A Middleware for Concurrent Programming in MPI Applications

Tobias Berka, Helge Hagenauer and Marian Vajteršic

September 13, 2011
Outline

1 Introduction
   - Emergent Parallel Applications
   - The Need for Concurrency

2 Programming Model
   - Concurrency using Threads
   - Thread Collectives
   - In Actual Use

3 The MPI Threads API
   - The MPIT Interface Definition
   - Constructs and Features
   - Performance Overhead

4 Summary & Conclusions
Introduction
Parallelism is abundant in today’s data centers:
- Multi-core CPUs,
- High-bandwidth low-latency interconnection networks,
- Accelerator hardware.

Exciting new applications in today’s information economy:
- Information retrieval (i.e. search),
- Online analytical processing,
- Recommender systems,
- Data mining.
Use Case: Parallel Search Engine

- Requirements beyond the classic batch-job operation:

  - Add Document
  - Update Document
  - Remove Document
  - Query Documents
We group similar operations – short and long:
The Need for Concurrency

- Multi-User Operation
  - Multiple users,
  - Single back-end,
  - Single data base...
  ⇒ We need concurrency!

- Both layers can be used concurrently,
  - At the same time:
    - Answer queries,
    - Modify the data base,
  ⇒ We need concurrency!
Programming Model
How do we implement concurrent activities?

- Operations and queues:
  - Data structure to describe operations,
  - Queue holds operations,
  - “Main loop” pops operations and processes them.

- Threads:
  - One activity = one thread.
The pros and cons...

- **Operations and queues:**
  - Efficient,
  - No true concurrency,
  - Cannot process operation and receive independent messages,

- **Threads:**
  - Context switching overhead,
  - Shared data requires locking,
  - Very tidy abstraction,
  - Compositional (can always add more threads).
Use Case: Parallel Search Engine

Let’s use threads to implement these concurrent activities:

- Add Document
- Update Document
- Remove Document
- Query Documents
Programming Abstraction

- Key abstraction: thread collective,

- Goals:
  - Encapsulate concurrent activities,
  - Isolate concurrent communication,
  - Unify and simplify the design.

- Conflicting objectives:
  - Safety and ease of programmability,
  - Performance.
Thread Collectives

- Creates a new thread within every MPI process ($T_1 \rightarrow T_2$),
- Assigns a copy of the MPI communicator ($C_1 \rightarrow C_2$),
• Encapsulates computation: thread function(s) for \( T_2 \),
• Isolates communication: communicator \( C_2 \).
Parallel Search Engine

- $P_1$–$P_4$ each hold a part of all documents,
- $T_1$s: Answer queries (query layer),
- $T_2$s: Add, remove or update documents (maintenance layer).

![Diagram showing parallel search engine with $P_1$ to $P_4$ and $T_1$ and $T_2$ processes.](image-url)
What do we get?

- Simple, ready-made abstraction,
- Encapsulate & isolate,
- Compositional,
- Caveat: synchronization and locking.
The MPI Threads API
The MPIT Interface Definition

- Additional layer of middleware to provide what we need,
- Designed as a library for compatibility (not a new programming language),
- The “MPI Threads” (MPIT) interface definition,
- Written as a single C header file (157 physical SLOC\(^1\)).

\(^1\)According to David A. Wheeler’s “SLOCCount”. 
Constructs and Features

- Thread collectives,
  - One thread within every MPI process,
  - Separate MPI communicator,
- Conventional threads,
  - Portable thread interface,
  - We get it “for free” – we have all of the machinery.
- Process-local synchronization constructs,
  - Mutex locks, condition variables, semaphores and barriers,
  - Specifies reliable semantics.
Performance Overhead

- Additional layer of indirection (1 additional function call),
- Additional error and consistency checks:
  - Condition variable checks spurious wake-up,
  - Barrier verifies thread identity.

⇒ Execution time overhead.

- MPIT prototype implemented on top of POSIX threads (2,650 physical SLOC\(^2\)).

\(^2\)According to David A. Wheeler’s “SLOCCount”.
Thread Creation

- Difference: additional indirection.

create and join threads

<table>
<thead>
<tr>
<th>milliseconds</th>
<th>0</th>
<th>0.02</th>
<th>0.04</th>
<th>0.06</th>
<th>0.08</th>
<th>0.1</th>
<th>0.12</th>
<th>0.14</th>
</tr>
</thead>
<tbody>
<tr>
<td>threads</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MPIT prototype
POSIX threads
Difference: additional checks.

lock and unlock mutex

milliseonds

22 / 26
Wait / Wake on Condition Variable

- Difference: indirection & checks (different time scale).

![Graph showing comparison between MPIT prototype and POSIX threads in milliseconds for wait/wake on condition variable.](attachment:graph.png)

- The graph depicts the performance overhead for wait/wake operations on condition variables for both MPIT prototype and POSIX threads. The y-axis represents milliseconds, while the x-axis represents the number of threads.
Summary & Conclusions
Summary & Conclusions

- Parallel programs may require additional concurrency,
- New abstraction: thread collectives,
- MPIT interface specification,
- Performance penalty is acceptable...
- ...unless too many locks are used.
Thank you!