MapReduce

CCP A.A. 2008-2009
M. Danelutto
Aprile 2009

Contents

- Map pattern
- Reduce pattern
- MapReduce: implementations
- MapReduce: theoretical foundations
- MapReduce *à la google*
- Final course project

Map pattern

- operates on collections
  - vectors, arrays, lists, ...
- applies a function f to all the elements in the collection
  - f side effect free
- embarrassingly parallel pattern
- collection = stream ⇒ map = farm
  - different problems related to parallelism exploitation
  - same user view

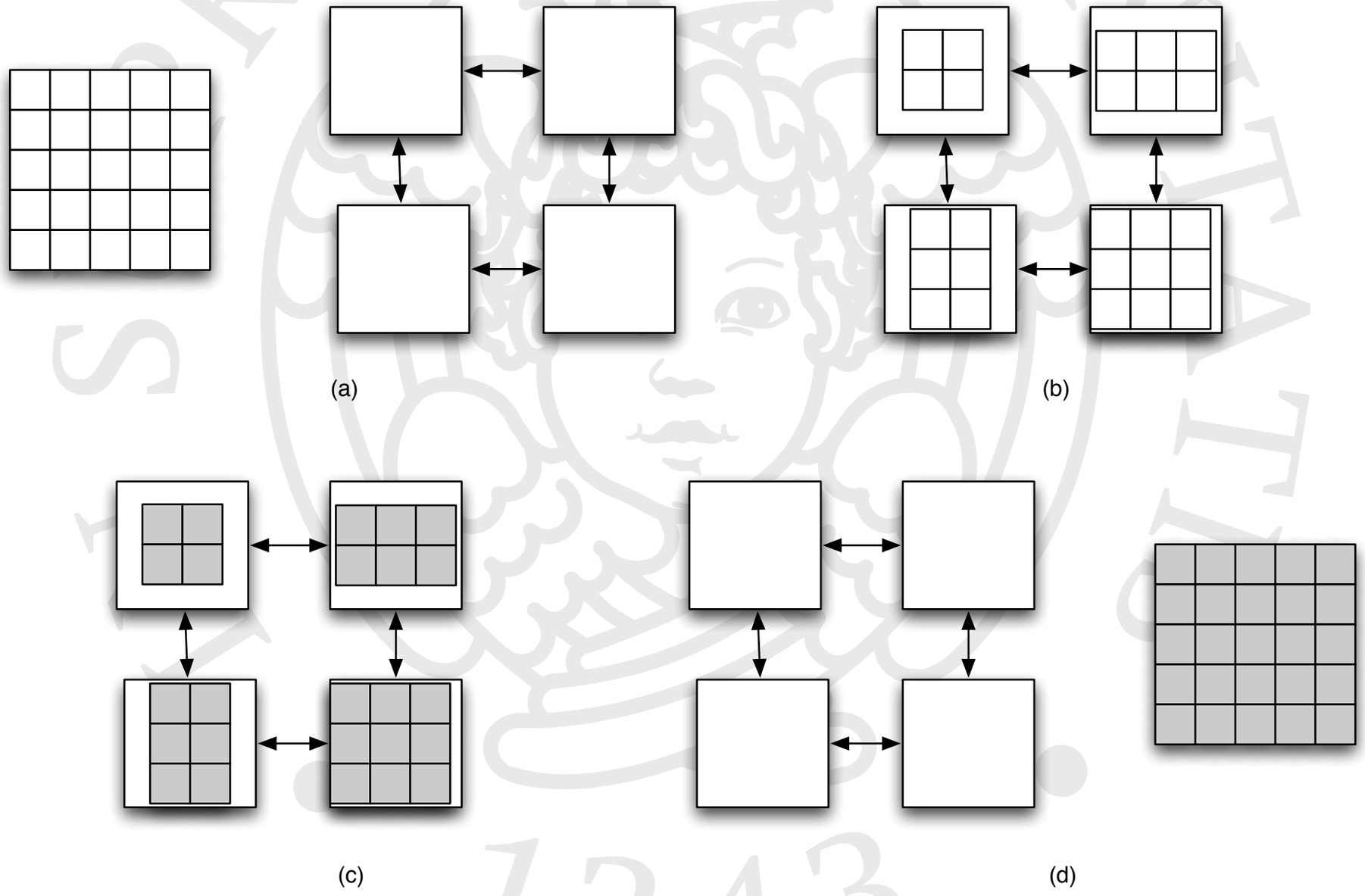
Reduce patter

- operates on collections
  - vectors, arrays, lists, ...
- “sums” up all the elements in the collection
  - using a binary operator $\oplus$
- definitely not an embarrassingly parallel pattern
  - with associative and commutative $\oplus$
    - tree reduction applies ($O(\log_2 n)$)
  - plain $\oplus \Rightarrow O(n)$

Map typical implementation

- data collection input
- data collection spread
  - with partitioning, load balancing ensured as much as possible
- local computation
- data collection gathering
- key point to efficiency: computation grain
  - fine grained f ⇒ iterative local computation
  - data dimensions (domain and codomain of f) to be considered

Map typical implementation



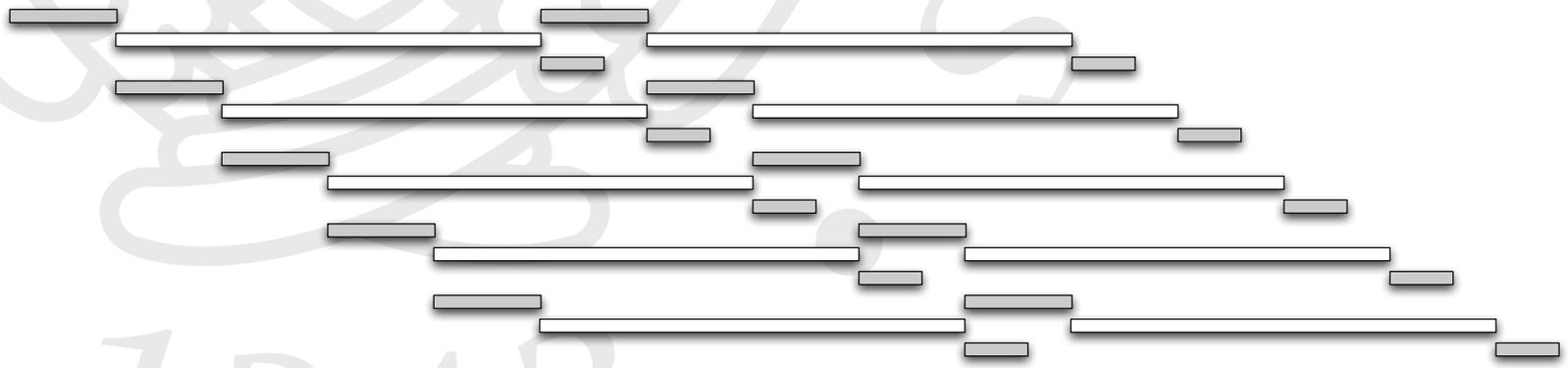
Map performance model (simple) (1)

- Data parallelism
  - overhead:
    - scattering of input data + gathering of results
  - improvement
    - total cost in addition to overhead if #partition*f

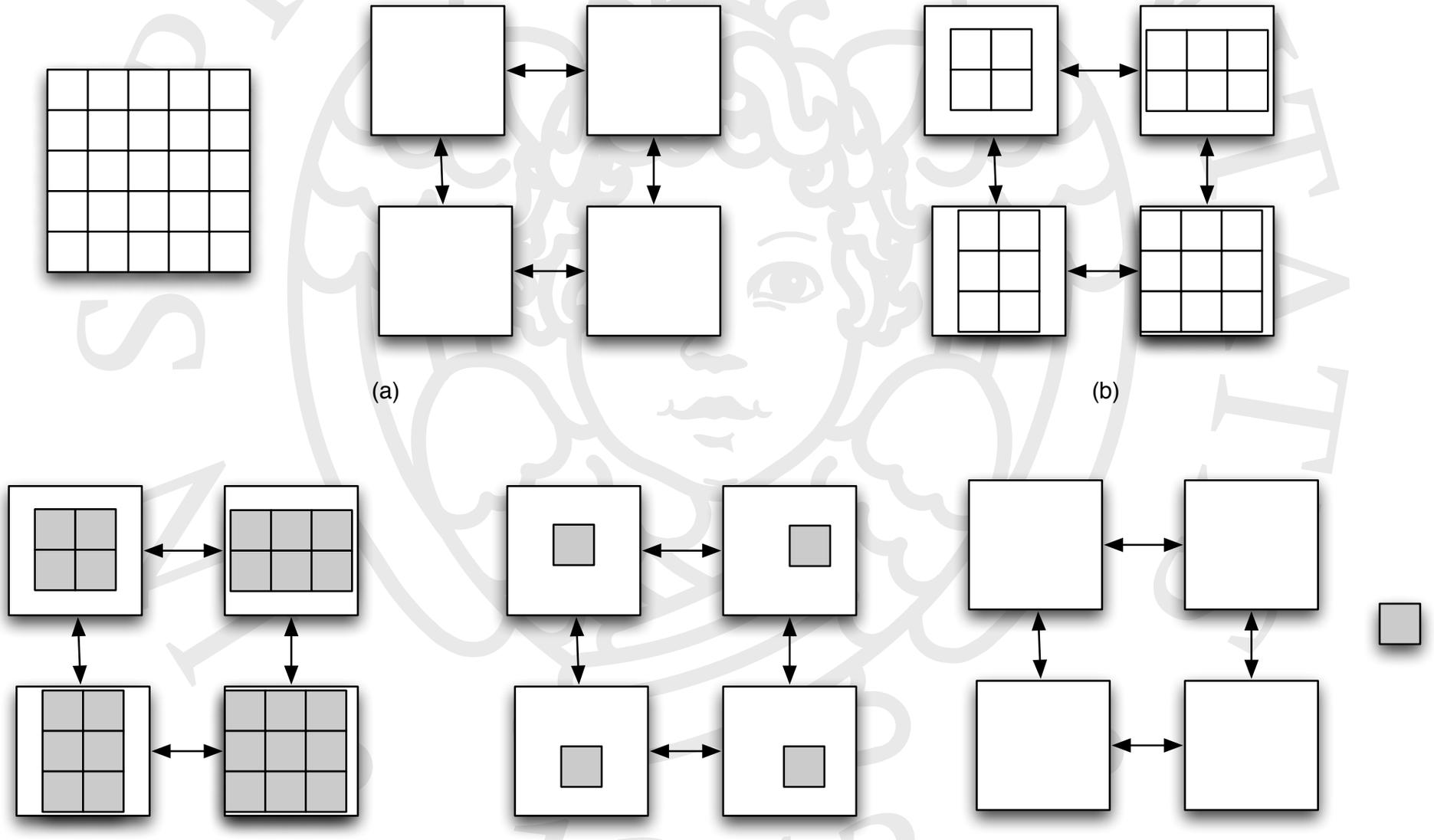
- T = nw * Tin +
  Tf * #partition +
  Tout

Map performance model (simple) (2)

- Three stage pipeline to be balanced to exploit stream parallelism
  - distribute input data (code?)
    - depends on the interconnection structure
    - depends on the data size
  - local map
    - proportional to partition size and function cost
  - collect results
    - depends on the interconnection structure



Reduce typical implementation

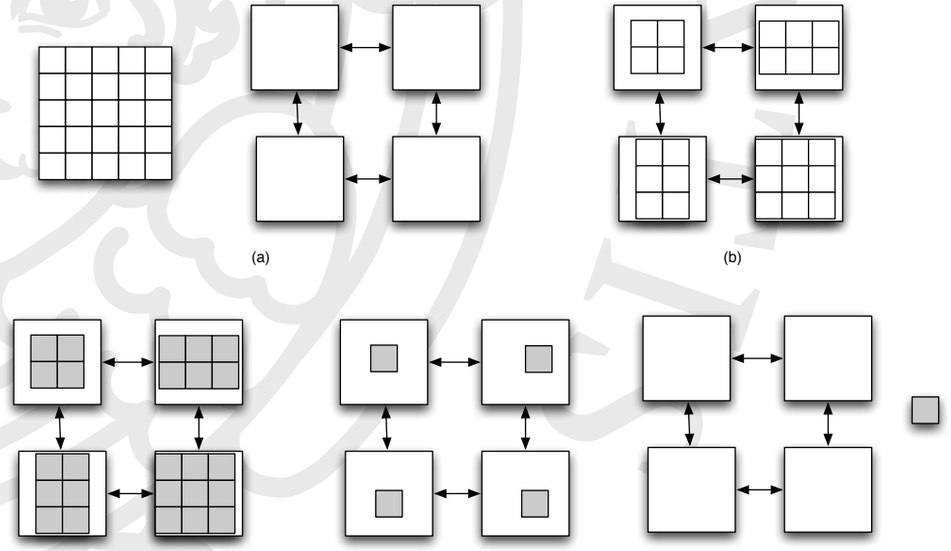


Reduce performance model (simple)

- Similar to map:
  - data parallel: overhead for distribution and reduce gathering
  - stream parallel: balance of the three stages
    - partitioning
    - local reduction
    - global reduction
- with similar problems & evaluation

MapReduce pattern

- common in applications
  - e.g. map(find this face(myface)) + reduce(is(true))
  - e.g. map(applyPhoneCallCostFunction) + reduce(sum)
  - ...
- exploitation synergies in common implementation
  - gather phase of the map is added up with the reduce computation



MapReduce pattern

- operates on collections
  - vectors, arrays, lists, ...
- applies function f on all the collection elements
  - f is side effect free
- “sums” up the results with function ⊕
- first part is embarrassingly parallel (map)
- reduce phase can be implemented with a reduction tree
  - the distribution phase is not needed anymore
  - data is already in place

History

- map and reduce patterns appeared first in 1977 Backus's Turing award lecture note
- *Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs (1978)*
- FP
  - objects + functions (obj→obj) + application + functional forms
    - objects: atoms, numbers, ⊥
    - functions: +,-,distl,distr,trans,lenght,...

FP (cont'd)

- functional forms:
  - composition: (f ○ g):x = f(g x)
  - construction: [f,g,...]:x = <f:x, g:x, ...>
  - insert: /f:<x1,x2,...> = f:<x1,/f:<x2,...>>
    - this is actually the *reduce*
  - apply to all αf:<x1,x2,...> = <f:x1, f:x2, ...>
    - this is actually the *map*
- algebra of programs defined with transformations

FP sample code

- def IP = (/+) ∘ (α x) ∘ trans;
  IP :<<1,2,3> <3,2,1>> =
  ((/+) ∘ (α x) ∘ trans) : <<1,2,3> <3,2,1>> =
  ((/+) ∘ (α x)) : <<1,3>,<2,2>,<3,1>> =
  (/+) : <3,4,3> =
  10
- def MM = (α α IP) ∘ (α distl) ∘ distr ∘ [1, trans ∘ 2]
  - [1, trans ∘ 2] transposes second matrix
  - distr creates pairs of the 1st matrix and a column of the 2nd
  - (α distl) creates row column pairs
  - (α α IP) performs IP on all the pairs

FP: rules in the algebra

- [f1, ... , fn] ○ g = [f1 ○ g, ... fn ○ g]
- αf ○ [g1, ... , gn] = [f ○g1, ... , f ○gn]
- /f ○ [g1, ... , gn] = f ○[g1, f ○ [g2, ... f ○[gn-1,gn]...]
- (αf) ○ (αg) = α(f ○ g)

- used to derive new programs from existing ones
- used to prove equivalence of two programs

Map and Reduce in ’80s and ’90s

- Bird Meertens formalism (Skillicorn ’92)
- Operate on lists:

$$[\cdot] : \alpha \rightarrow \alpha* \qquad [\cdot]a = [a]$$
$$[] : \text{unit} \rightarrow \alpha* \qquad [] = K_{[]}$$
$$+\!\!+ : \alpha* \times \alpha* \rightarrow \alpha* \qquad [as] +\!\!+ [bs] = [as, bs]$$

- Two operations (map&reduce):
  - For any function $f : \alpha \rightarrow \beta$ a function $f* : \alpha* \rightarrow \beta*$;
  - For any $M = (\alpha, \oplus, \text{id})$ a monoid, a reduction $\oplus/ : \alpha* \rightarrow M$
- Set of rules:

$$\begin{aligned} f* \cdot [\cdot]_\alpha &= [\cdot]_\beta \cdot f \\ f* \cdot +\!\!+_\alpha &= +\!\!+_\beta \cdot (f*, f*) \\ f* \cdot []_\alpha &= []_\beta \\ \text{id}_\alpha* &= \text{id}_{\alpha*} \end{aligned}$$

$$\begin{aligned} (g \cdot f)* &= g* \cdot f* \\ \oplus/ \cdot []_\alpha &= \text{id}_\oplus \\ \oplus/ \cdot [\cdot]_\alpha &= \text{id}_\alpha \\ \oplus/ \cdot +\!\!+ &= (\oplus/) \oplus (\oplus/) \\ h \cdot \oplus/ &= \otimes/ \cdot h* \ (h \text{ a homomorphism}) \\ h &= \otimes/ \cdot (h \cdot [\cdot]_\alpha)* \ (h \text{ a homomorphism}) \end{aligned}$$

Map, reduce and homomorphisms

- $h \circ /f = /g \circ \alpha h$
- a reduce with function f followed by the applications of an homomorphism

  can be implemented (is equivalent)

  to an map of function h followed by a reduce
- the function mapped h is in relation with the function f

Homomorphism

- informally: “Structure” preserving functions
- e.g. (from Wikipedia)

The real numbers are a ring, having both addition and multiplication. The set of all 2 × 2 matrices is also a ring, using matrix addition and matrix multiplication. Define a function between these rings by

$$f(r) = \begin{pmatrix} r & 0 \\ 0 & r \end{pmatrix}$$

where *r* is a real number. Then *f* is a homomorphism of rings, since *f* preserves both addition:

$$f(r+s) = \begin{pmatrix} r+s & 0 \\ 0 & r+s \end{pmatrix} = \begin{pmatrix} r & 0 \\ 0 & r \end{pmatrix} + \begin{pmatrix} s & 0 \\ 0 & s \end{pmatrix} = f(r) + f(s)$$

and multiplication:

$$f(rs) = \begin{pmatrix} rs & 0 \\ 0 & rs \end{pmatrix} = \begin{pmatrix} r & 0 \\ 0 & r \end{pmatrix}\begin{pmatrix} s & 0 \\ 0 & s \end{pmatrix} = f(r)\,f(s).$$

Skeleton frameworks with map/reduce

- P3L (Pisa, 1991)
- Gorlatch skeleton theory (Passau, ’90)
- ASSIST (Pisa, 2001)
  - embedded in parmod construct
- SkeTo (Tokio Univ. 2003)
- Muesli (Passau, 2005, but former work in Skil, late ’90)
- ...

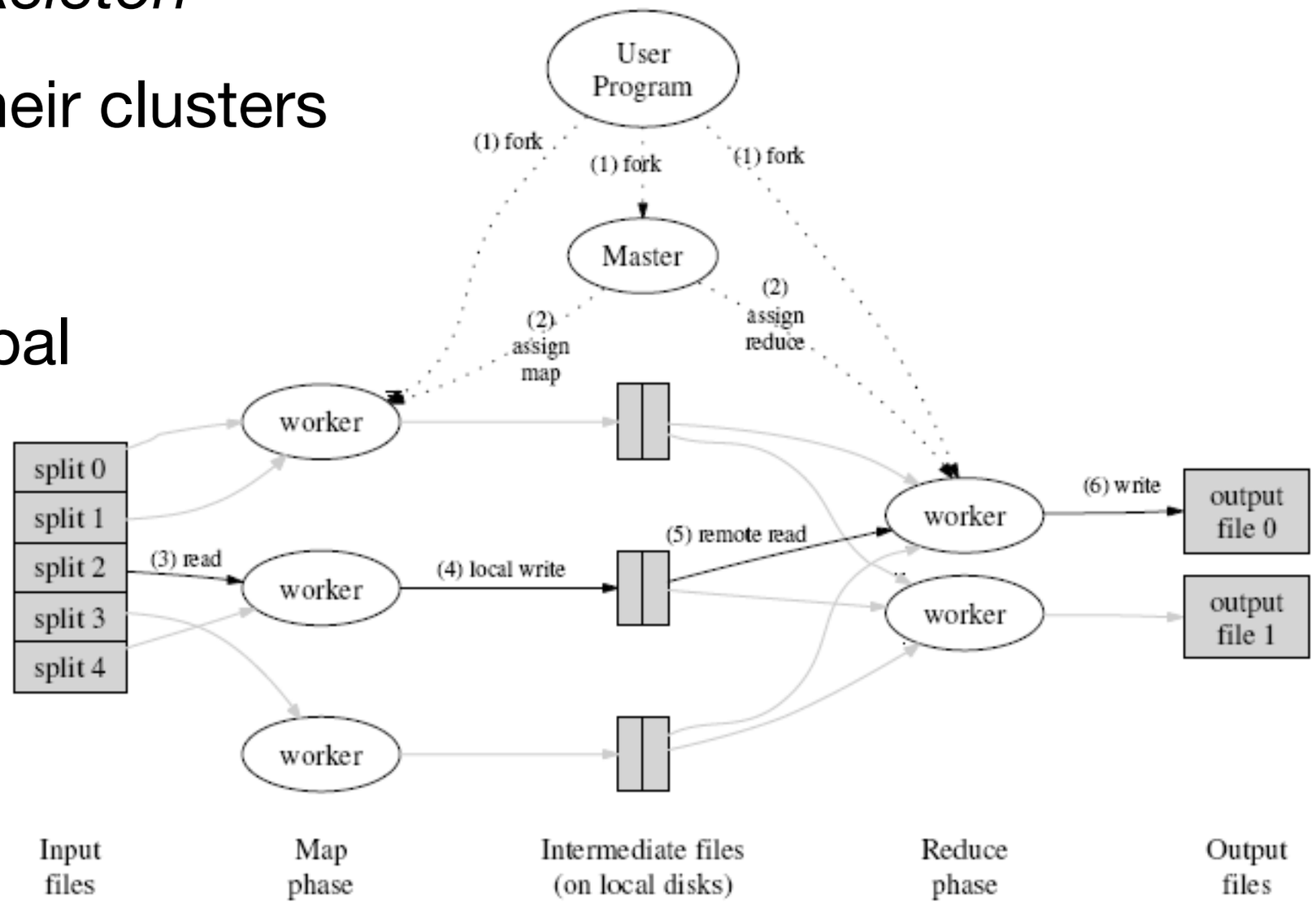
Google MapReduce

- The context
  - searching engine with huge data
  - most frequent operation is searching, all the matching results should be given to the user
    - data to be searched must be split among different machines (size too big for a single machine/storage + fault tolerance)

- The idea: paper by Jeffrey Dean and Sanjay Ghemawat 2004 *MapReduce: Simplified Data Processing on Large Clusters*
  - with a few/none references to skeleton work !

The concept

- map reduce *à la skeleton*
- implemented on their clusters
- map local
- reduce local + global



Google examples

- **Distributed Grep**: The map function emits a line if it matches a given pattern. The reduce function is an identity function that just copies the supplied intermediate data to the output.

- **Count of URL Access Frequency**: The map function processes logs of web page requests and outputs <URL, 1>. The reduce function adds together all values for the same URL and emits a <URL, total count> pair.

- **Reverse Web-Link Graph**: The map function outputs <target, source> pairs for each link to a target URL found in a page named "source". The reduce function concatenates the list of all source URLs associated with a given target URL and emits the pair: <target, list(source)>.

- **Term-Vector per Host**: A term vector summarizes the most important words that occur in a document or a set of documents as a list of <word, frequency> pairs. The map function emits a <hostname, term vector> pair for each input document (where the hostname is extracted from the URL of the document). The reduce function is passed all per-document term vectors for a given host. It adds these term vectors together, throwing away infrequent terms, and then emits a final <hostname, term vector> pair.

- **Inverted Index**: The map function parses each document, and emits a sequence of <word, document ID> pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a <word, list(document ID)> pair. The set of all output pairs forms a simple inverted index. It is easy to augment this computation to keep track of word positions. [1]

Google “skeleton”

- User [1] provides
  - map function
  - reduce function
- Google provides
  - mapreduce C++ class

[1] users are internal google users, actually

Google implementation

1. The MapReduce library in the user program first **shards the input files** into M pieces of typically 16 megabytes to 64 megabytes (MB) per piece. It then starts up many copies of the program on a cluster of machines.

2. One of the copies of the program is special: the master. The rest are workers that are assigned work by the master. There are M map tasks and R reduce tasks to assign. The **master picks idle workers and assigns each one a map task or a reduce task**.

3. A worker who is assigned a **map task** reads the contents of the corresponding input shard. It parses key/value pairs out of the input data and passes each pair to the user-defined Map function. The intermediate key/value pairs produced by the Map function are buffered in memory.

4. Periodically, the **buffered pairs are written to local disk**, partitioned into R regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers.

5. When a reduce worker is notified by the master about these locations, it uses **remote procedure calls to read the buffered data from the local disks of the map workers**. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. If the amount of intermediate data is too large to fit in memory, an external sort is used.

6. The **reduce worker iterates** over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user's Reduce function. The output of the Reduce function is appended to a final output file for this reduce partition.

7. When **all map tasks and reduce tasks have been completed, the master wakes up the user program**. At this point, the MapReduce call in the user program returns back to the user code.

Google optimizations

- Partitioning
  - data mapped to *hash(...) % R*
- Combiner function
  - leave the programmer express a local “reduction” applied before networking the results for the overall reduce
- Backup tasks
  - computation of the last tasks is replicated to avoid “slow” processor problems

Sample google code (1)

```
#include "mapreduce/mapreduce.h"

// User's map function
class WordCounter : public Mapper {
 public:
  virtual void Map(const MapInput& input) {
    const string& text = input.value();
    const int n = text.size();
    for (int i = 0; i < n; ) {
      // Skip past leading whitespace
      while ((i < n) && isspace(text[i]))
        i++;

      // Find word end
      int start = i;
      while ((i < n) && !isspace(text[i]))
        i++;
      if (start < i)
        Emit(text.substr(start,i-start),"1");
    }
  }
};
REGISTER_MAPPER(WordCounter);
```

Sample google code (2)

```
// User's reduce function
class Adder : public Reducer {
  virtual void Reduce(ReduceInput* input) {
    // Iterate over all entries with the
    // same key and add the values
    int64 value = 0;
    while (!input->done()) {
      value += StringToInt(input->value());
      input->NextValue();
    }

    // Emit sum for input->key()
    Emit(IntToString(value));
  }
};
REGISTER_REDUCER(Adder);
```

Sample google code (3)

```
int main(int argc, char** argv) {
  ParseCommandLineFlags(argc, argv);

  MapReduceSpecification spec;

  // Store list of input files into "spec"
  for (int i = 1; i < argc; i++) {
    MapReduceInput* input = spec.add_input();
    input->set_format("text");
    input->set_filepattern(argv[i]);
    input->set_mapper_class("WordCounter");
  }

  // Specify the output files:
  //     /gfs/test/freq-00000-of-00100
  //     /gfs/test/freq-00001-of-00100
  //     ...
  MapReduceOutput* out = spec.output();
  out->set_filebase("/gfs/test/freq");
  out->set_num_tasks(100);
  out->set_format("text");
  out->set_reducer_class("Adder");
```

Sample google code (4)

```
  // Optional: do partial sums within map
  // tasks to save network bandwidth
  out->set_combiner_class("Adder");

  // Tuning parameters: use at most 2000
  // machines and 100 MB of memory per task
  spec.set_machines(2000);
  spec.set_map_megabytes(100);
  spec.set_reduce_megabytes(100);

  // Now run it
  MapReduceResult result;
  if (!MapReduce(spec, &result)) abort();

  // Done: 'result' structure contains info
  // about counters, time taken, number of
  // machines used, etc.

  return 0;
}
```

Usage of the Java Hadoop

- Installation
  - download the hadoop.xxx.tar.gz
  - unpack and set up in the HADOOP_HOME
- $HADOOP_HOME/bin in the PATH
- $HADOOP_HOME/hadoop-xxx-core.jar in the CLASSPATH
- run local examples
  - conf/*xml as of distrib (with hadoop_env.xml JAVA_HOME set, at least)
- run non local (see web page and sample code)