CPU Torrent – CPU Cycle Offloading to Reduce User Wait Time and Provider Resource Requirements

Swapneel Sheth, Gail Kaiser
Department of Computer Science, Columbia University, New York, NY 10027
{swapneel, kaiser}@cs.columbia.edu

Abstract

Developers of novel scientific computing systems are often eager to make their algorithms and databases available for community use, but their own computational resources may be inadequate to fulfill external user demand – yet the system's footprint is far too large for prospective user organizations to download and run locally. Some heavyweight systems have become part of designated "centers" providing remote access to supercomputers and/or clusters supported by substantial government funding; others use virtual supercomputers dispersed across grids formed by massive numbers of volunteer Internet-connected computers. But public funds are limited and not all systems are amenable to huge-scale divisibility into independent computation units. We have identified a class of scientific computing systems where "utility" sub-jobs can be offloaded to any of several alternative providers thereby freeing up local cycles for the main proprietary jobs, implemented a proof-of-concept framework enabling such deployments, and analyzed its expected throughput and response-time impact on a real-world bioinformatics system (Columbia's PredictProtein) whose present users endure long wait queues.

1. Introduction

We are concerned with the problems of reducing the turnaround time and increasing the throughput of CPU-intensive processes in scientific computing domains where "batch" jobs are still commonplace [21], sometimes with wait queues backed up for days and – when finally scheduled for execution – individual jobs running for hours. Most previous research has concentrated on improving performance through the use of parallelization, whether localized (e.g., supercomputers [16, 27]) or massively distributed (e.g., grid computing [5, 13, 25]). These approaches are ideally suited for embarrassingly parallel computations, but do not apply well to inherently sequential computations, and particularly not to pipelined workflows, as occur frequently in scientific areas like biometrics [15] and computer animation [4]. There have been numerous grid workflow languages and tools developed for grid workflow composition and execution [6, 28]; however, these require the execution of all workflow elements on the same local grid.

In the context of provider organizations with relatively limited computational resources, the performance of such pipelined workflows could potentially be improved by *offloading* the prefix of each workflow to an alternative service provider while the suffix remains queued, reducing local processing demands for a given workflow and reducing queue delays for other jobs. This model is appropriate when prefix elements of the workflow run utility preprocessing or data transformation computations that are readily available from multiple providers, but later elements are proprietary to a given provider and/or too heavyweight to be transported to alternative hosts. In this paper, we introduce an approach, called "CPU Torrent", that supports and manages this offloading model. CPU Torrent is orthogonal to and can leverage any parallelization available for individual workflow elements when resources permit.

We have identified a class of scientific computing systems where computation jobs can be divided into two parts, which we call *utility* and *proprietary*. Utility sub-jobs are, by definition, those offered by a number of service providers, which usually advertise a programmatic way of submitting jobs and retrieving results. Proprietary sub-jobs, on the other hand, are, by definition, offered by very few (and sometimes only one) service providers. The objective of splitting a job into utility and proprietary sub-jobs is to use the output produced by the utility sub-job as part of the input for the proprietary sub-job. Thus, the utility sub-job needs to run before the proprietary sub-job and can be seen as preprocessing.

There are many examples of such computations in various scientific fields. In the world of image processing: before adding its own special effects to an image, a provider often runs Histogram Equalization to improve the contrast of the image. Histogram Equalization, a well-known mechanism in the image processing community, can be thought

of as the utility sub-job. On the other hand, novel algorithms producing special effects, such as adding fog, can be thought of as proprietary sub-jobs. In the field of bioinformatics: sequence alignment using an algorithm like blast [1] or clustal [14] is often the first step before more complex analysis of genes and DNA is carried out. Blast and clustal are offered by a number of service providers, with packages available in various programming languages and for various operating systems. Thus, they can be thought of as utility sub-jobs. Examples of available-to-the-general-public service providers for blast include the National Center for Biotechnology Information [18], the European Bioinformatics Institute [10], and the DNA Data Bank of Japan [8].

We call our system **CPU Torrent** in tribute to BitTorrent [7]. In traditional client-server file transfers, the entire bandwidth load has to be borne by the server. BitTorrent alleviates this by offloading part of the bandwidth resource requirement to the clients by dynamically constructing a peer-to-peer architecture for transferring files, where each client gets parts of the (usually very large) file from its peers. This decentralized approach helps reduce the bandwidth needed at the server to a significant extent, to the point that a centralized server is not needed at all after enough peers seed the file for others to download. CPU Torrent is similar in some ways – we propose to offload part of the CPU computation load to alternative providers, thereby reducing the proprietary provider's resource requirements. The analogy is far from perfect, however, since the heavyweight computations targeted here are ordinarily not amenable to offloading onto arbitrary user clients and only pre-existing alternative providers are leveraged (although the latter set can be dynamically updated in our implementation, in practice it tends to remain static over months to years).

The plan of this paper is as follows. Section 2 describes the background and motivation for this paper. Section 3 gives more details about our approach and the general architecture of the CPU Torrent system. Section 4 covers the implementation details of CPU Torrent in connection with a specific scientific computing system, PredictProtein, which implements protein sequence analysis, structure and function prediction [22]. In Section 5, we present a theoretical analysis using Queuing Theory. Finally, in Sections 6 and 7, we conclude the paper with a discussion of related and future work.

2. Background and Motivation

We are collaborating with researchers at Columbia University's Center for Computational Biology & Bioinformatics (C2B2) and its Multiscale Analysis of Genomic and Cellular Networks (MAGNet) Center, who have developed a variety of software packages and databases supporting research in molecular and systems biology. Our goal is to explore new ways to improve the experience of the external and internal users of their scientific computing systems. In this paper, we address the problem that the current users of Columbia's PredictProtein frequently endure long wait queues due to limited CPU resources. This problem appears to be common, with similarly long waits for the results from analogous genomic analysis services provided by other contributors to the biomedical community.

These systems operate as Internet services to which users submit their jobs and retrieve results over the Web. Many systems [8, 10, 11] also allow users to retrieve the results non-interactively via email. Users may choose the latter option if there is likely to be a long wait for the results. The jobs are often implemented by a workflow that first invokes a utility service (e.g., blast in PredictProtein's case) prior to executing one or more proprietary services specific to the provider (e.g., PredictProtein includes several different methods, for predictions of disordered/natively unstructured regions, inter-residue contacts, domain assignments, and protein-protein interaction and protein-DNA binding residues). The output produced by the utility service is used as part of the input for the proprietary service.

Both utility and proprietary analyses demand substantial processing cycles, generally with each workflow executed to completion on a dedicated CPU. For example, PredictProtein runs as a *daemon* (system service executed on demand) on a cluster with 23 nodes. Each node has two CPUs, thus allowing a maximum of 46 jobs to be processed simultaneously. Incoming jobs are processed as follows:

1. A new job is put on the queue.
2. When a node becomes available, the PredictProtein daemon takes a job from the head of the queue, and dispatches it to the node for processing.
3. PredictProtein periodically checks a shared location for the results of each dispatched job.
4. When results become available, they are sent back to the requesting user.

If more than 46 jobs have been submitted before the earliest arriving jobs complete, they must wait in the queue, increasing the response time for those jobs as well as later arrivals.

The wait time on the queue can sometimes be substantial. Figure 1 illustrates the job queue for PredictProtein monitored over the period of a typical week. During that week, there were two spikes as shown. During the first spike, the number of jobs waiting in the queue reached a high of 469. During the second spike, the number of jobs waiting reached a high of 130. For the entire duration of the spikes, the cluster was running at a full load of 46 jobs. The PredictProtein system remained at full load for more than

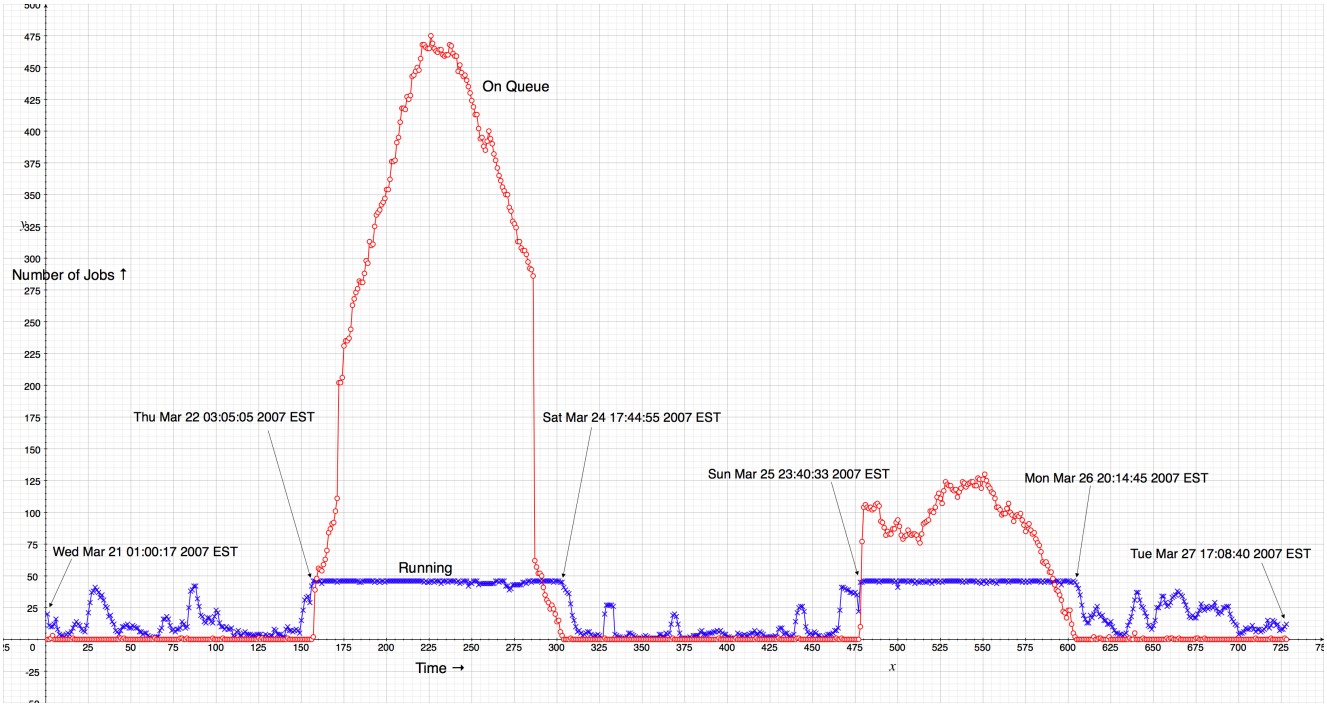

Figure 1. PredictProtein Job Queue

2.5 days during the first spike and almost 1 day for the second spike. Other than these two spikes, the cluster always ran below full load and the number of jobs on queue was always 0. However, empirical evidence suggests that users do not conveniently delay their job submissions – and thus their genomic research – until idle periods.

An apparently simple solution would be to package a downloadable program for users to install on their own machines and run the services they want, using their own CPUs. This is not feasible for two reasons: First, some (not all) so-called proprietary services are, indeed, proprietary in the sense that inventors need to safeguard and control the distribution and use of their algorithms and data, which may be subject to experimental refinements and other changes as well as to contractual and regulatory restrictions. Second, many of the services require huge databases (on the order of multiple terabytes) as part of their analyses. Even the basic blast service involves a 70GB database, which might not be considered all that large given today's disk sizes – except that the database is updated and redistributed *daily* (without support for incremental update [20]).

However, the so-called utility services that prefix typical analysis workflows are offered by a number of provider organizations, which usually supply a programmatic mechanism for submitting jobs and retrieving results. Thus, in principle, users could individually send their own utility jobs to appropriate providers, and then submit the results to the proprietary service. But this would require each user to know the internal workflow steps and file format transformations – which these kinds of systems aim to encapsulate – and is tedious and error-prone.

With CPU Torrent, we present a means to reduce the load on scientific computation systems like PredictProtein by offloading utility sub-jobs to external service providers.

3. Model and Architecture

In order to leverage CPU Torrent, a scientific computation system must fulfill the following requirements: (1) some reasonable fraction of the jobs submitted to the system can be automatically split into sub-jobs; (2) some of these sub-jobs run programs that can be executed remotely by at least one alternative provider (i.e., a utility provider); and (3) those alternative providers supply some practical means (see discussion below) for remotely submitting jobs and retrieving the results. The system does not need to have any explicit notion of "workflow" – although if it does, that would likely simplify splitting. We discuss primarily in terms of a prefix for exposition simplicity, but offloading intermediate and suffix sub-jobs would also reduce user wait time and provider resource requirements.

The scientific computation system sends each utility sub-job to CPU Torrent while the proprietary remainder waits in its queue (if queuing isn't necessary because the system has resources immediately available, CPU Torrent should

never be invoked). CPU Torrent, in turn, sends the sub-job to some utility provider and waits for the results. If CPU Torrent receives acceptable utility sub-job results before its configurable timeout or local resources have become available to run the entire job, the results are transmitted to the scientific computation system; if not, then that system is notified that is should run the sub-job itself. Our approach is thus fault tolerant, handling cases when no utility provider is available/reachable or sufficiently lightly loaded, or if inputs or outputs are deemed corrupted. In that case, the utility sub-job must be scheduled locally together with the remainder proprietary sub-job(s), as the scientific computation system would normally do without CPU Torrent.

The only requirement for external organizations to act as utility providers is that they must publish some mechanism(s) for external users to submit jobs and retrieve results. Some programmatic means is best, e.g., Web Service, API, command-line downloadable client. If there is no programmatic mechanism available and instead the utility provider presumes an interactive (but still remote) end-user, more complex wrappers would need to be written to do some kind of "screen scraping" to submit jobs and retrieve results. If the external service returns results asynchronously, say by email, corresponding wrappers would then be needed to parse the email contents and provide the results to the scientific computation system. CPU Torrent, in principle, allows for all these possibilities – but with the obvious additional burden on the user organizations to develop and maintain the extra code.

Figure 2 shows the general structure of the CPU Torrent system as a form of middleware. The clients (proprietary service providers) must be modified to split their workflows to submit utility jobs to the middleware via its API. The servers (utility service providers) are necessarily used "as is", through whatever mechanisms are already available: utility service providers do not need to cooperate with or even know about CPU Torrent. The middleware is responsible for choosing sites to process the utility jobs and all subsequent interactions with the utility service providers on behalf of the proprietary service providers.

4. Implementation

Figure 3 shows the structure of the CPU Torrent system as implemented, in proof-of-concept form, initially targeted to Columbia's PredictProtein system. The PredictProtein daemon is the client that submits jobs to the CPU Torrent middleware. The only utility job, in this case, is blast. We implemented two possible options for where it can run:

1. SGE Blast (Local Server):

   This is the local installation of blast on a Sun Grid Engine (SGE) [24] hosted at Columbia's C2B2 center.

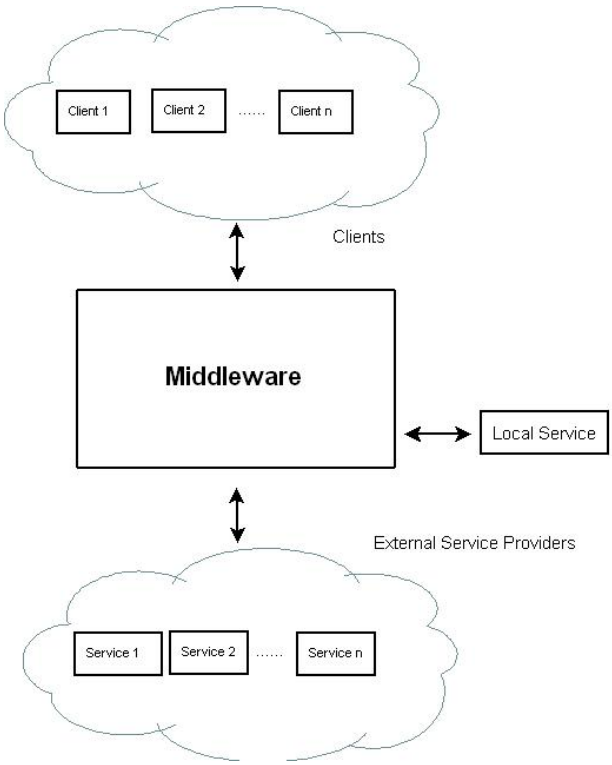

Figure 2. CPU Torrent and a Generic System

2. NCBI netblast (External Server):

   This uses the National Center for Biotechnology Information (NCBI) netblast client [19].

Figure 4 shows the internal details of the CPU Torrent middleware. The components consist of:

1. Middleware API

   The Middleware API is the interface that any Proprietary Service provider would use to interact with the CPU Torrent middleware. It is implemented using Java's Remote Method Invocation (RMI) and supports the following methods:

   (a) Submit a Job

   ```
   String submit(byte[] file,
                 Hashtable options)
   ```

   The `submit` method takes a file as input and returns a `String` that represents the Job ID, serving as a handle to this utility sub-job. It also takes a list of options in the form of a `Hashtable`. The file will be the input to the utility job (the API could easily be extended to allow for an open-ended number of file inputs). The options are for runtime configuration.

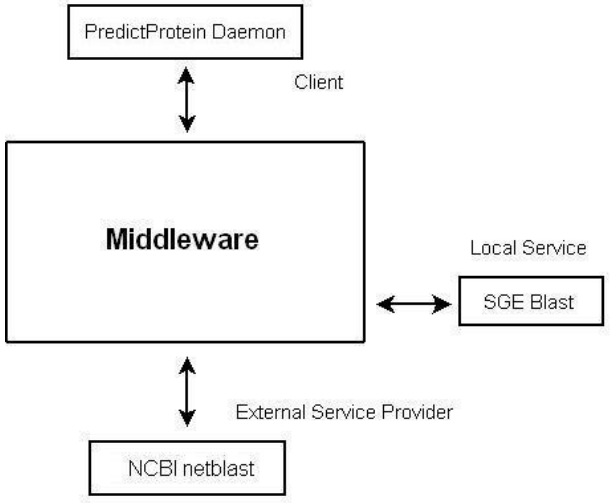

Figure 3. CPU Torrent and PredictProtein

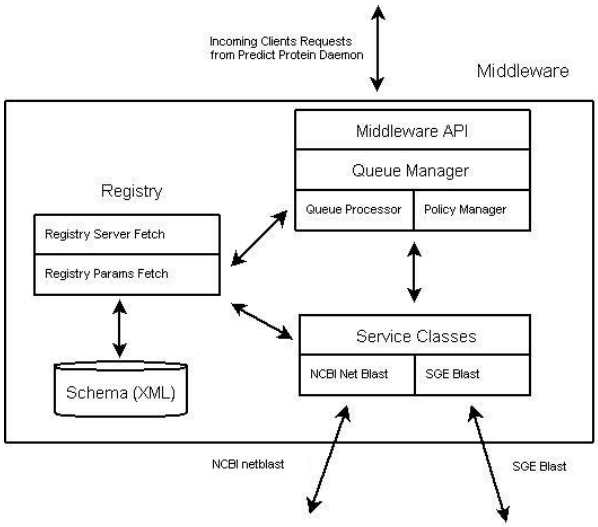

Figure 4. CPU Torrent Middleware Internal Details

In the case of PredictProtein, the input format is a FASTA file [17], which is the same input file format as for blast jobs. FASTA files are reasonably small, typically on the order of kilobytes. In the case of blast, the list of options would include the program name (e.g., blastp, blastn) and the name of the database against which to run the query.

(b) Query the Status of a Job

```
int getStatus(String jobId)
```

The `getStatus` method takes a Job ID as input and returns an integer representing the status of the job. The currently implemented status codes are shown below:

Status Number	Description
0	New Job
1	Job Processing
2	Job Completed
3	Job Failed

Table 1. Job Status Codes

The job status is used by Proprietary Service providers waiting for a job to complete. After submitting a utility sub-job to the CPU Torrent middleware, the provider would poll its status, say every two minutes, until the status changes to "Job Failed" or "Job Completed". The job status could be "Job Failed" if no External Server is available, the input file is not formatted correctly, etc. In the "Job Completed" case, the Proprietary Service provider can then retrieve the results.

(c) Get the Results

```
byte[] getResult(String jobId)
```

The `getResult` method takes a Job ID as input and returns a byte stream, which can then be converted to a regular file containing the results of the sub-job. The result could be one of the following:

i. The result file, if the job completed successfully.
ii. "Job not found", if an invalid Job ID was passed.
iii. "Failed", if the job did not complete for some reason.
iv. "Processing", if the job is still being processed.

2. Registry

CPU Torrent's Registry component keeps track of the various Local and External Server providers registered with the middleware, and what services each of these hosts provide. These details are maintained in two XML files.

The ServerFetch component is used to retrieve the list of prospective servers for a given utility service. The ParamsFetch component is used to retrieve the various parameters needed to invoke the service on a particular host. The most important of these parameters is the

class name of the Service Wrapper used to invoke the service. Other parameters, which may be optional, can be used to set the configuration parameters required by the Service Wrappers. Examples of such parameters include the name of the program, the command-line to execute, and the priority.

Figure 5 shows part of the XML configuration file when there are two External Server providers: NCBI netblast and DDBJ Blast [9]. There is one required parameter common to both cases, the class name. In the case of NCBI netblast, where we use their command-line client, we need to provide the path to the executable. In the case of DDBJ Blast, where we might use its RESTful interface [12], we would need to provide the URL of the web service. The other optional and required parameters are as shown.

3. Queue Manager

   CPU Torrent's Queue Manager component provides basic features for managing the middleware queue. Utility sub-jobs are added to and removed from the Queue via the Queue Manager. The results of a sub-job are also retrieved via the Queue Manager.

4. Queue Processor

   The Queue Processor processes the Job Queue. When a new sub-job is added to the queue, the Queue Processor is activated and uses the Policy Manager to decide to which server to send the utility sub-job.

5. Policy Manager

   The Policy Manager uses a configuration file to determine the "best" server to which a utility sub-job can be sent. This file specifies what ratio of such sub-jobs should be sent to each server. For example, if there were three servers, the Proprietary Service organization could define the ratios such that 30% of the utility sub-jobs are sent to server 0, 50% to server 1, and the remaining 20% to server 2 (all Utility Service providers). A more elaborate implementation of CPU Torrent might automatically adjust the ratios based on various factors, such as success rate, monitoring of each server's own job queue if possible, etc., see Section 7.

6. Service Wrappers

   For each {service, invocation mechanism} pair, it is necessary to implement a *wrapper* (treated as a plug-in) that handles the specifics of how a sub-job is submitted and how the results are retrieved. This code is invoked by the Queue Processor when it chooses to run a particular sub-job on a particular server. The Queue Processor uses an XML file as explained above to obtain the configuration parameters for that Utility Service provider. For example, in the case of NCBI netblast, the XML file indicates the file system path to the program to run and the options to that program (command-line arguments to the executable). This will be *InvokeNetBlast*, the path to the local installation of NCBI netblast, and *blastp*, respectively. This is shown in figure 5.

In the case of PredictProtein, we decided to use one Local Server and one External Server utility provider. The PredictProtein daemon splits all jobs when they arrive, and sends the utility part of the job (blast) to the CPU Torrent middleware. It then periodically checks the status of those sub-jobs, and retrieves the results when they become available. It would then schedule the remaining part of the job and proceed as normal.

More generally: if a Proprietary Service organization wants to use CPU Torrent, it would have to specify and implement Service Wrappers for the various External Server utility providers to which it plans to offload the utility parts of relevant jobs as well as the priorities associated with each of those providers (i.e., if more than one, or if a Local Server should be used for some of these sub-jobs – if everything is to be sent to an External Server, then 100%). The Proprietary Service server code would need to be modifed to use CPU Torrent's API (outlined above) to send the utility part of the job to the CPU Torrent middleware. It would have to periodically check the status of that sub-job, and if/when the results are available, retrieve them and use them during the rest of the proprietary job.

5. Theoretical Analysis

To study the utility of the CPU Torrent system, we conducted a theoretical analysis using Queuing Theory. This analysis shows the potential benefits for an existing Proprietary Service provider installation to use CPU Torrent. The expected benefits are the reduction in both user wait time and the CPU load on provider resources. The analysis is presented below.

Let t be the total time required for a job. Let t_q be the time spent waiting on the queue and t_p be the actual processing time. Thus, we have

$$t = t_p + t_q \tag{1}$$

Let t_o be the processing time of the offloadable sub-jobs like blast. Let t_{no} be the processing time of non-offloadable sub-jobs. With CPU Torrent, we can improve t by using the time spent on the queue to process the offloadable sub-jobs elsewhere. Let $t_{torrent}$ be the time taken by a proprietary service provider to complete an entire job, including its use

```
<server id="0" name="NCBI netblast">
    <parameters>
      <parameter required="true" name="classname">InvokeNetBlast </parameter>
      <parameter required="true" name="command">C:\\netblast\\blastcl3 </parameter>
    </parameters>
</server>
<server id="1" name="DDBJ Blast using REST">
    <parameters>
      <parameter required="true" name="classname">InvokeDDBJBlast </parameter>
      <parameter required="true" name="url">
          http://xml.ddbj.nig.ac.jp/rest/Invoke </parameter>
      <parameter required="true" name="service">Blast </parameter>
      <parameter required="false" name="method">searchSimple </parameter>
    </parameters>
</server>
```

Figure 5. Sample XML configuration file for NCBI netblast and DDBJ Blast using REST

of CPU Torrent. Thus, we have

$$t_{torrent} = t_q + t_{no} \tag{2}$$

In equation (2), we make the assumption that $t_q > t_o$. We consider the alternative case, where $t_q < t_o$, below. Percentage Time Improvement, t_i, is given by

$$\begin{aligned} t_i &= \frac{t - t_{torrent}}{t} \times 100 \\ &= \frac{t_o}{t_p + t_q} \times 100 \end{aligned} \tag{3}$$

From Queuing Theory, we assume the M/M/1 Model with

- 1 Server with Server Time as $m\mu$ (μ is the service time of each individual node, m is the number of nodes)
- Poisson Arrivals
- Exponential Service Time Distribution

The Average Waiting Time, $\overline{W}$, is given by,

$$\overline{W} = \frac{\lambda}{m\mu(m\mu - \lambda)} \tag{4}$$

where, λ = Rate of Incoming Jobs.

Thus, from (1), (2), (3), and (4), we have,

$$t_i = \frac{\alpha m(m - t_p\lambda)}{t_p\lambda(1-m) + m^2} \times 100 \tag{5}$$

where $\alpha = \frac{t_o}{t_p}$, i.e., the fraction of the entire process that can be offloaded.

As m is the number of nodes, this number is fixed for a given Proprietary Service installation. Also, λ is more or less constant once the system has reached a steady state. For example, PredictProtein, during the period it was monitored, received about 19 requests/hour. Hence, $\lambda = 19$. Also, we know that the implementation of PredictProtein has 46 nodes. Hence, $m = 46$.

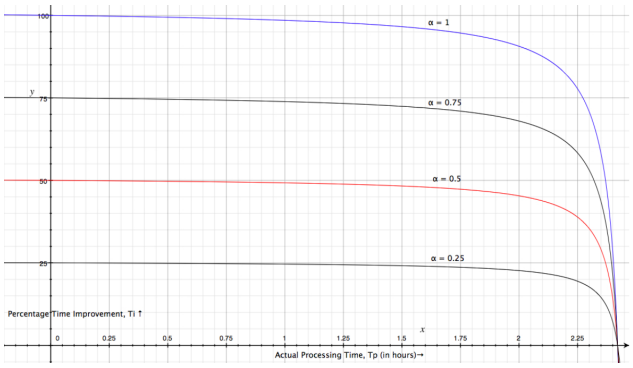

Figure 6. Processing Time Improvement

Figure 6 shows the plot of t_i as a function of t_p for different values of α and treating m and λ as constants. From (5), we see that $t_i \propto \frac{1}{t_q}$. This means that as the time spent on the queue increases, the time improvement is of diminishing importance. This will be even further amplified if $t_q \gg t_p$.

As shown in figure 1, there were many periods during which the job queue for PredictProtein was empty. Thus, both t_q and t_i were 0. With an empty queue, there are no time improvements gained by offloading some computation; in fact, it may actually increase the time required due to overhead like network delays. However, even in this case, the load on the Local Server can be reduced by CPU Torrent at the cost of a marginally slower response time (i.e.,

$t_p + \delta$ instead of t_p, where δ = overhead due to factors like network delays). This is shown below.

Let l_p be the CPU load for the entire job submitted. Let l_o and l_{no} be the CPU load for the offloadable and non-offloadable sub-jobs, respectively. Thus, we have

$$l_p = l_o + l_{no} \tag{6}$$

Let $l_{torrent}$ be the CPU load for the entire process when we use CPU Torrent.

$$l_{torrent} = l_{no} + \beta l_o \tag{7}$$

where β = fraction of the offloadable sub-jobs that we process locally. Percentage Load Improvement, l_i is given by

$$\begin{aligned} l_i &= \frac{l_p - l_{torrent}}{l_p} \times 100 \\ &= \frac{l_o(1-\beta)}{l_o + l_{no}} \times 100 \end{aligned} \tag{8}$$

Let $\alpha = \frac{l_o}{l_p}$. Thus α is the fraction of the entire process which can be offloaded. Thus, from (8), we have

$$l_i = \alpha(1-\beta) \times 100 \tag{9}$$

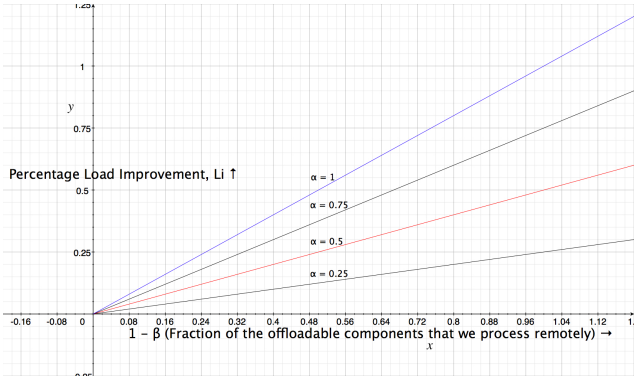

Figure 7. Load Improvement

Figure 7 shows the plot of l_i as a function of $1 - \beta$ for different values of α. Thus, from equation (9) and figure 7, we see the potential benefits of using CPU Torrent even in cases where the wait time on the Proprietary Service provider queue is zero. In such situations, we can reduce the load on the Local Servers. This could prove critical if there is a sudden spike in the rate of incoming jobs, as has been observed in the case of PredictProtein. The same analysis would also hold in the case where $t_q < t_o$, from equation (2).

6. Related Work

There are a variety of systems that have previously implemented some form of CPU cycle offloading, but they do not address the application characteristics outlined above in Section 1 with respect to non-parallelizability and sequential workflow nature.

The SETI@home [3] project was one of the first experiments in public-resource computing. It uses millions of computers on the Internet to analyze radio signals from space, by dividing the work into small chunks that are offloaded to end-user's personal computers. These small chunks are analyzed when the CPU is idle. When each small analysis is complete, the results are sent back to the central server. As mentioned in [3], only certain kinds of tasks are amenable to public-resource computing: First, the task should have a high computing-to-data ratio, i.e., tasks that require a lot of CPU time, but involve very small amounts of transported data. Second, the tasks need to be inherently parallelizable. It is difficult to handle issues like data dependencies using SETI@home's model.

However, neither of these constraints hold for CPU Torrent's target applications: For example, in the case of the blast utility sub-job, the data required to perform the sequence alignment is around 70GB, whereas the CPU time required is typically a few minutes. Hence, we have a very low computing-to-data ratio. Also, in the case of PredictProtein, there are dependencies between the utility and the proprietary jobs. Usually, the output of the utility job is used as part of the input for the proprietary job. Due to these reasons, the SETI@home model would not work and we need a different model. In our model, we handle dependencies by using the time spent waiting on the queue to process part of the job externally. Due to the low computing-to-data ratio, we do not offload the computation to end users, but instead, to other service providers.

The Berkeley Open Infrastructure for Network Computing (BOINC) [2] evolved from the SETI@home project. Their goal is to provide an infrastructure for public-resource computing and to reduce the barriers of entry to create and operate a public-resource computing project. BOINC makes it easy for scientific computing researchers to harvest the power of public-resource computing. This is evident from the large numbers of projects which use BOINC such as Climateprediction.net [23] and Predictor@home [26]. The BOINC project has a good generalized architecture and many useful features, like keeping track of user contributions. However, its model is not directly useful for the applications targeted by CPU Torrent.

7. Limitations and Future Work

We call our system CPU Torrent due to the analogy to BitTorrent outlined in Section 1 above. However, the comparison to BitTorrent is admittedly weak. In particular, it is unlikely we could send utility sub-jobs to the end-user's laptop: We can only submit to pre-existing heavyweight Utility

Service providers that can handle the CPU cycle and storage load. Our approach is, at present, more akin to using mirrors to download data rather than the BitTorrent peer-to-peer architecture.

Also, in BitTorrent, *ad hoc* clients can join/leave a file distribution community on the fly. When deciding how much bandwidth to allocate to a given client, real-time rather than historic contribution is used. The longer term goals for CPU Torrent include organizing communities of users and user organizations based in part on their usage as well as provisioning of both utility and proprietary service providers. For example, user jobs might be given priority on the basis of a *Karma Rating*. We have in mind that this would be the measure of "goodness" of a user derived from the user's or group's (or other identifiable unit's) historic/recent contribution of "quality of service" (max CPU cycles, min queue wait time) to the community.

Another useful addition to CPU Torrent would be the ability to dynamically load-balance among the various service providers. This could override the static configuration provided by the Policy Manager, and would enable CPU Torrent to send jobs to servers that have a smaller wait time and/or job queue. This would require some way for CPU Torrent to monitor the status of the job queues for the external service providers. This could be done programmatically, if provided by the service provider, or through screen scraping, if necessary. If neither of these are viable, one option would be to send fake jobs to the service providers and infer the load on the provider by the time taken for the job to complete.

Finally, we need to analyze the real-life benefits and compare them to the theoretical ones in section 5. We have transferred the technology and our implementation to the developers of PredictProtein and we await final integration with their system. Once this is complete, we will be able to measure the real benefits of using CPU Torrent.

8. Conclusion

We have presented an approach called CPU Torrent to offload computations of sub-parts of heavyweight scientific computation workflows onto external service providers, which has been implemented in limited proof-of-concept form. CPU Torrent reduces user wait time and provider resource requirements.

9. Acknowledgements

We would like to thank Guy Yachdav and Eyal Mozes for their assistance with PredictProtein. We would also like to thank Aaron Fernandes, Shruti Gandhi, and Bhavesh Patira for their collaboration on the implementation of the CPU Torrent system. The authors are members of the Programhming Systems Lab, funded in part by NSF CNS-0717544, CNS-0627473 and CNS-0426623, and NIH 1 U54 CA121852-01A1.

References

[1] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *J Mol Biol*, 215(3):403–410, 1990.

[2] D. P. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.

[3] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: an experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, 2002.

[4] D. Bitouk and S. K. Nayar. Creating a Speech Enabled Avatar from a Single Photograph. In *Proceedings of IEEE Virtual Reality*, Mar 2008.

[5] A. R. Butt, R. Zhang, and Y. C. Hu. A Self-Organizing Flock of Condors. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 42, Washington, DC, USA, 2003. IEEE Computer Society.

[6] J. Chen and W. van der Aalst. On scientific workflow. *TCSC Newsletter, IEEE Technical Committee on Scalable Computing*, 9(1), 2007.

[7] B. Cohen. BitTorrent Protocol 1.0. http://www.bittorrent.org/beps/bep_0003.html, 2002.

[8] DNA Data Bank of Japan. http://www.ddbj.nig.ac.jp.

[9] DNA Data Bank of Japan. DDBJ Blast. http://blast.ddbj.nig.ac.jp/top-e.html, 2008.

[10] European Bioinformatics Institute. http://www.ebi.ac.uk.

[11] V. A. Eyrich and B. Rost. META-PP: single interface to crucial prediction servers. *Nucleic Acids Research*, 31(13):3308–10, 2003.

[12] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[13] I. Foster and C. Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., USA, 2003.

[14] D. Higgins, J. Thompson, and T. Gibson. Using CLUSTAL for multiple sequence alignments. *Methods Enzymol.*, 266:383–402, 1996.

[15] D. Maltoni, D. Maio, A. K. Jain, and S. Prabhakar. *Handbook of Fingerprint Recognition*. Springer-Verlag, New York, USA, 2003.

[16] J. E. Moreira, V. Salapura, G. Almasi, C. Archer, R. Bellofatto, P. Bergner, R. Bickford, M. Blumrich, J. R. Brunheroto, A. A. Bright, M. Brutman, n. José G. Castaños D. Chen, P. Coteus, P. Crumley, S. Ellis, T. Engelsiepen, A. Gara, M. Giampapa, T. Gooding, S. Hall, R. A. Haring, R. Haskin, P. Heidelberger, D. Hoenicke, T. Inglett, G. V. Kopcsay, D. Lieber, D. Limpert, P. McCarthy, M. Megerian, M. Mundy, M. Ohmacht, J. Parker, R. A. Rand, D. Reed,

R. Sahoo, A. Sanomiya, R. Shok, B. Smith, G. G. Stewart, T. Takken, P. Vranas, B. Wallenfelt, M. Blocksome, and J. Ratterman. The blue gene/L supercomputer: a hardware and software story. *Int. J. Parallel Program.*, 35(3):181–206, 2007.

[17] National Center for Biotechnology Information. FASTA format description. http://www.ncbi.nlm.nih.gov/blast/fasta.shtml.

[18] National Center for Biotechnology Information. http://ncbi.nih.gov.

[19] National Center for Biotechnology Information. NCBI netblast. ftp://ftp.ncbi.nih.gov/blast/documents/netblast.html.

[20] National Center for Biotechnology Information. NCBI Blast README. ftp://ftp.ncbi.nih.gov/blast/db/README, 2007.

[21] National Center for Computational Sciences. Interactive Batch Jobs. http://www.nccs.gov/computing-resources/jaguar/running-jobs/interactive-batch-jobs/.

[22] B. Rost, G. Yachdav, and J. Liu. The PredictProtein Server. *Nucleic Acids Research*, 32(Web Server Issue):W321–W326, 2004.

[23] D. Stainforth, J. Kettleborough, A. Martin, A. Simpson, R. Gillis, A. Akkas, R. Gault, M. Collins, D. Gavaghan, and M. Allen. Climateprediction.net: Design Principles for Public-Resource Modeling Research. In *Proceedings of the 14th IASTED International Conference on Parallel and Distributed Computing Systems*, 2002.

[24] Sun Microsystems. Sun Grid Engine. http://www.sun.com/software/gridware, 2008.

[25] H. Takemiya, Y. Tanaka, S. Sekiguchi, S. Ogata, R. K. Kalia, A. Nakano, and P. Vashishta. Sustainable adaptive grid supercomputing: multiscale simulation of semiconductor processing across the pacific. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 106, New York, NY, USA, 2006. ACM.

[26] The Scripps Research Institute. Predictor@home. http://predictor.scripps.edu.

[27] N. Venkateswaran, D. Srinivasan, M. Manivannan, T. P. R. S. Sagar, S. Gopalakrishnan, V. Elangovan, K. Chandrasekar, P. K. Ramesh, V. Venkatesan, A. Babu, and Sudharshan. Future generation supercomputers I: a paradigm for node architecture. *SIGARCH Comput. Archit. News*, 35(5):49–60, 2007.

[28] J. Yu and R. Buyya. A taxonomy of scientific workflow systems for grid computing. *SIGMOD Rec.*, 34(3):44–49, 2005.