Using nakhur-derived objects in MATLAB

Malcolm Lidierth
King's College London
26th October 2010

Nakhur objects allow you to manipulate huge data arrays in MATLAB in a memory and speed efficient way.

MATLAB is a 'matrix laboratory' and supports algebraic manipulation of matrices. The word algebra derives from the title of a book by the 9th Century Baghdad mathematician Abu Ja'far Muhammad ibn Musa al-Khwarizmi: *'ilm al-jabr wa'l-muqābala* (the science of restoring what is missing and equating like with like).

In Persian Arabic, a nakhur is a camel who refuses to yield milk until her nostrils are tickled.

For the nakhur objects in MATLAB, the data are missing from the object but mapped into it from disc or through the system virtual memory. The nakhur class methods restore the missing data as MATLAB needs them, allowing nakhur objects to be used in MATLAB expressions in the same way as standard matrices while using much less MATLAB memory space. Nakhur methods can 'tickle' the data by performing a transform on them before yielding them to the MATLAB expression that called the method, e.g. int16 data from an analogue-to-digital converter might be scaled and offset to double precision values in real world units when accessed through a nakhur-derived object passed into a standard Signal Processing Toolbox m-file function. There is no need to edit the m-file to suit your data or to write custom code to load the data and transform them prior to calling the m-file – you can just replace the standard MATLAB matrix given as input to the function with a nakhur object.

The nakhur class methods are implemented through other classes that extend the nakhur superclass. The most generic of these is the nmatrix class, which can be used to represent any real-valued, non-sparse MATLAB matrix. To create an nmatrix object from a variable in a standard MATLAB MAT-file just call:

```
m=nmatrix(filename, variablename);
```

The output m, is an nmatrix object that will typically require about 2kB of MATLAB memory. The data that m represents may, in theory, be up to 256 terabytes in size.

Performance

The performance figures below were obtained on a MacBook Pro with a 2.26 GHz Intel Core Duo, 3 MB L2 Cache and 4Gb RAM running Snow Leopard.

MATLAB R2010b was used with JVM 1.6.0_22 and 256Mb Java Heap Space assigned in the MATLAB preferences.

Accessing a 2D matrix
A 1Gb double precision 2D matrix was created, stored to a MAT-file and accessed by columns. Columns were accessed randomly until a total of 1Gb had been read.

Elements per column	Columns	Total time with fread (s)	Total time using VM (s)
67108864	2	105.512	27.857
33554432	4	18.744	1.456
16777216	8	18.810	1.449
8388608	16	12.703	1.194
4194304	32	12.751	1.223
2097152	64	12.833	1.264
1048576	128	13.014	1.345
524288	256	13.325	1.462
262144	512	12.966	1.709
131072	1024	13.298	2.133
65536	2048	11.999	1.982
32768	4096	13.534	3.087
16384	8192	18.219	5.598
8192	16384	24.563	10.587
4096	32768	43.151	20.669
2048	65536	78.890	40.850
1024	131072	150.151	81.330
512	262144	293.409	162.907

A 32Mb double precision matrix was created, stored to MAT file and accessed by column. Columns were accessed sequentially (1,2,3 etc) until all 32Mb had been read, or randomly such that the total number of reads and total data read were the same as with sequential reading.

A. Sequentially

B. At random

8388608	2	1.4907433	0.136164393
4194304	4	1.491311618	0.145317385
2097152	8	1.506240124	0.155362341
1048576	16	1.537793759	0.163707284
524288	32	1.567026914	0.180577425
262144	64	1.554293759	0.210750264
131072	128	1.584793183	0.26061416
65536	256	1.482990967	0.252423322
32768	512	1.680311108	0.38258595

16384	1024	2.228224509	0.699861048
8192	2048	3.074037116	1.304188884
4096	4096	5.430077358	2.568871684
2048	8192	9.747997944	5.074820289
1024	16384	18.77147884	10.12732702
512	32768	36.28412909	20.19039919
256	65536	72.18078795	40.38889005