SCS5108 Parallel Systems

Unit II

Parallel Programming

Programming parallel machines is notoriously difficult. Factors contributing to this difficulty include the complexity of concurrency, the effect of resource allocation on performance and the current diversity of parallel machine models. The net result is that effective portability, which depends crucially on the predictability of performance, has been lost. Functional programming languages have been put forward as solutions to these problems, because of the availability of implicit parallelism. However, performance will be generally poor unless the issue of resource allocation is addressed explicitly, diminishing the advantage of using a functional language in the first place.

We present a methodology which is a compromise between the extremes of explicit imperative programming and implicit functional programming. We use a repertoire of higher-order parallel forms, *skeletons*, as the basic building blocks for parallel implementations and provide program transformations which can convert between skeletons, giving portability between differing machines. Resource allocation issues are documented for each skeleton/machine pair and are addressed explicitly during implementation in an interactive, selective manner, rather than by explicit programming.

Threads

A **thread** of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system.[1] The implementation of threads and processes differs between operating systems, but in most cases a thread is a component of a process. Multiple threads can exist within the same process, executing concurrently (one starting before others finish) and share resources such as memory, while different processes do not share these resources. In particular, the threads of a process share its instructions (executable code) and its context (the values of its variables at any given moment).

On a single processor, multithreading is generally implemented by time slicing (as in multitasking), and the central processing unit (CPU) switches between different software threads. This context switching generally happens frequently enough that the user perceives the threads or tasks as running at the same time (in parallel). On a multiprocessor or multi-core system, multiple threads can be executed in parallel (at the same instant), with every processor or core executing a separate thread simultaneously; on a processor or core with hardware threads, separate software threads can also be executed concurrently by separate hardware threads.

Threads made an early appearance in OS/360 Multiprogramming with a Variable Number of Tasks (MVT) in 1967, in which they were called "tasks". Process schedulers of many modern operating systems directly support both time-sliced and multiprocessor threading, and the operating system kernel allows programmers to manipulate threads by exposing required functionality through the system call interface. Some threading implementations are called kernel threads, whereas lightweight processes (LWP) are a specific type of kernel thread that share the same state and information. Furthermore, programs can have user-space threads

when threading with timers, signals, or other methods to interrupt their own execution, performing a sort of ad hoc time-slicing.

Message passing

Message passing is a form of communication used in parallel programming and object-oriented programming. Communications are completed by the sending of messages (functions, signals and data packets) to recipients. Message Passing Interface (MPI) is a standardized and portable message-passing system designed by a group of researchers from academia and industry to function on a wide variety of parallel computers

MPI

1)Developed by MPI forum (made up of Industry, Academia and Govt.)
2)They established a standardised Message-Passing Interface (MPI-1) in 1994
3)It was intended as an interface to both C and FORTRAN.
4)C++ bindings were deprecated in MPI-2. Some Java bindings exist but are not standard yet.
5) Aim was to provide a specification which can be implemented on any parallel computer or cluster; hence portability of code was a big aim.

Advantages of MPI
1)Portable, hence protection of software investment
2)A standard, agreed by everybody
3)Designed using optimal features of existing message-passing libraries
4)"Kitchen-sink" functionality, very rich environment (129 functions)
5)Implementations for F77, C and C++ are freely downloadable

Disadvantages of MPI

1)"Kitchen-sink" functionality, makes it hard to learn all (unnecessary: a bare dozen are needed in most cases)
2) Implementations on shared-memory machines is often quite poor, and does not suit the programming model
3) Has rivals in other message-passing libraries (e.g. PVM)

Features of MPI

MPI provides support for:

1) Point-to-point & collective (i.e. group) communications
2) Inquiry routines to query the environment (how many nodes are there, which node number am I, etc.)
3) Constants and data-types
4) All MPI identifiers are prefixed by 'MPI_'.
5) C routines contain lower case (i.e. 'MPI_Init'),
6) Constants are all in upper case (e.g. 'MPI_FLOAT ' is an MPI C data-type).
7) C routines are actually integer functions which return a status
8) code (you are strongly advised to check these for errors!).
9) Number of processors used is specified in the command line,
10) when running the MPI loader that loads the MPI program onto the processors, to avoid hard-coding this into the program e.g. mpirun -np N exec

Scalability and Portability

In heterogeneous computing systems, computing nodes might be different whereas each computing node contains different system resources such as processors, graphics processing units, memories, networks, storage units, etc. These computing nodes and their internal resources have to collaborate well to provide required computing capacity.

Portability is always an issue in heterogeneous computing systems. Data generated on one machine might not be able to be used by others directly because of the incompatibility issues. Data type, endianness, size and padding situation are different in heterogeneous and even in homogeneous computing systems. Data marshaling procedure is indispensable, especially in open systems. In this thesis, a portable data exchange toolkit is proposed.

Meanwhile, scalability is another major issue for Grand-Challenge applications. How to utilize system resources efficiently is critical when problem size increases. Based on system resources' different features, workload should be scheduled properly among them. In this thesis, a novel GPU-based MD5-Blowfish encryption algorithm is designed and implemented to handle scaled data with some optimization features of NVIDIA Fermi architecture.

Transactional Memory

Transactional memory attempts to simplify concurrent programming by allowing a group of load and store instructions to execute in an atomic way. It is a concurrency control mechanism analogous to database transactions for controlling access to shared memory in concurrent computing.

The motivation of transactional memory lies in the programming interface of parallel programs. The goal of a transactional memory system is to transparently support the definition of regions of code that are considered a transaction, that is, that have atomicity, consistency and isolation requirements. Transactional memory allows writing code like this example:

```
def transfer_money(from_account, to_account, amount):
    with transaction():
        from_account -= amount
        to_account += amount
```

In the code, the block defined by "transaction" has the atomicity, consistency and isolation guarantees and the underlying transactional memory implementation must assure those guarantees transparently.

Hardware transactional memory systems may comprise modifications in processors, cache and bus protocol to support transactions.Load-link/store-conditional (LL/SC) offered by many RISC processors can be viewed as the most basic transactional memory support; however, LL/SC usually operates on data that is the size of a native machine word, so only single-word transactions are supported.

Software transactional memory provides transactional memory semantics in a software runtime library or the programming language,[6] and requires minimal hardware support (typically an atomic compare and swap operation, or equivalent). As the downside, software implementations usually come with a performance penalty, when compared to hardware solutions.

Owing to the more limited nature of hardware transactional memory (in current implementations), software using it may require fairly extensive tuning to fully benefit from it. For example, the dynamic memory allocator may have a significant influence on performance and likewise structure padding may affect performance (owing to cache alignment and false sharing issues); in the context of a virtual machine, various background threads may cause unexpected transaction aborts

ZPL

ZPL (short for *Z-level Programming Language*) is an array programming language designed to replace C and C++ programming languages in engineering and scientific applications. Because its design goal was to obtain cross-platform high performance, ZPL programs run fast on both sequential and parallel computers. Highly-parallel ZPL programs are simple and easy to write because it exclusively uses implicit parallelism.

Originally called *Orca C*, ZPL was designed and implemented during 1993-1995 by the Orca Project of the Computer Science and Engineering Department at the University of Washington.

ZPL uses the array abstraction to implement a data parallel programming model. This is the reason why ZPL achieves such good performance: having no parallel directives or other forms of explicit parallelism, ZPL exploits the operational trait that when aggregate computations are described in terms of arrays, many scalar operations must be (implicitly) performed to implement the array operations. This *implied* computation can be automatically allotted to different processors to achieve concurrency: Parallelism arises from the semantics of the array operations.

ZPL is translated into a conventional abstract syntax tree representation on which program analysis and program optimizations are performed. ANSI C code is generated as the object code. This C program (which is machine independent because it implements certain operations in abstract form) is then compiled using the native C compiler on the target machine with custom libraries optimized to the specific platform.

The creators of ZPL were: Brad Chamberlain, Sung-Eun Choi, E Christopher Lewis, Calvin Lin, Jason Secosky, Larry Snyder, and W. Derrick Weathersby with assistance from Ruth Anderson, A.J. Bernheim, Marios Dikaiakos, George Forman, and Kurt Partridge.

Automatic parallelization, also **auto parallelization**, **autoparallelization**, or **parallelization**, the last one of which implies automation when used in context, refers to converting sequential code into multi-threaded or vectorized (or even both) code in order to utilize multiple processors simultaneously in a shared-memory multiprocessor (SMP) machine. The goal of automatic parallelization is to relieve programmers from the tedious and error-prone manual parallelization process. Though the quality of automatic parallelization has improved in the past several decades, fully automatic parallelization of sequential programs by compilers remains a grand challenge due to its need for complex program analysis and the unknown factors (such as input data range) during compilation.

The programming control structures on which autoparallelization places the most focus are loops, because, in general, most of the execution time of a program takes place inside some form of loop. There are two main approaches to parallelization of loops: pipelined multi-threading and cyclic multi-threading.

For example, consider a loop that on each iteration applies a hundred operations, runs for a thousand iterations. This can be thought of as a grid of 100 columns by 1000 rows, a total of 100,000 operations. Cyclic multi-threading assigns each row to a different thread. Pipelined multi-threading assigns each column to a different thread.

Chapel

Parallel computing has resulted in numerous significant advances in science and technology over the past several decades. However, in spite of these successes, the fact remains that only a small fraction of the world's programmers are capable of effectively using the parallel languages and programming models employed within HPC and mainstream computing. Chapel is an emerging parallel language being developed at Cray Inc. with the goal of addressing this issue and making parallel programming far more productive and generally accessible.

Chapel originated from the DARPA High Productivity Computing Systems (HPCS) program, which challenged vendors like Cray to improve the productivity of high-end computing systems. Engineers at Cray noted that the HPC community was hungry for alternative parallel programming languages and developed Chapel as part of our response. The reaction

from HPC users so far has been very encouraging—most would be excited to have the opportunity to use Chapel once it becomes production-grade.

Chapel Overview

Though it would be impossible to give a thorough introduction to Chapel in the space of this article, the following characterizations of the language should serve to give an idea of what we are pursuing:

- **General Parallelism:** Chapel has the goal of supporting any parallel algorithm you can conceive of on any parallel hardware you want to target. In particular, you should never hit a point where you think "Well, that was fun while it lasted, but now that I want to do *x*, I'd better go back to MPI."
- **Separation of Parallelism and Locality:** Chapel supports distinct concepts for describing parallelism ("These things should run concurrently") from locality ("This should be placed here; that should be placed over there"). This is in sharp contrast to conventional approaches that either conflate the two concepts or ignore locality altogether.
- **Multiresolution Design:** Chapel is designed to support programming at higher or lower levels, as required by the programmer. Moreover, higher-level features—like data distributions or parallel loop schedules—may be specified by advanced programmers within the language.
- **Productivity Features:** In addition to all of its features designed for supercomputers, Chapel also includes a number of sequential language features designed for productive programming. Examples include type inference, iterator functions, object-oriented programming, and a rich set of array types. The result combines productivity features as in Python™, Matlab®, or Java™ software with optimization opportunities as in Fortran or C.

Chapel's implementation is also worth characterizing:

- **Open Source:** Since its outset, Chapel has been developed in an open-source manner, with collaboration from academics, computing labs, and industry. Chapel is released under a BSD license in order to minimize barriers to its use.
- **Portable:** While Cray machines are an obvious target for Chapel, the language was designed to be very portable. Today, Chapel runs on virtually any architecture supporting a C compiler, UNIX-like environment, POSIX threads, and MPI or UDP.
- **Optimized for Crays:** Though designed for portability, the Chapel implementation has also been optimized to take advantage of Cray-specific features.

Chapel: Today and Tomorrow

While the HPCS project that spawned Chapel concluded successfully at the end of 2012, the Chapel project remains active and ongoing. The Chapel prototype and demonstrations developed under HPCS were considered compelling enough to users that Cray plans to continue the project over the next several years. Current priorities include:

- **Performance Optimizations:** To date, the implementation effort has focused primarily on correctness over performance. Improving performance is typically considered the number one priority for growing the Chapel community.
- **Support for Accelerators:** Emerging compute nodes are increasingly likely to contain accelerators like GPUs or Intel® MIC chips. We are currently working on extending our locality abstractions to better handle such architectures.
- **Interoperability:** Beefing up Chapel's current interoperability features is a priority, to permit users to reuse existing libraries or gradually transition applications to Chapel.
- **Feature Improvements:** Having completed HPCS, we now have the opportunity to go back and refine features that have not received sufficient attention to date. In many cases, these improvements have been motivated by feedback from early users.
- **Outreach and Evangelism:** While improving Chapel, we are seeking out ways to grow Chapel's user base, particularly outside of the traditional HPC sphere.
- **Research Efforts:** In addition to hardening the implementation, a number of interesting research directions remain for Chapel, including resilience mechanisms, applicability to "big data" computations, energy-aware computing, and support for domain specific languages.

Map Reduce

MapReduce is a programming model and an associated implementation for processing and generating large data sets with a parallel, distributed algorithm on a cluster. Conceptually similar approaches have been very well known since 1995 with the Message Passing Interface standard having reduce and scatter operations.

A MapReduce program is composed of a **Map()** procedure (method) that performs filtering and sorting (such as sorting students by first name into queues, one queue for each name) and a **Reduce()** method that performs a summary operation (such as counting the number of students in each queue, yielding name frequencies). The "MapReduce System" (also called "infrastructure" or "framework") orchestrates the processing by marshalling the distributed servers, running the various tasks in parallel, managing all communications and data transfers between the various parts of the system, and providing for redundancy and fault tolerance.

The model is inspired by the map and reduce functions commonly used in functional programming, although their purpose in the MapReduce framework is not the same as in their original forms. The key contributions of the MapReduce framework are not the actual map and reduce functions, but the scalability and fault-tolerance achieved for a variety of applications by optimizing the execution engine once. As such, a single-threaded implementation of MapReduce will usually not be faster than a traditional (non-MapReduce) implementation, any gains are usually only seen with multi-threaded implementations. The use of this model is beneficial only when the optimized distributed shuffle operation (which reduces network communication cost) and fault tolerance features of the MapReduce framework come into play. Optimizing the communication cost is essential to a good MapReduce algorithm.

MapReduce libraries have been written in many programming languages, with different levels of optimization. A popular open-source implementation that has support for distributed shuffles is part of Apache Hadoop. The name MapReduce originally referred to the proprietary Google technology, but has since been genericized. By 2014, Google were no longer using MapReduce as a Big Data processing model, and development on Apache Mahout had moved on to more capable and less disk-oriented mechanisms that incorporated full map and reduce capabilities

MapReduce is a framework for parallel computing. Programmers get a simple API and do not have to deal with issues of parallelization, remote execution, data distribution, load balancing, or fault tolerance. The framework makes it easy for one to use thousands of processors to process huge amounts of data (e.g., terabytes and petabytes).

From a user's perspective, there are two basic operations in MapReduce: *Map* and *Reduce*.

The **Map** function reads a stream of data and parses it into intermediate *(key, value)* pairs. When that is complete, the **Reduce** function is called once for each unique key that was generated by *Map* and is given the key and a list of all values that were generated for that key as a parameter. The keys are presented in sorted order.

As an example of using MapReduce, consider the task of counting the number of occurrences of each word in a large collection of documents. The user-written *Map* function reads the document data and parses out the words. For each word, it writes the (key, value) pair of (word, 1). That is, the word is treated as the key and the associated value of 1 means that we saw the word once. This intermediate data is then sorted by MapReduce by keys and the user's *Reduce* function is called for each unique key. Since the only values are the count of 1, *Reduce* is called with a list of a "1" for each occurence of the word that was parsed from the document. The function simply adds them up to generate a total word count for that word. Here's what the code looks like:

```
map(String key, String value):
// key: document name, value:
document contents for each word w in value:
EmitIntermediate(w, "1");
reduce(String key, Iterator values):
 // key: a word;
values: a list of counts
int result = 0;
 for each v in values: result += ParseInt(v);
 Emit(AsString(result));
```

Detailed Explanation of Map Reduce:

MapReduce is largely seen as an API: communication with the various machines that play a part in execution is hidden. MapReduce is implemented in a master/worker configuration, with one master serving as the coordinator of many workers. A worker may be assigned a role of either a map worker or a reduce worker.

Step 1. Split input

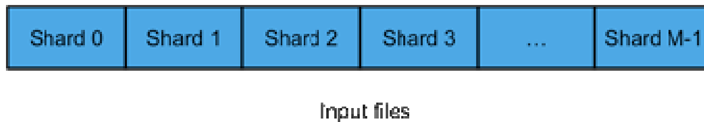

Figure 1. Split input into shards

The first step, and the key to massive parallelization in the next step, is to split the input into multiple pieces. Each piece is called a split, or shard. For M map workers, we want to have M shards, so that each worker will have something to work on. The number of workers is mostly a function of the amount of machines we have at our disposal.

The MapReduce library of the user program performs this split. The actual form of the split may be specific to the location and form of the data. MapReduce allows the use of custom readers to split a collection of inputs into shards, based on specific format of the files.

Step 2. Fork processes

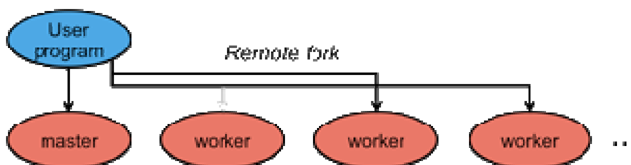

Figure 2. Remotely execute worker processes

The next step is to create the master and the workers. The master is responsible for dispatching jobs to workers, keeping track of progress, and returning results. The master picks idle workers and assigns them either a map task or a reduce task. A map task works on a single shard of the original data. A reduce task works on intermediate data generated by the map tasks. In all, there will be M map tasks and R reduce tasks. The number of reduce tasks is the number of partitions defined by the user. A worker is sent a message by the master identifying the program (map or reduce) it has to load and the data it has to read.

Step 3. Map

Figure 3. Map task

Each map task reads from the input shard that is assigned to it. It parses the data and generates (key, value) pairs for data of interest. In parsing the input, the map function is likely to get rid of a lot of data that is of no interest. By having many map workers do this in parallel, we can linearly scale the performance of the task of extracting data.

Step 4: Map worker: Partition

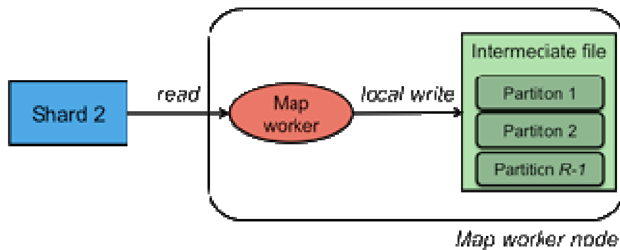

Figure 4. Create intermediate files

The stream of (key, value) pairs that each worker generates is buffered in memory and periodically stored on the local disk of the map worker. This data is partitioned into R regions by a partitioning function.

The partitioning function is responsible for deciding which of the R reduce workers will work on a specific key. The default partitioning function is simply a hash of key modulo R but a user can replace this with a custom partition function if there is a need to have certain keys processed by a specific reduce worker.

Step 5: Reduce: Sort (Shuffle)

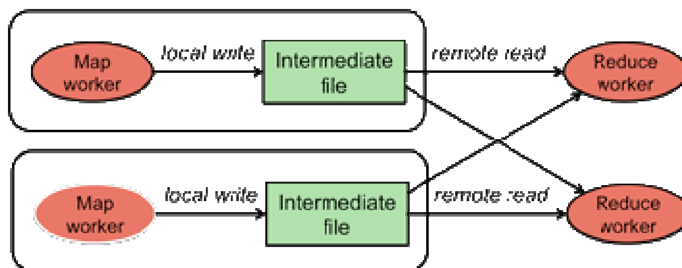

Figure 5. Sort and merge partitioned data

When all the map workers have completed their work, the master notifies the reduce workers to start working. The first thing a reduce worker needs to is to get the data that it needs to present to the user's reduce function. The reduce worker contacts every map worker via remote procedure calls to get the (key, value) data that was targeted for its partition. This data is then sorted by the keys. Sorting is needed since it will usually be the case that there are many occurrences of the same key and many keys will map to the same reduce worker (same partition). After sorting, all occurrences of the same key are grouped together so that it is easy to grab all the data that is associated with a single key.

This phase is sometimes called the shuffle phase.

Step 6: Reduce function

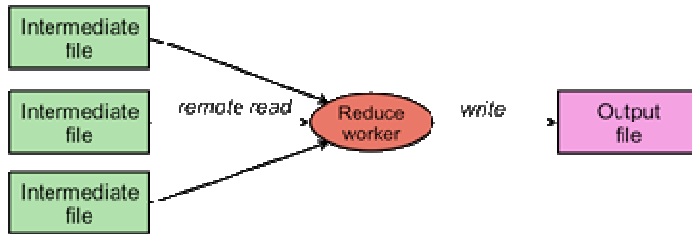

Figure 6. Reduce function writes output

With data sorted by keys, the user's Reduce function can now be called. The reduce worker calls the Reduce function once for each unique key. The function is passed two parameters: the key and the list of intermediate values that are associated with the key.

The Reduce function writes output sent to file.

Step 7: Done!

When all the reduce workers have completed execution, the master passes control back to the user program. Output of MapReduce is stored in the R output files that the R reduce workers created.

The big picture

Figure 7 illustrates the entire MapReduce process. The client library initializes the shards and creates map workers, reduce workers, and a master. Map workers are assigned a shard to process. If there are more shards than map workers, a map worker will be assigned another shard when it is done. Map workers invoke the user's Map function to parse the data and write intermediate (key, value) results onto their local disks. This intermediate data is partitioned into R partitions according to a partioning function. Each of R reduce workers contacts all of the map workers and gets the set of (key, value) intermediate data that was targeted to its partition. It then calls the user's Reduce function once for each unique key and gives it a list of all values that were generated for that key. The Reduce function writes its final output to a file that the user's program can access once MapReduce has completed.

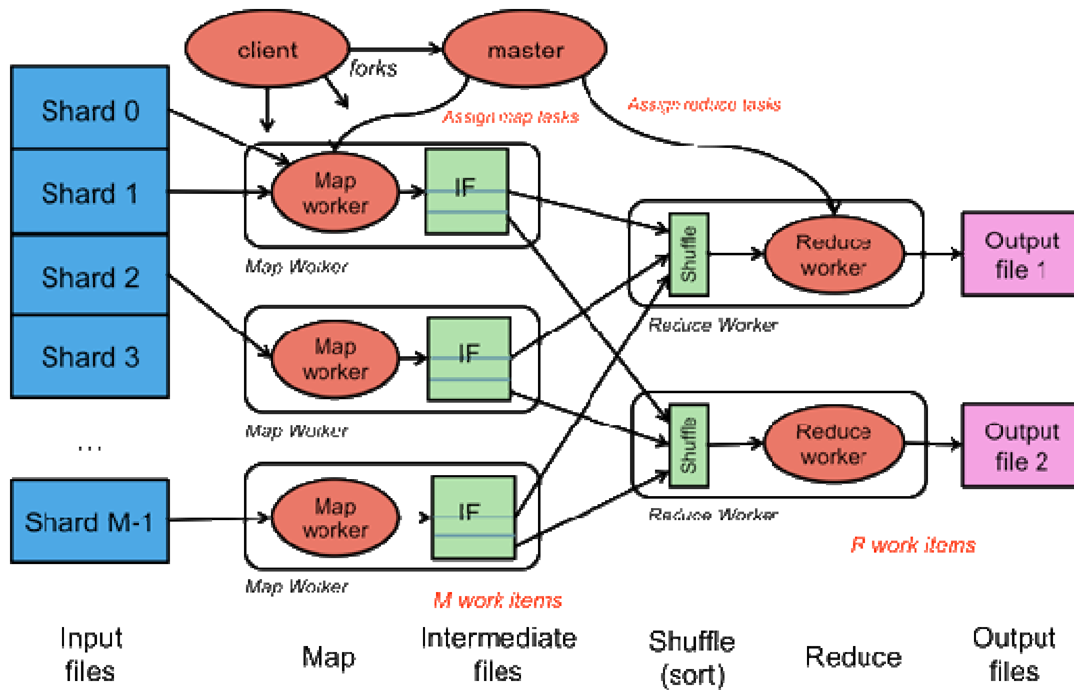

Figure 7. MapReduce

References:

1. https://en.wikipedia.org/wiki/MapReduce#Implementations_of_MapReduce
2. https://en.wikipedia.org/wiki/Chapel_(programming_language)
3. http://www.cray.com/blog/chapel-productive-parallel-programming/
4. https://en.wikipedia.org/wiki/Message_Passing_Interface