Array variables and its allocation in Fortran on OSX

Koun Shirai
ISIR, Osaka University
8-1 Mihogaoka Ibaraki Osaka 567-0047
koun@sanken.osaka-u.ac.jp
15 Jun 2006

Abstract
In Fortran execution on Mac OSX, one of repeated questions is a problem of memory allocation. A year ago, I had tackled this problem, and gotten some idea to solve this. The following is a summary of this understanding. Consistent and coherent usage of memory in Fortran and terminal commands are required. Although the mechanism of OSX is investigated, the basic theory and conclusions are equally valid to others.

1 Introduction
A big problem of working with Fortran programs on OSX is memory management. When executing a program of large size, we sometimes encounter segmentation fault, even though the size of array is less than the physical limit of the machine. Table 1 shows such an example, where a simple matrix multiplication is examined as a function of the matrix size \( n \). The size of matrix is explicitly declared, as

```
parameter(n=2000)
dimension tarray(2), a(n,n), b(n,n), c(n,n)
```

Actually, it is not special to MacOSX, but IBM machine (IBM Powr4 with AIX v5.1) has the same result. On the other hand, pentinum machine (Pentium IV with Linux rhlinux01) does not have this problem. Table 2 shows comparison of this simple test among different platforms. You can see that pentinum machine handle matrices up to the size 7000, in spite of small stack size. This indicates that effects of memory manipulation by ulimit is not unique over different platforms or compilers. Without careful use, you would fall in more confusion in program development.

As a Fortran programmer, we are free to define any array variables as particular type. such as assumed-size array. But, which type is appropriate for particular situation? Even if we can find a guideline for this issue, could it be applied equally for other platforms. If not, such a guideline becomes eventually useless.

As a normal user, we can manage the memory by using "ulimit -d" or "ulimit -s", allocation of data region and stack region. But, which portion of memory in Fortran program are we allocating by these command? Roughly speaking, we may answer that local variables and temporal variables are allocated in the stack region. This is the mechanism of working subroutines. But, this rough guideline does not work. The actual situation is so complicated, as seen later. In this note, I intend to give a guideline the memory usage in
Fortran on MacOSX, based on my experience. I cannot studiedly say that I am an expert of system programmer. A huge references of OSX are outside of my library. Such documents are too complicated and even irrelevant for typical Fortran users to read. The following is obtained by my painful experience of trial and error approach.

In many respect, MacOX is not a typical platform on which Fortran programs are executed. The system is not intended for such an application as the major market. Darwin is very special. But, at the same time, I can say that the basic idea of OSX is no longer special among contemporary workstations. Because the application area of other platforms is basically the same as OSX. The scientific application is only a part of their business.

Table 1: Simple test on MacOSX with real memory of 4 GB. SF indicates Segmentation Fault in execution time. The value of memory is recorded from Activity Monitor and in indicated in MB.

<table>
<thead>
<tr>
<th>RAM</th>
<th>ulimit stack</th>
<th>time</th>
<th>MFLOPS</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absoft ProFortran</td>
<td></td>
<td>500</td>
<td>83</td>
<td>6.51</td>
</tr>
<tr>
<td>1000</td>
<td>53</td>
<td>38</td>
<td>23.68</td>
<td></td>
</tr>
<tr>
<td>1500</td>
<td>216</td>
<td>31</td>
<td>52.29</td>
<td></td>
</tr>
<tr>
<td>1700</td>
<td>–</td>
<td>–</td>
<td>SF</td>
<td></td>
</tr>
<tr>
<td>IBM XL Fortran</td>
<td></td>
<td>1000</td>
<td>35</td>
<td>56</td>
</tr>
<tr>
<td>1500</td>
<td>248</td>
<td>27</td>
<td>44.53</td>
<td></td>
</tr>
<tr>
<td>1700</td>
<td>–</td>
<td>–</td>
<td>SF</td>
<td></td>
</tr>
</tbody>
</table>

2 Memory management

Management of memory is too complicated. In particular, the contemporary parallel processing renders the memory management more complicated. I think that the management should be considered at least at three different classes; OS level, compiler level, and FORTRAN level.

2.1 Memory management as a UNIX system

As a unix user, we can manage the memory by

```
ulimit -s
ulimit -d
```

In this way, we can set the data size to be hardware limit. Of course, ”hardware limit” does not mean exactly the hardware limit, say, not 4GB for 4GB machine. However, we are satisfied with this, because we can manage a large-size of data. On the other hand, the situation is different for the stack. You may soon find that it cannot be set more that 64 MB! It is really the system limitation. For more details, see Appendix A. This default setting for the stack size is reasonable in the modern system. Recall that it is...
common for one machine to execute many processes simultaneously. If the default stack size is large, even tiny programs which do not use much memory consume large memory. This is waste of memory. Large memory space should be reserved only when it is really needed. This is the essential reason of having of the heap zone.

To circumvent the problem of stack size, there are some methods such as a linker option (see Appendix B). However, such a command is not recommended, unless you are professional at the system program. It is recommended to manage the memory within Fortran level as far as possible, leaving the system untouched.

2.2 Memory construction of OSX

Before discussing memory allocation in Fortran, we must lean a little bit the memory construction of OSX. Concerning this issue, I am an amateur. It is difficult to find what we want to know in huge volumes of OSX references. The following is my speculation based on my experience.

**Original management in Classic Mac** In the traditional Mac era, the memory space was classified by the stack and heap region.

**stack** The physical memory space is contiguous, and is shrunk or expand according to the program execution. The last comes will get out first.

**heap** When execution, the space is allocated and deallocated. Accordingly, fragmentation in the memory space may occur.

The two spaces grown from the top and bottom of the memory space, and users should take care in order not to overlap these regions.

Because of difference in major applications from other workstations, usage of these two region might be different from those of the usual scientific computations. Local variables and temporal variables are stored in the stack. Pointer-type variables are stored in the heap. Pointer-type variables are not popular in Fortran community (in my opinion), while are frequently used in the developing tool of Mac applications. In "Inside Mac", most of description are concerned with handling the pointer.

In Fortran, "local or temporal" suggests those variables in subroutines. However, it is not clear whether array variables in the main program or those declared in common statement is classified as this category or not. I think that in Mac world these variables are also classified as local variable, even though seems global operationally.\(^1\)

---

\(^1\)I am not sure if this is really true for all the compilers. Some compiler might allocate common variables in the heap. I think that decision of allocation of specific-type variables to specific region of the memory is highly
Only those variables which are explicitly allocated are stored in the heap, except program code itself. This suggests that almost all variables of Fortran code in Mac are allocated in the stack, unless explicit allocation is made.

OSX The above is my understanding on the memory manager of MacOS. Now, MacOS has been significantly changed to OSX, based on Darwin. I don’t know the situation of the memory management in the above sense. It is difficult to see such a description in huge references of OSX. Fortunately, I heard from Apple technical staff about this issue (Appendix B). Peters’ note is pretty good, and is probably enough for the present purpose. Based on his note, I speculate the memory construct of OSX as Figure 1.

Stack may be allocated even in the data region, when multiprocessing. This issue is too complicated. Let stop considering multiprocessing. Then, stack is allocated in the free region. Roughly speaking, ”data region” in his note is the heap region in the traditional MacOS, while ”free region” is the stack region. In this situation, user can expand the data size up to 2GB, by issuing ulimit -d unlimited. By issuing ulimit -s, we actually control the stack in the free region. The upper limit is fixed to be 64 MB, as described earlier. 64 MB Probably, most of Fortran users feels that this is too small. A matrix of $2000 \times 2000$ of double precision exceeds this limit. But, from the developer’s standpoint, remember that OSX is a unix system, i.e., multiuser system. Each process (or each user?) possesses its own stack. Accordingly, the size must be limited in that way.

3 memory allocation in Fortran

Now, we discuss the memory allocation in Fortran.

3.1 various data types in Fortran

Before discussing this issue, we must distinguish different types of data array in Fortran language. The details are described in [2, 3]. In that textbook, there are a couple of types of array,

1. Explicit-Shape Array

   \[
   \text{REAL, DIMENSION(10,10) :: A}
   \]

2. Dummy Array

   \[
   \text{SUBROUTINE sub(A, n, m)}
   \]

   \[
   \text{INTEGER, INTENT(IN) :: n, m}
   \]

   \[
   \text{REAL, DIMENSION(n,m) :: A}
   \]

   Dummy array is further classified as the following way.

   (a) Explicit-shape dummy array
   (b) Assumed-shape dummy array
   (c) Assumed-size dummy array

3. Automatic Array

   \[
   \text{SUBROUTINE sub(n, m)}
   \]

   \[
   \text{INTEGER, INTENT(IN) :: n, m}
   \]

   \[
   \text{REAL, DIMENSION(n,m) :: A}
   \]

4. Deferred-Shape Array

   \[
   \text{REAL, ALLOCATABLE :: A}
   \]

   \[
   \text{ALLOCATE( A(10,10), STATUS=istat)}
   \]

\footnote{a freedom of the compiler’s developer.}

\footnote{Later, I found a good document in Apple’s home page, refer to [1].}
Each type has its merit and demerit operationally. For deferred-shape array, it is a little bit annoying to declare it and explicitly allocate that variable.

Table 3: Simple test on MacOSX by using allocatable variables. The memory size is indicated in MB.

<table>
<thead>
<tr>
<th>n</th>
<th>time</th>
<th>MFLOPS</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>39</td>
<td>50.28</td>
<td>22.70</td>
</tr>
<tr>
<td>2000</td>
<td>1735</td>
<td>9.22</td>
<td>92.36</td>
</tr>
<tr>
<td>3000</td>
<td>9170</td>
<td>5.89</td>
<td>206.45</td>
</tr>
<tr>
<td>4000</td>
<td>-</td>
<td>-</td>
<td>367.06</td>
</tr>
<tr>
<td>5000</td>
<td>40240</td>
<td>6.21</td>
<td>573.02</td>
</tr>
<tr>
<td>6000</td>
<td>-</td>
<td>-</td>
<td>824.79</td>
</tr>
<tr>
<td>7000</td>
<td>114207</td>
<td>6.01</td>
<td>1.10 G</td>
</tr>
<tr>
<td>8000</td>
<td>170559</td>
<td>6.00</td>
<td>1.43 G</td>
</tr>
<tr>
<td>9000</td>
<td>-</td>
<td>-</td>
<td>1.81 G</td>
</tr>
<tr>
<td>10000</td>
<td>-</td>
<td>-</td>
<td>2.24 G</td>
</tr>
</tbody>
</table>

IBM XLFortran on Xserve

<table>
<thead>
<tr>
<th>n</th>
<th>time</th>
<th>MFLOPS</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>27772</td>
<td>9.00</td>
<td>573.2</td>
</tr>
<tr>
<td>6000</td>
<td>-</td>
<td>-</td>
<td>824.0</td>
</tr>
<tr>
<td>7000</td>
<td>78775</td>
<td>8.71</td>
<td>1.10 G</td>
</tr>
<tr>
<td>8000</td>
<td>116363</td>
<td>8.80</td>
<td>1.43 G</td>
</tr>
<tr>
<td>10000</td>
<td>224968</td>
<td>8.89</td>
<td>2.24 G</td>
</tr>
</tbody>
</table>

3.2 use of different types of array data

Managing these different types is complicated problem, and the reader is referred to [2, 3]. In my experience with OSX, those variables are allocated in the following scheme.

\[
\begin{align*}
\text{Explicit} & \rightarrow \text{Shape Array} \\
\text{Dummy Array} & \rightarrow \text{stack} \\
\text{Automatic Array} & \rightarrow \text{data} \\
\text{Deferred} & \rightarrow \text{Shape Array} \\
\end{align*}
\]

(1)

According to this, it is recommended that, whenever a large-size matrix is handled, *it is declared as deferred-shape array*. It is not enough to declare it in module, if it is not deferred-shape type. This may be also true for common variables in COMMON statement. In my experience, in some compiler allocates differently. For example, automatic array is allocated in the data region in Absoft, while, is in the stack in XLF.³ However, this guideline is the safest way to manage large-size matrices. Table 3 shows the result of revised program by using allocatable variables.

As said at the beginning, although MacOS is a very special system, it is no longer special from viewpoint of memory management. In most of platform, default size of their stack is not enough to load large-size matrices. If your program does not work on OSX, it is also likely true for other systems, if memory management is concerned. Memory manipulation by ulimit is, therefore, not recommended, if you are developing codes.

4 Conclusion

It is common that the stack size is not large. You can expand it by using ulimit. This way of manipulation of memory is not recommended. Even if you can get a desired result, it could be wrong with other platforms or compilers. The safest way of manipulation is use of allocatable variables.

Appendix

Information of this appendix is gathered from mailing list fortran-dev and scitech.

³In some cases, we can handle the heap size by specific compiler options. For example see Appendix C.
A. stack size

Usually, unix users can control the size of stack by issuing

```
ulimit -s unlimited
```

However, the largest size is limited up to 65 MB. This is the limitation for normal users. Even with administrator privilege, this limitation cannot be changed in this way. Someone reported that recompile of the kernel was attempted, without success.

(23 Oct 2004)
Edward Mansell
CIMMS/University of Oklahoma/National Severe Storms Lab

ld has a stacksize option, but that practically only takes you up to about 256MB on the linking command with xlf, for example), which may or may not be related to the maximum that Josh sees even after hacking the kernel. Trying to set the stacksize higher via the linker results inevitably in a seg fault. Rather annoying for porting code that uses a lot of automatic array space. Let’s hope it gets fixed in Tiger. (Please!)

For normal users, it is too difficult to overcome this limitation. For this problem, Apple’s staff replied as,

(23 Oct 2004)
Warner Yuen
Research Computing Consultant
Apple Computer

There is a hard limit on stacksize limit on Mac OS X. If you use:

```
ulimit -s hard
```
you’ll get the following:

```
stack size (kbytes, -s) 65536
```

Mac OS X uses other portions of the memory for its own stuff. You may want to consider moving things to malloc where there’ll be much more room.

The last option in his comment is not recommended, unless you are familiar with system programming.

B. memory construction

A more detailed description is given by another Apple’s staff.

(23 Oct 2004)
Steve Peters
Numerics and Vectorization
Apple Computer

I just created a 1Gb stack with:

```
gcc ... -Wl, -stack_size -Wl,0x40000000 -Wl,-stack_addr -Wl,0xf0000000
```

N.B., every program consists of ”program text” (i.e. the code for instructions), ”data” (i.e. static variables, constants, arrays, etc.) and ”stack”. In addition, there are system libraries for I/O, math, threading, etc. All of these must be positioned in non-overlapping ranges of the 4Gb address space. The default scheme for Mac OS X puts TEXT and DATA segments starting from 0x00000000 up to 0x8ffffff, shared
system libraries and read-only system data from 0x90000000 to 0xffffffff, and per-application window system resources up above 0xf0000000. So, that leaves 0xb0000000 through 0xffffffff – that’s the 1Gb range I used above.

It may be possible to push a bit higher into the addresses above 0xf00000000 and further down into the addresses below 0xb00000000, but you do so at your own risk.

Furthermore, the "heap" for "malloc’d" allocations sits in the same range as TEXT and DATA,(from 0x00000000 up to 0x8fffffff). When additional threads are created, their *stack* is allocated from the ”heap”.

In his note, we see that the size of stack can be handled by a linker option, Wl, .... Again, this option is not recommended. This option should be used only when you know what you are doing with memory management.

C. An example of handling heap

(16 Jun 2006)

M. Haylock

Note that on XLF you can tell the compiler too use the heap where possible using ”-qsmallstack=dynlenonheap”. I was trying to solve the memory issues a year ago and wrote the following program. This fails on the automatic array allocation with default compiler options but works fine with the above switch.

```
program test_memory
! test allocation of large arrays using 3 different methods
implicit none
integer n_explicit,n_allocatable,n_automatic
parameter(n_explicit=100, n_allocatable=100 , &
 n_automatic=100)
! size in MB
character a(n_explicit*1024*1024)
character, allocatable :: b(:)
integer err
write(*,*)'allocating'
allocate(b(n_allocatable*1024*1024),stat=err)
if(err/=0)stop 'allocation error'
write(*,*)'allocating done'
write(*,*)'automatic'
call allocate_big(n_automatic)
write(*,*)'automatic done'
deallocate(b)
stop 'done'
end
!!! subroutine allocate_big(n)
implicit none
integer n
character c(n*1024*1024)
```

References

