CM40212: Internet Technology

Julian Padget
Tel: x6971, E-mail: jap@cs.bath.ac.uk

Distributed Programming Patterns and Middleware

November 1, 2011
Outline

1 Web programming
   - Applets
   - Servlets
   - Cookies
   - AJAX

2 Distributed programming
   - MIMD
   - SIMD/SPMD

3 Programming frameworks
   - MPI
   - Map Reduce
   - Memcached
   - HADOOP

4 Summary
Objectives

- Establish scope of programming task: not a single thread, not a single computer
- Several complementary frameworks—achieve similar/different goals... but just technology
- Better to understand programming models and their strengths and weaknesses
  - Expressiveness
  - Security
  - Debuggability (?) and maintainability
  - Reliability!
- Task: to recognize the appropriate abstraction to use (or invent)
Objectives

- Establish scope of programming task: not a single thread, not a single computer
- Several complementary frameworks—achieve similar/different goals... but just technology
- Better to understand programming models and their strengths and weaknesses
  - Expressiveness
  - Security
  - Debuggability (?) and maintainability
  - Reliability!
- Task: to recognize the appropriate abstraction to use (or invent)
Objectives

- Establish scope of programming task: not a single thread, not a single computer
- Several complementary frameworks—achieve similar/different goals... but just technology
- Better to understand programming models and their strengths and weaknesses
  - Expressiveness
  - Security
  - Debuggability (?) and maintainability
  - Reliability!
- Task: to recognize the appropriate abstraction to use (or invent)
Objectives

- Establish scope of programming task: not a single thread, not a single computer
- Several complementary frameworks—achieve similar/different goals... but just technology
- Better to understand programming models and their strengths and weaknesses
  - Expressiveness
    - Security
    - Debuggability (?) and maintainability
    - Reliability!
- Task: to recognize the appropriate abstraction to use (or invent)
Objectives

- Establish scope of programming task: not a single thread, not a single computer
- Several complementary frameworks—achieve similar/different goals... but just technology
- Better to understand programming models and their strengths and weaknesses
  - Expressiveness
  - Security
  - Debuggability(?) and maintainability
  - Reliability!
- Task: to recognize the appropriate abstraction to use (or invent)
Objectives

- Establish scope of programming task: not a single thread, not a single computer
- Several complementary frameworks—achieve similar/different goals... but just technology
- Better to understand programming models and their strengths and weaknesses
  - Expressiveness
  - Security
  - Debuggability(?) and maintainability
  - Reliability!

- Task: to recognize the appropriate abstraction to use (or invent)
Objectives

- Establish scope of programming task: not a single thread, not a single computer
- Several complementary frameworks—achieve similar/different goals... but just technology
- Better to understand programming models and their strengths and weaknesses
  - Expressiveness
  - Security
  - Debuggability(?) and maintainability
  - Reliability!

- Task: to recognize the appropriate abstraction to use (or invent)
Objectives

- Establish scope of programming task: not a single thread, not a single computer
- Several complementary frameworks—achieve similar/different goals... but just technology
- Better to understand programming models and their strengths and weaknesses
  - Expressiveness
  - Security
  - Debuggability (?) and maintainability
  - Reliability!
- Task: to recognize the appropriate abstraction to use (or invent)
The coding space

- browser
  - applet
- web server
  - servlet
- databases etc.

or? and?

Condor pool

laptop/desktop

National Grid Service

Local Cluster

Compute Cloud
The coding space

- browser
  - applet
- web server
  - servlet
- databases etc.

or? and?

Condor pool

laptop/desktop

National Grid Service

Local Cluster

Compute Cloud
The coding space

or? and?

Condor pool

Local Cluster

National Grid Service

Compute Cloud

browser

applet

web server

servlet

databases etc.

laptop/desktop

Browser

Applet

Web server

Servlet

Databases etc.
Content

1. Web programming
   - Applets
   - Servlets
   - Cookies
   - AJAX

2. Distributed programming

3. Programming frameworks

4. Summary
Places to put code

- In the browser: applets
- In the server: servlets
- In the back-end: Perl, Python, PHP, Ruby-on-Rails
- Back-end frameworks: Ajax (Backbase, Dojo, Spry)
Places to put code

- In the browser: applets
- In the server: servlets
- In the back-end: Perl, Python, PHP, Ruby-on-Rails
- Back-end frameworks: Ajax (Backbase, Dojo, Spry)
Places to put code

- **In the browser:** applets
- **In the server:** servlets
- **In the back-end:** Perl, Python, PHP, Ruby-on-Rails
- **Back-end frameworks:** Ajax (Backbase, Dojo, Spry)
Places to put code

- In the browser: applets
- In the server: servlets
- In the back-end: Perl, Python, PHP, Ruby-on-Rails
- Back-end frameworks: Ajax (Backbase, Dojo, Spry)
Applets

Applets can use the browser API to:

- Received notification of milestones
- Load data files (specified relative to the URL of the applet or the page in which it is running)
- Display status strings
- Make the browser display a document
- Find other applets running in the same page
- Play sounds
- Access parameters specified by the user in the <APPLET> tag

Constrained environment for security
Applets

Applets can use the browser API to:

- Received notification of milestones
- Load data files (specified relative to the URL of the applet or the page in which it is running)
- Display status strings
- Make the browser display a document
- Find other applets running in the same page
- Play sounds
- Access parameters specified by the user in the <APPLET> tag

**Constrained environment for security**
Applets

Applets can use the browser API to:

- Received notification of milestones
- Load data files (specified relative to the URL of the applet or the page in which it is running)
- Display status strings
  - Make the browser display a document
  - Find other applets running in the same page
  - Play sounds
  - Access parameters specified by the user in the `<APPLET>` tag

**Constrained environment for security**
Applets

Applets can use the browser API to:

- Received notification of milestones
- Load data files (specified relative to the URL of the applet or the page in which it is running)
- Display status strings
- Make the browser display a document
  - Find other applets running in the same page
  - Play sounds
  - Access parameters specified by the user in the `<APPLET>` tag

**Constrained environment for security**
Applets

Applets can use the browser API to:

- Received notification of milestones
- Load data files (specified relative to the URL of the applet or the page in which it is running)
- Display status strings
- Make the browser display a document
- Find other applets running in the same page
- Play sounds
- Access parameters specified by the user in the &lt;APPLET&gt; tag

Constrained environment for security
Applets

Applets can use the browser API to:

- Received notification of milestones
- Load data files (specified relative to the URL of the applet or the page in which it is running)
- Display status strings
- Make the browser display a document
- Find other applets running in the same page
- Play sounds
- Access parameters specified by the user in the <APPLET> tag

**Constrained environment for security**
Applets

Applets can use the browser API to:

- Received notification of milestones
- Load data files (specified relative to the URL of the applet or the page in which it is running)
- Display status strings
- Make the browser display a document
- Find other applets running in the same page
- Play sounds
- Access parameters specified by the user in the <APPLET> tag

**Constrained environment for security**
The Applet Interface

There are four important Applet methods that enable a subclass to handle major events:

1. **init**: To initialize the applet each time it's loaded (or reloaded)
2. **start**: To start the applet's execution, such as when the applet is loaded or when the user revisits a page that contains the applet
3. **stop**: To stop the applet's execution, such as when the user leaves the applet's page or quits the browser
4. **destroy**: To perform a final cleanup in preparation for unloading

For example:

```java
class Simple extends JApplet {
    ...
    public void init() { ... }
    public void start() { ... }
    public void stop() { ... }
    public void destroy() { ... }
    ...
}
```
The Applet Interface

There are four important Applet methods that enable a subclass to handle major events:

1. **init**: To initialize the applet each time it’s loaded (or reloaded)
2. **start**: To start the applet’s execution, such as when the applet is loaded or when the user revisits a page that contains the applet
3. **stop**: To stop the applet’s execution, such as when the user leaves the applet’s page or quits the browser
4. **destroy**: To perform a final cleanup in preparation for unloading

For example:

```java
public class Simple extends JApplet {
    ...
    public void init() { . . . }
    public void start() { . . . }
    public void stop() { . . . }
    public void destroy() { . . . }
    ...
}
```
Security Restrictions

- Applets pose a security risk: downloaded code (from where?) is executed within the browser on client machine
- Different browsers have different security policies and policies change, so cannot hard-code...
- Specifically:
  - Cannot load libraries or define native methods
  - Cannot (normally) read or write files on the client
  - Cannot make network connections except to the originating host
  - Cannot start any program on the client
  - Cannot read arbitrary system properties
  - Applet created windows must look different

- SecurityManager object on browser implements security policies: throws a SecurityException when violation is detected. Applet can catch the SecurityException and process it
Security Restrictions

- Applets pose a security risk: downloaded code (from where?) is executed within the browser on client machine
- Different browsers have different security policies and policies change, so cannot hard-code...

Specifically:
- Cannot load libraries or define native methods
- Cannot (normally) read or write files on the client
- Cannot make network connections except to the originating host
- Cannot start any program on the client
- Cannot read arbitrary system properties
- Applet created windows must look different

SecurityManager object on browser implements security policies: throws a SecurityException when violation is detected. Applet can catch the SecurityException and process it.
Security Restrictions

- Applets pose a security risk: downloaded code (from where?) is executed within the browser on client machine
- Different browsers have different security policies and policies change, so cannot hard-code...
- Specifically:
  - Cannot load libraries or define native methods
  - Cannot (normally) read or write files on the client
  - Cannot make network connections except to the originating host
  - Cannot start any program on the client
  - Cannot read arbitrary system properties
  - Applet created windows must look different

- SecurityManager object on browser implements security policies: throws a SecurityException when violation is detected. Applet can catch the SecurityException and process it
Security Restrictions

- Applets pose a security risk: downloaded code (from where?) is executed within the browser on client machine.
- Different browsers have different security policies and policies change, so cannot hard-code...
- Specifically:
  - Cannot load libraries or define native methods
  - Cannot (normally) read or write files on the client
  - Cannot make network connections except to the originating host
  - Cannot start any program on the client
  - Cannot read arbitrary system properties
  - Applet created windows must look different

- SecurityManager object on browser implements security policies: throws a SecurityException when violation is detected. Applet can catch the SecurityException and process it.
Applet Capabilities

- The `java.applet` package API allows:
  - Creation of network connections to originating host
  - Display of HTML documents in the same browser
  - Invocation of public methods of other applets on the same page
  - Locally-loaded applets are not subject to the same restrictions

- Circumventing the restrictions:
  - Use a server application on the applet’s host
  - Allows saving files—on the applet host rather than the client
  - Communicate over sockets
Applet Capabilities

- The java.applet package API allows:
  - Creation of network connections to originating host
  - Display of HTML documents in the same browser
  - Invocation of public methods of other applets on the same page
  - Locally-loaded applets are not subject to the same restrictions

- Circumventing the restrictions:
  - Use a server application on the applet’s host
  - Allows saving files—on the applet host rather than the client
  - Communicate over sockets
Servlets I

- Supersedes Common Gateway Interface (CGI) scripts
- Servlet is a Java class for extending servers' capabilities via a request-response programming model
- Servlets run in a container on the host: insulates servlets from one another and the host from them
- All servlets must implement the servlet interface
- The servlet life cycle:

  1. An incoming request is mapped to a servlet. If there is no servlet instance:
     - Load the servlet class
     - Create an instance
     - Initialize instance via init method
  
  2. Invoke the service method
  
  3. Servlet removal achieved via destroy method (includes finalization)
Servlets II

- Container typically creates a thread for each request: can ensure *at most* one by implementing `SingleThreadModel` interface

- Otherwise must use synchronized methods and/or synchronized statements:

```java
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

- Does not prevent concurrent access to static variables or external objects
Sessions

- Client state—because HTTP is stateless—e.g. for shopping carts:
  - Interacts with cookies (q.v.)
  - Session is attribute of HttpServletRequest
  - Provides simple key/value association mechanism

- HTTP client cannot signal session is ended, so use a time-to-live counter

- If cookies are disabled, session is encoded in URL
Sessions

- Client state—because HTTP is stateless—e.g. for shopping carts:
  - Interacts with cookies (q.v.)
  - Session is attribute of HttpServletRequest
  - Provides simple key/value association mechanism

- HTTP client cannot signal session is ended, so use a time-to-live counter
  - If cookies are disabled, session is encoded in URL
Sessions

- Client state—because HTTP is stateless—e.g. for shopping carts:
  - Interacts with cookies (q.v.)
  - Session is attribute of HttpServletRequest
  - Provides simple key/value association mechanism

- HTTP client cannot signal session is ended, so use a time-to-live counter

- If cookies are disabled, session is encoded in URL
Cookies I

- Invented to solve problem of persistent data
- Defines three new headers, Cookie, Cookie2, and Set-Cookie2, to carry state information between participating origin servers and user agents:
  1. User Agent → Server
     POST /acme/login HTTP/1.1
     [form data]
     User identifies self via a form.
  2. Server → User Agent
     HTTP/1.1 200 OK
     Set-Cookie2: Customer="WILE_E_COYOTE"; Version="1"; Path="/acme"
     Cookie reflects user’s identity.
Cookies II

3 User Agent → Server

POST /acme/pickitem HTTP/1.1
Cookie: $Version="1"; Customer="WILE_E_COYOTE"; $Path="/acme"

[form data]

User selects an item for "shopping basket".

4 Server → User Agent

HTTP/1.1 200 OK
Set-Cookie2: Part_Number="Rocket_Launcher_0001"; Version="1";
Path="/acme"

Shopping basket contains an item.

5 and so on...

- Minimum implementation requirements:
  - At least 300 cookies
  - At least 4096 bytes per cookie
  - At least 20 cookies per unique host or domain name

- Applications should use as few and as small cookies as possible, and they should cope gracefully with the loss of a cookie.
Asynchronous JavaScript and XML

- Main feature: allows web applications to retrieve data from the server asynchronously in the background without interfering with the display and behavior of the existing page.

- Uses XMLHttpRequest object or Remote Scripting

- Does not require Javascript, XML or asynchronism

Pros:

- Reduction in client-server traffic: only send changes, not fresh pages
- Better (perceived) response time
- Reduction in server connections

Cons:

- Dynamically-created pages ↞ browser “back” button
- Difficulty in book-marking
- Difficulty with indexing
- Accessibility restrictions—e.g. Javascript disabled
- Absence of a standard
Asynchronous JavaScript and XML

- Main feature: allows web applications to retrieve data from the server asynchronously in the background without interfering with the display and behavior of the existing page.
- Uses XMLHttpRequest object or Remote Scripting
- Does not require Javascript, XML or asynchronism
- Pros:
  - Reduction in client-server traffic: only send changes, not fresh pages
  - Better (perceived) response time
  - Reduction in server connections
- Cons:
  - Dynamically-created pages do not "back" button
  - Difficulty in bookmarking
  - Difficulty with indexing
  - Accessibility restrictions—e.g. Javascript disabled
  - Absence of a standard
Asynchronous JavaScript and XML

- Main feature: allows web applications to retrieve data from the server asynchronously in the background without interfering with the display and behavior of the existing page.
- Uses XMLHttpRequest object or Remote Scripting
- Does not require Javascript, XML or asynchronism

Pros:
- Reduction in client-server traffic: only send changes, not fresh pages
- Better (perceived) response time
- Reduction in server connections

Cons:
- Dynamically-created pages ↔ browser “back” button
- Difficulty in book-marking
- Difficulty with indexing
- Accessibility restrictions—e.g. Javascript disabled
- Absence of a standard
Asynchronous JavaScript and XML

- Main feature: allows web applications to retrieve data from the server asynchronously in the background without interfering with the display and behavior of the existing page.
- Uses XMLHttpRequest object or Remote Scripting
- Does not require Javascript, XML or asynchronism

Pros:
- Reduction in client-server traffic: only send changes, not fresh pages
- Better (perceived) response time
- Reduction in server connections

Cons:
- Dynamically-created pages \(\n\) browser “back” button
- Difficulty in book-marking
- Difficulty with indexing
- Accessibility restrictions—e.g. Javascript disabled
- Absence of a standard
Asynchronous JavaScript and XML

- Main feature: allows web applications to retrieve data from the server asynchronously in the background without interfering with the display and behavior of the existing page.
- Uses XMLHttpRequest object or Remote Scripting
- Does not require Javascript, XML or asynchronism
- Pros:
  - Reduction in client-server traffic: only send changes, not fresh pages
  - Better (perceived) response time
  - Reduction in server connections
- Cons:
  - Dynamically-created pages $\n\leftarrow$ browser “back” button
  - Difficulty in book-marking
  - Difficulty with indexing
  - Accessibility restrictions—e.g. Javascript disabled
  - Absence of a standard
Content

1 Web programming

2 Distributed programming
   - MIMD
   - SIMD/SPMD

3 Programming frameworks

4 Summary
Places to put code

- Distribute function across network
- Distribute data across network
- Control: synchronous or asynchronous
Places to put code

- Distribute function across network
- Distribute data across network
- Control: synchronous or asynchronous
Places to put code

- Distribute function across network
- Distribute data across network
- Control: synchronous or asynchronous
Concurrency and Parallelism

- Programs components can appear to, or actually execute at the same time → new ways for making mistakes:
  1. Synchronization:
     - Deadlock: suspended waiting for resources
     - Livelock: repeatedly checking for resources
  2. Race conditions
  3. Non-determinism

- Flynn’s classification identifies four forms:

<table>
<thead>
<tr>
<th>SISD</th>
<th>SIMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Instruction</td>
<td>Single Instruction</td>
</tr>
<tr>
<td>Single Datum</td>
<td>Multiple Data</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MISD</th>
<th>MIMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple Instruction</td>
<td>Multiple Instruction</td>
</tr>
<tr>
<td>Single Datum</td>
<td>Multiple Data</td>
</tr>
</tbody>
</table>

Julian Padget (CS/Bath)
Concurrencies and Parallelism

- Programs components can appear to, or actually execute at the same time → new ways for making mistakes:
  1. Synchronization:
     - Deadlock: suspended waiting for resources
     - Livelock: repeatedly checking for resources
  2. Race conditions
  3. Non-determinism

- Flynn’s classification identifies four forms:

<table>
<thead>
<tr>
<th>SISD</th>
<th>SIMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Instruction Single Datum</td>
<td>Single Instruction Multiple Data</td>
</tr>
<tr>
<td>MISD</td>
<td>MIMD</td>
</tr>
<tr>
<td>Multiple Instruction Single Datum</td>
<td>Multiple Instruction Multiple Data</td>
</tr>
</tbody>
</table>
Basic concurrency concepts

- What makes it different from sequential computation?
  - non-determinism:
    - computations proceed at different rates
    - communications vary in speed and reliability
  - communication: channels, messages, shared variables
  - interference: contention for resources
  - atomicity: several (interdependent) steps (must) appear as one and succeed, or fail and all be undone (c.f. finally etc.)
  - cooperation vs. pre-emption: voluntary ceding of control (suspend) or time-slicing
  - coordination: mechanisms to balance cooperation or competition between tasks while limiting interference
Basic concurrency concepts

What makes it different from sequential computation?

- non-determinism:
  - computations proceed at different rates
  - communications vary in speed and reliability

- communication: channels, messages, shared variables
- interference: contention for resources
- atomicity: several (interdependent) steps (must) appear as one and succeed, or fail and all be undone (c.f. finally etc.)
- cooperation vs. pre-emption: voluntary ceding of control (suspend) or time-slicing
- coordination: mechanisms to balance cooperation or competition between tasks while limiting interference
Basic concurrency concepts

- What makes it different from sequential computation?
  - non-determinism:
    - computations proceed at different rates
    - communications vary in speed and reliability
  - communication: channels, messages, shared variables
    - interference: contention for resources
    - atomicity: several (interdependent) steps (must) appear as one and succeed, or fail and all be undone (c.f. finally etc.)
  - cooperation vs. pre-emption: voluntary ceding of control (suspend) or time-slicing
  - coordination: mechanisms to balance cooperation or competition between tasks while limiting interference
Basic concurrency concepts

- What makes it different from sequential computation?
  - non-determinism:
    - computations proceed at different rates
    - communications vary in speed and reliability
  - communication: channels, messages, shared variables
  - interference: contention for resources
    - atomicity: several (interdependent) steps (must) appear as one and succeed, or fail and all be undone (c.f. finally etc.)
    - cooperation vs. pre-emption: voluntary ceding of control (suspend) or time-slicing
    - coordination: mechanisms to balance cooperation or competition between tasks while limiting interference
Basic concurrency concepts

- What makes it different from sequential computation?
  - non-determinism:
    - computations proceed at different rates
    - communications vary in speed and reliability
  - communication: channels, messages, shared variables
  - interference: contention for resources
  - atomicity: several (interdependent) steps (must) appear as one and succeed, or fail and all be undone (c.f. finally etc.)
  - cooperation vs. pre-emption: voluntary ceding of control (suspend) or time-slicing
  - coordination: mechanisms to balance cooperation or competition between tasks while limiting interference
Basic concurrency concepts

■ What makes it different from sequential computation?

■ non-determinism:
  ■ computations proceed at different rates
  ■ communications vary in speed and reliability
■ communication: channels, messages, shared variables
■ interference: contention for resources
■ atomicity: several (interdependent) steps (must) appear as one and succeed, or fail and all be undone (c.f. finally etc.)
■ cooperation vs. pre-emption: voluntary ceding of control (suspend) or time-slicing
■ coordination: mechanisms to balance cooperation or competition between tasks while limiting interference
Basic concurrency concepts

- What makes it different from sequential computation?
  - non-determinism:
    - computations proceed at different rates
    - communications vary in speed and reliability
  - communication: channels, messages, shared variables
  - interference: contention for resources
  - atomicity: several (interdependent) steps (must) appear as one and succeed, or fail and all be undone (c.f. finally etc.)
  - cooperation vs. pre-emption: voluntary ceding of control (suspend) or time-slicing
  - coordination: mechanisms to balance cooperation or competition between tasks while limiting interference
Concurrent patterns

- Concurrent programs frequently follow one or a mixture of a few well-known patterns:
  - **producer/consumer**: one thread generates data for another – communication channel becomes source of contention
  - **pipeline**: general case of producer/consumer
  - **master/slave**: break task into independent parts and distribute, collect and combine results
  - **divide and conquer**: general case of above, where slaves become masters of smaller parts

- Essential coordination aspect is access to shared data:
  - **Aim**: only one task can change data at one time
  - **But**: many can access data – consistency problem
  - **Solution**: mechanisms for *mutual exclusion*
Concurrent programs frequently follow one or a mixture of a few well-known patterns:

- **producer/consumer**: one thread generates data for another – communication channel becomes source of contention
- **pipeline**: general case of producer/consumer
- **master/slave**: break task into independent parts and distribute, collect and combine results
- **divide and conquer**: general case of above, where slaves become masters of smaller parts

Essential coordination aspect is access to shared data:

- Aim: only one task can change data at one time
- But: many can access data – consistency problem
- Solution: mechanisms for *mutual exclusion*
Object spaces

- Low-level synchronization mechanisms are (very) difficult to use safely
  - (A) Solution: Decouple sender and receiver
    - temporally – need not co-exist
    - spatially – need no information about each other
  - Origins:
    - Linda [Carriero and Gelernter, 1989]
    - Chemical Abstract Machine (CHAM) [Berry and Boudol, 1990]
    - Unity [Chandy and Misra, 1988]
  - Conceptually:
Object spaces

- Low-level synchronization mechanisms are (very) difficult to use safely
- (A) Solution: Decouple sender and receiver
  - temporally – need not co-exist
  - spatially – need no information about each other

Origins:
- Linda [Carriero and Gelernter, 1989]
- Chemical Abstract Machine (CHAM) [Berry and Boudol, 1990]
- Unity [Chandy and Misra, 1988]

Conceptually:

```
T_n

message/tuple pool

T_1

T_{n-1}

T_2
```
Object spaces

- Low-level synchronization mechanisms are (very) difficult to use safely
- (A) Solution: Decouple sender and receiver
  - temporally – need not co-exist
  - spatially – need no information about each other

Origins:
- Linda [Carriero and Gelernter, 1989]
- Chemical Abstract Machine (CHAM) [Berry and Boudol, 1990]
- Unity [Chandy and Misra, 1988]

Conceptually:
Object spaces

- Low-level synchronization mechanisms are (very) difficult to use safely
- (A) Solution: Decouple sender and receiver
  - temporally – need not co-exist
  - spatially – need no information about each other
- Origins:
  - Linda [Carriero and Gelernter, 1989]
  - Chemical Abstract Machine (CHAM) [Berry and Boudol, 1990]
  - Unity [Chandy and Misra, 1988]
- Conceptually:

```
T_n

message/tuple pool

T_1

T_{n-1}

T_2
```
Object space operations

- Operations in Linda (cf. Javaspaces [Freeman et al., 1999]):
  - `in(pattern)`: where pattern = `tag(elt1, ..., elt_n)`, tag is a literal, elt is constant or a variable
    - action: match pattern against tuples in pool, select one (if exists) and remove, otherwise block
  - `out(tagged tuple)`: constructs tuple and stores in pool
  - `rd(pattern)`: as input but fails instead of blocking if no match. ≡ testing a semaphore before waiting...
    - equally useless (race condition).
  - `eval(expression)`: creates new tasks to evaluate expression, result is written to pool.

- Conceptually attractive, easy to learn, easy to use, but can be problematic in practice.

Object space operations

- Operations in Linda (cf. Javaspaces [Freeman et al., 1999]):
  - `in(pattern)`: where pattern = `tag(elt1,...,eltn)`, tag is a literal, elt is constant or a variable action: match pattern against tuples in pool, select one (if exists) and remove, otherwise block
  - `out(tagged tuple)`: constructs tuple and stores in pool
  - `rd(pattern)`: as input but fails instead of blocking if no match. ≡ testing a semaphore before waiting... equally useless (race condition).
  - `eval(expression)`: creates new tasks to evaluate expression, result is written to pool.

- Conceptually attractive, easy to learn, easy to use, but can be problematic in practice.

Object space operations

- Operations in Linda (cf. Javaspaces [Freeman et al., 1999]):
  - `in(pattern)`: where pattern = `tag(elt1,...,eltn)`, tag is a literal, elt is constant or a variable
    action: match pattern against tuples in pool, select one (if exists) and remove, otherwise block
  - `out(tagged tuple)`: constructs tuple and stores in pool
  - `rd(pattern)`: as input but fails instead of blocking if no match. ≡ testing a semaphore before waiting...
    equally useless (race condition).
  - `eval(expression)`: creates new takes to evaluate expression, result is written to pool.

- Conceptually attractive, easy to learn, easy to use, but can be problematic in practice.

Data Parallelism

- SIMD: single instruction multiple data; requires special hardware (or data abstraction)

- History:
  - vector processing – 1970s, 80s, 90s. Manufacturers: Control Data Corporation (CDC), Cray. Vector extensions to FORTRAN.
  - array processors – 1980s, 90s. Manufacturers: ICL (Distributed Array Processor), Thinking Machines Connection Machines, Maspar. Architecture: thousands of small processors (e.g. 1 bit ALU, 16K memory), local + global connection network. Array extensions to FORTRAN.
  - Languages: C*, FORTRAN*, *Lisp

- SPMD: single program multiple data; can be realized on a Condor pool, commodity hardware cluster or compute cloud
Data Parallelism

- SIMD: single instruction multiple data; requires special hardware (or data abstraction)

- History:
  - vector processing – 1970s, 80s, 90s. Manufacturers: Control Data Corporation (CDC), Cray. Vector extensions to FORTRAN.
  - array processors – 1980s, 90s. Manufacturers: ICL (Distributed Array Processor), Thinking Machines Connection Machines, Maspar. Architecture: thousands of small processors (e.g. 1 bit ALU, 16K memory), local+global connection network. Array extensions to FORTRAN.
  - Languages: C*, FORTRAN*, *Lisp

- SPMD: single program multiple data; can be realized on a Condor pool, commodity hardware cluster or compute cloud
Data Parallelism

- SIMD: single instruction multiple data; requires special hardware (or data abstraction)

- History:
  - vector processing – 1970s, 80s, 90s. Manufacturers: Control Data Corporation (CDC), Cray. Vector extensions to FORTRAN.
  - array processors – 1980s, 90s. Manufacturers: ICL (Distributed Array Processor), Thinking Machines Connection Machines, Maspar. Architecture: thousands of small processors (e.g. 1 bit ALU, 16K memory), local+global connection network. Array extensions to FORTRAN.
  - Languages: C*, FORTRAN*, *Lisp

- SPMD: single program multiple data; can be realized on a Condor pool, commodity hardware cluster or compute cloud
Content

1 Web programming

2 Distributed programming

3 Programming frameworks
   - MPI
   - Map Reduce
   - Memcached
   - HADOOP

4 Summary
MPI summary I

- De facto standard for SPMD programming
- Point-to-point and collective communications
- MPI belongs in layers 5 and higher (session, presentation, application) of the OSI Reference Model
- Implementations cover most layers of the reference model, with socket and TCP being used in the transport layer
- Functionality:
  - Virtual topology
  - Synchronization
  - Communication between a set of processes that have been mapped to nodes/servers/computer instances
- API extends several widely-used languages
MPI summary II

- Features:
  - Point-to-point rendezvous-type send/receive operations—synchronous, asynchronous, buffered, and ready forms
  - Choice of Cartesian or graph-like logical process topology
  - Exchange of data between process pairs (send/receive operations)
  - Combination of partial results of computations (gathering and reduction operations)
  - Synchronization of nodes (barrier operation)
  - Environmental enquiry operations (number of processes in session, processor identity, neighbouring processes, etc.)
**MPI**

- Primary user is e-Science (teraflops, terabytes)
- Distributed systems + small programs = slow
- Message overheads > gains from parallelism
- MPI supports SPMD + message-passing
- Functionality:
  - library of messaging functions
  - works with (unmodified) C, Fortran, etc.
  - even Java, Python and others
  - MPI is a standard not an implementation
- Example:
MPI

- Primary user is e-Science (teraflops, terabytes)
- Distributed systems + small programs = slow
  - Message overheads > gains from parallelism
- MPI supports SPMD + message-passing
  - Functionality:
    - library of messaging functions
    - works with (unmodified) C, Fortran, etc.
    - even Java, Python and others
    - MPI is a standard not an implementation
- Example:
MPI

- Primary user is e-Science (teraflops, terabytes)
- Distributed systems + small programs = slow
- Message overheads $>$ gains from parallelism
- MPI supports SPMD + message-passing

Functionality:
- library of messaging functions
- works with (unmodified) C, Fortran, etc.
- even Java, Python and others
- MPI is a standard not an implementation

Example:
MPI

- Primary user is e-Science (teraflops, terabytes)
- Distributed systems + small programs = slow
- Message overheads > gains from parallelism
- MPI supports SPMD + message-passing

  - Functionality:
    - library of messaging functions
    - works with (unmodified) C, Fortran, etc.
    - even Java, Python and others
    - MPI is a standard not an implementation

- Example:
MPI

- Primary user is e-Science (teraflops, terabytes)
- Distributed systems + small programs = slow
- Message overheads > gains from parallelism
- MPI supports SPMD + message-passing
- Functionality:
  - library of messaging functions
  - works with (unmodified) C, Fortran, etc.
  - even Java, Python and others
  - MPI is a standard not an implementation

Example:
MPI

- Primary user is e-Science (teraflops, terabytes)
- Distributed systems + small programs = slow
- Message overheads > gains from parallelism
- MPI supports SPMD + message-passing
- Functionality:
  - library of messaging functions
  - works with (unmodified) C, Fortran, etc.
  - even Java, Python and others
  - MPI is a standard not an implementation
- Example:
#include <stdio.h>
#include <mpi.h>

int main(int argc, char **argv)
{
    int rc, myrank, nproc, namelen;
    char name[128];

    rc = MPI_Init(&argc, &argv);
    if (rc != MPI_SUCCESS) {
        printf("Error starting MPI program\n");
        MPI_Abort(MPI_COMM_WORLD, rc);
    }

    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);

    if (myrank == 0) {
        printf("main reports %d procs\n", nproc);
    }

    namelen = 128;
    MPI_Get_processor_name(name, &namelen);
    printf("hello world %d from 's'\n", myrank);

    MPI_Finalize();
    return 0;
}
**MPI**

Notes:

- **MPI_Init(&argc, &argv);** – system set up; creates all the network connections;
- **rc** – check it worked
- **MPI_COMM_WORLD** – system can be divided into subsets of processors called *communicators*:
  - The *WORLD* communicator is all processors
  - **MPI_COMM_SELF** refers to the calling processor
- **MPI_Comm_rank** – each process in a communicator has a unique rank \([0, \text{(size of the communicator } - 1)]\)
- For *WORLD* the rank is 0 to \((\text{# of processors } - 1)\)
- **MPI_Comm_size** – Size of the communicator
- **if (myrank == i)** – All processors run the same code (SPMD). This is how to get a kind of MIMD
- **MPI_Finalize** – Tidy up. Forces receipt of messages. Establishes a barrier synchronization.
MPI

A basic problem is how to get data from one processor to another

Processor A sends data (integers, floats, strings, etc.) to B

A uses a send function, and B uses a receive function

```c
int n[5];
...

if (myrank == 0) {
    MPI_Send(n, 5, MPI_INT, 1, 99, MPI_COMM_WORLD);
}
else if (myrank == 1) {
    MPI_Status stat;
    MPI_Recv(n, 5, MPI_INT, 0, 99, MPI_COMM_WORLD, &stat);
}
```

This supposes A has rank 0, B rank 1
A basic problem is how to get data from one processor to another

Processor A sends data (integers, floats, strings, etc.) to B

A uses a send function, and B uses a receive function

```c
int n[5];
...
if (myrank == 0) {
    MPI_Send(n, 5, MPI_INT, 1, 99, MPI_COMM_WORLD);
}
else if (myrank == 1) {
    MPI_Status stat;
    MPI_Recv(n, 5, MPI_INT, 0, 99, MPI_COMM_WORLD, &stat);
}
```

This supposes A has rank 0, B rank 1
MPI

- A basic problem is how to get data from one processor to another
- Processor A sends data (integers, floats, strings, etc.) to B
- A uses a *send* function, and B uses a *receive* function

```c
int n[5];
...
if (myrank == 0) {
    MPI_Send(n, 5, MPI_INT, 1, 99, MPI_COMM_WORLD);
}
else if (myrank == 1) {
    MPI_Status stat;
    MPI_Recv(n, 5, MPI_INT, 0, 99, MPI_COMM_WORLD, &stat);
}
```

- This supposes A has rank 0, B rank 1
MPI

- A basic problem is how to get data from one processor to another
- Processor A sends data (integers, floats, strings, etc.) to B
- A uses a send function, and B uses a receive function

```c
int n[5];
...
if (myrank == 0) {
    MPI_Send(n, 5, MPI_INT, 1, 99, MPI_COMM_WORLD);
}
else if (myrank == 1) {
    MPI_Status stat;
    MPI_Recv(n, 5, MPI_INT, 0, 99, MPI_COMM_WORLD, &stat);
}
```

- This supposes A has rank 0, B rank 1
MPI

A sends

- \( n \) – A pointer to a memory location containing the data; can be a single variable or an array of values
- 5 – The number of items to send
- \( MPI\_\text{INT} \) – The type of the items
- 1 – The rank of the destination
- 99 – An integer \( tag \) to help identify a particular communication in the message logs...
- \( MPI\_\text{COMM\_WORLD} \) – The communicator
MPI

B receives

- \( n \) – A pointer to a memory location from where to read the data
- 5 – The number of items to read
- MPI_INT – The type of the items
- 0 – The rank of the source
- 99 – The tag on the message receiver wants: use MPI_ANY_TAG otherwise
- MPI_COMM_WORLD – The communicator
- stat – A structure containing the status of the transfer, in particular the source and tag; and the error type in case of an error
MPI

B receives

- \( \text{n} \) – A pointer to a memory location from where to read the data
- 5 – The number of items to read
  - MPI_INT – The type of the items
  - 0 – The rank of the source
  - 99 – The tag on the message receiver wants: use MPI_ANY_TAG otherwise
  - MPI_COMM_WORLD – The communicator
  - stat – A structure containing the status of the transfer, in particular the source and tag; and the error type in case of an error
MPI

B receives

- \( n \) – A pointer to a memory location from where to read the data
- 5 – The number of items to read
- MPI_INT – The type of the items
- 0 – The rank of the source
- 99 – The \( \text{tag} \) on the message receiver wants: use MPI_ANY_TAG otherwise

- MPI_COMM_WORLD – The communicator
- stat – A structure containing the status of the transfer, in particular the source and tag; and the error type in case of an error
B receives

- \( n \) – A pointer to a memory location from where to read the data
- 5 – The number of items to read
- `MPI_INT` – The type of the items
- 0 – The rank of the source
- 99 – The `tag` on the message receiver wants: use `MPI_ANY_TAG` otherwise
- `MPI_COMM_WORLD` – The communicator
- `stat` – A structure containing the status of the transfer, in particular the source and tag; and the error type in case of an error
MPI

B receives

- n – A pointer to a memory location from where to read the data
- 5 – The number of items to read
- MPI_INT – The type of the items
- 0 – The rank of the source
- 99 – The tag on the message receiver wants: use MPI_ANY_TAG otherwise

- MPI_COMM_WORLD – The communicator
- stat – A structure containing the status of the transfer, in particular the source and tag; and the error type in case of an error
**MPI**

B receives

- n – A pointer to a memory location from where to read the data
- 5 – The number of items to read
- MPI_INT – The type of the items
- 0 – The rank of the source
- 99 – The *tag* on the message receiver wants: use MPI_ANY_TAG otherwise
- MPI_COMM_WORLD – The communicator

- stat – A structure containing the status of the transfer, in particular the source and tag; and the error type in case of an error
MPI

B receives

- $n$ – A pointer to a memory location from where to read the data
- 5 – The number of items to read
- MPI_INT – The type of the items
- 0 – The rank of the source
- 99 – The tag on the message receiver wants: use MPI_ANY_TAG otherwise

- MPI_COMM_WORLD – The communicator
- stat – A structure containing the status of the transfer, in particular the source and tag; and the error type in case of an error
MPI

- Types include:
  
  ```
  MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG, 
  MPI_FLOAT, MPI_DOUBLE, MPI_BYTE
  ```

- MPI_Send and MPI_Recv are *blocking* operations:
  - MPI_Send waits until the data has been sent.
  - The buffer n in A can be safely reused when MPI_Send returns.
  - But the data may not have reached or been read by B.

- Similarly, MPI_Recv waits until all data is copied.
- This provides weak synchronisation between A and B.
- But *asynchronism* requires careful programming.
- Messages take a significant time to be transmitted.
- A send and a receive are not simultaneous.
MPI

- Types include:
  - MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG,
  - MPI_FLOAT, MPI_DOUBLE, MPI_BYTE

- MPI_Send and MPI_Recv are blocking operations:
  - MPI_Send waits until the data has been sent
  - The buffer in A can be safely reused when MPI_Send returns
  - But the data may not have reached or been read by B

- Similarly, MPI_Recv waits until all data is copied
- This provides weak synchronisation between A and B
- But asynchronism requires careful programming
- Messages take a significant time to be transmitted
- A send and a receive are not simultaneous
MPI

- Types include:
  - `MPI_CHAR`, `MPI_SHORT`, `MPI_INT`, `MPI_LONG`, `MPI_FLOAT`, `MPI_DOUBLE`, `MPI_BYTE`

- `MPI_Send` and `MPI_Recv` are *blocking* operations:
  - `MPI_Send` waits until the data has been sent
  - The buffer `n` in A can be safely reused when `MPI_Send` returns
  - But the data may not have reached or been read by B
  - Similarly, `MPI_Recv` waits until the all data is copied
  - This provides weak synchronisation between A and B
  - But *asynchronism* requires careful programming
  - Messages take a significant time to be transmitted
  - A send and a receive are not simultaneous
MPI

- Types include:
  
  \[
  \text{MPI\_CHAR, MPI\_SHORT, MPI\_INT, MPI\_LONG, MPI\_FLOAT, MPI\_DOUBLE, MPI\_BYTE}
  \]

- \texttt{MPI\_Send} and \texttt{MPI\_Recv} are \textit{blocking} operations:
  - \texttt{MPI\_Send} waits until the data has been sent
  - The buffer \( n \) in A can be safely reused when \texttt{MPI\_Send} returns
  - But the data may not have reached or been read by B
  - Similarly, \texttt{MPI\_Recv} waits until the all data is copied
  - This provides weak synchronisation between A and B
  - But \textit{asynchronism} requires careful programming
  - Messages take a significant time to be transmitted
  - A send and a receive are not simultaneous
MPI

- Types include:
  
  \texttt{MPI\_CHAR, MPI\_SHORT, MPI\_INT, MPI\_LONG, MPI\_FLOAT, MPI\_DOUBLE, MPI\_BYTE}

- \texttt{MPI\_Send} and \texttt{MPI\_Recv} are \textit{blocking} operations:
  - \texttt{MPI\_Send} waits until the data has been sent
  - The buffer \texttt{n} in A can be safely reused when \texttt{MPI\_Send} returns
    - But the data may not have reached or been read by B
  - Similarly, \texttt{MPI\_Recv} waits until the all data is copied
  - This provides weak synchronisation between A and B
  - But \textit{asynchronism} requires careful programming
  - Messages take a significant time to be transmitted
  - A send and a receive are not simultaneous
MPI

- Types include:
  - MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG,
  - MPI_FLOAT, MPI_DOUBLE, MPI_BYTE

- MPI_Send and MPI_Recv are *blocking* operations:
  - MPI_Send waits until the data has been sent
  - The buffer \( n \) in A can be safely reused when MPI_Send returns
  - But the data may not have reached or been read by B

- Similarly, MPI_Recv waits until the all data is copied
- This provides weak synchronisation between A and B
- But *asynchronism* requires careful programming
- Messages take a significant time to be transmitted
- A send and a receive are not simultaneous
MPI

- Types include:
  
  ```
  MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG, 
  MPI_FLOAT, MPI_DOUBLE, MPI_BYTE
  ```

- **MPI_Send** and **MPI_Recv** are *blocking* operations:
  - **MPI_Send** waits until the data has been sent
  - The buffer `n` in A can be safely reused when **MPI_Send** returns
  - But the data may not have reached or been read by B

- Similarly, **MPI_Recv** waits until the all data is copied
  - This provides weak synchronisation between A and B
  - But *asynchronism* requires careful programming
  - Messages take a significant time to be transmitted
  - A send and a receive are not simultaneous
MPI

- Types include:
  
  ```
  MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG, MPI_FLOAT, MPI_DOUBLE, MPI_BYTE
  ```

- ```MPI_Send``` and ```MPI_Recv``` are *blocking* operations:
  - ```MPI_Send``` waits until the data has been sent
  - The buffer in A can be safely reused when ```MPI_Send``` returns
  - But the data may not have reached or been read by B

- Similarly, ```MPI_Recv``` waits until all data is copied
- This provides weak synchronisation between A and B

  - But *asynchronism* requires careful programming
  - Messages take a significant time to be transmitted
  - A send and a receive are not simultaneous
MPI

- Types include:
  - MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG, MPI_FLOAT, MPI_DOUBLE, MPI_BYTE

- MPI_Send and MPI_Recv are blocking operations:
  - MPI_Send waits until the data has been sent
  - The buffer $n$ in A can be safely reused when MPI_Send returns
  - But the data may not have reached or been read by B

- Similarly, MPI_Recv waits until the all data is copied
- This provides weak synchronisation between A and B
- But asynchronism requires careful programming
  - Messages take a significant time to be transmitted
  - A send and a receive are not simultaneous
**MPI**

- Types include:
  
  - `MPI_CHAR`, `MPI_SHORT`, `MPI_INT`, `MPI_LONG`,
  - `MPI_FLOAT`, `MPI_DOUBLE`, `MPI_BYTE`

- `MPI_Send` and `MPI_Recv` are *blocking* operations:
  - `MPI_Send` waits until the data has been sent
  - The buffer `n` in `A` can be safely reused when `MPI_Send` returns
  - But the data may not have reached or been read by `B`

- Similarly, `MPI_Recv` waits until the all data is copied
- This provides weak synchronisation between `A` and `B`
- But *asynchronism* requires careful programming
- Messages take a significant time to be transmitted
- A send and a receive are not simultaneous
MPI

- Types include:
  
  ```
  MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG, 
  MPI_FLOAT, MPI_DOUBLE, MPI_BYTE
  ```

- **MPI_Send** and **MPI_Recv** are *blocking* operations:
  - **MPI_Send** waits until the data has been sent
  - **MPI_Recv** waits until the all data is copied
  - The buffer `n` in A can be safely reused when **MPI_Send** returns
  - But the data may not have reached or been read by B
  - Similarly, **MPI_Recv** waits until the all data is copied
  - This provides weak synchronisation between A and B
  - But *asynchronism* requires careful programming
  - Messages take a significant time to be transmitted
  - A send and a receive are not simultaneous
Other communication operators include:

- `MPI_Ssend` – Waits until the destination has *started* to receive the message
- `MPI_Isend` – Send, but do not wait – how soon can buffer be (re-)used?
- `MPI_Irecv` – Indicate a buffer for data storage, but do not wait for it; the data transferred asynchronously
- `MPI_Wait` – Block until a given non-blocking send or recv has completed
- And lots more
Other communication operators include:

- **MPI_Ssend** – Waits until the destination has *started* to receive the message

- **MPI_Isend** – Send, but do not wait – how soon can buffer be (re-)used?

- **MPI_Irecv** – Indicate a buffer for data storage, but do not wait for it; the data transferred asynchronously

- **MPI_Wait** – Block until a given non-blocking send or recv has completed

- And lots more
Other communication operators include:

- **MPI_Ssend** – Waits until the destination has *started* to receive the message
- **MPI_Isend** – Send, but do not wait – how soon can buffer be (re-)used?
- **MPI_Irecv** – Indicate a buffer for data storage, but do not wait for it; the data transferred asynchronously
- **MPI_Wait** – Block until a given non-blocking send or recv has completed
- And lots more
Other communication operators include:

- **MPI_Ssend** – Waits until the destination has *started* to receive the message

- **MPI_Isend** – Send, but do not wait – how soon can buffer be (re-)used?

- **MPI_Irecv** – Indicate a buffer for data storage, but do not wait for it; the data transferred asynchronously

- **MPI_Wait** – Block until a given non-blocking send or recv has completed

- And lots more
MPI

Other communication operators include:

- **MPI_Ssend** – Waits until the destination has *started* to receive the message
- **MPI_Isend** – Send, but do not wait – how soon can buffer be (re-)used?
- **MPI_Irecv** – Indicate a buffer for data storage, but do not wait for it; the data transferred asynchronously
- **MPI_Wait** – Block until a given non-blocking send or recv has completed
- And lots more
MPI

- Simple synchronisation: `MPI_Barrier(MPI_Comm comm)`;
  - This blocks until all the processes in the communicator have reached the barrier.
- Properties of messaging:
  - MPI messages are in order, but not fair.
  - “not fair” means “not guaranteed fair”.
  - Normally events happen as expected, but not always.
MPI

- Simple synchronisation: `MPI_Barrier(MPI_Comm comm)`;
- This blocks until all the processes in the communicator have reached the barrier

Properties of messaging:
- MPI messages are *in order*, but *not fair*:
  - However
  - “not fair” means “not guaranteed fair”
  - Normally events happen as expected, but not always
MPI

- Simple synchronisation: `MPI_Barrier(MPI_Comm comm)`;
- This blocks until all the processes in the communicator have reached the barrier.
- Properties of messaging:
  - MPI messages are *in order*, but *not fair*:
    - A sends $M_1$ then $M_2$ to B
    - B receives $M_1$ then $M_2$
    - Because messages from one source to the same destination do not overtake each other.
  - However:
    - A prior (but unread) message from A to B may be overtaken by a later message from C to B
    - There is no guarantee of order on messages from different sources.
  - “not fair” means “not guaranteed fair”
  - Normally events happen as expected, but not always
MPI

- Simple synchronisation: `MPI_Barrier(MPI_Comm comm);`
- This blocks until all the processes in the communicator have reached the barrier
- Properties of messaging:
  - MPI messages are *in order*, but *not fair*:
    - A sends $M_1$ then $M_2$ to B
    - B receives $M_1$, then $M_2$
    - because messages from one source to the same destination do not overtake each other
  - However:
    - a prior (but unread) message from A to B may be overtaken by a later message from C to B
    - there is no guarantee of order on messages from different sources

“not fair” means “not guaranteed fair”
- Normally events happen as expected, but not always
MPI

- Simple synchronisation: `MPI_Barrier(MPI_Comm comm)`;
- This blocks until all the processes in the communicator have reached the barrier
- Properties of messaging:
  - MPI messages are *in order*, but *not fair*:
    - A sends $M_1$ then $M_2$ to B
    - B receives $M_1$, then $M_2$
    - because messages from one source to the same destination do not overtake each other
- However:
  - a prior (but unread) message from A to B may be overtaken by a later message from C to B
  - there is no guarantee of order on messages from different sources
- “not fair” means “not guaranteed fair”
- Normally events happen as expected, but not always
MPI

- Simple synchronisation: `MPI_Barrier(MPI_Comm comm)`;
- This blocks until all the processes in the communicator have reached the barrier.
- Properties of messaging:
  - MPI messages are *in order*, but *not fair*:
    - A sends $M_1$ then $M_2$ to B
    - B receives $M_1$, then $M_2$
      - because messages from one source to the same destination do not overtake each other
  - However:
    - a prior (but unread) message from A to B may be overtaken by a later message from C to B
    - there is no guarantee of order on messages from different sources
- “not fair” means “not guaranteed fair”
- Normally events happen as expected, but not always
Simple synchronisation: `MPI_BARRIER(MPI_COMM_WORLD)`;

This blocks until all the processes in the communicator have reached the barrier.

Properties of messaging:

- MPI messages are *in order*, but *not fair*:
  - A sends $M_1$ then $M_2$ to B
  - B receives $M_1$, then $M_2$
  - because messages from one source to the same destination do not overtake each other

However:

- a prior (but unread) message from A to B may be overtaken by a later message from C to B
- there is no guarantee of order on messages from different sources

“not fair” means “not guaranteed fair”

Normally events happen as expected, but not always
**MPI**

- **Simple synchronisation:** `MPI_BARRIER(MPI_Comm comm);`
  - This blocks until all the processes in the communicator have reached the barrier

- **Properties of messaging:**
  - **MPI messages are *in order*, but *not fair*:**
    - A sends $M_1$ then $M_2$ to B
    - B receives $M_1$, then $M_2$
    - because messages from one source to the same destination do not overtake each other
  - **However:**
    - a prior (but unread) message from A to B may be overtaken by a later message from C to B
    - there is no guarantee of order on messages from different sources

  - “not fair” means “not guaranteed fair”
  - Normally events happen as expected, but not always
MPI

- Simple synchronisation: `MPI_Barrier(MPI_Comm comm)`;
- This blocks until all the processes in the communicator have reached the barrier.
- Properties of messaging:
  - MPI messages are *in order*, but *not fair*:
    - A sends $M_1$ then $M_2$ to B
    - B receives $M_1$, then $M_2$
    - because messages from one source to the same destination do not overtake each other
  - However:
    - a prior (but unread) message from A to B may be overtaken by a later message from C to B
    - there is no guarantee of order on messages from different sources
- “not fair” means “not guaranteed fair”
- Normally events happen as expected, but not always
MPI

- Simple synchronisation: MPI_Barrier(MPI_Comm comm);
- This blocks until all the processes in the communicator have reached the barrier
- Properties of messaging:
  - MPI messages are in order, but not fair:
    - A sends $M_1$ then $M_2$ to B
    - B receives $M_1$, then $M_2$
    - because messages from one source to the same destination do not overtake each other
  - However:
    - a prior (but unread) message from A to B may be overtaken by a later message from C to B
    - there is no guarantee of order on messages from different sources

“not fair” means “not guaranteed fair”

Normally events happen as expected, but not always
MPI

- Simple synchronisation: MPI_Barrier(MPI_Comm comm);
- This blocks until all the processes in the communicator have reached the barrier
- Properties of messaging:
  - MPI messages are *in order*, but *not fair*:
    - A sends $M_1$ then $M_2$ to B
    - B receives $M_1$, then $M_2$
    - because messages from one source to the same destination do not overtake each other
- However:
  - a prior (but unread) message from A to B may be overtaken by a later message from C to B
  - there is no guarantee of order on messages from different sources
- “not fair” means “not guaranteed fair”
- Normally events happen as expected, but not always
MPI

- Simple synchronisation: `MPI_Barrier(MPI_Comm comm)`;
- This blocks until all the processes in the communicator have reached the barrier;
- Properties of messaging:
  - MPI messages are *in order*, but *not fair*:
    - A sends $M_1$ then $M_2$ to B
    - B receives $M_1$, then $M_2$
    - because messages from one source to the same destination do not overtake each other
  - However:
    - a prior (but unread) message from A to B may be overtaken by a later message from C to B
    - there is no guarantee of order on messages from different sources
- “not fair” means “not guaranteed fair”
- Normally events happen as expected, but not always
MPI

- Send and receive are for point-to-point messages: one source and one destination
- There are several others:
  - broadcast
  - scatter
  - gather
  - reduce
  - scan
MPI

- Send and receive are for point-to-point messages: one source and one destination
- There are several others:
  - broadcast
  - scatter
  - gather
  - reduce
  - scan
MPI

- Send and receive are for point-to-point messages: one source and one destination
- There are several others:
  - broadcast
  - scatter
  - gather
  - reduce
  - scan
Send and receive are for point-to-point messages: one source and one destination

There are several others:

- broadcast
- scatter
- gather
- reduce
- scan
MPI

- Send and receive are for point-to-point messages: one source and one destination
- There are several others:
  - broadcast
  - scatter
  - gather
  - reduce
  - scan
Send and receive are for point-to-point messages: one source and one destination

There are several others:
- broadcast
- scatter
- gather
- reduce
- scan
Send and receive are for point-to-point messages: one source and one destination

There are several others:
- broadcast
- scatter
- gather
- reduce
- scan
MPI

MPI_Bcast(void* buffer, int count,
MPI_Datatype datatype, int root,
MPI_Comm comm);

- Data sent from process with rank root to all other processes (in the communicator)
MPI

- All processes, including the receivers, should call MPI_Bcast with the same value for root

```c
int n[2];
if (mynum == 1) {
    n[0] = 23;
    n[1] = 42;
}
...
MPI_Bcast(n, 2, MPI_INT, 1, MPI_COMM_WORLD);
```

- All processes will now have the same values for their copies of n
MPI

**MPI_Scatter**

```c
MPI_Scatter(void* sendbuf, int sendcount,
MPI_Datatype sendtype, void* recvbuf,
int recvcount, MPI_Datatype recvtype,
int root, MPI_Comm comm);
```

Takes the data `sendbuf` in processor with rank `root`, and sends `sendcount` items from the array to every other processor (and to itself) to be stored in `recvbuf`:

Before:
```
3 1 4 1 9
2 7 1 8 4
1 4 1 4 6
6 2 0 0 8
```

After:
```
3 1 4 1 2
2 7 1 8 7
1 4 1 4 1
6 2 0 0 8
```
MPI

MPI_Gather(void* sendbuf, int sendcount,
    MPI_Datatype sendtype, void* recvbuf,
    int recvcount, MPI_Datatype recvtype, int root,
    MPI_Comm comm);

- Takes sendcount elements of data sendbuf from each processor and puts them in the array recvbuf on processor root

MPI_Gather is the inverse of MPI_Scatter
MPI

MPI_Reduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);

- Applies a reduction operation op to each value in sendbuf, putting the result(s) into recvbuf on processor root

- Operations include: max, min, +, *, ∧, ∨
- Can also use programmer-define reduction operators
MPI

MPI_Scan(void* sendbuf, void* recvbuf,
int count, MPI_Datatype datatype,
MPI_Op op, MPI_Comm comm);

- A prefix scan of the source sendbuf
- Processor of rank $i$ gets the reduction of values from processors $0...i$ stored in its recvbuf

![Prefix scan example]

- Prefix scans are very useful in parallel algorithms
MPI

- Need to think carefully about messages to get maximum efficiency
  - For example, reduce latency through non-blocking operations
    - Compute something else while a receive operation completes, then go back
    - Requires careful programming
    - Overlapping communication and computation is a good, but delicate
  - Also:
    - Messaging has a high overhead: needs (very) large programs
    - Hard to program effectively: needs experience
    - Lacks dynamism: processor count is fixed and cannot change during execution
MPI

- Need to think carefully about messages to get maximum efficiency
- For example, reduce latency through non-blocking operations
  - Compute something else while a receive operation completes, then go back
  - Requires careful programming
  - Overlapping communication and computation is a good, but delicate
- Also:
  - Messaging has a high overhead: needs (very) large programs
  - Hard to program effectively: needs experience
  - Lacks dynamism: processor count is fixed and cannot change during execution
MPI

- Need to think carefully about messages to get maximum efficiency
- For example, reduce latency through non-blocking operations
  - Compute something else while a receive operation completes, then go back
  - Requires careful programming
  - Overlapping communication and computation is a good, but delicate

Also:
- Messaging has a high overhead: needs (very) large programs
- Hard to program effectively: needs experience
- Lacks dynamism: processor count is fixed and cannot change during execution
MPI

- Need to think carefully about messages to get maximum efficiency
- For example, reduce latency through non-blocking operations
  - Compute something else while a receive operation completes, then go back
  - Requires careful programming
  - Overlapping communication and computation is a good, but delicate
- Also:
  - Messaging has a high overhead; needs (very) large programs
  - Hard to program effectively; needs experience
  - Lacks dynamism; processor count is fixed and cannot change during execution
MPI

- Need to think carefully about messages to get maximum efficiency
- For example, reduce latency through non-blocking operations
  - Compute something else while a receive operation completes, then go back
  - Requires careful programming
  - Overlapping communication and computation is a good, but delicate

- Also:
  - Messaging has a high overhead: needs (very) large programs
  - Hard to program effectively: needs experience
  - Lacks dynamism: processor count is fixed and cannot change during execution
MPI

- Need to think carefully about messages to get maximum efficiency
- For example, reduce latency through non-blocking operations
  - Compute something else while a receive operation completes, then go back
  - Requires careful programming
  - Overlapping communication and computation is a good, but delicate
- Also:
  - Messaging has a high overhead: needs (very) large programs
  - Hard to program effectively: needs experience
  - Lacks dynamism: processor count is fixed and cannot change during execution
MPI

- Need to think carefully about messages to get maximum efficiency
- For example, reduce latency through non-blocking operations
  - Compute something else while a receive operation completes, then go back
  - Requires careful programming
  - Overlapping communication and computation is a good, but delicate
- Also:
  - Messaging has a high overhead: needs (very) large programs
    - Hard to program effectively: needs experience
    - Lacks dynamism: processor count is fixed and cannot change during execution
MPI

- Need to think carefully about messages to get maximum efficiency
- For example, reduce latency through non-blocking operations
  - Compute something else while a receive operation completes, then go back
  - Requires careful programming
  - Overlapping communication and computation is a good, but delicate
- Also:
  - Messaging has a high overhead: needs (very) large programs
  - Hard to program effectively: needs experience
  - Lacks dynamism: processor count is fixed and cannot change during execution
MPI

- Need to think carefully about messages to get maximum efficiency
- For example, reduce latency through non-blocking operations
  - Compute something else while a receive operation completes, then go back
  - Requires careful programming
  - Overlapping communication and computation is a good, but delicate
- Also:
  - messaging has a high overhead: needs (very) large programs
  - hard to program effectively: needs experience
  - lacks dynamism: processor count is fixed and cannot change during execution
MPI

MPI has succeeded for many reasons

- An open standard, inviting several competing implementations
- Thus implementations are optimised and efficient
- MPI is simple in concept, so straightforward to program (not necessarily easy to program...)
- MPI is flexible as it contains lots of kinds of communication
- MPI is supported by many languages and environments
- MPI scales well to very large problems

The MPI standard is still being developed and updated
MPI

MPI has succeeded for many reasons

- An open standard, inviting several competing implementations
- Thus implementations are optimised and efficient
  - MPI is simple in concept, so straightforward to program (not necessarily easy to program...)
  - MPI is flexible as it contains lots of kinds of communication
  - MPI is supported by many languages and environments
  - MPI scales well to very large problems

The MPI standard is still being developed and updated
MPI

MPI has succeeded for many reasons

■ An open standard, inviting several competing implementations
■ Thus implementations are optimised and efficient
■ MPI is simple in concept, so straightforward to program (not necessarily easy to program...)
■ MPI is flexible as it contains lots of kinds of communication
■ MPI is supported by many languages and environments
■ MPI scales well to very large problems

The MPI standard is still being developed and updated
MPI

MPI has succeeded for many reasons

- An open standard, inviting several competing implementations
- Thus implementations are optimised and efficient
- MPI is simple in concept, so straightforward to program (not necessarily easy to program...)
- MPI is flexible as it contains lots of kinds of communication
  - MPI is supported by many languages and environments
  - MPI scales well to very large problems

The MPI standard is still being developed and updated
MPI

MPI has succeeded for many reasons

- An open standard, inviting several competing implementations
- Thus implementations are optimised and efficient
- MPI is simple in concept, so straightforward to program (not necessarily easy to program . . .)
- MPI is flexible as it contains lots of kinds of communication
- MPI is supported by many languages and environments
- MPI scales well to very large problems

The MPI standard is still being developed and updated
MPI

MPI has succeeded for many reasons

- An open standard, inviting several competing implementations
- Thus implementations are optimised and efficient
- MPI is simple in concept, so straightforward to program (not necessarily easy to program...)
- MPI is flexible as it contains lots of kinds of communication
- MPI is supported by many languages and environments
- MPI scales well to very large problems

The MPI standard is still being developed and updated
The Google problem

- **Motivation:** Large Scale Data Processing
  - Many tasks: Process lots of data to produce other data
  - Want to use hundreds or thousands of CPUs... but this needs to be easy
  - MapReduce provides:
    - Automatic parallelization and distribution
    - Fault-tolerance
    - I/O scheduling
    - Status and monitoring

The Google problem

- **Motivation**: Large Scale Data Processing
- **Many tasks**: Process lots of data to produce other data
  - Want to use hundreds or thousands of CPUs... but this needs to be easy
- **MapReduce provides**:
  - Automatic parallelization and distribution
  - Fault-tolerance
  - I/O scheduling
  - Status and monitoring

The Google problem

- Motivation: Large Scale Data Processing
- Many tasks: Process lots of data to produce other data
- Want to use hundreds or thousands of CPUs... but this needs to be easy

MapReduce provides:
- Automatic parallelization and distribution
- Fault-tolerance
- I/O scheduling
- Status and monitoring

The Google problem

- Motivation: Large Scale Data Processing
- Many tasks: Process lots of data to produce other data
- Want to use hundreds or thousands of CPUs... but this needs to be easy
- MapReduce provides:
  - Automatic parallelization and distribution
  - Fault-tolerance
  - I/O scheduling
  - Status and monitoring

The MapReduce model I

- Input & Output: each a set of key/value pairs
- Programmer specifies two functions:

  \[
  \text{map (in\_key, in\_value) -> list(out\_key, intermediate\_value)}
  \]

  - Processes input key/value pair
  - Produces set of intermediate pairs

  \[
  \text{reduce (out\_key, list(intermediate\_value)) -> list(out\_value)}
  \]

  - Combines all intermediate values for a particular key
  - Produces a set of merged output values (usually just one)

- Inspired by similar primitives in LISP and other languages
The MapReduce model II

细粒度任务：比机器多得多的 map 任务

- 最小化故障恢复时间
- 可以将shuffle管道与map执行一起进行
- 动态负载均衡更好

Fine granularity tasks: many more map tasks than machines

- Minimizes time for fault recovery
- Can pipeline shuffling with map execution
- Better dynamic load balancing
The MapReduce model III

- Often use 200,000 map/5000 reduce tasks w/ 2000 machines
Fault tolerance + Redundancy

- When processes break: re-execute
  - Detect failure via periodic heartbeats
  - Re-execute completed and in-progress map tasks
  - Re-execute in progress reduce tasks
  - Task completion committed through master
- Master failure: not considered
- Robust: task completed even when 1600 out of 1800 machines failed
- Slow workers significantly lengthen completion time
- Solution: Near end of phase, spawn backup copies of tasks—whichever one finishes first “wins”
- Effect: Dramatically shortens job completion time
- Rewrote Google’s production indexing system into 24 MapReduce operations
- Statistics: #jobs = 29,423, avg. duration = 634secs, total machine time = 79,186days, input data = 3,288TB
Fault tolerance + Redundancy

- When processes break: re-execute
  - Detect failure via periodic heartbeats
  - Re-execute completed and in-progress map tasks
  - Re-execute in progress reduce tasks
  - Task completion committed through master

- Master failure: not considered
  - Robust: task completed even when 1600 out of 1800 machines failed
  - Slow workers significantly lengthen completion time
  - Solution: Near end of phase, spawn backup copies of tasks—whichever one finishes first “wins”
  - Effect: Dramatically shortens job completion time
  - Rewrote Google’s production indexing system into 24 MapReduce operations
  - Statistics: \#jobs = 29,423, avg. duration = 634secs, total machine time = 79,186days, input data = 3,288TB
Fault tolerance + Redundancy

- When processes break: re-execute
  - Detect failure via periodic heartbeats
  - Re-execute completed and in-progress map tasks
  - Re-execute in progress reduce tasks
  - Task completion committed through master

- Master failure: not considered

- Robust: task completed even when 1600 out of 1800 machines failed
  - Slow workers significantly lengthen completion time
  - Solution: Near end of phase, spawn backup copies of tasks—whichever one finishes first “wins”
  - Effect: Dramatically shortens job completion time
  - Rewrote Google’s production indexing system into 24 MapReduce operations
  - Statistics: #jobs = 29,423, avg. duration = 634secs, total machine time = 79,186days, input data = 3,288TB
Fault tolerance + Redundancy

- When processes break: re-execute
  - Detect failure via periodic heartbeats
  - Re-execute completed and in-progress map tasks
  - Re-execute in progress reduce tasks
  - Task completion committed through master
- Master failure: not considered
- Robust: task completed even when 1600 out of 1800 machines failed
- Slow workers significantly lengthen completion time
  - Solution: Near end of phase, spawn backup copies of tasks—whichever one finishes first “wins”
  - Effect: Dramatically shortens job completion time
- Rewrote Google’s production indexing system into 24 MapReduce operations
  - Statistics: #jobs = 29,423, avg. duration = 634 secs, total machine time = 79,186 days, input data = 3,288 TB
Fault tolerance + Redundancy

- When processes break: re-execute
  - Detect failure via periodic heartbeats
  - Re-execute completed and in-progress map tasks
  - Re-execute in progress reduce tasks
  - Task completion committed through master
- Master failure: not considered
- Robust: task completed even when 1600 out of 1800 machines failed
- Slow workers significantly lengthen completion time
- Solution: Near end of phase, spawn backup copies of tasks—whichever one finishes first “wins”
  - Effect: Dramatically shortens job completion time
  - Rewrote Google’s production indexing system into 24 MapReduce operations
  - Statistics: #jobs = 29,423, avg. duration = 634secs, total machine time = 79,186days, input data = 3,288TB
Fault tolerance + Redundancy

- When processes break: re-execute
  - Detect failure via periodic heartbeats
  - Re-execute completed and in-progress map tasks
  - Re-execute in progress reduce tasks
  - Task completion committed through master
- Master failure: not considered
- Robust: task completed even when 1600 out of 1800 machines failed
- Slow workers significantly lengthen completion time
- Solution: Near end of phase, spawn backup copies of tasks—whichever one finishes first “wins”
- Effect: Dramatically shortens job completion time
- Rewrote Google’s production indexing system into 24 MapReduce operations
- Statistics: #jobs = 29,423, avg. duration = 634secs, total machine time = 79,186days, input data = 3,288TB
Fault tolerance + Redundancy

- When processes break: re-execute
  - Detect failure via periodic heartbeats
  - Re-execute completed and in-progress map tasks
  - Re-execute in progress reduce tasks
  - Task completion committed through master
- Master failure: not considered
- Robust: task completed even when 1600 out of 1800 machines failed
- Slow workers significantly lengthen completion time
- Solution: Near end of phase, spawn backup copies of tasks—whichever one finishes first “wins”
- Effect: Dramatically shortens job completion time
- Rewrote Google’s production indexing system into 24 MapReduce operations

Statistics: 

- $\#$ jobs = 29,423, avg. duration = 634 secs,
- total machine time = 79,186 days, input data = 3,288 TB
Fault tolerance + Redundancy

- When processes break: re-execute
  - Detect failure via periodic heartbeats
  - Re-execute completed and in-progress map tasks
  - Re-execute in progress reduce tasks
  - Task completion committed through master

- Master failure: not considered

- Robust: task completed even when 1600 out of 1800 machines failed

- Slow workers significantly lengthen completion time

- Solution: Near end of phase, spawn backup copies of tasks—whichever one finishes first “wins”

- Effect: Dramatically shortens job completion time

- Rewrote Google’s production indexing system into 24 MapReduce operations

- Statistics: #jobs = 29,423, avg. duration = 634secs, total machine time = 79,186days, input data = 3,288TB
memcached

- High-performance, distributed memory object cache
  - Purpose: speed up dynamic web applications by reducing database load
  - Functionality: a giant hash table distributed across multiple machines
  - Behaviour: when the table is full, subsequent inserts cause older data to be purged in least recently used (LRU) order
  - Use case: developed by Danga Interactive for LiveJournal.com:
    - Site statistics: 20 million+ dynamic page views per day for 1 million users
    - Reduced database load close to 0; faster page load times; better resource utilization, and faster access to databases on a memcache miss.
  - Summary at http://en.wikipedia.org/wiki/Memcached
memcached

- High-performance, distributed memory object cache
- Purpose: speed up dynamic web applications by reducing database load
- Functionality: a giant hash table distributed across multiple machines
- Behaviour: when the table is full, subsequent inserts cause older data to be purged in least recently used (LRU) order
- Use case: developed by Danga Interactive for LiveJournal.com:
  - Site statistics: 20 million+ dynamic page views per day for 1 million users
  - Reduced database load close to 0; faster page load times; better resource utilization, and faster access to databases on a memcache miss.
- Summary at http://en.wikipedia.org/wiki/Memcached
memcached

- High-performance, distributed memory object cache
- Purpose: speed up dynamic web applications by reducing database load
- Functionality: a giant hash table distributed across multiple machines
- Behaviour: when the table is full, subsequent inserts cause older data to be purged in least recently used (LRU) order
- Use case: developed by Danga Interactive for LiveJournal.com:
  - Site statistics: 20 million+ dynamic page views per day for 1 million users
  - Reduced database load close to 0; faster page load times; better resource utilization, and faster access to databases on a memcache miss.

memcached

- High-performance, distributed memory object cache
- Purpose: speed up dynamic web applications by reducing database load
- Functionality: a giant hash table distributed across multiple machines
- Behaviour: when the table is full, subsequent inserts cause older data to be purged in least recently used (LRU) order
- Use case: developed by Danga Interactive for LiveJournal.com:
  - Site statistics: 20 million+ dynamic page views per day for 1 million users
  - Reduced database load close to 0; faster page load times; better resource utilization, and faster access to databases on a memcache miss.
- Summary at http://en.wikipedia.org/wiki/Memcached
memcached

- High-performance, distributed memory object cache
- Purpose: speed up dynamic web applications by reducing database load
- Functionality: a giant hash table distributed across multiple machines
- Behaviour: when the table is full, subsequent inserts cause older data to be purged in least recently used (LRU) order
- Use case: developed by Danga Interactive for LiveJournal.com:
  - Site statistics: 20 million+ dynamic page views per day for 1 million users
  - Reduced database load close to 0; faster page load times; better resource utilization, and faster access to databases on a memcache miss.

memcached

- High-performance, distributed memory object cache
- Purpose: speed up dynamic web applications by reducing database load
- Functionality: a giant hash table distributed across multiple machines
- Behaviour: when the table is full, subsequent inserts cause older data to be purged in least recently used (LRU) order
- Use case: developed by Danga Interactive for LiveJournal.com:
  - Site statistics: 20 million+ dynamic page views per day for 1 million users
  - Reduced database load close to 0; faster page load times; better resource utilization, and faster access to databases on a memcache miss.

Summary at http://en.wikipedia.org/wiki/Memcached
HADOOP

- Distributed processing of large data sets
- On clusters of computers
- Simple programming model
- Scalable
- Reliability model in software $\leadsto$ high-availability on unreliable hardware
- Components:
  - Avro: A data serialization system.
  - Cassandra: A scalable multi-master database with no single points of failure.
  - HBase: A scalable, distributed database that supports structured data storage for large tables.
  - Hive: A data warehouse infrastructure that provides data summarization and ad hoc querying.
  - Mahout: machine learning and data mining library.
  - Pig: A high-level data-flow language and execution framework for parallel computation.
HADOOP

- Distributed processing of large data sets
- On clusters of computers
- Simple programming model
- Scalable
- Reliability model in software $\leadsto$ high-availability on unreliable hardware

Components:
- Avro: A data serialization system.
- Cassandra: A scalable multi-master database with no single points of failure.
- HBase: A scalable, distributed database that supports structured data storage for large tables.
- Hive: A data warehouse infrastructure that provides data summarization and ad hoc querying.
- Mahout: machine learning and data mining library.
- Pig: A high-level data-flow language and execution framework for parallel computation.
HADOOP

- Distributed processing of large data sets
- On clusters of computers
- Simple programming model
  - Scalable
  - Reliability model in software $\Rightarrow$ high-availability on unreliable hardware
- Components:
  - Avro: A data serialization system.
  - Cassandra: A scalable multi-master database with no single points of failure.
  - HBase: A scalable, distributed database that supports structured data storage for large tables.
  - Hive: A data warehouse infrastructure that provides data summarization and ad hoc querying.
  - Mahout: machine learning and data mining library.
  - Pig: A high-level data-flow language and execution framework for parallel computation.
HADOOP

- Distributed processing of large data sets
- On clusters of computers
- Simple programming model
- Scalable

  - Reliability model in software → high-availability on unreliable hardware

  - Components:
    - Avro: A data serialization system.
    - Cassandra: A scalable multi-master database with no single points of failure.
    - HBase: A scalable, distributed database that supports structured data storage for large tables.
    - Hive: A data warehouse infrastructure that provides data summarization and ad hoc querying.
    - Mahout: machine learning and data mining library.
    - Pig: A high-level data-flow language and execution framework for parallel computation.
**HADOOP**

- Distributed processing of large data sets
- On clusters of computers
- Simple programming model
- Scalable
- Reliability model in software $\leadsto$ high-availability on unreliable hardware

**Components:**

- Avro: A data serialization system.
- Cassandra: A scalable multi-master database with no single points of failure.
- HBase: A scalable, distributed database that supports structured data storage for large tables.
- Hive: A data warehouse infrastructure that provides data summarization and ad hoc querying.
- Mahout: machine learning and data mining library.
- Pig: A high-level data-flow language and execution framework for parallel computation.
HADOOP

- Distributed processing of large data sets
- On clusters of computers
- Simple programming model
- Scalable
- Reliability model in software $\Rightarrow$ high-availability on unreliable hardware

Components:
- Avro: A data serialization system.
- Cassandra: A scalable multi-master database with no single points of failure.
- HBase: A scalable, distributed database that supports structured data storage for large tables.
- Hive: A data warehouse infrastructure that provides data summarization and ad hoc querying.
- Mahout: machine learning and data mining library.
- Pig: A high-level data-flow language and execution framework for parallel computation.
HADOOP users

**Facebook:** We use Hadoop to store copies of internal log and dimension data sources and use it as a source for reporting/analytics and machine learning. Currently we have 2 major clusters:

- A 1100-machine cluster with 8800 cores and about 12 PB raw storage.
- A 300-machine cluster with 2400 cores and about 3 PB raw storage.
- Each (commodity) node has 8 cores and 12 TB of storage.
- We are heavy users of both streaming as well as the Java APIs. We have built a higher level data warehousing framework using Hive.

http://wiki.apache.org/hadoop/PoweredBy
**HADOOP users**

**Facebook:** We use Hadoop to store copies of internal log and dimension data sources and use it as a source for reporting/analytics and machine learning. Currently we have 2 major clusters:

- A 1100-machine cluster with 8800 cores and about 12 PB raw storage.
- A 300-machine cluster with 2400 cores and about 3 PB raw storage.

- Each (commodity) node has 8 cores and 12 TB of storage.
- We are heavy users of both streaming as well as the Java APIs. We have built a higher level data warehousing framework using Hive

http://wiki.apache.org/hadoop/PoweredBy
HADOOP users

Facebook: We use Hadoop to store copies of internal log and dimension data sources and use it as a source for reporting/analytics and machine learning. Currently we have 2 major clusters:

- A 1100-machine cluster with 8800 cores and about 12 PB raw storage.
- A 300-machine cluster with 2400 cores and about 3 PB raw storage.
- Each (commodity) node has 8 cores and 12 TB of storage.

We are heavy users of both streaming as well as the Java APIs. We have built a higher level data warehousing framework using Hive

http://wiki.apache.org/hadoop/PoweredBy
HADOOP users

Facebook: We use Hadoop to store copies of internal log and dimension data sources and use it as a source for reporting/analytics and machine learning. Currently we have 2 major clusters:

- A 1100-machine cluster with 8800 cores and about 12 PB raw storage.
- A 300-machine cluster with 2400 cores and about 3 PB raw storage.
- Each (commodity) node has 8 cores and 12 TB of storage.
- We are heavy users of both streaming as well as the Java APIs. We have built a higher level data warehousing framework using Hive

http://wiki.apache.org/hadoop/PoweredBy
Content

1 Web programming
2 Distributed programming
3 Programming frameworks
4 Summary
Summary

- Web programming is a special case of distributed programming
- Large number of tools/frameworks—best learnt on need
- Basic principle: code in browser + code in server + communication
- Distributed programming in general is hard to debug and hard to get right
- Fortunately many problems are *embarrassingly* parallel and—with the right abstractions—can be mapped reliably to stock hardware
Summary

- Web programming is a special case of distributed programming
- Large number of tools/frameworks—best learnt on need
  - Basic principle: code in browser + code in server + communication
- Distributed programming in general is hard to debug and hard to get right
- Fortunately many problems are embarassingly parallel and—with the right abstractions—can be mapped reliably to stock hardware
Summary

- Web programming is a special case of distributed programming
- Large number of tools/frameworks—best learnt on need
- Basic principle: code in browser + code in server + communication
- Distributed programming in general is hard to debug and hard to get right
- Fortunately many problems are embarrassingly parallel and—with the right abstractions—can be mapped reliably to stock hardware
Summary

- Web programming is a special case of distributed programming
- Large number of tools/frameworks—best learnt on need
- Basic principle: code in browser + code in server + communication
- Distributed programming in general is hard to debug and hard to get right
  - Fortunately many problems are *embarrassingly* parallel and—with the right abstractions—can be mapped reliably to stock hardware
Summary

- Web programming is a special case of distributed programming
- Large number of tools/frameworks—best learnt on need
- Basic principle: code in browser + code in server + communication
- Distributed programming in general is hard to debug and hard to get right
- Fortunately many problems are embarassingly parallel and—with the right abstractions—can be mapped reliably to stock hardware
"Never believe one source"

- Java applets tutorial: http://java.sun.com/docs/books/tutorial/deployment/applet/appletsonlyindex.html
- Java servlets tutorial http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Servlets.html
- Message-passing interface: http://www.open-mpi.org/
- Tuple spaces: http://en.wikipedia.org/wiki/Tuple_space
Resources II

- memcached:
  http://en.wikipedia.org/wiki/Memcached and
  http://www.danga.com/memcached/
- RFC2965, HTTP State Management Mechanism, D.Kristol, L.Montulli, October 2000, ftp:
  //ftp.rfc-editor.org/in-notes/rfc2965.txt
References


