Scalable Data Mining on Parallel, Distributed and Cloud Computing Systems

Domenico Talia
DIMES, Università della Calabria & DtoK Lab
Italy
talia@dimes.unical.it

Lecture Goals

- Parallel systems, distributed computing infrastructures, like Grids and P2P networks, and Cloud computing platforms, offer an effective support for addressing both the
  - computational and
  - data storage needs
  of Big Data mining and parallel analytics applications.

- Complex data mining tasks involve data- and compute-intensive algorithms that require large storage facilities together with high performance processors/systems to get results in suitable times.
Lecture Goals

- In this lecture we introduce the most relevant topics and the main research issues in high performance data mining including:
  - parallel data mining strategies,
  - distributed analysis techniques, and
  - knowledge services and Cloud-based data mining.

- We discuss parallel models, scalable algorithms and techniques, data mining programming tools and applications.

General Outline

LESSONS

1. Parallel Data Mining Strategies

2. Distributed and Service-oriented Data Mining

3. Data Mining Tools on Clouds
Data Availability or Data Deluge?

- Some decades ago the main problem was the shortage of information, now the challenge is
  - the very large volume of information to deal with and
  - the associated complexity to process it and to extract significant and useful parts or summaries.

Complex Big Problems ...

☑️ Big and complex problems need to be solved by HPC systems, Clouds, and large scale distributed computing systems.

☑️ We need also scalable algorithms, techniques, and systems.
...and Big Data

- Data sources are larger and larger and ubiquitous (Web, sensor networks, mobile devices, telescopes, social media, bio labs, large scientific instruments, ...).

- Large data sources in many fields **cannot be read by humans** so

- The huge amount of data available today requires smart data analysis techniques to help people to deal with it.

Data Analysis

- ... Although storing data is a big issue today, a vital issue is **analyse, mine, and process data for making it useful**.
- We are interested in the **value of data**.
Distributed Data Intensive Apps

- In this scenario, Parallel, Distributed, Service-oriented and Cloud computing systems can provide an effective 
  computational and data storage support.

- Clouds, HPC systems, many core systems and distributed
  systems allow for running distributed data intensive
  applications and data mining in large and distributed data
  sets.

- Clouds can be used in integrated platforms through service
  interfaces for manage large data sources and process them.

Data Mining and Computational Needs

- High performance computers and parallel data mining
  algorithms together offer hardware and software layers to
  implement data analysis solutions for mining very large data
  sets.

Parallel and distributed data mining
LESSON 1

Parallel Data Mining Strategies

Need for Parallel Formulations

- Need to handle very large datasets (Gbytes, Tbytes, streams).

- Memory limitations of sequential computers cause sequential algorithms to make multiple expensive I/O passes over data.

- Need for scalable, efficient (fast) data mining computations
  - Gain competitive advantage.
  - Handle larger data for greater accuracy in shorter times.
Data Mining and Computational Needs

- Is not uncommon to have sequential data mining applications that require some days or weeks to complete their task.

- Parallel computing can bring significant benefits in the implementation of data mining and knowledge discovery applications by means of the exploitation of inherent parallelism of data mining algorithms.

- Main goals:
  - performance improvements of existing techniques,
  - implementation of new (parallel) techniques and algorithms,
  - concurrent analysis with different data mining techniques and result integration to get a more accurate model → Concurrent Learning

Data Mining and Computational Needs

- Big is a term to name large and very complex datasets. It refers to data that cannot be managed/analyzed with conventional computers.

- When
  - large data sets are coupled with
  - Complex, heterogeneous, real-time features,

  it is necessary to combine different technologies for implementing high-performance knowledge discovery systems.

- Parallel computing systems are part of this solution scenario.
Parallelism

Main reasons to use parallelism

- The real world is inherently parallel;
- Many real life problems are naturally parallel/distributed and can be modeled more directly in a parallel way;
- Parallelism makes available more computational performance;
- There are limits to sequential computing (the speed of light is a bound);
- A multiprocessor is still likely to be more cost-effective for many applications than using leading-edge uniprocessors.

These reasons may lead to a mass market if the software design is not very complex.

Parallel Computing

- Parallel computers represent an opportunity.
- This opportunity is driven by parallel (concurrent) programming languages that make high-performance machines usable and useful.
- Parallel languages allows users to design parallel algorithms as a set of concurrent actions mapped onto different computing elements.
- Cooperation among actions can be performed in several ways according to the selected paradigm.

High-level languages might decrease both the design time and the execution time Easier Approach to Parallelism for New Users.
Parallel Computing

Typical issues in parallel programming are:
- process creation,
- synchronization,
- communication handling,
- deadlock, and
- process termination.

These issues arise because there are many flows of control through the program (one per process).

Languages should make the programming of multicomputers to be not much harder than programming sequential computers.
Parallel Computing

- Parallel (and distributed) computers consist of three building blocks:
  - Processors,
  - Memory modules, and
  - Interconnection network.

- There has been steady development of the sophistication of each of these building blocks, but it is their arrangement that most differentiates one parallel computer from another.

Background

Flynn’s Taxonomy

- Two parameters: Instruction stream, Data stream.

- **SISD (Single Instruction stream - Single Data stream)**
  - sequential computers.

- **SIMD (Single Instruction stream - Multiple Data stream)**
  - several processors execute the same instruction on different data.

- **MISD (Multiple Instruction stream - Single Data stream)**
  - many processors execute the different instructions on the same data. (not practical!)

- **MIMD (Multiple Instruction stream - Multiple Data stream)**
  - several processors execute different instructions on different data.
SIMD Computer

von Neumann Computer

Parallel Hardware

- Distributed-memory machines
  - Each processor has local memory and disk
  - Communication via message-passing
  - Harder to program: explicit data distribution
  - Goal: minimize communication

- Shared-memory machines
  - Shared global address space and disk
  - Communication via shared memory variables
  - Easier of programming
  - Goal: maximize locality, minimize false sharing

- Parallel Solutions today: Cluster of SMPs, ManyCore, Clouds, Exascale systems.
Distributed Memory Architecture (Shared Nothing)

Shared Disk Architecture
Shared Memory Architecture (Shared Everything)

Cluster of SMPs / Multi-core / Many-core
Bus Based e switch based Interconnection

Some Network Topologies

Linear Array

Ring

Mesh

Fully Connected
Hypercube

<table>
<thead>
<tr>
<th>1110</th>
<th>1111</th>
</tr>
</thead>
<tbody>
<tr>
<td>1010</td>
<td>1011</td>
</tr>
<tr>
<td>0010</td>
<td>0011</td>
</tr>
</tbody>
</table>

$d = 4$

Clusters

Programming Environment

Middleware

OS

I/O

Interconnection Network
Multi-core & Many-core

- Parallel computing as a solution to the Moore law trend.
- Parallelism in mobile PC, smartphones, sensors
- Dual-core, quad-core
- Many-core: more than 16 processors on a single PC
- Massive Parallelism in the box.
- Exascale Systems

Grid & Cloud

- New architectures for parallel and distributing computing
- Large scale integration of parallel systems.
- Heterogeneity (Grid)
- Homegeneous systems (Cloud)
- Parallel & Distributed virtual machines
**Metrics: Speedup**

- **Speedup**
  - $T_s$ sequential execution time (on 1 processor)
  - $T_n$ parallel execution time (on $n$ processors)

  $$S_n = T_s / T_n$$

- Ideal speedup on $n$ processors is equal to $n$.
- Two speedups:
  - *Relative*
  - *Real*: using the most efficient sequential algorithm.

**Efficiency**

- **Efficiency.**

  $$E_n = S_n / n = T_s / nT_n$$

- Measure the effective use of each single processor

  - *Generally*

  $$1 \leq S_n \leq n$$

  so

  $$1/n \leq E_n \leq 1$$
Basic parallel mechanisms

- Parallelism in programming languages was originally investigated as a branch of operating systems programming.

- Basic mechanisms for process synchronization and communication
  - semaphores,
  - conditional critical regions,
  - monitors, and
  - send and receive.

Semaphores and Monitors

- Semaphores are a simple mechanism for the implementation of synchronization between processes that access the same resource. Each access to a shared resource must be preceded by a request (wait) operation and followed by a signal operation on the same semaphore.

- In conditional critical regions each shared resource may be accessed only using conditional critical region statements (region) which guarantee mutual exclusion on the resource.

- Monitors encapsulate both a resource and operations that manipulate it. Resources defined in a monitor must be accessed using only the operations defined by the monitor itself.
Basic Message Passing

P0 :: Send (P1, M_A);  
P0 :: receive (P1, M_B);  
P1 :: Send (P0, M_B);  
P1 :: Receive (P0, M_A);
Parallel programming languages : SM

- The concept of shared memory is a useful way to separate program control flow issues from issues of data mapping, communication, and synchronization.

- Processes cooperate through a shared memory space where shared variables are stored.

- The actual process of communication is thus straightforward. What is difficult is detecting when it is safe either to put a value into the location or to remove it.

Parallel programming languages : SM

- One way to make shared-memory programming easier is to use techniques adapted from operating systems to enclose accesses to shared data in critical sections.

- In shared-memory parallel architectures this programming style can be mapped directly.

- These paradigms are generally used on multiprocessors using not many processing elements (from 2 ten up to twenty).
Parallel programming languages: SM

- Physical shared memory cannot be implemented on massively parallel computers, but it is a useful abstraction, even if the implementation it hides is distributed.

- A useful approach for massively parallel computers is to provide a high-level abstraction of shared memory.

- One way to do this is called virtual shared memory. The programming language presents a view of memory as if it is shared, but the implementation may or may not be.

- The other way is to build a system based on a useful set of sharing primitives.

Virtual Shared Memory

- Reading and writing a shared variable is implemented by message passing between a user process and the process that manages that shared variable.

- The goal of such approach is to emulate shared memory well enough that the same number of messages travel around the system when a program executes as would have traveled if the program had been written to pass messages explicitly.

- The emulation of shared memory imposes no extra message traffic.
Some languages for parallel programming provide basic mechanisms for data sharing.

- Shared Memory Languages:
  - Linda,
  - Orca,
  - SDL,
  - OpenMP,
  - Pthreads,
  - Ease,
  - Opus,
  - Java.

Linda provides an associative memory abstraction called **tuple space**.

Threads communicate with each other only by placing tuples in and removing tuples from this shared associative memory.

Sequential languages can be augmented with tuple space operations to create a new parallel programming language.

Linda is called a **coordination language** because the tuple space abstraction coordinates, but is orthogonal to, the computation activities.
In Linda tuple space is accessed by four actions:

- **out(T)** to place a tuple T in the tuple space;
- **in(T)** to remove a tuple T from the tuple space;
- **rd(T)** to read a tuple T from the tuple space;
- **eval(T)** to evaluate T before storing the result in the tuple space.

The **eval** operation creates a process to evaluate the tuple components. For example,

\[
\text{eval}("R", f(x,y))
\]

creates a process to compute "R" and f(x,y) then to store result of tuple (R,f(x,y)).

---

A simple Linda program that finds prime numbers.

```linda
main()
{
    int i, ok;
    for(i=2; i < LIMIT; ++i)
        eval("primes", i, Is_Prime(i));
    for(i=2; i <= LIMIT; ++i)
    {
        rd("primes", i, ? ok);
        if (ok)
            printf("%d\n", i);
    }
}
```

---
Linda

- There are two main difficulties with Linda:

1. The first is that a single, shared, associative memory does not provide any way to structure the processes that use it, so that Linda programs have no natural higher-level structure.

2. The second is that, as programs get larger, the lack of scoping in tuple space makes the optimizations of tuple space access described above less and less efficient.

Orca

- Orca is a language based on a useful set of primitives for sharing of data among processes.

- The Orca system is a hierarchically structured set of abstractions.
  - At the lowest level, reliable broadcast is the basic primitive so that writes to a replicated structure can rapidly take effect throughout a system.
  - At the next level of abstraction, shared data are encapsulated in passive objects that are replicated throughout the system.

- On these levels, Orca itself provides an object-based language to create and manage objects.
Each object consists of a **specification** and an **implementation** section.

For process creation Orca uses an explicit fork

```plaintext
fork proc1 (shared data) [on (processor)];
```

Processes communicate indirectly by **shared data objects**.

A process may pass an object as shared to its children

```plaintext
process proc1 (Id: integer; ObjX: shared ObjectType);
begin
  . . . .
end;
```

---

**Orca**

```plaintext
object specification NewObj;
  operation Op1 (x:integer);
  operation Op2 (x:integer);
end;

object implementation NewObj;
  x:integer;
  operation Op1 (y:integer);
  begin
    . . .
  end;
  operation Op2 (z:integer);
    . . . . .
```
OpenMP

- OpenMP is a library (application program interface - API) that supports parallel programming on shared memory parallel computers.

- OpenMP has been developed by a consortium of vendors of parallel computers (DEC, HP, SGI, Sun, Intel, ...) with the aim to have a standard programming interface for parallel shared-memory machines.

- The OpenMP functions can be used inside Fortran, C and C++ programs.

- They allow the parallel execution of code (parallel DO loop), the definition of shared data (SHARED) and synchronization of processes.

OpenMP allows a user

- to define regions of parallel code (PARALLEL) where it is possible to use local (PRIVATE) and shared variables (SHARED);
- to synchronize processes by the definition of critical sections (CRITICAL) for shared variables (SHARED);
- to define synchronization points (BARRIER).

- OpenMP has been proposed last year with the goal to become a standard for shared-memory architectures (like PVM and MPI for distributed memory machines).

- Support for general task parallelism is not included in the OpenMP specification!
Java

- An important shared-memory programming language is Java that is popular because of its connection with platform-independent software delivery on the Web.

- Java is an object-oriented language that supports the implementation of concurrent programs by process (called threads) creation (new) and execution (start):

  ```java
  new proc (arg1a, arg1b, ..) ;
  new proc (arg2a, arg2b, ..) ;
  ```

  where proc is an object of a thread class.

- Java threads communicate and synchronize through condition variables.

Java

- Shared variables are accessed from within synchronized methods.

- Java programs execute synchronized methods in a mutually exclusive way generating a critical section (misleadingly called monitor).

- However, notify and wait operations must be explicitly invoked within such sections, rather than being automatically associated with entry and exit.

- Java does not provide any mechanism for deadlock detection or prevention!
Java

- This concurrent programming model is useful for using Java on a sequential computer (pseudo-parallelism) or on shared-memory parallel computers.

- To use Java on distributed-memory parallel computer there are different solutions:
  - sockets,
  - RMI (Remote Method Invocation),
  - Java + CORBA.

Distributed Memory Paradigms

- These languages reflect the model of distributed-memory architectures composed of a set of processing elements (PE) connected by a network (multicomputers, clusters, LANs).

- In this model, a parallel program consists of a set of processes cooperating by message passing and located on one or many computers.

- The two major issues in designing distributed parallel languages are related to process spawning and process cooperation.
Distributed Memory Paradigms

- Some languages provide primitives for explicit process creation during the parallel program execution (**dynamic creation**): `fork/join`, `new` and `create`.

- While in others the total number of processes is defined at compile time (**static creation**): `PAR`, `parbegin`, `cobegin/coend`.

- Defined mechanisms for the cooperation of concurrent processes can be divided into four main classes:
  - `explicit message passing`,
  - `rendezvous`,
  - `remote procedure call`,
  - `data parallelism`.

Distributed Memory Paradigms

- Distributed memory models and languages:
  - `Ada`, `CSP`, `Occam`, `Concurrent C`, `CILK`, `Fortran M`.
  - `SR`, `DP`, `CLU`.
  - `PVM`, `MPI`, `P4`, `Express`.
  - `HPF`, `C*`, `SETL`, `UC`, `ZPL`.

- Some of them are complete languages, others are libraries or **toolkits** that are used inside sequential languages (**pros and cons**).
MPI

- MPI (Message Passing Interface) is a de-facto standard message-passing interface for parallel applications defined since 1992 by a Forum with a participation of over 40 organizations.

- MPI provides a rich set of messaging primitives (129), including point-to-point communication, broadcasting, barrier, and the ability to collect processes in groups and communicate only within each group.

- MPI has been implemented on massively parallel computers, workstation networks, PCs, etc. ~ MPI programs are portable!

- An MPI parallel program is composed of a set of processes running on different processors that use MPI functions for message passing.

---

MPI

- Point-to-Point communication
  
  **MPI_Send** (mess, strlen(mess)+1, type, 1, tag, MPI_COM);
  
  **MPI_Recv** (mess, leng, type, 0, tag, MPI_COM, &status);

- Group communication
  
  **MPI_Bcast** (inbuf, incnt, intype, root, comm);
  
  **MPI_Gather** (outbuf, outcnt, outtype, inbuf, incnt,..);
  
  **MPI_Reduce** (inbuf, outbuf, count, type, op, root, ...)

- As PVM, MPI offers a low-level programming model, but it is widely used for its portability.
MPI

- For program initialize and termination the `MPI_Init` `MPI_Finalize` functions are used.

- **MPI 1 does not make provision for process creation!**

- In the MPI 2 version additional features should be provided for
  - active messages,
  - process startup, and
  - dynamic process creation.

### Sample Code

```c
#include "mpi.h"

main(int argc, char *argv[]) {
    char msg[25] ; int myid, err, np, tag=99; MPI_Status status ;
    err = MPI_Init(&argc, &argv) ; /* MPI start*/
    if (err != MPI_SUCCESS) {
        fprintf(stderr, "error in MPI init\n") ;
        exit(1) ;
    }
    MPI_Comm_size(MPI_COMM_WORLD, &np) ; /* proc num*/
    MPI_Comm_rank(MPI_COMM_WORLD, &myid) ;
    if (myid == 0) /* process 0 */
        { strcpy(msg, "Message to be printed") ;
          MPI_Send(msg, strlen(msg)+1, MPI_CHAR,1,tag, MPI_COMM_WORLD);
        }
    else if (myid == 1) /* process 1 */
        MPI_Recv(msg, 25, MPI_CHAR,0, tag, MPI_COMM_WORLD, &status) ;
    MPI_Finalize() ;
}
```
HPF

- HPF (High Performance Fortran) is the result of an industry/academia/user effort to define a de facto consensus on language extensions for Fortran-90 to improve data locality, especially for distributed-memory parallel computers.

- HPF is a language for programming computationally intensive scientific applications on SIMD, MIMD and vector processors.

- A programmer writes the program in HPF using the Single Program Multiple Data (SPMD) style and provides info about desired data locality or distribution by annotating the code with data-mapping directives.

- The program is then compiled by an architecture-specific compiler that generates the appropriate code optimized for the selected architecture.

HPF

- HPF is based on exploitation of loop parallelism.

- Iterations of the loop body that are conceptually independent can be executed concurrently.

- For example,

  \[
  \text{ForAll (I = 1:N, J = 1:M)} \\
  A(I,J) = I \times B(J) \\
  \text{End ForAll}
  \]

- Data-mapping directives: Align, Distribute.

  \[
  \text{!HPF$ Distribute D2 (Block, Block)} \\
  \text{!HPF$ Align A(I,J) with B(I+2, J+2)}
  \]
The data-parallel C* is an extension of C language that incorporates features of the data SIMD programming model.

C* was designed by Thinking Machines Corp. to program the Connection Machine.

However, C* can be used to program several multicomputers using the data parallel approach.

In this way, each processing element executes, in parallel, the same statement for each instance of the specified data type.

Efficient compilers for C* on MIMD parallel computers are available.

Keywords poly or domain are used to define data to be acted upon in parallel.

The language lets programmer express algorithms as if data could be mapped onto an unbounded number of processors. The compiler automatically maps data onto processing elements.

```c
domain vector (int a, b, min) data[40];

{domain vector]. { if (a < b) min=a;
                           else min=b;
 } 
```

The elements are mapped on different PEs and each PE performs the if-else operation in parallel on different elements of data.
Object-Oriented Parallelism

- An object is a unit that encapsulates private data and a set of associated operations or methods that manipulate the data and define the object behavior.

- In a sequential program, at any time, only one object is active.

- The parallel object-oriented paradigm is obtained by combining the parallelism concepts of process activation and communication with the object-oriented concepts of modularity, data abstraction and inheritance.

- Object modularity makes them a natural unit for parallel execution.

Object-Oriented Parallelism

- Parallelism in object-oriented languages can be exploited in two principal ways:
  - using the objects as the unit of parallelism assigning one or more processes to each object;
  - defining processes as components of the language.

- In the first approach languages are based on active objects. Each process is bound to a particular object for which has been created.

- Intra- and inter-object parallelism can be implemented.

- Languages that use this model are: ABCL/1, Actor model, MPL, Charm++, e Concurrent Aggregates.
Object-Oriented Parallelism

- In the latter approach two different kinds of entities are defined, **objects** and **processes**.

- A process is not bound to a single object, but it is used to perform all the operations required to satisfy an action.

  **THEN**

- A process can execute within many objects changing its address space when an invocation to another object is made.

- Languages based on this model are: HPC++, Argus, Presto, Orca, Nexus and Java.

HPC++

- High Performance C++ is a standard library for parallel programming based on the C++ language.

- The HPC++ consortium consists of people from research groups within Universities, Industry and Government Labs that aim to build a common foundation for constructing portable parallel applications.

  ➔ *as alternative to HPF*

- HPC++ is composed of two levels:
  1. Level 1 consists of a specification for a set of class libraries based on the C++ language.
  2. Level 2 provides the basic language extensions and runtime library needed to implement the full HPC++.
HPC++

- There are two conventional modes of executing an HPC++ program.
- The first is **multi-threaded shared memory** where the program runs within one context.
  - Parallelism comes from the parallel loops and the dynamic creation of threads.
  - This model of programming is very well suited to modest levels of parallelism.
- The second mode of program execution is an **explicit SPMD model** where \( n \) copies of the same program are run on \( n \) different contexts.
  - Parallelism comes from parallel execution of different tasks.
  - This model is well suited for massively parallel computers.

Java (2)

- Java is an object-oriented language that was born for distributed computing, although it embodies a shared memory programming model.
- To develop parallel distributed programs in Java could be used
  - sockets,
  - RMI (Remote Method Invocation),
  - Java + CORBA.
- Java for high performance scientific applications:
  - Java Grande, HPJava.
  - MPIJava, JPVM, JavaPP, JCSP.
Parallel Declarative Paradigms

- **Parallel Functional languages** and **parallel logic** languages are two declarative approaches to parallel programming, concentrating on what is to be done rather than how it is done.

- Programs do not specify in any direct way how they are to be executed in parallel, so that decomposition does not need to be explicit.

- It is still an open question how efficiently these approaches can be implemented.

Parallel Logic Languages

- There are two major forms of parallelism in logic programs are: **AND parallelism** and **OR parallelism**.

- OR parallelism means the parallel evaluation of several clauses whose head unifies with the goal.
  
  If we have the subgoal \(?p(X)\) and the clauses:
  
  \[
  \begin{align*}
  p(X) & :- q(X). \\
  p(X) & :- r(X). \\
  p(X) & :- s(X).
  \end{align*}
  \]

  OR parallelism is exploited by unifying in parallel the subgoal with the head of the three clauses.

- AND parallelism consists of the parallel evaluation of each subgoal that composes the current goal. If the goal to be solved is
  
  \(?p(X), q(Y)\).

  using AND-parallelism, subgoals \(p(X)\) and \(q(Y)\) are solved in parallel.
Parallel Logic Languages

- Parallel logic programming is born from the integration of logic programming and parallel programming to evaluate logic clauses in parallel.

- These models can be divided according to how they exploit parallelism
  - **Explicit parallelism**
    The programmer specifies parallelism in a logic program by annotations.
    *Languages*: PARLOG, Concurrent Prolog, GHC, Delta-Prolog, Strand.
  - **Implicit parallelism**
    Parallelism is extracted both during static analysis and at run-time.
    *Models*: PPP, AND/OR Process model, ANDORRA, Reduce/OR.

Parallel Functional Languages

- Parallel functional languages
  - Multilisp,
  - ParAlfl,
  - SISAL,
  - Concurrent Lisp,
  - *Lisp,
  - Erlang.

- The Multilisp language is an extension of Lisp in which opportunities for parallelism are created using *futures*.

- A future applied to an expression (future (x)) creates a task to evaluate that expression (which begins immediately, that is eagerly).
Sisal

- Sisal (*Streams and Iterations in Single Assignment Language*) is an interesting parallel functional language.

- Most of the parallelism in Sisal programs comes from parallel loops.

- Sisal syntax is very like conventional imperative languages, but the meaning of most statements is different in important ways.

- Sisal is a single-assignment language, so that only a single value can be assigned to each named variable in each scope:

  instructions ⇔ functional expressions

Sisal

- To assign a new value to a variable on the basis of its previous value the keyword *old* must be used

  \[
  \text{for initial} \\
  \text{\hspace{1em} } i:=1; \\
  \text{\hspace{1em} } x:=y[1] \\
  \text{\hspace{1em} while } i<n \text{ repeat} \\
  \text{\hspace{2em} } i:= \text{old } i + 1; \\
  \text{\hspace{2em} } x:= \text{old } x + y[i] \\
  \text{\hspace{2em} returns array of } x \\
  \text{end for ;}
  \]

- Exists a powerful Sisal compiler for shared-memory parallel machines.

- Many Sisal scientific programs have equal or better speedups than equivalent Fortran programs.
ERLANG

- ERLANG (Ericsson language) is a parallel functional language.

- Designed for parallel/distributed Web applications.

- ERLANG creates processes using the operation spawn:

  ```
  say_something(What, 0) ->
      done;
  say_something(What, Times) ->
      io:format("p/n", [What]),
      say_something(What, Times - 1).
  start() ->
      spawn(tut14, say_something, [hello, 3]),
      spawn(tut14, say_something, [goodbye, 3]).
  ```

ERLANG

- Sending messages

  ```
  Fid ! Message
  ```

- Receiving messages

  ```
  receive
  pattern1 ->
      action1;
  pattern2 ->
      action2;
  ....
  patternN
      actionN
  end.
  ```
Skeletons

- **Skeletons** are predefined parallel computation forms.

- They abstract away from issues such as the number of processors in the target architecture, the decomposition of the computation into processes, and communication—structured parallel programming.

- But, cannot be wrote arbitrary programs, but only those that are compositions of the predefined structures.

- Most used skeletons: geometric, farm, divide&conquer, pipeline.

- Models of performance can be obtained.

BSPlib

- The BSP model is not a skeleton model, but it represents a related approach in which the structured operations are single threads.

- Computations are arranged in supersteps, each of which is a collection of these threads.

- A superstep does not begin until all of the global communications initiated in the previous superstep have completed (so that there is an implicit barrier synchronization at the end of each superstep)

  computational results are available to all processes after each superstep

- **Oxford BSPlib** is a toolkit that implements the BSP operations in C and Fortran programs.
Some examples of BSPlib functions are
- `bsp_init(n)` to execute $n$ processes in parallel,
- `bsp_sync` to synchronize processes,
- `bsp_push_reg` to make a local variable readable by other processes,
- `bsp_put e bsp_get` to get and put data to/from another process,
- `bsp_bcast` to send a data towards $n$ processes.

Process mapping and communication patterns are done automatically.
A Critical Classification

- A parallel programming language should
  - be easy to program, by providing mechanisms for
    - Decomposition of a program into parallel threads;
    - Mapping threads to processors;
    - Communication and synchronization among threads.
  - provide a software development methodology;
  - be architecture-independent;
  - have guaranteed performance on different architectures;
  - provide cost measures of programs.

- These requirements are quite demanding and several subsets of them are strongly in tension with each other.

A Critical Classification

- To define a classification of parallel programming languages, Skillicorn and Talia used the level of abstraction that languages provide in terms of
  - implicit or explicit parallelism,
  - implicit or explicit decomposition,
  - implicit or explicit mapping,
  - implicit or explicit communication,
  - implicit or explicit synchronization.

- Within each of the obtained categories we consider languages in which
  - thread structure is dynamic;
  - thread structure is static but communication is not limited;
  - thread structure is static and communication is limited;
A Critical Classification

<table>
<thead>
<tr>
<th>Nature Explicit, Parallelism Implicit</th>
<th>Mapping Explicit, Communication Implicit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic Structure</td>
<td>Dynamic Structure</td>
</tr>
<tr>
<td>Higher-order Functional/Prolog</td>
<td>Higher-order Functional/Prolog</td>
</tr>
<tr>
<td>Concurrent Computing: Omni, Scheme</td>
<td>Concurrent Computing: Omni, Scheme</td>
</tr>
<tr>
<td>Functional Logic Languages: Prolog, Lambda, Haskell</td>
<td>Functional Logic Languages: Prolog, Lambda, Haskell</td>
</tr>
<tr>
<td>Static Structure</td>
<td>Static Structure</td>
</tr>
<tr>
<td>Algorithmic Skeleton: Prolog, LISP</td>
<td>Algorithmic Skeleton: Prolog, LISP</td>
</tr>
<tr>
<td>Future Trends</td>
<td>Future Trends</td>
</tr>
<tr>
<td>From low-level languages to more abstract languages to simplify the task of designers of parallel programs.</td>
<td>From low-level languages to more abstract languages to simplify the task of designers of parallel programs.</td>
</tr>
<tr>
<td>Middle-level models as a tradeoff between abstraction and high performance.</td>
<td>Middle-level models as a tradeoff between abstraction and high performance.</td>
</tr>
<tr>
<td>Architecture-independent models and languages → PORTABILITY.</td>
<td>Architecture-independent models and languages → PORTABILITY.</td>
</tr>
<tr>
<td>Programming environments not just languages.</td>
<td>Programming environments not just languages.</td>
</tr>
<tr>
<td>Unified models of parallel computation.</td>
<td>Unified models of parallel computation.</td>
</tr>
</tbody>
</table>
Comments

- Parallel programming languages support the implementation of high-performance applications in many areas: from the Internet to computational science.

- New models, methods and languages allow users to develop more complex programs with minor efforts.

- Parallel computation can be a technology of the next century if:
  - it will be architecture independent;
  - \( O(\text{parallel programming}) = O(\text{sequential programming}) \);
  - there will be portable and standard parallel software.

Parallel and Distributed Data Mining

- Parallel data mining
  - Task or control parallelism
  - Independent parallelism
  - SPMD parallelism
  - Hybrid parallelism

- Distributed data mining
  - Voting
  - Meta-learning, ensemble learning etc.
Parallel and Distributed Data Mining

- Three main strategies in the exploitation of parallelism in data mining algorithms:
  - independent parallelism
  - control parallelism
  - SPMD parallelism.
- Independent parallelism: processes are executed in parallel in an independent way; generally each process accesses the whole data set.
- Control parallelism (or Task parallelism): each process executes different operations on (a different partition of) a data set.
- SPMD parallelism: a set of processes execute in parallel the same algorithm on different partitions of a data set; processes exchange partial results.

Parallel Data Mining Strategies

- These three basic strategies are not necessarily alternative for parallelizing data mining algorithms.
- They can be combined to improve both performance, scalability and accuracy of results.
- With parallel strategies different data partition strategies can be used
  - Sequential (horizontal/vertical) partitioning
    - separate partitions without overlapping
  - cover-based partitioning
    - some data can be replicated on different partitions
  - range-based query
    - partitions based on some queries that select data according to attribute values.
Parallelism in Data Mining Techniques

- Parallel Decision Trees (Classification)
  - tree construction in parallel (processes ↔ subtrees)
- Discovery of Association Rules in Parallel
  - rule and/or data partitioning on different processors
- Parallel Neural Networks (Classification)
  - parallelism exploitation: training, layers, neurons, weights
- Parallel Cluster Analysis
  - different clustering in parallel, data partitioning, computing similarity matrix in parallel.

Data Classification

- Classification: assigning the items of a data set to predefined classes.
- Using classification we can predict the membership of a data instance to a given class from a set of predefined classes.
- For instance, a set of outlet store clients can be grouped in three classes: high spending, average spending and low spending clients.
- Classification models
  - decision trees, supervised neural networks, regression models, production rules.
Data Classification

Data are partitioned in two sets

- **Training set**: set of examples, where each example is a feature vector (i.e., a set of \((\text{attribute}, \text{value})\) pairs) with its associated class. Model built on this set.

- **Test set**: a set of examples disjoint from the training set, used for testing the accuracy of a model.

Many techniques have been designed to select the best training and test sets.

Decision Trees

- **Classification**: assigning new items to predefined classes.
  - The tree nodes are labeled with the names of attributes,
  - The arcs are labeled with the possible values of the attribute, and
  - The leaves are labeled with the different classes.

- An item is classified by following a path along the tree formed by the arcs corresponding to the values of its attributes.
Parallel Decision Trees

- Task parallel approach
  One process is associated to each sub-tree.
  - The search occurs in parallel in each sub-tree.
  - The degree of parallelism $P$ is equal to the number of active processes at a given time.
  - Processes communicate to exchange information at the border of its region of the tree.

- SPMD approach
  Each process classifies the items of a subset of data.
  - The $P$ processes search in parallel on the whole tree using a partition of the data set $D/P$.
  - The global result is obtained by exchanging partial results.

- The data set partitioning can be operated:
  - partitioning the tuples of the data set: $(D/P)$ per processor.
  - partitioning the $N$ attributes of each tuple: $D$ tuples of $(N/P)$ attributes per processor.
Association Rules

- **Association discovery**: automatic discovery of associations in a data set.
- This task aims at finding sets of items that occur together in records of a data set and the relationships among those items to derive multiple correlations that meet the specified thresholds.
- It is intended to identify strong rules discovered in large data sets using measures of interestingness.

  - Example: “74% of a customers of an on-line bookstore that decide to register, order a book in one week from registration”.

Association Rules

- Given a set of items/attributes, and a set of objects containing a subset of the items
- Find rules: *if A then B* (supp, conf)

  - SUPPORT: percentage of transactions that contains A and B:
    \[ \text{Supp}(A \Rightarrow B) = P(A \cup B) \]
  - CONFIDENCE: percentage of transactions containing A that contains B:
    \[ \text{Conf}(A \Rightarrow B) = P(B \mid A) \]

- User specifies “interestingness”
  - Minimum support (minsup)
  - Minimum confidence (minconf)
- Find all frequent itemsets (> minsup)
Association Rules

<table>
<thead>
<tr>
<th>TID</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bread, Milk</td>
</tr>
<tr>
<td>2</td>
<td>Beer, Diaper, Bread, Eggs</td>
</tr>
<tr>
<td>3</td>
<td>Beer, Coke, Diaper, Milk</td>
</tr>
<tr>
<td>4</td>
<td>Beer, Bread, Diaper, Milk</td>
</tr>
<tr>
<td>5</td>
<td>Coke, Bread, Diaper, Milk</td>
</tr>
</tbody>
</table>

Association Rule: \( X \Rightarrow_{s,\alpha} Y \)

Support: \( s = \frac{\sigma(X \cup Y)}{|T|} \) \((s = \text{P}(X, y))\)

Confidence: \( \alpha = \frac{\sigma(X \cup Y)}{\sigma(X)} (\alpha = \text{P}(y \mid X)) \)

Example: \( \{\text{Diaper, Milk}\} \Rightarrow_{s,\alpha} \text{Beer} \)

\( s = \frac{\sigma(\text{Diaper, Milk, Beer})}{\text{Total Number of Transactions}} = \frac{2}{5} = 0.4 \)

\( \alpha = \frac{\sigma(\text{Diaper, Milk, Beer})}{\sigma(\text{Diaper, Milk})} = \frac{2}{3} = 0.66 \)

Discovery of Association Rules in Parallel

- **SPMD approach**
  - The data set is partitioned among the processors but candidate itemsets are replicated on each processor.
  - All the \( P \) processes count the partial support in parallel of the global itemsets on its local partition of the data set \( D/P \).
  - At the end of this phase the global support is obtained by collecting all local supports.
  - The replication of the candidate itemsets minimizes communication, but do not use memory efficiently.
  - Due to low communication overhead, scalability is good.
Discovery of Association Rules in Parallel

- **Task parallel approach**
  - Both data set and candidate itemsets are partitioned on each processor.
  - Each process $p$ counts the global support of its candidate itemset on the entire data set $D$.
  - To do that, after scanning its local data set partition, a process must scan all remote partitions for each iteration.
  - The partitioning of data set and candidate itemsets minimizes the use of memory but requires high communication overhead (to exchange data partitions).
  - Due to communication overhead this approach is not scalable as the previous one.
Discovery of Association Rules in Parallel

- **Hybrid approaches**
  - Combination of different parallelism approaches can be designed.
  - For example, SPMD and task parallelism can be combined by defining clusters of processors composed of the same number of processing nodes.
    - The data set is partitioned among the clusters, thus each cluster is responsible to compute the partial support of the candidate itemsets according to the SPMD approach.
    - Each processor in a cluster uses the task parallel approach to compute the support of its disjoint set of candidates scanning the data set stored on the processors of its cluster.
    - At the end of each iteration the cluster communicate to compute the global support.

Cluster Analysis

- **Clustering**: arrange data items into several groups, called clusters, so that similar items fall into the same group.

- Items in the same cluster are more similar to each other and items in different clusters tend to be dissimilar, according to some measure of similarity or proximity.

- This is done without any suggestion from an external supervisor, so classes are not given a priori but they must be discovered by the algorithm.
Cluster Analysis

Euclidean Distance Based Clustering in 3-D space.

- Intracluster distances are minimized
- Intercluster distances are maximized

Cluster Analysis

- Clustering algorithms can roughly be classified into the three following types (Everitt 1977): hierarchical, partitional and density-based techniques.
- **Hierarchical methods** generate a nested sequence of clusters by a hierarchical decomposition of a set of $N$ objects represented by a dendrogram. *(BIRCH)*
- **Partitional methods** produce a partation into $K$ clusters by optimizing a criterion. One of the most known criterion is the squared error criterion *(K-means)*.
- **Density search techniques** consider items as points in a metric space and suggest that clusters are those parts of the space characterized by a high density of data. *(E-M)*.
K-means algorithm

Initial seeds

K-means algorithm

New centers
When used to classify large data sets, clustering algorithms are very computing demanding.

Parallelism in clustering algorithms can be exploited both

- In the computation of the similarity or distance among the data items by computing on each processor the distance/similarity of a different partition of items.

- In the clustering strategy.
Parallel Cluster Analysis

- Also in this case three main parallel strategies can be adopted:
  - **Independent parallel:**
    - Each processor uses the whole data set and it performs a different classification based on a different number of clusters.
    - To get the load among the processors balanced a new classification is assigned to a processor that completed its task.

Parallel Cluster Analysis

- **Task parallel approach:**
  - Each processor executes a different task that composes the clustering algorithm and cooperate with other processors exchanging partial results.
  - In partitioning methods processors can work on disjoint regions of the search space using the whole data set.
  - In hierarchical methods a processor can be responsible of one or more clusters.
    1. It finds the nearest neighbor cluster by computing the distance among its cluster and the others.
    2. Then all the local shortest distances are exchanged to find the global shortest distance between two clusters that must be merged.
    3. The new cluster will be assigned to one of the two processors.
Parallel Cluster Analysis

- **SPMD approach**:  
  - Each processor executes the same algorithm on a different partition of the data set to compute partial clustering results.  
  - Local results are then exchanged among all the processors to get global values on every processor.  
  - The global values are used in all processors to start the next clustering step until a convergence is reached or a certain number of steps are executed.  
  - The SPMD strategy can be also used to implement clustering algorithms where each processor generates a local approximation of a model (classification) that at each iteration can be passed to the other processors that can use it to improve their clustering model.

Parallel Clustering algorithms

- **Parallel K-means**  
  - P-Cluster  
  - Several other versions  
- **Parallel k-nearest neighbors**  
- **P-AutoClass**  
- **Parallel Hierarchical Clustering**
**Parallel K-means**

Consider $N$ data items (each one is vector) and $P$ processors.

1. Assign $N/P$ data points to each processor.
2. Node 1 randomly choose $K$ items and assigns them as cluster means (centroids) and broadcast.
3. In each processor for each data item find membership using the cluster means.
4. Recalculate local means for each cluster in each processor.
5. Globally broadcast all local means to every processor so to find the global mean.
6. Go to step (3) and repeat until membership change is less than a threshold $\delta$.

**AutoClass** [Cheeseman, Stutz]

- AutoClass is a clustering algorithm based upon the Bayesian approach for determining optimal classes in large datasets.
- Given a set $X=${$X_1$, ..., $X_n$} of data instances $X_i$ with unknown classes, the goal of Bayesian classification is to search for the best class description that predicts the data in a model space.
- Class membership is expressed probabilistically.
AutoClass

- An instance is not assigned to a unique class, but it has a probability (expressed as weight values) of belonging to each of the possible classes.

- AutoClass calculates the likelihood of each instance belonging to each class \( C \) and then calculates a set of weights \( w_{ij} = \frac{C_i}{\sum_j C_j} \) for each instance.

- Weighted statistics relevant to each term of the class likelihood are calculated for estimating the class model.

AutoClass [Cheeseman, Stutz]

- The classification step is the most computationally intensive. It computes the weights of every instance for each class and computes the parameters of a classification.

- These operations are executed by the function `base_cycle` that calls the three functions
  - `update_wts`,
  - `update_parameters` and
  - `update_approximations`. 
P-AutoClass [Pizzuti, Talia]

- SPMD Parallel implementation of AutoClass
  - data set partitioning among the processors,
  - parallel updating on different processors of
    - weights and
    - parameters of classifications.
P-AutoClass

Figure 4. The parallel version of the update_wts function.

P-AutoClass

Figure 5. The parallel version of the update_parameters function.
Some Final Remarks

- Parallel and distributed data mining (will) play a more and more important role for data analysis and knowledge extraction in several application contexts. Scientific data mining needs parallel techniques.

- **Parallelism and scalability**
  - Millions/Billions of records
  - Thousands of attributes/dimensions
  - Single pass algorithms
  - Parallel I/O and file systems

- **New parallel architectures**
  - Adaptation/definition of parallel algorithms
  - Hundred thousands/Billions of threads
  - Local Data access and processing
Thanks

Questions?