Generating the *Transactions and Proceedings of the Royal Society of New Zealand* website, using XSLT running on a Condor grid

Conal Tuohy
NZ Electronic Text Centre
29 January 2007

Executive summary

This is a report on an experimental NZETC project to speed up the automated regeneration of a large website by using VUW's Condor computing grid.

The aim of the project was to investigate the feasibility of using the grid as a platform for website generation. The project did not extend to generating the entire website, though it did produce the bulk of it, and did successfully demonstrate the feasibility of using the grid for similar work in future.

The grid-based application has shown the grid to be a good platform for generating large websites from XML source materials. The application was able to generate around 80 thousand web pages in about 4 hours, making it many times quicker than a system based on a single computer.

The project also highlighted a few technical issues with the Condor grid, relating to the use of file folders in grid jobs; prioritisation of grid nodes based on the opening hours of the computer labs; and the special configuration of computers running Microsoft Windows.

Background and motivation

In 2006 the NZETC partnered with the National Library of NZ to digitise the *Transactions and Proceedings of the Royal Society of New Zealand,* and to make the material available on the Web.

The 89 bound volumes of the journal, each containing several hundred pages of text, were transcribed into 89 XML files according to the Text Encoding Initiative guidelines. Using the Apache Cocoon framework, we built a web application to dynamically transform these XML documents, using XSLT, into HTML format.

To achieve better performance (i.e. a more responsive website), we then used a web-crawler to crawl the Cocoon-based website and save the generated pages. The build process produced about 90 thousand HTML pages, which could then served up by another web-server without the overhead of XSLT transformation.

The process of crawling the Cocoon website was very time-consuming. Crawling the site at the NZETC took almost 8 days, and at the National Library it took over 10 days.

Subsequently we sought a way to reduce the time required to rebuild the site from the XML source files, and we decided to try to re-implement the build process using the Condor grid.

The Condor grid at VUW is a distributed computing facility consisting of about 1000 Windows XP computers. These computers are normally used by students, but are available to run jobs for the grid when otherwise idle.

Change to application architecture

Although almost all of the XSLT transformations used in the Cocoon-based application were usable without any change in the new application, there were considerable changes to the application architecture required to enable the application to be hosted on the Condor grid..

Determining the pages to be generated

Cocoon itself could not be hosted on the grid. The Cocoon application itself, plus the XML data files, was over 200MB in size, which would have made network traffic a bottleneck.

More problematically, the scope of the Cocoon-based build process was not even predefined, making it harder to break the job up into independent jobs which could run in parallel.

Rather than being predefined, the number of web pages generated, and the URLs of each of those pages, were the result of several different XSLT transformations of the XML source files. For instance, the XSLT which generated a table of contents for a given volume would generate HTML hyperlinks to the chapters and subsections of the volume, and by following those links, the crawler would invoke Cocoon to generate those pages, which might include new links for the crawler to follow, and so on, in a recursive fashion. As the crawler encountered new links, it would add them to its list, and the entire build process stopped only when the web crawler had crawled the last page on its list without finding any new links on that page. The centralised function of the crawler did not fit readily with the distributed architecture of a computing grid.

For this reason, the grid-based version of the application was written to first compute the job specifications for every required page, and then to submit those jobs to the grid.

The first phase of the application therefore ran on a central host, generating Condor job specifications and Windows batch files from the XML source files. This phase took only a few minutes to run.

Most of the work required to port the Cocoon application to the grid was in determining the actual set of URLs which the website should generate.

Performing the XSLT transformations

XSLT processor

Cocoon is a Java-based application which uses Apache's Java-based XSLT transformer Xalan to execute XSLT transformations. In the grid-based application, to eliminate the dependency on Java, and to improve performance, we opted to use a native Windows executable instead of Xalan for Java. We opted to use Microsoft's command-line XSLT transformer msxsl.exe to perform the XSLT transformations, because all dependent libraries were already present on the Condor nodes, and the msxsl.exe file is only 25kB in size. The original transformations were written in pure XSLT without extensions, and this meant that the change of XSLT processors did not require any changes to the XSLT code.

XSLT pipelines

In the Cocoon-based application, each HTML file was produced by a "pipeline" of several distinct XSLT (or other) transformations, where the output of each transformation forms the input of the next transformation in the "pipeline". The pipelines themselves are defined in a "sitemap" file with an XML syntax.

Pipelining is simply a technique for modularising XSLT, allowing for distinct XSLT transformations to each handle distinct aspects such as TEI normalisation, structural analysis and chunking of TEI documents, hyperlink-resolution, conversion of TEI to HTML, and addition of website navigation and branding elements.

In the grid-based application, these individual XSLT transformations were retained almost entirely unchanged, and the pipelines themselves were re-implemented for the Windows command-line in a standard "Unix style", by running msxsl once for each stage in the pipeline, and using pipes to connect the output of each stage to the input of the next.

XSLT pipeline example

```
msxsl rsnz_73.xml tei-normalise.xsl |
msxsl - tei-move-pb.xsl |
msxsl - tei-split-paragraphs.xsl |
msxsl - resolve-refs.xsl div-id="rsnz_73_00_000010" |
msxsl - extract-div.xsl div-id="rsnz_73_00_000010" |
msxsl - extract-pages.xsl |
msxsl - rsnz-page.xsl div-id="rsnz_73_00_000010" volume="rsnz_73" te-ao-hou-root="/rsnz/" |
msxsl - strip-empty-divs.xsl |
msxsl - xinclude.xsl > "rsnz_73_00_000010.html"
```

The Cocoon application defined several pipelines, each generating pages of a particular type (such as articles, tables of contents, or page facsimiles). Only the two most important of these pipelines were re-implemented in the grid-based application: the pipeline for generating page facsimiles and the pipeline for generating articles and sub-sections. Together these two pipelines were responsible for producing the vast bulk of the pages of the site.

XInclude

The original Cocoon pipelines used a dedicated XInclude processor as the last transformation stage, to import boilerplate text in XHTML format into the generated web pages. This was re-implemented as an XSLT transformation which uses the XSLT "document" function to implement just the "include" element from the XInclude specification.

Job granularity

When dividing up the overall job into tasks which can run independently on individual nodes, it's important to make sure that the jobs are optimally-sized to make best use of the grid overall.

If the individual jobs are too small (e.g. a few seconds runtime), then, in total, more time will lost in overheads, queuing jobs, handshaking, transferring files, and so on.

However, there's a downside to making individual jobs too large, as well. The overall job should be divided into at least as many jobs as there are available nodes on the grid, otherwise some of the grid will remain uselessly idle.

Also, if individual jobs are made larger and run for longer, they run a greater risk of being interrupted by an interactive user, causing them to be suspended and eventually restarted. If a student starts using a machine which is running a grid job, the grid job is first suspended, and after a grace period of several minutes, if the machine has not become vacant, the job is killed and will be run on a new machine. This is less of an issue when running jobs overnight, when the chance of interruption is less, however at least some of the grid nodes are available to students for extended hours, so grid availability is not perfect.

Condor provides a kind of match-making facility for finding available nodes which are best matches for particular jobs, which can take into account a number of different criteria. It might be possible to improve throughput overall by reducing the likelihood of use machines located in areas where they are currently accessible to students. This could be achieved by configuring each node to advertise the opening and

closing hours of the lab where it is located. For instance, machines in a 24-hour lab might be configured to make this fact known to Condor, which could then preferentially assign jobs to machines located in a lab which was currently closed to interactive users.

In my application, I could have divided the task into individual jobs which produced a single web page each. This would have produced 80 thousand jobs, each running to completion in less than a second. As part of my initial testing, I did run hundreds of tiny jobs at the same time, in order to test how many well the system managed to run a large number of jobs. However, after the testing I chose to make each job larger.

To begin with, the jobs which generated "article" web pages were programmed to produce up to 20 web pages, and the jobs which generated "page image" web pages were programmed to produce 100 pages, because these jobs were simpler and ran more quickly. This produced about 2500 jobs, with a typical runtime (for both types of jobs) of about 10s. The jobs were submitted in the evening, and the entire batch took about 4 hours to complete. Despite the brief runtime of individual jobs, about a third of the jobs were suspended at least once during their execution.

In a subsequent test, the job sizes were increased to 50 articles and 250 pages, producing a total of 690 jobs. About ¾ of these jobs were suspended at least once during execution, the last of the jobs terminating about 5 hours after submission.

It's clear that a number of factors need to be considered when defining job granularity, including file size, typical runtime of individual jobs, grid workload, and the level of availability of nodes (i.e. the degree to which nodes are available to the grid without interruption from interactive users).

Files

Condor jobs may read and write from a shared file system, or alternatively, data files can be sent and output files retrieved as part of the Condor job itself.

Using a shared file system brings some complications. Condor jobs run as unprivileged users, rather than under the account of the person submitting the jobs, so gaining access to network shares can require special treatment.

In this exercise, Condor's own facility for file transfer was used, rather than a shared file system. However, this facility has an irritating limitation. Condor does not provide support for file system directories either for input or for output. All input files are transferred in a single folder, and all output files must be created, and are returned, in that same folder.

Although input files may be copied from a number of folders, when they are transferred to the grid node where the job is to run, all the specified files are copied together into one folder. If some of these files have the same name, then one of those files will overwrite the others with the same name.

If the software being transferred is set up to expect certain files to be in particular folders (e.g. "data", "bin", "imports", etc), as is usual practice, then it will not run correctly when all the files are lumped together in one place. For this exercise I first reorganised the input files so that they could exist in a single folder, and I wrote a harness script to create all the required output folders on my computer, and submit each of the Condor jobs from the appropriate folder, so that the output files would arrive where they were supposed to.

With the benefit of hindsight, I think a simpler and more generic approach may have been to pack the data files, including their folder structure, into a single archive file, for transfer, and have each job start by unpacking the directory structure. At the end of the job, the output files could be packed up and returned in the same way. This would probably require transferring a file-archiving utility program as well, and would add extra overhead to the jobs, but would have simplified development.

Issues

File handles

Submitting a large number of jobs from a Windows machine required a change to Condor's configuration, apparently due to a bug in Condor.

When each job is submitted, a Condor scheduling process is started to listen for progress on the job, and this requires the use of a file handle provided by the operating system, Windows. Condor attempts to keep track of file handles in order not to deplete the operating of too many, however its checking appears to be too conservative, such that it unnecessarily restricted the number of schedulers which could start, and hence the number of jobs which could run simultaneously.

The "file descriptor safety" checking may be disabled by adding the following line to the Condor configuration file:

```
NETWORK_MAX_PENDING_CONNECTS = -1
```

Windows memory configuration

Submitting jobs to the Condor grid from a Windows machine requires a change to Windows' default memory allocation settings and a reboot.

By default, Windows reserves a heap of just 512kB for all non-interactive applications running under a given session or "Desktop". Condor tries to create a "condor_shadow" process for each job submitted, which quickly exhausts the heap and thereby prevents more than about 120 jobs from starting. However, the heap size can be substantially increased by changing a Windows Registry setting. Approximately 4kB of heap is required for each "condor_shadow" process.

More information on the shared heap is available from Microsoft at
http://support.microsoft.com/kb/184802

Information about how the shared heap affects Condor is available from the Condor FAQ at
http://www.cs.wisc.edu/condor/manual/v6.8.2/7_4Condor_on.html#SECTION00841300000000000000

How to configure

To configure the heap size, run RegEdit and navigate to the key "HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\SubSystems". Within this key, the value named "Windows" contains structured data which includes the figure specifying the heap size. The heap size, in kB, is specified by the third figure following the "SharedSection" parameter, highlighted below. Once the setting is changed, the system must be rebooted for the new setting to take effect.

Default configuration

```
%SystemRoot%\system32\csrss.exe ObjectDirectory=\Windows SharedSection=1024,3072,512
Windows=On SubSystemType=Windows ServerDll=basesrv,1
ServerDll=winsrv:UserServerDllInitialization,3 ServerDll=winsrv:ConServerDllInitialization,2
ProfileControl=Off MaxRequestThreads=16
```

Tweaked configuration

```
%SystemRoot%\system32\csrss.exe ObjectDirectory=\Windows SharedSection=1024,3072,2048
Windows=On SubSystemType=Windows ServerDll=basesrv,1
ServerDll=winsrv:UserServerDllInitialization,3 ServerDll=winsrv:ConServerDllInitialization,2
ProfileControl=Off MaxRequestThreads=16
```

"Hung" jobs

On a very few occasions during testing, a single job would hang and would appear to run for hours without terminating.

Using the condor_q command showed the job as having a ridiculously long runtime. To release the job from its stalled state I used the condor_hold command, followed by condor_release. This seemed to wake up a local Condor process to the fact that the job was not running, and after failing to communicate with the node where the job was supposed to be executing, it eventually rescheduled the job on a different node, and a few minutes later the job was done.

The node on which the job appeared to have stalled had earlier successfully executed another job of the same type, so it was certainly capable of it.

At this stage the reason for the problem is still not known. It is possible that the node where the job was running had been rebooted in the middle of the job, and that the system had not realised this. This hypothesis could be tested by submitting a job to one particular machine, and then rebooting that machine with the job in progress. Possibly the job itself could reboot the machine.