional data gives us a very similar out come.



Optimizing number of workers

In our benchmark we aim to use all available resources for our computation. The fact is that the benchmark consumes 100% of the available CPU power as shown in figure X and figure X.

Image Name	User Name	CPU	Memory (...)
python.exe	Moss	13	21,620 K
python.exe	Moss	13	21,628 K
python.exe	Moss	13	21,652 K
python.exe	Moss	11	21,652 K
python.exe	Moss	11	21,656 K
python.exe	Moss	10	21,624 K
python.exe	Moss	08	50,080 K
python.exe	Moss	07	48,424 K
python.exe	Moss	07	21,652 K
python.exe	Moss	04	21,632 K
python.exe	Moss	04	46,696 K

Figure X – Worker processes currently active

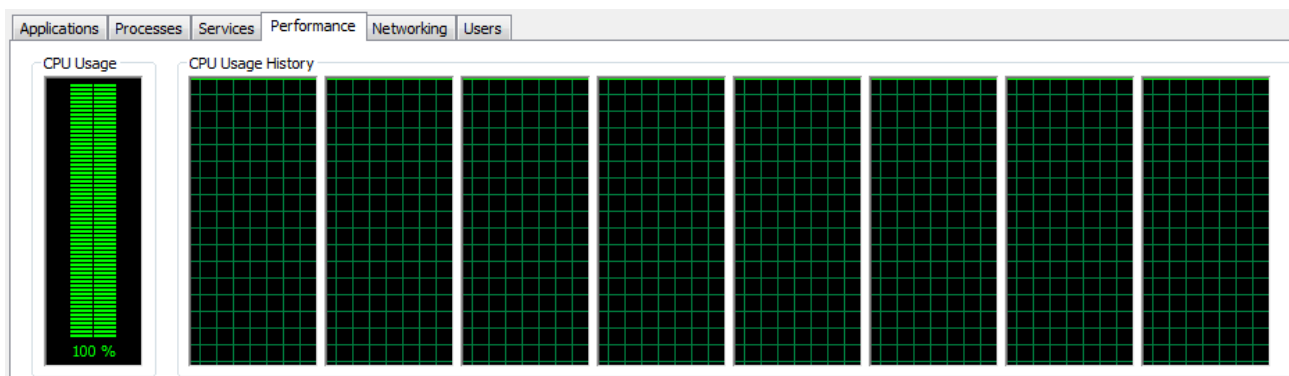


Figure X – System load during the benchmark

Mandelbrot

The Mandelbrot sample comes in two flavors. One using the `ITaskSystem` and using direct tasks, in both version we have to consider that the network plays an important role within the time consumed during the computation.

The application

```
class MandelbrotNode(ApplicationNode):
    """
    Application node distributing the computation of the mandlebrot set using an autonomous task
    system
    """
    implements(IApp)

    use_optimized_task = BoolItem('mandlebrotsample', 'use_optimized_task', True,
        """Should we use the data optimized task or the lazy task""")

    factor = IntItem('mandlebrotsample', 'factor', 1,
        """How many workloads does a single task get assigned, in our a workload is considered a
        row""")

    iters = IntItem('mandlebrotsample', 'iters', 20, """Mandlebrot iterations per pixel""")
    height = IntItem('mandlebrotsample', 'height', 1024, """Height of the mandlebrot set image""")
    width = IntItem('mandlebrotsample', 'width', 1536, """Width of the mandlebrot set image""")

    def app_init(self):
        """
        Called just before the main entry. Used as the initialization point instead of the ctor
        """
        super(MandelbrotNode, self).app_init()

    def app_main(self):
        """
        Applications main entry
        """
        return super(MandelbrotNode, self).app_main()

    def get_task_system(self):
        """
        Called from the base class when we are connected to a MasterNode and we are
        able to send computation tasks over
        """
        self.start_time = time.time()
        self.system = MandelbrotTaskSystem(-2.0, 1.0, -1.0, 1.0, self.height, self.width, self.iters,
        self.factor, self.use_optimized_task)
        return self.system
```

```

def work_finished(self, result, task_system):
    """
    Called when the work has been done, the results is what our ITaskSystem
    sent back to us. Check result for more info
    """
    print("Total time: {}".format(time.time() - self.start_time))
    self.shutdown_main_loop()
    # Reassemble result to be processed further
    try:
        self.system.image = np.zeros((self.height, self.width), dtype = np.uint8)
        self.system.do_post_run(result)
    except:
        traceback.print_exc()

def push_tasksystem_response(self, result):
    """
    We just added a ITaskSystem on the framework. Check result for more info
    """
    self.log.info("Tasks system send to computation framework")

def push_tasksystem_failed(self, result):
    """
    We failed to push a ITaskSystem on the computation framework!
    """
    self.log.error("Tasks system failed to be send to framework!")
    # Check if the result dict contains a traceback
    if "t" in result:
        self.log.error(result["t"])

```

Code X – Application node using a tasks system for its work

```

class MandlebrotTaskSystem(ITaskSystem):
    """
    The task system that is executed on the MasterNode and controls what jobs are required to be
    performed
    """

    def __init__(self, min_x, max_x, min_y, max_y, height, width, iters, factor, optimized):
        """
        Default constructor used to initialize the base values. The ctor is
        executed on the ApplicationNode and not called on the MasterNode so we can
        use it to initialize values.
        """
        super(MandlebrotTaskSystem, self).__init__()

        # Init task related stuff
        self.min_x = min_x
        self.max_x = max_x
        self.min_y = min_y

```

```

self.max_y = max_y
self.image = None
self.iters = iters
self.factor = factor
self.optimized = optimized

self.height = height
self.width = width
self.pixel_size_x = (self.max_x - self.min_x) / self.width
self.pixel_size_y = (self.max_y - self.min_y) / self.height

def do_post_run(self, result):
    """
    Once the computation finished we reassemble the image here
    """
    for x in range(self.width):
        for y in range(self.height):
            self.image[y, x] = result[x][y]

    imshow(self.image)
    show()

def init_system(self, master):
    """
    Initialize the system
    """
    pass

def generate_tasks(self, master):
    """
    Devide image in width part to distribute work
    """

    # Create a number of jobs that will be processed
    self.jobs = 0
    self.finished_jobs = 0
    self.result_dict = {}

    job_list = []
    workload = []

    rows = 0
    x = 0
    if self.optimized:
        num_tasks, reminder = divmod(self.width, self.factor)
        self.jobs = num_tasks + reminder

    for i in xrange(0, self.jobs):
        job_list.append(MandlebrotTaskOptimized("m", self.system_id, None,
            iters = self.iters, start_x = i, rows = self.factor, cols = self.height,

```

```

        pixel_size_x = self.pixel_size_x, pixel_size_y = self.pixel_size_y,
        min_x = self.min_x, min_y = self.min_y))
    else:
        for x in range(self.width):
            # Distribute using rows
            rows += 1

            real = self.min_x + x * self.pixel_size_x
            for y in range(self.height):
                imag = self.min_y + y * self.pixel_size_y
                workload.append((x, y, real, imag, self.itsers))

            # every self.factor rows create a task with the workload
            if rows == self.factor:
                job_list.append(MandlebrotTask("mandle_{}".format(x), self.system_id, None, itsers =
self.itsers, workload = workload))
                workload = []
                rows = 0

            # Add last task with rest of workload
            if len(workload) > 0:
                job_list.append(MandlebrotTask("mandle_{}".format(x), self.system_id, None, itsers =
self.itsers, workload = workload))

    self.jobs = len(job_list)
    self.start_time = time.time()
    return job_list

def task_finished(self, master, task, result, error):
    """
    Called once a task has been performed
    """
    self.finished_jobs += 1
    if result:
        self.result_dict.update(result)

def gather_result(self, master):
    """
    Once the system stated that it has finsihed the MasterNode will request the required results
that
are to be send to the originator. Returns a tuple like (result, Error)
    """
    print("Calculated in {} seconds!".format(time.time() - self.start_time))
    return self.result_dict

def is_complete(self, master):
    """
    Ask the system if the computation has finsihed. If not we will go on and generate more tasks.
This
gets performed every time a tasks finishes.

```

```

"""
    #print("%d -> %d" % (self.finished_jobs,self.jobs))
    # Wait for all tasks to finish
    return self.finished_jobs == self.jobs

```

Code X – Tasks system used by the MandlebrotNode application

```

class MandlebrotSimpleNode(ApplicationNode):
    """
        Application node distributing the computation of the mandlebrot set using just tasks
    """
    implements(IApp)

    use_optimized_task = BoolItem('mandlebrotsample', 'use_optimized_task', True,
        """Should we use the data optimized task or the lazy task""")

    send_task_batch = BoolItem('mandlebrotsample', 'task_batch', True,
        """Should we send all tasks one by one or should we batch them into a hughe list""")

    factor = IntItem('mandlebrotsample', 'factor', 1,
        """How many workloads does a single task get assigned, in our a workload is considered a row""")

    iters = IntItem('mandlebrotsample', 'iters', 20, """Mandlebrot iterations per pixel""")
    height = IntItem('mandlebrotsample', 'height', 1024, """Height of the mandlebrot set image""")
    width = IntItem('mandlebrotsample', 'width', 1536, """Width of the mandlebrot set image""")

    def app_init(self):
        """
            Called just before the main entry. Used as the initialization point instead of the ctor
        """
        super(MandlebrotSimpleNode, self).app_init()

    def app_main(self):
        """
            Applications main entry
        """
        return super(MandlebrotSimpleNode, self).app_main()

    def get_task_system(self):
        """
            Called from the base class when we are connected to a MasterNode and we are
            able to send computation tasks over
        """
        # Do not create a tasks system, we will handle tasks on our own
        return None

    def start_processing(self):

```

"""

Called when the app is not using a ITaskSystem and will instead just add tasks and will take care of the task flow itself

"""

```
self.log.info("Starting computation")
if self.send_task_batch:
    self.log.info(" Task batching enabled")

self.start_time = time.time()
self.image = np.zeros((self.height, self.width), dtype = np.uint8)

# Init task related stuff
self.min_x = -2.0
self.max_x = 1.0
self.min_y = -1.0
self.max_y = 1.0

self.pixel_size_x = (self.max_x - self.min_x) / self.width
self.pixel_size_y = (self.max_y - self.min_y) / self.height

# Job handling (very optimistic :D)
self.jobs = 0
self.finished_jobs = 0

job_list = []
workload = []

rows = 0
x = 0

if self.use_optimized_task:
    num_tasks, remainder = divmod(self.width, self.factor)
    self.jobs = num_tasks + remainder

    for i in xrange(0, self.jobs):
        if self.send_task_batch:
            job_list.append(MandlebrotTaskOptimized("m", None, self.node_id_str,
                iters = self.iters, start_x = i, rows = self.factor, cols = self.height,
                pixel_size_x = self.pixel_size_x, pixel_size_y = self.pixel_size_y,
                min_x = self.min_x, min_y = self.min_y))
        else:
            self.push_task(MandlebrotTaskOptimized("m", None, self.node_id_str,
                iters = self.iters, start_x = i, rows = self.factor, cols = self.height,
                pixel_size_x = self.pixel_size_x, pixel_size_y = self.pixel_size_y,
                min_x = self.min_x, min_y = self.min_y))
    else:
        for x in range(self.width):
            # Distribute using rows
            rows += 1
```

```

    real = self.min_x + x * self.pixel_size_x
    for y in range(self.height):
        imag = self.min_y + y * self.pixel_size_y
        workload.append((x, y, real, imag, self.itsers))

        # every self.factor rows create a task with the workload. Note that in this case we will
        # force the system_id to be None while setting the client id
        if rows == self.factor:
            if self.send_task_batch:
                job_list.append(MandlebrotTask("mandle_{}".format(x), None, self.node_id_str,
                itsers = self.itsers, workload = workload))
            else:
                self.push_task(MandlebrotTask("mandle_{}".format(x), None, self.node_id_str,
                itsers = self.itsers, workload = workload))
                self.jobs += 1
                workload = []
                rows = 0

        # Add last task with rest of workload
        if len(workload) > 0:
            if self.send_task_batch:
                job_list.append(MandlebrotTask("mandle_{}".format(x), None, self.node_id_str, itsers
                = self.itsers, workload = workload))
            else:
                self.push_task(MandlebrotTask("mandle_{}".format(x), None, self.node_id_str, itsers =
                self.itsers, workload = workload))
                self.jobs += 1

        if self.send_task_batch:
            self.jobs = len(job_list)

        # Send batch or check for eventual end condition
        if self.send_task_batch:
            self.push_tasks(job_list)
        else:
            # Check in case we are already done!
            self.check_finished()

def task_finished(self, task, result, error):
    """
    Called when a task has been done
    """
    # Integrate results in our image
    if result:
        for x, column in result.iteritems():
            for y, value in column.iteritems():
                self.image[y, x] = value

    self.finished_jobs += 1
    self.check_finished()

```



```

def check_finished(self):
    """
    Check if we finished all computation or not
    """
    if self.finished_jobs == self.jobs:
        self.log.info("All tasks finished!!")
        print("Calculated in {} seconds!".format(time.time() - self.start_time))
        self.shutdown_main_loop()
        imshow(self.image)
        show()

def push_task_response(self, result):
    """
    We just add a Task to the computation framework
    """
    pass
    #self.log.info("Task send to computation framework")

def push_task_failed(self, result):
    """
    We failed to add a Task to the computation framework
    """
    self.log.info("Failed to send task send to computation framework")

def push_tasks_response(self, result):
    """
    We just add a set of Tasks to the computation framework
    """
    self.log.info("Tasks send to computation framework")

def push_tasks_failed(self, result):
    """
    We failed to add a set of Tasks to the computation framework
    """
    self.log.info("Failed to send tasks send to computation framework")

```

Code X – Application using plain tasks and task batches

```

def do_mandel(x, y, max_iters):
    """
    Given the real and imaginary parts of a complex number,
    determine if it is a candidate for membership in the Mandelbrot
    set given a fixed number of iterations.
    """
    c = complex(x, y)
    z = 0.0j
    for i in range(max_iters):
        z = z*z + c
        if (z.real*z.real + z.imag*z.imag) >= 4:

```

```

        return i

    return max_iters

class MandlebrotTask(Task):

    def __init__(self, name, system_id, client_id, **kwargs):
        Task.__init__(self, name, system_id, client_id)
        self.workload = kwargs['workload']
        self.itsers = kwargs['itsers']

    def __call__(self):
        """
        Calculate assigned work
        """
        result = {}
        for work in self.workload:
            if not work[0] in result:
                result[work[0]] = {}
            result[work[0]][work[1]] = do_mandel(work[2], work[3], self.itsers)
        return result

    def finished(self, result, error):
        """
        Once the task is finished. Called on the MasterNode within the main thread once
        the node has recovered the result data.
        """
        pass

    def clean_up(self):
        """
        Called once a task has been performed and its results are about to be sent back. This is used
        to optimize our network and to cleanup the tasks input data
        """
        self.workload = None
        self.itsers = None

class MandlebrotTaskOptimized(Task):

    def __init__(self, name, system_id, client_id, **kwargs):
        Task.__init__(self, name, system_id, client_id)
        self.start_x = kwargs['start_x']
        self.rows = kwargs['rows']
        self.cols = kwargs['cols']
        self.min_y = kwargs['min_y']
        self.min_x = kwargs['min_x']
        self.pixel_size_y = kwargs['pixel_size_y']
        self.pixel_size_x = kwargs['pixel_size_x']
        self.itsers = kwargs['itsers']

```

```

def __call__(self):
    """
    Calculate assigned work
    """
    result = {}
    for x in xrange(self.start_x, self.start_x+self.rows):
        real = self.min_x + x * self.pixel_size_x
        for y in xrange(0, self.cols):
            if not x in result:
                result[x] = {}
            imag = self.min_y + y * self.pixel_size_y
            result[x][y] = do_mandel(real, imag, self.itsers)
    return result

def finished(self, result, error):
    """
    Once the task is finished. Called on the MasterNode within the main thread once
    the node has recovered the result data.
    """
    pass

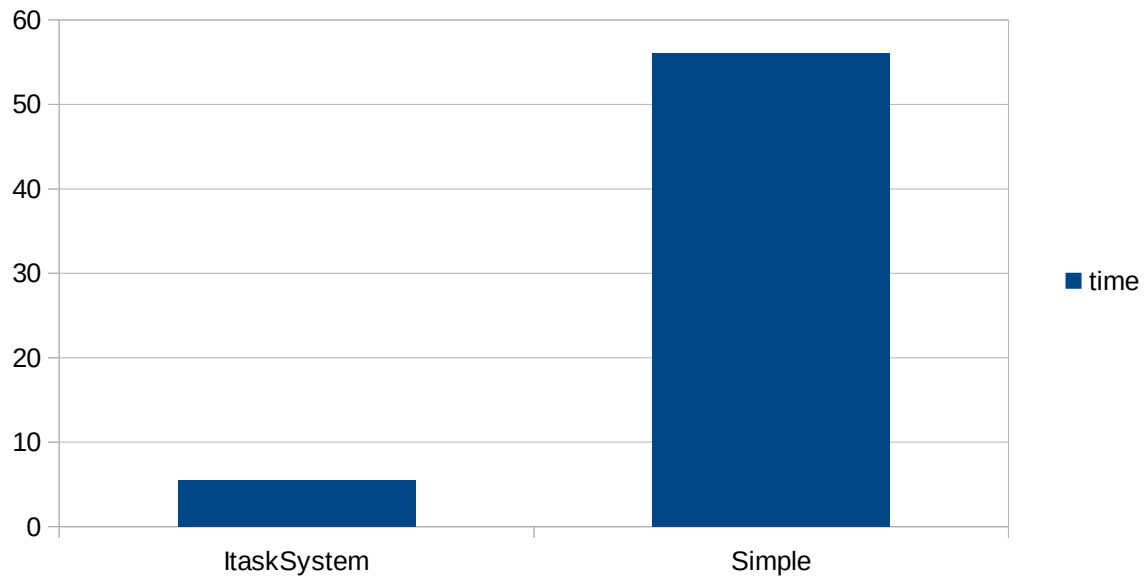
def clean_up(self):
    """
    Called once a task has been performed and its results are about to be sent back. This is used
    to optimize our network and to cleanup the tasks input data
    """
    self.start_x = None
    self.rows = None
    self.cols = None
    self.min_y = None
    self.pixel_size_y = None
    self.itsers = None

```

Code X – Mandlebrot task, optimized and non-optimized

ITaskSystem vs plain tasks

Viewing both implementations we see that the one using the task system is a bit more complex while the plain tasks implementation is more like a traditional task system. Chat X shows the results of 50 runs of the sample using the default configuration of the application using both approaches



What we see from the benchmark is clear, sending an application to the master which send creates the tasks performs better then sending the tasks on their own. The reason is that we avoid two round-trips per tasks, we do not have to send them to the master and then sending them back. Another reason is that once we are over 100 messages per second the network starts to stutter.

Net congestion and workloads

The sample application is a very good sample testing network conditions and we see that the time to send the tasks itself is by far bigger then the time spend on the calculation. This gives us insights of what type of computations are useful on a distributed tasks system and that the key for implementing a fast implementation is basically enforcing locality of the data.

Once data usage has been optimized we can see that both ways of performing the work are actually balanced. In the optimized version of the mandelbrot set we generate the required values for the computation within the actual tasks and not from the outside. Figure X show the benchmark outputs using the optimized task implementation.

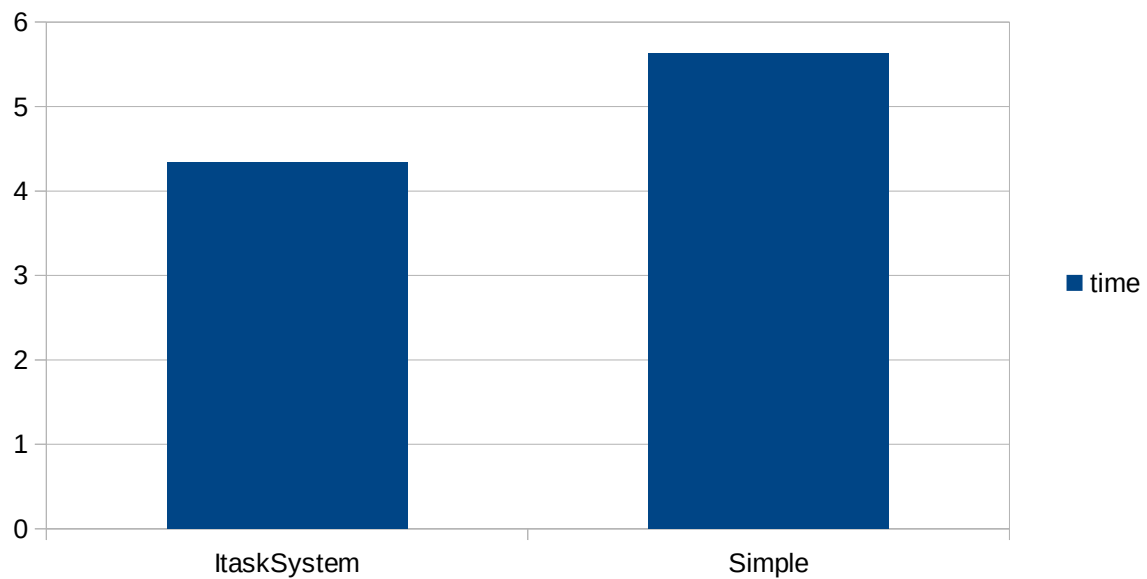


Figure X – Optimized mandelbrot tasks implementation benchmark

DNA Curve analysis

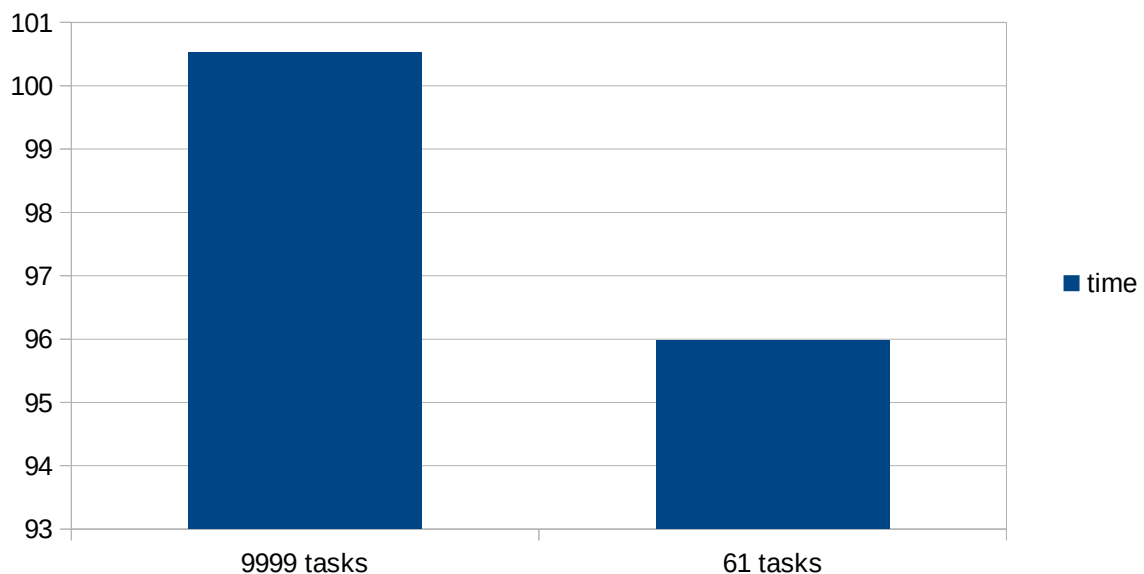
The DNA sample is just a simple way to try to overload the system with over 10 thousand separate tasks. Each task requires a considerable amount of data and so sending it all at once has its drawbacks.

The tests and benchmarks has been performed on a single machine and so we setup the worst possible distributed network resulting in network overloads. This the sample is not intended for real use, it outlines the stability of the framework.

The code of the sample can be reviewed online, we will not add it to the documentation because of its size.

Net congestion and workloads

As for the other samples the amount of data send through the network is considerable, the sample itself reaches a high memory load up to 3 GB on the master.



Consideration on data flow

Sending huge amounts of data is the real bottleneck of any distributed task framework. We spend more time sending data then performing the actual computation.

Conclusions

Building concurrent gave us a broad view of how much effort is required to produce a usable and stable distributed tasks system. Concurrent is not to be considered finished, it is in its first steps and will evolve in time to fill the gap in the space of reusable and easy to use concurrent tasks systems.

Python has proven itself to be a perfect match for both feasibility and performance using native modules such as Cython. Its ability to serialize using pickles and networking libraries such as ZeroMQ gave us the power to achieve a stable and well performing system. There is still room for both optimization and features to make the framework even more user friendly.

Creating or adapting application to be run on concurrent has been a very easy tasks due to the way we create and organize the workload.

Once point where we still have a lot of work to do is the network side of the framework. Sending huge amounts of data, specially over the MTU size, impacts the global performance severely. Data synchronization and initial setup is an important way to lower the amount of data send over the network. Using a distributed file system would also help on leveraging the data flow. The locality of the required is yet another optimization on the data usage side that has to be considered for future releases.

On the performance side we can verify that using batching and our proposed ITaskSystem is performing better over sending plain tasks on the system. This is mainly due to the lack of a considerable number of round-trips the tasks are saving. Sending a whole system over lets us save a lot of those small sendings the same as for a batch of tasks. The difference between sending a system over and a batch of tasks is that the return of results for a batch of tasks does not feature that round-trip saving because we do not collect the whole results on the master, as we do for the ITaskSystem. While the ITaskSystem does the return saving also it comes with an increased of memory used by the master to hold the intermediate results until the system finishes.

Amdahl's Law is another consideration that we have to talk about in concurrent systems. In our system the key part is the proportion of concurrent parts of an application that we are able to use. That proportion depends totally on how the application developer creates their applications. A good example is the Mandelbrot sample where we where able to achieve a higher proportion of parallelism by simply doing more within each tasks. We where able to split each row within a single tasks and so we ended up of a proportion of the width over the number of pixels. Using a task for each pixel would be the highest proportion of parallelism we could possibly achieve and so having.

The final speed up possible for the Mandelbrot sample on an 8 core setup would then be:

$$2.4 = 1 / ((1 - P) + (P / N)) \text{ where } P = 0.67; N = 8$$

Seeing the amount of maximum improvement over the sequential version that we actually had using our testing machine, the improvement where about by 2, we verify that concurrent is performing very well.

The benchmark example has shown that we had achieved an improvement by 7, applying Amdahl's Law we are can achieve a maximum improvement of 8 considering the overhead of the framework more than acceptable for its first release cycle.

$$8 = 1 / ((1 - P) + (P / N)) \text{ where } P = 1; N = 8$$

Results

Now that we have spoken about what *Concurrent* is all about we have come to a point where we can describe what are the expected results of this research project:

- Framework eliminating GIL issues where possible
- SDK to setup a computational node and to create concurrent systems to be executed on the framework
- Example project using the SDK (data analysis) to demonstrate scientific calculations such as data analysis algorithms, brute force cracking examples, benchmarks, DNA analysis and a Mandelbrot sample outlining different ways to optimize an application run on concurrent.

The exposed sample application gives an overview on how to use Concurrent and how your code can get executed on a wide range of clusters and remote machines. We have achieve a flexible yet stable parallel computation framework that goes a step further then simple function and local processing.

The main code is spread to many servers which requires the programmer to change its mind about local data and shared data, we are no longer able to just read everything from every where. The proposed programming model requires are more planned and strict work-flow to perform as expected. Debugging tools and real-time monitoring tools help in the development of fully concurrent systems.

Appendix A

Gantt project plan

The following Gantt project plan is our main planning mechanism. Due to the fact that we have chosen a mixed methodology our Gantt will track the execution of all iterations and its WBSs. While the project progresses tasks may flow from one Iteration to another and so WBS will in turn spread over Iterations. After every Iteration meeting the project plan is updated with the latest changes and the scope/tasks will get adapted to keep the initial schedule.

As you can see in figure X and X the initial project plan does cover all aspects, while no WBS has been added to them we will update the plan on a weekly basis to add which part of the WBS has been performed during which iteration. At the end of the project each WBS and its stories will then be assigned to a specific iteration to revise the work flow. Tasks that have already been achieved has been highlighted in blue.

	🕒	Name	...	Start	Finish	P...	Resource Names
1		☐ Exploration	...	2/25/14 8:00 AM	3/17/14 5:00 PM		
2		☐ Spike	...	2/25/14 8:00 AM	3/6/14 5:00 PM		
3		Methodolgy adaption	...	2/25/14 8:00 AM	2/27/14 5:00 PM		Moritz Wundke
4		Sphinx Documentation	...	2/28/14 8:00 AM	2/28/14 5:00 PM	3	Moritz Wundke
5		UnitTest Project	...	2/28/14 8:00 AM	2/28/14 5:00 PM	3	Moritz Wundke
6		Prototypes	...	2/25/14 8:00 AM	3/6/14 5:00 PM		Moritz Wundke
7		☐ Planning	...	3/7/14 8:00 AM	3/17/14 5:00 PM	2	
8		Create WBS	...	3/7/14 8:00 AM	3/7/14 5:00 PM		Moritz Wundke
9		Create User Stories	...	3/10/14 8:00 AM	3/11/14 5:00 PM	8	Moritz Wundke
10		Create Iteration Plan	...	3/12/14 8:00 AM	3/13/14 5:00 PM	9	Moritz Wundke
11		Create Project Plan	...	3/14/14 8:00 AM	3/17/14 5:00 PM	10	Moritz Wundke
12		☐ Iterations	...	3/18/14 8:00 AM	6/20/14 10:00 AM	1	
13		Iteration 1	...	3/18/14 8:00 AM	3/31/14 3:00 PM		Moritz Wundke
14		Iteration 2	...	3/31/14 3:00 PM	4/14/14 1:00 PM	13	Moritz Wundke
15		Iteration 3	...	4/14/14 1:00 PM	4/28/14 10:00 AM	14	Moritz Wundke
16		Iteration 4	...	4/28/14 10:00 AM	5/9/14 5:00 PM	15	Moritz Wundke
17		Iteration 5	...	5/12/14 8:00 AM	5/23/14 3:00 PM	16	Moritz Wundke
18		Iteration 6	...	5/23/14 3:00 PM	6/6/14 1:00 PM	17	Moritz Wundke
19		Iteration 7	...	6/6/14 1:00 PM	6/20/14 10:00 AM	18	Moritz Wundke
20		☐ Project Corrections	...	6/20/14 10:00 AM	6/25/14 5:00 PM	12	
21		Presentation	...	6/20/14 10:00 AM	6/25/14 5:00 PM		Moritz Wundke
22		Video Presentation	...	6/20/14 10:00 AM	6/25/14 5:00 PM		Moritz Wundke
23		SDK Documentation	...	6/20/14 10:00 AM	6/25/14 5:00 PM		Moritz Wundke

Fig. 16 – Gantt initial tasks organization

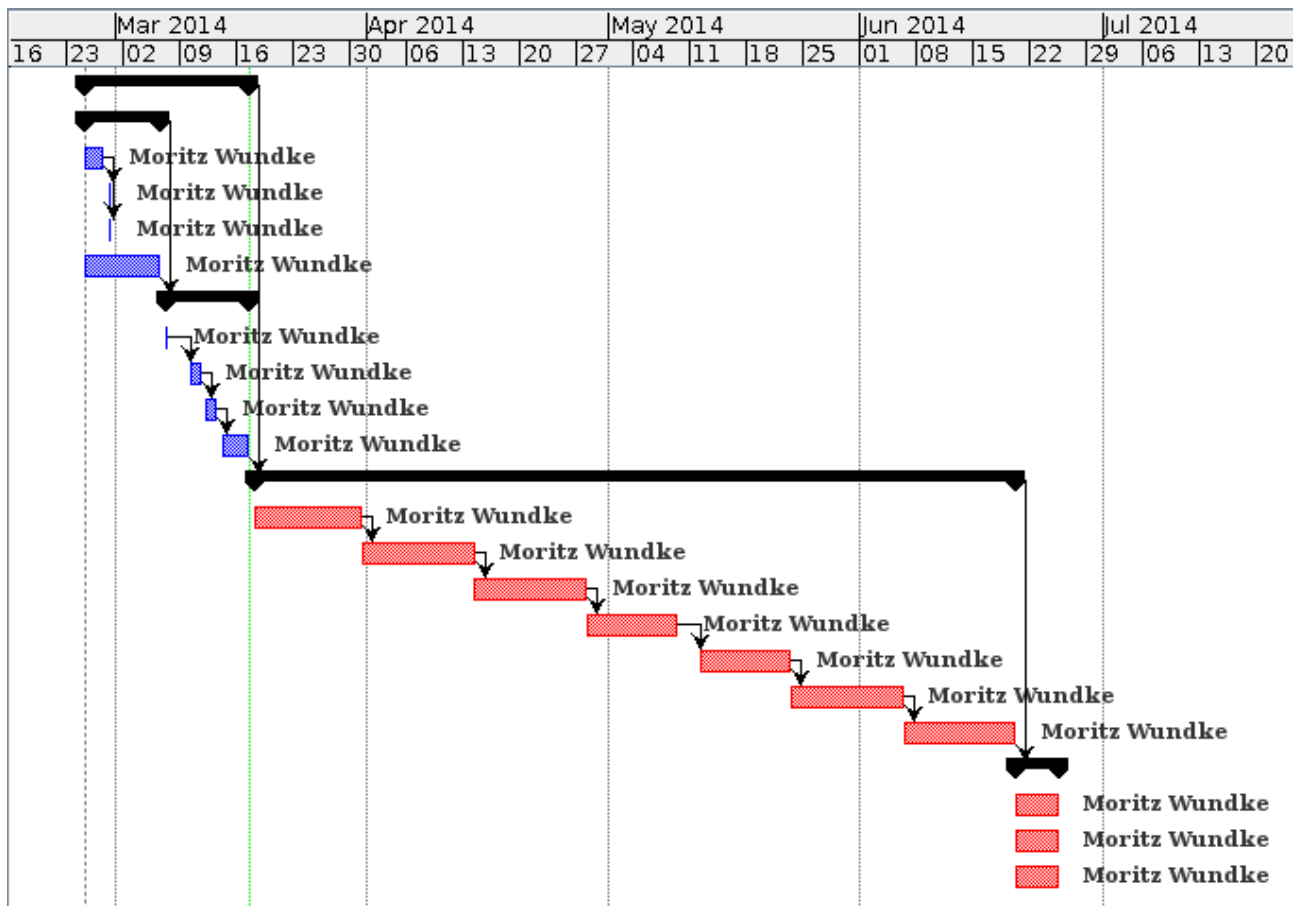


Fig. 17 – Gantt initial tasks organization (cont.)

Appendix B

Project Page

The project is hosted on GitHub. For more info check the projects page: <http://moritz-wundke.github.io/Concurrent/>

API

The API for version 0.1.1 can be found in the projects GitHub hosted page: <http://moritz-wundke.github.io/Concurrent/API/>

Slides

Presentation video and slides can be found hosted on GitHub aswell: <http://moritz-wundke.github.io/Concurrent/slides/>

References

1. SETI@Home: <http://en.wikipedia.org/wiki/SETI@home>
2. SciPy: <http://www.scipy.org/stackspec.html>
3. NumPy: <http://www.numpy.org/>
4. Pandas: <http://pandas.pydata.org/>
5. Matplotlib: <http://matplotlib.org/>
6. Pythons GIL: <https://wiki.python.org/moin/GlobalInterpreterLock>
7. Jython concurrency: <http://www.jython.org/jythonbook/en/1.0/Concurrency.html>
8. IronPython: <https://wiki.python.org/moin/IronPython>
9. Cython: <http://cython.org/>
10. OpenCL: <https://www.khronos.org/opencv/>
11. OpenMP: <http://openmp.org/>
12. Extreme Programming: <http://www.extremeprogramming.org/>
13. ProjectLibre: <http://www.projectlibre.org/>
14. Trello: <https://trello.com/> / <http://www.nullpaperexception.com/2013/trello/>
15. ZeroMQ: <http://zeromq.org/>
16. Google File System: http://en.wikipedia.org/wiki/Google_File_System