Algorithm 891: A Fortran Virtual Memory System

JOHN K. REID and JENNIFER A. SCOTT
Rutherford Appleton Laboratory

`Fortran_Virtual_Memory` is a Fortran 95 package that provides facilities for reading from and writing to direct-access files. A buffer is used to avoid actual input/output operations whenever possible. The data may be spread over many files and for very large data sets these may be held on more than one device. We describe the design of `Fortran_Virtual_Memory` and comment on its use within an out-of-core sparse direct solver.

Categories and Subject Descriptors: G.4 [**Mathematical Software**]

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Virtual memory, out-of-core, direct-access files, Fortran.

ACM Reference Format:

Reid, J. K. and Scott, J. A. 2009. Algorithm 891: A fortran virtual memory system. ACM Trans. Math. Softw. 36, 1, Article 5 (March 2009), 12 pages. DOI = 10.1145/1486525.1486530 http://doi.acm.org/10.1145/1486525.1486530

1. INTRODUCTION

We have recently designed and developed new direct solvers for the efficient solution of very large sparse linear systems [Reid and Scott 2006, 2009]. The new codes are serial multifrontal solvers that are written in Fortran 95 and are designed to allow the original matrix, its factorization, and the bulk of the intermediate data to be held on disk. For a given amount of main memory, this enables much larger problems to be solved than would otherwise be possible. Fortran 95 offers two forms of file access: sequential and direct. Using sequential files is too restrictive for our purposes because the multifrontal algorithm requires that the data of the original matrix be accessed nonsequentially and other data has to be accessed backwards as well as forwards and, in our experience, backwards access is slow. We therefore use direct-access files. A serious

This work was partly funded by the EPSRC Grant GR/S42170.
Authors' addresses: Computational Science and Engineering Department, Atlas Centre, Rutherford Appleton Laboratory, Oxon OX11 0QX, U. K.; email: {john.reid,jennifer.scott@stfc.ac.uk}

limitation of direct-access files is that each file has fixed-length records but we need to be able to read and write different amounts of data at each stage of the multifrontal computation and, thus, to enable the use of direct-access files, we need to buffer the data. Since such buffering may be needed by other applications, we have developed a separate package, called Fortran_Virtual_Memory (which for convenience we will abbreviate as FVM in this article), to do this. Although designed for use by our new multifrontal solvers, FVM is a general-purpose package. Our aim here is to describe the design and key features of FVM and to illustrate its performance.

This article is organized as follows. In the remainder of this introductory section, we give a brief overview of FVM and the language used by FVM. In Section 2, we briefly describe the user interface. Key features of FVM are discussed in Section 3. Numerical experiments are reported on in Section 4 and, finally, some concluding remarks are made in Section 5.

We note that FVM supersedes the Fortran 77 code OF01, which was written by the first author in the early 1980s for use within his out-of-core solver for symmetric positive-definite finite-element problems [Reid 1984] and is now available within the HSL Archive Library.[1] There is a version of FVM in the HSL 2007 Library [HSL 2007] under the name HSL_OF01.

1.1 Brief Overview of FVM

FVM offers facilities for reading from and writing to direct-access files using a buffer (i.e., a work array held in main memory) to avoid actual input/output operations whenever possible (how this is achieved is explained in Section 3.2). A separate buffer is used for each data type that needs to be stored but, because more than one set of data of a given type may need to be held, a single buffer may be associated with more than one direct-access file. The buffer is divided into fixed-length *pages* that correspond to the fixed-length records of the files. All input/output is performed by transferring a single page of the buffer to or from a single record of a file.

In order to provide a friendly interface to FVM, each set of data is addressed as a virtual array, that is, as if it were a very large array. Any contiguous section of the virtual array may be read or written. This is done by first looking for parts of the section that are in the buffer and performing a direct copy of these. For any remaining parts, there may have to be actual input and/or output of pages of the buffer. If room for a new page is needed in the buffer, by default the page that was least recently accessed is written to its file (if necessary) and is overwritten by the new page.

All the entries of the virtual array are regarded as having the initial value zero, but a note is kept of the final page in which data has been placed explicitly so that pages beyond this need not be written to the corresponding file record.

In our early tests with our multifrontal solver, we sometimes found that the virtual array was too big for a single file, either because the disk partition was too small or because the file size was limited by 32-bit addressing. We

[1] http://hsl.rl.ac.uk/archive/hslarchive.html.

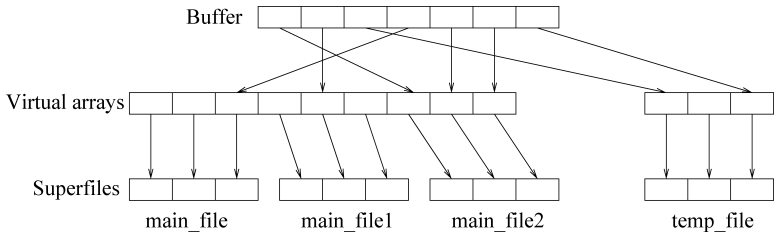

Fig. 1. The buffer, virtual arrays, and files of the superfiles.

therefore added the ability to handle secondary files, which may reside on different devices. We refer to the primary file and its secondaries as a *superfile*.

We illustrate in Figure 1 the correspondences between the pages of the buffer, sections of the virtual arrays, and records of the files of the superfiles for a case of two superfiles associated with the buffer, one with three files and one with one file.

1.2 Language

`FVM` is written in Fortran 95. We have adhered to the Fortran 95 standard except that we use allocatable structure components and dummy arguments. These are part of the official extension that is defined by Technical Report TR 15581(E), [ISO/IEC 2001] and is included in Fortran 2003.

The extension allows arrays to be of dynamic size without the computing overheads of pointers. Addressing is less efficient in code that implements pointer arrays since it has to allow for the possibility that the array is associated with an array section, such as `a(i,:)`, that is not a contiguous part of its parent. Furthermore, optimization of a loop that involves a pointer may be inhibited by the possibility that its target is also accessed in another way in the loop.

The extension also avoids the memory-leakage dangers of pointers since an allocatable array never creates an anonymous object that can be accessed only through another anonymous object. Each allocatable array is either allocated or not allocated.

To allow the package to be used for very large data sets, we selectively make use of *long* (64-bit) integers, declared in Fortran 95 with the syntax `selected_int_kind(18)` and supported by all the Fortran 95 compilers to which we have access. These long integers are used for addresses within virtual arrays.

Fortran 95 offers only sequential-access files and direct-access files; Fortran 2003 offers a third form of file access: stream. This is more flexible but, at the time of writing, no compilers fully support Fortran 2003. Since the Nag Fortran compiler does support stream access, we have constructed a version of `FVM` that uses it, which allows us to consider the advantages and disadvantages of stream access. This is reported on in Section 4.2.

2. DESCRIPTION OF THE USER INTERFACE

Before looking at the key features of `FVM`, we briefly describe its user interface. Full details are given in the accompanying user documentation.

We provide separate versions of the package for double precision and for integer data. It is straightforward to use the double precision version to generate versions for single precision, complex, and double complex data; instructions on how to do this are included in the leading comments in the double precision code. We refer to the type of the data as the *package type*. Each version of `FVM` is provided in the form of a module and uses an array of the package type as a buffer for reading to and writing from one or more virtual arrays. The advantage of having one buffer for several virtual arrays is that the available memory is dynamically shared between them according to their needs at each stage of the computation. It might be desirable in some applications to have a single buffer for two or more data types (for example, for the real and integer data), but this is not possible in standard Fortran 95 without some copying overheads. Thus, if it is required to read and write more than one data type, more than one module must be employed (our multifrontal solvers, for example, employ the real and integer modules). The module procedures are all generic, that is, they can be called by the same name for all types.

For the user's convenience, the buffer and all the data needed to index it and access it efficiently are held in an object of derived type that must be declared by the user and passed as an argument to each of the `FVM` subroutines. This allows the package to be run in a threaded environment with a separate object for each thread. The type for this object is `FVM_data` and is accessible from the `FVM` module. If more than one module is in use, this type must be renamed on all but one of the `USE` statements.

`FVM` has six subroutines that can be called by the user:

—`FVM_initialize` must be called once to initialize the object of type `FVM_data` that holds the buffer and its data.

—`FVM_open` must be called for each superfile that is to be accessed through `FVM`. It gives the superfile an index and opens its file or files.

—`FVM_read` performs reading from a superfile.

—`FVM_write` performs writing to a superfile.

—`FVM_close` should be called for each superfile that is no longer required to be accessed through `FVM` to close the primary file and its secondary files. Any information that belongs to the superfile but is in the buffer is optionally transferred to the files and the files are closed. An option exists to delete the files on closure.

—`FVM_end` should be called to deallocate array components of the object of type `FVM_data` that holds the buffer and its data once no further superfiles are required to be accessed through it.

The object of type `FVM_data` has a number of components that the user can access to control the execution of the package and to obtain information. The control components include parameters that determine the size of each file (the default value is 2^{21}, measured in scalars of the package type) and the number and size of each page of the in-core buffer (the defaults are 1600 and 2^{12}, respectively). Information is available on the the total number of calls made to `FVM_read` and `FVM_write`, and the number of records read and written (i.e., the

actual number of executions of Fortran read and write statements). Note that a call to FVM_read may cause an actual write to occur in order to free a page. Error checking is provided and, in the event of an error, a flag is set, a message is optionally printed, and, where possible, additional information (such as the Fortran STAT parameter) is made available.

The default settings for the controls are provided as initial values in the type declaration, which means that the values will be set when the user declares an object of the type. Our choice of default value for the file size is discussed in Section 3.1 and for the number npage and length lpage of the buffer pages in Section 4.1. If values other than the defaults are required, optional arguments may be used on the call to FVM_initialize to specify the required values.

The derived type also has a component that holds private data, not intended to be accessed by the user. Since Fortran 95 does not permit selected components to be declared as private, we cannot prevent user access and rely simply on not documenting the private data. They include the buffer itself as an allocatable array and several other allocatable arrays that are needed by the package.

3. KEY FEATURES OF FVM

As already explained, FVM performs all actual input/output by transferring pages between its buffer and the records of one or more direct-access files. The buffer is an array of shape (lpage,npage) that is allocated on the call of FVM_initialize. If lpage and npage are chosen sufficiently large and no allocation error is returned, actual input/output to and from a file can be avoided, except on calling FVM_close with a request to keep the files associated with the superfile.

Each superfile is addressed, using 64-bit integers, as a virtual array of rank one with lower bound 1. FVM_read and FVM_write allow any contiguous section of this virtual array to be read and written without regard to page boundaries. The superfile is given an index when it is presented to the package by a call of FVM_open. This index is used to refer to the superfile during subsequent read and write actions and by an eventual call of FVM_close to terminate the connection. The secondary files are given identifiers that consist of the superfile name appended by 1,2,... . FVM_open finds unused units for the primary file and any wanted secondary files and opens them all. If a subsequent call of FVM_write needs more secondary files, units are found for these and they are opened.

3.1 FVM Files

To allow the secondary files to reside on different devices, the user may optionally supply (on the call to FVM_initialize) an array path of path names. For example, the directories /home/sct and /home/jkr might lie on different devices. If path is absent, the behaviour is as if it were present with size 1 and the value (/''/). The *full name* of a file is the concatenation of a path name with the file name, for example, /home/sct/ship_003/reals.data. We need to limit the character lengths of the superfile and path names because the Fortran OPEN statement requires the file name to be specified as a scalar character expression and Fortran 95 lacks a means to construct a character scalar of variable

length. We have chosen to limit the length of path names and superfile names to 400. Since we believe that these lengths should always be sufficient and we are anxious to minimize the number of parameters the user needs to understand and be concerned about, we do not offer the user the option of resetting these lengths. To avoid wasting memory, we record the names in allocatable arrays of character length 400, but avoid copying them except to a temporary scalar variable needed for the file name in an `OPEN` statement

When a new file is opened by `FVM_open` with `size(path)>1`, all the alternatives in `path` are tried until one is found on which a file may be opened, fully written with data (we take the opportunity to initialize the file data to zero), closed, and reopened. If this fails, the next path is tried. Checking that the whole file can be written adds an overhead, but without this check we would run the risk of a later failure from which recovery is not possible. This is because the input/output may be buffered by the system. If there is no room when writing a record for the first time or when unloading the system buffer before closing the file, there is no way to recover the data still in the system buffer. We note, however, that our check cannot fully guarantee that subsequent writes will be successful as other processes may be writing to the same file system.

If the user is sure that there is enough space, the check may be avoided by specifying only one path. Each of the superfiles may still be placed on different devices by suitable choices of file names in the calls of `FVM_open` (but for each superfile, the primary and any secondary files will reside on a single device).

The overheads of these checks on the files make it important that they are not much bigger than necessary. This is why we have chosen the default file size to be 2^{21} scalars of the package type, which on our test platform with a file of type double precision, took 0.16 seconds to establish a local file and 2.0 seconds for a remote file. For machines with 32-bit addressing, there is a limit on the file size. If the system uses signed integers, it is probably $2^{31} - 1$ bytes ($2^{28} - 1$ 8-byte reals). If the system uses unsigned integers, it is probably $2^{32} - 1$ bytes ($2^{29} - 1$ 8-byte reals).

Note that the file size does not limit the size of a superfile, which in practice may consist of many files.

3.2 `FVM` Buffer

The most active pages of the virtual array are held in the `FVM` buffer. For each buffer page, the index of the superfile and the page number within the virtual array are stored. Wanted pages are found quickly with the help of a simple hashing function, and hash clashes are resolved by holding doubly-linked lists of pages having identical hash codes. A special test is made for the page being the one most recently accessed since it can happen that there are many short reads and writes that fit within a single page.

Once the buffer is full and another page is wanted, the least active buffer page is freed. It is identified quickly with the aid of a doubly-linked list of pages in order of activity. By default, whenever a page is accessed, it is regarded as the most active, is removed from its old position in the doubly linked list

and is inserted at the start, unless it is already there. In the factorization phase of our multifrontal codes, this means that the most active part of the multifrontal stack (its top) will be in the buffer while the less active part (its bottom) is written, if necessary, to a file. Furthermore, during the solution phase of the symmetric solver, as many as possible of the later columns of the matrix factor, read during forward substitution, are retained in the buffer for the back substitution.

The user may know that data being read will never be needed again; this happens in the multifrontal method if the data is for a generated element that is being merged into the current frontal matrix. `FVM_read` therefore has an optional logical argument `discard`; its presence with the value `.true.` indicates that the data read will not be needed again. We will refer to a call of `FVM_read` as a *discarding read* if the argument `discard` is present with the value `.true.` and as a *retaining read* if `discard` is absent or present with the value `.false.` For each superfile, a range of discarded entries is kept. If a discarding read touches the discarded range at either end, the discarded range is expanded to include the newly discarded entries; otherwise, the discarded range consists of the newly discarded entries. If the discarded range is overlapped on an `FVM_write`, it is reset to be null (it was not felt worthwhile to identify a part of the old discarded range that is not overlapped). If a page that is held only in the buffer is found to lie in the discarded range, the page is freed without writing its data to the actual file.

Another possibility is that the user knows that data being written will not be needed for a long time; this happens in the multifrontal method if the data is part of the matrix factor and will not be needed until back substitution. `FVM_write` therefore has an optional long integer argument `inactive` whose presence indicates that there is a range including the part being written that will not be needed for a long time. If the part being written is `loc:loc+n-1`, the range is `inactive:loc+n-1` if `inactive < loc` and `loc:max(loc+n-1,inactive)` otherwise. This interface has been chosen to take advantage of the situation where the user is writing a large amount of inactive data gradually in contiguous pieces. We will refer to a call of `FVM_write` as an *active write* if the argument `inactive` is absent and as an *inactive write* if it is present. On an inactive write, any page involved that lies entirely within the inactive range is regarded as the least active of the buffer pages; any other page involved is regarded as having unchanged activity.

Note that a call of `FVM_read` may cause an actual write to occur in order to free a page and that a call of `FVM_write` may cause an actual read to occur if only part of the page is changed.

A "dirty" flag is kept for each page to indicate whether it has changed since its entry into the buffer so that only pages that have been changed need be written to file when they are freed. On each call of `FVM_read` or `FVM_write`, all wanted pages that are in the buffer are accessed before those that are not in the buffer in order to avoid freeing a page that may actually be needed.

The efficiency of reading to and writing from files using `FVM` depends on the size of the buffer and the size of each page. This is discussed further in Section 4.1.

3.3 FVM Read with Mapped Accumulation

As already noted, FVM was designed with the requirements of our multifrontal solvers in mind. Because of the importance of assembly steps in the multifrontal method, we provide an option in FVM_read to add a section of the virtual array into an array under the control of a map. If the optional array argument map is present and the section starts at position loc in the virtual array, FVM_read behaves as if the virtual array were the array virtual_array and the statement

```
read_array(map(1:k)) = read_array(map(1:k))  &
                       + virtual_array(loc:loc+k-1)
```

were executed. Without this, a temporary array would be needed and the call would behave as if the following two statements were executed:

```
temp_array(1:k) = virtual_array(loc:loc+k-1)
read_array(map(1:k)) = read_array(map(1:k)) + temp_array(1:k)
```

FVM_write does not provide the corresponding write feature

```
virtual_array(map(1:k)) = virtual_array(map(1:k)) &
                          + write_array(1:k)
```

because we see no way of doing this more effectively than reading the portion of the virtual array that is subject to change into a temporary array, adding write_array(1:k) into it, and writing it back to the virtual array. This can be better done in the calling procedure because bounds on the values of map(1:k), which determine the size of the temporary array, may be known there and a single temporary array may suffice for many calls.

3.4 Use of BLAS

The real and complex versions of FVM_read and FVM_write use the Level-1 BLAS _copy for copying data from the array argument to the buffer or vice versa. In our test environment, using tuned BLAS (such as the ATLAS BLAS math-atlas.sourceforge.net) was found to be faster than in-line code and users of FVM should always use either vendor-supplied BLAS or other appropriately tuned BLAS. Efficient implementations of the BLAS are obtained by loop unrolling. Fortran code for _copy is available online,[2] and uses 7-fold loop unrolling. Since there is no integer version of _copy, we have written our own subroutine FVM_icopy that uses 7-fold loop unrolling. For FVM_read with the optional argument map present, we have also written special code for each version of the package that uses 7-fold loop unrolling.

3.5 Closing and Reopening Superfiles

When a superfile is closed by FVM_close, the default action is for its changed pages in the buffer to be written to the files and the files to be closed and kept. Trailing pages that still have their initial value of zero are not written. The number of pages in the superfile, excluding these trailing pages, is returned in

[2] http://www.netlib.org/blas.

Table I. Positive-Definite Test Matrices and Their Characteristics (n denotes the order. $nz(A)$ and $nz(L)$ denote the number of entries in A and L, respectively, in millions. *front* denotes the maximum front size. *indicates pattern only. †indicates stored in element form.)

Identifier	n	$nz(A)$	$nz(L)$	*front*	Application/Description
m_t1	97,578	4.926	34.613	1926	Tubular joint
shipsec1	140,874	3.977	40.353	2532	Ship section
crankseg_1	52,804	5.334	33.714	2124	Linear static analysis
troll*†	213,453	6.099	63.678	2643	Structural analysis
af_shell3	504,855	17.562	97.715	2205	Sheet metal forming matrix
inline_1	503,712	18.660	179.269	3261	Inline skater

the argument lenw. There is an option for the files to be deleted. If the virtual array still has its initial value of zero, the file will be empty and it is always deleted.

A superfile that has been closed by FVM_close may be reopened by calling FVM_open and specifying the number of pages in the optional argument lenw, whose value must be as returned by FVM_close.

Once all the input/output through the buffer is complete and the superfiles have been closed, the subroutine FVM_end should be called to deallocate the buffer and the other private arrays that have been used by the package.

4. NUMERICAL EXPERIMENTS

Since our multifrontal solver HSL_MA77 [Reid and Scott 2006] makes extensive use of the proposed virtual memory management system, we use its performance on large positive-definite problems to choose default values of the FVM control parameters and to make comparisons with stream input/output. Our numerical results were obtained using double precision (64-bit) reals on a Dell Precision 670 with 4 GB of RAM. The Nag f95 compiler with the -O option was used together with ATLAS BLAS and LAPACK.[3] Unless stated otherwise, the files used by HSL_MA77 are held locally on our test machine.

The test problems reported on here are a subset of those used by Reid and Scott [2006] and are listed in Table I. Each test example arises from a practical application and is available from the University of Florida Sparse Matrix Collection [Davis 2007]. All times are wall clock times in seconds.

4.1 Effect of Buffer Size on Performance

We first examine how sensitive the performance of HSL_MA77 is to the size of the FVM buffers. As noted in Section 3.2, the number of pages in the FVM buffers and the number of scalars held in each page are parameters (called npage and lpage, respectively) that are under the user's control. The code allows for separate values for the real and integer buffers, but we did not use this option in our tests. When called by HSL_MA77, FVM allocates a real buffer and an integer buffer each as an array of size npage∗lpage. If HSL_MA77 returns an allocation error, the user may be able to rerun his or her problem successfully by reducing npage and/or lpage.

[3]math-atlas.sourceforge.net.

Table II. HSL_MA77 Complete Solution Times for Different Values of npage and lpage

npage	lpage	m_t1	shipsec1	crankseg_1	troll	af_shell3	inline_1
400	2^{10}	20.7	27.6	33.0	39.2	50.8	142.7
100	2^{12}	22.0	27.8	30.9	39.8	53.1	131.1
1600	2^{10}	19.8	26.9	31.8	36.9	50.6	126.5
400	2^{12}	21.4	24.5	30.2	37.0	52.3	120.2
100	2^{14}	22.0	28.2	31.8	37.5	50.7	127.4
6400	2^{10}	20.4	27.9	33.2	36.3	49.0	124.0
1600	2^{12}	19.4	24.1	30.3	35.8	50.5	121.5
400	2^{14}	18.6	26.7	31.5	38.7	49.1	119.8
100	2^{16}	19.2	27.0	33.1	37.6	54.4	126.1

In Table II, we report the complete solution times for different choices of npage and lpage, grouped by buffer size npage∗lpage. We see that, as the buffer size increases (i.e., npage∗lpage increases), the timings generally reduce but the precise choice of npage and lpage is not critical, with a range of values giving similar performances. However, using either a small number of pages or a small page length can adversely effect performance. Based on our findings, we have chosen the default values for these control parameters to be 1600 and 2^{12}, respectively, so that the buffer size is 6.6×10^6 (measured as the number of scalars).

4.2 Stream Access

We have constructed a version of FVM that uses the stream access which is part of Fortran 2003. This allows the file to be positioned by "file storage units" (probably bytes) starting at position 1 and allows any number of variables to be accessed. On the assumption that the system will provide buffering for short transfers, this version of FVM performs no buffering of its own, which significantly simplifies the code and reduces the number of parameters. Other aspects of the code are retained and remain important. It makes sure that there is room on the device for a file when opening it and tries other paths if necessary. It allows superfiles to span many files, perhaps not all residing on the same device.

Besides code simplicity, the stream access approach has the advantage that the system will be able to share buffering for files holding data of different types. Also, there is no way in Fortran to specify that system buffers should not be used for direct-access transfers, so the direct-access version cannot avoid additional memory being used for two sets of buffers and additional copying between them.

On the other hand, stream access has the disadvantages of there being no way to specify the size of the buffer or the size of the data transfers, and there is no way to tell the system that the data will either not be needed again or not needed for some time.

In Table III, we compare the complete solutions times for HSL_MA77 run on our test problems using FVM with default values of npage and lpage with those with the files replaced by arrays held in core and those with the stream-access version of FVM. It may be seen that stream-access version is always slower than the out-of-core and in-core versions and indeed the margin by which it is slower is greater than the margin by which the direct-access version of HSL_MA77 is

Table III. HSL_MA77 Complete Solution Times

	m_t1	shipsec1	crankseg_1	troll	af_shell3	inline_1
Out-of-core	19.4	24.1	30.3	35.8	50.5	121
In-core	15.2	21.1	25.0	31.6	40.1	97.8
Stream	23.3	28.3	32.0	48.0	55.0	189
Remote files	56.1	75.6	74.9	101	144	280

slower than the in-core version. Furthermore, our experience has been that the speed of the stream-access version is more sensitive to other activity on the machine. In Table III, we also include timings for the out-of-core version of HSL_MA77 when the files are held remotely (i.e., on a file system not belonging to the test machine). It is clear that this has a significant adverse effect on the efficiency of HSL_MA77.

5. CONCLUDING REMARKS

We have explained the design of the Fortran virtual-memory system FVM, which uses direct-access files for any actual input/output that is needed. It is able to handle several files, perhaps on different devices, as a single superfile. FVM is used to support input/output operations in our new out-of-core multifrontal packages HSL_MA77 and HSL_MA78 and we have used the speed on these packages on large actual applications to choose default values for the page and buffer sizes. We have constructed a version of FVM that uses stream access instead of direct access and in our tests found this caused a deterioration in the performance of HSL_MA77, with a margin that ranged between about 5% and 55%. As a result, we do not currently plan to make the stream-access version available.

The version of FVM that is used by HSL_MA77 and HSL_MA78 is called HSL_OF01. Together with the multifrontal solvers, it is included in the HSL mathematical software library [HSL 2007]. All use of HSL requires a license. Licenses are available without charge to individual academic users for their personal (noncommercial) research and for teaching; for other users, a fee is normally charged. Details of how to obtain a licence and further details of all HSL packages are available online.[4]

ACKNOWLEDGMENTS

We are very grateful to our colleagues Nick Gould and Iain Duff for their help. Nick made constructive suggestions during the design of FVM and Iain read a draft of this article and suggested improvements to the presentation. It was the very helpful reports of the anonymous referees of our article on HSL_MA77 that led us to make comparisons with stream input/output and we would like to thank them for this. We would also like to thank the anonymous referees of this article for their constructive comments.

REFERENCES

DAVIS, T. 2007. The University of Florida sparse matrix collection. Tech. rep. University of Florida, Gainesville, FL. http://www.cise.ufl.edu/~davis/techreports/matrices.pdf.

[4]www.cse.clrc.ac.uk/nag/hsl/.

HSL. 2007. A collection of Fortran codes for large-scale scientific computation. http://hsl.rl.ac.uk/.

ISO/IEC. 2001. TR 15581(E): Information technology—Programming languages—Fortran—Enhanced data type facilities, (2nd ed.), edited by Malcolm Cohen. ISO/IEC Tech. rep. ISO, Geneva, Switzerland.

REID, J. 1984. TREESOLV, a Fortran package for solving large sets of linear finite-element equations. Report CSS 155. AERE Harwell, Harwell, U.K.

REID, J. AND SCOTT, J. 2006. An out-of-core sparse Cholesky solver. *ACM Trans. Math. Softw.* To appear. Tech. rep. RAL-TR-2006-013, Rutherford Appleton Laboratory.

REID, J. AND SCOTT, J. 2009. An efficient out-of-core multifrontal solver for large-scale unsymmetric element problems. *Int. J. Numer. Math. Eng. 77*, 7 (Feb.). 901–921.

Received November 2007; revised April 2008; accepted August 2008