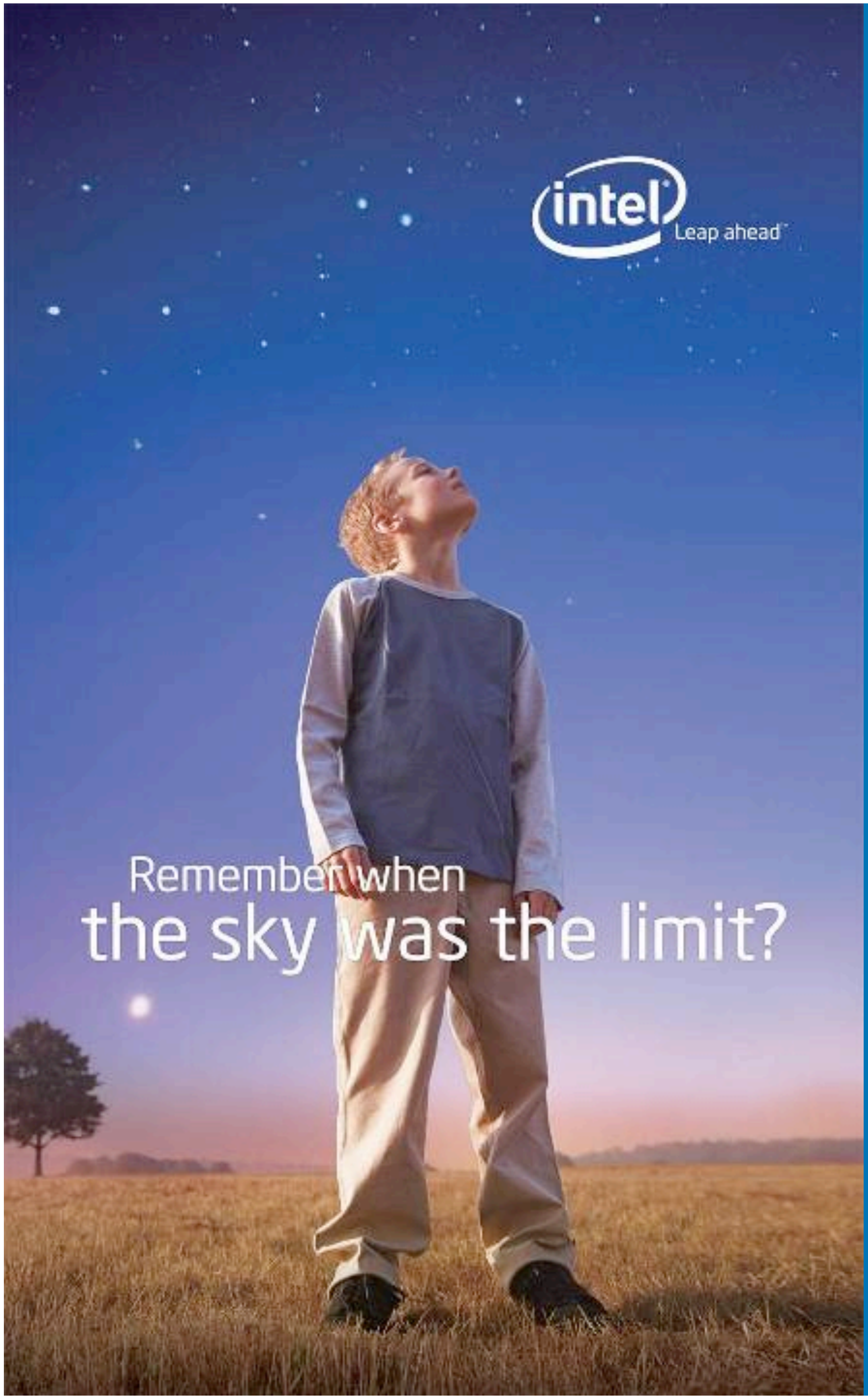

# Fortran Compiler Use of Temporaries

Improving Performance, Reducing Stack Utilization

# Problems and Concerns: Agenda

- Stack Application runs out of stack and aborts
- Application creating temporary copies of actual arguments before procedure call.
- Application creating temporary copies of arrays because of Fortran 95 statements or array syntax
- OpenMP Considerations

# General Stack Exhaustion and Increasing Stack Space

# Intel Fortran Compiler Stack Usage

- Driven by array temporaries
- OpenMP puts a heavy demand on stack (all thread PRIVATE data is put on stack)
- -heap-arrays option added, v9.1 Aug 06
  - Linux: 9.1.037 and later
  - Windows: 9.1.029 and later
  - Mac OS* X: present in all ifort versions

# Symptoms and Solutions to Stack Exhaustion

- Symptoms:
  - Linux: process aborts with SEGV (sigsegv), segmentation fault
  - Mac OS X: process aborts with "illegal instruction"

- Solutions/Workarounds
  - Use 9.1 or greater compiler option –heap-arrays
  - Linux: unlimit stack via C system call
  - Linux, Windows, Mac OS X: Use loader options to increase stack size and possibly stack starting address
  - System: Increase system wide user shell stack limit
    - Via default system /etc/login /etc/csh.cshrc
    - Via kernel params and custom kernel builds
  - User: Increase stack size in user shell
    - User login scripts
    - Setting stack size just before running (wrapper scripts)

# -heap-arrays

- -heap-arrays[:size]
- Default is **no** -heap-arrays
- Optional [:size] – arrays of size or smaller are stack allocated, larger arrays are heap allocated
- From Release_Notes: “May have slight performance penalty”
  - Varies by application
  - Stack memory management is fast and simple (allocate/deallocate straightforward, fast)
  - Heap management: large amounts of allocations/frees of differing sizes can frag heap, impact performance.
  - Use [:size] to restrict to large allocations and avoiding fragmentation

# -heap-arrays

- -heap-arrays affects automatic arrays and temporaries only. For example:

```
RECURSIVE SUBROUTINE F( N )

INTEGER :: N

REAL :: X ( N )                          ! an automatic array

REAL :: Y ( 1000 )                     ! an explicit-shape
   local array on the stack
```

Array X in the example above is affected by the heap-array option. **Array Y is not**.

# Linux:  unlimiting stack via C system call

```c
#include <stdio.h>         // perror
#include <stdlib.h>        // exit
#include <sys/time.h>      // setrlimit
#include <sys/resource.h>  // setrlimit
#include <unistd.h>        // setrlimit

void unlimit_stack_(void) {
    struct rlimit rlim = { RLIM_INFINITY, RLIM_INFINITY };

    if ( setrlimit(RLIMIT_STACK, &rlim) == -1 ) {
        perror("setrlimit error");
        exit(1);
    }
}
```

# Linker/Loader Option for Stack Size

- Adds stack size change to executable image
- Loader will ignore shell limits and give process the requested, non-default, stack size

Example: Increase to 256MB on Mac OS X:

```
ld –stacksize 0x10000000 –o foo foo.o
```

ifort:

```
ifort –o foo –Wl,-stack_size,0x10000000,-stack_addr,0xc0000000 foo.f
```

# Temporary Creation on Procedure Call

# Case: Local Variables

```
subroutine sub( a )
real(8) :: a(1000,1000)
real(8) :: temp(1000,1000), work(1000,1000)
```

- Local arrays **`temp`** and **`work`** allocated on stack (assuming default options)
- Work arounds:
  - SAVE atttribute will cause allocation in heap
  - **`-save`** compiler option (same effect) but affects entire source file(s)

- Default: default of –auto (same as –automatic) default compiler option

# Case: Array Temporaries in Fortran Automatic Arrays

```
subroutine sub( f, x, y, z )

integer :: x, y, z

real(8) :: f(x,y,z)  !...argument

real(8) :: temp(x,y,z) !stack alloc'ed automatic array
```

- Replace with allocatable array – allocation occurs in heap

```
Subroutine sub( f, x, y, z )

...

real(8), allocatable :: temp(:,:,:)

allocate ( temp(x,y,z) )
```

# Case: Array Temporaries in Fortran Passing Non-contiguous Array Sections

- If passing a noncontiguous array section to another routine, have the called routine accept it as a deferred-shape array

- an explict INTERFACE is required

- Example:  BEFORE (using explicit-shape dummy )

```
real(8) :: f(1800,3600,1)

external sub                  subroutine sub( f )

call sub( f(1:900,:,:)        real(8) :: f(900,3600,1)
```

Sub is expecting a contiguous array 900x3600x1
a temp is created on entry (gather) and copied back on exit (scatter)

# Continued: Array Temporaries in Fortran Passing Non-contiguous Array Sections

- Explicit interface and assumed shape arrays avoid the temporary

```
real(8) :: f(1800,3600,1)

interface

   subroutine sub(f)

   real(8) :: f(:,:,:)

   end subroutine sub

end interface

call sub( f(1:900,:,:)
```

```
subroutine sub( f )

real(8) :: f(:,:,:)

...

end subroutine sub
```

Downside: within ‘sub’, the optimizer must assume that ‘f’ might be non-contiguous

# Continued: Array Temporaries in Fortran Passing Non-contiguous Array Sections

-gen-interfaces option can be used to generate INTERFACE blocks for SUBROUTINES and FUNCTIONS in your source

- Creates 2 source files for each:
  - A <subroutine>_mod.f90 file with the INTERFACE inside a MODULE
  - A <subroutine>_mod.mod file (the MOD file for the above)
  - Placed in –module <dir>, or –I <dir>, or in current directory

- CHECK YOUR WORK: -check arg_temp_created
  - Runtime check to print warnings when temporaries are created at procedure calls.

# Temporaries Creation By Fortran Statements and Intrinsics

# Case: Array Temporaries in Fortran WHERE statement

- WHERE statement will always create an array temporary for the array expression:

```
real(8) :: f(1800,3600)

!...requires 8x1800x3600 = 51,840,000 byte temp array

where ( f .gt. 0 )

 f = log10(f)

else where

 f = -1.0

end where
```

- **Only workaround is to avoid WHERE (explicitly write DO loop with conditional) – not advised**

# Case:  Array Temporaries Caused by Cray Pointers

- Cases vary: in general, anytime the compiler cannot determine if there is overlap in the RHS and LHS expressions

- Cray pointers – compiler errs on the side of safety

```
pointer (pb, b)
pb = getstorage()
do i = 1, n
b(i) = a(i) + 1    !...assumes b may overlap with a, makes
 temporary of ‘a’
enddo
```

- –safe-cray-pointers JUDICIOUSLY

```
pointer (pb, b)
pb = loc(a(2))
do i=1, n
b(i) = a(i) +1      !... –safe-cray-pointers will avoid temp.
 but give wrong results
enddo
```

# Case: Array Temporaries Created by Fortran Pointers

```
real, pointer, dimension (:,:) :: xptr, yptr

real, target :: z(100,100)

allocate ( xptr(100,100) )

allocate ( yptr(100,100) )

…

xptr = yptr*2    !...the compiler must assume overlap

z = xptr * yptr  !...X or Y or both could point to Z
```

# Continued: Array Temporaries Created by Fortran Pointers

When a pointer-based array appears in an assignment statement on the LHS of the assignment, and a TARGET or another POINTER appears on the RHS, the compiler will assume a possible overlap condition and will create array temporaries.

Similarly, when a TARGET appears on the LHS and a POINTER appears on the RHS expression, a temporary is created. Again, any time there is a possible overlap in the LHS and RHS expression, the compiler will choose the safest path and create an array temporary.

In general ONLY use POINTER-based arrays where absolutely necessary. If you can use ALLOCATABLE arrays instead, do so

# Array Temporaries in Fortran
# Others (work in progress)

- Array-valued function procedures return values on the stack
  - Only work around is to convert to subroutine procedures and pass the array as an argument ( INTENT OUT or INOUT )

- Intrinsics often use array temporaries

- RESHAPE

- MERGE

- SUM

- (others (tbd))

# Array Syntax and Temporaries

- Does array syntax create temporaries?
- If the compiler is doing it's job, NO. (caveat: we have been finding and fixing such cases over the years)
- If you find such a case, please open a bug report

# OpenMP Stack Considerations

# -openmp Interaction with –heap-arrays

- -openmp will cause the compiler use slightly different behavior for –heap-arrays
- Procedure local data with –heap-arrays and –openmp are STACK allocated (therefore, thread-safe) – must explicitly override with SAVE attribute to get on heap
- Automatic arrays:  descriptor allocated on stack, data allocated in heap (thus, also thread-safe).
- OpenMP puts a heavy load on stack, threadprivate variables need stack allocation
- Use stack-increasing methods – you will need much more stack than an non-OpenMP application

# Summary Recommendations

- Code to avoid temporaries on procedure calls, use –check arg_temp_created to verify

- -heap-array:<size> may be used for codes needing large array temporaries

- Requires 9.1.x or greater compilers since August 2006

- 9.0 and older compilers: Use either loader options and/or setrlimit() to bypass shell stack size limitations

- When passing array sections, use assumed shape arrays and explicit INTERFACE