PROCEEDINGS OF SCIENCE

Automatic dynamic stack management in large scientific applications: A case study using a global spectral model

Ramesh Naidu Laveti[1]
Center for Development of Advanced computing
C-DAC Knowledge Park, Byappanahalli, Bangalore, India
E-mail: rameshl@cdac.in

Compute and data intensive scientific applications demand compilers to allocate more temporaries on the stack. For example, the change resolution component of a global spectral model changes the resolution of the input files using Nearest Neighbour Interpolation which requires large temporaries on the stack. Temporaries include sub-arrays, automatic arrays and, sub-sections corresponding to actual arguments of a subroutine. If the infrastructure cannot provide adequate stack space at runtime relative to the total size of the temporaries, then the application program runs out of stack and aborts. Allocating the heap memory to store the large temporaries introduced around 25% additional runtime because of allocation and deallocation of the memory. This is observed in various components of a global spectral model.

We propose an automatic dynamic stack management framework which uses application profile information and the information related to the required stack memory. It does not mandate any hardware configuration changes. This technique manages stack frames on RAM by the compiler-inserted code into the application binary. Our experiments with a global spectral model show that we are able to obtain an average runtime savings of 21% along with a compile time overhead of 4%. The actual gain depends on the size of the temporaries in an application and the size of the RAM. Currently, it supports sequential and OpenMP applications. We further enhance our framework to deal with the complex MPI and GPU programming paradigms.

International Symposium on Grids and Clouds 2016
13-18 March 2016
Academia Sinica, Taipei, Taiwan

[1]Speaker

PoS(ISGC 2016)027

1. Introduction

When an executable is loaded into memory, it is divided into mainly three segments: code segment, stack segment and heap segment. In code segment, the compiled code itself will reside. In any computer program, by default, local variables, automatic variables, non-initialized and non-saved variables, non-allocated variables of a routine are stored in stack segment. Also, whenever it is required the compiler will make a temporary copy of an array on the stack. This is widely and conveniently used in many applications developed using FORTRAN, C and C++. Most of the scientific applications are compute and data intensive, and demand compilers to allocate large temporaries on the stack. For example, the change resolution component of a global spectral model changes the resolution of the input files using Nearest Neighbor Interpolation [1] requires large temporaries to perform its computations.

If the infrastructure cannot provide adequate stack space at run-time relative to the total size of the temporaries, then it may cause several issues such as lack of access control to temporaries, segmentation fault, allocation error, stack pointer error and corrupting other segments such as the heap. The aforementioned issues make the program either error prone or it can make the applications crash and abort. In general, large scientific applications such as climate models are time-critical and sensitive towards the accuracy. A crash or corruption of data leads to conduct the same experiments again after resolving the issues manually. Manual intervention and resolving the stack overflow or data corruption issues by researchers wastes several man hours. Some straightforward solutions were present to address this issue such as directing the compiler to keep all the temporary arrays on to dynamically allocated heap memory segment, or a user may try to adjust the process stack size for his particular application. To store the data on the heap, we have to use dynamic memory allocation (DMA) methods. DMA and fetching the data from the heap is slow when compared to stack. Therefore, it is often desirable to handle the temporaries stored on stack memory dynamically without human intervention at run time is a better solution.

The appearance of high-performance clusters with many-core processors allows large scientific applications to run efficiently. To exploit the parallelism provided by HPC clusters, we need to use shared memory programming paradigms or distribute programming paradigms, such as OpenMP [2] or MPI [3] or a hybrid paradigm which combines OpenMP and MPI. The applications developed using OpenMP or MPI involve many threads or processes. Hence thread safety of the applications needs to be addressed. Temporaries cause no problems with traditional MPI implementations since each process image contains a separate instance of the variable. However, in the case of OpenMP, we need to maintain thread safety using a specific privatization directive for the key variables. It leads to the creation of several numbers of copies of private temporary variables which results in the usage of huge stack space. When the amount of stack used by an execution thread exceeds an anticipated size, unexpected events happen. We call this as “stack overflow” condition. Stack overflow is a major problem for many applications and the existing standalone compile-time or run-time solutions are not always available or appropriate for all the parallel applications. In some situations, we can prevent overflows from occurring in the first place, which requires guesstimating the worst case stack space requirements of an application prior to execution. This information can be incorporated into the techniques used to address the stack overflow problems at run-time.

In this paper, we present a framework that automatically handles large temporaries via compile and run-time techniques. Handling the problem in an automatic fashion relieves programmers and researchers from the onerous and error-prone process of manually changing their application code. Also, it allows the use of the same source code on different platforms where distinct limits of stack sizes are available. In addition to these benefits, our techniques can be uniformly applied to various programming paradigms that target multi-threaded execution. We also present the advantages and disadvantages of our automatic stack handling framework, and discuss the performance gain. We demonstrate the usefulness of these techniques while running large scientific applications in a multi-threaded environment. In addition, we show that this framework allows utilizing the stack space uniformly by all the threads across processors to achieve better load balance.

The rest of this paper is organized as follows. Section 2 describes the parallel design and the implementation details of the Global Spectral Models and their kernels (GSM). Section 3 describes in detail the problem posed by large temporaries and the importance of properly handling it. In section 4, we present the techniques and framework we used to address the stack memory problem. Section 5 contains our experimental performance results. Finally, Section 6 has conclusions and the future directions of our work.

2. Design and Implementation details of Global Spectral Model kernels (GSM)

The spectral method is a widely used numerical technique in which the prognostic field variables are represented as a sum of a finite set of spectral modes rather than at grid points. The spectral modes may be Fourier modes in case of 1-Dimensional fields or double Fourier modes or spherical harmonics in the case of 2-Dimensional fields. Spectral methods are high order methods which allow for either obtaining very accurate results or reducing the number of degrees of freedom for a fixed standard accuracy. Climate modeling and weather forecasting have long been application areas of spectral methods.

In a global spectral model, spectral transformation uses a combination of a Fourier transform and a Legendre transform. The functions used in most global atmospheric spectral models as the basis functions are spherical harmonics, a combination of sine and cosine functions that represent the zonal structure and associated Legendre functions that represent the meridional structure [4].

Numerical weather prediction or climate simulations must often meet stringent computational efficiency as well as accuracy requirements. For many applications, global spectral transform models can satisfy this pair of requirements [5]. However, these models require a significant amount of memory and computational resources to generate forecast at a high resolution. It is quite important to understand the stack usage patterns of these models to conduct the forecast experiments. For dynamic stack management experiments, we choose two climate model kernels: Helmholtz equation based uniform resolution spectral model kernel and the change resolution component of an atmospheric general circulation model named Seasonal Forecast Model (SFM) [6]. We discuss these models in the next subsection.

2.1 A spectral model kernel using Helmholtz equation

A global spectral model contains many dynamical processes. Each dynamical process can be represented by a set of partial differential equations such as Helmholtz equation, which should be solved accurately and efficiently. We have developed a spectral method based solver for Helmholtz equation on a unit sphere with uniform forecast resolution on the globe. Here, the uniform resolution grid means the number of latitudes taken on the grid is half of the number of longitudes. The Helmholtz equation is second order non-linear partial differential equation. In a two-dimensional case, it can be written as,

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

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

We solve this equation for the unknown variable 'U' from the known variable 'R'. The discrete values of the function 'R' will be computed analytically at each point (λ, μ) and from these values we can compute the spectral coefficients 'Rmn'. Using spectral method, we can obtain the spectral coefficients 'Umn' of the unknown parameter 'U'. The 'Umn' values will be used to compute the discrete values of the unknown 'U'. In solving Helmholtz equation, we use forward Fourier transform applied in the zonal (east-west) direction and the Gaussian quadrature are evaluated in the meridional (north-south) direction. The forward Fourier transform is computed at each circle of latitude using a discrete fast Fourier transform (FFT). It involves a lot of computations as well as large temporaries.

2.2 Change resolution kernel of global spectral model - SFM

Seasonal Forecast Model (SFM) was developed by Experimental Climate Prediction Center (ECPC), USA. SFM is an efficient, stable, state-of-the-art atmospheric general circulation model designed for seasonal prediction and climate research. Change resolution component is one of the important modules of SFM. This component is used to change the resolution of the input files such as sigma and surface restart files from the global spectral model. The input files should have header records identifying their respective resolutions. Nearest neighbor interpolation is performed to transform the climate model's input conditions from high resolution to low resolution or vice versa. Resolution is represented using spectral truncation [4], the number of pressure levels, the number of longitudes and the number of latitudes. This component requires the huge number of computations and large temporaries if we want to convert the input data from very low resolution to very high resolution.

2.3 Design and implementation details of spectral models

The computational complexity of the spectral transform method is dominated by Legendre Transformation, which requires the time complexity of $O(n^2)$. To increase the speed, we have designed a parallel decomposition algorithm of spectral transformation. Initially, 1-D decomposition method has been introduced, and performance metrics were captured. In this method, decomposition is done in the Longitudinal direction. In 1-D decomposition, FFT requires communication among all the processes or threads and the Legendre transformation can be done without any communication among the processes or threads. However, it did not yield

expected performance improvements. To increase the performance, we have introduced 2-D decomposition method which does the decomposition of the data in the both Longitudinal and Latitudinal directions. Fast Fourier Transformation requires all the arrays in the X-direction to reside in one processor, but arrays in Y and Z directions can be separated into different processors. Similarly, Fourier summation in the Y-direction requires that the entire array in Y-direction must reside in one processor, but arrays in X and Z directions can be in separate processors. The physical process calculations require all the variables at all levels for each grid point, so that arrays in the Z-direction need to be in the same processor, but arrays in the X and Y directions can be separated onto different processors. This array distribution requires the entire arrays be rearranged into different configurations before the computational operations. This transpose method is named as 2-Dimensional decomposition because one of the dimensions is fixed but the other two are distributed. This decomposition gives flexibility in the choice of the number of processors, it allows any number of processors.

The advent of powerful hardware technologies can employ a wide range of programming paradigms such as OpenMP, MPI and OpenCL to harness the many levels of architectural parallelism [7], including models to exploit parallelism in CPUs and GPUs. We have developed Helmholtz equation based global spectral model using hybrid programming model, OpenMP and MPI. The change resolution component of SFM was developed using pure OpenMP.

3. Application program stack

In FORTRAN or C programming language, 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. The amount of memory allocated depends on several factors such as machine architecture, OS, application design, and the amount of total memory available. If the program requires more memory for its stack than the allocated memory then the stack overflows, without warning in most cases. It can corrupt the memory areas of other programs and often results in a crash or even a program malfunction. It is very difficult to trace back the stack overflow, causing application developers to spend a significant amount of time and energy to find the underlying cause of the problem that the application exhibits. As a result, they tend to allocate more stack memory than required as a precaution.

The limitations of the stack memory are generally ignored by the developers as well as the users. This is due to the assumption that stack data is volatile and it can be managed at the run time very easily. However, we need to consider the significance of stack data, since scientific applications such as global spectral models can have the significant amount of memory references to stack data. These climate model kernels store large temporaries on the stack and access the data allocated on stack very frequently. It is very common in these kernels to increase the stack size and put more computation data on the stack, which leads to stack overflow. Therefore, it is important to investigate the stack usage patterns because it helps us to understand how these kernels use the memory objects on the stack and then we can consider a better stack data placement. However, it may not be easy to find out the exact patterns of stack data patterns in a multi-threaded or shared memory programming environment such as OpenMP due to the following reasons.

- Automatic data placement in a shared memory space is non-trivial.
- Dynamically allocated memory such as heap memory is usually allocated via system calls in Linux/UNIX like systems and dynamic stack space adaption is managed by the OS kernel. Therefore, the application program or run-time libraries can not easily distinguish underneath stack and heap management so that all stack and heap operations are automatically performed under control of the OS Kernel.
- The traditional memory model of a sequential program can only trace a single heap and a single stack within a single address space. So, conventional heap and stack management techniques are unsuitable for shared or distributed parallel computations.

3.1 Stack inspection

In OpenMP, each thread may have drastically different stack size requirements. A thread may cause the stack overflow if it tries to keep more data than the size of the stack. In the case of OpenMP, the 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 to prevent the stack overflows is manual inspection of the stack segment and the stack pointers to find out the possibilities of stack overflow. An example stack of a single thread is shown in Figure 1.

We have investigated the application stack memory storage and access patterns of both the climate model kernels as described in Section 2. In our study, We have included relevant parameters such as stack size, stack pointer position, starting address of the stack, end of the stack and the stack usage at a particular instance. The examining of these parameters is done using GNU debugger (gdb). The stack frames of all the function calls in a thread are obtained from the call stack. The call stack contains several stack frames, each frame corresponds to a function call. We can obtain the complete information about a particular stack frame using "gdb" commands.

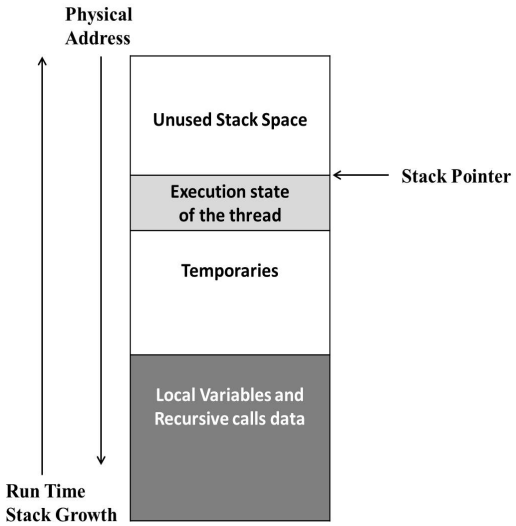

Figure 1. Example thread stack and its sub-segments

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

A wrapper was developed to consolidate the information about all the stack frames of an application at a particular instant and obtain the summary report as shown in Table 1.

We can obtain the summary report of the stack usage patterns of all the threads of our global spectral models and inspect the stack space used and free space left for a thread. After inspecting this information, we classify each thread stack state into two categories: Normal and Critical. If the free space left in stack segment is more than 20% then we categorize it as Normal. Otherwise, it is categorized as Critical. We can 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. This stack checking is a good technique, but it does have some shortcomings. One such shortcoming is that the stack overflow detection is still made by the developer using the information produced by gdb. If the developer does not identify the overflow, the problem may go unidentified. Also, there is no provision to abort/suspend the application immediately when the overflow occurs. This is where automated dynamic run-time stack management comes into play. The proposed framework is described in Section 4.

4. A framework for automatic dynamic stack management

The automatic dynamic stack management framework instruments stack data in two phases. In the first phase, we use a static analysis tool which captures the stack information from the executable. In the second phase, we use dynamic analysis tool which captures the stack usage patterns at the run time.

4.1 Static Analysis

Static analysis tool analyzes and examines the program's executable file which is in Executable and Linkable Format (ELF). This tool was built around the features provided by GNU C compiler (gcc) and GNU FORTRAN compiler (gfort). These compilers provide us several stack information options such as “-fstack-usage”, “fstack-check” and “-fcallgraph-info”. The information obtained from the executable allows us to analyze the application stack space consumption patterns. It is useful to obtain the possible worst case stack usage prior to execution. It also provides the precise information about the possible maximum stack usage by each thread. This method has a couple of advantages such as there is no run-time overhead and

not constrained by the target machine's hardware resources. However, it has several disadvantages in which few are listed below.

- Finding unexpected function calls and jump instructions may affect the worst case stack usage patterns of a program.
- Estimation of the number of control flow iterations may not be possible at compile time. The worst case stack usage depends on the maximum possible control flow iterations, which is unknown at compile time.
- Hardware interrupts and signals may also use the stack. It is difficult to get to know whether they are using dedicated stack space or not.

To overcome the aforementioned problems, we have implemented a dynamic stack analysis framework which will be used at the run time, we call it as the second phase. This will use the information obtained from the first phase, i.e. static analysis tool and incorporates this data into the framework at application's run time. Hence, this can be called as hybrid approach.

4.2 Dynamic Analysis

The proposed framework was mainly built using the following technique. First, we trace the number of accesses, i.e. total number of read and write operations to the entire program stack using the static analysis report. In particular, for each memory reference, we store the position of the current stack pointer for each thread and also capture the memory reference information. We also store the information about the maximum value that the stack pointer has had during the execution of the program. We assume that the stack pointer grows downwards. If the referenced memory address stays between the maximum stack pointer and the current stack pointer then this memory reference is counted as a valid stack memory reference. However, this information itself may not suffice to conclude about a valid stack reference. We need to analyze the stack data at a finer granularity at the run time. To incorporate the finer granularity to investigate a thread's stack, we need to obtain the information about each routine's stack frame from the call stack. To implement this, we store the addresses of all function calls and the return addresses. During the program execution, for each memory reference, the proposed framework go through our call stack. It is always possible that the routine which is executing may access a stack frame underneath the current routine’s frame. In this case, the memory reference is attributed to the underneath stack frame. We use a technique called backtrace to get the details of the. A backtrace is a list of the function calls that are currently active in a thread. The usual way to inspect a backtrace of a program is to use an external debugger such as “gdb”. We obtain the backtrace information programmatically by injecting small code snippets into the program at compile time. This framework mainly relies on the following information.

- The information obtained from the selected stack frames of each thread.
- The information obtained from the backtraces: Information starting with the currently executed frame, its caller and other frames up in the stack.
- Stack frame information such as: Address of the frame, the address of the next frame down and up, the programming language used, the address of the frame's arguments, address of the frame's local variables, the program counter and the register information.
- Application profile information using static analysis.

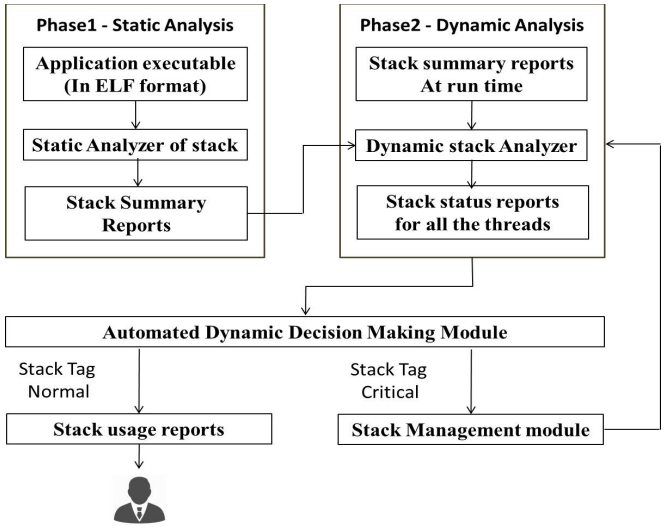

Figure 2. Automated dynamic stack management framework

This two-phase framework allows us to examine the stack of every thread during the execution. It also gives us the facility to analyze the stack usage patterns of failed thread or suspended threads of OpenMP applications. It keeps track of high stack usage for each thread, which helps us to tune the stack size at the run time. To do this, it will keep a watermark which represents the high stack usage by a thread. The address of this watermark to be subtracted from the ending address of the stack to obtain the total amount of stack space used in a particular instance. If an overflow condition occurs, it can identify and suspend all the threads related to it. After that, it will try to move data which caused the overflow to the other threads which have enough stack space to hold this data or it will try to migrate the data to the heap. The discussed framework is depicted in Figure 2.

4.3 Limitations

The known limitations of the proposed framework are:

- It may fail in the case of tail call optimization because tail call optimization replaces one stack frame with another; frame pointer elimination will stop backtrace from interpreting the stack contents correctly.
- This framework may fail for inline functions.
- It works only on the systems which generate the executable in ELF format.
- It can only work with sequential and shared memory programming models (OpenMP).

5. Discussions and Results

This section presents the results of our experiments conducted using two applications: Helmholtz solver based uniform resolution spectral model and global spectral model of SFM. We have conducted the experiments using both the applications with various grid resolutions on the sphere. As we increase the grid resolution, the size of the temporaries will increase proportionally. The grid resolution is directly proportional to the amount of stack space required to execute this application. We first captured the percentage of references to stack data out of

total memory references. For this application, the average percentage of stack data is around 40% if we fix the grid resolution as 300km x 300km. As we increase the grid resolution to 10km x 10km, the stack data percentage increased significantly to 85% and model aborts due to stack overflow. We further increased it to 1km x 1km which also caused stack overflow problem after the completion of the 70% of the model execution. This experiment helped us to understand how the dynamic stack management tool manages the stack overflow.

Initially, we tried using heap arrays to overcome the stack overflow error. It had introduced around 25% more execution time in both the cases. Later, we have conducted the same experiments on the same computational resources using the framework. It could manage the stack overflow condition with an additional overhead of just 4% of the total execution time. We also observed how the stack data size changes in time during the entire model execution. The results are depicted in Table 2.

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%

T*able 2: Stack memory access patterns and the comparison of overheads incurred by heap arrays and dynamic stack management framework*

The results proved that the proposed framework can handle large temporaries with a minimal run time overhead and can tackle stack overflow problem efficiently.

6. Conclusions and Future work

We presented automatic dynamic stack management framework which uses application memory access profile information at the run time and the information related to required stack memory at the compile time. It helps us to handle stack overflow dynamically without introducing much run time overheads. This manages stack frames on RAM by compiler-inserted code into the application binary and handles the large temporaries without user's intervention. Our experiments with a global spectral model show that we are able to obtain an average run-time savings of 21% when compared to heap arrays as a solution to handle large temporaries of global spectral model of SFM. We also noticed that it introduced around 4% of compile time overhead which is very minimal. The actual gain depends on the size of the temporaries in an application and the size of the RAM. Currently, it supports sequential and OpenMP applications. We further enhance our framework to deal with the complex MPI and GPU programming paradigms.

References

[1] Rukundo Olivier and Cao Hanqiang, *Nearest Neighbour Value Interpolation*, (IJACSA) International Journal of Advanced Computer Science and Applications, Vol. 3, No. 4, 2012.

[2] *The OpenMP® API specification for parallel programming:* http://openmp.org/wp/, Accessed on 22nd February, 2016.

[3] The Message Passing Interface (MPI) standard: http://www.mcs.anl.gov/research/projects/mpi/, Accessed on 22-02-2016.

[4] T. N. Krishnamurti, H. S. Bedi, V. M. Hardiker, *An Introduction to Global Spectral Modeling.*

[5] Charles T. Gordon and William F. Stern, *A description of the GFDL global spectral model,* Vol. 110, No. 7, Monthly Weather Review, July 1982.

[6] *The ECPC seasonal prediction system:* http://ecpc.ucsd.edu/projects/G-RSM/docs/INT/ , Accessed on 20th February, 2016.

[7] Jack Dongarra et al. The international exascale software project road map. International Journal of High Performance Computing Applications, 25(1):3–60, February 2011.