Optimizing Latency and Throughput for Spawning Processes on Massively Multicore Processors

Abhishek Kulkarni*, Michael Lang¶, Latchesar Ionkov¶ and Andrew Lumsdaine*

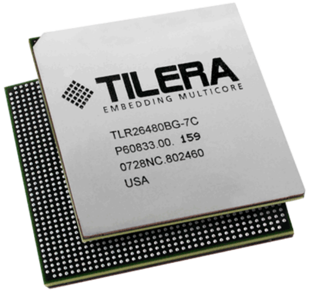
*Center for Research in Extreme Scale Technologies (CREST)
Indiana University, Bloomington

¶ Ultrascale Systems Research Center
New Mexico Consortium
Los Alamos National Laboratory

June 29 2012

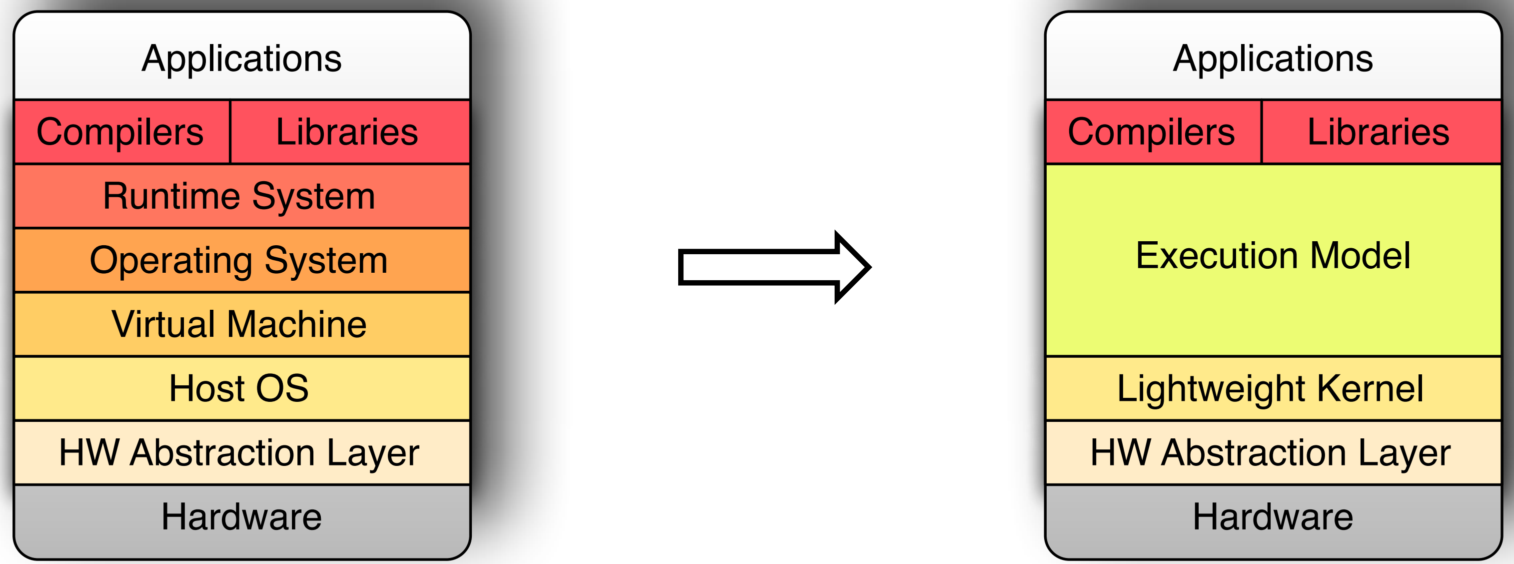
The Exascale Era

- "Going to the exascale" is a challenging venture
    - Clock rates and ILP have reached points of diminishing returns
    - Memory and Power wall limits performance
    - Higher soft and hard error rates at smaller feature sizes
    - Massive intra-node parallelism
- Manycore processors and integrated accelerators
    - Intel MIC (Knights Ferry, Knights Corner)
    - Tilera Tile Gx



We need transformation, not evolution

- Emerging hardware will drive the transformation of the traditional software stack
- New programming models, execution models, runtime systems and operating systems



Applications
Compilers | Libraries
Runtime System
Operating System
Virtual Machine
Host OS
HW Abstraction Layer
Hardware

Applications
Compilers | Libraries
Execution Model
Lightweight Kernel
HW Abstraction Layer
Hardware

Challenges

- New operating systems and lightweight kernels
  - Tessellation, Barrelfish, Kitten, fos
- Native operating system (OS) support for **accelerators**
- Evolutionary approaches
  - Identifying potential **bottlenecks** in existing operating systems (Linux)
  - OS scalability efforts such as partitioning, many-core inter-processor communication, symbiotic execution, virtualization.

Optimizing process management

- Reducing the latency to spawn new processes **locally** results in faster **global** job launch
- Emerging dynamic and resilient execution models are considering the feasibility of maintaining **process pools** for fault isolation
- Higher throughput makes such runtime systems viable
- **Memory overcommiting** can lead to unpredictable behavior including excessive swapping or OOM killing random processes
- Optimizing memory locality and **NUMA-aware** process spawning

“Process control in its modern form was designed and implemented within a couple of days. It is astonishing how easily it fitted into the existing system; at the same time it is easy to see how some of the slightly unusual features of the design are present precisely because they represented small, easily-coded changes to what existed.”

Dennis M. Ritchie
The Evolution of the Unix Time-sharing System, 1979

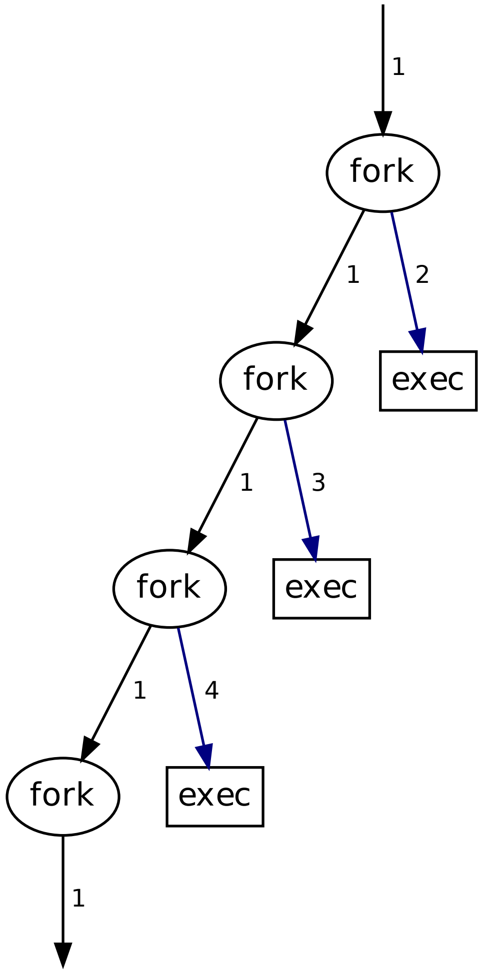
fork and exec in early Unix

- Separate fork-exec borrowed from the Berkeley time-sharing system (1965).
- The initial implementation of fork only required:
  - Expansion of the process table
  - Addition of a fork call that copied the current process to the disk swap area, using the already existing swap IO primitives, and made some adjustments to the process table.
- In fact, the PDP-7's fork call required precisely 27 lines of assembly code.
- **exec** as such did not exist; its function was already performed, using explicit IO, by the shell.

Source: The Evolution of the Unix Time-sharing System

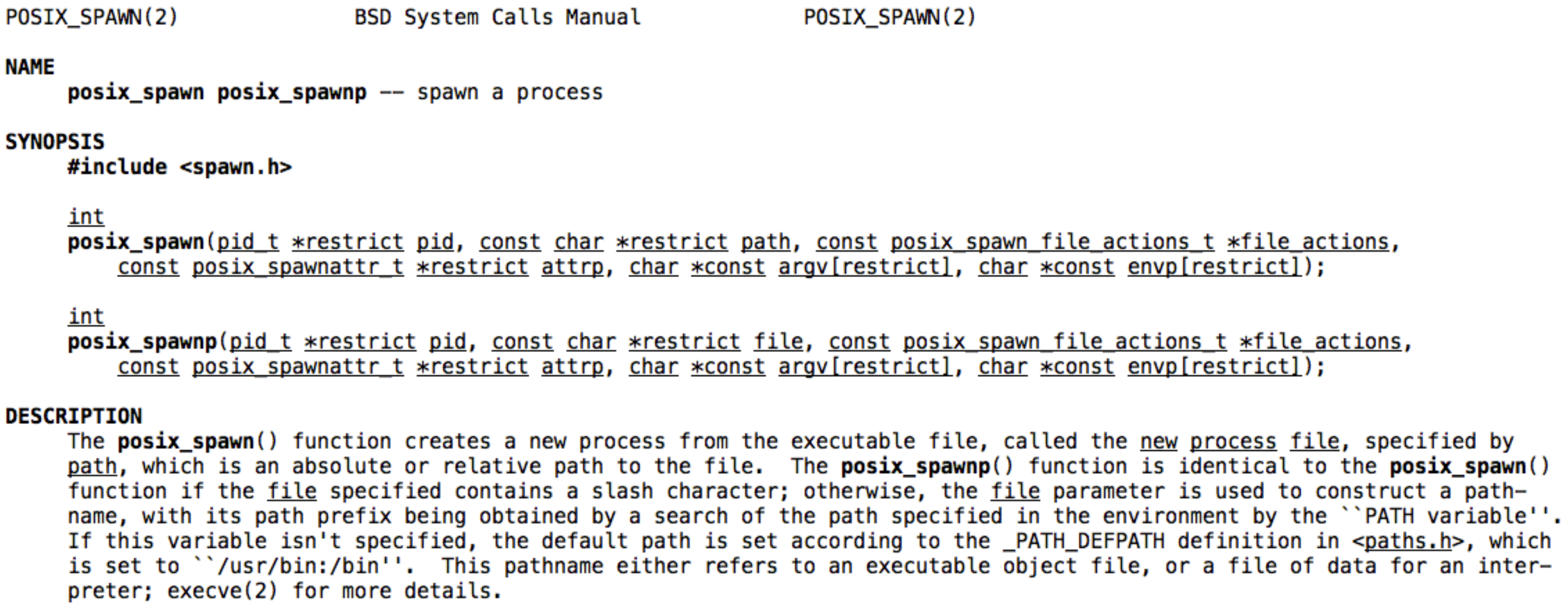
Spawning a group of processes

```
for i in n:
        pid = fork()
        if  pid == 0:                        // child
                exec(cmd)
        else if pid:                         // parent
                sched_setaffinity(pid)
                continue
```



What's wrong with the traditional approach?

- Serial execution
- ~3-4 system calls / context-switches
- Redundant operations shared between fork and exec
  - fork copies page tables, sets up a TCB
  - exec wipes off the control block and reinitializes the address space
  - Solution: vfork
- Relies on memory overcommit for process spawning
  - Solution: posix_spawn

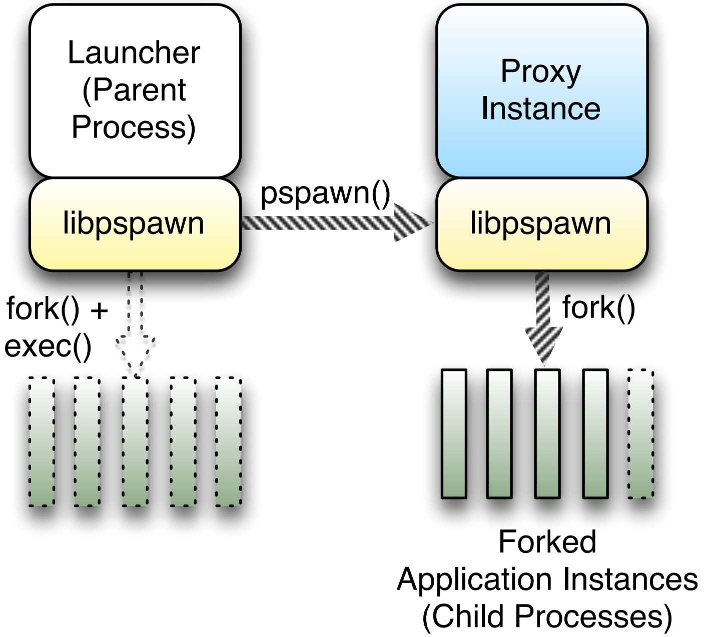


```
POSIX_SPAWN(2)              BSD System Calls Manual             POSIX_SPAWN(2)

NAME
     posix_spawn posix_spawnp -- spawn a process

SYNOPSIS
     #include <spawn.h>

     int
     posix_spawn(pid_t *restrict pid, const char *restrict path, const posix_spawn_file_actions_t *file_actions,
         const posix_spawnattr_t *restrict attrp, char *const argv[restrict], char *const envp[restrict]);

     int
     posix_spawnp(pid_t *restrict pid, const char *restrict file, const posix_spawn_file_actions_t *file_actions,
         const posix_spawnattr_t *restrict attrp, char *const argv[restrict], char *const envp[restrict]);

DESCRIPTION
     The posix_spawn() function creates a new process from the executable file, called the new process file, specified by
     path, which is an absolute or relative path to the file.  The posix_spawnp() function is identical to the posix_spawn()
     function if the file specified contains a slash character; otherwise, the file parameter is used to construct a path-
     name, with its path prefix being obtained by a search of the path specified in the environment by the ``PATH variable''.
     If this variable isn't specified, the default path is set according to the _PATH_DEFPATH definition in <paths.h>, which
     is set to ``/usr/bin:/bin''.  This pathname either refers to an executable object file, or a file of data for an inter-
     preter; execve(2) for more details.
```

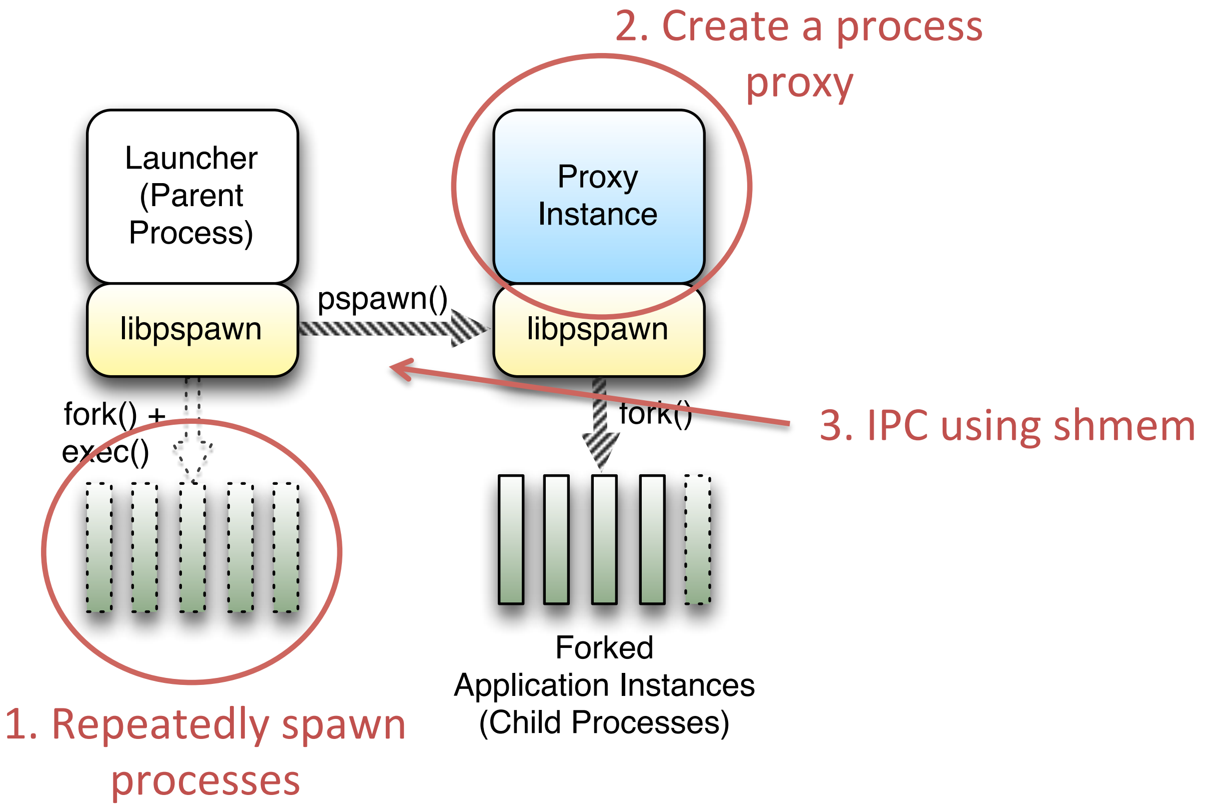
- **posix_spawn** typically implemented as a combination of fork and exec!

libpspawn: a userspace library for high throughput process spawning

- Transform the fork-exec-fork-exec pattern to fork-exec-fork-fork pattern
- Effectively, each fork-exec is replaced by a single fork
- Hijack application to act as a process spawn proxy
- Intercept process spawn system calls (fork exec) and relay them to the proxy



Spawning group of processes using libpspawn



pspawn Interface

Create a new pspawn context

int pspawn_context_create(pspawn_context_t *ctx, char *path, int core)

Destroy a pspawn context

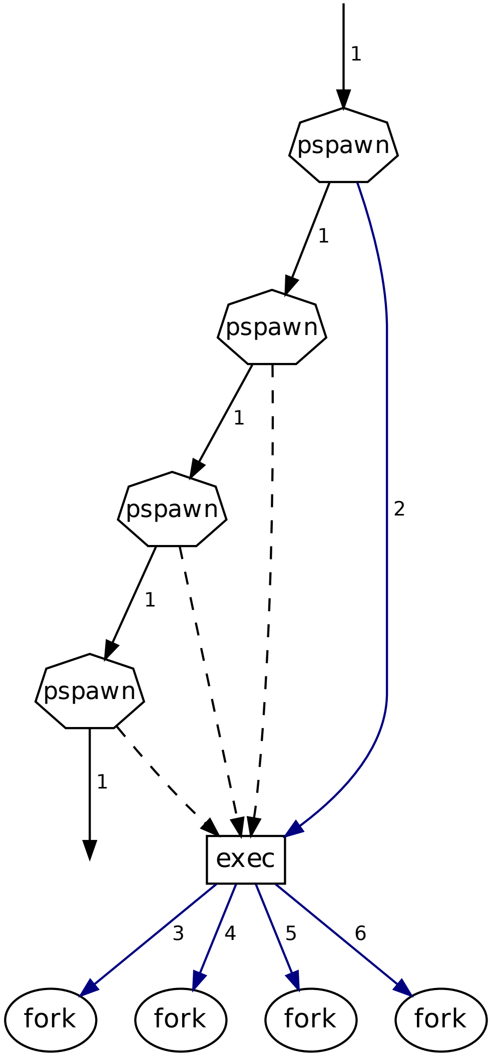
void pspawn_context_destroy(pspawn_context_t *ctx)

Launch a single process

pid_t pspawn (pspawn_context_t *ctx, char *argv[ ])

Launch ’n’ processes

int pspawn_n(pid_t *pids , pspawn_context_t *ctx, int nspawns, char *argv[ ])



Evaluating libpspawn

- **sfork** (Synchronous Fork)
  - Parent forks and waits for the child

- **afork** (Asynchronous Fork)
  - Parent waits after forking all the children

- **vfork** (VM Fork)
  - Parent forks child and shares virtual memory

- **pfork** (Parallel Fork)
  - Parent forks 1 child which forks the other (n-1) children

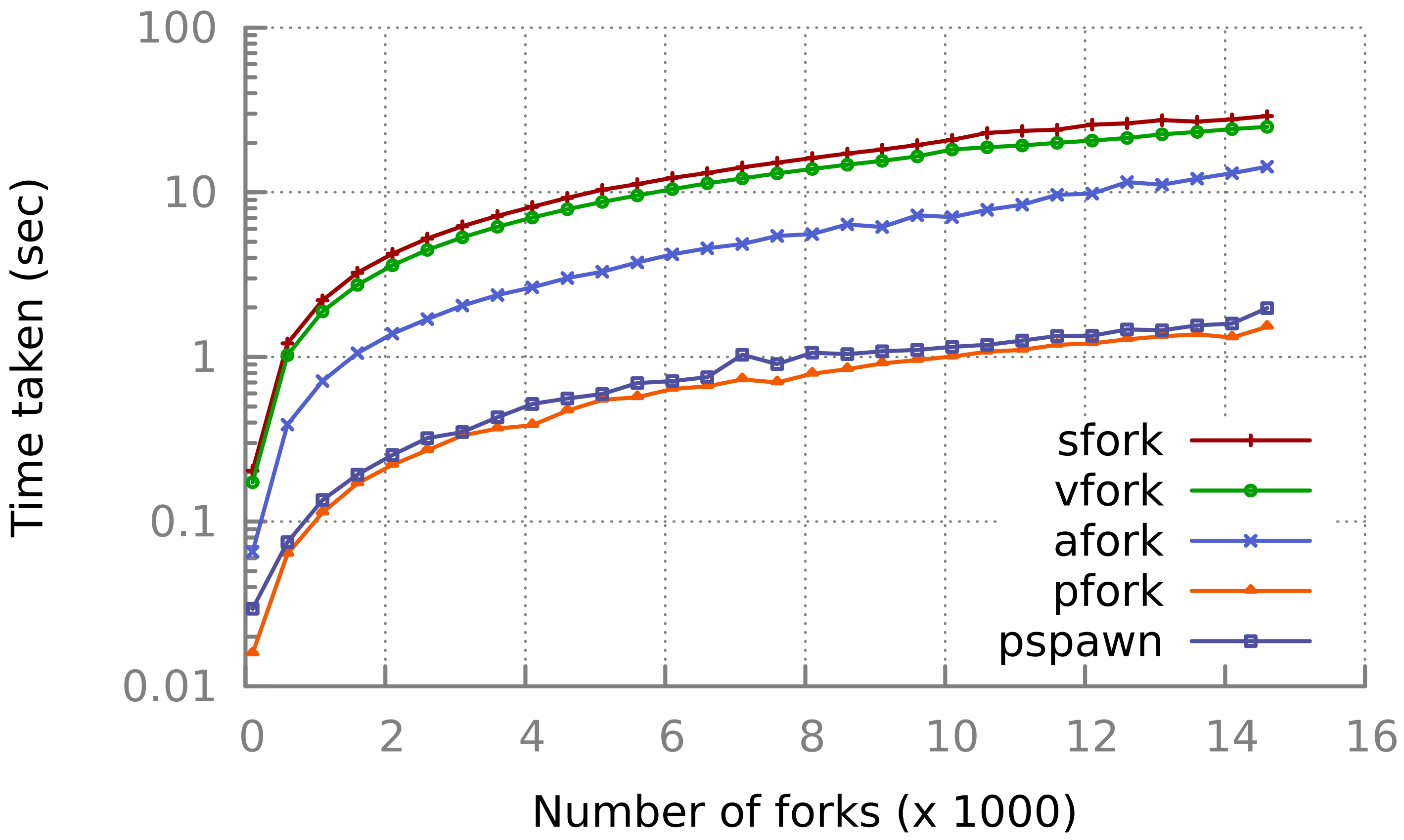
Experimental Setup

1. A quad-socket, quad-core (**16 cores** total, 4 NUMA domains) AMD Opteron node with 16 GB of memory and running Linux kernel 2.6.32-220.13.1.el6.x86_64.

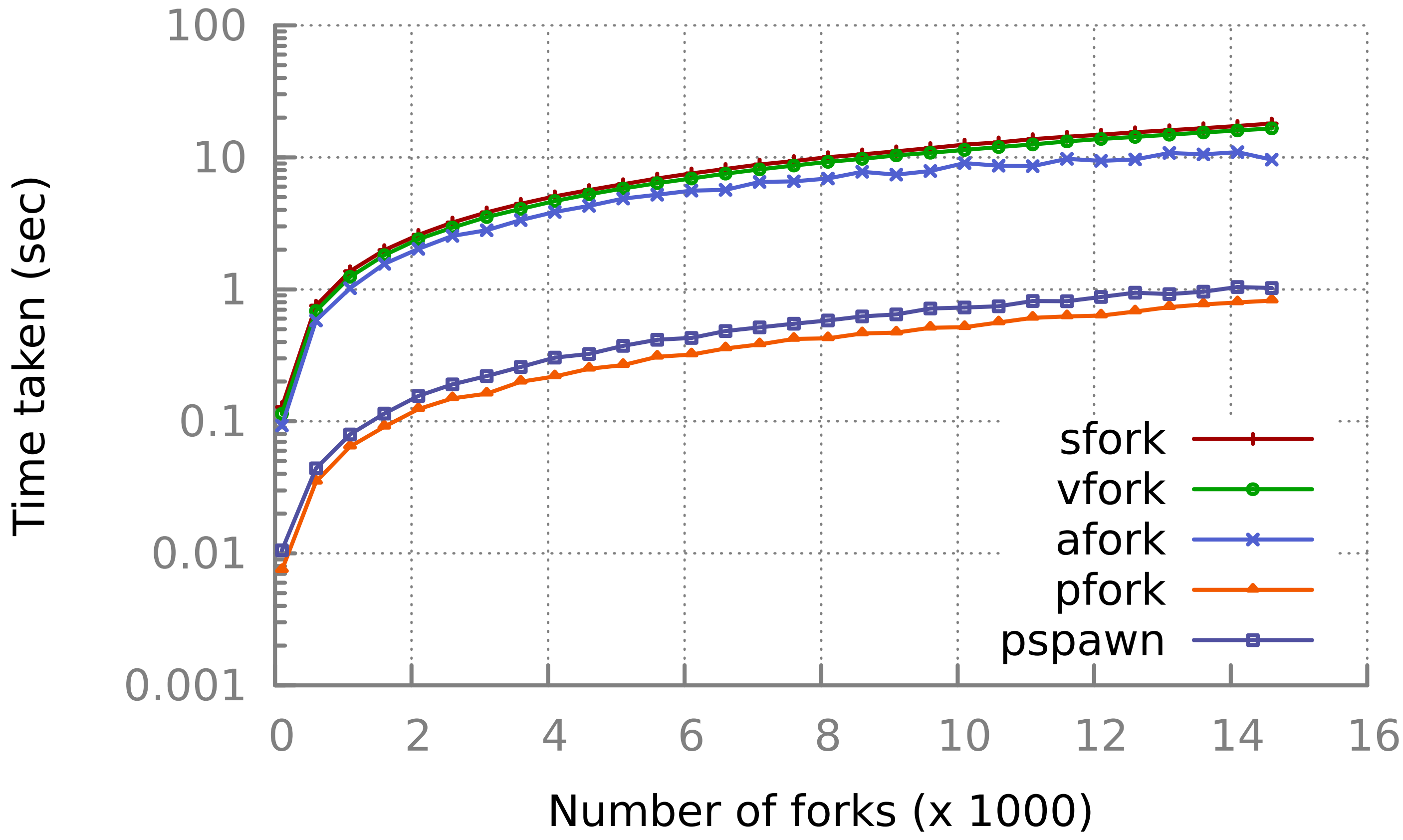
2. A two-socket, 12-core (**24 cores** total, 4 NUMA domains) AMD Istanbul node with 48 GB of memory and running Linux kernel 3.4.0-rc7.

3. An eight-socket, 8-core (**64 cores** total, 8 NUMA domains) Intel Nehalem node with 128 GB of memory and running Linux kernel 2.6.35.10-74.fc14.x86_64.

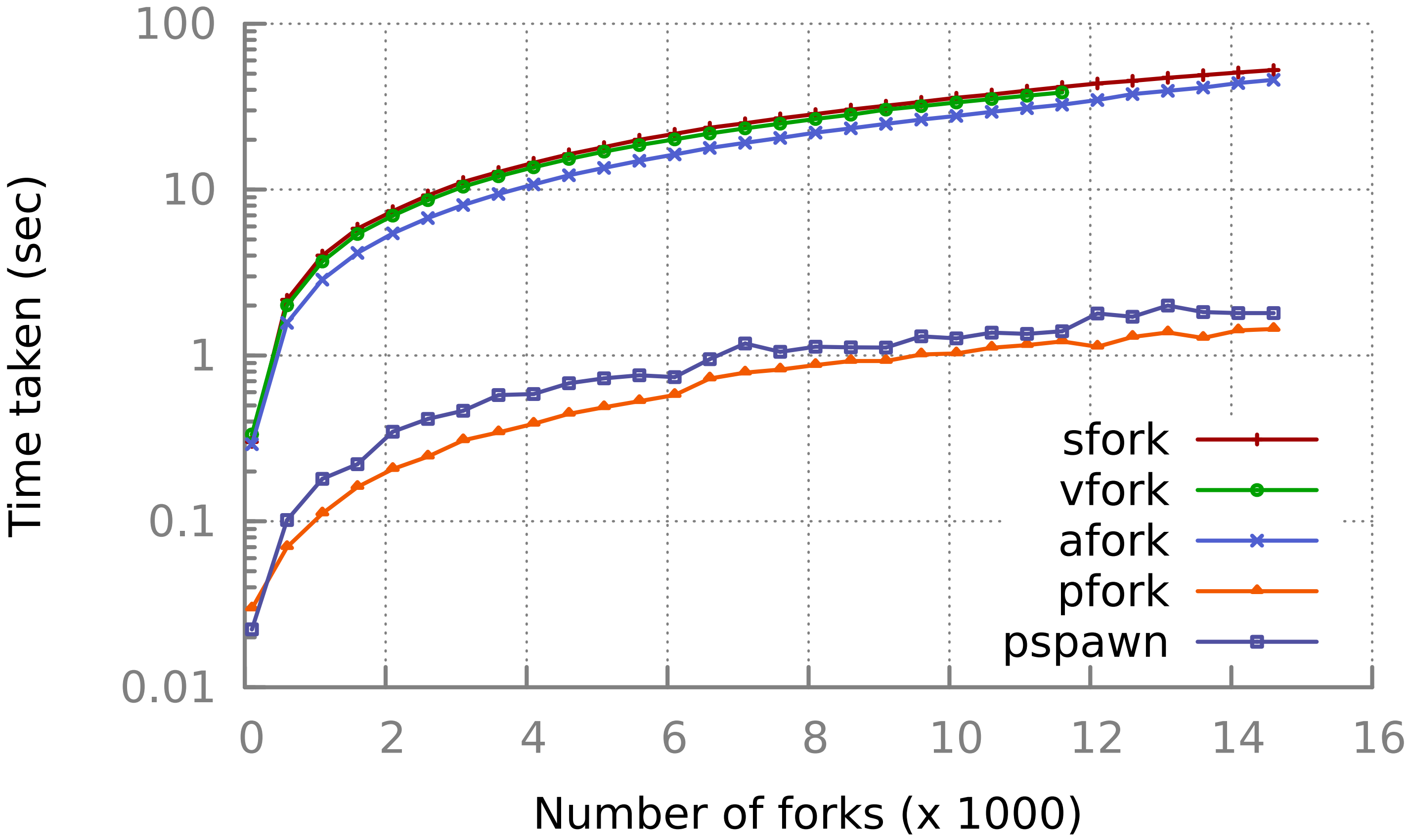
Comparing process spawn schemes: 16-core node



Comparing process spawn schemes: 24-core node



Comparing process spawn schemes: 64-core node

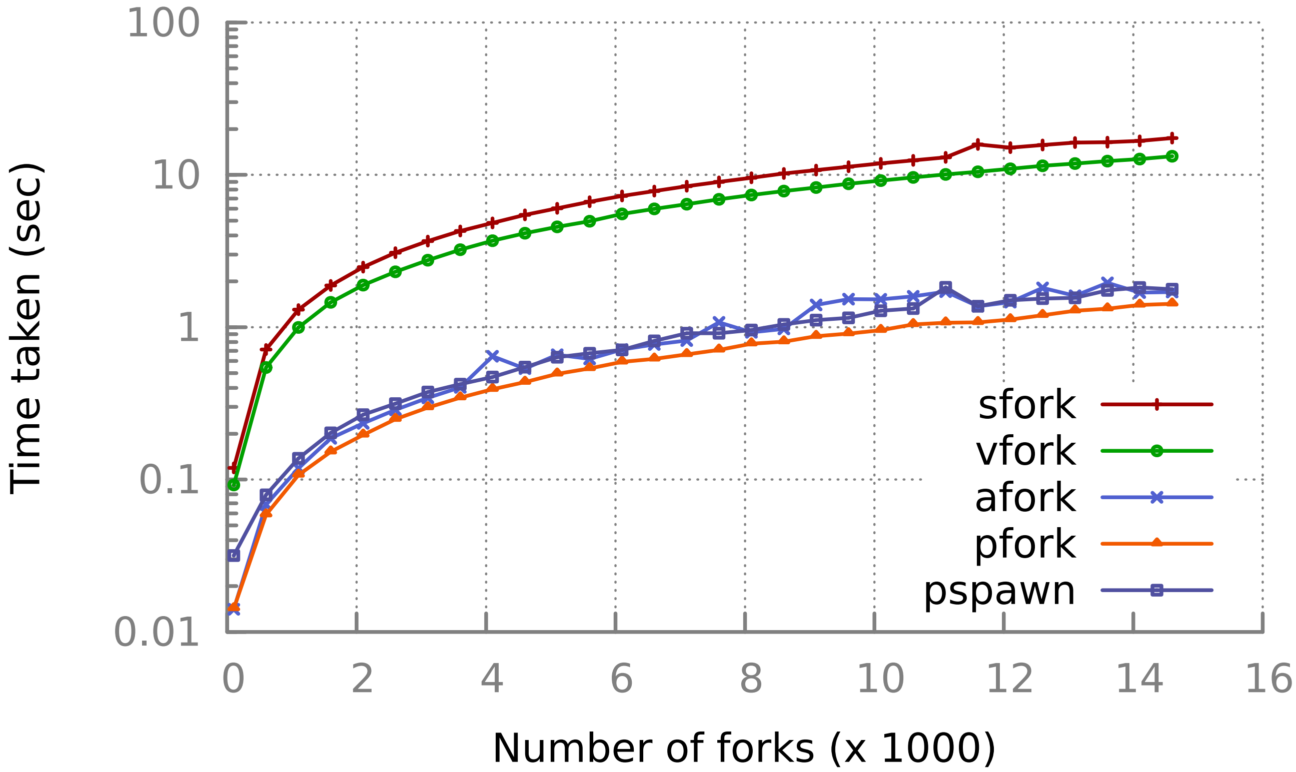


Typical setup on production machines

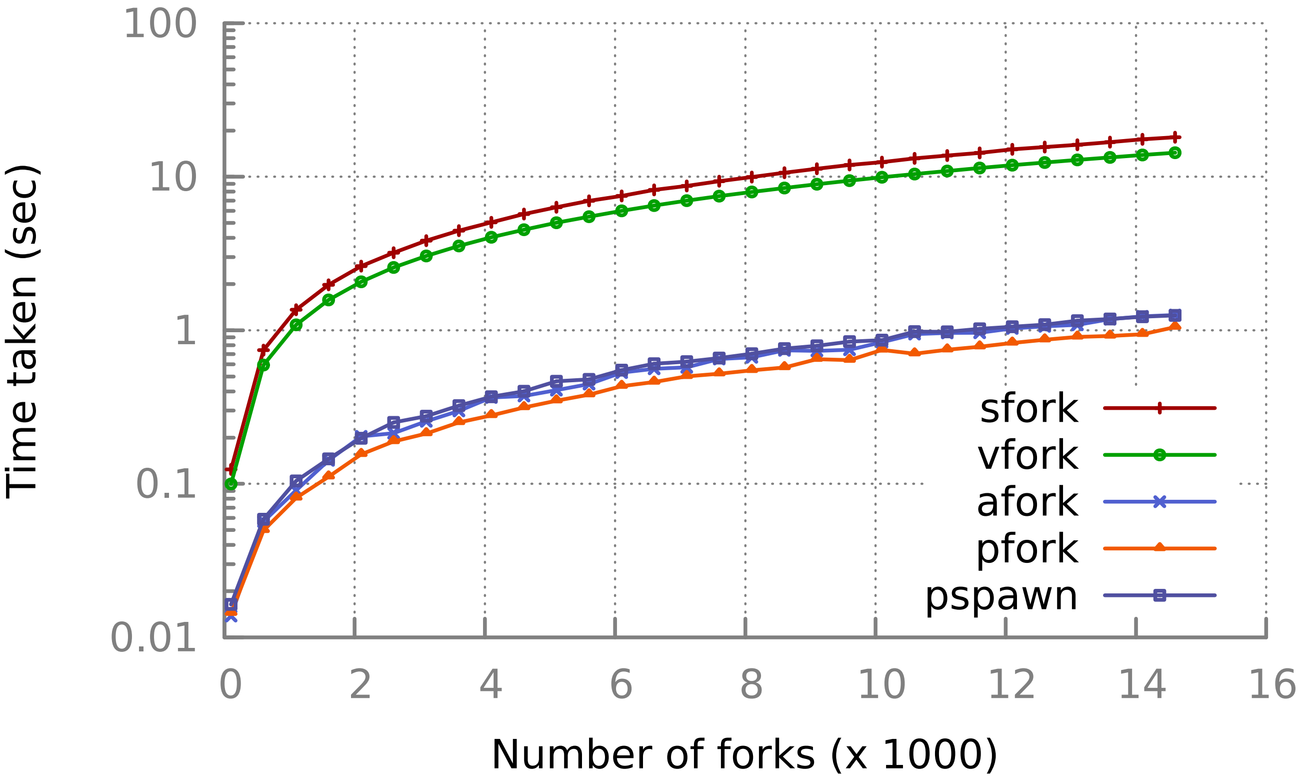
```
$ echo LD_LIBRARY_PATH
LD_LIBRARY_PATH=/home/user/opt/open64/x86_open64-4.2.5/lib:/home/user/
opt/mpi/openmpi-1.5.3-openf90/lib/:/home/user/opt/open64/x86_open64-4.2.5/
open64-gcc-4.2.0/lib/:/home/user/x86_open64-4.2.5/lib/gcc-lib/x86_64-open64-
linux/4.2.5:/home/GotoBLAS2/1.13_bsd:/home/cuda/cuda4.0.11/lib64:/home/
cuda/cuda4.0.11/lib:/home/user/opt/open64/x86_open64-4.2.5/lib:/home/user/
opt/mpi/openmpi-1.5.3-openf90/lib/:/opt/cuda/lib64
```

- Large number of entries in the environment (env) too!

Empty library path and env: 16-core node

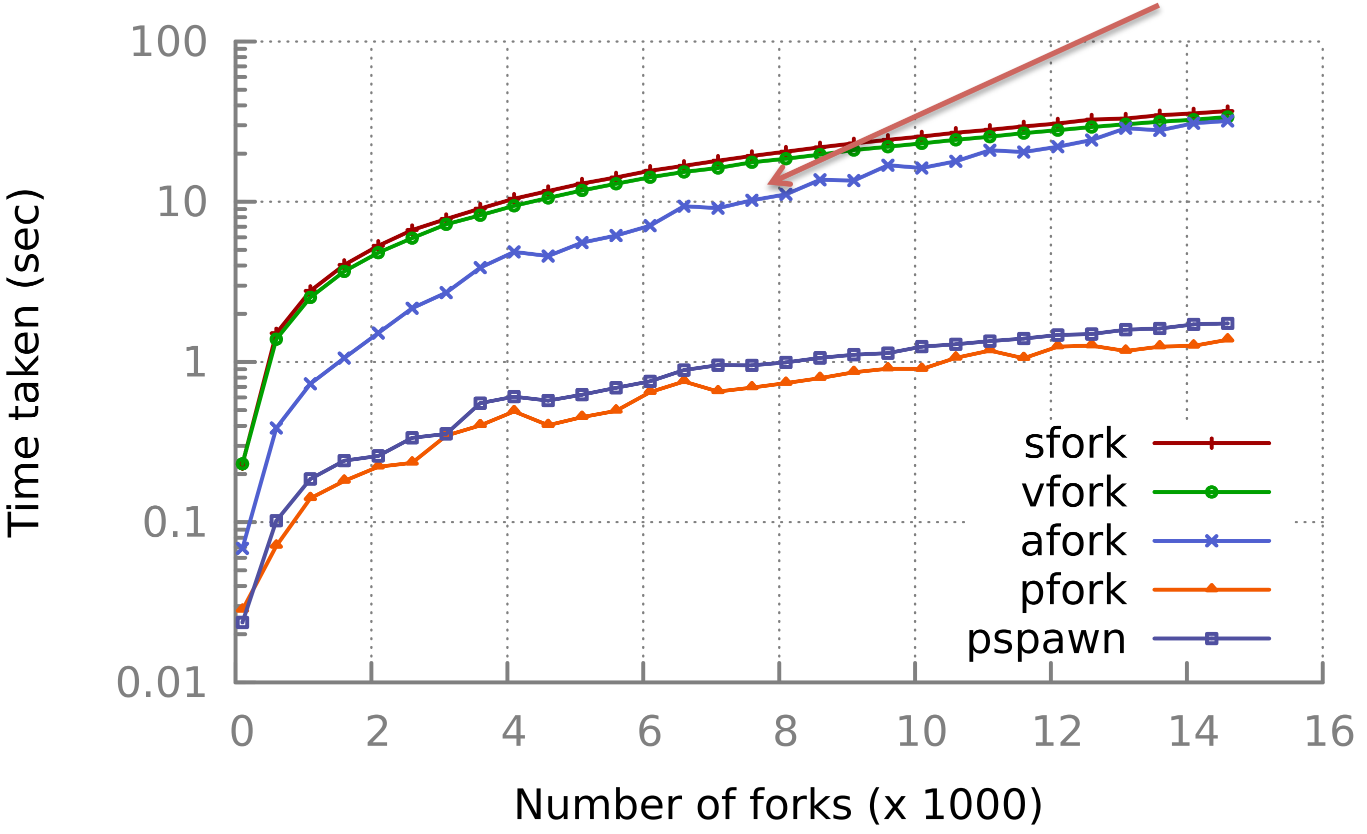


Empty library path and env: 24-core node



Empty library path and env: 64-core node

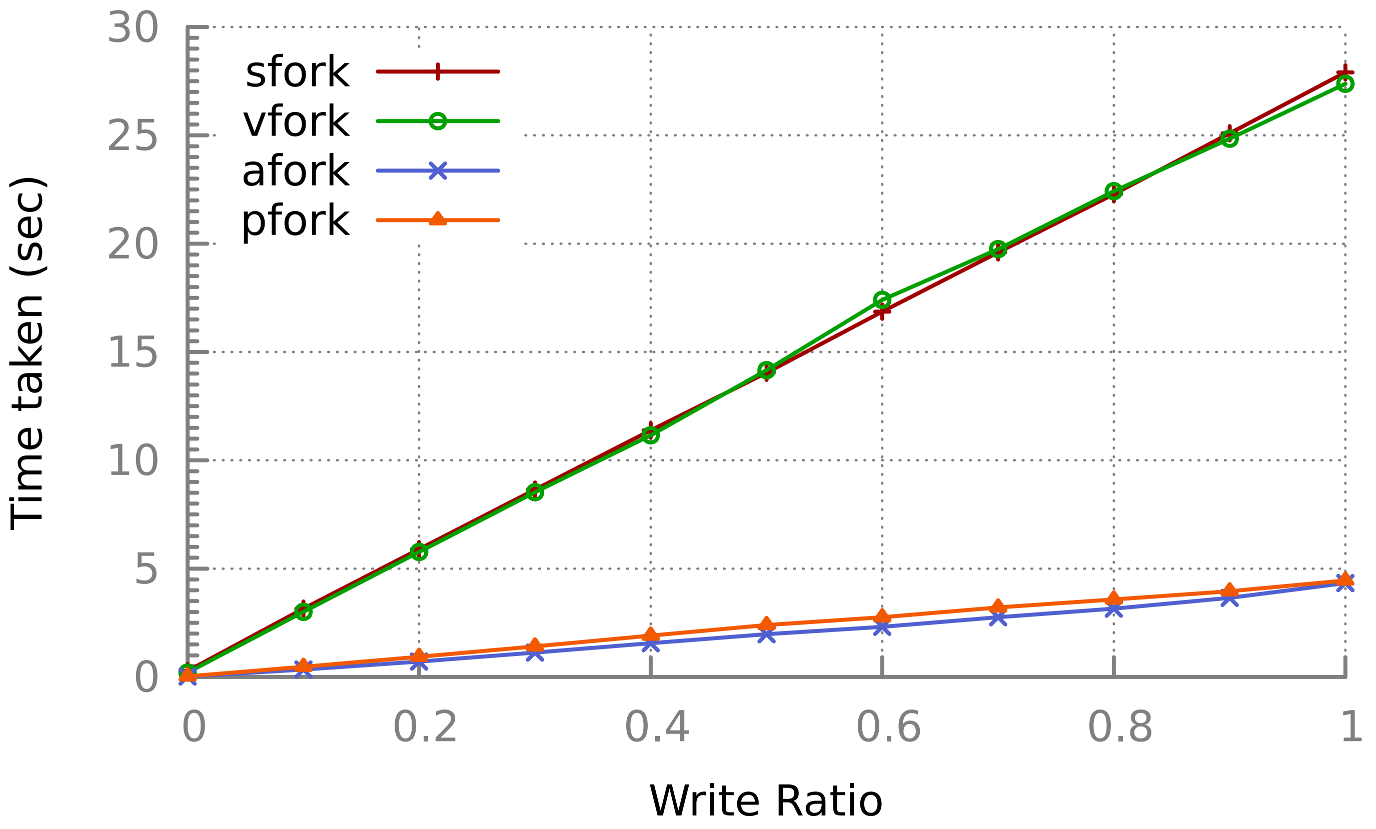
execve system call is expensive at higher core counts



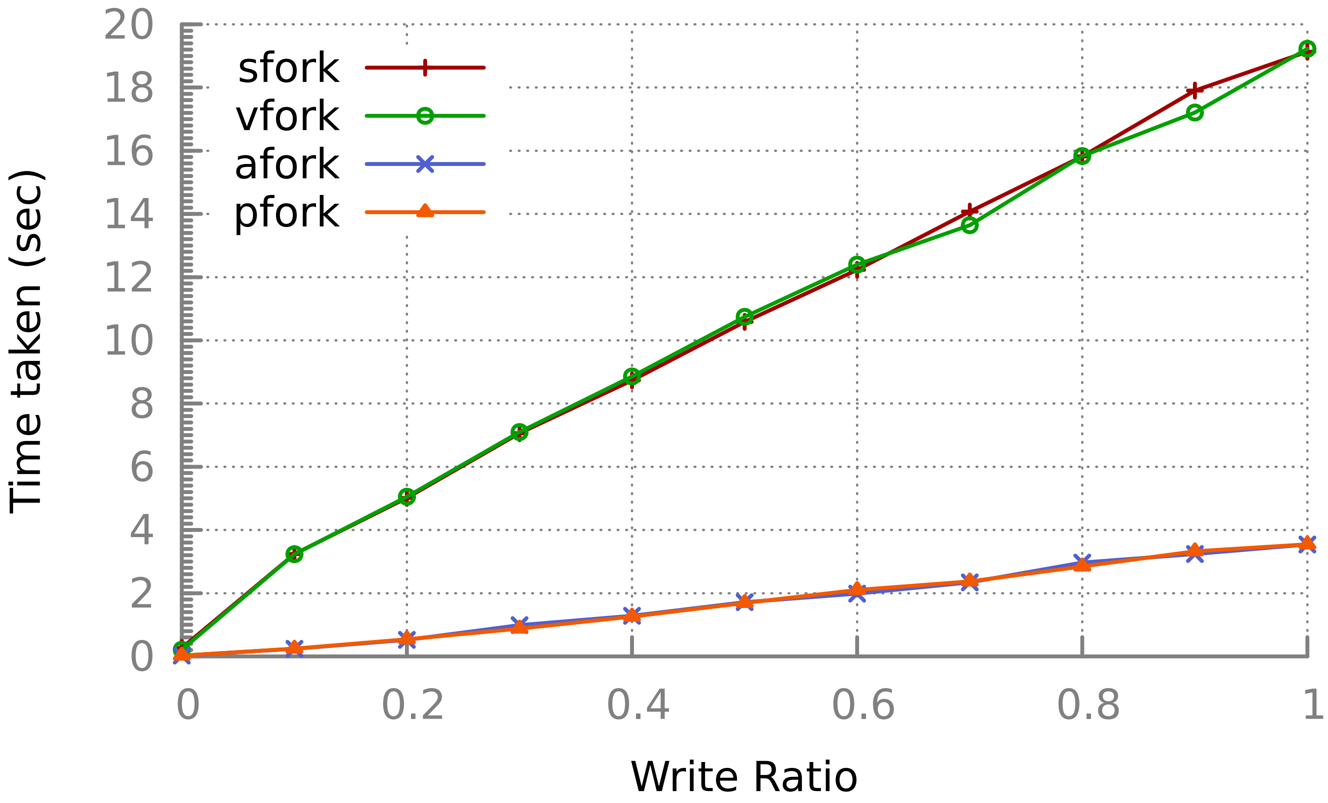
Accounting for the copy-on-write (CoW) overhead

- exec-fork-fork results in all children sharing the address space
- Executable image is typically read-only but accesses to heap, BSS would result in copying of pages
- We modified the microbenchmarks to run with an executable size of 100MB such that first byte of every page in the 100MB range was written to
- 200 instances of the microbenchmark were spawned

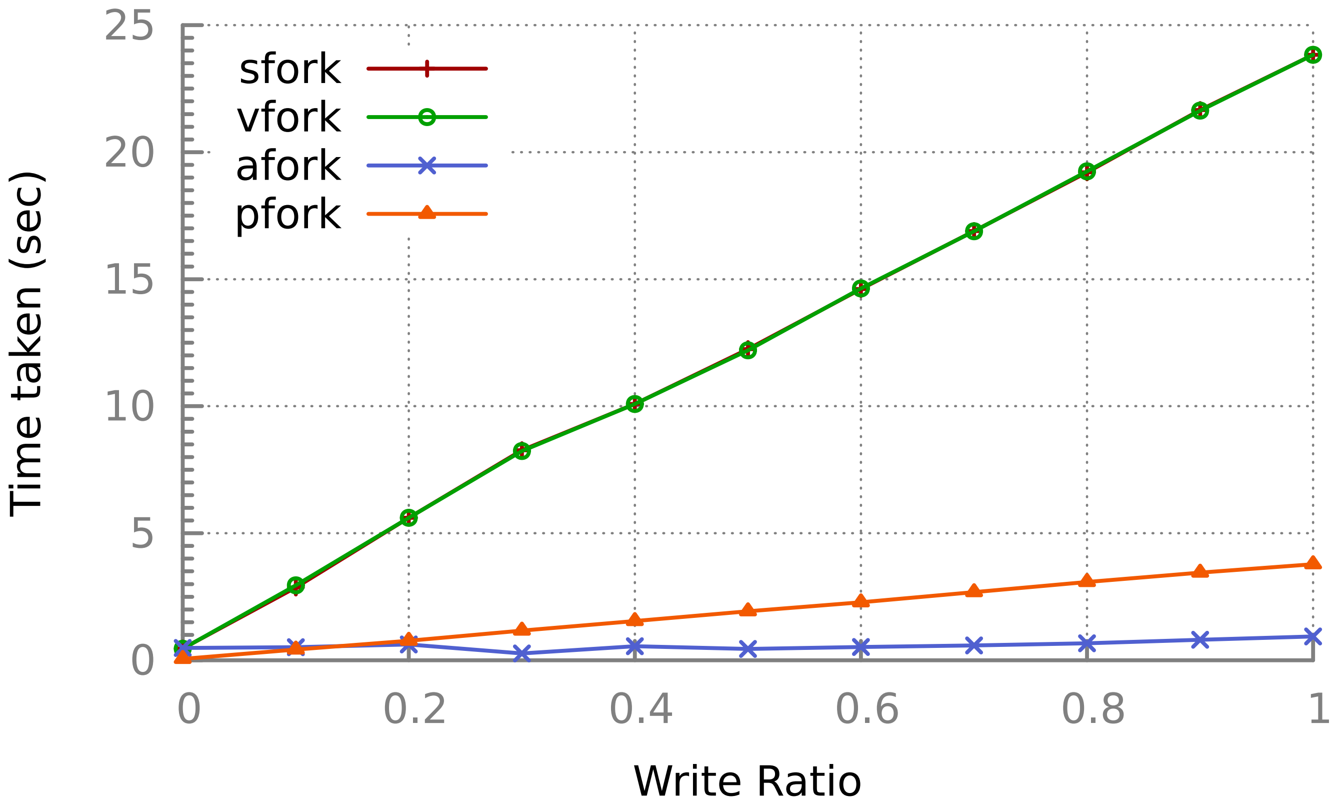
CoW overhead: 16-core node



CoW overhead: 24-core node

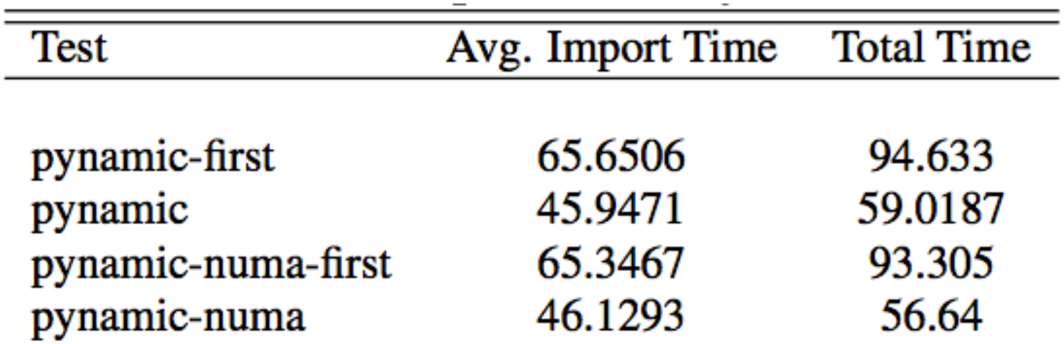


CoW overhead: 64-core node



NUMA-aware process spawning

- We replicated the shared libraries and executables local to each domain
- Isolated page caches using process containers (cgroups) in Linux
- Pynamic, the Python Dynamic Benchmark, is a synthetic benchmark to stress dynamic-linking and loading of applications
- 64 instances of the benchmark were run across 4 NUMA-domains on the 16-node cluster with 495 shared libraries each at an aggregate total of 1.4 GB

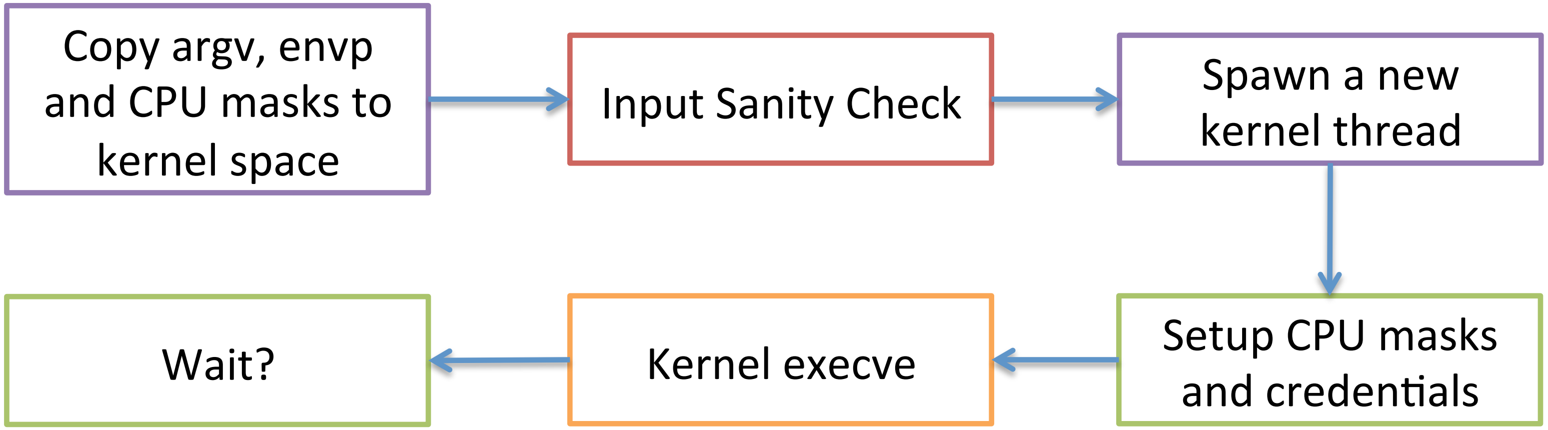


Test	Avg. Import Time	Total Time
pynamic-first	65.6506	94.633
pynamic	45.9471	59.0187
pynamic-numa-first	65.3467	93.305
pynamic-numa	46.1293	56.64

pspawn system call

- A vector system call to spawn a group of processes (**fork** and **exec**) on specific cores (**setaffinity**)
- Synchronous and Asynchronous system call interface
- Accepts a list of CPU mask flags that dictate what CPUs the processes are allowed to run on
- Accepts a list of arguments and environments (joined together) for each instance of the process spawn

pspawn system call



Copy argv, envp and CPU masks to kernel space

Input Sanity Check

Spawn a new kernel thread

Setup CPU masks and credentials

Kernel execve

Wait?

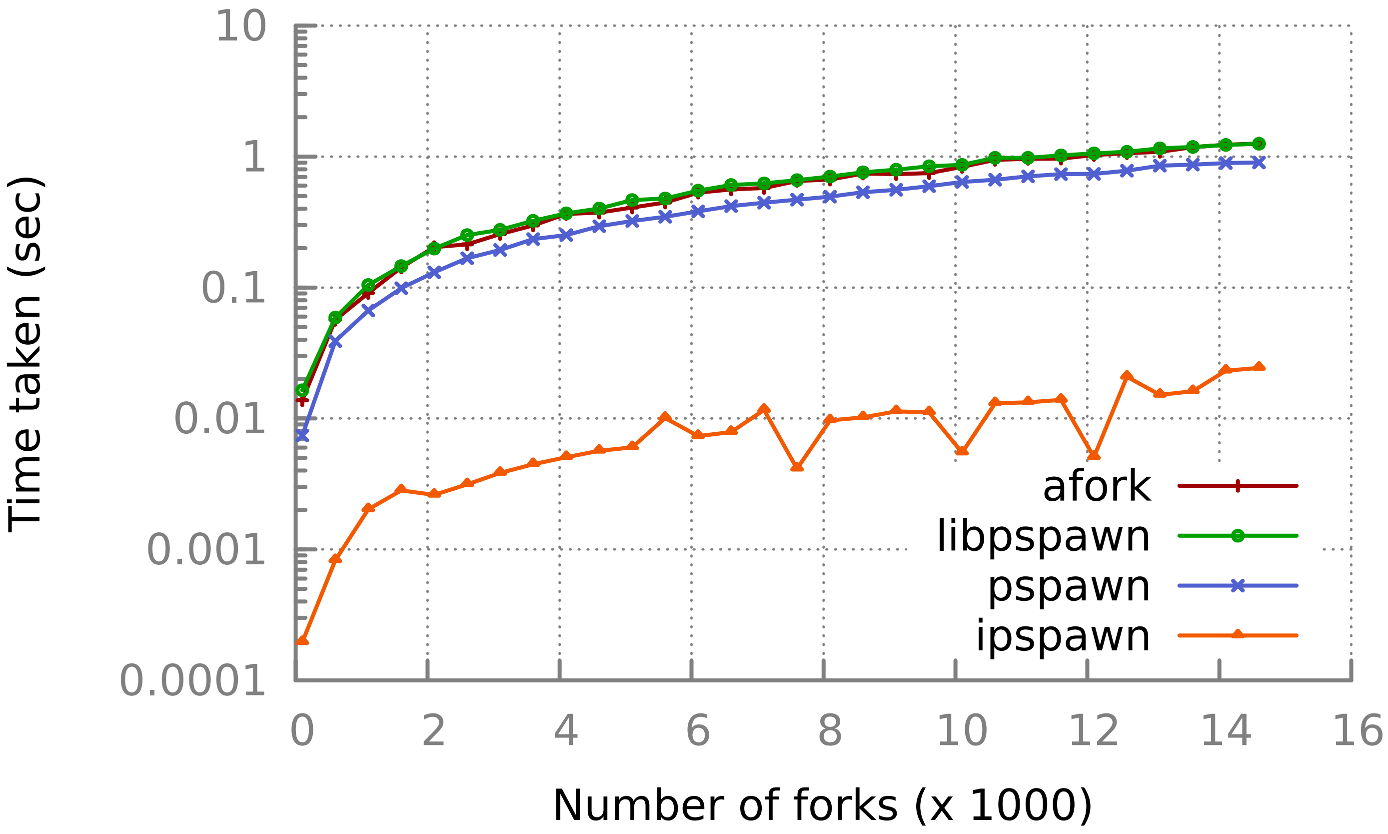
Synchronous pspawn system call

`int pspawn(char *filename, char **argv, char **envp, unsigned int nspawns, unsigned int clen, cpu_set_t **mask, enum pspawn_flags flags, pid_t *pids)`

Asynchronous ipspawn system call

`int ipspawn(char *filename, char **argv, char **envp, unsigned int nspawns, cpu_set_t **mask, enum pspawn_flags flags)`

pspawn system call performance



Related and Future Work

- Related work
    - CoW characterization and benchmarking
    - The Vector Operating System (VOS)
    - Google Chrome and the Dalvik Virtual Machine
- Future work
    - Runtime integration (MPI, SLURM)
    - Isolated page-caches
    - Better error reporting for asynchronous interfaces

Conclusion

- Significant focus on two related areas in the past:
  - i) improving OS interfaces for increased scalability
  - ii) increasing throughput of global distributed launches
- This work explores the several limiting factors for efficient intra-node spawning of processes on manycore architectures
- Synchronous system call interfaces are expensive at higher core-counts
- Vector operating system interfaces introduce an opportunity for parallelism!

Questions?