Parallel Programming and the IMSL® Libraries

An Overview

A White Paper by Rogue Wave Software.

February 2012

Rogue Wave Software
5500 Flatiron Parkway,
Suite 200
Boulder, CO 80301, USA
www.rougewave.com

Parallel Programming and the IMSL® Libraries

An Overview

by Rogue Wave Software

Printed in the United States of America

Publishing History:

February 2012 - Update

October 2007 – First Publication

TABLE OF CONTENTS

Document Scope and Audience

This white paper provides an overview of the fundamentals of parallel computing and details regarding the use of Rogue Wave Software products in the area of High Performance Computing (HPC). The product focus is the IMSL® Fortran Numerical Library. The intended audience is a software developer with little experience in parallel computing and/or minimal familiarity with the IMSL Family of products.

The Basics

Programmers are on an eternal quest for higher performance in their applications. Whether a program takes one minute, one day or one week to run, if its execution time can be reduced there will be more time available to analyze the results. Additionally, the sooner one model finishes, the sooner another can start, which allows a scientist to more easily vary initial conditions or test additional parameters.

The most obvious way to enhance performance is to utilize faster hardware. Today's common desktop, with a CPU clock speed measured in gigahertz (billions of cycles per second) is more powerful than supercomputers of a decade ago, mainly because they are able to perform more operations per second. The rate at which faster CPUs are available is truly amazing, but it is not practical for one to continuously upgrade processors for incremental increases in performance. Beyond increasing the performance of individual hardware components, another obvious way to increase execution of a program is to use additional hardware. That is, if one could break a problem into halves and run each half on an identical machine, the total execution time would be half. This scenario is where parallel programming enters the picture.

A larger collection of resources means that more operations can be performed at one time or in parallel. If a model takes T seconds to run, and it can be broken into N parts and run on N CPUs, the time to run the model drops to T/N seconds. There is some overhead involved in communication and configuration. Given the speed increase in operations per second, the data being analyzed in a parallel architecture must be stored and accessed by the system (data locality) as needed. At the same time operations of one processor or node must be able to communicate with other nodes performing operations on the same data or problem. Linear speedup is an ideal scenario; however, since data locality and communications are inexplicitly intertwined with parallel processing, it is nonetheless advantageous and cost effective to acquire the necessary hardware and programming skills to successfully utilize 'Parallel Programming'.

Hardware

The two fundamental hardware configurations for parallel programming are:

- A single computer with one or more multi-core CPUs
- Parallel processing using a networked set of discreet computers

In the first case one may have a single computer with multiple identical CPUs. In this scenario each CPU shares the main system memory and is controlled by a single operating system instance. The hardware architecture is referred to as a symmetric multiprocessing architecture or SMP architecture. SMP architecture applies to the CPU cores, treating them as separate processors. Nearly all hardware manufactures from UNIX vendors, such as IBM, HP, and Sun, to PC manufacturers including Intel and AMD offer CPUs capable of being configured in an SMP environment. Details of the "multi-core" CPU internal architectures vary, but the general concept is that a quad-core CPU is literally four CPU cores on a single silicon die. Physically it looks like a traditional CPU, but electronically there are four CPUs in the package. The performance issues and dependencies on the communications bus has been reduced with these multi-core CPUs. They may or may not share low level CPU cache memory depending on the architecture. Common configurations are 2, 4, 6, and 8 CPU PC systems based on one of Intel, or AMD architectures. But there are also 16, 32, 64 and larger CPU configurations available from traditional UNIX vendors and from Intel.

The second hardware configuration to make use of Parallel Programming is a networked set of discrete computers or clusters. Here, each computer is a separate machine with its own CPU and memory connected to the others by a high-performance network connection. This cluster of computers is referred to as a distributed memory parallel processing configuration. The main advantage of such a system for most users is cost because it can be less expensive to assemble a large number of standard desktop computers than to purchase an SMP system. The tradeoff is performance since communication between individual nodes occurs over a network, where as in SMP architectures the processors share memory (locality of data) and processors share a high-speed bus to interconnect the processors (communications). However, one may be able to afford more nodes in a distributed configuration so overall performance cannot be compared so simply.

Finally, it is common to find a cluster of SMP machines. When this is the case, the programming aspects become more complex because the problem needs to be distributed to each node on the network. Each node should then be capable of using multiple CPUs in an SMP context.

Software

The software used in creating a parallel application depends on the hardware architecture. There are advantages to using an SMP system even if applications are not specifically written to take advantage of multiple CPUs as modern operating systems (e.g., Windows or Linux) are inherently multi-tasking systems. That is, the operating system takes care of providing very short time slices to each application to give the appearance to the user that

multiple applications are running simultaneously. On a single CPU system the reality is that the CPU can only respond to a single application at a time. With multiple CPUs available, the operating system spreads the time slices across all the CPUs, enabling true multi-tasking. Therefore if a computationally intensive numerical model is running on a dual CPU platform it may run faster than on a single CPU system because it is not interrupted for other tasks. However for that same model to run even faster on a quad CPU platform it must be programmed to take advantage of multiple CPUs. Similarly if the program were placed on a cluster, it must know there are other nodes available for work and be able to transfer data to them for processing.

The standard programming interface for shared-memory parallel programming is OpenMP[i]. OpenMP can be used from C/C++ and Fortran on a wide variety of platforms including UNIX and Windows. OpenMP is designed to be flexible and easy to use. Making use of OpenMP on a supported platform (including compiler support) is as easy as adding some directives to the source code. In Fortran, the directives are lines such as `!$OMP PARALLEL DO`. This directive would signal an OpenMP-aware compiler that the following block of code (until it reaches an `!$OMP END PARALLEL DO` statement) can be distributed across available CPUs on the SMP system. A compiler that does not support the OpenMP implementation ignores those lines in the source code since a leading exclamation point is recognized as a standard Fortran comment. In C/C++ the compiler directive might look like `#pragma omp parallel`, which is similarly ignored in unsupported environments. To configure the number of threads into which the main process forks, one may use the `omp_set_num_threads()` library function or set the `OMP_NUM_THREADS` environment variable.

Hence, the term ***thread safe*** enters into the realm of SMP programming. Basically, thread safety means that an object or function maintains a valid state while in use by multiple threads. In an SMP system, multiple instances of the same application may be running in parallel so that anything inside or called by the application must be thread safe. If a program is thread-unsafe, it will be SMP-unsafe since all threads share the same address space. Thus code is thread safe if each thread possesses its own copy of critical data such that the parallel function calls do not “step” on each other. Note that one may write a multi-threaded application using the threading model of the chosen language and platform without using OpenMP. Such multi-threaded programs rely on the operating system to distribute threads among available processors.

For distributed systems (i.e., clusters or grids), the standard means of achieving parallelism is through the Message Passing Interface or MPI[ii]. Clusters follow the design style of message passing architectures in which, at the core of the architecture, message passing between processors is accomplished through explicit I/O operations. As such MPI is a specification and standard for performing these inter-processor communications. Unlike OpenMP, which is a set of compiler directives supported by the compiler itself, MPI requires the installation of an implementation of the standard. A freely available example of a portable implementation of MPI is MPICH[iii], currently at version 2. Once MPICH2 is installed, one builds and compiles source code as usual and links in the MPI libraries at link time.

Then using the command `mpirun`, one executes the resulting executable. In this case, settings such as the number of processors to use are passed as command line parameters to `mpirun`. Inside the source code, specific calls to MPI library functions are made to configure the environment. Many details will appear in the user's source code like `MPI_INIT()`, `MPI_COMM_SIZE()`, `MPI_COMM_RANK()`, `MPI_BCAST()`, `MPI_REDUCE()`, `MPI_FINALIZE()`. Clearly, programming using MPI is not as simple as adding compiler directives into existing code. One must have prerequisite knowledge of how the problem can be parallelized, what information needs to be broadcast to and from the nodes, and the like. There are further complications involving things like the "communicator" and timing that are left to the programmer.

IMSL® Fortran Numerical Library

The use of multiple processors on a computer or a cluster via the IMSL Fortran Numerical Library can be applied in a number of ways. First, some IMSL functions are either SMP-enabled or MPI-capable. Additionally, certain subprograms may be parallelized and there are interfaces and modules available to make the use of MPI easier for the IMSL programmer.

Using Symmetric MultiProcessing (SMP) with the IMSL Fortran Numerical Library

Perhaps the easiest way to use Parallel Processing with the IMSL Fortran Numerical Library is to call one of the many SMP-enabled functions. When called, if there are multiple processors on the local machine the library automatically distributes threads across the CPUs. The primary advantage is that there is no additional code for the programmer to write. In fact, no knowledge of SMP or multi-threading is required by the user. Routines that fall into this category vary by platform and there are approximately 50 available including the following: `LIN_SOL_LSQ`, `LQRRV`, `LIN_EIG_GEN`, `EVCCG`, `NRIRR`, `LSARG` (and similar), `BVPMS`, `DASPG`, `BCNLS`, `DLPRS`, `ARMME`, `FRVAR`, `RNCHI`. These functions cover a variety of linear algebra, statistical analysis and optimization functionality.

The functions are multi-threaded and some underlying Basic Linear Algebra Subprograms (BLAS) may also be SMP-enabled (multi-threaded). These BLAS include both the standard implementation shipped with the product as well as optimized vendor-supplied BLAS for a particular platform. In this area, there are upwards of a hundred IMSL Fortran Numerical Library functions that could benefit if the vendor-supplied BLAS are SMP-enabled. Vendor-supplied libraries are platform-specific and include libraries such as Intel's Math Kernel Library, IBM's ESSL and the Sun Perflib. Almost every hardware platform has optimized libraries available and IMSL is able to utilize them wherever IMSL is supported.

Finally, programmers may write multi-threaded programs of their choosing. As mentioned previously, such multi-threading requires thread safe code. The IMSL Fortran Library is thread safe on platforms where the compiler supports OpenMP 2.0 or higher. While there are no additional SMP-enabled

routines on these platforms, the library has been tested to be thread-safe within the context of OpenMP and is suitable for use by programmers in such an environment.

Using Message Passing Interface (MPI) with the IMSL Fortran Numerical Library

The input data of an MPI application must be distributed across the network so that each node has its own block of data with which to work. The ***box data type*** is used with some routines and operators that are MPI-enabled. This notation derives from "a box of problems" that refers to independent linear algebra computations of the same kind and dimension, but different data. The box contains a number of racks of distinct problems. Each problem is independent of other problems in consecutive racks of the box, thus parallelism is a powerful tool for speeding computation of these disjoint problems.

The IMSL Fortran Numerical Library includes generic operators and functions that make linear algebra simple to code. This topic is covered in detail in the IMSL Fortran Library User's Guide, Math/Library, Chapter 10, Linear Algebra Operators and Generic Functions. Several of these operators are applicable to this discussion of parallelism because they work with the box data type and can be parallelized through MPI. The applicable operators are `.x.`, `.ix.`, `.xi.`, `.tx.`, `.xt.`, `.hx.` and `.xh.`. The applicable generic functions are `CHOL`, `COND`, `DET`, `EIG`, `FFT_BOX`, `IFFT_BOX`, `NORM`, `ORTH`, `RANK` and `SVD`.

Through the use of MPI modules and interfaces, the IMSL Numerical Libraries make writing a distributed application easier for the programmer. Instead of becoming intimately familiar with all of the MPI functions listed previously, one can simply use the IMSL function `MP_SETUP()`. When `MP_SETUP()` is called, standard MPI functions like `MPI_Initialized()` and `MPI_Init()` are called as necessary behind the scenes. The default MPI communicator (`MPI_COMM_WORLD`) has its handle returned, and various other initializations are done. If one calls `MP_SETUP(n)`, which takes an integer parameter, all the initialization is done and all of the nodes are further ranked according to performance by doing a quick matrix multiplication of size `nXn` on the available CPUs. When the parallel part of their application is completed, the programmer calls `MP_SETUP("Final")` and MPI execution is halted, the communicator is cleaned up, and any error messages (from IMSL or the system) are returned. Instead of having to learn and understand anywhere from six to over a hundred `MPI_` functions, the IMSL Fortran programmer can simply use `MP_SETUP()`.

The IMSL Fortran Library also includes two optimization routines that can leverage the MPI architecture directly:

- `PARALLEL_NONNEGATIVE_LSQ`
- `PARALLEL_BOUNDED_LSQ`

Complete details can be found in the IMSL Fortran User's Guide, Math Library, Section 1.2, "Large-Scale Parallel Solvers". The onus is on the user to break the data up into the appropriate number of blocks (equal to the

number of processors) and provide that information to the routine using the `IPART()` parameter.

ScaLAPACK[iv] is a suite of dense linear algebra solvers applicable for large-scale problems. The IMSL Fortran Libraries have been integrated with ScaLAPACK, but the ScaLAPACK libraries must be installed on a user's system independently of the IMSL Fortran libraries product. Using the ScaLAPACK integrated modules is recommended because many routines have lengthy parameter lists and the modules aid in avoiding mistakes such as missing arguments or mismatches of type, kind, or rank. Individual modules may be used or the inclusive `ScaLAPACK_Support` module can be used to save writing lines of code at the expense of an increase in compile time. Similar to the IMSL MPI utilities discussed previously, there are a number of ScaLAPACK routines than help developers create parallel applications without requiring expertise in the details of ScaLAPACK. The primary configuration function is `ScaLAPACK_SETUP` which sets up the processor grid. To calculate dimensions of a local distributed array using row and column blocking factors, `ScaLAPACK_GETDIM` is used. The routines `ScaLAPACK_MAP` and `ScaLAPACK_UNMAP` are used to map array data between global and local arrays in the two-dimensional block-cyclic form, while `ScaLAPACK_READ` and `ScaLAPACK_WRITE` aid with file input and output in this data format. Finally, `ScaLAPACK_EXIT` is called to clean up and finalize operations.

There are many examples in the IMSL Fortran User's Guide that use the Message Passing Interface. See the Math Library, Section 1.2 for two examples, each using `PARALLEL_NONNEGATIVE_LSQ` and `PARALLEL_BOUNDED_LSQ`, example 9 for `PDE_1D_MG` in the Math Library, Section 5.2.1, and the Introduction of the Math Library, Chapter 10, and the documentation for ScaLAPACK Supporting Modules in the Math Library, Section 11.1.

Parallelization with IMSL Libraries other than Fortran

The other IMSL libraries may also be used in multi-threaded applications. The IMSL C Numerical Library has been thread-safe since version 4.0 released in July 2000. Both OpenMP and MPI have implementations in C/C++. Starting in version 7.0, released in November 2008, OpenMP directives have been added to a variety of functions in the IMSL C Numerical Library. Like the Fortran Numerical Library, the C Numerical Library takes advantage of high performance vendor libraries for the best performance for many linear algebra functions. While not as popular as Fortran in the super computing world there are many cases where C is the programming language of choice and parallel processing is required. As a thread safe numerical library, the IMSL C product plays a key part in many multi-threaded and distributed applications.

For more details see The Parallel Performance of the IMSL C Numerical Library whitepaper at http://www.roguewave.com/resources/white-papers.aspx.

Java™ and the Microsoft™ .NET Framework have their own threading models. The IMSL C# Numerical Library uses the Task Parallel Library

(TPL) in .NET Framework 4.0 to enable parallelism on shared memory systems, especially multi-core systems. Staring with version 6.5, released April 2010, codes using TPL were added to several classes in the IMSL C# Library. While the JMSL™ Numerical Library is not multi-threaded, by utilizing the appropriate interfaces of the applicable language like labeling methods `synchronized` or requesting a `lock`, a programmer can make use of these IMSL Numerical Libraries in multi-threaded applications on their respective platforms. Further, the static methods in either JMSL Numerical Library or IMSL C# Numerical Library are thread safe, although objects instantiated using the classes are not guaranteed to be thread safe.

PV-WAVE®

PV-WAVE[v] contains an OpenMP implementation for the Oracle Solaris and Windows environments. One aspect of OpenMP is the straightforward manner in which one can apply loop-level parallelization to a section of code in an existing application that would otherwise be unsuitable for multi-threading.

Affected routines are any that apply unary and binary PV-WAVE operators to array data. These routines include, but are not limited to, the following:

- Trigonometry functions (`SIN`, `COS`, `TAN`, etc.)
- Binary operators (`+`, `-`, `*`, `/`, `MOD`, etc.)
- Relational operators (`GT`, `LT`, `LE`, `EQ`, etc.)
- Unary operators (`TOTAL`, `NOT`, `PRODUCT`, `AVG`, `ABS`, `WHERE`, etc.)
- Matrix multiplication (`#`)
- Tensor Ops (`TENSOR_ADD`, `TENSOR_MUL`, etc.)

The PV-WAVE user may customize the environment by using environment variables or PV-WAVE procedures. Such settings include the threshold array size to parallelize, the number of threads to create, the scheduling algorithm, and whether or not to use dynamic threading. Nearly linear speed-up was seen for a large variety of tests on a four CPU UltraSPARC Solaris machine for array sizes as small as 100 elements.

Summary

Increasing application performance is a common goal for many software developers for a variety of reasons. Historically, increasingly faster hardware has enabled applications to perform faster but not many organizations have the time or resources to continually upgrade processors.

Using additional hardware and running applications across multiple machines, in parallel, can also increase performance. The two historical parallel configurations, a single computer with multiple CPUs or using a networked set of discreet computers, each have benefits and tradeoffs. Single parallel computers with multiple CPUs are readily available from every major hardware manufacturer, but can be more costly than a computer with a single CPU. Networking discreet computers can be less costly but performance will not be as good as

in a shared memory multi-CPU environment. Newer configurations such as multi-core CPU offer their own set of advantages and disadvantages.

The software used in creating a parallel application depends on the hardware architecture and various standards, such as OpenMP and MPI have emerged that are ideal for different configurations.

URLs for in-line web links

[i] http://www.openmp.org/

[ii] http://www-unix.mcs.anl.gov/mpi/

[iii] http://www-unix.mcs.anl.gov/mpi/mpich/

[iv] http://www.netlib.org/scalapack/

[v] http://www.roguewave.com/products/pv-wave-family/pv-wave.aspx