Automatic Stack Management in Large Scientific Applications

Ramesh Naidu Laveti

Centre for Development of Advanced Computing (C-DAC)

Bangalore, India

Contact: rameshl@cdac.in

Outline

1. Introduction
2. Design of Global Spectral Model
3. Application Program Stack
4. Automatic Stack Management Framework
5. Discussions and Results
6. Conclusions

1. Introduction

Large scientific applications demand compilers to allocate large temporaries on the stack.

What if your computing infrastructure can't provide enough stack space??

- **Program may become error prone**
- **May overwrite other memory segments (or) areas of other program's address space**
- **Application may crash**
- **Application may abort**

Example Application - 1

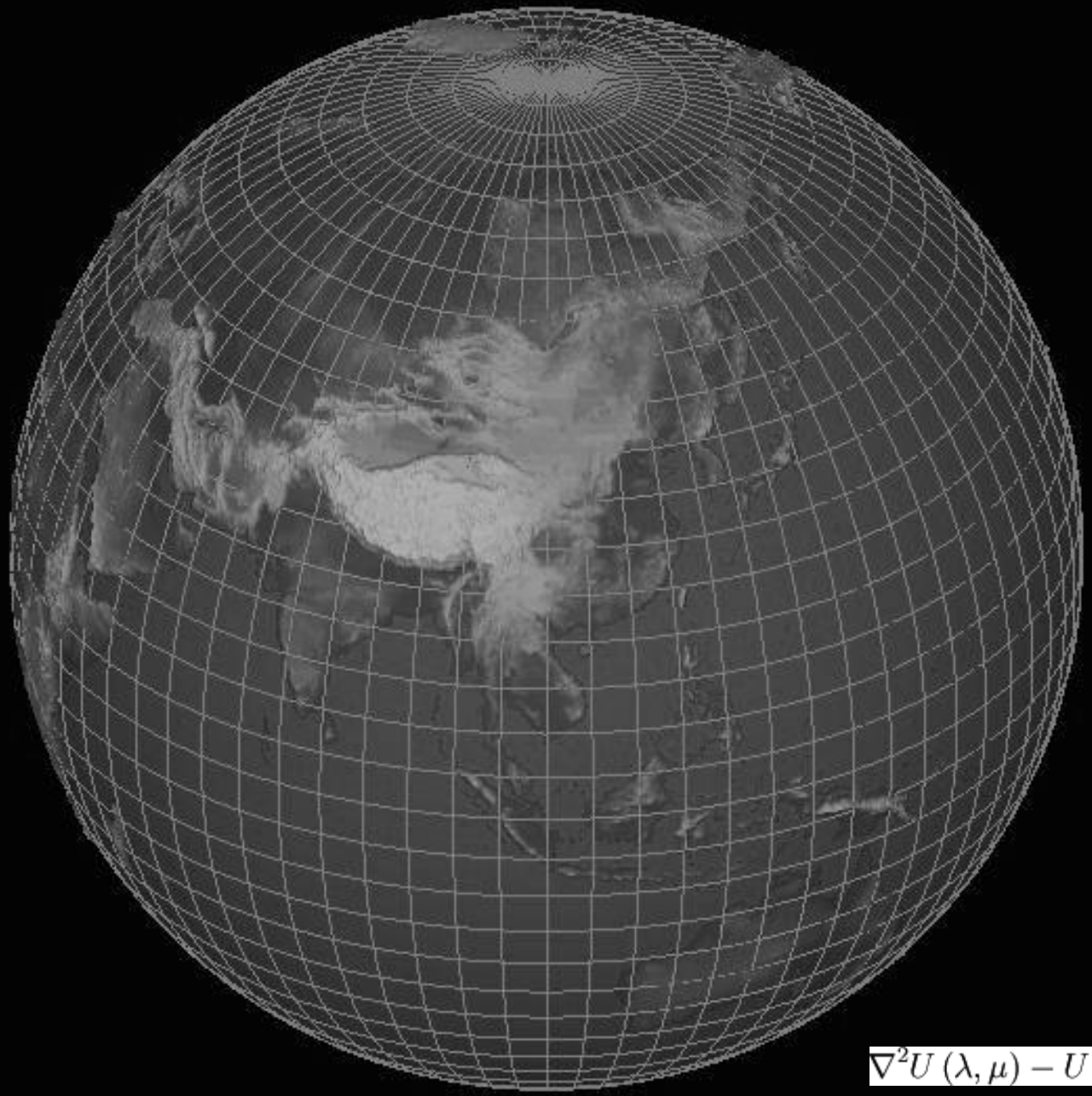
(a) A spectral model kernel using Helmholtz equation

$$\nabla^2 U(\lambda, \mu) - KU(\lambda, \mu) = R(\lambda, \mu)$$

Where, λ - Longitude, in the interval [0 , 2Π],
μ - sin(Latitude), in the interval [- 1 , 1], and

$$\nabla^2 U = \frac{\partial}{\partial U}\{(1-\mu^2)\frac{\partial U}{\partial \mu}\} + \frac{1}{(1-\mu^2)}\frac{\partial^2 U}{\partial \lambda^2}$$

- We solve this equation for the unknown 'U' from the known function 'R'. The discrete values of the function 'U' will be computed analytically at each (λ , μ)
- Need to be solved with high accuracy and speed
- Many ways to solve a PDE – Spectral Method is one option

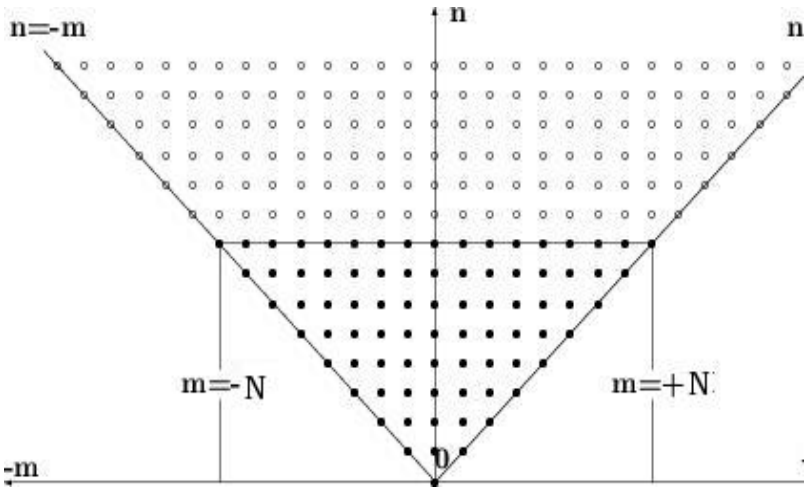


$$\nabla^2 U(\lambda,\mu) - U(\lambda,\mu) = R(\lambda,\mu)$$

Spectral Method

- **Spectral method** is a numerical method for solving partial differential equations in which the dependent variables are expanded as a series of basis functions and the original equations are reduced to a set of algebraic equations.

$$R(\lambda,\mu) = \sum_{m=-\infty}^{\infty} \sum_{n=|m|}^{\infty} R_n^m * Y_n^m(\lambda,\mu)$$



- Triangular truncation - Infinite series to a finite series
- Derivatives can be computed exactly, Easier to solve, Fast and accurate
- **Difficulties in handling very high resolution (T>1000) → Compute and data intensive → Hard to implement and simulate……..**

Solving Helmholtz Equation

To solve Helmholtz equation using spectral method, Let

$$R(\lambda,\mu)=\sum_{m=-N}^{N}\sum_{n=|m|}^{N}R_n^m Y_n^m(\lambda,\mu) \tag{1}$$

$$U(\lambda,\mu)=\sum_{m=-N}^{N}\sum_{n=|m|}^{N}U_n^m Y_n^m(\lambda,\mu) \tag{2}$$

Where,

$$R_n^m=\int_0^{2\Pi}\int_{-1}^{1}R(\lambda,\mu)\,\bar{Y}_n^m(\lambda,\mu)\,d\lambda d\mu$$

$$U_n^m=\int_0^{2\Pi}\int_{-1}^{1}U(\lambda,\mu)\,\bar{Y}_n^m(\lambda,\mu)\,d\lambda d\mu.$$

After substituting (1) and (2) into Helmholtz equation, we obtain

$$-n(n+1)\left[\sum_{m=-N}^{N}\sum_{n=|m|}^{N}U_n^m Y_n^m\right]-\left[\sum_{m=-N}^{N}\sum_{n=|m|}^{N}U_n^m Y_n^m\right]=\left[\sum_{m=-N}^{N}\sum_{n=|m|}^{N}R_n^m Y_n^m\right]$$

Comparing the coefficients of Y_n^m on both sides we will get

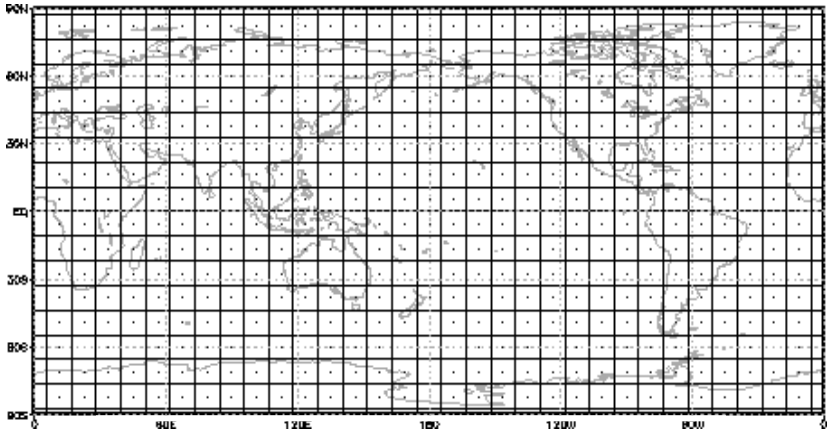
$$-\left[n(n+1)+1\right]U_n^m=R_n^m$$

Thus

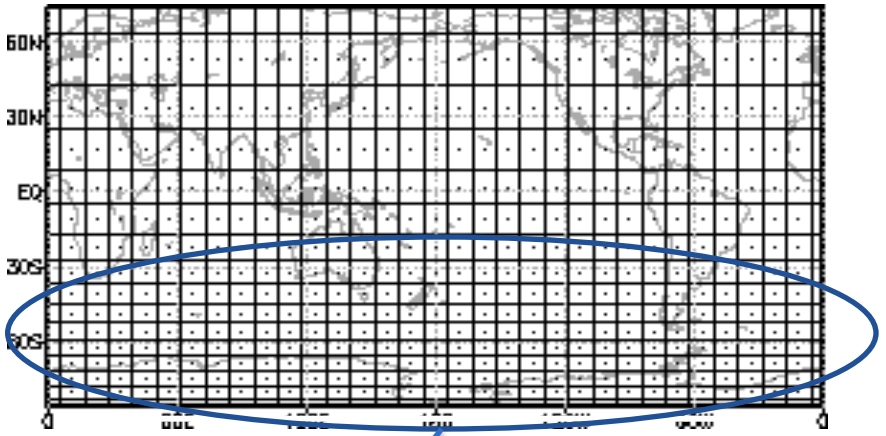
$$U_n^m=\frac{R_n^m}{-\left[n(n+1)+1\right]} \tag{3}$$

Example Application - 2

2. Change resolution component of a global spectral model - SFM



Uniform Resolution



Variable resolution

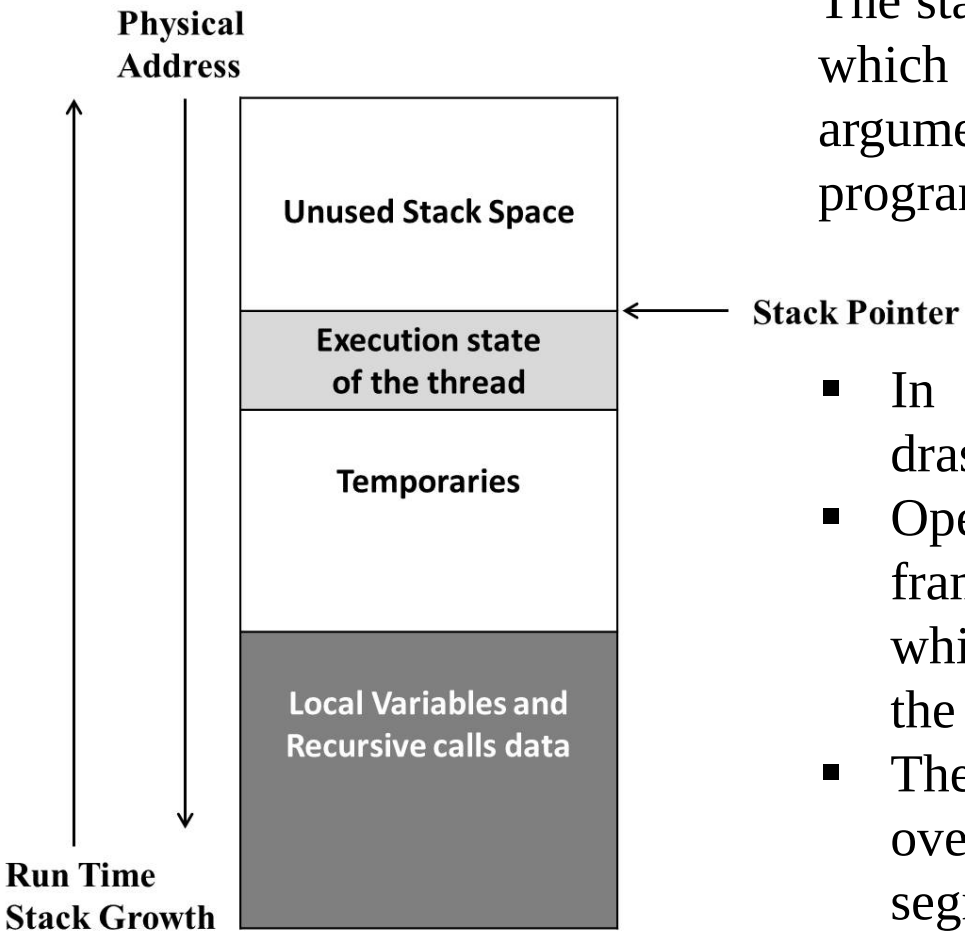
High Resolution → Large temporaries on stack

2. Design details of spectral model kernels

- **Parallelization strategy used**
  - It uses **2-D decomposition** method
  - So, flexible to run any number of processors (Except a prime number)
  - Gives best performance if we choose the number of processors as $2^p \times 3^q \times 5^r$

    where p, q and r are integers
- **Portability**
  - Can run as sequential or shared memory or distributed memory
  - It can run on multiple platforms – Little Endian as well as Big Endian
  - Good applications to experiment
- **Hybrid parallel programming paradigm**
  - Used – OpenMP + MPI (C, C++ and FOPRTARAN)

3. Application program stack



The stack—a region or a segment of memory in which local variables are located and function arguments are passed—is allocated by the programmer.

- In OpenMP, each thread may have drastically different stack size requirements.
- OpenMP runtime library generates stack frames that are not part of the user code, which could confuse users when finding out the reasons for stack overflow.
- The simple technique - prevent stack overflows is **manually inspect the stack** segment and the stack pointers to find out the possibilities of stack overflow.

3. Application program stack…

Manual Stack Inspection

Thread ID	Stack Start	Stack End	Stack Pointer	Stack Size (Bytes)	Stack Usage (Bytes)	Category
0	0x1000 2000	0x1000 2255	0x1000 2121	1024	484	Normal
1	0x1000 3000	0x1000 3255	0x1000 3244	1024	976	Critical
2	0x1000 4000	0x1000 4255	0x1000 4148	1024	592	Normal
3	0x1000 5000	0x1000 5255	0x1000 5201	1024	804	Critical

Table 1: Sample Summary report of stack information of all the threads of an application

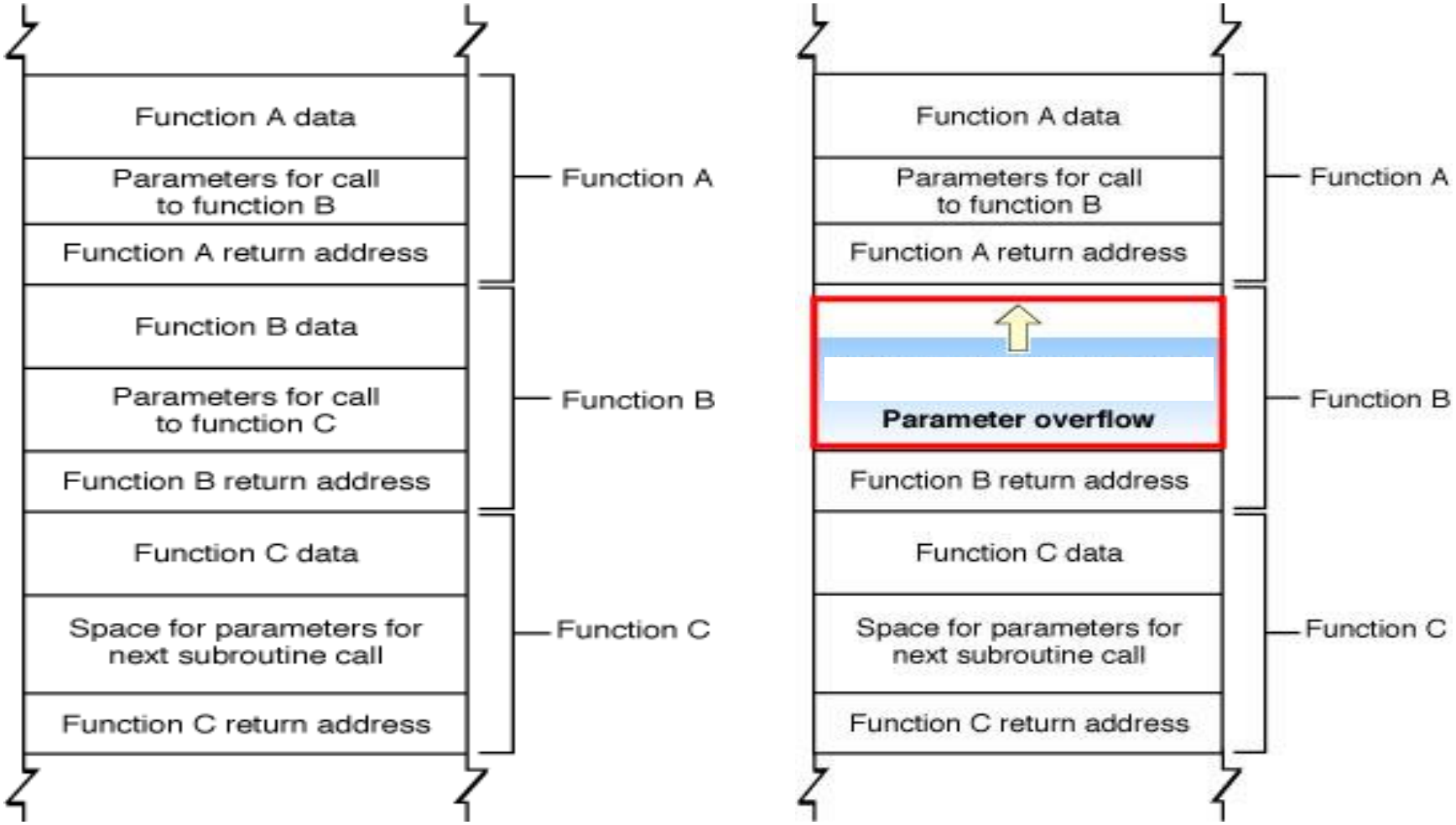
3. Application program stack

Manual Stack Inspection

- Obtain the summary report of the stack usage patterns of all the threads of global spectral model
- Inspect the stack space used and free space left for a thread
- Classify each thread stack state into two categories:
    - Noraml (<80% stack space usage)
    - Critical (>80% stack space used).
- Investigate all the threads during the entire period of the application execution and observe the patterns.
- This helps us to identify the threads which use more than 80% of the stack space allocated to it and address the stack overflow problem manually.

3. Application program stack

Manual Stack Inspection

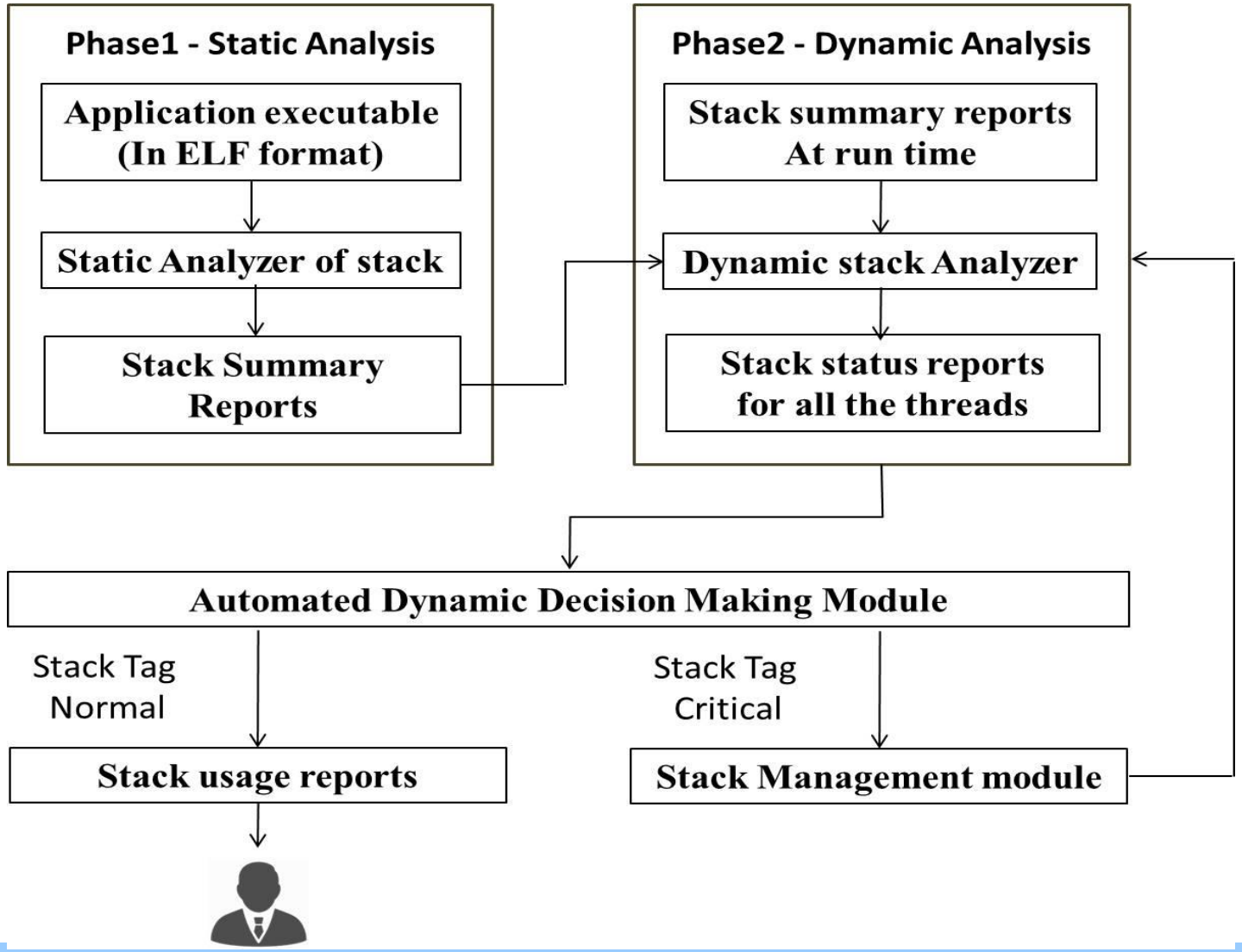


3. Application program stack

Manual Stack Inspection - Shortcomings

- **Stack overflow detection is still made by the developer** using the information produced by gdb.
- If the developer doesn’t identify the overflow, the problem may go unidentified.
- **No provision to abort the application immediately when the overflow occurs.**
- Estimation of number of control flow cycles may not be possible at compile time.
- Finding unexpected/unidentifiable function calls and control jumps may effect the worst case stack usage patterns of a program.
- Hardware interrupts or signals may also use stack and it is difficult to get to know whether they are using dedicated stack space or not.

4. Automatic dynamic stack management framework



4. Automatic dynamic stack management framework

Phase 1: Static Analysis

- Analyzes and examines the program's executable file (ELF file).
- Built around the features provided by “gcc” and “gfort” compilers.
  - fstack-usage
  - fstack-check
  - fcallgraph-info
- **This method allows us to compute to analyze**
  - The application **stack space consumption patterns**
  - Possible **worst case stack usage** prior to execution.
- Provides precise information about **the possible maximum stack usage** from each thread.

4. Automatic dynamic stack management framework

Phase 2: Dynamic Analysis and Decision Making

- Trace the number of read and write operations to the stack
- Record the memory reference information
- Maximum value of the stack pointer (Heuristics)
- Use backtrace to collect the information about currently active function calls
    - The information obtained from the selected stack frames
    - Information starting with the currently executed frame, its caller and other frames up in the stack.
    - Stack frame info such as: Address of the frame, address of the next frame down and up, the programming language used, address of the frame's arguments, address of the frames local variables, the program counter, etc.
- Stack status reports and decision making

5. Experiments and Results

Model Resolution	% of Stack Space Used	Stack Tag	Overhead due to heap memory	Overhead due to Dynamic stack framework
300 km x 300 km	40.00%	Normal	-NA-	-NA-
150 km x 150 km	58.50%	Normal	-NA-	-NA-
50 km x 50 km	72.50%	Normal	-NA-	-NA-
10 km x 10 km	85.00%	Critical	25.00%	4.00%
1 km x 1km	98.00%	Critical	25.00%	4.00%

Table 2: Stack memory access patterns and the comparison of overheads incurred by Heap arrays and dynamic stack management framework

5. Experiments and Results

Resolution

256 Longitudes x

128 Latitudes

Error Range

-0.2 to 1.4

Top panel

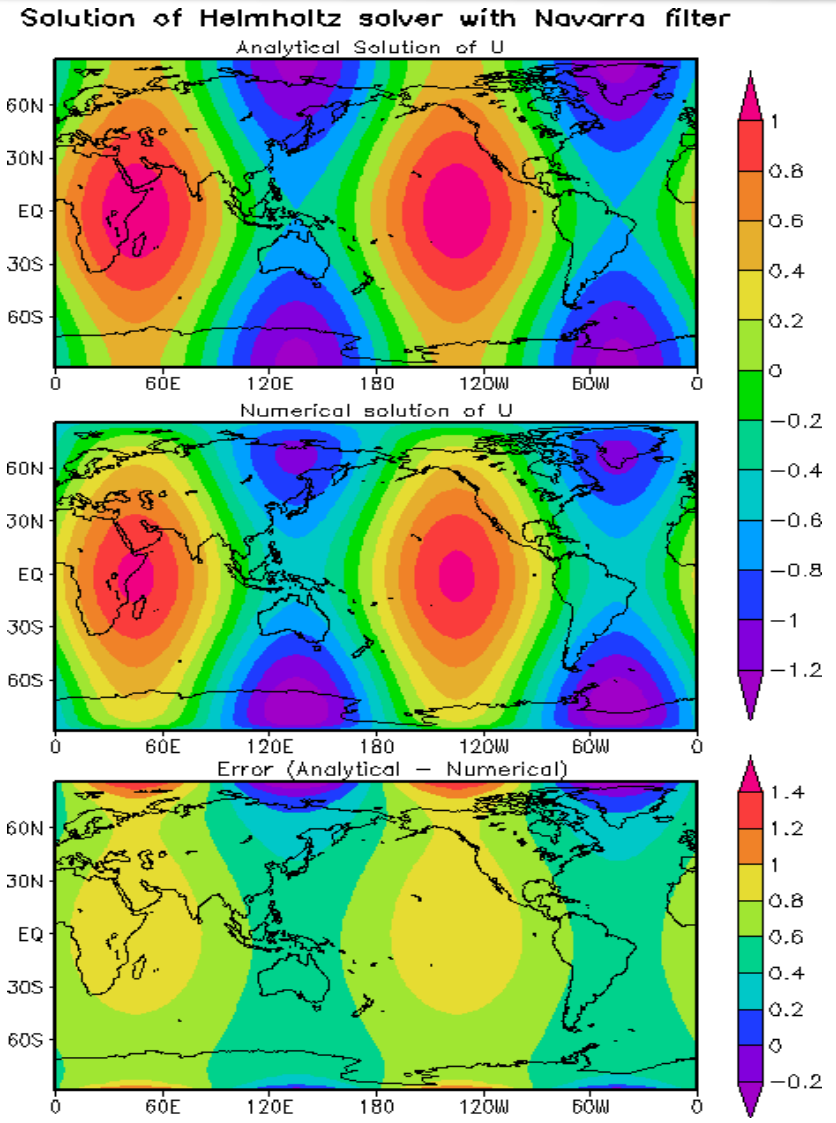
Analytical solution of the Helmholtz Solver

Middle panel

Numerical solution of the Helmholtz Solver

Bottom panel

Error in solution of the Helmholtz Solver



5. Experiments and Results

Resolution

1024 Longitudes x

256 Latitudes

Error Range

-0.2 to 1.4

Top panel

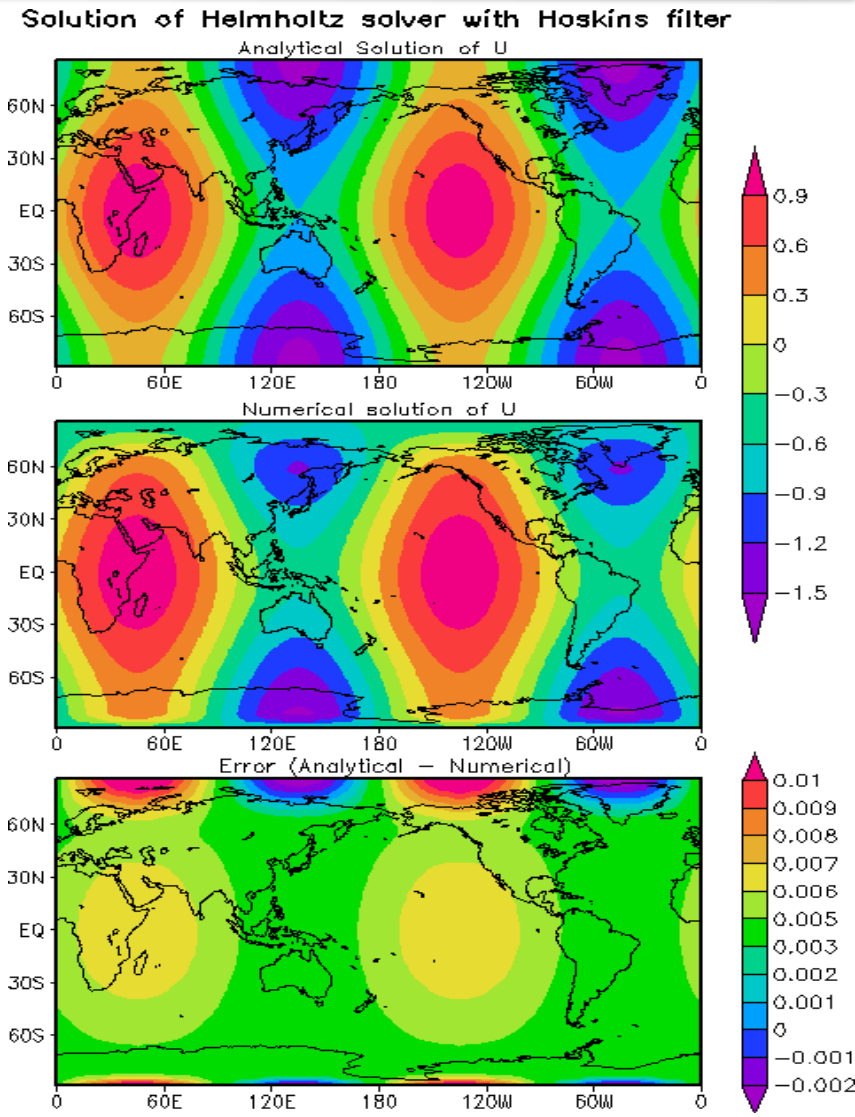
Analytical solution of the Helmholtz Solver

Middle panel

Numerical solution of the Helmholtz Solver

Bottom panel

Error in solution of the Helmholtz Solver



Conclusions

- A sophisticated 2-Phase solution to handle stack overflows
- Do not introduce much run time overheads (<4% of total turnaround time).
- Handles the large temporaries without user's intervention
- Experienced 21% performance improvements when compared with heap arrays
- We also noticed that it introduces around 4% overhead, which can be ignored
- The actual gain depends on the size of the temporaries in an application
- Supports sequential and OpenMP applications
- We further enhance our framework to deal with the complex MPI and GPU programming paradigms

THANK YOU