Survey on Parallel Programming Model

Henry Kasim[1,2], Verdi March[1,3], Rita Zhang[1], and Simon See[1,2]

[1] Asia-Pacific Science and Technology Center (APSTC), Sun Microsystems
[2] Department of Mechanical & Aerospace Engineering, Nanyang Technological University
[3] Department of Computer Science, National University of Singapore
{henry.kasim,verdi.march,rita.zhang,simon.see}@sun.com

Abstract. The development of microprocessors design has been shifting to multi-core architectures. Therefore, it is expected that parallelism will play a significant role in future generations of applications. Throughout the years, there has been a myriad number of parallel programming models proposed. In choosing a parallel programming model, not only the performance aspect is important, but also qualitative the aspect of how well parallelism is abstracted to developers. A model with a well abstraction of parallelism leads to a higher application-development productivity. In this paper, we propose seven criteria to qualitatively evaluate parallel programming models. Our focus is on how parallelism is abstracted and presented to application developers. As a case study, we use these criteria to investigate six well-known parallel programming models in the HPC community.

Keywords: shared memory, distributed memory, Pthreads, OpenMP, CUDA, MPI, UPC, Fortress.

1 Introduction

The aim of parallel computing is to increase an application's performance by executing the application on multiple processors. While parallel computing has been traditionally associated with the HPC (high performance computing) community, it is becoming more prevalent for the mainstream computing due to the recent development of commodity multi-core architecture. The multi-core architecture, and soon many-core, is a new paradigm in keeping up with the Moore's law. It is motivated by challenges to traditional paradigm of continuously increasing CPU frequency: physical limit of transistors size, power consumption, and heat dissipation [1, 2]. Consequently, it is expected that future generations of applications would heavily exploit the parallelism offered by the multi-core architecture.

There are two main approaches to parallelize applications: *auto parallelization* and *parallel programming*; they differ in terms of the achievable application performance and ease of parallelization. The auto-parallelization approach, e.g. ILP (instruction level parallelism) or parallel compilers [3], automatically parallelizes applications that have been developed using sequential programming

models. The advantage of this approach is that existing/legacy applications need not be modified, e.g. applications just need to be recompiled with a parallel compiler. Therefore, programmers need not to learn new programming paradigms. However, this also becomes a limiting factor in exploiting a higher degree of parallelism: it is extremely challenging to automatically transform algorithms with a sequential nature into parallel ones. In contrast to auto parallelization, with the parallel programming approach, applications are specifically developed to exploit parallelism. Generally, developing a parallel application involves partitioning workload into tasks and mapping of tasks into workers. Parallel programming is perceived to result in higher performance gain than auto parallelization, but at the expense of more parallelization efforts.

Throughout the years, there have been a myriad number of parallel programming models proposed. A typical consideration in choosing a model is the performance of the resulted applications. However, it is equally important to also consider qualitative aspects of models. One such qualitative aspect is how parallelism is abstracted and presented to application developers. To evaluate this aspect, we propose that each model is evaluated based on seven criteria: (i) system architecture, (ii) programming methodologies, (iii) worker management, (iv) workload partitioning scheme, (v) task-to-worker mapping, (vi) synchronization, and (vii) communication model. Out list of criteria is inspired by Asanovic et. al. [4].

In this paper, we describe seven qualitative criteria to evaluate parallel programming models. Our goal is to emphasize to people new to parallelism that apart from performance of resulted applications, one should also consider how the chosen programming model affects the productivity of software development. The contributions of this paper are two fold. Firstly, we extend the four criteria in Asanovic et. al. [4] with three new criteria (i.e. system architecture, programming methodologies and worker management). Secondly, we present an investigation of six parallel programming models in the HPC community: three well-established models (i.e. Pthreads [5], OpenMP [6, 7], and MPI [8]) and three relatively new models (i.e. UPC [9, 10], Fortress [11], and CUDA [12]).

The remainder of this paper is organized as follow. Section 2 defines the seven criteria and Section 3 present a study of six parallel programming models based on the criteria. Finally, Section 4 summarizes this paper.

2 The Seven Criteria

In this section, we describe seven criteria to qualitatively evaluate a parallel programming model.

1. *System Architecture*
   We consider two architectures: *shared memory* and *distributed memory*. Shared memory architecture refers to systems such as an SMP/MPP node whereby all processors share a single address space. With such models, applications can run and utilize only processors within a single node. On the other hand,

distributed memory architecture refers to systems such as a cluster of compute nodes whereby there is one address space per node.
Fig. 1 illustrates the supported system architecture of the six programming models. As can be seen, Pthreads, OpenMP and CUDA support shared memory architecture, and thus can only run and utilize processors within a single node. On the other hand, MPI, UPC and Fortress also support distributed memory architecture so that applications developed with these model can run on single node (i.e. shared memory architecture) or multiple nodes.

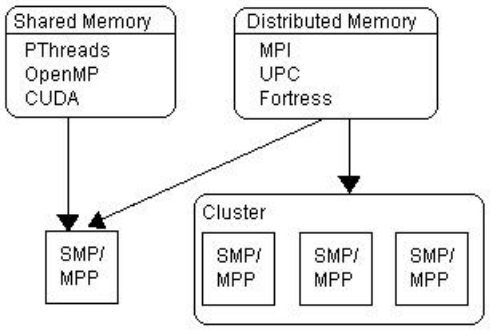

Fig. 1: Six Programming Models and their Supported System Architecture

2. *Programming Methodologies*
   We look at how parallelism capabilities are exposed to programmers. For examples, API, special directives, new language specification, etc.
3. *Worker Management*
   This criteria looks at the creation of the unit of worker, threads or processors. Worker management is *implicit* when there is no need for programmers to manage the lifetime of workers. Rather, they need to only specify, for example, the number of unit of workers required or the section of code to be run in parallel. In *explicit* approach, programmer needs to code the creation and destruction of workers.
4. *Workload Partitioning Scheme*
   Worker partitioning defines how the workload are divided into smaller chunks called *tasks*. In *implicit* approach, typically programmers needs to only specify that a workload can be processed in parallel. How the workload is actually partitioned into tasks need not be managed by programmers. In contrast, with the *explicit* approach, programmers need to manually decide how workload is partitioned.
5. *Task-to-Worker Mapping*
   Task-to-worker mapping defines how tasks are map onto workers. In the *implicit* approach, programmers do not need to specify which worker is re-

sponsible for a particular task. In contrast, the *explicit* approach requires programmers to manage how tasks are assigned to workers.

6. *Synchronization*
   Synchronization defines the time order in which workers access shared data. In *implicit* synchronization, there is no or little programming effort done by programmers: either no synchronization constructs are needed or it is sufficient to only specify that a synchronization is needed. In *explicit* synchronization, programmers are required to manage the worker's access to the shared.
7. *Communication Model*
   This aspect looks at the communication paradigm used by a model.

3 Parallel Programming Model

In this section, we evaluate six parallel programming models using the criteria presented in Section 2. The overall summary is shown in Table 1.

Table 1: Evaluation of Six Parallel Programming Models

(a) Shared Memory

Criteria	**MPI**	**UPC**	**Fortress**
Unit of Workers	Thread	Thread	Thread
Programming Methodologies	API, C, Fortran	API, C, Fortran	API, Extension to C
Worker Management	Explicit	Implicit	Implicit
Workload Partitioning	Explicit	Implicit	Explicit
Worker Mapping	Explicit	Implicit	Explicit
Synchronization	Explicit	Implicit	Explicit
Communication Model	Shared Address Space	Shared Address Space	Shared Address Space

(b) Distributed Memory

Criteria	**MPI**	**UPC**	**Fortress**
Unit of Workers	Process	Thread	Thread
Programming Methodologies	API, C, Fortran	API, C	New Language
Worker Management	Implicit	Implicit	Implicit/Explicit
Workload Partitioning	Explicit	Implicit/Explicit	Implicit/Explicit
Worker Mapping	Explicit	Implicit/Explicit	Implicit/Explicit
Synchronization	Implicit	Implicit/Explicit	Implicit/Explicit
Communication Model	Message Passing	Partitioned Global Address Space	Global Address Space

3.1 Pthreads

Pthreads or Portable Operating System Interface (POSIX) Threads is a set of C programming language types and procedure calls [5]. Pthreads is implemented as a header (`pthread.h`) and a library for creating and manipulating each of the workers called threads.

Worker management in Pthreads requires programmer to explicitly create and destroy threads by making use of `pthread_create` and `pthread_exit` function. Function `pthread_create` requires four parameters: (i) the thread used to run tasks, (ii) attribute, (iii) tasks to be run by thread in routine call, and (iv) routine argument. The thread created will run the routine until `pthread_exit` function has been called.

Workload partitioning and task mapping are explicitly specified by programmers as arguments to `pthread_create`. The workload partitioning is specified by programmers on the third passing parameter in the form of a routine call, while task mapping is specify on the first passing parameters in the `pthreads_create` function. A thread can join other threads using `pthread_join`. When the function is called, the calling thread will hold its execution until the target thread finish before joining the threads.

When multiple threads access the shared data, programmers have to be aware of data race and deadlocks. To protect critical section, i.e. the portion of code that accesses shared data, Pthreads provides mutex (mutual exclusion) and semaphore [13]. Mutex permits only one thread to enter a critical section at a time, whereas semaphore allows several threads to enter a critical section.

3.2 OpenMP

OpenMP is an open specification for shared memory parallelism [6, 7]. It consists of a set of compiler directives, callable runtime library routines and environment variables that extend Fortran, C and C++ programs. OpenMP is portable across the shared memory architecture. The unit of workers in OpenMP is threads.

The worker management is implicit. Special directives are used to specify that a section of code is to be run in parallel. The number of threads to be used is specified using an out-of-band mechanism which is an environment variable. Thus, unlike Pthread, there is no need for programmers to manage the lifetime of threads.

Workload partitioning and task-to-worker mapping require a relatively few programming effort. Programmers just need to specify compiler directives to denote a parallel region, namely (i) `#pragma omp parallel {}` for C/C++, and (ii) `!$omp parallel` and `!$omp end parallel` for Fortran. OpenMP also abstracts away how workload (e.g. an array) is divided into tasks (e.g. sub-arrays) and how tasks are assigned to threads.

OpenMP supports several constructs to support implicit synchronization where programmers specify only where synchronization occurs (Table 2). The actual synchronization mechanism is thus relieved from the programmers' responsibility.

Table 2: Synchronization Constructs in OpenMP

Construct	**Description**
`Barrier`	Allow synchronization on all threads within the same group
`Atomic`	Allow all threads execute, but only one of load or store at a time
`Ordered`	Allow the block of code to be execute sequentially
`Flush`	Ensure all threads have a consistent view of certain objects in memory

3.3 CUDA

CUDA (Compute Unified Device Architecture) is the extension of C programming language designed to support of parallel processing on Nvidia GPU (Graphics Processing Unit) [12]. CUDA views a parallel system as consisting of a host device (i.e. CPU) and computation resource (i.e. GPU). The computation of tasks is done in GPU by a set of threads that run in parallel. The GPU architecture for threads consist of two-level hierarchy, namely *block* and *grid* (Fig. 2). Block is a set of tightly coupled threads where each thread is identified by a thread ID, while grid is a set of loosely coupled of blocks with similar size and dimension.

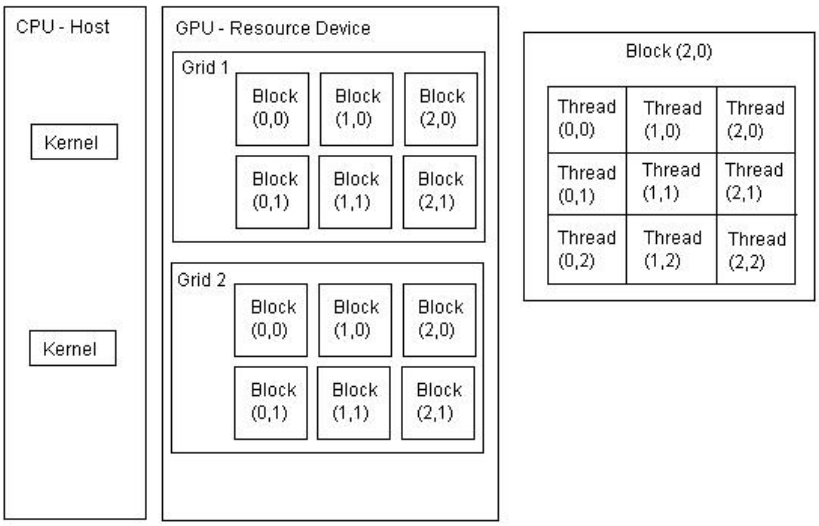

Fig. 2: CUDA Architecture

Worker management in CUDA is done implicitly; programmers do not manage thread creations and destructions. They just need to specify the dimension

of the grid and block required to process a certain task. While workload partitioning and worker mapping in CUDA is done explicitly. Programmers have to define the workload to be run in parallel by using `Global_Function<<<dimGrid, dimBlock>>> (Arguments)` construct where (i) `Global_Function` is the global function call to be run in threads, (ii) `dimGrid` is the dimension and size of the grid, (iii) `dimBlock` is the dimension and size of each block and (iv) `Arguments` represent the passing value for the global function. The task to worker mapping of CUDA programming is defined on `<<<dimGrid, dimBlock>>>` within the command call mentioned before.

Synchronization for all threads in CUDA is done implicitly through function `__syncthreads()`. This function will coordinate communication among threads of the same block. The function requires a minimum of 4 clock cycles as the overhead, i.e. when no thread is waiting for other threads.

3.4 MPI

Message Passing Interface (MPI) is a specification for message passing operations [8]. It defines each worker as a *process*. MPI is currently the de-facto standard for developing HPC applications on distributed memory architecture. It provides language bindings for C, C++[4], and Fortran. Some of the well-known MPI implementation includes OpenMPI [14], MVAPICH [15], MPICH [16], GridMPI [17], and LAM/MPI [18].

Worker management is done implicitly whereby it is not necessary to code the creation, scheduling, or destruction of processes. Instead, one only needs to use a command-line tool, mpirun, to tell the MPI runtime how many processes are needed, and optionally the mapping of processes to processors. Based on this information, the runtime infrastructure will then carry out the worker management on behalf of users.

Workload partitioning and task mapping have to be done by programmers, similar to Pthread. Programmers have to manage what tasks to be computed by each process. As an example, given a 2-D array (i.e. the workload), one can use a process' identifier (i.e. rank) to determine which sub-array (i.e. a task) the process will compute. Communication among processes adopts the message-passing paradigm where data sharing is done by one process sending the data to other processes. MPI broadly classifies its message-passing operations as *point-to-point* and *collective*. Point-to-point operations such as the `MPI_Send`/`MPI_Recv` pair facilitate communications between processes, whereas collective operations such as `MPI_Bcast` facilitate communications involving more than two processes.

`MPI_Barrier` is used to specify that a synchronization is needed. The barrier operation blocks each process from continuing its execution until all processes have enter the barrier. A typical usage of barrier is to ensure that global data has been dispersed to appropriate processes.

[4] Supported only on MPI-2

3.5 UPC

UPC (Unified Parallel C) is a parallel programming language for shared memory architecture and distributed memory architecture [9, 10]. Regardless of the system architecture, UPC adopts the concept of partitioned memory. With this concept, programmers view the system as one global address space which is logically partitioned into a number of per-thread address spaces. Each thread has two types of memory accesses: to its own private address space or to other threads' address space. Accesses to both types of per-thread address space use the same syntax. To improve the performance of memory accesses, UPC introduces the concept of thread affinity. With this feature, UPC optimizes memory-access performance between a thread and the per-thread address space where the thread has been bound.

In UPC, workload management is implicit, while workload partitioning and worker mapping can be either implicit or explicit. For worker management, programmers just need to specify number of threads required during the call on the command-line tools, `upcrun`. Implicit workload partitioning and task mapping are supported through an API called `upc_forall` which is similar to for iteration in C programming, except that the content of the iteration will be run in parallel. When this API is used, there is no need for additional programming effort for programmers to map the task to threads. The explicit approach in UPC for workload partitioning and worker mapping is similar to the one in MPI, where programmers have to specify on what will be run by each threads.

In UPC, communication among threads adopt the *Partitioned Global Address Space* (PGAS) paradigm by making use of *pointers*. There are three types of pointer commonly used in UPC [10]: (i) *private pointer* where the private pointers point to their own private address space, (ii) *private pointer-to-share* where the private pointers point to the shared address space, and (iii) *shared pointer-to-share* where the shared pointers from one address space point to the other shared address space.

UPC provides several synchronization mechanisms [9]. Table 3 briefly describes the synchronization mechanisms available in UPC together with the programming effort require by programmers.

Table 3: Synchronization Constructs in UPC

Mechanism	**Syntax**	**Implicit/ Explicit**	**Description**
Barrier	`upc_barrier expression`	Implicit	A blocking synchronization, similar to `MPI_Barrier` on MPI
Split phase barriers	`upc_notify expression` `upc_wait expression`	Implicit	A non-blocking synchronization
Fence	`upc_fence`	Implicit	Ensure that all shared references is completed before `upc_fence`
Locks	`upc_lock()` `upc_unlock()`	Explicit	Protect the shared data against multiple processors
Memory consistency control	`#include <upc_strict.h>` `#include <upc_relaxed.h>`	Explicit	There are two memory consistency models: 1. *Relaxed* Shared data can be reordered during compile time or runtime 2. *Strict* Accesses to shared data is serialized

3.6 Fortress

Fortress is a specification programming language designed for High Performance Computing [11]. The unit of worker is threads. Worker management, workload partitioning and worker mapping in Fortress can be implicit or explicit. In the implicitly approach, the iterative `for loops` is parallel by default. Programmer does not have to specify which threads to be run on each iteration. In the explicit approach, the creation of a thread can be done by using spawn keyword. As an example, in `t = spawn Global_Function(Arguments)`, `t` denotes the thread created, and `Global_Function` denotes the tasks to be run by `t`. Note that apart from a global function call, a task can be an expression as well. Stopping thread `t` is achieved through `t.stop()`. The workload partitioning and worker mapping for explicitly spawned threads are similar with in CUDA. One needs to decide how a workload is partitioned into tasks and how tasks are assigned to threads.

To avoid abnormal behavior and data races in one program, programmers have to specify the synchronization constructs explicitly. There are two synchronization constructs called reductions and `atomic` expression. The use of reduction is to avoid the need for synchronization by performing a computation as local as possible. Second construct, atomic can be used to control the data among the parallel executions. `Atomic` expression consists of atomic keyword follow by body expression. In body expression, all data reads and writes will appear to occur simultaneously in a single atomic step [11].

4 Summary

Using seven criteria, we have reviewed the qualitative aspects of six representative parallel programming models. Our goal is to provide a basic guideline in evaluating the appropriateness of a programming model in various development environments. The system-architecture aspect indicates the type of computing infrastructure (e.g. single node versus a cluster) supported by each of the programming models. The remaining aspects, which complement the typical performance evaluation, are meant to aid users in evaluating the ease-of-use of models. It should be noted that the seven criteria are by no means exhaustive. Other implementation issues such as debugging support should be considered as well when evaluating a parallel programming models.

References

1. Kish, L.B.: End of Mooreś Law: Thermal (noise) Death of Integration in Micro and nano electronics. Physics Letters A **305** (Dec 2002) 144-149
2. Kish, L.B.: Mooreś Law and the Energy Requirement of Computing Versus performance. circuits, devices and systems. **151**(2) (Apr 2004) 190-194
3. Sun Studio 12. `http://developers.sun.com/sunstudio`

4. Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., Yelick, K.A.: The Landscape of Parallel Computing Research: a view from Berkeley. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California at Berkeley (Dec 2006)
5. Butenhof, D.R.: Programming with POSIX Threads. Addison-Wesley, pub-AW:adr (1997)
6. OpenMP. http://www.openmp.org
7. Chapman, B., Jost, G., Van Der Pas, R.: Using OpenMP: Portable Shared Memory Parallel Programming. MIT Press (2007)
8. Pacheco, P.S.: Parallel Programming with MPI. Morgan Kaufmann (1996)
9. Consortium, U.: UPC Language Specifications, v1.2. Technical report (2005)
10. Husbands, P., Iancu, C., Yelick, K.: A Performance Analysis of the Berkeley UPC Compiler. In: ICS '03: Proceedings of the 17th annual international conference on Supercomputing, New York, NY, USA, ACM (2003) 63–73
11. Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J.W., Ryu, S., Jr., G.L.S., Tobin-Hochstadt", S.: The Fortress Language Specification Version 1.0 beta. Technical report (mar 2007)
12. Corporation", N.: NVIDIA CUDA Programming Guide, version 1.1. Technical report (nov 2007)
13. Grama, A., Karypis, G., Kumar, V., Gupta, A.: Introduction to Parallel Computing 2nd Edition. Addison-Wesley, Boston, MA, USA (jan 2003)
14. OpenMPI. http://www.open-mpi.org
15. MVAPICH. http://mvapich.cse.ohio-state.edu
16. MPICH. http://www.mcs.anl.gov/research/projects/mpich2
17. GRIDMPI. http://www.gridmpi.org
18. LAM/MPI. http://www.lam-mpi.org