TECHNICAL JOURNAL

Number	JGW 107
Author	James G. Wheeler
Subject	Fun with File Mapped and Appendable Arrays
Date	3/30/2004 9:59 AM Last Updated: 6/25/2004 10:05 AM

Fun with File Mapped and Appendable Arrays

There are a number of exciting new features in SmartArrays release 3, but perhaps the most interesting are important new techniques for working with large arrays.

- The **file array** facility allows a file to be used as the data portion of an array by mapping it to memory.
- **Pre-sizing** and **appending** provide efficient ways to work with arrays that grow by repeatedly adding new data.

Together these techniques let you use memory up to and even beyond the limits of what a machine can hold. The file array facility also provides fast and powerful ways of maintaining array data in files and sharing it between programs or even over a grid of separate computers.

File Mapped Arrays

Consider what information the SmartArrays engine needs to keep for an array. There are the "metadata" values (the shape and datatype of the array), which are held in the array engine's internal array catalog structures, and there are the actual data values, which are stored in a chunk of contiguous memory. SmartArrays data values are held in segments of memory that are allocated from the operating system. For more details, see the *Implementation Details* appendix in the SmartArrays User Manual.

Some Background on Virtual Memory and Memory-Mapped Files

All operating systems provide ways to allocate memory to a user program and this is how SmartArrays normally obtains memory to hold array data. But modern virtual-memory operating systems (like Windows NT/2000/XP or Linux or Unix) also provide for *memory-mapped files*, which allow a disk-based file to be associated with a range of memory addresses. This allows a program to read from or write to a file by referencing or modifying values in memory. The operating system copies fixed-size chunks of storage called *pages* between memory and disk in order to keep the disk image of the file consistent with its memory copy.

Paging between disk and memory forms not only the heart of the virtual-memory facility but also is used beneath the covers for all regular file I/O. When you write code that reads from a file, the operating system at its lowest level is mapping sections of the disk drive to memory and using that memory for a file buffer. When you read one byte from a file, you actually cause a whole page of memory, which typically has a size of 4096 bytes, to be filled with values from disk. If you then read

the next byte in the file, chances are that the value is already in memory and the disk does not need to be looked at again.

Paging also allows for the programs running on your computer to appear to use more memory than you actually have installed by saving data to disk when it hasn't been used recently and bringing it in only when a program actually references it. The operating system usually can get away with this beause many "running" programs are idle for much of the time, or aren't actively using all the memory they have allocated. Of course, if the amount of virtual memory in active use exceeds the physical memory of the machine to a significant degree, a computer can get into a situation where it is spending most of its time paging, frantically copying pages of data between memory and disk, with dire performance degradation as a result.

Normally, though, virtual memory works very well. Because paging and virtual memory are among the most essential things an operating system does, these features are very carefully crafted to be both reliable and fast. Memory-mapped files make all this wonderful machinery available to user programs – letting a user program request that a disk file be "mapped" into memory. Once this is done, the program can use the file by reading or writing memory addresses. Since SmartArrays is based on memory-resident arrays, it can easily work with memory addresses that happen to be mapped to a file.

File Arrays: Using Memory-Mapping with SmartArrays

Four new array methods, new in Release 3, let you exploit memory-mapped files with SmartArrays. They are:

- **fileArray()** - create an array whose data is mapped to a file. Changes to the array are saved in the file.
- **fileArrayRead()** - create an array whose data is mapped read-only to a file and that does not permit changes to values in the array.
- **isFileArray()** - Return true if the subject array is a file-based array.
- **toFileArray()** - create a new file and file-mapped array from the contents of an existing array.

Let's look first at creating an array mapped to a file. Suppose we have a regular memory-resident array containing the numbers from 0 to 999999:

```
SmArray v = SmArray.sequence(1000000);
```

We can write these values to a file by casting them to bytes and writing with the fileWriteBinary() method:

```
v.cast(SmArray.dtByte).fileWriteBinary( "mydata.xxx" );
```

Now we have a file containing 1,000,000 values in 4,000,000 bytes, 4 bytes for each integer value. Suppose that at some later time we want to create a new SmArray and populate it with these values. The "old fashioned" way is to read it into a byte array with **fileReadBinary()** and then cast the values to integer type:

```
SmArray t = SmArray.fileReadBinary( "mydata.xxx");
t.castInto( SmArray.dtInt );
```

This works, but takes time because **fileReadBinary** will physically copy all of the file's data into memory and return an array of bytes. Then we use **castInto()**, another of the new features of Release 3, to reinterpret the byte array as 4-byte integers.

The "modern" way is to map the file to an array:

```
      SmArray t = SmArray.fileArray( "mydata.xxx", SmArray.dtInt );
      t.showDebug();
I*[1000000] 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 ...
```

This produces the apparently identical effect to reading the file, but now the data portion of the array is in memory that is *mapped* to the file. Your code can treat it like any other array, but there are important differences in how it uses machine resources:

- The data is not brought into memory until it is actually used. Mapping the file reserves a range of addresses that are large enough to hold the entire array, but the data hasn't yet been read into memory. The appropriate pages are read from disk only when an SmArray method uses them, so if you never reference large sections of an array, they are never brought in to memory
- The memory is allocated to the operating system, so it does not count against memory limits that apply to user processes.
- The data is inherently non-volatile because the *array and its file are the same thing*. If you change a single value in the array, the change will be reflected in the file and the operating system is obligated to write the change back to disk at some point.
- Separate programs can use the same array simultaneously with only one copy of the file data physically occupying memory. For more on this, see *Shared Arrays* below.

Note by the way how the array **t** was displayed in the output of showDebug(). The '*' indicates that the array is a file array and not an ordinary memory array. You can also use the method **isFileArray()** to determine whether or not an array is mapped to file or stored in volatile memory.

File Mapping Methods

The file mapping methods **fileArray()** and **fileArrayRead()** are static methods with the same syntax:

```
SmArray x = SmArray.fileArray(    // or fileArrayRead for read-only
      "filename",      // the file name (string)
      type,            // one of the SmArray.dtXXX values
      shape,           // optional shape to apply to the data
      offset,          // optional offset in file to start of data
      length );        // optional segment length for calculated shape
```

These methods map the file to memory and return an SmArray with the requested characteristics that uses the mapped memory. The file name and type must always be supplied, but the other parameters are optional. Let's look at each optional parameter in turn.

Type Parameter

Memory-mapped arrays are only suitable for simple numeric or character data. The allowable types are:

- **SmArray.dtByte** – 1-byte character or arbitrary binary data
- **SmArray.dtChar** – 2-byte characters
- **SmArray.dtBoolean –** 1-bit numbers. Note that offset and length must be a multiple of 8.
- **SmArray.dtInt** – 4-byte integers
- **SmArray.dtDouble** – 8-byte IEEE double-precision
- **SmArray.dtComplex –** pairs of 8-byte doubles.

You cannot map a file as **dtString**, since the 4-byte string identifiers used in a string array refer to the string table in your current instance of the SmartArrays engine, and there is no reason to expect them to be valid for a different instance of the engine. Similarly, nested arrays refer to locations outside

the array itself and therefore cannot be mapped to a file. Mixed-type arrays (SmArray.dtMixed) are also not mappable because they may contain string or nested items.

Shape Parameter

shape specifies the shape of the resulting array. If omitted, the array's shape will be inferred from the size of the file (i.e. a 4 million byte file mapped as SmArray.dtInt would be returned as a vector of 1 million integers). You can use **shape** to cause the array to be shaped as a matrix or higher rank array. Thus, specifying a shape of SmArray.vector(100,100,100) would produce an array of shape 100x100x100. You can also specify –1 as the first value of the shape vector, in which case the shape will be calculated based on the size of the mapped memory segment. For example, if shape is SmArray.vector(-1, 50), the result will have 50 columns and as many rows as fit in the file, or as will fit within the optional **length** parameter.

Offset and Length Parameters

The optional **offset** and **length** parameters allow you to specify a part of the file to map to the array. They specify the positions in the file as the number of data items, not the byte offset in the file. Thus, for an array mapped as SmArray.dtByte, an offset of 1000 begins 1000 bytes into the file, but if the file were mapped as SmArray.dtInt the offset of 1000 indicates data beginning 4000 bytes into the file. If **length** is omitted, the mapped segment extends to the end of the file rounded down to the size of an item of the indicated type. If the shape is to be calculated, as indicated by `–1` in the shape parameter, the **length** determines the size of the mapped segment and the shape will be calculated based on this size.

If an expicit shape (one with no leading –1) is passed, then this shape determines the size of the mapped file segment and the **length** parameter is ignored.

Processing a Large File in Chunks

One of the interesting tricks you can perform with memory mapped files is to process a very large array in chunks. Suppose you have a flat file of binary data containing 100 million floating point numbers and you want to calculate the total of those values. You could try to map the entire file to memory, but the operating system probably will refuse to let you allocate 800 megabytes of virtual address space. The solution: process the file by mapping successive chunks. Here's a function in C# that calculates the total without ever allocating more than a specified maximum number of items.

```
public SmArray totalDoubleFile(
    string filename,
    int chunksize, // maximum number of doubles to process in each chunk
    int filesize   // total number of doubles to process in the file
    )
{
    int offset = 0;
    bool running = true;
    SmArray total = SmArray.scalar( 0.0 );
    while( running )
    {
        // Map the next chunk of the file
        if ( offset + chunksize > filesize   )
        {
            running = false;
            chunksize = filesize - offset;
        }
```

```
        SmArray chunk = SmArray.fileArrayRead(
            filename,
            SmArray.dtDouble,
            SmArray.vector(-1),
            offset,
            chunksize );

        offset += chunksize;

        // Add to the cumulative total
        total = total.plus( chunk.reduce( Sm.plus ) );

        // Explicitly release the array, rather than wait for the
        // garbage collector, so we are certain the memory segment
        // has been deleted.  This is necessary in .NET or Java, but
        // not in C++, where the destructor runs immediately once
        // chunk goes out of scope.
        chunk.release();
    }

    return total;
}
```

This code is simple, fast, and uses memory in a predictable way no matter how large the file is. Execution time is about as optimal as humanly possible – the bulk of the time used is that required to copy the data from disk to memory – which would have to be done no matter what.

Read-Only versus Read-Write Mappings

fileArrayRead() maps a file read-only. This means that the array's contents cannot be modified. For example:

```
SmArray t1 = SmArray.fileArrayRead( "mydata.xxx", SmArray.dtInt );
t1.setInt( -1, 0 ); // produces an error
```

Any of the setType methods or the "into" methods like indexInto() that try to change the data in an array will fail. However, there is nothing stopping you from assigning a complete new array to the SmArray object -- it's just the original array's values that are unmodifiable.

A file mapped with **fileArray()** produces a modifiable array. You can write new values into the array. These new values affect all other SmArrays that are mapped to the same file because they all refer to the same file. Modifications to the data in the array are reflected in the file, because the file and the array are one and the same. Any changes made will be permanently reflected in the file.

Determining if an Array is Memory Mapped

The method array.isFileArray() returns true if the array is mapped to a file, and false if it is an ordinary memory array.

Creating a File Array from Another Array

The method **array.toFileArray(filename)** writes the data in a suitable array to file and returns a new file array that is mapped to that file. It provides a simple and efficient way to turn the data of an array into a binary file and to make that data non-volatile. Only simple numeric or character arrays can be converted to file arrays; string, nested, or mixed arrays are not mappable.

toFileArray() is a handy way to write data to file, even if you're not going to use the file array it returns. In the first example above, instead of

```
v.cast(SmArray.dtByte).fileWriteBinary( "mydata.xxx" );
```

we could have written:

```
v.toFileArray( "mydata.xxx" );
```

Because the result of **toFileArray()** isn't assigned to a variable, it is discarded and the memory mapping is dissolved.

Sharing Arrays

Because an array can now be based on a file, arrays can be shared just like files can be shared. This has a number of tantalizing implications:

- Within a single program, multiple SmArray objects can be created that refer to the same file. These file arrays may map the same, or different, or overalapping segments of the file.
- Read-write file arrays can be created and modified and all arrays that refer to the file will immediately see the changes.
- Other processes can map the same file and operate on it. File arrays therefore supply a means for interprocess sharing of arrays.
- The other processes do not even need to be running on the same machine. Multiple programs running on *separate computers* can map the same file to arrays, which means that it's possible to perform grid computing with SmartArrays.

Shared arrays thus open the door to new approaches to using the full power of multi-processor and grid machines in array-based computing. When a task can be partitioned in a way that allows it to be performed in "chunks", you can set tasks to spread that task over multiple CPUs in the same machine, or even separate machines.

Shared Arrays Over a Network

File arrays make it possible for an array on one computer to mapped to a file on a different computer. This works, but if the array is being modified you will need to take care in how you keep the array states synchronized. There is no synchronization built into SmartArrays, so the behavior will be much the same as a file shared over a network because a file array is, in essence, an open file.

If a remote machine maps a file to an array and modifies the array, these changes will modify the file. However, the changes are usually buffered and may not be "flushed" to file's host machine for some time. One technique to hasten the delivery of updates over the network is to close the file mapping by calling the array's **release()** method, then re-map the file with a new call to **fileArray().** It you are developing applications where data is modified across a network, you will need to give careful thought to synchronization, a topic that is beyond the scope of this paper.

Fortunately, for many grid-architecture solutions there is no need for different machines to change an array but only to be able to read it. If a SmartArrays-based data cache resides in files on a one machine and does not change, any number of other machines can map to those files read-only, and safely compute with these arrays. This is a very powerful technique for long-running computations, but it is also useful in large web applications, where a multiple web servers may need to provide computations on the same data.

Appending Data To Arrays

Often an application needs to use arrays that grow and whose ultimate size can't be known in advance. One way to grow an array is to use catenate():

```
array = array.catenate( newdata );
```

But suppose this needs to be done many times. Each time catenate() is called it creates a new array, copies the original array's contents into it followed by the new data. This can be terribly slow because all the data needs to be copied each time. Consider the following case (but don't try it at home unless you have a lot of time to wait):

```
int initialsize = 0;
int items_to_add = 1000000;
SmArray v = SmArray.scalar(0).reshapeBy( initialsize);

for ( int i=0; i<items_to_add; i++ )
{
    v = v.catenate(i ); // a dummy value
}
```

The number of data items copied the first time around the loop is 1, since the array is initially empty. But by the time the millionth item is being catenated, we are copying a million values. In total, the above loop needs to copy 500,000 x 1,000,000 data values or about 500,000,000,000 items, and take a completely unacceptable amount of time. Of course, experienced array developers would never do this; up to now, the best practice has been to create the array and insert values into it with **indexInto()** or **setInt()**.

But now we have a still better way, and one that works well when the eventual size of the array is not known in advance. The new method **append()** in SmartArrays release 3 provides an efficient way to repeatedly add data at the end of an array. The full syntax is

```
array.append( newdata, extra );
```

Where **newdata** is an array of new values to be appended to the array. If the new values will not fit in the array's current memory block, a new block will be allocated with room to hold the original data, the new data, plus **extra** additional items (or units along its first dimension if it is a matrix or high-rank array).

Suppose that the above example were written to use append() and grow the array by 200,000 items each time it needs to be enlarged, like this:

```
int initialsize = 0;
int items_to_add = 1000000;
int increment = 200000;
SmArray v = SmArray.scalar(0).reshapeBy( initialsize);

for ( int i=0; i<items_to_add; i++ )
{
    v = v.append(i, increment );
}
```

In this case, the complete array only gets copied when the block is full. So an extra 200,000 items will be copied the first time the array fills up, then again when it reaches 400,000 items, etc. In total, only 30,000,000 items get copied in the process, an improvement of more than 1000 over using catenate.

Choosing the increment of growth is a trade-off between tying up empty space that will not be used in arrays versus having to allocate and copy new arrays each time they grow. If you have a pretty good idea of how large an array will become, you can pre-allocate this amount of space once and then fill the array with append(). See *Pre-Allocating Array Space* below.

Extra Space in Arrays

One of the reasons **append()** is effective is that most arrays have some extra space. The SmartArrays array engine uses a two-level strategy for allocating memory:

- Arrays smaller than a certain size (currently 64K bytes) are allocated in a block whose size is a power of 2. Thus, an array of 35,000 bytes is given a storage block large enough to hold 65536 bytes, and append() will use this space if the new data fits.
- Arrays larger than 64K are stored in blocks whose size is rounded up to the operating system's page size, typically 4K.

Caution: Be careful when writing code that depends on the SmartArrays storage manager's internal behavior because it may change in future releases.

Pre-Allocating Array Space

The most effective use of **append()** is when you intentionally allocate extra space in arrays based on the expected behavior of your data. There are two ways to obtain extra space – by specifying the **extra** argument to **append()**, or by creating a large array and then reshaping it in-place using **reshapeByInto()**. Here is an example of the latter technique.

```
int initialsize = 0;
int items_to_add = 1000000;

// pre-allocate a larger array
int reservedsize = 1000000;

// may never be required unless we go over reserved size
int extra_space = 200000;
SmArray v = SmArray.scalar(0).reshapeBy( reservedsize );

// set the shape smaller, but keep the storage block.
v.reshapeByInto(initialsize);

for ( int i=0; i<items_to_add; i++ )
{
    v = v.append(i, extra_space );
}
```

Append and Reference Counts

For **append()** to be used effectively, the array must have a reference count of 1, which means that only one SmArray object can refer to the array. If there is more than one reference to array, then append() must create a copy before appending to it. This is not unique to **append()**; the same is true of any array method that modifies an array, such as **indexInto()** or **setInt()**.

Multiple references can occur when the same array is assigned to separate SmArray objects, or when the array is referenced in a nested array.

```
SmArray a = SmArray.scalar( "hello" );  // reference count is 1
SmArray b = a;  // reference count is now 2, so any modification requires a copy
```

One case that deserves consideration is a nested array that holds a related set of arrays, such as might be used to represent the set of data columns of a relational data table. Such arrays are often very large, so appending to one of the items ought not to create an extra copy needlessly. If you select an item out of the array with pick() in order to append to it, you create an extra reference to the array.

```
// Create a 3-item nested array of vectors
SmArray columns = SmArray.sequence(100).enclose()
    .catenate( SmArray.sequence(100,1000).enclose() )
    .catenate( SmArray.sequence(100,2000).enclose() );

// Pick one of the subarrays
SmArray column1 = columns.pickBy( 1 );
```

The SmArray object **column1** now shares a reference to the array at position 1 in **columns.** Thus, appending to it will cause a copy to be created, even if we simply plan to store it back into **columns.** The solution is to set **columns[1]** to an empty vector, which releases the extra reference. Now **column1** can be appended in-place, then stored back into the nested array.

```
columns.pickByInto( SmArray.empty(), 1 ); // release the extra reference
column1.append( 3 );  // add the new data in-place
columns.pickByInto( column1, 1 );  // plant it back in the nested array
```

Append and File Arrays

You can use **append()** with any array, whether it is a regular memory array or a file array. However, one special consideration applies to file arrays, because **append()** will not expand the mapped memory segment used in a file array. To make effective use of file arrays that may grow, the best practice is to initialize the file to its expected ultimate size and then use reshapeByInto() to truncate the array with space left for appends. Here is one example:

```
SmArray a = SmArray.scalar(0).reshapeBy(1000000);
a = a.toFileArray();
a.reshapeByInto(0); // truncate the shape but keep the memory segment.
```

Now it is safe to append data to this array up to the initial limit of 1,000,000 items. If the code tries to append more data to the file than the mapped memory segment can hold, append() will produce an SA_FILE_ARRAY_LIMIT exception.

Note that you cannot use the **length** parameter of **fileArray()** to reserve extra space for appending. The only way to reserve extra space is to create a large array and then truncate it with **reshapeByInto().**

Caution: Be careful when using **append()** with file arrays that are shared between processes or between computers. A separate instance of the SmartArrays engine will not see any changes to the shape information of the array, since the shape is stored in the engine's private data structures and not in the file.

Conclusion

Using file arrays and append operations requires a bit of care, but the reward is being able to handle much larger data objects and achieve greater performance than would otherwise be practical.