Platform Independent Cloud Scripting

Kelly Smith
Advisor: Dr. Deborah Whitfield
Department of Computer Science
Slippery Rock University

ABSTRACT

Previous work in minimizing the inherent difficulties of parallel programming has yielded three popular methods for automated, scaled parallelization: OpenMP, MPI, and CUDA. This research borrows much of its philosophy from these methods, adds parallel language constructs to JavaScript, and provides a general-purpose framework for spawning algorithms to a heterogeneous, platform-independent cloud of processing entities including mobile devices. A prototype has been successfully created from scratch, and currently the cloud consists of clients on the following platforms: Windows, Linux, OSX, Android, and any modern web browser. Initial results have been encouraging, and this framework has proven to be effective in bringing parallel computing to untapped computing resources.

1. PROBLEM AND MOTIVATION

In recent years it has become well understood that Moore's promised exponential growth in computational power can only continue if there is a shift in the fundamental design of software to exploit parallelism [7]. Previous work in minimizing the inherent difficulties of parallel programming has yielded three popular methods for automated, scaled parallelization: OpenMP, MPI, and CUDA. Each method has its respective strengths and weaknesses, but each is generally restricted to conventional computing devices (i.e. Desktops and laptops). In the same way that scripting and high-level languages represent a natural progression towards ease of development and platform independence, so too does this research represent a progression from these other parallelization methods to a Platform Independent Cloud Scripting (PICS) method.

Much of the motivation for this research stems from three primary realizations which form the philosophy by which this project has been created. The first motivation is that in recent years, mobile phone sales have surpassed conventional computing devices [8]. As smartphones and other mobile devices become more powerful, it is apparent that a large pool of untapped computing resources exists, and will continue to exist with increasing prominence as mobile devices continue to crowd the computing market share.

Next, JavaScript as a language has become so ubiquitous, that it has been called the assembly language of the web [3]. In fact, in recent years, one would be hard pressed to find a mainstream, personal computing device that does not include some sort of JavaScript interpreter with its software. A general purpose parallel platform could make use of the widespread availability of JavaScript to gain access to much of the untapped computing resources previously mentioned.

Finally, this research makes the assumption that available, inexpensive, and powerful sources of computing are desirable. This research recognizes that the existing general purpose parallel frameworks are effective relative to their particular areas of expertise, and seeks to provide an alternative, open source and platform independent solution. Another implication of this assumption is that it disregards any societal and financial impacts. PICS has been designed and implemented as a tool, and while this research has identified several areas to which PICS would be well suited, no particular real-world implications have been identified such as the effect on energy consumption per unit of performance.

2. BACKGROUND AND RELATED WORK

At the time that this research was conceived and implemented, three primary sources of inspiration can be identified: OpenMP, MPI, and nVidia's CUDA. Recently, an additional related work has been identified.

OpenMP The syntax and functionality of OpenMP served as the basis for much of the syntax and 'feel' of PICS [1]. It has been found that this style of syntax is flexible and capable for all of the functional needs of PICS. It is important to note however, that OpenMP is designed to run on a shared memory platform, while PICS runs on a distributed memory platform. While this does not necessarily effect the adoption of OpenMP syntax and structure, it does effect the implementation.

MPI While OpenMP has served as inspiration for much of the syntax of PICS, it is believed that MPI is a closer fit to the actual functionality of PICS. As a result, this research

[1]It is important to note that the decision to utilize a similar syntax structure to OpenMP was a stylistic decision and was not produced using a HCI study.

SIGCSE'12, February 28–March 4 2012, Raleigh, North Carolina.

has come to the conclusion that many of the same challenges faced by MPI are also shared by PICS. It follows logically then that many of the solutions that MPI has implemented to overcome said problems would also lend themselves well to PICS. These problems include dealing with distributed memory, dealing with network latency and limitations, and load balancing. Load balancing is a particularly interesting problem for PICS and is discussed in Section 4.2.

nVidia CUDA Initially it was believed that CUDA could be an excellent platform to draw inspiration from as its structure was similar to the perceived structure of PICS at the time. An example of this is the ability of CUDA to easily scale to thousands of processing units while dealing with distributed memory problems [2]. It was also believed that the concepts of 'kernels' and 'blocks' used by CUDA would easily translate well[6]. In the end however, it was found that OpenMP and MPI served as much better examples of what it means to be a successful parallel platform given the design that was eventually selected for PICS.

Intel Parallel Javascript In late 2011, Intel released a project for executing Javascript in parallel directly inside of the web browser [4]. By this point in time, PICS was already designed and implemented, and Intel's platform played no role in the inspiration for PICS. That said, it is certainly an interesting project which is striving to solve some of the same problems that PICS is.

3. APPROACH AND UNIQUENESS

3.1 Initial Design

The task of creating a platform-independent framework capable of automated, scalable parallelism has no straightfoward answer. In the early design stages of this research, many potential ideas were considered including: compiling high level language(s) to a universal bytecode, using a scripting language to perform nearly the same function, and adapting MPI to function on mobile devices. In the end, it was determined that the desireable solution must meet the following criteria:

- It must be truely platform indpendent and work on both mobile and conventional[3] computing devices simultaneously[4].
- It must be able to be rapidly prototyped. This was necessary because the sole author of this paper was also the sole developer of the project.
- It must utilize intutitive syntax and run on a lightweight, platform-indpendent backend. The purpose for this criterion was to allow for PICS to easily and quickly be deployed in a variety of different applications discussed in Section 4.1.

[2]CUDA uses a hierarchical system of processing units called threads and blocks where only threads in the same block can access the same memory

[3]Conventional computing devices refers to standard desktop and laptop configurations, and the popular associated operating systems.

[4]This has significant implications which are discussed in Section 4.2

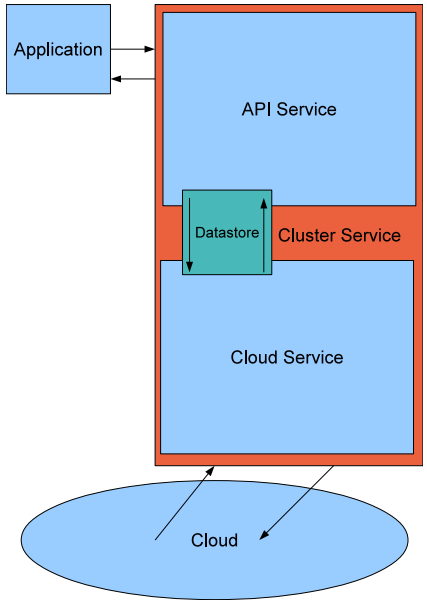

Figure 1: Initial Design

At this stage, many different designs were also created to determine the best way to tie all three of the criteria together to solve the problem of platform indpendent automated parallelism. In the end it was determined that the design shown in Figure 1 was practical, powerful, and flexible enough to meet each of the above criteria.

3.2 Prototype

The development of the PICS prototype was an ongoing process with the design constantly being updated as various approaches either worked or did not. This process can essentially be divided into five primary problems:

3.2.1 Executing Arbitrary Javascript

The success or failure of this research depended entirely on the ability to execute arbitrary Javascript on each desired platform and generate meaningful answers. After searching for possible solutions to this problem, it was determined that three primary methods were available:

- Google's v8 Javascript Engine
- Mozilla's Rhino
- Javascript interpreters built into browsers

Google's v8 engine[2] has been used in conjunction with Node.js to execute Javascript directly on conventional computing platforms including Windows, OSX, and Linux. v8 has also been used with PyV8[9] to execute Javascript on the cloud server as is described in Section 3.2.4. Mozilla's Rhino was used in conjunction with Android to execute Javascript directly inside of an Android application. Unfortunately, the performance of Rhino is lacking, and a native Android solution using Node.js still needs to be developed. Finally, a solution for executing Javascript directly within a browser is a well-defined process and has been leveraged to work in the PICS cloud using asynchronous requests to a web server.

3.2.2 Develop An Augmented Javascript Syntax

```
1  /* Approximate Pi */
2  // Initialize variables
3  var a = 0;
4  var b = 1;
5  var N = 500000000; // 500 million
       partitions
6  var area = 0;
7  var dx = (b-a)/N;
8  //@pics pllfor out=area in=a,dx,area
       reduction=area:+
9  //#
10 for(var i=0; i<N; i++){
11 var xi = a + i*dx;
12 area = area + (4/(1+( xi * xi )));
13 }
14 //#
15 area = area/N;
16 // report answer
17 JSON.stringify(area);
```

Listing 1: Example PICS Script

Lines 8 through 14 above are examples of the parallel langauage constructs that have been added to Javascript to allow for the specification of parallel behavior. This script approximates Pi using iterative integration with a parallel for loop. The parallel directive *//@pics pllfor* borrows much of its syntax and functionality from OpenMP. The functionality of each parameter that has been implemented is described bellow:

- **pllfor** - Specifies a parallel for loop. Also supported: single and parallel
- **in, out, inout** - Set to a comma-separated list of variables and specifies how variables persist in and out of the parallel section of code.
- **reduction=var:operator** - Specifies that *var* should persist out of the parallel section of code as a value reduced from each of the parallel 'units' using the specified *operator*. Supported operators: +, -, *.
- **default=in|out|inout** - Specifies a default setting for variable persistance.
- **chunks=*integer*** - For the parallel directive. Specifies how many parallel units should be spawned to cloud[5].
- **threadId=*identifier*** - For the parallel directive. Specifies the name of the variable which holds the identifier for each individual parallel unit.

3.2.3 Parsing PICS Scripts

The next stage in this research was to develop a system for parsing augmented Javascript syntax so that it can be compiled and then spawned out to the cloud. As a result, Algorithm 1 was developed, and the result is shown in Figure 2.

In summary, Algorithm 1 parses the PICS script and separates the file into blocks of sequential and parallel code. This can be observed in the transformation from the code

[5]There are future plans to add an automatic setting for this to work in conjunction with dynamic load balancing.

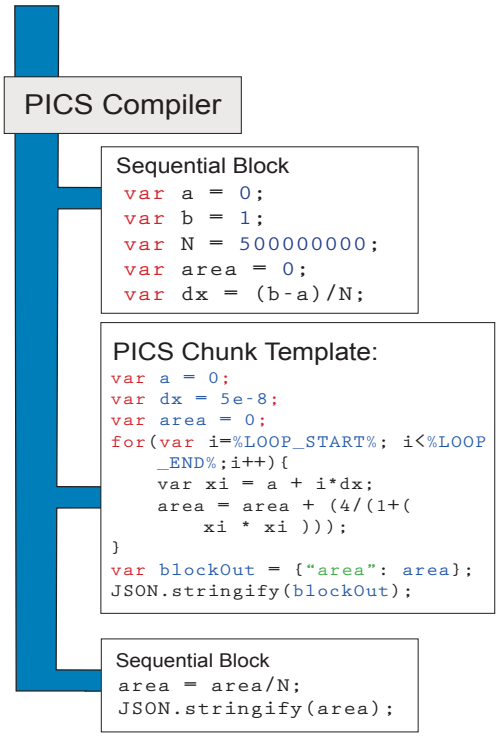

Figure 2: Parsed and Compiled PICS Script

in Listing 1 to the parsed blocks in Figure Figure 2. It then loops through each parallel block and parses each of the arguments, as detailed in Section 3.2.2, and creates a list of all of the identifiers found throughout the block for future use. Finally, the algorithm stores all of the PICS block data to the datastore so that it can be accessed later and compiled by the cloud service.

Algorithm 1 Parse a PICS Script

Require: removeComments()
  $picsblocks, seqblocks \leftarrow detectPICS()$
  if $\neg picsblocks$ OR $\neg seqblocks$ **then** error()
  end if
  store(*seqblocks*)
  for all *block* in *picsblocks* **do**
    $args \leftarrow$ parseArgs(*block*)
    $ids \leftarrow$ parseIdentifiers(*block*)
    compile(*block*, *args*, *ids*)
    sore(*block*)
  end for

3.2.4 Construction of a Cloud Service

The server responsible for interfacing with the cloud was created using Python to create a light weight socket server. Django[1] was used to handle access to the MySQL datastore. The purpose of the cloud service is to generate and process **PICS Chunks** which are executed on the cloud and then returned to the server. A PICS chunk is code generated from a PICS parallel block of code which is compiled into a form that can be directly executed by a Javascript interpreter to produce the intended results. An example of a PICS chunk can be found in Figure 2. This process is

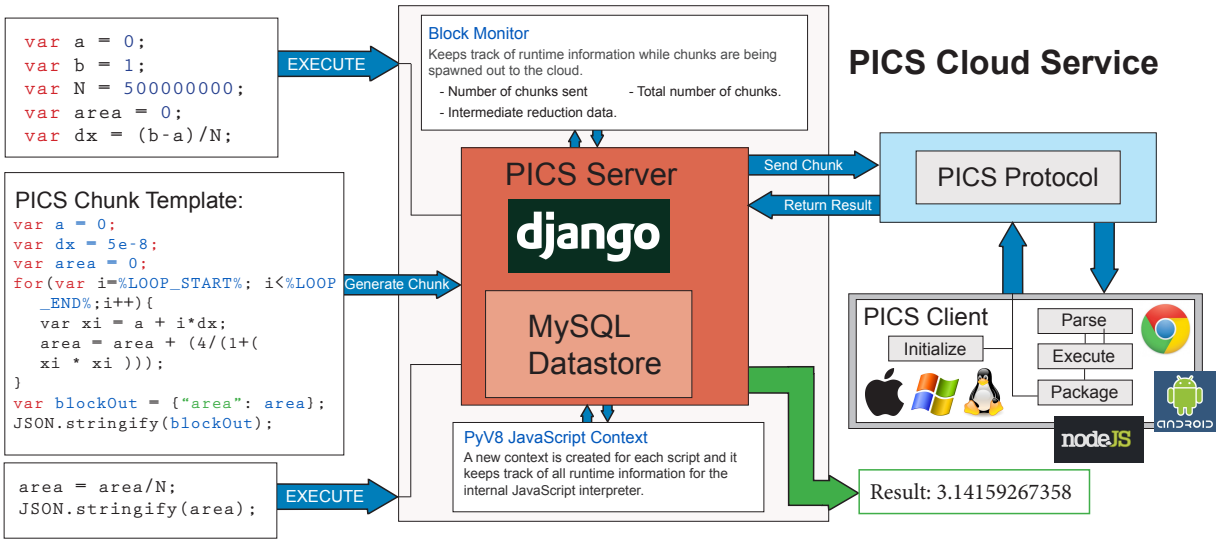

Figure 3: PICS Cloud Service Design

described in full in Section 3.2.4.

The functionality of the cloud service is one of the primary ways that PICS differentiates itself from other existing parallel platforms. Unlike MPI, OpenMP, and CUDA, PICS uses a centralized server to organize and maintain state throughout the execution of a parallel block of code. The implications of functioning this way are numerous, but one of the greatest benefits is that it allows PICS to be truly platform and location independent. For instance, a laptop in Germany, a desktop in Australia, and a tablet pc in China could all three be simultaneously connected to the same PICS cloud contributing to the same problem.

Construction of Socket Server. It was determined that a light-weight Python socket server would be optimal to use for the cloud service because python is ideal for rapid prototyping and PyV8 allows for arbitrary execution of Javascript within a Python runtime environment. Figure 3 shows a general design of the PICS cloud service.

Handling Client Requests. The PICS cloud services accepts incoming client requests on a first come first serve basis. The process of handling a request can be divided into three classifications:

New Client. New Request. This classification deals with clients which have just connected to the cloud for the first time. In this case, the server skips ahead to the process of chunk compilation, and proceeds to send a chunk to the client for execution.

Existing Client. New Request. In this case, a client which has previously been given a chunk to execute is returning the results of that execution. At this point, the server processes the execution results and updates the global Javascript context associated with the PICS script by generating code and executing it using the server-side Javascript interpreter. In the case of the integration example used throughout this paper, the code that would be generated is: $area = 1570796336.79;$.

At this point, the server also uses a data structure which this research calls a **Block Monitor** to keep track of all of the various intermediate data associated with the execution of a PICS script including: intermediate reduction data, the number of chunks sent out, and the number of chunks remaining. If a chunk is returned to the server that comes from a block utilizing a reduction on a variable, the server will determine the intermediate value for the variable, apply the reduction operation to it, and store the result in the block monitor. Finally, the server proceeds to compile and prepare another chunk, which it sends back to the same client to continue execution.

Existing Client. End Request. In this case, an existing client is returning the results of execution, but does not wish to receive another chunk for execution. The server simply follows the same procedure for processing the execution results as before, but does not prepare a new chunk for the client.

PICS Protocol. The PICS protocol is a light-weight, text-based protocol designed for the transfer of PICS chunks from server to client and back again. Below is an example of the format used by the protocol:

$<string\_size>JSON\_OBJECT$

Where *string_size* is the length of the transmission in bytes, and JSON_OBJECT is a JSON encoded string containing all of the information necessary for the client to execute the chunk and return the proper results. Below are several

examples of the protocol in action.

Server Responses

<567>{"uuid": "ea7d4cc4-0158-490a-8fe7-76486c62ceaf", "jsId": 33, "blockId": 33, "chunkId": 1, "threadId": 33, "type": "block", "block": **see Compilation of Chunks**}

<76>{"code": 1, "type": "error", "uuid": "aacfc628-7bc8-46f1-a27a-dd78349d98dc"};

Client Requests

<117>{"action":"new_request", "client":"node", "timeout":2000, "picsRating":500, "uuid":"*omitted*"}

<298>{"action":"return_request","client":"node","jsId":33, "blockId":33, "threadId":33, "block":" {"i":20000000,"N":20000000, "xi":0.99999995,"a":0, "dx":5e-8, "area":25740044.95173052}", "timeout":2000,"picsRating":500,"uuid":"*omitted*","chunkId":2, "execution_time":1391}

Of the attributes of the JSON object, *picsRating* is of particular interest. Currently this attribute is ignored and static, however it has been included for future use with dynamic load balancing which is described in Section 4.2. Due to the size constraints placed on this paper, a more in depth description of the protocol is not possible.

Compilation of Chunks. The compilation of PICS chunks occurs in two stages. The first stage occurs when a PICS script is uploaded to the cloud service. At this stage, a **Chunk Template** is created. A chunk template is simply a pics chunk with placeholders for certain values which vary between each parallel unit that will be spawned out. The compilation process involves determining which variables mush persist into the parallel block from the global context and generating javascript code to determine these values. These values are then inserted into the top of the PICS chunk as a way of simulating the persistence of the variables. Additionally, code is generated to package up variables that must persist out of the block. An example of this code being generated is shown in Figure 2.

3.2.5 Construction of a Client

A PICS client is any application which conforms to the PICS protocol and contains a Javascript interpreter for executing PICS chunks. Currently clients have been developed on the following platforms: OSX, Windows, Linux, Android, and any modern web browser.

4. RESULTS AND CONTRIBUTIONS

The primary result of this research has been the creation of a fully functional, general purpose framework for spawning out parallel code to a heterogeneous, platform-independent cloud of processing entities. This research is significant because it creates a unique way to perform parallel calculations, and is the only known way to utilize mobile devices for performing calculations in parallel.

4.1 Possible Applications

Education - Educators could easily turn a computer lab (which the institution already needed) into a way to teach the principles and pitfalls of parallel and distributed programming to students. The major advantage to using PICS over other existing solutions is that it can be set up in minutes to run on any number of devices and requires almost no investment in hardware over what is already available to most educators.

Business - Businesses that purchase smartphones for their employees could use the phones as additional general purpose computing resources while they are plugged in and charging at night.

Scientific Research - Similar to Stanford's "Folding@home" project, PICS could allow scientific researchers to spawn out computationally intensive algorithms to a cloud of willing participants to further their research goals.

4.2 Future Work

Dynamic Load Balancing - Implement a system to dynamically and intelligently spawn correctly sized chunks based the state of the cloud at any given point. This is important because a heterogenous cloud will be much less efficient without it due to the imbalanced of performance. Giving identically sized portions of work to each client will inevitably cause clients to spend time idle. An entire paper could be written on this topic, however the scope of this paper does not warrant further discussion.

Native Android Client - As was mentioned earlier, the current Android client using Rhino is too slow to be of any use. Future work could be done to port Node.js to Android to natively run Google's V8 engine for increased performance.

Compiler Optimizations - This research theorizes that existing compiler/interpreter optimizations will be necessarily disrupted by PICS due to its distributed nature and utilization of multiple interpreters at the same time. It is believed that existing and new compile-time and run-time optimizations could be implemented in PICS.

Intelligent Data Dependence - One of the major drawbacks that PICS experiences is due to the relatively slow network connections and latency inherent to cloud computing when compared to MPI, OpenMP, and CUDA. An implication of this is that PICS will not perform very well on problems dealing with large data sets due to the inefficiency of copying the data to each client. Existing methods[5] for data dependence analysis exist and could potentially be implemented in part or in whole on the PICS platform.

5. REFERENCES

[1] Foundation, D. S. Django. `https://www.djangoproject.com/`, 2012.

[2] Google. V8 javascript engine. `http://code.google.com/p/v8/`, 2011.

[3] Hanselman, S., and Meijer, E. Javascript is assembly language for the web: Semantic markup is dead! clean vs. machine-coded html. `http://www.hanselminutes.com/`, July 2011.

[4] Herhut, S. Building a computing highway for web applications. http://blogs.intel.com/research/2011/09/15/pjs/, September 2011.

[5] Maydan, D. E., Hennessy, J. L., and Lam, M. S. Efficient and exact data dependence analysis. *SIGPLAN Not. 26*, 6 (May 1991), 1–14.

[6] Ruetsch, G., and Oster, B. Getting started with cuda. http://www.nvidia.com/content/cudazone/download/Getting_Started_w_CUDA_Training_NVISION08.pdf, 2008.

[7] Sutter, H. A fundamental turn toward concurrency in software. *Dr. Dobbs* (March 2005).

[8] Tofel, K. C. 5 biggest losers as smartphone sales surpass pcs. http://gigaom.com/, January 2011.

[9] unlisted. pyv9: Python wrapper for google v8 javascript engine. http://code.google.com/p/v8/, 2011.