

Restricted Programming Paradigms for Parallelism

David McClosky
dmcc@cs.brown.edu

11th May 2006

1 Introduction

Large data sets (for example, snapshots of the web or radio telescope data) and expensive computations (molecular simulations, numerical optimization) typically exceed the capabilities of single processor systems. These problems demand techniques for distributing the load across processors and machines. The slowing increases of clock speeds [9] and the growing availability of multicore processors amplify this need. Sutter and Larus [10] point out that the difficulty in parallelizing programs stems from the amount of synchronization required. Synchronization is needed when data is shared between tasks in conflicting access modes, for example, whenever data may be left in an inconsistent state by one thread and read by another.

However, while some problems require significant coordination, there is a large class of problems that require very little coordination and can be parallelized fairly easily. Sutter and Larus describe three categories of parallelism: independent, regular, and unstructured. Independent or “embarrassingly parallel” tasks involve subtasks which are entirely independent of each other. The authors give the example of scalar multiplication of an array: the array can be safely partitioned among available processors without any synchronization. Regular parallelism is found when tasks contain some easily predictable dependencies, for example, parallel reads or predictable local modifications of a data set. This is found in more complex array operations such as stencil computations or on web servers where all threads read from a common database. In unstructured parallelism, the interactions are not known in advance. In this case, one must make use of techniques built around locking or compare-and-swap operations to protect from inconsistent states. An appealing method is transactional memory [7, 8] which requires bracketing critical sections with transactions.

In this paper, we examine the MapReduce [4] programming paradigm which encompasses a large number of common tasks that require minimal synchronization. The tasks require the programmer to specify *map* and *reduce* functions. Given these functions, the result of applying MapReduce to a data set D can be expressed as

$$reduce(\lambda_r, map(\lambda_m, D))$$

As in functional programming, *map* returns a list of the map function (λ_m) applied to all elements in the list and *reduce* repeatedly applies its function (λ_r) to pairs of elements in its list, eventually returning a single value. For example, if the map function returned the count of some event in the data and the reduce function was the addition operator, the MapReduce procedure would return the total number of times the event occurred in the entire data set.

This construct lends itself naturally to parallelism since the map function computation of each element is necessarily independent. Additionally, the reduce step can be done incrementally in any order, assuming a proper reduce function (reduce functions must be commutative and idempotent). Thus, MapReduce can address problems in the category of regular parallelism.

1.1 Roadmap

Before delving into the details of MapReduce, we will cover some of the other programming paradigms for parallelism. We will then describe the MapReduce implementation and some extensions to the basic model. Finally, we will mention Hogwash, a simplified implementation of MapReduce in Python.

2 Related work

MapReduce is not the first parallel system to propose a new paradigm. Each parallel system handles different sets of problems well with varying amounts of work to adapt to its paradigm. We examine four systems here, ordered from most general to most restricted models.

2.1 Message Passing Interface and Bulk Synchronous Programming

Message Passing Interface [6] and Bulk Synchronous Programming [11] are built on very simple concurrency primitives. Essentially, the main primitives allow one to send messages to other processes and broadcast messages to all processes. As these are very general, MPI and BSP could be used to implement most of the other models discussed in this paper. Both systems include more complex operations built from the primitives to automate some higher-level tasks. For example, both support the notion of performing the same operation in parallel, possibly on portions of an array or vector, and aggregating the results (e.g. by summing them, applying a reduction operator, etc.). Both support basic synchronization strategies such as barriers. The main difference between the two models is that BSP has the concept of a superstep — a barrier indicating the end of a certain phase of the parallel algorithm. A BSP program consists of a set of supersteps, where all operations within a superstep should be safely be executable in parallel. The MPI model does not have this restriction on control flow.

While the generality of these models allows them to do many things that restricted models cannot, their code is harder to write, maintain, and analyze. Thus, other approaches limit their model's power, sacrificing generality for usability.

2.2 NESL

Blelloch describes NESL [3], which provides vectors and functional parallel operators over vectors. Like MPI and BSP, NESL provides a mechanism for quickly aggregating operations on elements of a list. However, unlike the previous two systems, NESL abstracts away the explicit communication. NESL appears to be general enough to incorporate MapReduce, though NESL is harder to implement due to its extra power.

2.3 River

As the name implies, River [1] is stream based. Each process in a River network has incoming and outgoing streams which connect to other processes. This system produces a generalized pipeline, where some processes serve to generate data (by reading it from disk, etc.), some perform transformations on the data, and others write the processed data back to disk. While River could be used to implement the MapReduce paradigm, River does not easily allow a scheduler to assign an arbitrary number of machines to the task.

River's contribution is a technique called *graduated declustering* which acknowledges that streams do not always flow the same rate. If one process is not producing as much as other input processes (due to a slow disk, for example), the remaining input processes will attempt to compensate by taking on more work.

2.4 BAD-FS

The Batch-Aware Distributed Filesystem (BAD-FS) [2] approaches the problem from a different angle. It attempts to optimize performance (which for the tasks examined are assumed to be I/O bound) through knowledge of the overall batch workload. The goals are to minimize unnecessary data movement while still providing fault tolerance at a filesystem level. As result, BAD-FS uses a simple, easily analyzable model where the user selects multiple input data sets and a sequence of operations to apply to each of them. In a sense, this is similar to the map operation. The reduce operation as well as other more complex constructions are not supported by this model.

3 MapReduce

Dean and Ghemawat present an implementation of MapReduce [4] used at Google. Their model is slightly different than the purely functional model presented in the introduction. Their map function is required to operate on key-value pairs, where keys and values are strings, and outputs intermediate key-value pairs. The MapReduce library groups together intermediate key-value pairs with the same key, producing a key-value pair with the same key in the list of all the values. The reduce function is passed a dictionary with keys and a list of values with each key. This implementation has the down side that other data types must be converted to and from strings.¹

3.1 Example

The authors provide several motivating examples such as distributed grep, URL and word counting, reverse web link graph generation, and distributed sort. Their pseudocode for obtaining the count of each word is shown below:

```
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));
```

The map function splits the contents of the document into words and converts each word into a key-value pair with the word as the key and one as the value. The MapReduce library groups the keys together and produces key-value pairs where words are keys and values are lists of counts. The reduce function sums these values and returns the total count of each word. For example, the results of this process on the first sentence of this paragraph is shown in Table 1.

¹Hogwash has a slightly more relaxed restriction that all arguments and results must be “pickleable,” Python’s automatic serialization system. As almost all objects implement this, the programmer is generally able ignore these issues.

Key	Value	Key	Value
The	{1}	key-value	{1}
a	{1}	map	{1}
and	{1, 1}	of	{1}
as	{1, 1}	one	{1}
contents	{1}	pair	{1}
converts	{1}	splits	{1}
document	{1}	the	{1, 1, 1, 1, 1}
each	{1}	value.	{1}
function	{1}	with	{1}
into	{1, 1}	word	{1, 1}
key	{1}	words	{1}

(a) Results after map step and key grouping

Key	Value	Key	Value
The	1	key-value	1
a	1	map	1
and	2	of	1
as	2	one	1
contents	1	pair	1
converts	1	splits	1
document	1	the	5
each	1	value.	1
function	1	with	1
into	2	word	2
key	1	words	1

(b) Results after reduce step

Table 1: Partial and final results of the MapReduce operation

3.2 Implementation

MapReduce follows the master-worker paradigm of job control. In this paradigm one machine directs the rest of the machines (workers) in their computation. The master machine knows the general state of the computation, i.e. which parts have already been completed and which parts are in progress. Google has a large number of commodity machines which makes this paradigm reasonable. The computational load for the master in this case is not high enough to require its distribution across machines.

As the computation has two phases, the master must first decide which step a worker will take part in. Next, once assignments have been made, the master connects map tasks with reduced tasks. The map tasks store their intermediate results locally and report this location to the master who in turn passes the locations to reduce tasks. The reduce tasks operate on the intermediate files, first sorting by key to make it easy to group by key. After processing their input, they append the results to the master output file, making use of the atomic append operation in the Google Filesystem[5].

To allow for parallel processing, the input data and the intermediate data from the map step are divided into partitions. The programmer specifies the number of input and reduce partitions. Reduce partitions are formed by hashing the keys, e.g. $hash(key) \bmod R$. Naturally, if $R = 1$, we obtain the expected result from functional programming (a single value) but cannot parallelize the reduction task. Thus, R is typically much larger than one, for example 5,000, resulting in R partial reduction files, each a reduction of $1/R$ of the input. To obtain a single data file at the end, one can perform an additional reduce task. One gets the impression that this is typically not done at Google since ultimately it is desired to have large files (such as indices) spread across many machines. MapReduce produces these for free.

As is commonly said by Google employees, when working with large amounts of machines, failures are the norm rather than the exception. Thus, MapReduce must tolerate unreliable workers. The master periodically checks in with each worker. If the worker is not responsive, the master selects new workers to restart the tasks performed by the failed worker. Finished reduce tasks are stored on the distributed filesystem (not locally) and do not need to be reexecuted. In evaluating this, the authors show how the master is quick to notice failed workers and restart their jobs elsewhere. The MapReduce implementation assumes that the master is unlikely to fail, but the authors note that its state could be checkpointed periodically if this were a concern.

3.3 Extensions

The others present several interesting extensions to MapReduce, which tend to focus on optimizations, debugging support, and general ease-of-use.

3.3.1 Enhancements to the Reduce Phase

Several of the refinements work towards adding functionality to the reduce phase. For the reduce phase, the user may specify the function that hashes the keys to determine their partition. This gives the programmer control over which keys will be found together in the same output file. Additionally, MapReduce guarantees that the intermediate key-value pairs are processed in sorted key order, allowing for fast lookups on the resulting file. Finally, to save network bandwidth (which for many tasks involving large data sets is the scarcest resource) the programmer can specify a combiner function which performs partial reductions of the keys.

3.3.2 Debugging Support

Since programmers cannot always anticipate how a distributed job will turn out, it is important to know if and how things are going wrong. MapReduce installs a signal handler to each job. If a job is killed by a signal, MapReduce notes which record was being processed at the time. MapReduce will retry computation on that record, but if another failure is detected, the record will be skipped. MapReduce also includes a mode where execution is only done locally to test or refine the task being developed. The status of a running MapReduce job can be viewed on a local web server. The web server indicates the number of machines in each phase of the processing as well as which tasks have failed.

3.3.3 Input/Output

As data is not always stored on disk as key-value pairs, MapReduce includes many readers transform different types of files and from these pairs. The authors give several examples. One reader takes a file and returns pairs where the key is the offset of a line in the value is the line itself. Another reader accepts web page access logs. In general, the readers are allowed to perform preprocessing of the data to make it easier for the map step to process it.

4 Hogwash

Hogwash stemmed from the desire to express transformations of data in Python functions and distribute the computation across the Brown Computer Science department's process distributor (Quahog). While Hogwash is based on many ideas from MapReduce, there are several differences. The most noticeable change is the lack of a reduce step. This is because of two reasons. First, we find that the reduce step is not necessary for the type of tasks that we typically do. Second, since Quahog does not have the same power as Google's clustering system, it is much harder to implement efficiently. Another large change from MapReduce is that the map function can be any Python function rather than specialized C++ classes. Hogwash map functions can accept any arguments and return (nearly — see footnote 1) any Python object. In practice, these arguments are often filenames, filenames with ranges of lines, or parameters to a model.

4.1 Dependencies

Hogwash also differs in that it supports dependencies. Any job may list other jobs as “parents” which must be executed before it can be run. The child job may also view results of the parent job and use them as

parameters, if desired. If a job is run which has unfulfilled dependencies, it will attempt to fulfill them itself if possible. If all unfulfilled dependencies are in progress or if any of them encountered errors, the job will place itself back on the queue.

4.2 Debugging Support

When using Quahog directly, the author saw firsthand how difficult it was to debug distributed programs. As a result, Hogwash takes several steps to provide status and debugging information. Since distributed programs are run with no terminal, Hogwash captures their standard out standard error streams and redirects the streams to a log file. Additionally, if the program throws an exception during its operation, Hogwash prints the stack trace to the log file and saves the exception as a result of the job. Finally, disruptive signals are caught and converted into exceptions and then raised, making use of the exception handler. When this happens, a job is notified that it should clean up and exit. As in MapReduce, jobs may be run locally for testing or selecting dedicated cycle machines. In fact, several users do not use the Quahog process distributor mode at all and simply use Hogwash as a means of performing a set of jobs on specific high-performance machines.²

4.3 Implementation

We define a “session” as a map function, a list of arguments, and the state of the computation (partial results, jobs in progress, etc.). Each argument becomes a numbered job (the number being its index in the list of arguments). A session is implemented as a directory on disk, containing subdirectories for job result and log files as well as files for job status and global state. Jobs exist in one of four states: queued, started, finished (completed successfully), and error (threw an exception). A *clear* operation is available to wipe out the results of a finished or error job, allowing it to be reexecuted. Certain exceptions, for example the response to the signal Quahog sends jobs when it needs to kill them, automatically clear jobs and return them to the queue. This exception can also be raised by the user which is a useful way of stating that the job cannot be run on a specific machine. In the past, when some machines had NFS mounting errors, this technique was used to retry jobs on different machines.

Hogwash contains Session and Job objects to allow the user to programmatically create sessions, inspect their state, or extend the model. These objects store their stage persistently on disk in the session’s directory. Job state is stored in a single shared file which is “tailed” by status programs. Jobs append entries to this file using NFS file locks and exponential back off. In practice, even if 200 jobs are active, the average wait time to append it is reasonable.

To make Hogwash easier to use for non-Python users and because there are a number of frequent tasks one might want to perform on a session, Hogwash includes a shell called Sty. Sty provides status information, statistics, and visualization of the results of jobs. It also includes facilities for basic job querying, where queries are Python expressions that can make use of job arguments, results, durations, dependencies, and state.

For users who do not want to use Python and merely desire to execute shell commands in a distributed fashion, there is a small program called Migrate. Migrate creates a Hogwash session where the function is effectively `os.system` (a function that runs its argument in a subshell and returns the exit code) and the arguments are a list of the commands desired, specified by a batch file.

²Jobs that require significant amounts of memory or computation time should be run locally since there is less chance that they will be scheduled on a machine with insufficient memory or interrupted by another user.

```

def wc(filename):
    lines, words, chars = 0, 0, 0
    f = file(filename, 'r')
    for line in f:
        line = line[:-1] # remove newline
        lines += 1
        words += len(line.split())
        chars += len(line)
    return lines, words, chars

# this next line is true when the script is run from the command line
if __name__ == "__main__":
    import sys, os
    from Hogwash import Session
    filenames = [os.path.abspath(f) for f in sys.argv[1:]]
    #           module      func  arguments  session name
    s = Session('wordcount', 'wc', filenames, name='hogcount')

```

Figure 1: File: `wordcount.py`, a simple Hogwash script

4.4 Example: Word Count

In Figure 1, we see the source code for a simple distributed word count application. The file is both the map function (`wc`) and the setup code (executed when `wordcount.py` is run by Python). It accepts a list of filenames, fully qualifies each path, and creates a session where `wc` is the map function and the filenames are arguments. Once created, the session can be controlled within the Sty shell.

5 Conclusion

MapReduce provides a restrictive programming environment which allows for a variety of tasks to be easily computed in parallel. It allows for simple parallel iteration over data (the map step) and parallel aggregation of results (to reduce step). Steps need to be taken to provide fault tolerance and load (CPU and network bandwidth) balancing. Since this restricted environment lends itself so well to parallelism, it is worth investigating other restricted paradigms which may be more general or handle some cases that MapReduce misses. Given that parallelism is difficult to reason about, simpler solutions may result in more stable code. Of course, these systems can only work on some of the simpler classes of parallelism and other attempts at automatic parallelization are still important.

References

- [1] Remzi H. Arpaci-Dusseau. Cluster I/O with River: Making the fast case common. In *Proceedings of the Annual Workshop on I/O in Parallel and Distributed Systems (IOPADS'99)*, 1999. UC, Berkeley.
- [2] John Bent, Douglas Thain, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny. Explicit control in the batch-aware distributed file system. In *NSDI*, pages 365–378. USENIX, 2004.
- [3] Guy E. Blelloch. Programming parallel algorithms. *CACM: Communications of the ACM*, 39, 1996.

- [4] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.
- [5] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 29–43, Bolton Landing, NY, USA, October 2003. ACM.
- [6] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 2nd edition, November 1999.
- [7] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, San Diego, California, May 17–19, 1993. ACM SIGARCH and IEEE Computer Society TCCA.
- [8] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC '95)*, pages 204–213, New York, August 1995. ACM.
- [9] Herb Sutter. The free lunch is over: a fundamental turn toward concurrency in software. *Dr. Dobbs's Journal*, 30(3), 2005.
- [10] Herb Sutter and James R. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, 2005.
- [11] Leslie G. Valiant. A bridging model for parallel computation. *CACM: Communications of the ACM*, 33, 1990.