Runtime Systems for Threading & Communication in a Parallel Environment

Tao Huang
Nathan Thomas

CPSC 689, Spring 2003
PARASOL Laboratory
Texas A&M University
Outline

- Active Messages
- Nexus
- Converse
Active Messages
Active Messages - Goal

- Allow communication to overlap computation.
  - If program alternates between computation and communication phases:
    \[ T = T_{\text{compute}} + T_{\text{communicate}} = T_{\text{compute}} + N_c(T_s + L_c T_b) \]
  - If communication and computation are overlapped:
    \[ T = \max (T_{\text{compute}} + N_c T_s, N_c L_c T_b) \]
  - We want \( T_{\text{compute}} >> N_c T_s \) (communication overhead).
AM - Goal

- Lightweight async communication mechanism.
  - Minimize communication overhead.
  - Send simple messages to simple handlers.

- More efficient than message passing approach.
  - No need for protocol handshaking (less messages).
  - No need for receive buffers (and maybe send).

- More efficient than message driven approach.
  - Cast of thread creation, context switches.
  - Scalability of message queue.
AM - Design

- Message contains address of user-level handler which is executed on arrival with the message body as the argument.

<table>
<thead>
<tr>
<th>Addr of handler</th>
<th>message body (data)</th>
</tr>
</thead>
</table>

- Handler only extracts the data from network and integrate it into the ongoing computation.
- Dest. Processor needn’t to do specific receive action (compared to send/receive).
AM - Model

Network

Active Message
data pc

data structs
primary computation

data structs
primary computation

handl-
### AM – Cost of Communication

<table>
<thead>
<tr>
<th>Task</th>
<th>Instruction count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>send</td>
</tr>
<tr>
<td>Compose/consume message</td>
<td>6</td>
</tr>
<tr>
<td>Trap to kernel</td>
<td>2</td>
</tr>
<tr>
<td>Protection</td>
<td>3</td>
</tr>
<tr>
<td>Buffer management</td>
<td>3</td>
</tr>
<tr>
<td>Address translation</td>
<td>1</td>
</tr>
<tr>
<td>Hardware set-up</td>
<td>6</td>
</tr>
<tr>
<td>Scheduling</td>
<td>–</td>
</tr>
<tr>
<td>Crawl-out to user-level</td>
<td>–</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>21</strong></td>
</tr>
</tbody>
</table>

Instructions required for one word transfer on the nCUBE/2
AM - Implications

- Relies on a uniform code image on all nodes (SPMD).

- Handler should be simple, and guaranteed to end quickly, without blocking.

- Storage for arriving data (if needed) is pre-allocated in sender’s user program – elimination of buffering.

- Handler itself does not compute (compared to RPC).
AM - Limitations

- Supports only single-thread applications.

- Program needs to poll on systems with high interrupt costs.
  - Try to mask in communication calls.
  - May require explicit poll in user code.

- Handler cannot initiate communication or allocate memory.
AM – Possible Network Support

- Large message support.
  - DMA on receiver that examines header before copy.
- Message Registers
  - Very efficient support for small messages.
- Reuse of message data.
  - Create new message with old message’s memory.
  - Reuse source bits, other data.
- Issue accelerators for frequent messages.
AM – Possible Processor Support

- **Fast Polling**
  - Message-ready signal (condition code).
  - Compiler inserted polls.
- **User-level Interrupts**
- **Program Counter (PC) injection**
  - Simple multithreading, change PC on message arrival.
  - Avoids pipeline flushing.
- **Simple, dedicated communication processor**
AM - New Implementation

- Applications communicate among objects called “communication endpoint” – any two endpoints can communicate.
- Combination of endpoints and threads.
- Handler could do more things.
AM - Endpoints

- Applications exchange two types of Point-to-Point messages.
  - Request
  - Reply
- Handlers are associated with each message.
  - Executed upon receipt of message
AM – Endpoint Components

- #1: a send pool, for sending message from the endpoint
  - Random access to entries
  - May be dynamically resized
- #2: a receive pool, for receiving messages to the endpoint
  - Random access to entries
  - May be dynamically resized
AM – Endpoint Components

- #3: Handler Table, for translating handler indices (carried by messages) into functions
  - Sender doesn’t need to have a priori knowledge of the addresses of handlers
- #4: A Virtual Memory Segment, for receiving bulk memory transfers
  - Base addr. And byte length specify a window into the receiver’s addr. Space into which endpoints with valid tags can write
AM – Endpoint Components

- #5: Translation Table, array associating indices with global-endpoint names and tags
  - Operations are atomic
  - Each application can maintain different tables
- #6: Tag, for authenticating messages arriving at the endpoint
  - 64-bit integers
AM – Anatomy of Endpoint
AM – Endpoint Bundle

- An endpoint bundle is a set of endpoints created by one process.
- One process can create multiple endpoints, can communicate through each of them or the gathering of some of them.
- Enables different combinations between threads (in processes) and endpoints.
AM – Endpoint Bundle Components

- #1: collection of endpoints.
  - An endpoint is a member of exactly one bundle at any time
- #2: thread synchronization variable – binary semaphore
  - Posted upon event from any endpoint in the bundle, but the ID of this specific endpoint is unavailable to application
  - The communication subsystem notifies applications of communication events by using this variable
AM – Endpoint Bundle Components

• #3: an event mask, which is under the control of the application
  – Selects which endpoint states or state transitions will generate events and post the sync. variable

• #4: an access mode flag that indicates if concurrent use of the bundle or its member endpoints is expected
  – Informs a system if multiple application threads will access the bundle or its components concurrently
AM – Anatomy of Endpoint Bundle

Diagram illustrating the anatomy of an endpoint bundle, showing components such as network, endpoints, semaphore, event mask, and process connections.
AM – Combination of Threads and Endpoints

- Single thread / process, one thread – one endpoint
AM – Combination of Threads and Endpoints

- Multiple threads/process, one thread-one endpoint
AM – Combination of Threads and Endpoints

- single thread/process, one thread-multiple endpoints
AM – Combination of Threads and Endpoints

- multiple threads/process, multiple threads – one (shared) endpoint
The Nexus Runtime System
Nexus

- Supports MIMD type applications.
- Multilanguage support.
- Heterogeneous platforms, transport protocols.
- Users
  - Global Grid Toolkit
  - nPerl
  - CC++
Nexus – Existing Runtime Systems

- **Send/receive**
  - SPMD model; sender and receiver pair.
  - Designed for explicit programmer use, not for good compiler targets; process-based, not thread-based.

- **Active message**
  - Handler in message given by sender.
  - Interrupt to execute handler on destination.
Nexus – Basic Idea

- Try to integrate threads and communication
- Don’t require “receive” operations.
  - Communication between threads is asynchronous
- Flexible implementation for handlers, instead of simple interrupt-like handler in AM
- General purpose runtime system
Nexus – Abstractions

- Core Abstraction
  - Nodes – physical machines.
  - Contexts – memory address space.
  - Threads – execute in a context.

- Global Pointers (GPs)

- Remote Service Requests (RSRs)
Nexus
Nexus – Global Pointer

- GP = (Node, Context, Local Address).
- Specifies destination for remote messages.

![Diagram](image)
Nexus – Remote Service Request

- Remote Service Request (RSR) consists of:
  - Handler Identifier – remote function to invoke.
  - Global Pointer – determines transport protocol.
  - Message Buffer – input parameter to remote handler.

- RSR complexity
  - Lightweight, non-suspending – execute directly.
    - Forego cost of thread creation, buffer copying costs.
  - Heavy, possibly suspending – create new thread.
  - Explicitly chosen by programmer.
Nexus – RSR Scheduling: An Implementation Issue

- How are RSRs scheduled on the receiver?
  - Dependent on OS / platform features
    - Wake a suspended thread waiting on communication.
    - Preemptive CPU scheduling.
    - Thread Priorities.

- Nexus offers 4 implementations for various cases.
Nexus – RSR Scheduling (1)

- A Probing Communication Thread
  ```
  while (1) {
    probe_for_incoming_RSR
    yield
  }
  ```

- Performance depends on OS
  - Priority support:
    - RSR processing latency <= scheduler time slice.
  - Priority support:
    - RSR processing latency = (num threads * time slice).
Nexus – RSR Scheduling (2)

- Explicit probing by computation thread.
  - Inside communication library calls.
  - Compiler inserted.
  - By user in code.

- Yield to comm. thread vs. direct execution.

- Comm thread + explicit probing.
  - Ensure bounded interval between probes.
Nexus – RSR Scheduling (3)

- Blocking Communication Thread
  - Use when OS supports blocking while waiting for I/O.

```c
while (1) {
    blocking_receive;
    if RSR threaded
        create_thread(process_RSR);
    else
        process_RSR;
}
```

- Optimizing with priorities
  - Giving Communication thread higher priority reduces RSR processing latency.
  - Otherwise latency is (CPU quantum * num_threads).
Nexus – RSR Scheduling (4)

- Interrupt-driven
  - New RSRs generate an interrupt, scheduler invoked.
    - Non-threaded executed directly.
    - Threads created for threaded RSRs.
  - May restrict actions of function (like AM).
    - Memory allocation, creating new RSRs.
  - High RSR latency, only infrequent messaging.
Nexus - Performance

CC++ program using Nexus vs. C++ program using PVM.
Converse

A Runtime System for Charm++
Converse – Goals

- Provides runtime support for Charm++
  - Charm++ predates Converse, re-targeted for Converse.

- Multilanguage
  - Component-based, only use features needed.
  - Never really discussed, manuals imply C.

- Multiparadigm
  - SPMD, message-driven objects, multi-threaded.
Converse - Facilities

- Messaging
- Threading
  - Creation, Management
  - Scheduling, Load Balancing
- Utility Functions
  - Timers, Atomic I/O, Context (i.e. CmiMyRank())
  - Global Variables, Global Allocation
  - Node Barriers and Locks
- Client Server Interface (CCS)
Converse – Execution Environment

- Nodes, Processors, and Threads.

- Startup and Termination
  - Similar to MPI
  - Converse-Init(), Converse-Exit()
Converse - Messaging

• Transport mechanism for raw byte sequence.
  – CmiAlloc, CmiFree

• 1\textsuperscript{st} word of message represents handler id
  – Handler is a standard C function
    • Single void pointer parameter
    • Void return value
  – Handler id is integer index into handler table.

• No Return Values
  – nothing like stapl::sync_rmi()
Converse – Handler Registration

- Naïve Registration (Standard)
  - int CmiRegisterHandler(CmiHandler h)
  - Same order of handler registration in all processes.

- Collective Registration
  - int CmiRegisterHandlerGlobal(CmiHandler h)
  - Programmer calls above on thread 0, then broadcasts.
  - All other threads then call explicit registration (below).

- Explicit Registration
  - int CmiNumberHandler(int n, CmiHandler h)
  - Programmer explicitly register all objects on all threads.
Converse – Sending a Message

● Send Function Types
  – Sync
    ● Return after buffer is free.
    ● Message not guaranteed to be handled.
  – Async
    ● Return immediately, query for completion.
  – Send and Free
    ● Return immediately, frees buffer when complete
  – Node
    ● Send to an address space.
    ● Any processor can handle message.

● Send Methods
  – Single Receiver
  – Broadcast
  – Multicast
Converse – Receiving Messages

- **Automatic Polling**
  - Background process receives message and calls handler. No user intervention required.

- **Explicit Polling**
  - Message received only when user program calls poll().
  - User’s design must guarantee forward progress / deadlock avoidance.
  - CmiDeliverMsgs(), CmiGetSpecificMsg()
Converse – Parameter Marshalling

- Concise means of remotely invoking functions
- Relies on a C preprocessor
  - Converse Parameter Marshaller (CPM).
  - Process functions with CpmInvokable return type.
  - Creates Cpm_function_name in .cpm.h header file
  - First argument defines receiver
    - Point to Point and Broadcast (id, CPM_ALL, CPM_OTHERS)
    - Immediate, Queued, and Threaded.
Converse – Parameter Marshalling

#include "myprog.cpm.h"

CpmInvokable print_integer(int n) {
    CmiPrintf("%d\n", n);
}

user_main(int argc, char **argv) {
    Cpm_print_integer(CpmSend(2), rand());
    Cpm_print_integer(CpmMakeThread(CPM_ALL),
    Cpm_print_integer(CpmEnqueueFIFO(CPM_OTHER),rand());
}

main(int argc, char **argv) {
    ConverseInit(argc, argv, user_main, 0, 0);
}

PARASOL Lab: Texas A&M
Converse – Parameter Packing

- Single Argument vs. Array Argument
  - CpmInvokable print_program_arguments(
    CpmDim argc, CpmStr *argv)

- User provides packing / unpacking functions
  - Simple (non pointer)
    - CpmDeclareSimple(mytype);
    - CpmPack_mytype(mytype *p);
    - CpmUnpack_coordinate(mtype *p);
  - Pointer
    - CpmDeclarePointer(mytype);
    - void CpmPtrPack_intptr(void *p, mytype* v)
    - mytype CpmPtrUnpack_intptr(void *p)
Converse – IntraProcessor Messages

- Interface same as InterProcessor.
- Scheduling
  - FIFO & LIFO priority queue ([0..1]).
  - Invoked with poll().
  - Not Invoked until all Immediate InterProcessor messages completed.
    - CsdEnqueue(void *Message, int strategy, 
      int priobits, int* prioptr)

- IntraNode Queue
  - Combined with local queue for scheduling decisions.
Converse – Threading Constructs

● Basically heavyweight Intraproc messages.
  – Scheduled in same priority queue.
  – Threads are non-preemptible.
  – Termination, Yield, Suspend until awaken

● Implemented on top of QuickThreads
  – CthThread CthCreate(CthVoidF fn, void *arg, int size)
  – void CthAwaken(CthThread t)
  – void CthSuspend()
  – void CthAwaken(CthThread t)
  – void CthYield()
Converse – Custom Scheduling

● Define two function declarations
  - void awakenfn(CthThread t, int strategy, int priobits, int *prio)
  - CthThread choosefn();

● Choose thread for custom scheduling
  - CthSetStrategy(CthThread t, CthAwkFn awaeknfn, CthThnFn choosefn)
  - CthSetStrategyDefault(CthThread t)
Converse – The Scheduler Thread

Scheduler() {
    while (not done) {
        CmiDeliverMsgs();
        get a message from the scheduler’s queue;
        call the message’s handler;
        get a message from custom queues;
        call the message’s handler;
    }
}

CmiDeliverMsgs() {
    while (there are messages in the network) {
        receive message from the network;
        call the message’s handler;
    }
}
Questions?
References

- **Active Messages**
    http://now.cs.berkeley.edu/AM/am-spec-2.0.ps.

- **Converse**
  - The Converse Programming Manual. Parallel Programming Laboratory, University of Illinois at Urbana-Champaign.

- **Nexus**
  - http://www.globus.org/nexus