Course Outline

• Introduction in algorithms and applications
• Parallel machines and architectures
  Overview of parallel machines, trends in top-500, clusters, many-cores
• Programming methods, languages, and environments
  Message passing (SR, MPI, Java)
  Higher-level language: HPF
• Many-core programming, 3 lectures (Ana Varbanescu)
• Applications
  N-body problems, search algorithms
• LOFAR software telescope (Rob van Nieuwpoort)
Approaches to Parallel Programming

- Sequential language + library
  - MPI, PVM
- Extend sequential language
  - C/Linda, Concurrent C++, HPF
- New languages designed for parallel or distributed programming
  - SR, occam, Ada, Orca
Paradigms for Parallel Programming

- Processes + shared variables
- Processes + message passing
- Concurrent object-oriented languages
- Concurrent functional languages
- Concurrent logic languages
- Data-parallelism (SPMD model)

SR and MPI
Java
HPF, CUDA
Paper:

Interprocess Communication and Synchronization based on Message Passing

Henri Bal
Overview

• Message passing
  – Naming the sender and receiver
  – Explicit or implicit receipt of messages
  – Synchronous versus asynchronous messages

• Nondeterminism
  – Select statement

• Example language: SR (Synchronizing Resources)
• Example library: MPI (Message Passing Interface)
Point-to-point Message Passing

- Basic primitives: send & receive
  - As library routines:
    - send(destination, &MsgBuffer)
    - receive(source, &MsgBuffer)
  - As language constructs
    - send MsgName(arguments) to destination
    - receive MsgName(arguments) from source
Direct naming

• Sender and receiver directly name each other
  – S: `send M to R`
  – R: `receive M from S`

• Asymmetric direct naming (more flexible):
  – S: `send M to R`
  – R: `receive M`

• Direct naming is easy to implement
  – Destination of message is known in advance
  – Implementation just maps logical names to machine addresses
Indirect naming

• Indirect naming uses extra indirection level
  – R: send M to P — P is a port name
  – S: receive M from P
• Sender and receiver need not know each other
• Port names can be moved around in a message
  – send ReplyPort(P) to U — P is name of reply port
• Most languages allow only a single process at a time to receive from any given port
• Some languages allow multiple receivers that service messages on demand -> called a mailbox
Explicit Message Receipt

- Explicit receive by an existing process
  - Receiving process only handles message when it is willing to do so

```java
process main()
{
    // regular computation here
    receive M( ....);  // explicit message receipt
    // code to handle message
    // more regular computations
    ....
}
```
Implicit message receipt

- Receipt by a new thread of control, created for handling the incoming message

```c
int X;
process main() {
    // just regular computations, this code can access X
}
message-handler M() // created whenever a message M arrives
{
    // code to handle the message, can also access X
}
```
Threads

- Threads run in (pseudo-) parallel on the same node
- Each thread has its own program counter and local variables
- Threads share global variables
Differences (1)

• Implicit receipt is used if it’s unknown when a message will arrive; example: global bound in TSP

```java
int Minimum;
process main( )
{
    // regular computations
}
message-handler
Update(m: int)
{
    if (m<Minimum)
        Minimum = m
}
```
Differences (2)

- Explicit receive gives more control over when to accept which messages; e.g., SR allows:
  - receive ReadFile(file, offset, NrBytes) by NrBytes
  - // sorts messages by (increasing) 3rd parameter, i.e. small reads go first
- MPI has explicit receive (+ polling for implicit receive)
- Java has implicit receive: Remote Method Invocation (RMI)
- SR has both
Synchronous vs. asynchronous Message Passing

• Synchronous message passing:
  – Sender is blocked until receiver has accepted the message
  – Too restrictive for many parallel applications

• Asynchronous message passing:
  – Sender continues immediately
  – More efficient
  – Ordering problems
  – Buffering problems
Message ordering

• Ordering with asynchronous message passing

  SENDER:
  • send message(1)
  • send message(2)

  RECEIVER:
  receive message(N); print N
  receive message(M); print M

• Messages may be received in any order, depending on the protocol
Example: AT&T crash

- P1 asks: Are you still alive?
- P1 crashes
- P2 is dead
- P2 sends: I’m back, Regular message
- Something’s wrong, I’d better crash!
- P2 is dead
Message buffering

• Keep messages in a buffer until the receive( ) is done
• What if the buffer overflows?
  – Continue, but delete some messages (e.g., oldest one), or
  – Use flow control: block the sender temporarily
• Flow control changes the semantics since it introduces synchronization
  – S: send zillion messages to R; receive messages
  – R: send zillion messages to S; receive messages
  • -> deadlock!
Nondeterminism

- Interactions may depend on run-time conditions
  - e.g.: wait for a message from either A or B, whichever comes first
- Need to express and control nondeterminism
  - specify when to accept which message
- Example (bounded buffer):
  - do simultaneously
    - when buffer not full: accept request to store message
    - when buffer not empty: accept request to fetch message
Select statement

• several alternatives of the form:
  – WHEN condition => RECEIVE message DO statement

• Each alternative may
  – succeed, if condition=true & a message is available
  – fail, if condition=false
  – suspend, if condition=true & no message available yet

• Entire select statement may
  – succeed, if any alternative succeeds -> pick one nondeterministically
  – fail, if all alternatives fail
  – suspend, if some alternatives suspend and none succeeds yet
Example: bounded buffer

select
  when not FULL(BUFFER) =>
    receive STORE_ITEM(X: INTEGER) do
      'store X in buffer'
    end;
  or
  when not EMPTY(BUFFER) =>
    receive FETCH_ITEM(X: out INTEGER) do
      X := 'first item from buffer'
    end;
end select;
Synchronizing Resources (SR)

- Developed at University of Arizona
- Goals of SR:
  - Expressiveness
    - Many message passing primitives
  - Ease of use
    - Minimize number of underlying concepts
    - Clean integration of language constructs
  - Efficiency
    - Each primitive must be efficient
Overview of SR

- Multiple forms of message passing
  - Asynchronous message passing
  - Rendezvous (synchronous send, explicit receipt)
  - Remote Procedure Call (synchronous send, implicit receipt)
  - Multicast (many receivers)

- Powerful receive-statement
  - Conditional & ordered receive, based on contents of message
  - Select statement

- Resource = module run on 1 node (uni/multiprocessor)
  - Contains multiple threads that share variables
Orthogonality in SR

- The send and receive primitives can be combined in all 4 possible ways

<table>
<thead>
<tr>
<th>Asynchronous send</th>
<th>Synchronous call</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit receive</td>
<td>1. asynchronous message passing</td>
</tr>
<tr>
<td>Implicit receive</td>
<td>2. fork</td>
</tr>
<tr>
<td></td>
<td>3. rendezvous</td>
</tr>
<tr>
<td></td>
<td>4. RPC</td>
</tr>
</tbody>
</table>
Example

body S  #sender
  send R.m1  #asynchr. mp
  send R.m2  # fork
  call  R.m1  # rendezvous
  call  R.m2  # RPC
end S

body R  #receiver
  proc M2( )  # implicit receipt
    # code to handle M2
  end

initial  # main process of R
  do true -> #infinite loop
    in m1( )  # explicit receive
      # code to handle m1
    ni
  od
end
end R
Managing a replicated variable in SR

- Use a BoundManager process to serialize updates

Assign: asynchr. + explicit ordered recv.
Update: synchr.+implicit recv.+multicast
SR code fragments for TSP

body worker

var M: int := Infinite

    # copy of bound

sem sema  # semaphore

proc update(value: int)

    P(sema)  # lock copy

    M := value

    V(sema)  # unlock

end update

initial  # main code for worker

- can read M (using sema) and

- sendBoundManager.Assign(value)

body BoundManager

var M: int := Infinite

    # copy of bound

do true ->

    in Assign(value) by value ->

        if value < M ->

            M := value

        co(i := 1 to ncpus)

        call

            worker[i].update(value)

        co

    fi

ni

od

end BoundManager