

Benchmarking Edina's OGSA-DAI extensions (Summary)

Michael Koutroumpas

November 30, 2007

1 Preamble

The following Edina OGSA-DAI activities were used during the tests:

WFSGetFeature will load data from a WFS server as blocks of bytes and stream it to the next activity. It checks whether the actual input is a valid WFS response and parses potential WFS error messages.

GDASGetData will load and parse all the requested data from a GDAS server in one iteration. No streaming is necessary because the whole GDAS process only takes about 1% of the total execution time. Memory is also not an issue since the even maximum GDAS response¹ takes up only a few kilobytes of RAM.

GLS takes GDAS census data and WFS geographic feature data and joins the matching attributes using stream-based processing: it starts producing its output immediately without waiting for the input activities to finish.

OGR uses the OGR library to convert between different geospatial formats. This library doesn't support streaming, however, so as a work-around we used Unix pseudofiles that act as pipes (named-pipes).

WriteToFile will save the input stream into a sandboxed area on the filesystem. No existing activity could provide a secure way to store a file to a specific directory and we wanted to avoid using `writeToDatastore/Stream` which store everything in memory. Thus this activity.

1.1 Hardware

All the tests we performed on a 64 bit machine with 2 dual core xeon processors and 8Gb RAM. Each dual core has hyperthreading making a total of 8 processing units. Of course, the hardware threads within each core share the same cache and are therefore not as efficient as 8 cores. Still, we can take advantage of the multiple processing units for concurrent execution.

¹MIMAS GDAS has a maximum response of 2 million values

1.2 Datasets

The tests were performed using the English and Scottish Output Areas 2001 from the UKBorders WFS. The GDAS input comprised the 2001CAS and 2001UV012 datasets from the MIMAS based GDAS server.

1.3 Implementation

The values for both memory and time were recorded during each block iteration inside every activity. Memory was measured as the difference of `Runtime.totalMemory()` - `Runtime.freeMemory()` in Java. The raw time values were recorded using `System.currentTimeMillis()`.

2 Speed

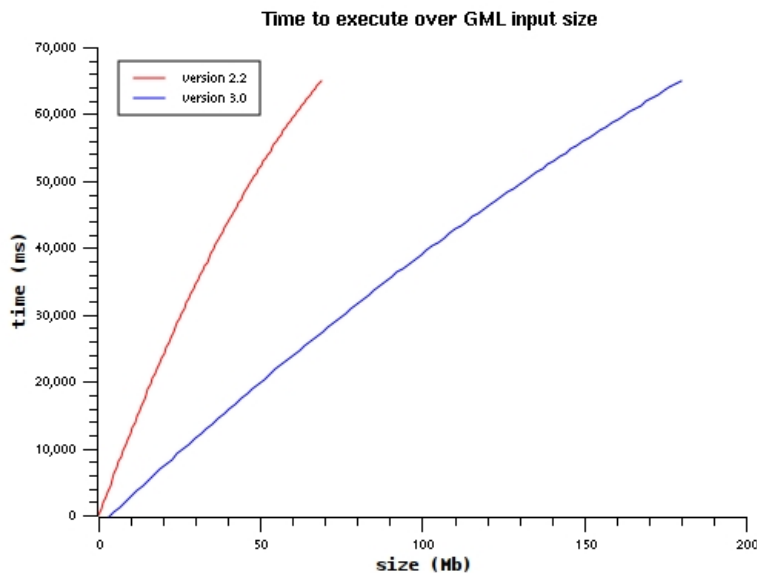


Figure 1: Time to complete as function of the request size.

Figure 1 is a plot of the total time required to complete different requests as a function of the data size coming from the WFS. It seems that version 3.0 is much faster and scales better than version 2.2. This might be related to the fact that version 3.0 executes each activity in a different thread and is therefore more amenable to parallelisation by utilising more processors.

3 Memory

Figure 2 is a plot of the mean memory used as a function of the output size. Since Java automatically allocates/frees its memory, there are random variations in every measure. These variations depend mostly on when the garbage collector is being activated.

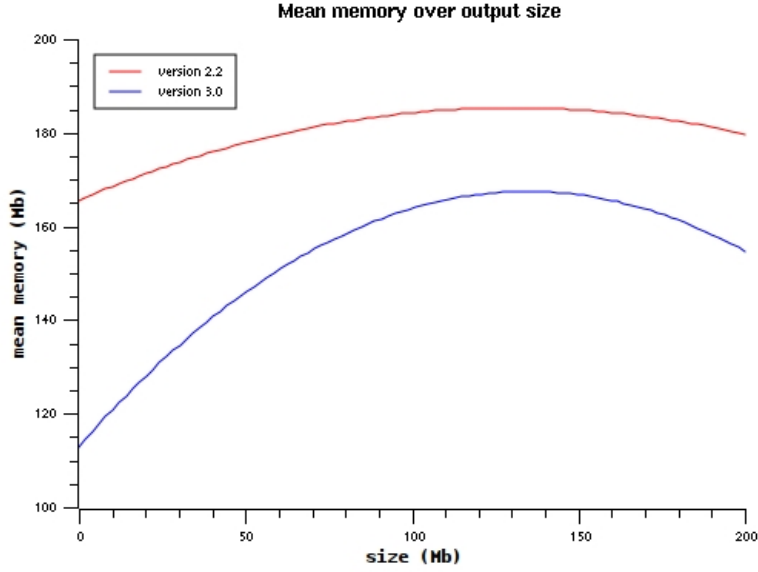


Figure 2: Mean memory used as a function of the output size.

As expected, the fluctuations in memory follow a pattern that is independent of the request. The memory usage is unrelated to the size of the data as long as all the data are carefully processed as a stream.

4 GLS specifics in version 3.0

This is an overview of the execution of the GLS workflow. Figure 3 shows the mean time (in milliseconds) that a block of data has to spend in each activity. GDAS is assigned 0ms because it finishes very early in the execution of the workflow before the actual stream processing starts.

The throughput of our workflow is equal to the number of data blocks that are processed in the unit of time. In a parallel pipeline, this is defined as the inverse of the maximum execution time among all tasks. So assuming t_i is the execution time for task i out of n total tasks:

$$\begin{aligned}
 \text{throughput} &= \frac{1}{\max_{1 \leq i \leq n} T_i} && \Rightarrow \\
 \text{throughput} &= \frac{1}{8 \text{ ms/block}} = 125 \text{ blocks/s}
 \end{aligned}$$

In our implementation 1 *block* = 8 *kb* which is equal to the default 64-bit linux kernel memory page. We can therefore convert the blocks to Kilobytes:

$$\text{throughput} = 125 \text{ blocks/s} \times 8 \text{ kb/block} = 1000 \text{ kb/s} \quad (1)$$

Figure 1 suggests that input is processed at a rate of 2500 *kb/s*, which is x2.5 times bigger than the one in equation (1). This is expected because the

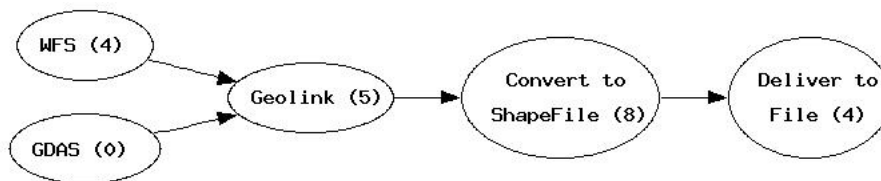


Figure 3: GLS workflow with mean processing times in milliseconds.

GML input is about 2.5 times bigger than the Shapefile output.

5 Conclusion

Both versions of ogsadai seem to perform and scale quite well. However version 3.0 seems to be much faster and lightweight than its predecessor. The initial setup time, the total processing time and the total memory used are much lower in the latest version.

Finally using stream-based processing provides two major advantages:

1. Enables processing of indefinite amounts of data without exhausting memory.
2. Can be used to form a pipeline processing model that is amenable to parallelisation.