Data Analysis Challenge 2008

Ibis Stargazers (dach004)

Team information

The two major contributors of the *Ibis Stargazers* (dach004) team are:

Jason Maassen (jason@cs.vu.nl)

Frank Seinstra (fjseins@cs.vu.nl)

Additional team members include:

Roelof Kemp (rkemp@cs.vu.nl)

Niels Drost (niels@cs.vu.nl)

Rob van Nieuwpoort (nieuwpoort@astron.nl)

Result

We have obtained the following result in the BT category:

```
SUCCESS TRIAL-final_dach-378425 2191.69422817 hiro000
Thu Aug 14 20:00:32 2008 2599f670309280b4c228de1ca1094669
```

Additional results are described in the Evaluation section below.

Software components

For this data challenge we have used software which has been developed as part of the Ibis project . Ibis is an *open source* Java grid software project of the Computer Systems Group at the Vrije Universiteit, Amsterdam, The Netherlands.

The main goal of the Ibis project is to create an efficient Java-based platform for grid computing. The Ibis project offers a variety of programming models, the Java Grid Application Toolkit (GAT) and the IPL and SmartSockets communication libraries. More information on Ibis and it components can be found at http://www.cs.vu.nl/ibis.

For this challenge we have used three existing Ibis components: Java Grid Application Toolkit (JavaGAT), IPL and SmartSockets. Using these components, we have developed a new master-worker framework and a File Transfer framework. Both will be added to the collection of libraries and programming models offered by Ibis. These two frameworks where then used to implement our application. Figure 1 shows the architecture of our solution.

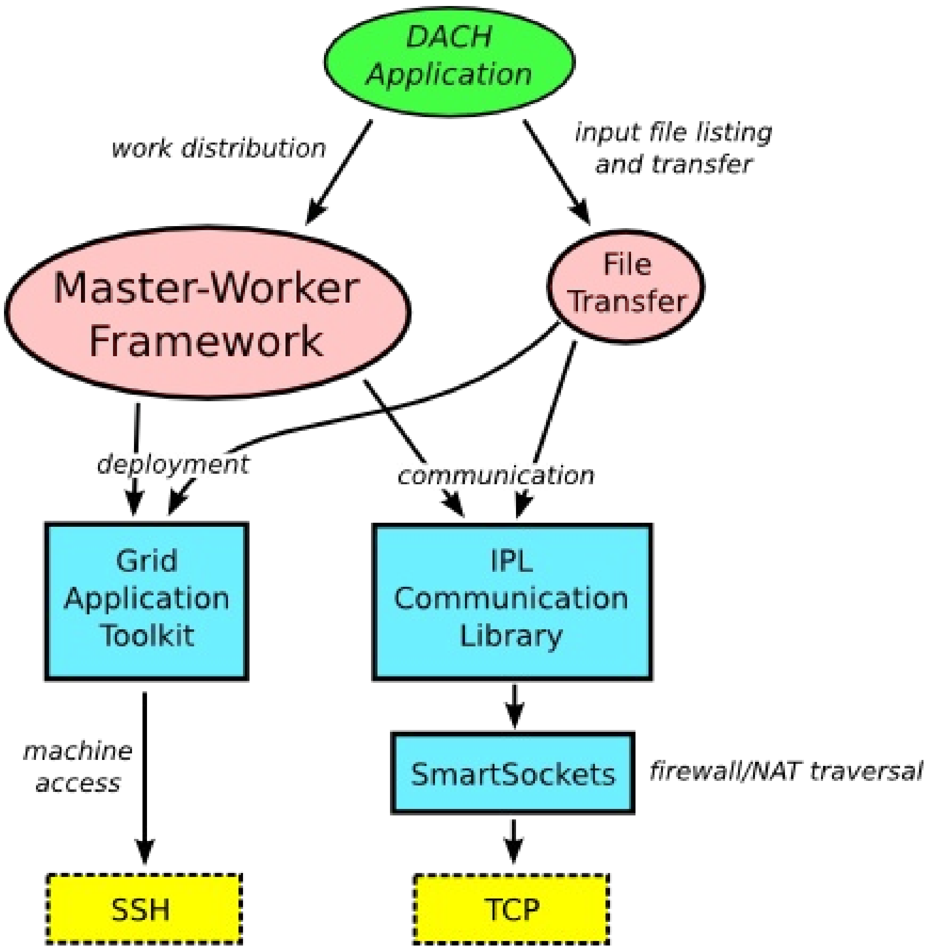

Figure 1: Design of the DACH Application

The DACH application starts by calling the *dach_api* to obtain a run identifier an a directory where the input files can be found. The application then uses the Ibis File Transfer library to obtain a list of which files are available on the different clusters of the InTrigger testbed.

To perform this operation, a simple *File Server* is started on each of the sites. This File Server is implemented in Java and is based on the IPL communication library. The JavaGAT is used to start these File Servers.

The JavaGAT is a toolkit that offers a set generic and flexible APIs for accessing grid services, such as job submission, file transfer and monitoring. JavaGAT sits between grid applications and middleware, such as Globus, SGE or SSH. This allows applications to use a single middleware independent interface, thereby increasing the portability of the application.

For example, on the InTrigger platform, the JavaGAT uses SSH for job submission. On different systems, however, this could easily be replaced by other middleware, such as Globus, without having to change the application code.

Once the DACH application has obtained a list of all input files that are available on the different clusters, it generates a set of Jobs. Each job contains a pair of files that need to be compared using the *superfind* application. The jobs also contains list of locations were each of the files can be found.

These jobs are passed on to the Master-Worker framework. This framework then uses the JavaGAT to start a single *Local Master* on each of the participating sites. In turn, each Local Master uses the JavaGAT to start a worker on each of the compute nodes of its local cluster. The framework itself acts as the *Global Master*. This results in a hierarchical master-worker setup, as shown in Figure 2.

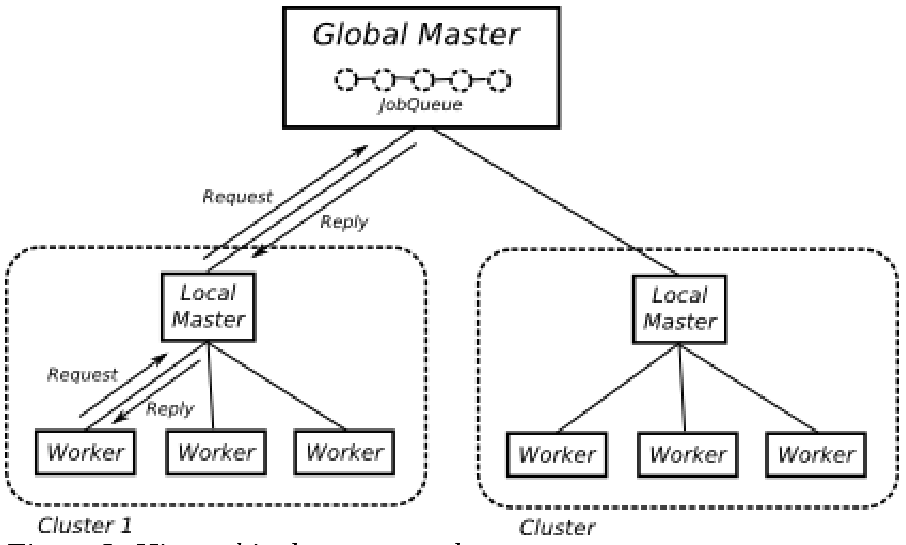

Figure 2: Hierarchical master-worker

Whenever a worker is idle, it will request a job from its Local Master, who forward this request to the Global Master that contains a central job queue. If a job is available, it will be returned using the reverse path.

We use this approach because the input files of some jobs are replicated on two or more clusters. As a result, a these jobs can be executed locally on more than one cluster. By using a single centralized job queue, we prevent these jobs from being run more that once. In addition, if all workers would directly connect to the Global Master, the resulting number of network connection may pose a problem. By routing all worker request though a Local Master on each cluster, the number of required network connections is significantly reduced.

Once a worker receives a job, it calls a DACH application specific function to process the job. This function copies the input files to a temporary directory on the local disk, either using NFS if the file is locally available , or using the Ibis File Transfer library if the file is on a remote cluster. It then calls the superfind application to process the files. The result of the comparison is then send to the Global Master (again via the Local Master), which returns it to the DACH application. The application then appends the to a result file. When all results have been received, DACH application invokes the *dach_api* to hand in the result file and check the answer.

Optimization

Using the approach described above, the minimal total run time is limited to the maximum run time of a single comparison. For example, if the largest comparison takes 45 minutes, the entire application run can never terminate in less then 45 minutes, even if the average comparison time is much smaller.

After analyzing the superfind code in some detail, we found that the most time consuming part of the original code is a convolution operation. This operation is performed in a step-wise manner. In each step only a sub-frame (approximately of size 750 by 750 pixels) of the complete reference input image is being convoluted   entirely independent of any earlier or later convolutions. Essentially, this means that the bare bones for a true data parallel execution of this part of the program are already present, albeit in an implicit manner.

We therefore changed the original *imsub3vp3* program to be able to run the convolutions separately. This results in a explicit split-and-merge approach: after preprocessing, each of the sub-frames is convoluted separately, and the results are written out to file. For the most common image size this result in 18 sub-frames for each image pair. For the largest images pairs 56 sub-frames files are generated. The resulting sub-frames (and other related intermediate results) are then merged together using a separate program, we have called *imsub3vp3_concat.*

By parameterizing the *imsub3vp3* program, we can indicate which subset of the total number of sub-frames should be calculated. Multiple calls to the new *imsub3vp3* will eventually result result in the calculation of all sub-frames.

Next, we have changed the *run_imsub3vp3.sh* script (used by superfind to run *imsub3vp3)* to detect the number of cores available on the worker machine. The *imsub3vp3* program is then invoked multiple times, once for each core. As a result, the speed of the image subtraction part of superfind is improved by a factor of 2 to 8, depending on the number of cores available in the machine.

Evaluation

After a number of experiments, we found that data transfer between sites is problematic. Occasionally, transfers take an excessive amount of time, significantly reducing the efficiency of the application. An example is shown in Figure 3. This figure shows the results of a run performed on 212 compute nodes (704 cores) of 10 different clusters. The distribution of workers is shown in Table 1.

Cluster	chiba	hongo	imade	kyoto	okubo	suzuk	mirai	kobe	kyushu	hiro
Workers	57	14	29	57	12	35	6	11	10	10
Cores	114	28	58	114	24	70	48	88	80	80

Figure 3 shows that the application runs fine for the first 30 minutes. After this time, however, the number of results produced per minute decreases sharply. The application finished successfully, as shown by the dach_api output:

```
SUCCESS TRIAL-final_dach-242250 3199.2698319 hiro000
Thu Aug 14 21:01:05 2008 e2ba74b10090c0dc70f7201e814121c0
```

However, as Figure 3 shows, of the 3199 seconds (53 minutes) used by the application, about half the time is spend waiting for the last 100 jobs.

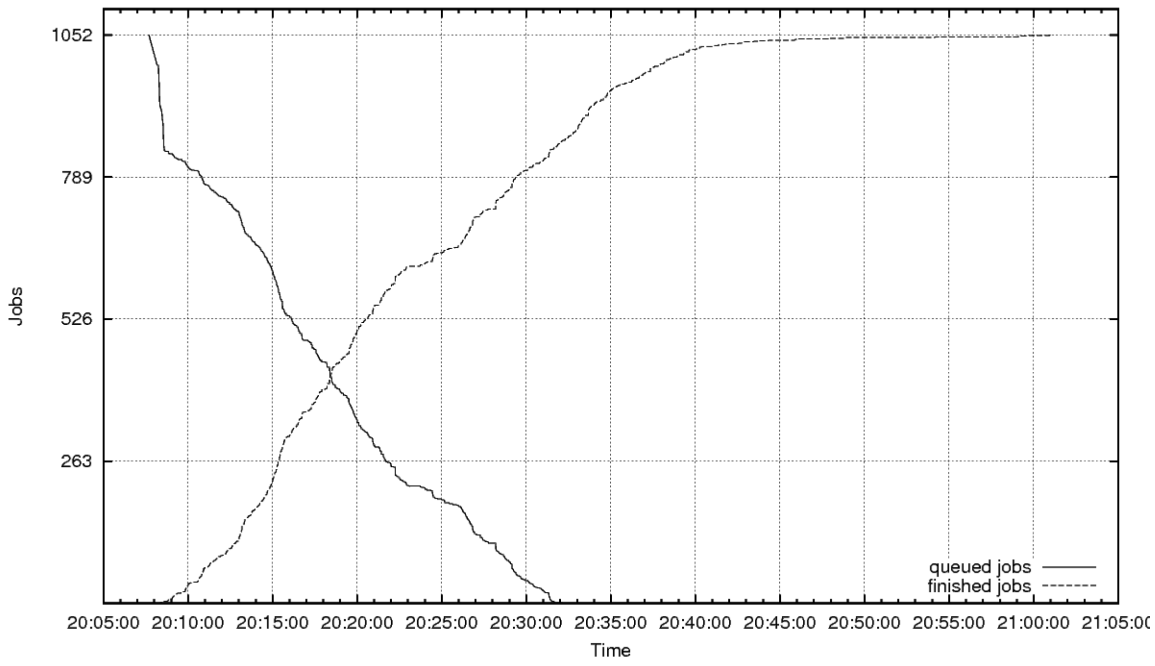

Figure 3: Number of jobs queued and finished

Figure 4 shows an explanation for this behavior. In this figure the total time and data transfer time are plotted for each job. As the figure shows, for a large number of jobs the data transfer time is a significant portion of the total job time.

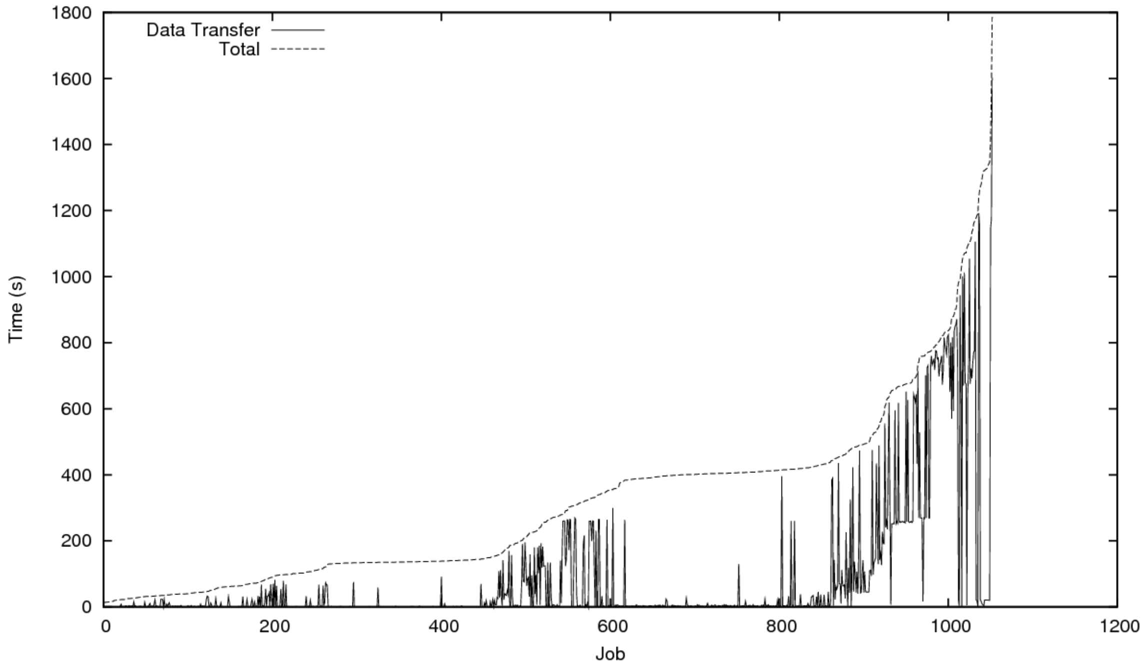

Figure 4: Total time and data transfer time per job (sorted)

In a second experiment, each cluster may only process its local files. As a result, some clusters may finish earlier than other, but no time is lost in data transfer. The results are shown in Figure 5. The experiment is run on the set of machines as shown in Table 1.

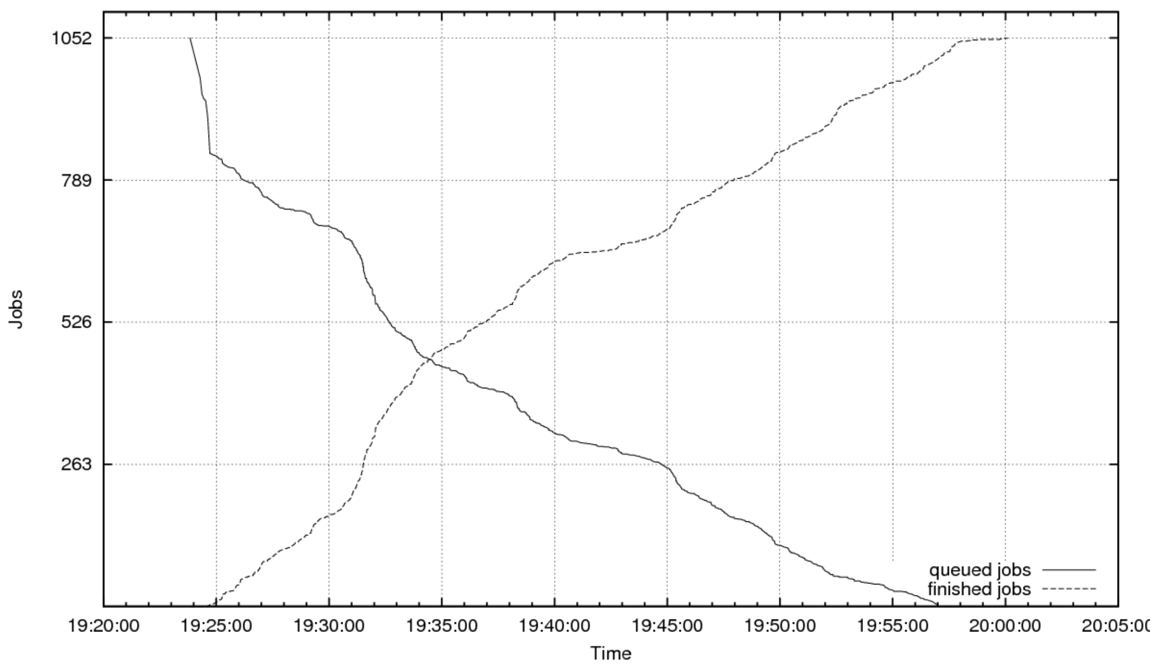

Figure 5: Number of jobs queued and finished (only local processing)

Figure 5 does not exhibit the 'long tail' behavior we saw in the previous experiment. As a result, the application terminates significantly faster, as shown by the dach_api output:

```
SUCCESS TRIAL-final_dach-378425 2191.69422817 hiro000
Thu Aug 14 20:00:32 2008 2599f670309280b4c228de1ca1094669
```

Figure 6 shows the total processing time per job. In addition the file copy time is also shown. However, these values are so small compared to the total completion time, that they are hardly visible in the graph. Figure 6 also shows that the maximum completion time of a job is 1355 seconds (22 minutes). This clearly shows the advantage of our multi core version of superfind.

This number also suggests that is should be possible to calculate the entire problem set in 22 minutes. However, due to the load imbalance caused by the distribution of the input files, it is unlikely that this result can be obtain when using only local processing.

Code complexity

Much of the work done for this challenge resulted in generic components which can be added to the Ibis project. As a result, the DACH application specific part is relatively small. The total application consisted of approximately 1252 lines of code. The DACH master application uses about 50% of this. About 20% is worker specific code, the rest is shared.

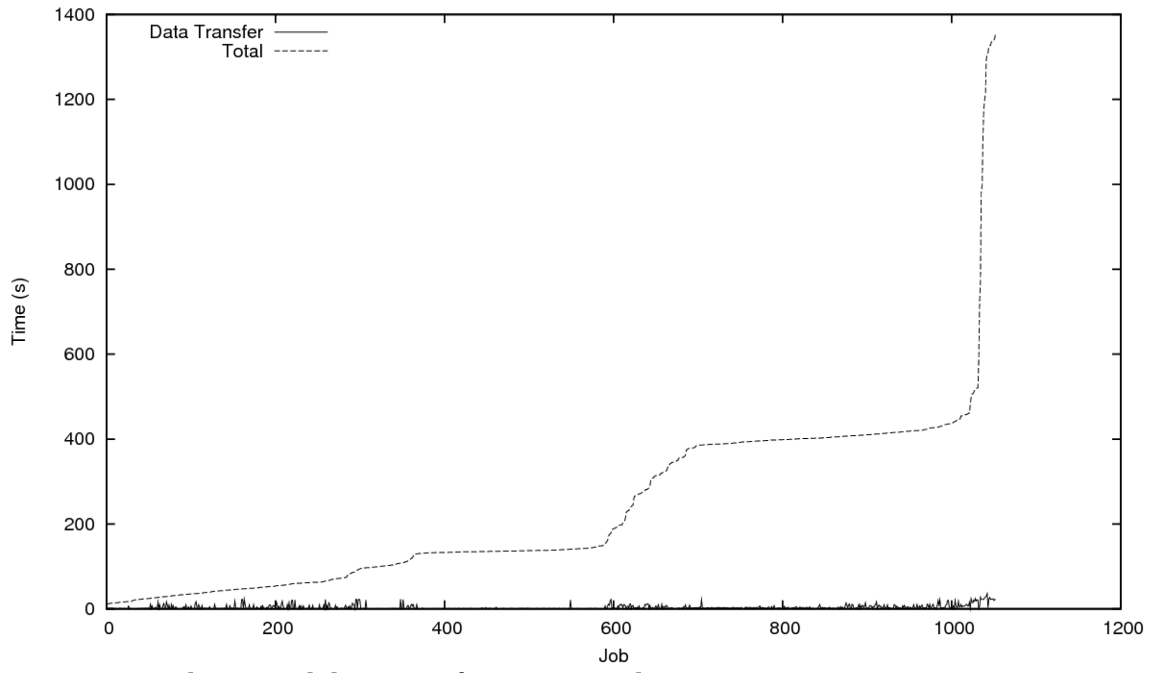

Figure 6: Total time and data transfer time per job

Future Work

Although the current results are encouraging, there is still much room for performance improvements. The experiments show that local processing caused load imbalance. However, due to the problems we encountered with data transfers, the experiment that processed both local and remote files had a lower performance. At the moment it is unclear what caused these data transfers problems.

Both the Master-Worker and File Transfer frameworks that where developed for this challenge will be added to the collection of programming models offered by the Ibis project. Further testing is required, and some work will need to be done to make this code suitable for distribution.

Unfortunately, we did not get a change to subject our code to the fault-tolerance challenge. However, since many parts of the Ibis project are designed with fault-tolerance in mind, it should be quite straight forward to add fault-tolerance to the Master-Worker framework.

Conclusion

In our experiments, we have shown that the components offered by the Ibis project are suitable to create a astronomical image processing application in a relatively short period of time. Although the current results are encouraging, there is still much room for performance improvements. Our experiments also showed that is is beneficial to parallelize the image processing itself to make optimal use of the multi core architectures in use today.