Overview of Lecture 8

- MPI Communication Patterns
- Distributed Programming
  - Asynchronous Load Balancing
  - Termination Detection
- Lecture based on material from
  - Peter Sanders (University of Saarbrücken)
  - Friedemann Mattern (ETH Zurich)
  - Stefano Cozzini (Democritos/INFM + SISSA)
  - Stefan Blom (UT)

## MPI Communication Patterns

### Data exchange between two processes:

#### Asymmetric

```
if (rank == 0) {
    c.send(to:1);
    c.recv(from:1);
} else {
    c.recv(from:0);
    c.send(to:0);
}
```

#### Symmetric

```
if (rank == 0) {
    req = c.irecv(from:1);
    c.send(to:1);
} else {
    req = c.irecv(from:0);
    c.send(to:0);
    req.wait();
}
```

```
if (rank == 0) {
    c.sendrecv(to:1,from:1);
} else {
    c.sendrecv(to:0,from:0);
}
```

### Collective Operations

- Collective operations provide a **higher-level way to organize a parallel program**
- Each process executes the same communication operation
- Tags are not used; different communicators deliver similar functionality
- MPI-2 has no non-blocking collective operations being worked on for MPI-3
- Three classes of operations:
  - synchronization
  - data movement
  - collective computation
In the distributed programming context, the `barrier` function is used to synchronize the processes. It ensures that all participants must enter and then leave the barrier, with the first participant leaving after the last has entered.

### Barrier

- **Function:** `void ibis.mpj.Intracomm.barrier()`
- **Usage:** All participants must enter and then leave the barrier.
- **Example:**
  ```c
  void barrier()
  ```

### Scatter

- **Function:** `void scatter(Object sendbuf, int sendoffset, int sendcount, Datatype sendtype, Object recvbuf, int recvoffset, int recvcount, Datatype recvtype, int root)`

### Gather

- **Function:** `void gather(Object sendbuf, int sendoffset, int sendcount, Datatype sendtype, Object recvbuf, int recvoffset, int recvcount, Datatype recvtype, int root)`

### Broadcast

- **Function:** `void bcast(Object buf, int offset, int count, Datatype datatype, int root)`
- **Communication:** One-to-all Communication
- **Example:**

```c
void bcast()
```
All-to-All

- void alltoall(Object sendbuf, int sendoffset, int sendcount, Datatype sendtype, Object recvbuf, int recvoffset, int recvcount, Datatype recvtype)

Reduction

- The reduction operation allows
  - Collect data from each process
  - Reduce collected data to single value
  - Store result on root process
  - Store results on all processes

Reduce, Parallel Sum

- void reduce(Object sendBuf, int sendOffset, Object recvBuf, int recvOffset, int count, Datatype datatype, Op op, int root)
- void allreduce(Object sendbuf, int sendoffset, Object recvbuf, int recvoffset, int recvcount, Datatype datatype, Op op)

General Message Passing Advice

- Waiting for every send to complete is expensive. Use non-blocking sends to overlap communication with other communication or computation
- Small messages are expensive due to protocol overhead (message headers, etc.) Use message combining: aggregate multiple small messages into a single bigger one.
  - Do not wait too long during combining (receiver might become idle (or deadlock!) meanwhile).
  - This is one of the few cases where timeouts are acceptable in a messaging protocol!
- Prefer symmetric algorithms
  - Easier to reason about
- Prefer existing algorithms and implementations
  - Difficult to reinvent the wheel correctly.
**Irregular computations with dynamic behavior**
- Graph algorithms
- Value Iteration/Fixpoint algorithms
- Approximate SAT solvers
- Data-mining algorithms for clustering
- ... 

**Heterogeneous distributed systems:**
- Nodes with varying speed
- Nodes with varying available resources (RAM, disk)

**Load Balancing**

- **Irregular computations with dynamic behavior**
  - Graph algorithms
  - Value Iteration/Fixpoint algorithms
  - Approximate SAT solvers
  - Data-mining algorithms for clustering
  - ...
  - Heterogeneous distributed systems:
    - Nodes with varying speed
    - Nodes with varying available resources (RAM, disk)

**Distributed Termination Detection**

- Consider system of **N workers**
- Workers cooperate by **message passing**
- Irregular computation (unpredictable sending patterns)
- System is **terminated**, when
  - all workers have become idle, and
  - no messages are pending (unreceived)
- Task: detect termination situation, assuming that
  - pending messages cannot be detected, but have to be deduced
  - idle workers can return to active state only by receiving messages

**Asynchronous Random Polling**

- **var** \( P, P' : \text{Subproblem} \)
- \( P := \text{if } \var{\text{no work}} \text{ then } P_{\text{root}} \text{ else } P_{\text{no work}} \)
  - while no global termination yet do
  - \( \var{\text{if } T(P) = 0 \text{ then}} \)
    - send a request to a PE chosen uniformly at random from \( N \)
  - \( \var{\text{else}} \)
    - \( P := \text{work}(P, \Delta t) \)
    - \( \text{if there is an incoming message } M \text{ then} \)
      - \( (P, P') := \text{split}(P) \)
      - send \( P' \) to PE \( j \)
    - \( \var{\text{else}} \)
      - \( P := M \)
  - Sanders, 1995
- Single “tunable” parameter \((\Delta t)\): granularity of work, too small: **load balancing overhead**, too big: **little work sharing**
- **Synchronous** variant used in LTSmin (multi-core): **excellent results**

**Safra’s Algorithm [EWD998]**

- Assume **asynchronous, directed communication** channels
- Each worker locally maintains
  - integer **count of sent-received messages**
    - **initially**: \( \var{\text{count}} = 0 \)
  - binary **status flag** (White | Black)
    - **initially**: status = White
- **Message types**: **work messages** and **termination messages**
- Upon sending a work message, increase message count
- Upon receiving a work message, decrease message count, and set status flag to Black

**Note:** requests from load balancer do not count as work messages!
Safra's Algorithm [EWD998]

- Termination message (token) has two fields
  - status flag (binary: White | Black)
  - global message count (integer)
- Passed from worker i to worker (i-1)
- Initiated at worker 0, sent to worker (N-1)
- At most one token may exist in the entire system. I.e., worker 0 must wait for the token to return before sending another one.

If worker 0 becomes idle ("spontaneously")
- set status$_0$ to White
- initiate probe by sending token <White,0> to worker N-1
- If worker 0 receives token <s,Δ> under the following conditions:
  - s = White
  - count$_0$ + Δ = 0
  - status$_0$ = White
- then conclude termination

Any worker i, with i>0:
- can receive token <s,Δ> at any time
- when active, keeps token
  (remember: upon reception of work message, status$i$ is set to Black)
- when idle, must forward token <s', Δ'> to worker (i-1):
  - s' = Black, if status$i$ = Black, else s' = s
  - Δ' = Δ + count$i$
  - and set status$i$ = White

Without status flags: unable to detect "crossing" work

Without dirty flag

W0

idle

Δ=0

work

W1

Δ=0

idle

W2

terminated

active

idle

idle

active

work

idle

Δ=0

active

work

idle

March 30, 2010

Stefan Blom
23/ 28
Safra's Algorithm [EWD998]

With **only token status flag**: still fails when crossing work

- W0
- W1
- W2

\[ \begin{array}{c}
\text{work} \\
\text{idle} \\
\text{terminated}
\end{array} \]

**Principle of operation:**

- count in termination token is used to detect messages *in transit* (sent, but not received)
- status flags detect activity which occurred *after* termination token was sent by worker 0

**Correctness argument**

Consider last termination probe:

- All white processes are “clean”: nothing has been received in the left part
- All processes are white, thus token status remains white
- Token count \( \Delta \) is sum of messages sent, but not received before diagonal
- If message was sent in the left part, \( \Delta \) will be positive
- If \( \Delta = 0 \), then no message was sent in left part
- Moreover, no message was pending at start of probe
- Thus, in the right part, no message can be received that reactivates a worker

**Limitations Termination Detection?**

- Can we simplify the termination check?
  - Algorithms is asymmetric and not reentrant: worker 0 must initiate
  - Weakness of token-based algorithms: token can get lost…
  - Alternatives for the status flag?

- Can we speed up the termination check?
  - Last probe has still \( O(N) \) latency due to ring traversal
Gerard Tel, Friedemann Mattern: Comments on "Ring Based Termination Detection Algorithm for Distributed Computations". Information Processing Letters 31, pp. 127-128, 1989

"Unfortunately, more incorrect termination detection algorithms have been published in Information Processing Letters in the past [2,5], see also [9,10]. These examples clearly show the importance of a thorough, mathematically tight correctness proof for even the simplest distributed algorithm."