



Process-in-Process: Techniques for Practical Address-Space Sharing

**HPDC'18, Tempe Arizona
June 14, Arizona**

Atsushi Hori (Riken) and Min Si (ANL)
B. Gerofi, M. Takagi, Y. Ishikawa (Riken),
J. Dayal (Intel), P. Balaji (ANL)

Motivation

- Conventional In-node Parallel Execution Models
 - Multi-Process: MPI
 - Multi-Thread: OpenMP
- Pros
 - Multi-Process: Privatized variables
 - Multi-Thread: Easy to exchange info.
- Cons
 - Multi-Process: Hard to exchange data
 - Multi-Thread: Locks on shared variables
- The rise of Many-Core
 - makes above cons worse

Conventional Parallel Execution Models

		Address Space	
		Isolated	Shared
Variables	Privatized	Multi-Process (MPI)	
	Shared	N/A	Multi-Thread (OpenMP)

- Pros

- Multi-Process: Privatized variables
- Multi-Thread: Easy to exchange data

- Cons

- Multi-Process: Hard to exchange data
- Multi-Thread: Locks on shared variables

Take the best of two worlds

		Address Space	
		Isolated	Shared
Variables	Privatized	Multi-Process (MPI)	<i>3rd Exec. Model</i>
	Shared	N/A	Multi-Thread (OpenMP)

- Pros
- 3rd Model

Privatized variables
Easy to exchange data

- Cons

- Multi-Process:

Hard to exchange data

- Multi-Thread:

Locks on shared variables

Take the best of two worlds

		Address Space	
		Isolated	Shared
Variables	Privatized	Multi-Process (MPI)	<i>3rd Exec. Model</i>
	Shared	N/A	Multi-Thread (OpenMP)

- Pros
- 3rd Model
- Cons

Privatized variables
Easy to exchange data

Variables and Data are SHARABLE

3rd Model Implementations

- Two Approaches
 - Process Approach
 - SMARTMAP (SNL) Kitten OS
 - PVAS (Riken) Patched Linux
 - Thread Approach
 - MPC (CEA) Compiler
- None of them is portable nor practical
- Our Goal:
 - Can we implement it at user-level ?
 - more portable and practical

Process-in-Process (PiP)

- User-Level Implementation
- 3 Technical Elements
 - 1. `dlmopen()` - not `dlopen()`
 - creates a new (symbol) name space for linking
 - can privatize statically allocated variables
 - 2. **Position Independent Executable (PIE)**
 - can load multiple programs in the same virtual address space
 - 3. `clone()` or `pthread_create()`
 - creates kernel threads to run on the same virtual address space
- PiP has only 3K LOC

/proc/*/maps Example of PiP

```
55555554000-55555556000 r-xp ... /PIP/test/basic  
555555755000-555555756000 r--p ... /PIP/test/basic  
555555756000-555555757000 rw-p ... /PIP/test/basic  
555555757000-555555778000 rw-p ... [heap]  
7ffffe800000-7ffffe8021000 rw-p ...  
7ffffe8021000-7ffffec00000 ---p ...  
7fffff0000000-7fffff0021000 rw-p ...  
7fffff0021000-7fffff4000000 ---p ...  
7fffff4b24000-7fffff4c24000 rw-p ...  
7fffff4c24000-7fffff4c27000 r-xp ... /PIP/lib/libpip.so  
7fffff4c27000-7fffff4e26000 ---p ... /PIP/lib/libpip.so  
7fffff4e26000-7fffff4e27000 r--p ... /PIP/lib/libpip.so  
7fffff4e27000-7fffff4e28000 rw-p ... /PIP/lib/libpip.so  
7fffff4e28000-7fffff4e2a000 r-xp ... /PIP/test/basic  
7fffff4e2a000-7fffff5029000 ---p ... /PIP/test/basic  
7fffff5029000-7fffff502a000 r--p ... /PIP/test/basic  
7fffff502a000-7fffff502b000 rw-p ... /PIP/test/basic  
7fffff502b000-7fffff502e000 r-xp ... /PIP/lib/libpip.so  
7fffff502e000-7fffff522d000 ---p ... /PIP/lib/libpip.so  
7fffff522d000-7fffff522e000 r--p ... /PIP/lib/libpip.so  
7fffff522e000-7fffff522f000 rw-p ... /PIP/lib/libpip.so  
7fffff522f000-7fffff5231000 r-xp ... /PIP/test/basic  
7fffff5231000-7fffff5430000 ---p ... /PIP/test/basic  
7fffff5430000-7fffff5431000 r--p ... /PIP/test/basic  
7fffff5431000-7fffff5432000 rw-p ... /PIP/test/basic  
...  
7fffff5a52000-7fffff5a56000 rw-p ...  
...  
7fffff5c6e000-7fffff5c72000 rw-p ...  
7fffff5c72000-7fffff5e28000 r-xp ... /lib64/libc.so  
7fffff5e28000-7fffff6028000 ---p ... /lib64/libc.so  
7fffff6028000-7fffff602c000 r--p ... /lib64/libc.so  
7fffff602c000-7fffff602e000 rw-p ... /lib64/libc.so  
7fffff602e000-7fffff6033000 rw-p ...  
7fffff6033000-7fffff61e9000 r-xp ... /lib64/libc.so  
7fffff61e9000-7fffff63e9000 ---p ... /lib64/libc.so  
7fffff63e9000-7fffff63ed000 r--p ... /lib64/libc.so  
7fffff63ed000-7fffff63ef000 rw-p ... /lib64/libc.so  
7fffff63ef000-7fffff63f4000 rw-p ...  
7fffff63f4000-7fffff63f5000 ---p ...  
7fffff63f5000-7fffff6bf5000 rw-p ... [stack:10641]  
7fffff6bf5000-7fffff6bf6000 ---p ...  
7fffff6bf6000-7fffff73f6000 rw-p ... [stack:10640]  
7fffff73f6000-7fffff75ac000 r-xp ... /lib64/libc.so  
7fffff75ac000-7fffff77ac000 ---p ... /lib64/libc.so  
7fffff77ac000-7fffff77b0000 r--p ... /lib64/libc.so  
7fffff77b0000-7fffff77b2000 rw-p ... /lib64/libc.so  
7fffff77b2000-7fffff77b7000 rw-p ...  
...  
7fffff79cf000-7fffff79d3000 rw-p ...  
7fffff79d3000-7fffff79d6000 r-xp ... /PIP/lib/libpip.so  
7fffff79d6000-7fffff7bd5000 ---p ... /PIP/lib/libpip.so  
7fffff7bd5000-7fffff7bd6000 r--p ... /PIP/lib/libpip.so  
7fffff7bd6000-7fffff7bd7000 rw-p ... /PIP/lib/libpip.so  
7fffff7dd000-7fffff7dfc000 r-xp ... /lib64/ld.so  
7fffff7edc000-7fffff7fe000 rw-p ...  
7fffff7ff7000-7fffff7ffa000 rw-p ...  
7fffff7ffa000-7fffff7ffc000 r-xp ... [vdsos]  
7fffff7ffc000-7fffff7ffd000 r--p ... /lib64/ld.so  
7fffff7ffd000-7fffff7ffe000 rw-p ... /lib64/ld.so  
7fffff7ffe000-7fffff7fff000 rw-p ...  
7fffffffde000-7fffffff000 rw-p ... [stack]  
ffffffffffff600000-ffffffffffff601000 r-xp ... [vsyscall]
```

OpenMP, MPI and PiP

OpenMP

```
int main() {
    int tid, seed, i;
    srand(1);
#pragma omp parallel ...
{
    seed = 1;
    tid = omp_get_thread_num();
    for ( i=0; i<3; i++ ) {
#pragma omp barrier
        printf( "<%d> %d : %d\n", tid,
            rand(), rand_r(&seed) );
    }
    return 0;
}
```

MPI

```
int main( ... ) {
    int i, rank, seed = 1;
    MPI_Init( ... );
    MPI_Comm_rank( ..., &rank );
    srand(1);
    for ( i=0; i<3; i++ ) {
        MPI_Barrier( ... );
        printf( "<%d> %d : %d\n", rank,
            rand(), rand_r(&seed) );
    }
    MPI_Finalize();
    return 0;
}
```

PiP

```
pthread_barrier_t barrier, *barrp;
int main() {
    int i, pipid, seed = 1;
    pip_init( &pipid, ... );
    if ( pipid == 0 ) {
        pthread_barrier_init( &barrier,
            NULL, ntasks );
        pip_export( &barrier );
        barrp = &barrier;
    } else {
        do {
            pip_import( 0, (void**) &barrp );
        } while( barrp == NULL );
        srand(1);
        for ( i=0; i<3; i++ ) {
            pthread_barrier_wait( barrp );
            printf( "<%d> %d : %d\n", pipid,
                rand(), rand_r(&seed) );
        }
        pip_fin();
        return 0;
    }
}
```

OpenMP, MPI and PiP

OpenMP

```
int main() {
    int tid, seed, i;
    srand(1);
#pragma omp parallel ...
{
    seed = 1;
    tid = omp_get_thread_num();
    for ( i=0; i<3; i++ ) {
#pragma omp barrier
        printf( "<%d> %d : %d\n", tid,
            rand(), rand_r(&seed) );
    }
    return 0;
}
```

MPI

```
int main( ... ) {
    int i, rank, seed = 1;
    MPI_Init( ... );
    MPI_Comm_rank( ..., &rank );
    srand(1);
    for ( i=0; i<3; i++ ) {
        MPI_Barrier( ... );
        printf( "<%d> %d : %d\n", rank,
            rand(), rand_r(&seed) );
    }
    MPI_Finalize();
    return 0;
}
```

PiP

```
pthread_barrier_t barrier, *barrp;
int main() {
    int i, pipid, seed = 1;
    pip_init( &pipid, ... );
    if ( pipid == 0 ) {
        pthread_barrier_init( &barrier,
NULL, ntasks );
        pip_export( &barrier );
        barrp = &barrier;
    } else {
        do {
            pip_import( 0, (void**) &barrp );
        } while( barrp == NULL );
        srand(1);
        for ( i=0; i<3; i++ ) {
            pthread_barrier_wait( barrp );
            printf( "<%d> %d : %d\n", pipid,
                rand(), rand_r(&seed) );
        }
        pip_fin();
        return 0;
    }
}
```

PiP Programming

OpenMP

```
<0> 1681692777 : 476707713
<0> 424238335 : 1186278907
<0> 596516649 : 505671508
<1> 1804289383 : 476707713
<1> 1957747793 : 1186278907
<1> 1649760492 : 505671508
<2> 846930886 : 476707713
<2> 1714636915 : 1186278907
<2> 719885386 : 505671508
```

rand()

rand_r()

MPI and PiP

```
<0> 1804289383 : 476707713
<0> 846930886 : 1186278907
<0> 1681692777 : 505671508
<1> 1804289383 : 476707713
<1> 846930886 : 1186278907
<1> 1681692777 : 505671508
<2> 1804289383 : 476707713
<2> 846930886 : 1186278907
<2> 1681692777 : 505671508
```

rand()

rand_r()

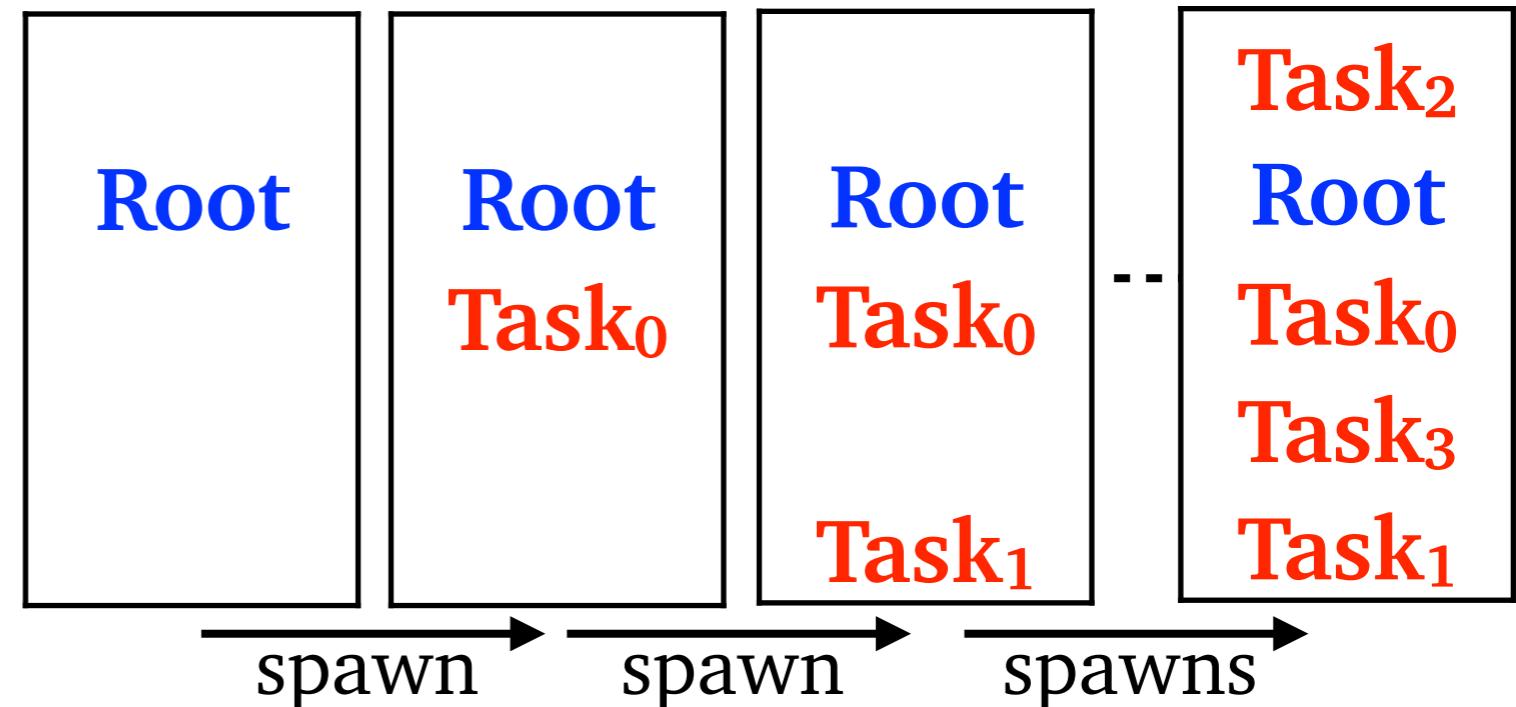
- **pthread_barrier (and pthread_mutex) works with PiP**
 - **VAS is shared**
- **The sequence of rand() is the same with MPI**
 - **Variables are privatized**
- **pip_import() and pip_export()**

PiP

```
pthread_barrier_t barrier, *barrp;
int main() {
    int i, pipid, seed = 1;
    pip_init( &pipid, ... );
    if ( pipid == 0 ) {
        pthread_barrier_init( &barrier,
                              NULL, ntasks );
        pip_export( &barrier );
        barrp = &barrier;
    } else {
        do {
            pip_import( 0, (void**) &barrp );
        } while( barrp == NULL );
        srand(1);
        for ( i=0; i<3; i++ ) {
            pthread_barrier_wait( barrp );
            printf( "<%d> %d : %d\n", pipid,
                    rand(), rand_r(&seed) );
        }
        pip_fin();
        return 0;
    }
}
```

How PiP Works

- Root Process
 - Spawn Tasks
- PiP Task
 - Share the same VAS with root
 - PIE



- PiP Execution Mode

- Process mode
- Thread mode

`clone()`
`pthread_create()`

- ✓ In either mode, variables are privatized
- ✓ `pthread_create()` is more portable than `clone()`

Potability

Table 2: Experimental platform hardware information

Name	CPU	# Cores	Clock	Memory	Network
Wallaby	Xeon E5-2650 v2	8×2(×2)	2.6GHz	64 GiB	ConnectX-3
OFP [†]	Xeon Phi 7250	68(×4)	1.4GHz	96(+16) GiB	Omni-Path
K [44]	SPARC64 VIIIfx	8	2.0GHz	16 GiB	Tofu

Table 3: Experimental platform software information

Name	OS	Glibc	PiP Exec. Mode(s)
Wallaby	Linux (CentOS 7.3)	w/ patch	process and thread
Wallaby	McKernel+CentOS 7.3	w/ patch	thread only
OFP [†]	Linux (CentOS 7.2)	w/ patch	process and thread
K	XTCOS	w/o patch	process and thread

- ✓ Glibc Patch: `dlopen()` can create up to 16 name spaces
- ✓ McKernel is a multi-kernel developed at Riken
- PiP also works on Arm64/Linux

/proc/*/maps Example of PiP

The diagram illustrates the memory layout of a process using the `/proc/*/maps` file. It shows various memory regions with their addresses, permissions (r=read, w=write, x=execute), and names.

Program: The main executable's memory space, containing code and data. It includes sections like `/PIP/test/basic`, `/PIP/lib/libpip.so`, and `/PIP/test/basic`.

Glibc: The standard C library's memory space, containing shared libraries like `/lib64/libc.so`.

Stack: The stack grows downwards from high addresses. It is labeled `[stack:10641]` at address `7ffff6bf5000` and `[stack:10640]` at address `7ffff73f6000`.

Libraries: Shared libraries used by the program, such as `/lib64/ld.so` and `/lib64/vdso`.

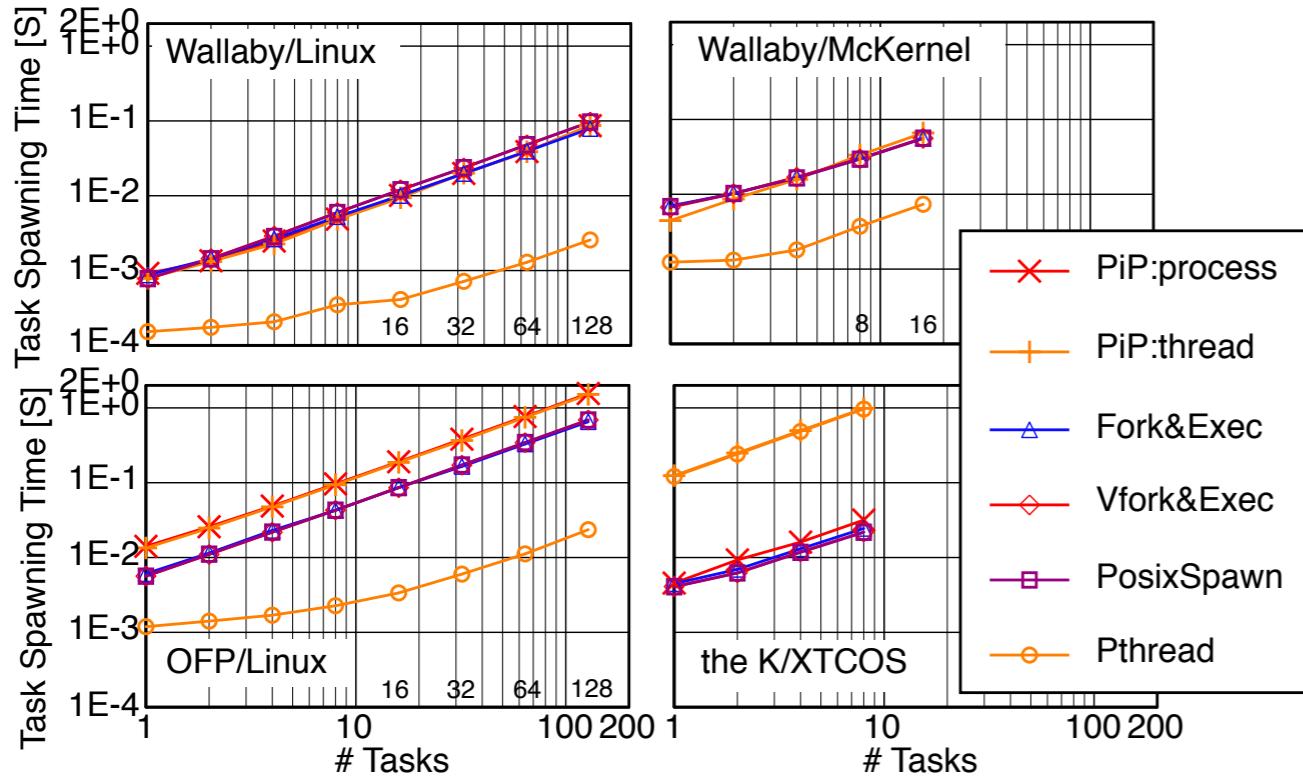
Vsyscall: The virtual system call page at address `ffffffffff600000-ffffffffffff601000`.

Address Range	Permissions	Name
55555554000-555555556000	r-xp	/PIP/test/basic
555555755000-555555756000	r--p	/PIP/test/basic
555555756000-555555757000	rw-p	/PIP/test/basic
555555757000-555555778000	rw-p	[heap]
7ffe8000000-7ffe8021000	rw-p	
7ffe8021000-7fffec000000	---p	
7fff0000000-7fff0021000	rw-p	
7fff0021000-7ffff4000000	---p	
7ffff4b24000-7ffff4c24000	rw-p	
7ffff4c24000-7ffff4c27000	r-xp	/PIP/lib/libpip.so
7ffff4c27000-7ffff4e26000	---p	/PIP/lib/libpip.so
7ffff4e26000-7ffff4e27000	r--p	/PIP/lib/libpip.so
7ffff4e27000-7ffff4e28000	rw-p	/PIP/lib/libpip.so
7ffff4e28000-7ffff4e2a000	r-xp	/PIP/test/basic
7ffff4e2a000-7ffff5029000	---p	/PIP/test/basic
7ffff5029000-7ffff502a000	r--p	/PIP/test/basic
7ffff502a000-7ffff502b000	rw-p	/PIP/test/basic
7ffff502b000-7ffff502e000	r-xp	/PIP/lib/libpip.so
7ffff502e000-7ffff522d000	---p	/PIP/lib/libpip.so
7ffff522d000-7ffff522e000	r--p	/PIP/lib/libpip.so
7ffff522e000-7ffff522f000	rw-p	/PIP/lib/libpip.so
7ffff522f000-7ffff5231000	r-xp	/PIP/test/basic
7ffff5231000-7ffff5430000	---p	/PIP/test/basic
7ffff5430000-7ffff5431000	r--p	/PIP/test/basic
7ffff5431000-7ffff5432000	rw-p	/PIP/test/basic
...		
7ffff5a52000-7ffff5a56000	rw-p	
...		
7ffff5c6e000-7ffff5c72000	rw-p	
7ffff5c72000-7ffff5e28000	r-xp	/lib64/libc.so
7ffff5e28000-7ffff6028000	---p	/lib64/libc.so
7ffff6028000-7ffff602c000	r--p	/lib64/libc.so
7ffff602c000-7ffff602e000	rw-p	/lib64/libc.so
7ffff602e000-7ffff6033000	rw-p	
7ffff6033000-7ffff61e9000	r-xp	/lib64/libc.so
7ffff61e9000-7ffff63e9000	---p	/lib64/libc.so
7ffff63e9000-7ffff63ed000	r--p	/lib64/libc.so
7ffff63ed000-7ffff63ef000	rw-p	/lib64/libc.so
7ffff63ef000-7ffff63f4000	rw-p	
7ffff63f4000-7ffff63f5000	---p	
7ffff63f5000-7ffff6bf5000	rw-p	[stack:10641]
7ffff6bf5000-7ffff6bf6000	---p	
7ffff6bf6000-7ffff73f6000	rw-p	[stack:10640]
7ffff73f6000-7ffff75ac000	r-xp	/lib64/libc.so
7ffff75ac000-7ffff77ac000	---p	/lib64/libc.so
7ffff77ac000-7ffff77b0000	r--p	/lib64/libc.so
7ffff77b0000-7ffff77b2000	rw-p	/lib64/libc.so
7ffff77b2000-7ffff77b7000	rw-p	
...		
7ffff79cf000-7ffff79d3000	rw-p	
7ffff79d3000-7ffff79d6000	r-xp	/PIP/lib/libpip.so
7ffff79d6000-7ffff7bd5000	---p	/PIP/lib/libpip.so
7ffff7bd5000-7ffff7bd6000	r--p	/PIP/lib/libpip.so
7ffff7bd6000-7ffff7bd7000	rw-p	/PIP/lib/libpip.so
7ffff7dd000-7ffff7dfc000	r-xp	/lib64/ld.so
7ffff7edc000-7ffff7fe0000	rw-p	
7ffff7ff7000-7ffff7ffa000	rw-p	
7ffff7ffa000-7ffff7ffc000	r-xp	[vdso]
7ffff7ffc000-7ffff7ffd000	r--p	/lib64/ld.so
7ffff7ffd000-7ffff7ffe000	rw-p	/lib64/ld.so
7ffff7ffe000-7ffff7fff000	rw-p	
7ffff7fffde000-7fffffff000	rw-p	[stack]
ffffffffff600000-ffffffffffff601000	r-xp	[vsyscall]

PiP Basic Performance

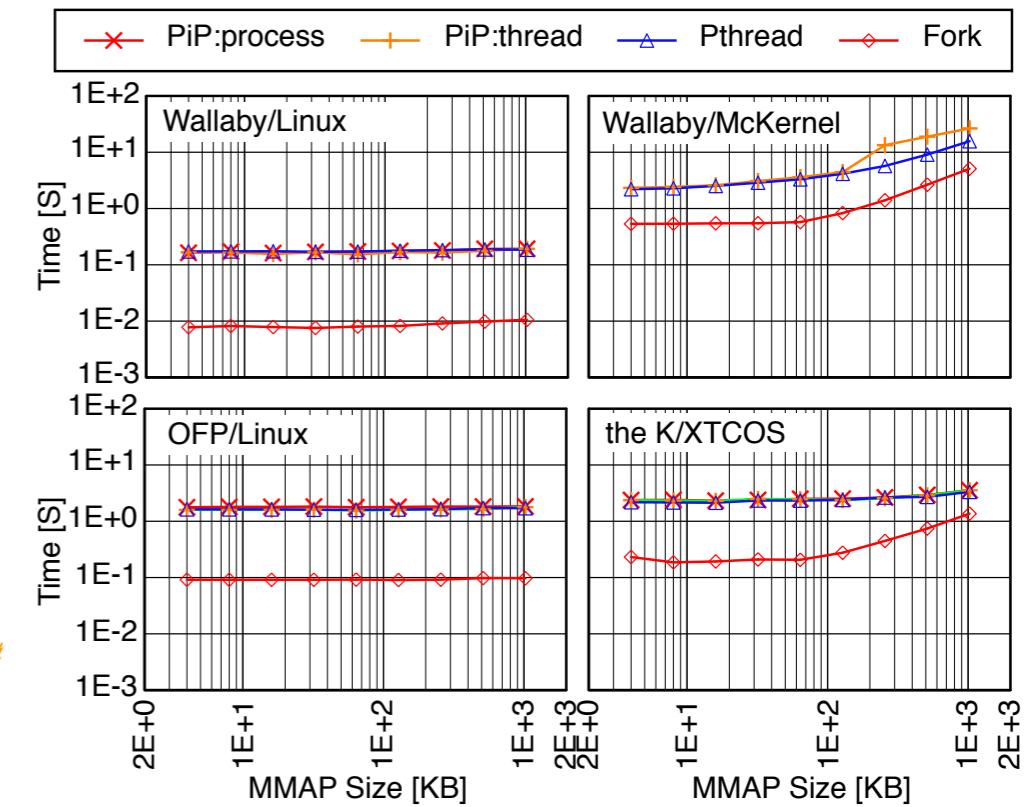
- A larger number of memory segments MAY affect task spawning time and/or mmap() time
- **Spawn:** Almost equal to processes
- **mmap():** Almost equal to pthreads

Task Spawning Time



Better

mmap() Time (10 tasks)



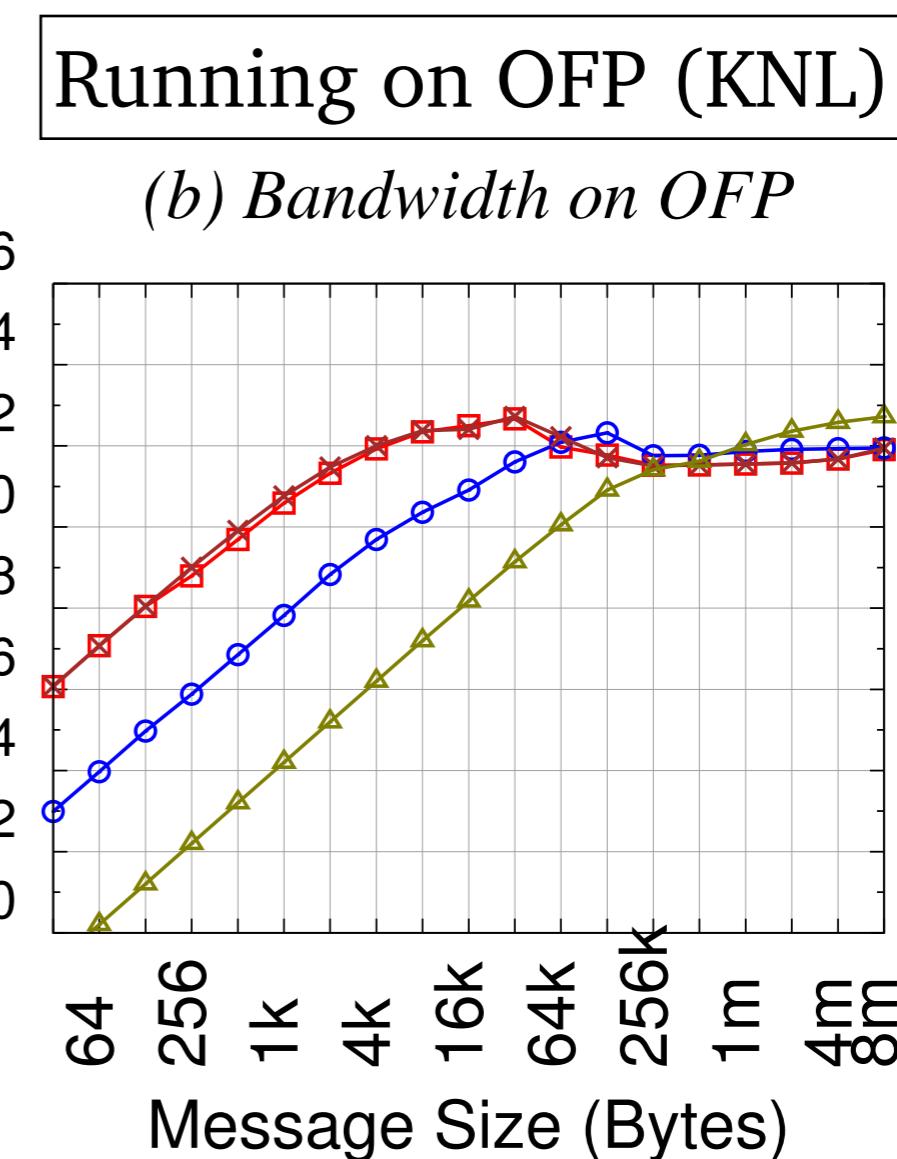
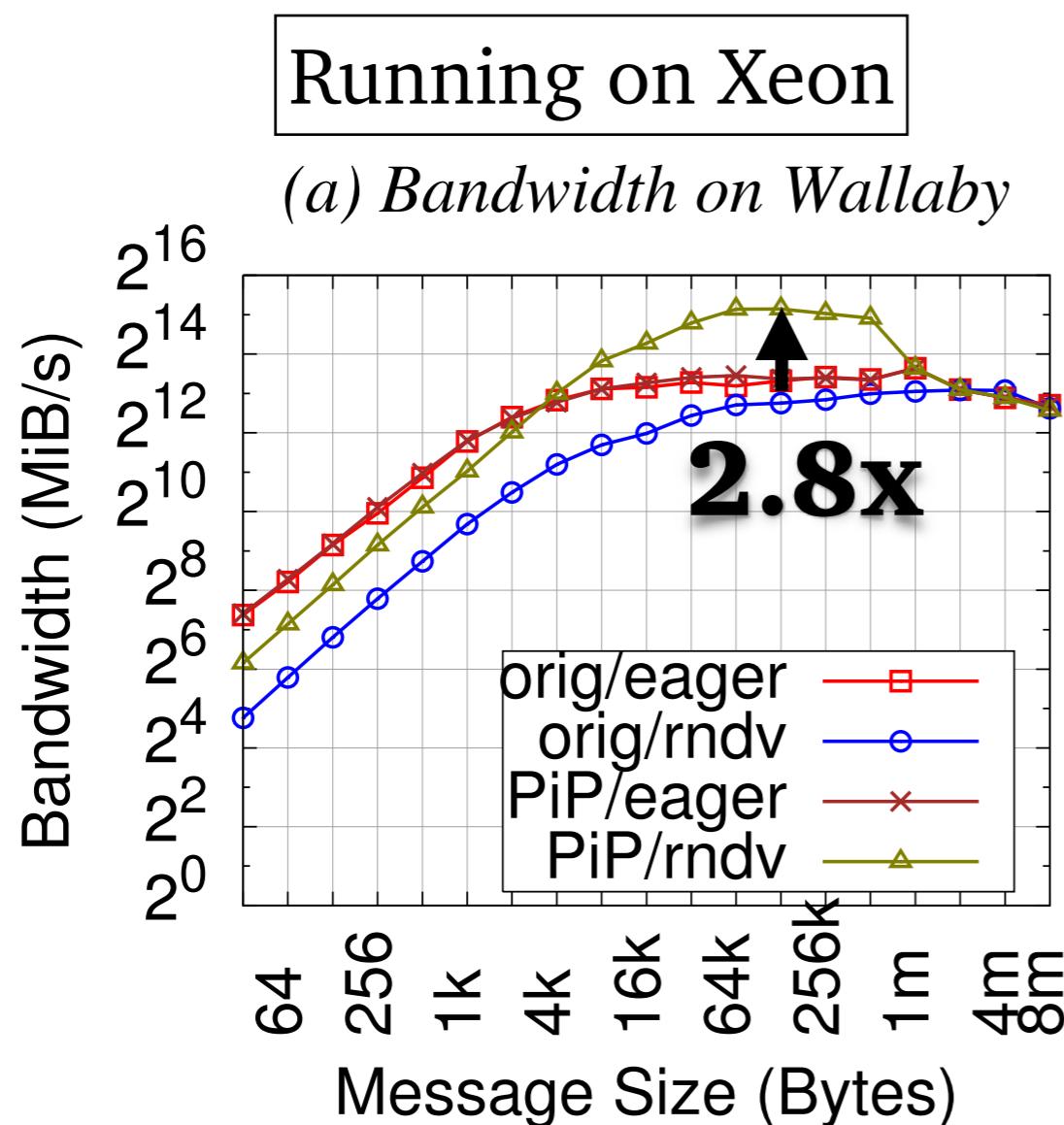
PiP-aware MPI Prototype

- PiP Root: Hydra process manager
- PiP Tasks: MPI Processes
- PiP Mode: Process Mode
- Base MPI: MPICH (v3.3a3)
- PiP-aware MPI Optimizations
 - Pt2Pt rendezvous protocol
 - Before: 2 copies
 - After: 1 copy
 - MPI_Win_allocate_shared()
 - Before: Allocate POSIX shmem and broadcast
 - After: malloc() at root and broadcast

pmi-proxy
MPI Proc-0
MPI Proc-1
MPI Proc-2
MPI Proc-3
:

MPI Pt2Pt (Intra-Node) Performance

- PiP rendezvous protocol outperform 2.8x
 - IMB Ping-Pong benchmark (Intra-Node)

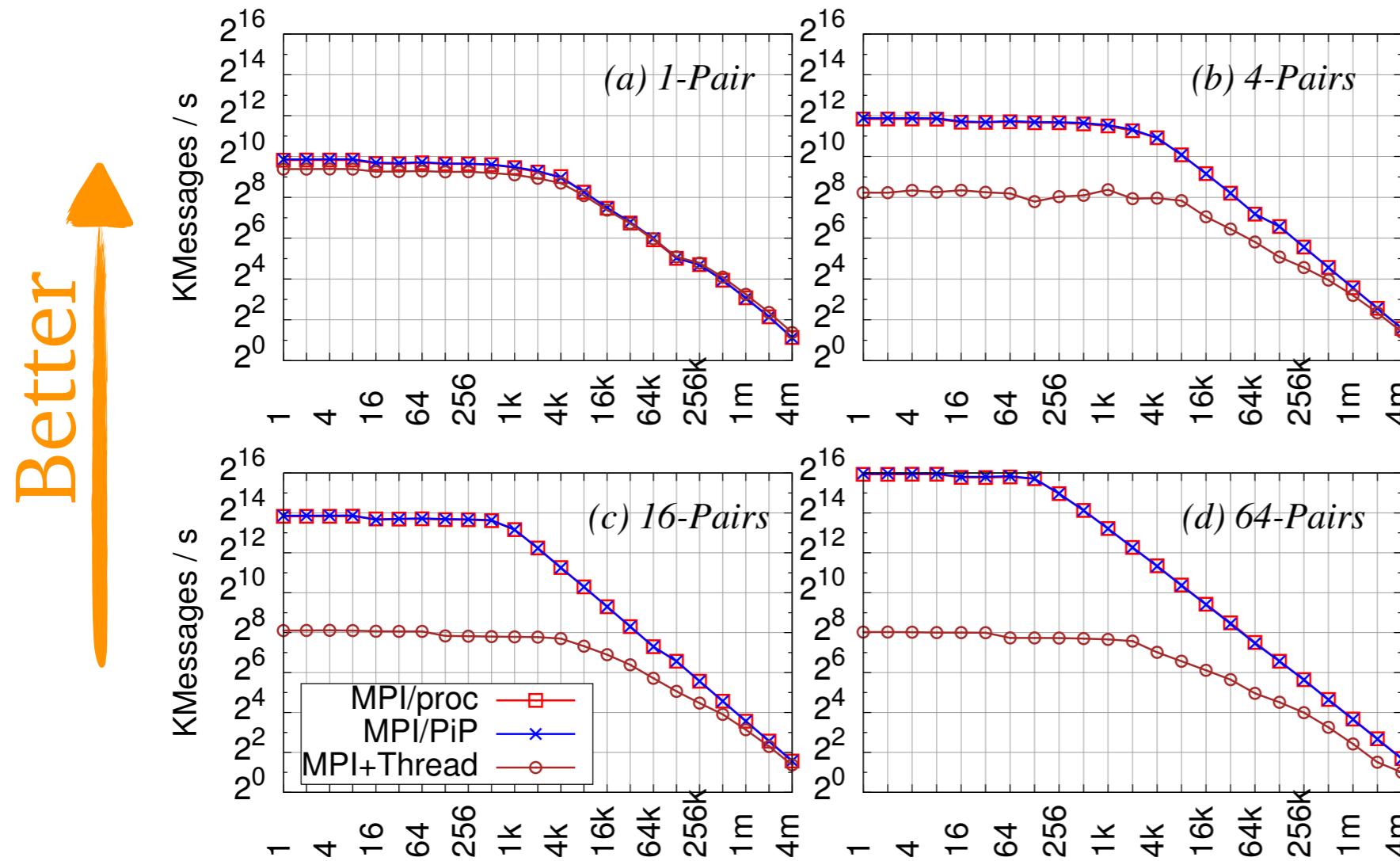


Better

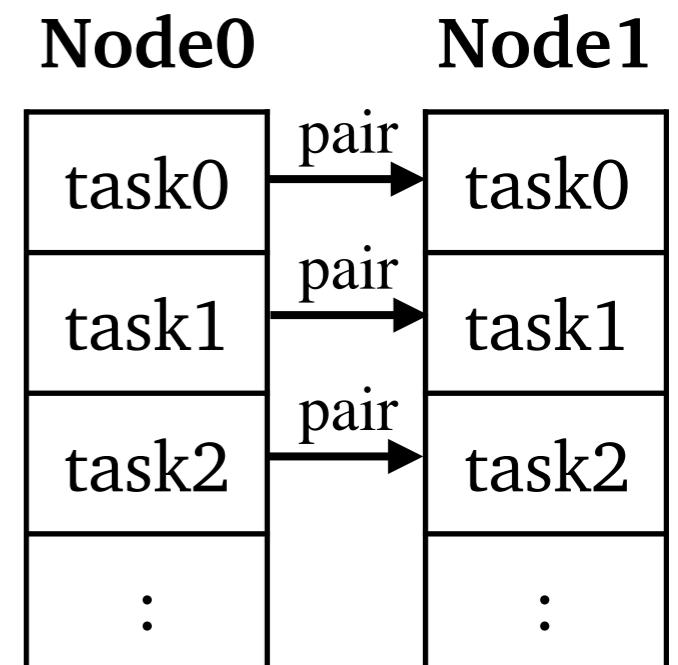
Note: The results of orig/eager and PiP/eager are overlapped.

MPI Multithreaded (Inter-Node) Performance

- Modified `osu_mbw_mr` benchmark program
 - to compare MPI+Thread and PiP-aware MPI



Note: The results of MPI/proc and MPI/PiP are overlapped in the four graphs.

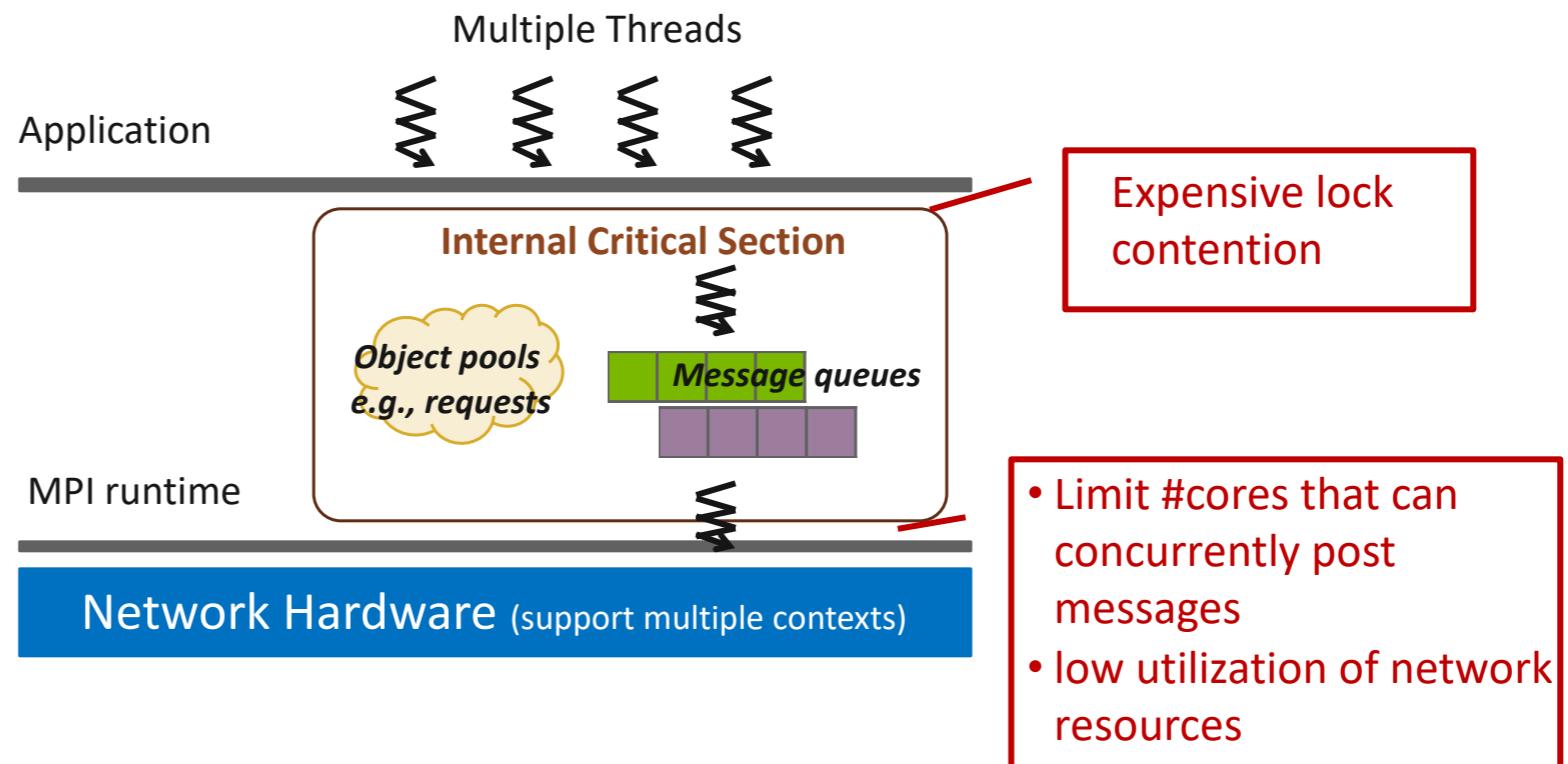


Running on
OFP (KNL)

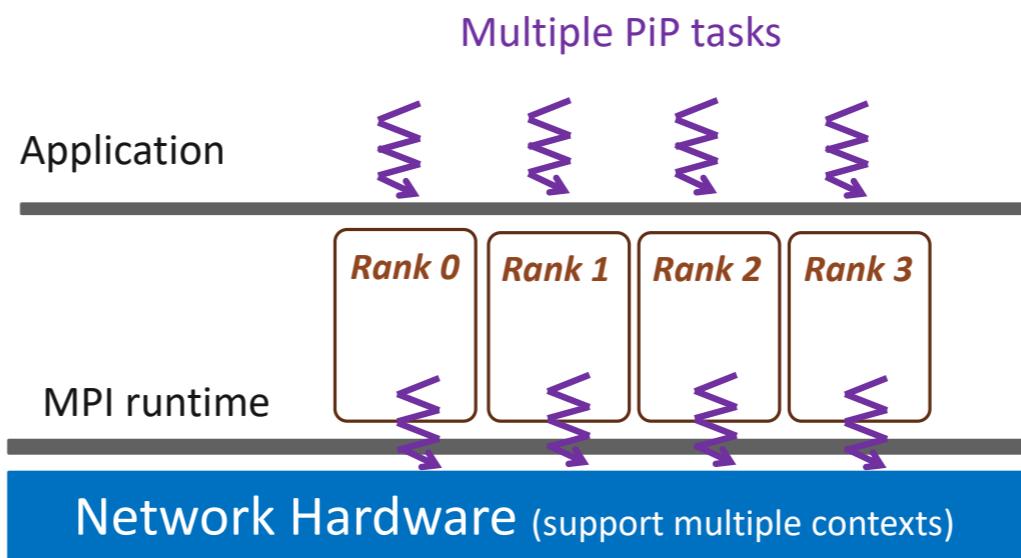
Figure 11: Multipair message rate between two OFP nodes

Why PiP is better ?

- MPI standard does not allow threaded comm.

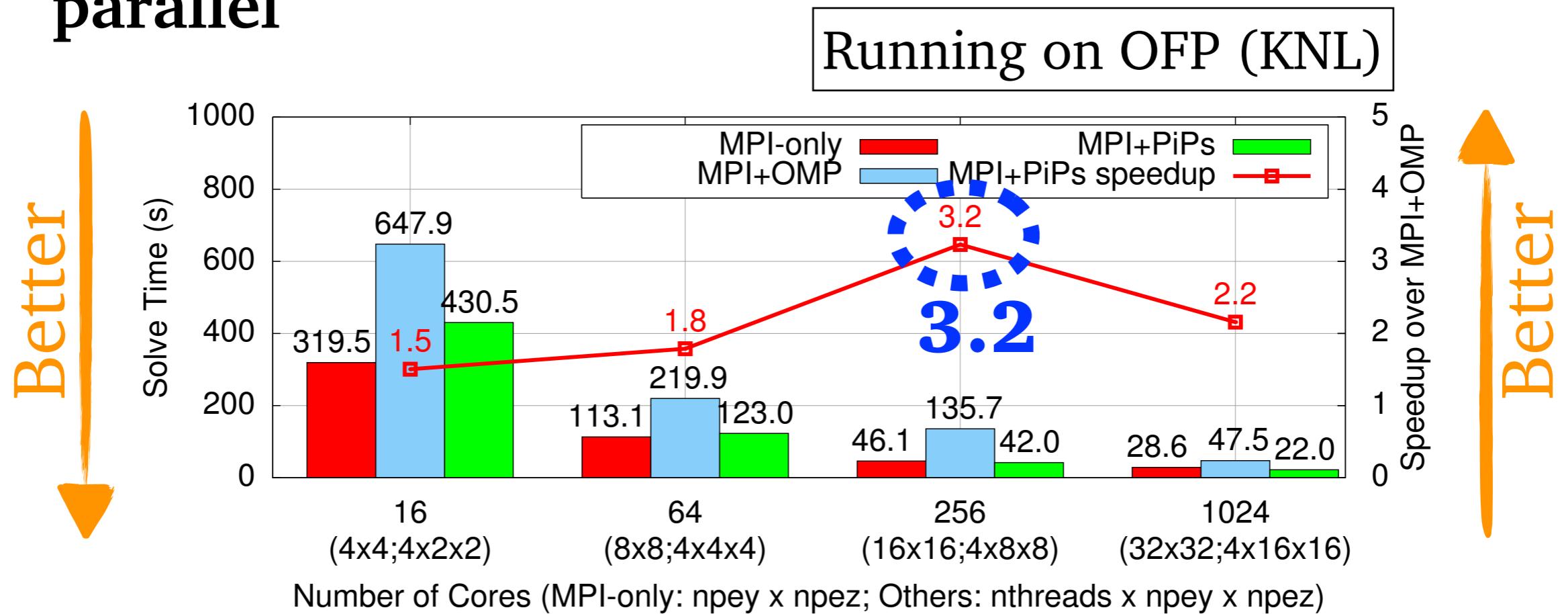


- Each PiP task has its own MPI lib. and they can run in parallel

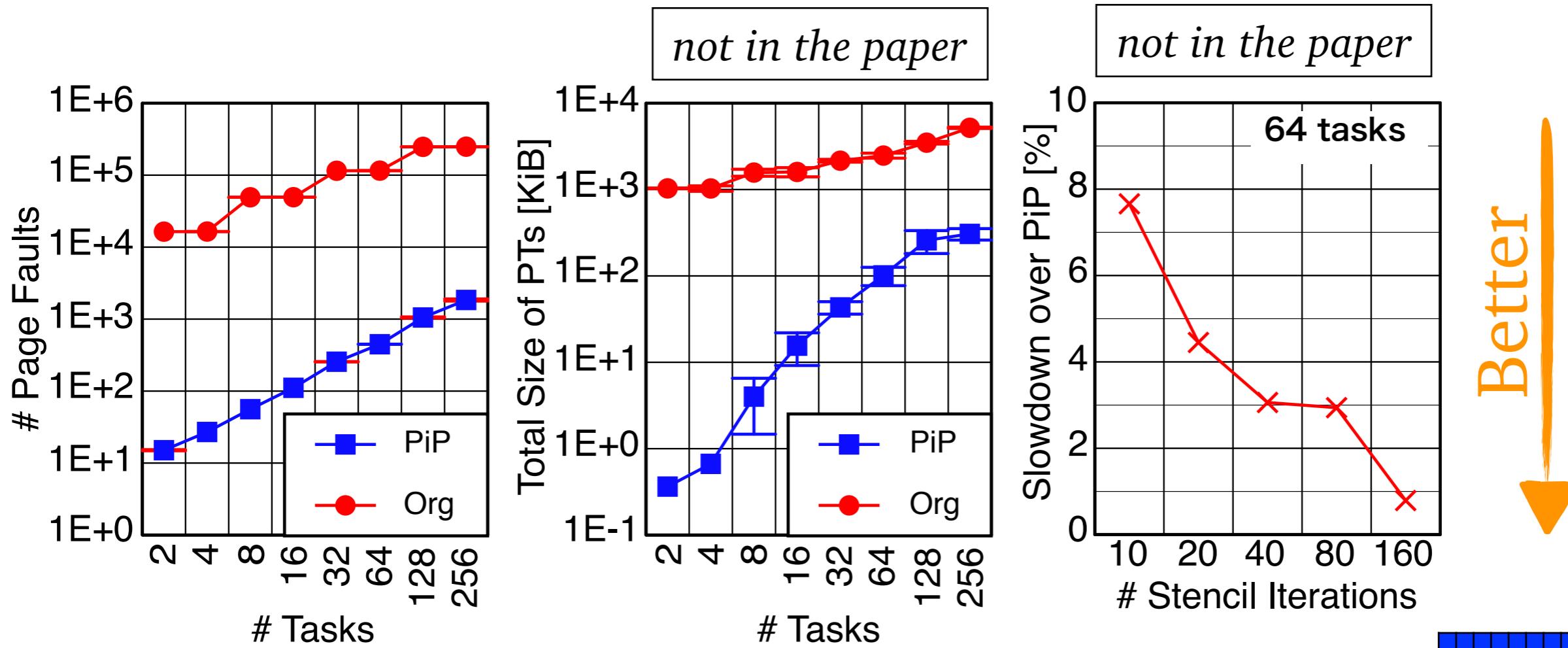


SNAP

- Before: MPI + OpenMP
 - SNAP call MPI functions from each OpenMP thread
 - Only one communication takes place at a time
- After: MPI + PiP
 - Each PiP task can perform communication in parallel

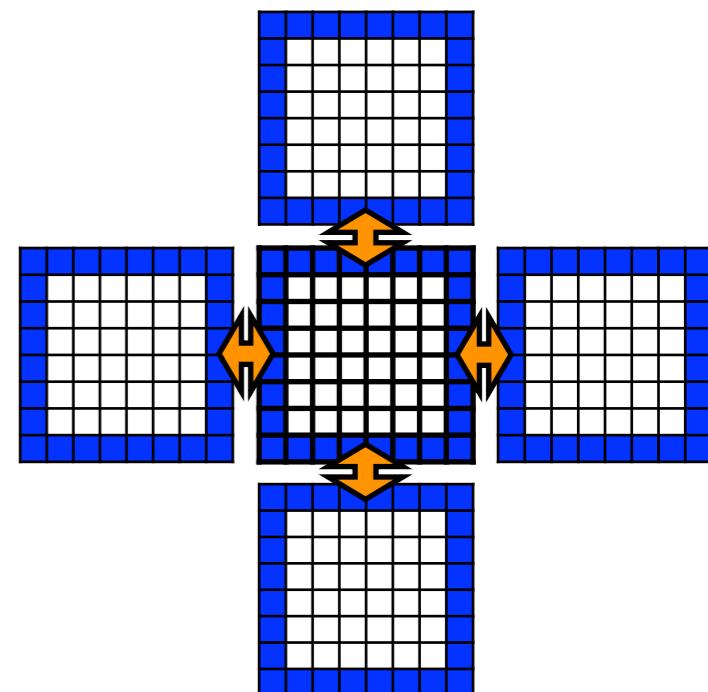


MPI_Win_allocate_shared()



- Org: **POSIX Shmem (Memory Mapping)**
- PiP: **VAS Sharing**

5P Stencil (double, 8K x 8K), Xeon Phi (KNL)
Example program `stencil_mpi_shmem.c` in Advanced MPI Programming -
Tutorial at SC14. <http://www.mcs.anl.gov/~thakur/sc14-mpi-tutorial>



Memory Mapping vs. PiP

- Memory Mapping Technique (POSIX Shmem, XPMEM, ...)
 - OH of sub-PT creation (create, attach, etc.)
 - OH of maintaining coherency (page fault)
 - Each process must have PT entries - $O(N^2)$
- * Those OH can be avoided by using PiP
Because Page table is shared

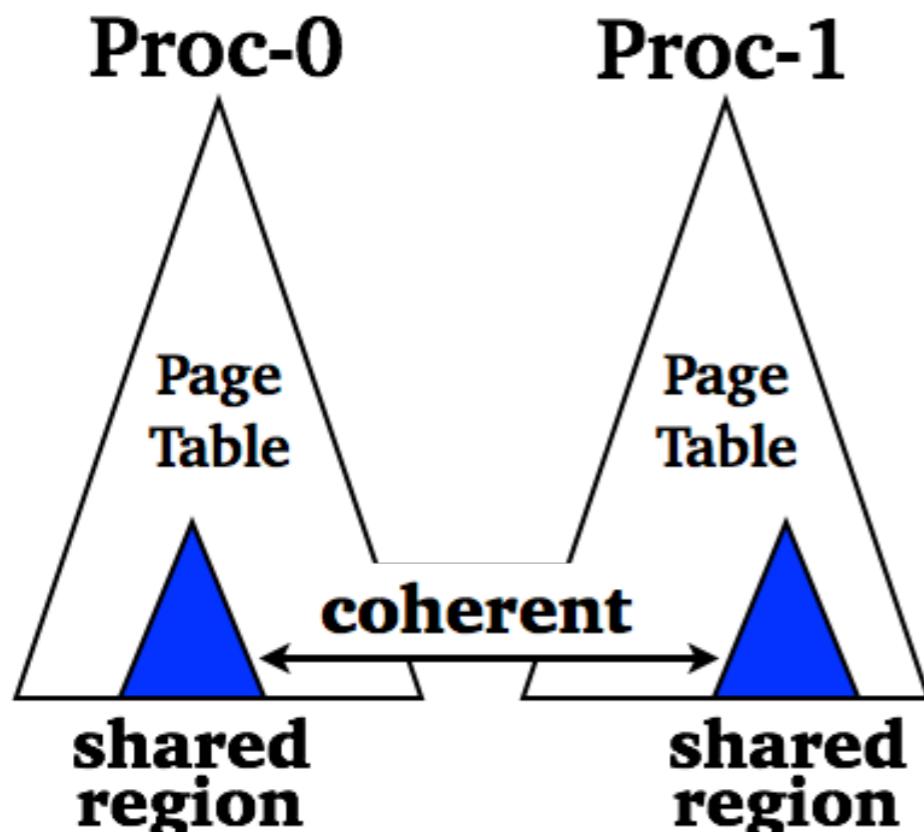


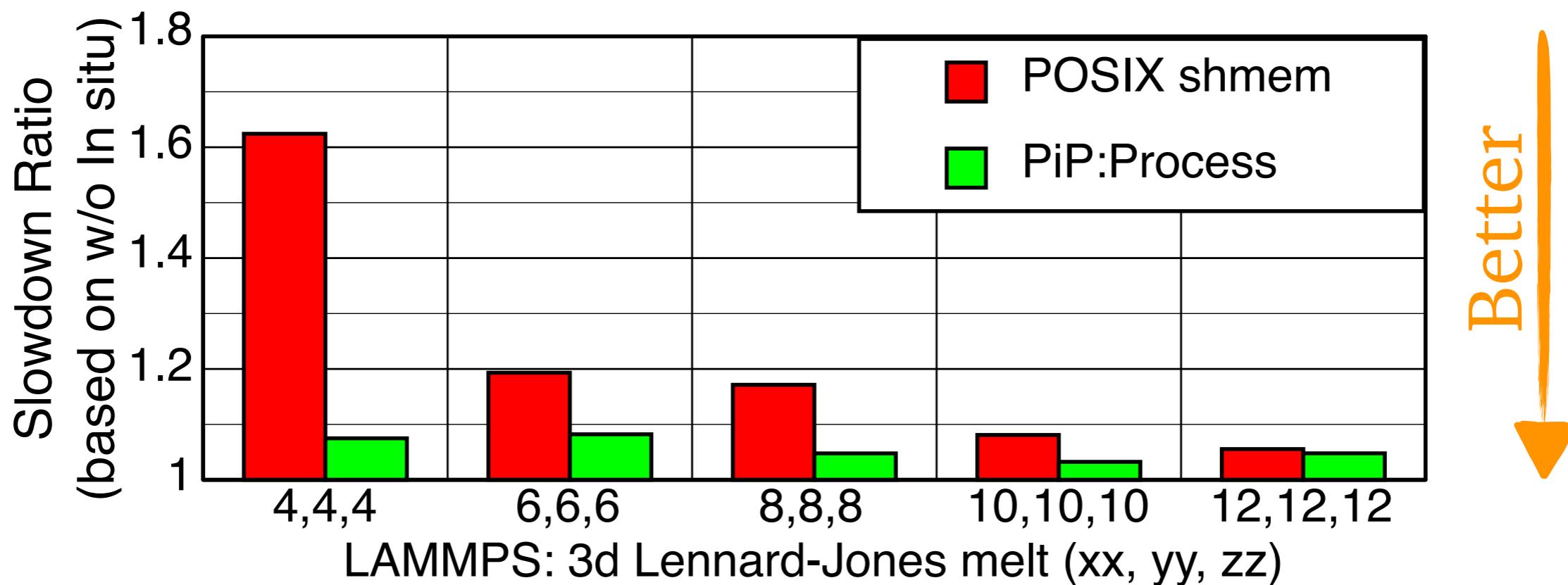
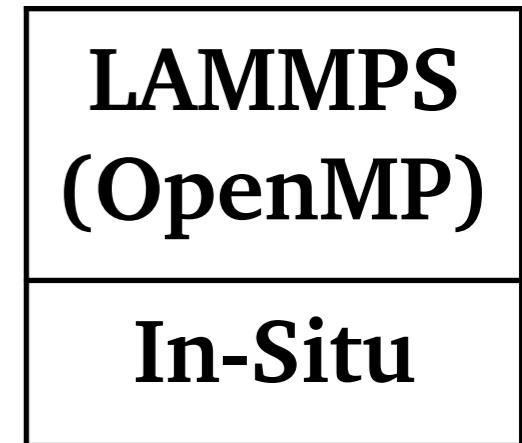
Table 5. Overhead of XPMEM and POSIX shmem functions (Xeon/Linux)

XPMEM	Cycles
xpmem_make()	1,585
xpmem_get()	15,294
xpmem_attach()	2,414
xpmem_detach()	19,183
xpmem_release()	693

	POSIX Shmem	Cycles
Sender	shm_open()	22,294
	ftruncate()	4,080
	mmap()	5,553
	close()	6,017
Receiver	shm_open()	13,522
	mmap()	16,232
	close()	16,746

LAMMPS with In-Situ

- Before: LAMMPS + In-Situ
 - Using POSIX shmem -> 2 Copies
- After: LAMMPS + In-Situ (as a PiP task)
 - LAMMPS passes a pointer to In-Situ and In-Situ copies simulation data -> 1 Copy





Summary

- PiP provides the 3rd execution model at user-level
 - Portable (on most Linux compatible OS)
 - `dlopen()`, PIE, and `clone()` or `pthread_create()`
 - Practical
 - Runs on big machines (K and OFP) in operation
- PiP outperforms Memory Mapping Techniques
 - Setup cost
 - Page fault overhead
 - Page table size
- PiP Showcases
 - PiP-aware MPI — `pt2pt`, `MPI_Win_allocate_shared()`
 - MPI+PiP outperforms MPI+OpenMP
 - More efficient In-Situ implementation

FAQs

✓ It is crazy to mix programs, I cannot debug

- Can't you debug multi-thread programs ?
- Do not mix independent programs.
Mix communicating and/or interacting programs.

✓ Is PiP a Programming Model ?

- No, it is a execution model

✓ If I use 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 require extra programming
 - More memory consumption

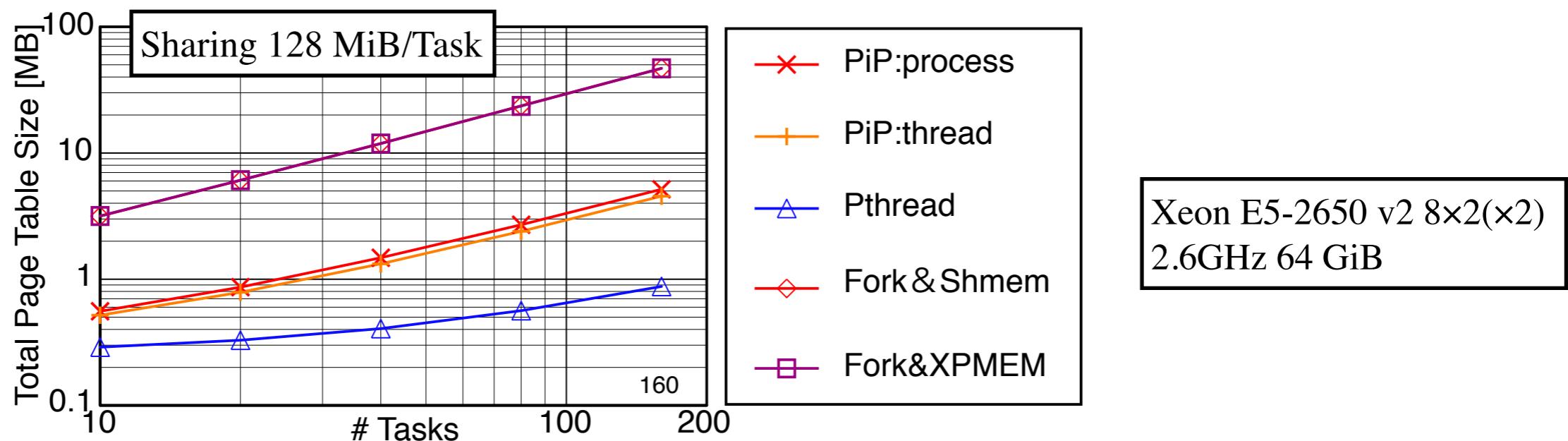
✓ We are already happy with MPI and OpenMP, no need of PiP

- PiP is for people who are not happy with them.

✓ PiP+OpenMP works ?

- In PiP Process Mode: YES. In Thread Mode, it depends.

Page Table Size



Note: The results of Fork&Shmem and Fork&XPMEM are overlapped.

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

Table 5: Total number of page table entries

Total Number of Page Table Entries	
Pthread	$M + D + \sum S_i$
PiP	$M + \sum D_i + \sum S_i$
Process + POSIX shmem	$(M \times N) + \sum D_i + \sum S_i$
Process + XPMEM	$(M \times N) + \sum D_i + \sum S_i$

M is the number of PT entries for the shared-memory region(s).

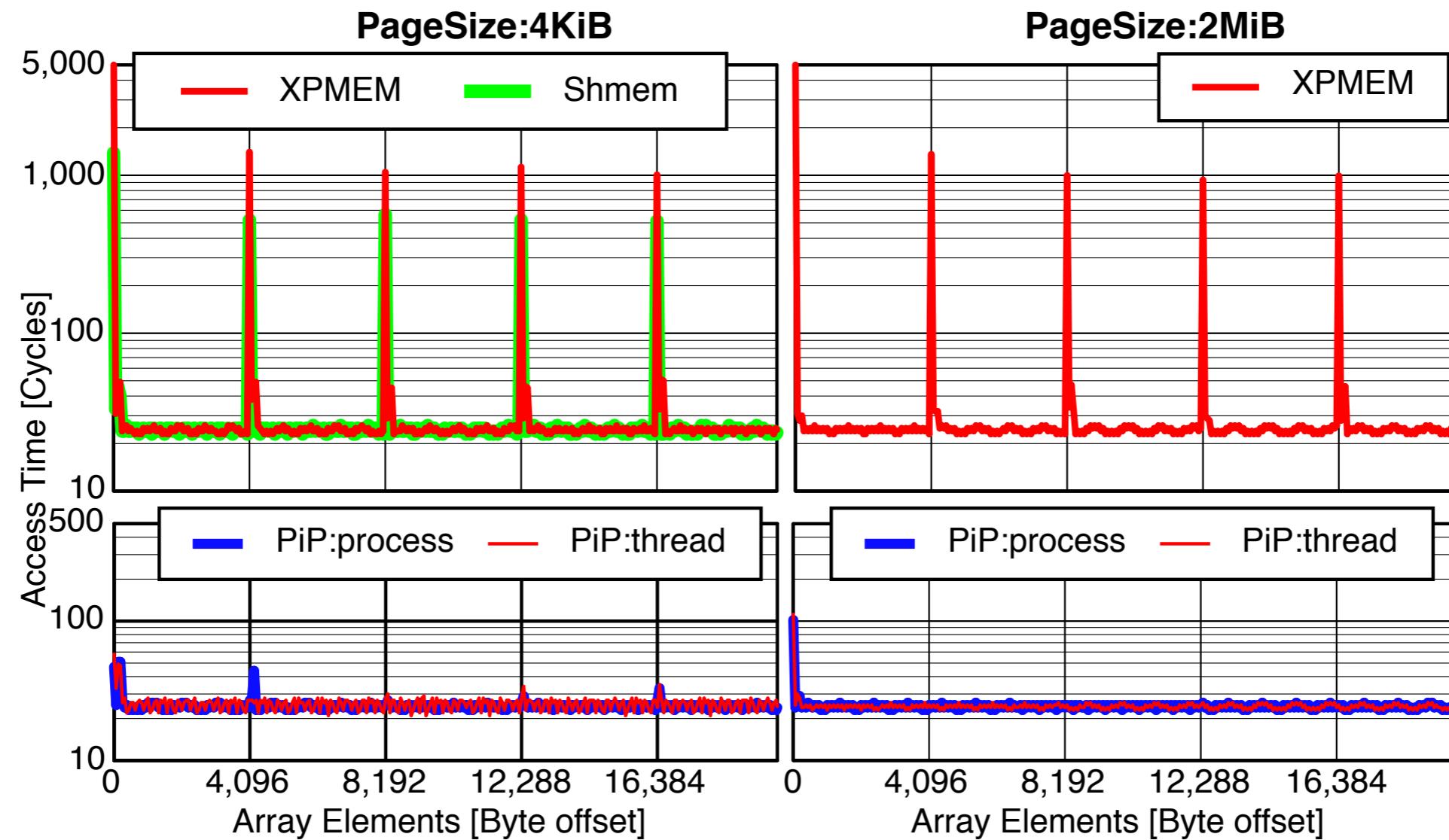
S_i is the number of PT entries for the stack segment of task i .

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 receiver



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

PIE and Static

