Why do we need another programing model?

Atsushi Hori  Min Si
Riken  ANL

B. Gerofi, M. Takagi, Y. Ishikawa (RIKEN)
J. Dayal (Intel), P. Balaji (ANL)
Thursday, 14 June

10:30 - 12:00  Session 4 - Runtime Systems (Memorial Union Ventana B&C)

**PShifter: Feedback-based Dynamic Power Shifting within HPC Jobs for Performance**
Neha Gholkar, Frank Mueller (North Carolina State University); Barry Rountree, Aniruddha Prakash Marathe (Lawrence Livermore National Laboratory)

**ADAPT: An Event-based Adaptive Collective Communication Framework**
Xi Luo (University of Tennessee, Knoxville); Wu Wei (Los Alamos National Laboratory); George Bosilca, Thananon Patinyasakdikul, Jack Dongarra (University of Tennessee, Knoxville); Linnan Wang (Brown University)

**Process-in-Process: Techniques for Practical Address-Space Sharing**
Atsushi Hori (RIKEN); Min Si (ANL); Balazs Gerofi, Masamichi Takagi (RIKEN); Jai Dayal (Intel);
Pavan Balaji (ANL); Yutaka Ishikawa (RIKEN)
Outline

• Multi-process and Multi-thread
  • Historical background
• Motivation
• New Execution Model
• Process-in-Process (PiP)
• Showing some numbers
Multi-Process

- **Beginning**
  - Multi-programming
    - Running “independent” programs at the same time
  - Multi-tasking and Time-sharing
    - Utilizing CPU idle time
- **Nowadays (in HPC)**
  - Running “familiar” programs
  - No need of utilizing idle CPU time (busy-wait)
  - Frequent communication among processes
    - IPC (e.g., pipes, sockets, ...) is too heavy
    - Shared memory is better, but ...
Multi-Thread

• **Beginning**
  • Interacting *Oversubscribed* Execution Entities
  • “Light-weight” process
    • Fast creation
      • Not loading and linking a program, but creating new context (incl. stack)
    • Easy to exchange information

• **Nowadays**
  • Its creation is still heavy
    • not to create threads on-demand
  • No *oversubscription*
  • Shared variables must be protected
My Experience

• A decade ago, developing low-level intra-node communication library for MPI
• By using shared mmap
  • Not easy at all !!
    • Setup part is NOT easy
    • Communication part is easy

• Wait, something is wrong
  • A process cannot access the other process
  • Processes access the same PHYSICAL memory !!
  • It is the OS to create the inter-process barrier
And Many-Core

• More parallelism in a node
  • from $10^0$ to $10^2$ (or more)

• More interaction between processes or threads
  • Multi-Process: Hard to communicate
  • Multi-Thread: Shared variables must be protected

• We need something new (if you are not happy)
  • Easy to communicate
  • No shared variables
Shared Memory and XPMEM

- “Hole in the wall” to go through the barrier
- Need of 2 copies to pass data
- Pointers in the shared memory are useless

- Setup (creation) cost
- Need of page table entries to map
- Coherency (page fault) overhead

![Diagram showing shared physical memory and page tables]
Let’s Break the Wall!

• Not making a tiny hole in the wall, but removing the whole wall!!!

• Removing the **walls** between processes
  • Keep variables private as in the same way of multi-process
    ➔ Easy to exchange data as easy as multi-thread because there is no **wall**

**AND**

• Build another **fence** between threads
  • Make variables private to each thread
  ➔ No need of protection on shared variables
### 3rd Execution Model

<table>
<thead>
<tr>
<th>Variables</th>
<th>Address Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Privatized</td>
<td>Isolated</td>
</tr>
<tr>
<td></td>
<td>Multi-Process (MPI)</td>
</tr>
<tr>
<td></td>
<td>3rd Exec. Model</td>
</tr>
<tr>
<td>Shared</td>
<td>Shared</td>
</tr>
<tr>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>Multi-Thread (OpenMP)</td>
</tr>
</tbody>
</table>
Implementation

• This idea is not new

• Pack processes into one virtual address space
  • SMARTMAP (SNL)
  • PVAS (Riken)

• Threads pretending processes
  • MPC (CEA)
    • Need of special compiler to privatize variables, converting static variables to TLS variables
Make it more practical and portable

• No need of virtual address space partitioning
  • Only OS can partition virtual address space

• Process-in-Process (PiP)
  • User-level library
  • Implementation
    • \texttt{dlmopen()} to privatize variables
    • create execution entities (processes or threads) to share the same virtual address space
      • i.e., \texttt{clone()} or \texttt{pthread_create()}
    • PiP programs must be PIE so that dlmopen() can load programs in different locations
/proc/*/maps example of PiP

Program

Glibc
# 3rd Execution Model

<table>
<thead>
<tr>
<th>Variables</th>
<th>Address Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Privatized</td>
<td>Isolated: Multi-Process (MPI)</td>
</tr>
<tr>
<td>Shared</td>
<td>Isolated: N/A</td>
</tr>
</tbody>
</table>
Sharing a Page Table

- Do PiP tasks and the root share the same page table?
- Evaluation of switching two tasks using futex


![Graph showing the relationship between working set size and context switch overhead for different processes.](attachment:graph.png)

**Xeon E5-2650 v2 8×2(×2) 2.6GHz 64 GiB**

**Number of load_cr3 function calls**

<table>
<thead>
<tr>
<th></th>
<th>PiP</th>
<th>Pthread</th>
<th>Fork</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIP</td>
<td>74.1</td>
<td>53.0</td>
<td>794535.4</td>
</tr>
</tbody>
</table>

**Figure 9: dTLB Miss distribution**

1,000 samples
How PiP works

• Execution Model
  • PiP Root Process
    • Root can spawn PiP tasks in the same virtual address space of the root
  • PiP Tasks
    • spawned by the root

• Execution Mode
  • Process mode
    • Tasks are created by clone()
  • Thread mode
    • Tasks are created by pthread_create()
    • Variables are privatized though
PiP vs. Shared Memory

- Setup Cost
- Page Table Size
- Number of Page Faults
We used four experimental platforms to cover several OS kernels. The Linux kernel on the K computer is old, and we gave up trying. In this microbenchmark, the root task created and initialized a 2 GiB thread execution mode. A combination described in Section 3.3, the PiP programs ran in the cores out of 16, and the Linux kernel ran on the remaining 2 cores. Only eight cores, thus PiP without the patched Glibc can still utilize and CPU architectures in our evaluation, as listed in Tables 2 and 3.  

### 5 EXPERIMENTAL SETTING

We evaluate the characteristics of PiP by using a set of in-house communication performance. Since the current McKernel is unable to handle the longer satisfy the network throughput, especially on many-core architectures where performance highly relies on the concurrence instruction — something no other existing technique can achieve, as Wallaby.

<table>
<thead>
<tr>
<th>Table 3: Experimental platform software information</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OS</strong></td>
</tr>
<tr>
<td>OFP w/ patch</td>
</tr>
<tr>
<td>Wallaby w/ patch</td>
</tr>
<tr>
<td>OFP w/o patch</td>
</tr>
</tbody>
</table>

### 6.2 Page Fault Overhead

The majority of hybrid MPI+Threads-based applications still process and thread only. We report the results of each experiment by averaging 10 executions.  

<table>
<thead>
<tr>
<th>Table 2: Experimental platform hardware information</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU</strong></td>
</tr>
<tr>
<td>OFP</td>
</tr>
<tr>
<td>Wallaby</td>
</tr>
</tbody>
</table>

**Xeon E5-2650 v2 8×2(x2) 2.6GHz 64 GiB**

| **Array Elements [Byte offset]** |
|---------------------------------
| Array | XPMEM | POSIX Shmem |
| 0 | 0 | 0 |
| 8,192 | 1,585 | 5,553 |
| 16,384 | 4,080 | 2,414 |
| 22,294 | 5,553 | 2,414 |

<table>
<thead>
<tr>
<th><strong>Offset of XPMEM and POSIX shmem functions</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Offset</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>8,192</td>
</tr>
<tr>
<td>16,384</td>
</tr>
<tr>
<td>22,294</td>
</tr>
</tbody>
</table>

**Allocation of 2 GiB Shared Memory**

<table>
<thead>
<tr>
<th><strong>Function</strong></th>
<th><strong>Cycles</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>xpmem_make()</td>
<td>1,585</td>
</tr>
<tr>
<td>xpmem_get()</td>
<td>15,294</td>
</tr>
<tr>
<td>xpmem_attach()</td>
<td>2,414</td>
</tr>
<tr>
<td>xpmem_detach()</td>
<td>19,183</td>
</tr>
<tr>
<td>xpmem_release()</td>
<td>693</td>
</tr>
</tbody>
</table>

**POSIX Shmem**

<table>
<thead>
<tr>
<th><strong>Function</strong></th>
<th><strong>Cycles</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>shm_open()</td>
<td>22,294</td>
</tr>
<tr>
<td>ftruncate()</td>
<td>4,080</td>
</tr>
<tr>
<td>mmap()</td>
<td>5,553</td>
</tr>
<tr>
<td>close()</td>
<td>6,017</td>
</tr>
</tbody>
</table>

**Xeon E5-2650 v2 8×2(x2) 2.6GHz 64 GiB**

ROSS 2018 at Tempe, AZ
The purpose of this microbenchmark is to measure this “hidden” loading time, and its cost is hidden from the time for accessing it.

In PiP, all memory mappings were done at the program startup. This contrasted with the need for separate PT entries for each memory region in Pthread.

### 6.4 Spawning Time

Figure 6 summarizes the number of PT entries required for each technique.

**Figure 6: Total page table size running on Wallaby/Linux**

**Table 5: Total number of page table entries**

<table>
<thead>
<tr>
<th>Technique</th>
<th>Total Number of Page Table Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pthread</td>
<td>( M + D + \sum S_i )</td>
</tr>
<tr>
<td>PiP</td>
<td>( M + \sum D_i + \sum S_i )</td>
</tr>
<tr>
<td>Process + POSIX shmem</td>
<td>( (M \times N) + \sum D_i + \sum S_i )</td>
</tr>
<tr>
<td>Process + XPMEM</td>
<td>( (M \times N) + \sum D_i + \sum S_i )</td>
</tr>
</tbody>
</table>

- \( M \) is the number of PT entries for the shared-memory region(s).
- \( S_i \) is the number of PT entries for the stack segment of task \( i \).
- \( D_i \) is the number of PT entries to map shared objects belonging to task \( i \).
- \( N \) is the number of tasks (processes or threads).
Page Fault

- Sender allocates memory region and set some values
- Receiver scan the data in the “shared” memory region
- Measure the each access time on the reciever
PiP Applications

- PiP application performance numbers will be shown in the main conference talk

- MPI
  - pt2pt communication
  - MPI_Win_alloate_shared()

- In-situ
  - By putting simulation program and in-situ program in the same virtual address space
  - 2 memory copies can be avoided

- MPI+OpenMP vs. MPI+PiP
Myths on PiP

• It is crazy to mix programs, I cannot debug!
  • Can’t you debug multi-thread programs?

• By using huge pages, PiP has no advantage!
  • PiP can work with huge pages
  • Pit falls of using huge pages
    • Transparent Huge Pages may hinder execution
    • Other Huge Page techniques need extra programming
    • Consumes more memory

• Shared memory is enough
  • PiP can do better than shared memory
PiP Summary

- 3rd parallel execution model
- User-level Implementation
  - No partitioning of virtual address space
  - `dlmopen()`, PIE, and `clone()`
  - Load multi-programs into the same virtual address space
- No communication (≈ copy), but accessing (no copy) by sharing virtual address space
# Comparison

<table>
<thead>
<tr>
<th></th>
<th>Multi-Process</th>
<th>Multi-Thread</th>
<th>3rd Execution Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel Execution</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Sharing</td>
<td>nothing shared</td>
<td>VAS and variables</td>
<td>VAS</td>
</tr>
<tr>
<td>Execution starts</td>
<td>main</td>
<td>arbitrary func</td>
<td>main</td>
</tr>
<tr>
<td>Multi-programming</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>SMARTMAP and PVAS</td>
<td>MPC</td>
<td>PiP</td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td>------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>VAS sharing</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Based on</td>
<td>process</td>
<td>thread</td>
<td>process or thread</td>
</tr>
<tr>
<td>Multi-programming</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Implementation</td>
<td>Kernel</td>
<td>Language</td>
<td>Library</td>
</tr>
<tr>
<td>Execution starts</td>
<td>main</td>
<td>any func</td>
<td>any func</td>
</tr>
</tbody>
</table>