jcis@epacis.org

Current paradigms in parallelization: a comparison of vectorization, OpenMP and MPI

Paulo F. Penteado[a] [1]

[a]Departamento de Astronomia, Instituto de Astronomia, Geofísica e Ciências Atmosféricas, Universidade de São Paulo

Received on *****, 2011 / accepted on *****, 2011

Abstract

This article presents a short overview of current parallelization concepts, focusing on vectorization, OpenMP and MPI, to obtain parallelization at the stream, thread and process levels. Vectorization allows compilers and interpreters to generate parallel code to perform the same operation over all the different elements of data arrays, and significantly improves in code robustness and organization. OpenMP is the most commonly used cross-platform solution to obtain thread parallelism. MPI is the most commonly used cross-platform solution to obtain process parallelism. The article ends with a discussion on the applications of each method. **Keywords**: Parallelization, vectorization, OpenMP, MPI.

1. Introduction

In recent years, the increase in computing power has been shifting from making faster processors to increasing the number of available processing cores. Therefore, to take advantage of greater processing power it is no longer sufficient to just buy faster computers to run code that only does one operation at a time; it has become necessary to make parallel code. However, parallelization with N units does not simply speedup a program by N times. There is an overhead, from the computation associated with managing the different processing elements. Also, a code that is only partly parallel (as is typical) has its performance gain limited by the serial (non-parallel) part. As the number of processing units increases, the parallelized portion of the code may take progressively less time, while the serial portion does not. Thus the serial part may quickly become the dominant in

[1]E-mail Corresponding Author: pp.penteado@gmail.com

the computation time. The simplest model to express this relation is Amdahl's law [1], which is valid when the parallel part scales linearly with the number of processing units, and the parallelization overhead is negligible. The speedup S for a code with a parallel fraction P running o N units is an asymptotic function ($S = (1 - P + P/N)^{-1}$).

The most basic two ways in which parallelization can be achieved are

- Data parallelism - The data to be processed is divided among the processing units. The processing can either be identical over all data values, or be allowed to vary depending on the data. This can be achieved by either a parallel program, or by calling multiple instances of a serial program (extrinsic parallelism).
- Task parallelism - There are multiple independent tasks to perform, which can then be run simultaneously, each on a different processing unit.

Another commonly used classification is Flynn's taxonomy [7]:

1. SISD - Single Instruction stream, Single Data stream - No parallelization (serial program).
2. SIMD - Single Instruction stream, Multiple Data stream - A form of data parallelism, where the same operation is performed by different processing units on different data elements.
3. MIMD - Multiple Instruction stream, Multiple Data stream - Task parallelism and some forms of data parallelism, where different processing units run (possibly) different code on different data elements.
4. MISD - Multiple Instruction stream, Single Data stream - Typically used in redundant systems.

In task parallelism, the maximum number of simultaneous tasks is determined by the problem's algorithm, and is usually only a few, while in data parallelism the number of simultaneous tasks is determined by the amount of data to process, typically making it possible to use whatever number of processing units available. This means that a program that uses only task parallelism will usually be more limited in the gain it can obtain from large numbers of computing cores.

Some of the most commonly used current parallelization paradigms are:

- Shared memory - each processing unit is a *thread*, and some of the program variables are shared among the threads.

  - Vectorization (language-dependent).
  - OpenMP (cross-platform).
  - Graphical Processing Unit (GPU) computing (varied options).
  - Manual thread management (highly language- and system- dependent).
- Distributed memory - each processing unit is a *process*, and all of a process' variables are private to it. Processes only communicate through messages or some shared resource (such as files).
  - MPI (cross-platform).
  - Manual process management (language- and system- dependent).
  - Grid / cloud computing (varied options).

Ahead, three of these paradigms will be discussed. All the code examples cited here can be found at `http://www.ppenteado.net/papers/iwcca/`, with `.f90`, `.c` and `.cpp` files for Fortran, C and C++ source code, and program use and outputs in the corresponding `.txt` files.

2. Vectorization

Vectorization means simply to express operation on arrays, of whatever number of dimensions, as array operations, instead of looping over elements. It is a concept independent of parallelization, and it is useful in making code easier to write, read and maintain, as well as more efficient, even if it is going to be executed serially. Because array operations are intrinsically SIMD, they are natural candidates for parallelization. Since array semantics already expresses what is to be done with the different array elements, it naturally provides the compiler or interpreter all the necessary information to parallelize the task. Parallelization through vectorization is implicit: the programmer only specifies the array operations to be done, and it is up to the compiler or interpreter to produce the parallel code. Vectorization is of particular relevance to scientific computing, where often substantial parts of the computation time are spent in operations on arrays. Without vectorization, these operations have to be done with loops over the elements, which are significantly worse for the readability and robustness of the code, since it is more verbose, and requiring manual bookkeeping with indexes and dimensions. Code execution can be much more efficient (up to several orders of magnitude) with array operations instead of loops, even without parallelization [4, 5].

Vectorization is not standardized over different languages. They vary widely both in the range of features offered and the semantics and syntax by which they are implemented:

1. C, Fortran 77, Perl - mo vectorization.
2. Fortran 90/95/2003/2008, C++, Java - simple vectorization: cumbersome for arrays over 1D (particularly over 2D); limited to operations over whole arrays or rectangular slices; improvement in Fortran from 90 to 2008.
3. C++ Boost.MultiArray (non-standard) library - improves on the functionality of the (standard) `vector`, with more advanced operations for over 1D.
4. R - better support for non-trivial manipulations, especially up to 2D (`matrix` class). As a dynamic language (interpreted at runtime), it offers more scope for more general array operations than the languages above (all static).
5. IDL, Python with the (non-standard) NumPy library - Also dynamic languages, with far more extensive vectorization capabilities, particularly for over 2D. Most support for complex slicing, element searches, inversion of indexes (histogram operations), non-rectangular slices (Numpy), fancy indexing, mixed dimension operations, redimensioning, 1D indexing in multidimensional arrays, empty arrays, and extensive language and library support to apply functions vectorially, including the ability to write code that works unaltered with arbitrary numbers of dimensions.

One important point to note when using multidimensional arrays is how they are stored internally: computer memory is always 1D, so all arrays over 1D are stored as a sequence of 1D slices. As such, there are two most obvious choices: to store contiguously either the leftmost or the rightmost dimension. That is: when storing the elements of a 2D array, will the first elements be the first line or the first row of the array? Different languages make different choices:

- Column-major storage - Fortran, R, Numpy and Boost.MultiArray; the leftmost dimension is contiguous.
- Row-major storage - C, C++ and Java, Numpy and Boost.MultiArray; the rightmost dimension is contiguous.

The most common situations when this choice matters are: moving data between different languages; reading or writing many many elements (more efficient, up to by orders of magnitude, if traveling in the same order as the elements

are stored); semantics of vector operations (to work with contiguous blocks of elements, to write literals, to read and write data, and to do concatenations).

With this large variation in the scope, semantics and syntax over different languages, a detailed discussion and comparison of them is beyond the scope of this article. The reader is referred to the documentation and books on each language for more information [10, 13, 11, 12, 8, 6, 16].

3. OpenMP

OpenMP (Open Multi-Processing) is a cross-platform standard released in 1997, to replace a multitude of incompatible vendor-specific solutions for thread parallelism. It has been implemented in Fortran, C and C++ compilers of most widely used platforms, and is at version 3.0 (2008), with 3.1 being discussed. Each code unit being executed simultaneously is a thread, with all of them having access to the same memory. All OpenMP constructs are compiler directives, which are interpreted as comments by compilers unaware of OpenMP. This facilitates writing code that can be compiled, unchanged, with and without OpenMP. Besides these constructs, OpenMP offers a few utility functions, typically used to query about the execution environment.

The structure of a typical OpenMP program starts with execution of a single thread (the *master thread*), which after any setup and non-parallelized work creates a team of other threads. At that point, execution proceeds in parallel over the threads, until the end of all the parallel regions, where the other threads are finished, with execution proceeding only at the master thread. There can be an arbitrary succession of such parallel regions enclosed by serial regions, and each parallel region can in principle have an arbitrary number of execution threads. The number of execution threads created is typically created based on external information (the environment variable `$OMP_NUM_THREADS`). Some implementations may also allow nested parallelism: parallel regions can have their own forks, creating their own sets of threads.

OpenMP's functionality is provided by a set of constructs, with optional clauses, and a set of utility library functions. More details are discussed by [2, 14, 3, 9], and at `http://openmp.org` and `http://compunity.org`.

- Constructs
  - Control constructs

    * `parallel` - The main OpenMP construct, it defines the extent of the regions to be executed in parallel. A simple `parallel` only specifies that the enclosed code lines are to be run, identically, by all threads. It is up to the programmer to make the threads do different work (examples `pp_omp_ex2` and `pp_omp_ex3`).
    * Conditional execution - Allow to only compile the corresponding lines of code if an OpenMP-aware compiler is used with OpenMP enabled. Often used in calls to OpenMP's functions, so that they are skipped when not using OpenMP (example `pp_omp_ex3`).
    * `if` - A form of conditional use of parallelization: If the condition evaluates to true at runtime, the code is run in multiple threads. Otherwise, the code runs serially (in a single thread). Commonly used to ensure that work is only separated in threads if the gains would compensate the overhead of parallelization.
- Worksharing constructs - specify several ways in which OpenMP can separate the work among threads. `loop` and `workshare` are intended for data parallelism, while `section` and `task` are intended for task parallelism.
    * `loop` - Called `for` for C and C++, and `do` for Fortran, provides a simple way to specify a loop whose iterations will be divided among the threads. Each thread is given a subset of the iterations, receiving the appropriate value for its loop counter (example `pp_omp_ex4`).
    * `section` - Each `section` is a region of code that will be executed by a thread. Often used for task parallelism, to specify each task to be run by a different thread (example `pp_omp_ex5`).
    * `workshare` - Used for vectorization by OpenMP: vector operations enclosed in a `workshare` region are executed in parallel by the threads (example `pp_omp_ex6`). Only available for Fortran, as it uses vectorial semantics.
    * `single` - Specifies lines in a parallel region that should be executed by only one of the threads.
    * `task` - An alternative to `section` for task parallelism, with the possibility of dynamic task creation.
- Synchronization constructs - change how threads execute depending on the state of the other threads. Used to avoid synchronization prob-

lems, such as reading values from a variable before all threads have finished updating its value.

  * `barrier` - When threads encounter a barrier, they must wait until all other threads have encountered that barrier before they can proceed. Used to ensure that a shared variable being written on by several threads has been completely updated when its values are needed.
  * `ordered` - Specifies that a region of code must be executed by the threads in the same order as the loop iterations.
  * `master` - Specifies lines of code to be executed only by the master thread, without the implicit barrier of `single`.
  * `critical` - Specifies a region of code to be executed by only one thread at a time. Often used for the lines of code that must access the same shared resource, such as input and output streams.
  * `atomic`, `reduction` - Specialized alternatives to `critical`.
  * `locks` - Lock variables control access to shared resources: only one thread at a time can hold a lock.

- Clauses - modify the behavior of constructs.
  - `shared` - Specifies that the same variable is common to all threads. Any changes to the variable done by a thread will be seen by any threads that subsequently read it.
  - `private` - Specifies that a copy of the variable will be created for each thread. Any changes to that variable by a thread will have no effect on the other threads. Block variables (C and C++) are always private.
  - `lastprivate`, `firstprivate` - Similar to `private`, but transporting the value of the variable when entering or exiting the parallel region.
  - `default` - Specifies which access clause is to be taken as default. It is usually recommended to set the default to `none`, so that the access will have to be explicitly specified for every variable.
  - `copyin`, `copyprivate` - Initialize or copy the value of a private variable from the master or single thread to the others.
  - `num_threads` - Requests that the parallel region be executed with a (compile-time) constant number of threads, instead of the usual run-time determination of thread numbers by the environment or the li-

brary functions. Typically used in task parallelism, where the algorithm determines how many threads would be used.

  - `nowait` - Suppresses a construct's implicit barrier.
  - `schedule` - Changes the algorithm to assign iterations among threads in a `loop` construct, for cases when some iterations may take longer than others, to balance the load among threads.

- Library functions - To communicate, get and set execution environment parameters, synchronize and share resources. Functions are typically placed under conditional compilation, so that they are ignored when the program is not compiled for OpenMP (example `pp_omp_ex3`).

4. MPI

To fulfill the need for a cross-platform standard for process parallelism, MPI (Message Passing Interface) was created, with the 1.0 version released in 1994, and is currently at version 2.2 (2009), with 3.0 under discussion. MPI is most often implemented for C, C++ and Fortran, though it has been implemented in other platforms (Java, Python, IDL). MPI consists solely of a library of functions. All parallelization in the code is explicit, achieved through calls to MPI's functions, which communicate (pass messages, as the name implies) with the MPI environment. The MPI environment is set up by the MPI process managers (`mpirun`, `mpiexec`), which starts the program, and handles creation and destruction of processes. As such, MPI is a relatively simple standard, at the price of added complexity when writing the code, since the interaction between processes must be written explicitly by the programmer. Though through conditional compilation it is possible to write an MPI program that also runs serially without MPI, this would usually be a cumbersome task.

Typically, an MPI program starts by the function calls to do the MPI initialization, and ends with the MPI clean-up. In between, the code usually will be common to all processes (SPMD - Single Process, Multiple Data), so that each process must make use of the MPI functions to decide what part of the job it must do. There is no shared memory among the processes, so all communication normally happens sending and receiving MPI messages. This paradigm can be applied to both data and task parallelism.

The processes send and receive data through the domains of the communicators, which can be used to separate processes in a hierarchy, and each process in a domain is identified by its rank in that communicator. The processes may be executed in the same processor core, by multiple cores in the same computer, or multiple computers over a network, but MPI always presents the same API, regardless of how the communication is implemented. MPI offers a variety of communication options:

- Blocking operations: When a process encounters a blocking operation, it must wait until it is finished before it can proceed. With non-blocking operations, the process may continue before the operation is finished.
- Synchronous operations: When a process sends data through a synchronous operation, it must wait until the target process(es) start receiving the data (that is, analogous to a phone call). With asynchronous communications, the process is free to proceed with other tasks while the data is being delivered (that is, analogous to sending an e-mail).

Some of the main MPI functions in the standard APIs (C, C++ and Fortran) are listed below. More details are given by [15, 14, 3, 9] and at `http://www.mpi-forum.org`.

- Execution control and environment information
  - `MPI_Init`, `MPI_Finalize` - Used by only one process to set up the MPI environment and to clean it up at the end.
  - `MPI_Abort` - Used by any process at any time, to request a termination of all processes. Typically used to handle exceptions.
  - `MPI_Barrier` - When a process encounters a barrier, it waits until all other processes reach the barrier before proceeding. Used to enforce synchronization between the processes, typically when processes will need some result being generated by other processes.
  - `MPI_Test`, `MPI_Wait` - Verifies if / waits until a data transfer has been completed.
  - `MPI_comm_size`, `MPI_Comm_rank`, `MPI_Get_processor_name` - Obtain the number of processes, the process rank and the name of the processor running a process.

- Data transfer
  - `MPI_Send`, `MPI_Recv`, `MPI_Sendrecv` - Send or/and receive a message (data), to a specific process, in a blocking operation.
  - `MPI_Isend`, `MPI_Irecv` - Send or receive a message to another process, in a non-blocking operation.
  - `MPI_Bcast` - Send the same message to all processes.
  - `MPI_Reduce` - Combines a data element from each process into a single value, through a series of binary operations (such as addition, or picking the maximum value), providing the result to all processes.
  - `MPI_Gather`, `Scatter` - Retrieve / send a data element from / to each process into / from a set of values, providing the result to all processes.

These functions are considerably simplified in the object-oriented API offered by the Boost.MPI library. In particular, this library can use the Boost serialization library and the C++ standard containers to make transfer of arrays and structures considerably simpler than with the standard API. This difference, as well as the general structure of a simple program and uses of some of the functions above, can be seen in the example files (`pp_mp_ex1`, `pp_mpi_ex2`).

5. Comparison and discussion

There is no single approach that will be the best for all problems, and most problems are best served by hybrid solutions, making use of different forms of parallelization for different parts of the work. Therefore, knowing the characteristics of the different options is paramount to making the best choices [15, 14, 3, 9]. Comparatively, these three paradigms can be resumed as:

1. Vectorization
   - Implicit: the compiler / interpreter does the parallelization,
   - Parallel code is indistinguishable from serial code.
   - Limited to shared memory, in the simplest cases of data parallelism.
   - Languages vary widely in capabilities, semantics and syntax.
2. OpenMP

- Language-, compiler- and platform-independent well-stablished standard for data and task parallelism.
- Easy to keep compatibility with serial code.
- Limited to shared memory.
- Usually only implemented for C, C++ and Fortran.

3. MPI

   - Most widely used standard for distributed memory HPC.
   - Multiple processes in a single computer or in several across a network.
   - Usually requires to structure the whole program for MPI.
   - Without the (non-standard) Boost.MPI library, cumbersome to transfer data more complex than arrays of primitive data types.
   - Still has some variation between implementations.

Parallelization is not always done directly when developing the application: it may be the result of simply using ready libraries that were made to use parallelization. This is often the case with tasks that are common to many scientific and computing areas, such as linear algebra, Fourier transforms, image processing operations, common algorithms (sorting, containers, etc.), and common scientific problems (CFD, MHD, N-bodies, nearest neighbours, etc.).

ACKNOWLEDGMENTS: This work was supported by a FAPESP grant (2007/57447-6) and the organizers of the I Workshop de Computação Científica em Astronomia, held at Unicsul in June 2011.

References

[1] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[2] B. Chapman, G. Jost, and R. Pas van der. *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press, 2007.

[3] R. Cook. *An Introduction to Parallel Programming with OpenMP, PThreads and MPI*. Cook's Books, 2011.

[4] N. G. Dickson, K. Karimi, and F. Hamze. Importance of explicit vectorization for CPU and GPU software performance. *Journal of Computational Physics*, 230:5383–5398, June 2011.

[5] D.F. Fanning. My IDL Program Speed Improved by a Factor of 8100!!! http://www.idlcoyote.com/code_tips/slowloops.html, July 2003.

[6] D.F. Fanning. The IDL Way. http://www.idlcoyote.com/idl_way/idl_way.php, September 2011.

[7] Michael J. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9):948 –960, sept. 1972.

[8] M. Galloy. *Modern IDL*. http://modernidl.idldev.com/, 2011.

[9] G. Hager and G. Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, 2010.

[10] Hans Petter Langtangen. *Python Scripting for Computational Science*. Springer Publishing Company, Incorporated, 2009.

[11] Community NumPy. *NumPy Reference Guide*. http://docs.scipy.org/doc/, August 2011.

[12] Community NumPy. *NumPy User Guide*. http://docs.scipy.org/doc/, August 2011.

[13] T.E. Oliphant. *A Guide to Numpy*. Trelgol, http://www.tramy.us/, dec 2006.

[14] P. Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann, 2011.

[15] T. Rauber and G. Rünger. *Parallel Programming: for Multicore and Cluster Systems*. Springer, 2010.

[16] J.D. Smith and D.F. Fanning. HISTOGRAM: The Breathless Horror and Disgust. http://www.idlcoyote.com/tips/histogram_tutorial.html, March 2011.