Measurement of Interprocess Communication Mechanisms

Adithya Bhat
adbhat@cs.wisc.edu
Department of Computer Sciences
University of Wisconsin-Madison

Stephen Lee
snlee@cs.wisc.edu
Department of Computer Sciences
University of Wisconsin-Madison

***Abstract*— Inter-process communication (IPC) refers to the mechanisms provided by Operating Systems to allow processes to co-ordinate and share information. Hence, One of the key characteristics of any operating system is the speed of its inter-process communication. We measured the latency and throughput of three forms of IPC: Unix pipes, TCP/IP sockets, and a bounded buffer implemented using shared memory. We used two different mechanisms to perform these measurements: the `clock_gettime()` system call and the Time Stamp Counter on our processors, after calibrating them to determine their precision. We hypothesized that pipes and the bounded buffer would be the fastest means of communication and sockets would be noticeably slower. We found that in fact our bounded buffer implementation is significantly slower on large message sizes.**

I. INTRODUCTION

Inter-process communication is an important technique for many software applications. While isolating code into different processes has many advantages, including fault-tolerance and security, interacting processes are common enough for IPC to be a major focus in OS design.

This paper measures the efficiency of three different forms of IPC: Unix pipes, Internet stream sockets, and shared memory. Pipes are a standard feature of Unix systems, are essentially unidirectional channels of communication via memory in kernel. Shared memory, is implemented as a bounded buffer using mutual exclusion locks and condition variables to manage concurrency. For socket-based communication, we use stream sockets with the Transmission Control Protocol.

We used two different mechanisms to measure the concurrency: the `clock_gettime()` system call and the Time Stamp Counter on the Intel® machines on which the measurements were run. We also calibrated the timing calls to determine their precision.

We hypothesized that pipes and shared memory would be comparatively faster, and that sockets would be noticeably slower, since the data has to be passed through the TCP/IP stack and could potentially be sent in fragments.

II. EXPERIMENTAL SETUP

A. System Specification

All of our experiments were run on a machine running RHEL Server release 6.7 with an x86_64 architecture. It had an Intel® Core i5-4570 processor. The CPU frequency returned by lscpu, utilized in our RDTSC measurements was 3.192475 GHz.

B. Timer Selection

The options for time measurement were `clock_gettime()`, `RDTSC`, and `gettimeofday()`. The resolution of `gettimeofday()` is in microseconds, as opposed to the nanosecond-level resolution provided by the other two timers. Hence, we use the other two.

We used the `clock_gettime()` system call with `CLOCK_MONOTONIC` as the option. this option ensures that the timer is not affected by changes such Network Time changes, and others, which could lead to forward or backward drift.

When measuring using the RDTSC Time Stamp Counter, we restricted the code to run on a single core by calling the `sched_setaffinity` system call immediately before timings began, and disabled out-of-order execution.

C. Timer Accuracy Calibration

We first calculated the overhead of the timers we were using. We did this by measuring the time taken by running 100,000 iterations of just timing calls, and processing the data by heuristically removing outliers, and then taking aggregates such as min and average, both of which were comparable over the observations. We found that `clock_gettime()` had around 65 ns of overhead and RDTSC had around 7ns on the machine we tested on. The range of variation in the overhead for both was +-2ns from the mean.

The resolution was calculated as the smallest discernible time that could be measured by our timers. This was calculated by finding the smallest set of instructions (multiple calls of the single increment `i++`) which consistently registered a difference between the preceding and succeeding timer calls. The average resolution for `clock_gettime` was around 3 nanoseconds, while the average resolution for RDTSC was around 1 nanosecond.

We also compared the values obtained from `clock_gettime()` and RDTSC, by plotting the variation from the mean of the two values as a percentage, and the results show that the disagreement between the timers used is always less than 0.5% of the mean of the two recorded values. Of course, this measure is only sensible if the absolute difference between the two is not large (in context), which was verified using the same methodology, plotting the percentage deviation of each from the other timer.

D. Experiment

For all of our experiments, we initialized two separate processes and sent messages between them using each of the three communication mechanisms. The controlled variable was message size and measured the time taken, using two different mechanisms. We used sizes from 4 bytes to 512 KB. We never measured the time in both processes, but always measured the difference in time from just one of the two communicating processes.

For each message size, we looped the timing call 100 times, and took the minimum of the results. The justification is that the first iterations right after the process starts are always outliers that take more time, after which the numbers converge to a value very close to the minimum.

While measuring latency, within each iteration, a message of the specified size is sent from one process to another, and this message is echoed back to the first.

While measuring throughput, the IPC channel is saturated by messages sent from from one process in batches of the specified size in each iteration. After the final iteration, the other process sends back a single byte message, which is used as an acknowledgment. The overhead of the single byte sent is taken to be negligible given the number of iterations.

E. Variables

The variables that we believe can affect IPC efficiency are message size, cache size, cache coherence

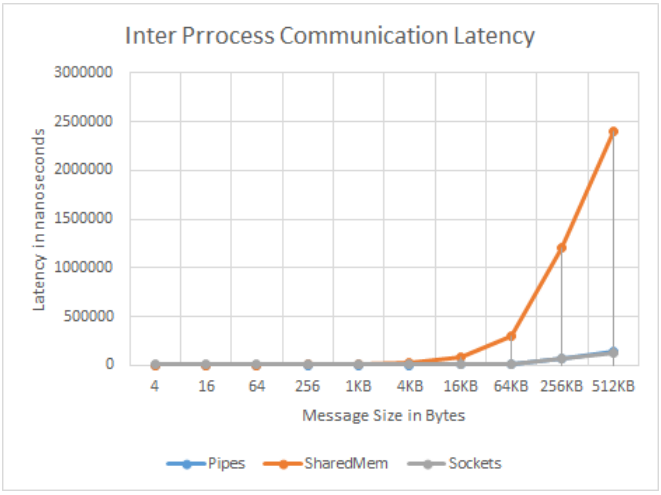

Fig. 1. The minimum latency of sending messages over different mechanisms.

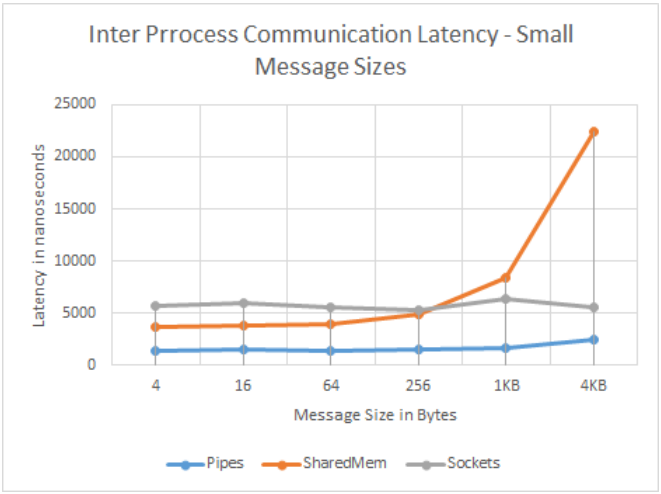

Fig. 2. The minimum latency of sending messages over different mechanisms for smaller message sizes.

mechanisms. Another important variable is whether or not the communicating processes are on the same processor/core or not.

Finally, there are the IPC implementation specific variables that can affect latency. In shared memory, mutual exclusion and coordination can be achieved by different mechanisms. In our experiment, we used the `pthread_mutex` and `pthread_cond` memory coordination mechanisms.

Socket communication latency can be changed by using UDP instead of TCP, or by setting or unsetting the TCP_NODELAY flag. This flag is set in our experiment.

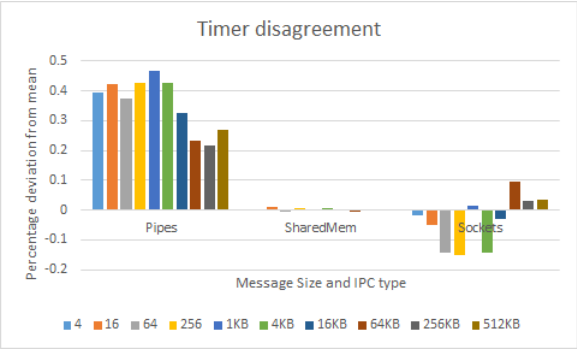

Fig. 3. The disagreement between the timers.

III. Results

A. Latency

We measured latency by determining the time taken for one process to send a message of the given size to the other and the other to send the message back.

The latency measurement results have been summarized in the accompanying Figure 1. The X-axis is essentially discrete. While that graph is for `clock_gettime()`, the graph for `rdtsc` was virtually indistinguishable. For smaller sizes upto 1KB, pipes and shared memory, were nominally more efficient. However, after 4KB, the latency of communication via shared memory increased hugely. Pipes and sockets were far more efficient, and had comparable performance.

The full data for minimum latency can be found in tables I and II. (Note that this gives measurements in nanoseconds, whereas the Y-axis of the graph is is microseconds.) Even with the actual numbers, you can see that the two timers returned very similar results. Under 256 bytes, all the mechanisms stayed fairly constant: this is probably because there is either some constant amount of time to send a message, or because the mechanism always sends messages of some minimum size. For pipes, this minimum size might well be 256 bytes.

B. Throughput

We measured throughput by saturating the IPC channel with messages sent from from one process in batches of the specified size, and then dividing the totla data sent by the number of messages. This gives measurements in bytes per second. You can see our results in figure III. Sockets had the highest throughput for messages with more that 64 KB of data, although pipes had higher throughput from 1 KB up to 16 KB.

TABLE I

GETTIME MINIMUM LATENCY MEASUREMENTS IN NANOSECONDS

Size	Pipes	Sockets	Shared Memory
4	1382	5689	3683
16	1460	6003	3752
64	1390	5504	3984
256	1504	5252	4898
1024	1678	6343	8387
4096	2455	5508	22441
16384	5226	8904	78307
65536	18095	16755	301261
262144	72622	61919	1201510
524288	146682	124026	2395788

TABLE II

RDTSC MINIMUM LATENCY MEASUREMENTS IN NANOSECONDS

Size	Pipes	Sockets	Shared Memory
4	928	5787	3680
16	952	6305	3720
64	951	6352	3985
256	974	6099	4871
1024	1042	6258	8368
4096	1588	6352	22307
16384	3769	9166	78058
65536	14310	15240	301892
262144	58324	60125	1196826
524288	111780	119740	2395543

TABLE III

THROUGHPUT MEASUREMENTS IN BYTES PER SECOND

Size	Pipes	Sockets	Shared Memory
4	2894356	703111	1086071
16	10958904	2665334	4264392
64	46043166	11627907	16064257
256	170212766	48743336	52266231
1024	610250298	161437806	122093717
4096	1668431772	743645606	182523061
16384	3135093762	1840071878	209227783
65536	3621773971	3911429424	217538945
262144	3609705048	4233660104	218178792
524288	3574317231	4227242675	218837393

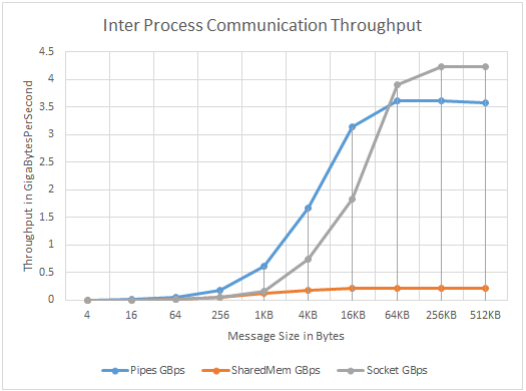

Fig. 4. Throughput for various message sizes

Shared memory remained the least efficient transfer mechanism.

IV. Conclusions

In this experiment, we have attempted to accurately benchmark and evaluate the performance of three commonly-used IPC mechanisms. We have also discussed methods for getting timer resolution and comparing timing mechanisms in order to get meaningful results.

Our results show that IPC via shared memory performs poorly in larger sized communications between processes on different cores, while pipes and INET sockets performed comparably and much better than shared memory.

With the caveat that there are several variables that affect these results that we have not comprehensively explored in this paper, we believe that the results obtained are accurately indicative of the comparative performance of these mechanisms.