On the Portability and Performance of Message-Passing Programs on Embedded Multicore Platforms

Shih-Hao Hung, Po-Hsun Chiu, Chia-Heng Tu, Wei-Ting Chou and Wen-Long Yang

Graduate Institute of Networking and Multimedia
Department of Computer Science and Information Engineering
National Taiwan University
Outline

- Motivation
- The MeSsaGe (MSG) passing library
- Design for Performance
- Experimental Results
- Summary
Motivation

- Heterogeneous multi-core architectures have gained popularity for embedded systems
  - Power efficiency
  - High performance
- No de facto solution to inter-core communications for such multicore chips
  - Platform dependent communication mechanisms
    - Qualcomm Snapdragon, TI DaVinci, IBM Cell, ITRI PACDuo
  - Developers have to learn new API constantly
  - Hurts the portability of multicore applications
Message Passing Model – MPI vs. MCAPI

- **MPI 2.1:**
  - For inter-computer communication
  - Communication
    - Point-to-point, Collective operations, One-sided communications
  - Synchronization
    - Implicit synchronization included in synchronous communications
  - Additional features
    - E.g., Parallel I/O, Process and group management

- **MCAPI:**
  - Proposed by the Multicore Association in 2008
  - Designed for embedded multicore platforms with emphasis on network-on-chips
  - Functionalities: Messages, Overkill
  - Not widely adopted by the HW vendors
  - Reference implemen...
The MSG Library
MeSsaGe Passing (MSG) Library

- Following MPI’s *interface* and MCAPI’s *philosophy*
  - MPI has been already a mature and popular interface for parallel computing
  - Better compatibility and portability
Software Architecture

- Management functions:
  - init, exit, comm_rank, comm_size
- Resource management:
  - Managing internal data structures for data transmission
- OS, platform dependent functions:
  - mmap, shmget, DMA, memcpy, etc

Diagram:

- Application layer:
  - Management functions
  - Point-to-point communication
  - Collective communication
  - One-sided communication

- Core layer:
  - Resource management
  - Internal Protocols
    - Shared memory architecture
    - Distributed memory architecture

- Porting layer:
  - OS dependent configuration functions
  - Platform dependent configuration functions
## Supported MSG APIs

- **Three types of communication protocols:**
  - **Point-to-point communication**
    - Blocking & Non-blocking send/recv
  - **Collective communication**
    - barrier
    - One-to-many & many-to-one
  - **One-side communication**
    - Put & get

<table>
<thead>
<tr>
<th>MPI</th>
<th>MSG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocking point-to-point communication</td>
<td></td>
</tr>
<tr>
<td>MPI_Send</td>
<td>msg_send</td>
</tr>
<tr>
<td>MPI_Recv</td>
<td>msg_recv</td>
</tr>
<tr>
<td>Non-blocking point-to-point communication</td>
<td></td>
</tr>
<tr>
<td>MPI_Isend</td>
<td>msg_Isend</td>
</tr>
<tr>
<td>MPI_Irecv</td>
<td>msg_Irecv</td>
</tr>
<tr>
<td>Synchronization routines for non-blocking operations</td>
<td></td>
</tr>
<tr>
<td>MPI_Test</td>
<td>msg_test</td>
</tr>
<tr>
<td>MPI_Wait</td>
<td>msg_wait</td>
</tr>
<tr>
<td>One-sided communication</td>
<td></td>
</tr>
<tr>
<td>MPI_Get</td>
<td>msg_get</td>
</tr>
<tr>
<td>MPI_Put</td>
<td>msg_put</td>
</tr>
<tr>
<td>Collective communication</td>
<td></td>
</tr>
<tr>
<td>MPI_Scatterv</td>
<td>msg_scatterv</td>
</tr>
<tr>
<td>MPI_Gatherv</td>
<td>msg_gatherv</td>
</tr>
<tr>
<td>MPI_Bcast</td>
<td>msg_bcast</td>
</tr>
<tr>
<td>MPI_BARRIER</td>
<td>msg_barrier</td>
</tr>
</tbody>
</table>
Design for Performance
Important Considerations

1. Placement of relay buffer
2. Buffer management and lock-free design
3. Data movement mechanism
Part I: 
Placement of Relay Buffers (1/3)

☐ A intermediate buffer for relaying the data?

☐ For blocking communication

- Not required for synchronous communication
- Simple, efficient and save memory space

☐ For non-blocking communication

- Buffing mechanism becomes necessary:
  - decouple executions of sender and receiver
    - reduce wait time of the sender for start and termination of data transmission
  - avoid security problem by isolating sender/receiver
- Complex protocol handling, longer latency, etc.
Part I:
Placement of Relay Buffers (2/3)

- Typically, there are several memory modules available on the systems
  - With different capacities and access latencies
- Characterizing communication performance for different buffer placement schemes is necessary
Part I:
Placement of Relay Buffers (3/3)

- **ITRI PAC Duo Platform:**
  - **ARM + 2 PAC DSPs**
  - Three memory modules available for PAC DSP
    - DSP-side DRAM, SRAM, Local data memory
    - Place the messages on the different locations according to their sizes

![Diagram of PAC Duo Platform]
Part II:
Buffer Management & Lock-free Design

- Scalability affected by memory usage & lock-free design
- 1-to-1 buffering: Better to avoid lock handling
  - One receive buffer (RB) for each pair of core connections
  - \( N(N-1) \) RBs are required for a \( N \)-core system
- But need to minimize the memory usage at the same time
  - One receive buffer per core
  - The lock handling will impact the performance
Part II:

Proposed Buffering Schemes (1/2)

- **Buffer Pool scheme**:  
  - Request Queues (RQ): Store small-size control messages  
  - Relay Buffer Pool: Allows connections to share buffers in the pool

- **Service Processor scheme**:  
  - A dedicated core to serve as a service processor  
    - Forward the *request* for the sender to the receiver  
  - Reduce the number of required RQ, but require longer comm. latency (further optimization can improve the performance)
Part II:
Proposed Buffering Schemes (2/2)

- **Buffer Pool scheme**:  
  - Request Queues (RQ): Store small-size control messages  
  - Relay Buffer Pool: Allows connections to share buffers in the pool

- **Service Processor scheme**:  
  - A dedicated core to serve as a service processor  
    - Forward the request for the sender to the receiver  
  - Reduce the number of required RQ, but require longer comm. latency (further optimization can improve the performance)

| The memory footprint of 1-to-1 buffering, buffer pool | and service processor approach (MB) |
|---|---|---|---|---|---|---|
| #core | 2 | 4 | 8 | 16 | 32 | 64 | 128 |
| 1-to-1 buffering | 0.125 | 0.75 | 3.5 | 15 | 62 | 252 | 1016 |
| Buffer Pool | 0.13 | 0.25 | 0.52 | 1.11 | 2.48 | 5.97 | 15.93 |
| Service Processor | 0.13 | 0.25 | 0.5 | 1.01 | 2.03 | 4.06 | 8.13 |
Part II: Reduce Memory Footprint – Buffer Pool

- **Request Queues (RQ):** Store small-size control messages
- **Relay Buffer Pool:** Allows connections to share buffers in the pool
  - One relay buffer pool and N–1 request queues per core
  - N*(N–1) request queues + N relay buffer pools for a N-core system
Part II:
Alternative Solution – Service Processor

- Buffer pool approach needs N–1 request queues for each core
  - Still takes a lot of on-chip memory space
- **Service Processor** approach:
  - Delegate a core to serve as the service processor (HW or SW design)
  - Service processor has N **Request Forward Queue** (RFQ)
  - Each core has one relay buffer pool and one request queue
  - 2*N request queues + N buffer pools in total
Part III:
Data Movement Mechanism (1/2)

- Several ways to move the data:
  - Processor: via the memory copy operations
  - DMA engine: via I/O processor, or peripheral device

- It is common that DMA engine is often used to ease the burden on the processor
  - Processor: move small pieces of data quickly
  - DMA engine: better for large messages
  - What is considered to be small or large?
Part III:
Data Movement Mechanism (2/2)

- Determining the threshold for different data transmission schemes:
  - E.g., on the PAC Duo, ARM/DSP can move data between the two memory modules
  - E.g., EMDMA saves ~25% of time to move 64B data between SRAM and DRAM

The latencies for moving large data blocks between 4 bytes to 16KB using memcpy (via DSP) and EMDMA on the PAC Duo platform.

[Graph showing latency comparison between different methods for data movement, indicating a 31.6x speedup.]
Experimental Results
Data Parallel Application on IBM CELL for Scalability Test

- RC5: A block cipher
  - Encryption and Decryption
  - PPE (master) + 8 SPEs (workers)

- Three implementations on Cell:
  - IBM Cell Intrinsic (baseline)
    - DMA operations performed by hardware
  - Message-Passing w/ MSG library (send/recv)
    - Require both sender and receiver to participate
  - One-sided Communication w MSG library (put/get)
    - Introduced to improve the performance on shared-memory
Scalability of MSG

- Communication schemes affects performance & scalability
  - Programming ability vs. performance

- Cell Intrinsic: 100%
  - Good scalability
  - DMA hardware overlaps communication and computation

- Send–Recv: 85%~57%
  - Acceptable scalability
  - PPE becomes bottleneck

- Put–Get: 99%~91%
  - Good scalability
  - Take advantage of DMA
  - The overhead is caused by extra efforts made for handling data size and alignment problems
Pipelined Execution Scheme on ITRI PAC Duo

- Streaming multimedia application
  - The handheld device will:
    - Receive encrypted data stream,
    - Decrypt received data,
    - Decode and display the data
Pipelined Execution Scheme on ITRI PAC Duo (2/2)

- Inter-core communications are done by MSG one-sided functions
- Double buffering schemes allow overlapped communication and computation on the DSP cores
- Experimented with two different configurations:
  - Solo ARM: 2.2 frames/second
  - Offload SSL/JPEG for pipelined execution: 14.6 frames/second
- We are exploring different offloading schemes
Summary
Summary

- We share the experiences for designing the MSG library for embedded multicore systems
- We implemented MSG on several platforms: IBM CELL, ITRI PAC Duo, TI Davinci, and x86/64
- The scalability and applicability of the MSG are presented with case studies
- Currently, the MSG implementation is available on the OpenFoundry; any feedback is welcomed