L22: Parallel Programming Language Features (Chapel and MapReduce)

December 1, 2009

Administrative

- Schedule for the rest of the semester
  - "Midterm Quiz" = long homework
    - Handed out over the holiday (Tuesday, Dec. 1)
    - Return by Dec. 15
  - Projects
    - 1 page status report on Dec. 3
      - handin cs4961 pdesc <file, ascii or PDF ok>
    - Poster session dry run (to see material) Dec. 8
    - Poster details (next slide)
- Mailing list: cs4961@list.eng.utah.edu

Poster Details

- I am providing:
- Foam core, tape, push pins, easels
- Plan on 2ft by 3ft or so of material (9-12 slides)
- Content:
  - Problem description and why it is important
  - Parallelization challenges
  - Parallel Algorithm
  - How are two programming models combined?
  - Performance results (speedup over sequential)

Outline

- Global View Languages
- Chapel Programming Language
- Map-Reduce (popularized by Google)
- Reading: Ch. 8 and 9 in textbook
- Sources for today's lecture
  - Brad Chamberlain, Cray
  - John Gilbert, UCSB

Shifting Gears

- What are some important features of parallel programming languages (Ch. 9)?
  - Correctness
  - Performance
  - Scalability
  - Portability

And what about ease of programming?

Global View Versus Local View

- P-Independence
  - If and only if a program always produces the same output on the same input regardless of number or arrangement of processors
- Global view
  - A language construct that preserves P-independence
  - Example (today's lecture)
- Local view
  - Does not preserve P-independent program behavior
  - Example from previous lecture?

What is a PGAS Language?

- PGAS = Partitioned Global Address Space
  - Present a global address space to the application developer
  - May still run on a distributed memory architecture
  - Examples: Co-Array Fortran, Unified Parallel C
- Modern parallel programming languages present a global address space abstraction
  - Performance? Portability?
- A closer look at a NEW global view language, Chapel
  - From DARPA High Productivity Computing Systems program
  - Language design initiated around 2003
  - Also X10 (IBM) and Fortress (Sun)

Chapel Domain Example: Sparse Matrices

Recall sparse matrix-vector multiply computation from P1&P2

```
for (j=0; j<nr; j++) {
  for (k = rowstr[j]; k<rowstr[j+1]-1; k++)
    t[j] = t[j] + a[k] * x[colind[k]];
}
```

Chapel Formulation

Declare a dense domain for sparse matrix

```
const dnsDom = [1..n, 1..n];
```

Declare a sparse domain

```
var spsDom: sparse subdomain(dnsDom);
Var spsArr: [spsDom] real;
```

Now you need to initialize the spsDom. As an example,

```
spsDom = [(1,2),(2,3),(2,7),(3,6),(4,8),(5,9),(6,4),(9,8)];
```

Iterate over sparse domain:

```
forall (i,j) in spsDom
  result[i] = result[i] + spsArr(i,j) * input[j];
```

12/01/09

I. MapReduce

- What is MapReduce?
- Example computing environment
- How it works
- Fault Tolerance
- Debugging
- Performance

What is MapReduce?

- Parallel programming model meant for large clusters
  - User implements Map() and Reduce()
- Parallel computing framework
  - Libraries take care of EVERYTHING else
    - Parallelization
    - Fault Tolerance
    - Data Distribution
    - Load Balancing
- Useful model for many practical tasks (large data)

Map and Reduce

- Borrowed from functional programming languages (eg. Lisp)
- Map()
  - Process a key/value pair to generate intermediate key/value pairs
- Reduce()
  - Merge all intermediate values associated with the same key

Example: Counting Words

- Map()
  - Input <filename, file text>
  - Parses file and emits <word, count> pairs
    - eg. <"hello", 1>
- Reduce()
  - Sums values for the same key and emits <word, TotalCount>
    - eg. <"hello", (3 5 2 7)> => <"hello", 17>

Example Use of MapReduce

- Counting words in a large set of documents

```
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: word
    //values: list of counts
    int results = 0;
    for each v in values
        result += ParseInt(v);
    Emit(AsString(result));
```

THE UNIVERSITY OF UTAH

Google Computing Environment

- Typical Clusters contain 1000's of machines
- Dual-processor x86's running Linux with 2-4GB memory
- **Commodity networking**
  - **Typically 100 Mbs or 1 Gbs**
- IDE drives connected to individual machines
  - Distributed file system

THE UNIVERSITY OF UTAH

How MapReduce Works

- User to do list:
  - indicate:
    - Input/output files
    - **M**: number of map tasks
    - **R**: number of reduce tasks
    - **W**: number of machines
  - Write *map* and *reduce* functions
  - Submit the job
- This requires no knowledge of parallel/distributed systems!!!
- What about everything else?

THE UNIVERSITY OF UTAH

Data Distribution

- Input files are split into **M** pieces on distributed file system
  - Typically ~ 64 MB blocks
- Intermediate files created from *map* tasks are written to local disk
- Output files are written to distributed file system

Assigning Tasks

- Many copies of user program are started
- Tries to utilize data localization by running *map* tasks on machines with data
- One instance becomes the Master
- Master finds idle machines and assigns them tasks

Execution (map)

- *Map* workers read in contents of corresponding input partition
- Perform user-defined *map* computation to create intermediate <key,value> pairs
- Periodically buffered output pairs written to local disk
  - Partitioned into **R** regions by a partitioning function

Partition Function

- Example partition function: hash(key) mod **R**
- **Why do we need this?**
- **Example Scenario:**
  - **Want to do word counting on 10 documents**
  - **5 *map* tasks, 2 *reduce* tasks**

Execution (reduce)

- Reduce workers iterate over ordered intermediate data
  - Each unique key encountered – values are passed to user's reduce function
  - eg. <key, [value1, value2,..., valueN]>
- Output of user's *reduce* function is written to output file on global file system
- When all tasks have completed, master wakes up user program

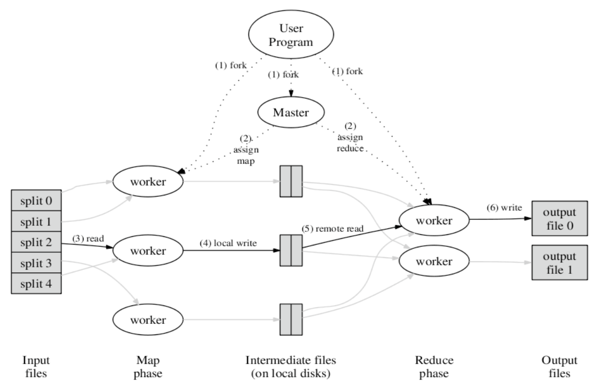
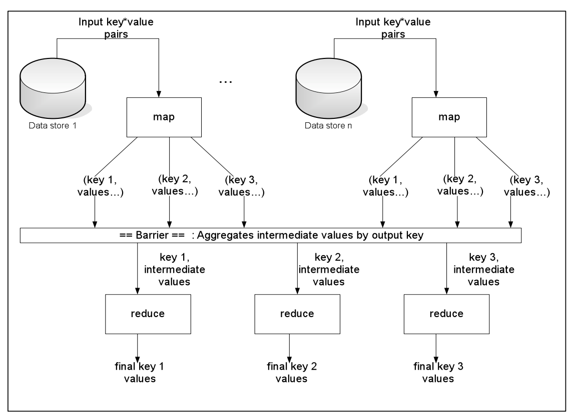


Figure 1: Execution overview

Observations

- No *reduce* can begin until *map* is complete
- Tasks scheduled based on location of data
- If *map* worker fails any time before *reduce* finishes, task must be completely rerun
- Master must communicate locations of intermediate files
- MapReduce library does most of the hard work for us!



Fault Tolerance

- Workers are periodically pinged by master
  - No response = failed worker
- Master writes periodic checkpoints
- On errors, workers send "last gasp" UDP packet to master
  - Detect records that cause deterministic crashes and skips them

Fault Tolerance

- Input file blocks stored on multiple machines
- When computation almost done, reschedule in-progress tasks
  - Avoids "stragglers"

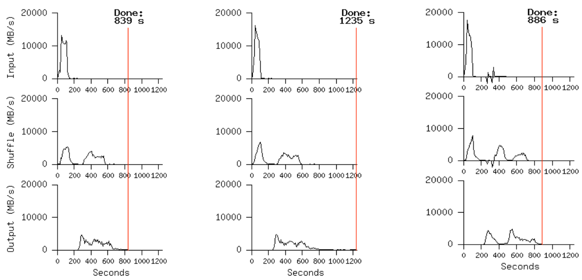
Debugging

- Offers human readable status info on http server
  - Users can see jobs completed, in-progress, processing rates, etc.
- Sequential implementation
  - Executed sequentially on a single machine
  - Allows use of gdb and other debugging tools

Performance

- Tests run on 1800 machines
  - 4GB memory
  - Dual-processor # 2 GHz Xeons with Hyperthreading
  - Dual 160 GB IDE disks
  - Gigabit Ethernet per machine
- Run over weekend – when machines were mostly idle
- Benchmark: Sort
  - Sort 10^10 100-byte records

Performance



MapReduce Conclusions

- Simplifies large-scale computations that fit this model
- Allows user to focus on the problem without worrying about details
- Computer architecture not very important
  - Portable model

References

- Jeffery Dean and Sanjay Ghemawat, MapReduce: Simplified Data Processing on Large Clusters
- Josh Carter, http://multipart-mixed.com/software/mapreduce_presentation.pdf
- Ralf Lammel, Google's MapReduce Programming Model – Revisited
- http://code.google.com/edu/parallel/mapreduce-tutorial.html

- RELATED
  - Sawzall
  - Pig
  - Hadoop