Journal of Computational Interdisciplinary Sciences (2012) 3(3): 119-125

Printed version ISSN 1983-8409 / Online version ISSN 2177-8833
http://epacis.net
doi: 10.6062/jcis.2012.03.03.0057

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

Paulo F. Penteado

Manuscript received on July 28, 2012 / accepted on February 6, 2014

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 code robustness and organization. OpenMP and MPI are the most commonly used cross-platform solutions to obtain thread and process parallelism.

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, due to thermal and electrical constraints limiting processors' operating frequencies. Therefore, 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. 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 on 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 may or may not be identical over all data values. 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.
2. SIMD – Single Instruction stream, Multiple Data stream – Data parallelism: Same operation performed by different processing units on different data.
3. MIMD – Multiple Instruction stream, Multiple Data stream – Task parallelism and some forms of data parallelism, where different processing units perform (possibly) different operations on different data elements.

Department of Physics and Astronomy, Northern Arizona University, Flagstaff, AZ USA – E-mail: pp.penteado@gmail.com

4. MISD – Multiple Instruction stream, Single Data stream – For redundancy.

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.

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: vectorization, OpenMP and MPI (Fig. 1). All the code examples cited here can be found at `http://www.ppenteado.net/papers/iwcca/`, with `.f90`, `.c`, `.cpp` and `.txt` files for Fortran, C and C++ source code, and program use and outputs.

2 VECTORIZATION

Vectorization means simply to express operations on whole arrays (or array slices), instead of looping over elements. It is a concept independent of parallelization, and it is important 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. 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. Languages vary widely both in the range of features offered and the semantics and syntax by which they are implemented:

1. C, Fortran 77, Perl – no 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 – Far more extensive vectorization capabilities, particularly for over 2D. Most support for complex slicing, element searches, inversion of indexes, non-rectangular slices (Numpy), 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.

Since computer memory is always 1D, all arrays with dimensionality greater than or equal to 2 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++, Java, Numpy and Boost.MultiArray; the rightmost dimension is contiguous.

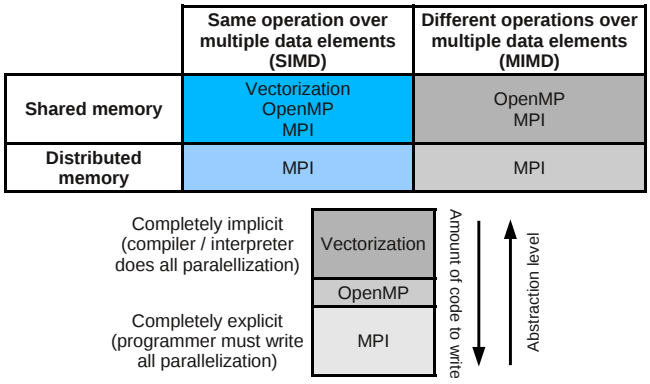

Figure 1 – Comparison of three parallelization paradigms. Top: With respect to the number of different simultaneous operations and the need for shared memory. Bottom: With respect to the abstraction level, which determines how much code the programmer has to write to achieve parallelization.

The most common situations when this choice matters are: moving data between different languages; reading or writing 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 with code examples 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].

Computationally heavy scientific codes almost always often have to apply the same independent operation to large numbers of array elements: elements in linear algebra matrices, pixels in images, spectra and data cubes, voxels (volume elements) and particles in simulations, points in time series and other functions, etc. Therefore, vectorization is of general use in scientific computing, either as the only form of parallelization, or inside code used in other forms of parallelization. Vectorization also allows for better compiler optimizations of memory access and use of vector instructions (such as SSE and AVX) to do SIMD on single processor cores.

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. In OpenMP 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 determined from 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`.

- **Control constructs:**
  - `parallel` – The main OpenMP construct, it defines the extent of the regions to be executed in parallel. If used without other constructs, it is left to the programmer to make different threads do different work (examples `pp_omp_ex2` and `pp_omp_ex3`).
  - Conditionals – Lines of code are only compiled by a compiler with OpenMP enabled. Used in calls to OpenMP's functions, so that they are skipped when

not using OpenMP (example `pp_omp_ex3`). In C/C++:

```
#pragma omp parallel
{printf("This is thread %i
\n", omp_get_thread_num());}
```

- `if` – A form of conditional use of parallelization, evaluated at runtime. 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` in C and C++, and `do` in Fortran. Each thread does a subset of the loop iterations (example `pp_omp_ex4`). In C/C++:

    ```
    #pragma omp parallel for
    shared(a,n)
    for (int i=0; i<n; i++)
    a[i]=2*i;
    ```

  - `section`, `task` – Used to specify regions of code that will be executed by different threads, in task parallelism (example `pp_omp_ex5`).

  - `workshare` – Vectorization: vector operations (Fortran only) in a `workshare` region are executed in parallel by the threads (example `pp_omp_ex6.f90`):

    ```
    !$omp parallel shared(b,n)
    private(j)
    !$omp workshare
    forall (j=1:n) b(j)=2*j
    !$omp end workshare
    !$omp end parallel
    ```

  - `single` – Code in a parallel region to be executed by only one thread.

- **Synchronization constructs** – change how threads execute depending on the state of the other threads, avoiding synchronization problems (such as reading values from a variable before all threads have finished updating it):

  - `barrier` – When threads encounter a barrier, they must wait until all threads have encountered that barrier before they can proceed.

  - `ordered`, `master` – Specify that a region of code is to be executed in order among threads or by just the master thread.

  - `critical`, `atomic`, `reduction` – Different specializations for code that must use a shared resource (shared variables, input/output streams) with restrictions to simultaneous use.

  - `locks` – Control access to shared resources: only one thread at a time can hold a lock.

- Main **Clauses** – modify the behavior of constructs:

  - `shared` – The same variable is common to all threads.

  - `private`, `lastprivate`, `firstprivate` – Each thread has its own instance of the variable. Block variables (C, C++) are always private.

  - `schedule` – Specifies 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. Function calls are typically placed under conditional compilation, so that they are ignored when the program is not compiled for OpenMP (example `pp_omp_ex3`).

Since scientific computing nearly always makes use of applying the same processing to many data elements, OpenMP most often is used to easily obtain data parallelism. Therefore, it is often applied to the same problems cited in the previous section as examples of vectorization uses, including those cases where the processing cannot be conveniently expressed in terms or array operations. Through OpenMP, these applications can make use of the multiple processors with shared memory present in most current computers, without all the difficulties of the explicit interprocess communication required by MPI.

4 MPI

To fulfill the need for a cross-platform standard for process parallelism, MPI (Message Passing Interface) was released in 1994.

It 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 pass messages to and from the MPI environment. The MPI environment is set up by the MPI process managers (`mpirun`, `mpiexec`), which start the program and handle creation and destruction of processes. As such, MPI is a relatively simple (low-level) standard, at the price of added complexity when writing the code, since the interaction between processes must be written explicitly by the programmer (Fig. 1). Though through conditional compilation it is possible to write an MPI program that also runs serially without MPI, this is usually cumbersome.

Typically, an MPI program begins with 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/Program, 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 by 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 MPI communicators. 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 Application Programming Interface (API). 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 (analogous to a phone call). With asynchronous communications, the process is free to proceed with other tasks while the data is being delivered (analogous to 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 to request a termination of all processes.
  - `MPI_Barrier` – When a process encounters a barrier, it waits until all processes reach the barrier, before proceeding.
  - `MPI_Test`, `MPI_Wait` – Verifies if/waits until a transfer completes.
  - `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/from 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).
  - `MPI_Gather`, `Scatter` – Retrieve/send a data element from/to each process into/from a set of values.

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_mpi_ex1`, `pp_mpi_ex2`).

MPI is most often used when one needs to make use of distributed memory systems (computer clusters). Since the message passing is usually much slower than memory access, MPI is most

efficient when the computation done by each process is largely independent from that done by the others. This is often the case with data parallel problems such as repeating processing for large numbers of experiments (different observations, models, images, etc), simulations where each particle/model region has computations independent of the others (inside the time steps in dynamical fluid or particle models, different wavelengths in radiation models, etc.). Another important class of problems well-suited for MPI is task parallelism, where there are different independent tasks that do not need much communication between them (generating visualizations and writing files of one data set while processing the next data set, for instance).

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 summarized as:

1. Vectorization
    - Implicit: the compiler/interpreter does the parallelization,
    - Limited to shared memory, in the simplest cases of data parallelism.
    - Languages vary widely in capabilities, semantics and syntax.
    - Can make use of SIMD in single processor cores.
    - Makes code more robust, readable and maintainable.
2. OpenMP
    - Language-, compiler- and platform-independent well-established standard for shared memory data and task parallelism.
    - Easy to keep compatibility with serial code.
    - Usually only implemented for C, C++ and Fortran.
3. MPI
    - Most widely used standard for distributed memory parallelization.
    - Multiple processes, in a single computer or in several computers.
    - 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.
    - Not completely standardized.

Parallelization is not always done directly when developing the application: it may be the result of simply using libraries that make use of 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 neighbors, etc.).

In summary: in scientific computing usually there is large scope for vectorization, and it should be applied everywhere possible, even when parallelization is not relevant, as it makes the code considerably more robust, maintainable, verifiable and optimizable. The next step is identifying where the problem can be parallelized beyond vectorization, either by data or task parallelism. These operations can run with the fast and simple shared memory parallelization offered by OpenMP. If further parallelization is desired, to run the code in multiple computing nodes, the code can be structured to run several processes, with communication done by MPI. In this nested approach, MPI is used to run processes in different computing nodes, with each process using OpenMP to run several threads in different cores, and vectorization to organize the code and allow further optimizations, including SIMD in each processor core.

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] AMDAHL GM. 1967. 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. ACM.

[2] CHAPMAN B, JOST G & VAN DER PAS R. 2007. Using OpenMP: Portable Shared Memory Parallel Programming. MIT Press.

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

[4] DICKSON NG, KARIMI K & HAMZE F. 2011. Importance of explicit vectorization for CPU and GPU software performance. Journal of Computational Physics, 230: 5383–5398, June.

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

[6] FANNING DF. 2011. The IDL Way. http://www.idlcoyote.com/idl_way/idl_way.php, September.

[7] FLYNN MJ. 1972. Some computer organizations and their effectiveness. Computers, IEEE Transactions on, C-21(9): 948–960, September.

[8] GALLOY M. 2011. Modern IDL. http://modernidl.idldev.com/.

[9] HAGER G & WELLEIN G. 2010. Introduction to High Performance Computing for Scientists and Engineers. CRC Press.

[10] LANGTANGEN HP. 2009. Python Scripting for Computational Science. Springer Publishing Company, Incorporated.

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

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

[13] OLIPHANT TE. 2006. A Guide to Numpy. Trelgol.

[14] PACHECO P. 2011. An Introduction to Parallel Programming. Morgan Kaufmann.

[15] RAUBER T & RÜNGER G. 2010. Parallel Programming: for Multicore and Cluster Systems. Springer.

[16] SMITH JD & FANNING DF. 2011. HISTOGRAM: The Breathless Horror and Disgust. http://www.idlcoyote.com/tips/histogram_tutorial.html, March.