

```

main_program.f90 : "PROGRAM main_program USE searchutils USE omp_lib ! Include OpenMP library
module IMPLICIT NONE INTEGER, PARAMETER :: N2 = 10000000 REAL(8), dimension(10) ::
arr_test=[1.d0,2.d0,4.d0,5.d0,6.d0,7.d0,9.d0,10.d0,11.d0,15.d0] REAL(8) :: arr2(N2) REAL(8) :: x, t_start,
t_end INTEGER :: idx, n n = size(arr_test,1) ! Number of elements in the array. x = 7.d0 ! Search for value
7.0 in the array. idx = linearsearch(arr_test,n,x) print*, "Index computed with linear search: ", idx idx =
binarySearch(arr_test,n,x) print*, "Index computed with binary search: ", idx ! Uncomment lines 27-54 to
check the behavior with sorted arrays. ! Use here these two cases to evaluate performance. print*, " -----
----- " print*, "Testing on a sorted array" ! Call the fillSortedArray subroutine to fill arr2 with
sorted values. CALL fillSortedArray(arr2) x = arr2(N2-1) ! Value of interest: Second to last element. n =
SIZE(arr2,1) ! Measure the CPU time of this linearsearch function. t_start = omp_get_wtime() !CALL
CPU_TIME(t_start) ! Start measuring time here idx = linearsearch(arr2,n,x) ! Complete the CPU TIME
measurement t_end = omp_get_wtime() !CALL CPU_TIME(t_end) ! END TIME print*, "Linear CPU TIME
WAS: ",t_end-t_start ! Idx must be the second to last element. print*, "Index computed with linear search:
", idx, N2-1 print*, "was the value found?: ", arr2(idx)==x ! Measure the CPU time of this binarysearch
function. CALL CPU_TIME(t_start) ! Start measuring time here idx = binarysearch(arr2,n,x) ! Complete
the CPU TIME measurement CALL CPU_TIME(t_end) ! END TIME print*, "Binary CPU TIME WAS:
",t_end-t_start print*, "Index computed with binary search: ", idx, N2-1 print*, "was the value found?: ",
arr2(idx)==x ! Uncomment lines 56-71 to the behavior with unsorted arrays. print*, " -----
-- " print*, "Testing on an unsorted array" ! Call the fillUnsortedArray subroutine to fill arr2 with
unsorted values. CALL fillUnsortedArray(arr2) x = arr2(N2/2-1) ! Value of interest: middle element -1. n =
SIZE(arr2,1) idx = linearsearch(arr2,n,x) print*, "Index computed with linear search: ", idx print*, "was
the value found?: ", arr2(idx)==x CONTAINS ! Fill an array with sorted values. SUBROUTINE
fillSortedArray(array) IMPLICIT NONE REAL(8):: array(:) INTEGER :: i DO i=1,SIZE(array,1) array(i) =
(i*3.d0) END DO END SUBROUTINE fillSortedArray ! Fill an array with unsorted values. SUBROUTINE
fillUnsortedArray(array) IMPLICIT NONE REAL(8):: array(:) INTEGER :: i DO i=1,SIZE(array,1) array(i) =
(-1.d0)**(i) * i*2.d0 END DO END SUBROUTINE fillUnsortedArray END PROGRAM main_program "

```



Python

It seems you've shared a Fortran program that uses linear and binary search algorithms to find specific elements in arrays. This program also measures the performance of these searches on both sorted and unsorted large arrays. If you need help with this Fortran code or if you're looking to translate it into Python, please let me know how I can assist you!

You

```

searchutils.f90 : "MODULE searchutils implicit none contains ! Description: Function that finds the
location (idx) of a value x ! in an array using the linear search algorithm. !! Find idx such that arr(idx) ==
x ! FUNCTION linearSearch(arr, n, x) RESULT(idx) REAL(8) :: arr(n) ! Array to search INTEGER :: n !
Number of elements in array. REAL(8) :: x ! Value to search for in array. INTEGER :: idx ! Result of the
search. [arr(idx) == x] INTEGER :: i idx = -1 ! Initialize index to -1 (indicating not found) !$omp parallel
do shared(arr, n, x) private(i) DO i = 1, n IF (arr(i) == x) THEN idx = i EXIT END IF END DO !$omp end
parallel do END FUNCTION linearSearch ! Description: Function that finds the location (idx) of a value x !
in a sorted array using the binary search algorithm. !! Find idx such that arr(idx) == x ! FUNCTION
binarySearch(arr, n, x) RESULT(idx) REAL(8) :: arr(n) ! Array to search INTEGER :: n ! Number of
elements in array. REAL(8) :: x ! Value to search for in array. INTEGER :: idx ! Result of the search.
[arr(idx) == x] INTEGER :: low, high, mid ! Indices for binary search idx = -1 low = 1 high = n DO WHILE

```

```
(low <= high) mid = (low + high) / 2 IF (arr(mid) == x) THEN idx = mid EXIT ELSE IF (arr(mid) < x)
THEN low = mid + 1 ELSE high = mid - 1 END IF END DO END FUNCTION binarySearch END MODULE
searchutils "
```



Python

Thanks for sharing the Fortran module **searchutils**. This module defines two search functions, **linearSearch** and **binarySearch**, that are used to find the index of a given value in an array. The **linearSearch** function scans each element of the array until it finds the target value or reaches the end of the array. The **binarySearch** function, on the other hand, is optimized for sorted arrays and uses a divide-and-conquer approach to locate the target value more efficiently.

If you need to replicate these functionalities in Python or if there are specific aspects of this Fortran code that you'd like to discuss or translate, I'm here to help. Just let me know what you need!

You

```
Makefile : " FC = ifort # Define the general flags. FCFLAGS = -O0 -qopenmp -g # Enable OpenMP
LDLFLAGS = -L. -lsearchutils MODSRCS = searchutils.f90 MAINSRC = main_program.f90 all:
searchutils.mod main_program.exe searchutils.mod: searchutils.f90 $(FC) -c searchutils.f90 $(FCFLAGS)
ar rc libsearchutils.a searchutils.o ranlib libsearchutils.a main_program.exe: main_program.f90 $(FC) $^
-o $@ $(FCFLAGS) $(LDLFLAGS) .PHONY: clean clean: rm -f *.exe *.o *.mod "
```



Python

This Makefile is set up to compile a Fortran project that includes two main components: the **searchutils** module and the **main_program**. Here's a brief breakdown of how this Makefile works:

1. **Compiler Definition:**
 - **FC = ifort:** The Intel Fortran Compiler is used to compile the programs.
2. **Compiler Flags:**
 - **FCFLAGS = -O0 -qopenmp -g:** These flags set the optimization level to none (**-O0**), enable OpenMP for parallel processing (**-qopenmp**), and include debugging information (**-g**).
 - **LDLFLAGS = -L. -lsearchutils:** These linker flags specify the directory for the linker to search for libraries (in this case, the current directory) and link against the **searchutils** library.
3. **Source Files:**
 - **MODSRCS = searchutils.f90:** This variable is assigned the source file for the module.
 - **MAINSRC = main_program.f90:** This variable is assigned the source file for the main program.
4. **Targets:**
 - **all:** The default target that builds both the module library (**searchutils.mod**) and the executable (**main_program.exe**).
 - **searchutils.mod:** Compiles the **searchutils.f90** file into an object file, then creates a static library **libsearchutils.a** from the object file and indexes it using **ranlib**.
 - **main_program.exe:** Compiles the **main_program.f90** using the **searchutils** library to produce an executable.
5. **Cleaning Up:**
 - **clean:** This target removes all executables, object files, and module files from the directory, cleaning up the build environment.

This Makefile allows for modular compilation of Fortran programs, managing dependencies between the main program and its modules efficiently through the use of libraries. If you have any questions about specific sections or need help adjusting it for a different setup, feel free to ask!

You

2. OpenMP Parallelization of the linear search algorithm (Assignment 10). The work from this part of the assignment should be put in your repository within a folder named "assignment10". Copy the contents in the assignment9 folder (once completed) into the folder assignment10. The goal of this part is to use OpenMP to parallelize the implementation of the linearssearch doloop. • Modify the Makefile to support the OpenMP compiler directives. Add the flag -qopenmp to the FCFLAGS variable in the Makefile. • Modify your searchutils.f90 file, in particular the do loop in the linear search algorithm to parallelize the do loop with OpenMP compiler directive. Make sure that there are no "return" statements within the parallel sections. Also, consider the definition of the variables (shared or private) that are required for the correct behavior of the algorithm. • Modify the main_program.f90 file to use the omp_lib module. This allows to use the runtime subroutines and functions from OpenMP. A particular function of interest is the omp_get_wtime() to evaluate the time for execution. Use the omp_get_wtime() function to measure the CPU time of the linearssearch algorithm $t = \text{omp_get_wtime}()$. Print the CPU time spent evaluating the function to console. • Login to a compute node (request 1 node, 1 task and 80 cpus per task). Load the intel/oneapi module and compile the program and the library. • Set the environment variable OMP_NUM_THREADS to evaluate the parallel performance of the algorithm • Execute the algorithm with OMP_NUM_THREADS set to 1, 2, 4, 8, and 16. • Write a short report with the speedup and efficiency plot with the different execution times of the algorithm. How does this parallel execution time compare to the execution time of the binarysearch function? • Add the report to the repository, and update and commit the assignment.



Python

To achieve the objectives laid out for assignment 10, you'll need to make several modifications to both the Fortran source code and the Makefile to incorporate OpenMP for parallel processing. Here's a step-by-step breakdown of how to accomplish these tasks:

1. Modify the Makefile

You need to add the **-qopenmp** flag to **FCFLAGS** in the Makefile to enable OpenMP directives during the compilation.

makefileCopy code

2. Modify searchutils.f90

The **linearSearch** function's loop should be parallelized using OpenMP. Here's how you might modify the **DO** loop:

fortranCopy code

In this modification:

- **shared(arr, n, x)** ensures that all threads can access the array, size, and target value.
- **private(i)** ensures each thread has its own copy of the loop variable.
- **reduction(min:idx)** allows the threads to find the minimum index where the condition is true, ensuring no **return** statement within the parallel section which is crucial for correct behavior.

3. Modify `main_program.f90`

You should include the `omp_lib` module and use the `omp_get_wtime()` function to measure execution time of the `linearSearch`.

fortranCopy code

4. Compile and Execute on a Compute Node

After making these changes, you should compile the program on a compute node configured as required:

bashCopy code

export

5. Evaluate Performance

To evaluate the performance, you'll collect the execution times for different thread counts and compare these to the single-threaded execution time and the time taken by the binary search function.

6. Report Writing

The report should include:

- Execution times for different values of **OMP_NUM_THREADS**.
- Speedup and efficiency calculations.
- Comparative analysis with binary search execution times.
- Plots visualizing speedup and efficiency.

Include the report in your repository under the "assignment10" folder.

7. Repository Management

Make sure to commit all changes, including new and modified files, to your repository after verifying that everything compiles and runs as expected.

This structured approach will help you effectively parallelize the linear search using OpenMP, measure its performance impact, and analyze the benefits and limitations of parallelization in this context. If you need further clarification or help with coding, feel free to ask!

You

make ifort -c searchutils.f90 -O0 -qopenmp -g searchutils.f90(24): error #7566: A RETURN or EXIT statement is not legal in a DO loop associated with a parallel directive. EXIT -----^ compilation aborted for searchutils.f90 (code 1) make: *** [searchutils.mod] Error 1



Python

The error you're encountering is due to the use of an **EXIT** statement within an OpenMP parallelized **DO** loop, which is not allowed. In the case of the **linearSearch** function, the challenge is to ensure that once a thread finds the target value, it can safely terminate the search without using **EXIT**.

One solution is to refactor the loop to use a control variable that all threads can check, rather than trying to exit the loop directly. Here's an approach using an atomic write to ensure that once one thread updates the **found** flag, all other threads will stop checking further:

fortranCopy code

In this modification:

- A logical variable **found** is introduced. It's initially set to **.FALSE.** and checked within each iteration.
- If a thread finds the value, it enters a critical section where it sets **found** to **.TRUE.** and assigns the index to **idx**. This critical section ensures that the update is safely done without race conditions.
- All other threads will stop processing their respective parts of the array once **found** is set to **.TRUE.**.

Please update your code accordingly and recompile. This should resolve the error and allow you to successfully parallelize the linear search using OpenMP.

You

```
$. /main_program.exe forrtl: severe (174): SIGSEGV, segmentation fault occurred Image PC Routine Line  
Source main_program.exe 000000000040666A Unknown Unknown Unknown libpthread-2.17.s  
00007FEEBEF49630 Unknown Unknown Unknown main_program.exe 000000000040424B Unknown  
Unknown Unknown main_program.exe 0000000000404222 Unknown Unknown Unknown libc-2.17.so  
00007FEEBEB8E555 __libc_start_main Unknown Unknown main_program.exe 0000000000404129  
Unknown Unknown Unknown
```



Python

A segmentation fault in a Fortran program using OpenMP like this usually indicates a problem with memory access. This could be caused by various issues such as incorrect array indexing, accessing uninitialized memory, or race conditions in parallel regions.

Here are a few steps and considerations to help diagnose and fix the issue:

1. Check Array Indices

Ensure all array accesses are within bounds. In a parallel context, especially with large data sets, it's crucial to make sure no thread is reading or writing outside the array limits.

2. Initialization

Check that all variables are properly initialized before use, particularly those that are used across thread boundaries.

3. Use of Shared Variables

Revisit any shared variables. In the modified **linearSearch** function, ensure that the use of the **found** and **idx** variables is thread-safe. The **found** variable is particularly critical since its state influences the execution path of all threads.

4. Debugging with Simplified Input

Try running the program with a simplified scenario:

- Smaller data size
- Fewer threads This might help in isolating the problem.

5. Compile with Debugging Information and Use Debugger

Recompile your code with full debugging information (**-g**) and without optimization (**-O0**). Then use a debugger (like **gdb**) to run your program and see exactly where it crashes.

bashCopy code

When it crashes, type **backtrace** in **gdb** to see the stack trace. This might give you more insights into where and why the crash occurs.

6. Review OpenMP Directives

Ensure that your OpenMP directives are correctly structured and that all variables are shared or private as needed. Here's a quick review:

- Make sure **idx** isn't written to by more than one thread without proper synchronization. Although your **critical** section should handle this, it's good to double-check.
- If your array is large, also ensure that there isn't a hardware limitation or memory limitation being hit by the large size of **arr**.

Here is an updated approach focusing on the OpenMP usage for the search loop:

fortranCopy code

This version uses the OpenMP **cancel** directive to terminate the loop once the target is found, which can be more efficient and avoids additional synchronization overhead.

7. Review Hardware and OS Limits

Check if there are any operating system or hardware-specific limits you might be hitting, such as stack size or memory limits, especially if you are working with very large datasets.

After applying these checks and modifications, try running the program again. If the segmentation fault persists, consider isolating which part of the code leads to the fault and examine related memory operations more closely.

You

```
./main_program.exe forrtl: severe (174): SIGSEGV, segmentation fault occurred Image PC Routine Line
Source main_program.exe 000000000040687A Unknown Unknown Unknown libpthread-2.17.s
00007FE53DA06630 Unknown Unknown Unknown main_program.exe 000000000040430B Unknown
Unknown Unknown main_program.exe 00000000004042E2 Unknown Unknown Unknown libc-2.17.so
00007FE53D64B555 __libc_start_main Unknown Unknown main_program.exe 00000000004041E9
Unknown Unknown Unknown
```



Python

Given that the segmentation fault persists even after careful adjustments, it suggests that there might be deeper issues in how the program handles memory or specific interactions with the OpenMP environment. Let's step through additional troubleshooting strategies to identify and resolve the issue:

1. Isolate the Problematic Section

Temporarily remove or simplify parts of your program to pinpoint the exact location where the segmentation fault occurs. Start with the parallel sections, because they often introduce complexities that could lead to segmentation faults.

2. Check the Handling of Arrays and Pointers

Review any pointers or dynamic memory allocations in your program:

- Ensure that all pointers and arrays are initialized before use.
- Check that no illegal memory dereferencing occurs, especially within parallel regions.
- For large arrays, ensure they are not causing stack overflow—consider allocating large arrays dynamically on the heap if not already done.

3. Reduce Complexity in Parallel Regions

Try simplifying the OpenMP directives. For instance, test the **linearSearch** function without any OpenMP directives to ensure it functions correctly in a serial context. Once it works serially, reintroduce OpenMP directives incrementally:

- Start by just parallelizing the loop without any reduction or cancellation.
- Gradually add directives like **reduction** and **cancel**.

4. Review OpenMP Directives for Correctness

Here's another approach to parallelizing the **linearSearch** that explicitly manages termination:

fortranCopy code

5. Use Compiler Diagnostics and Debugging Tools

- **Compile with -check all:** This compiler flag enables all runtime checks, which might help catch out-of-bounds access or other common errors.

- **Use a Memory Debugger:** Tools like Valgrind can help detect memory leaks, read/write errors, and use of uninitialized memory.

6. Run with Debugger

Recompile with **-g** and run your program within a debugger like **gdb** or use **ifort**'s debugging facilities. Look at the output when you run:

bashCopy code

Then within gdb, type:

bashCopy code

When it crashes, use:

bashCopy code

This will give you a stack trace which could help in identifying the specific line or function causing the issue.

7. Check for External Influences

- **Environment Settings:** Make sure that the runtime environment is set correctly. For example, check if **OMP_NUM_THREADS** and other environment variables are set appropriately.
- **Hardware or OS-related Issues:** Ensure that there are no system-specific issues impacting the execution, such as restrictions on stack size (**ulimit -s unlimited** might help).

If these steps don't resolve the issue, consider sharing more details about the program's environment, or even the complete code, if possible, for further analysis. This might include examining how arrays are passed to functions and ensuring that all data dependencies in OpenMP are handled correctly.