Shared Memory Remote Procedure Calls

Jon Chesterfield
jonathanchesterfield@gmail.com

ABSTRACT

The remote procedure call (RPC) is a simple interface for executing code on a different machine. Almost none of the well known problems inherent to RPC apply on a shared memory system. Further, a shared memory system is sufficient to implement a RPC library, so that said simple interface can be more widely available.

This paper includes a minimal implementation of the proposed algorithm, with a real world implementation tested on x86-64, AMDGPU and NVPTX architectures on GitHub[4]. This can bring host capabilities to the GPU or offload code without using kernel launch APIs. The client and server both compile and run on each architecture.

KEYWORDS

Shared memory, GPU compute, Remote procedure calls

ACM Reference Format:
Jon Chesterfield. 2021. Shared Memory Remote Procedure Calls. In *LLPP '21: The First Workshop on LLVM in Parallel Processing (LLPP), August 9th, 2021, Chicago (Argonne National Lab), Illinois, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The remote procedure call (RPC) is a simple interface for executing code on a different machine. It's a function call, same as any other. That surface simplicity hides a variety of well known problems as characterised by Tanenbaum and Renesse[9]. Vinoksi's[10] paper arguing to retire RPC on similar grounds is titled "Convenience over Correctness".

The observation behind this paper is that almost none of the problems inherent to RPC apply on a shared memory system, such as a heterogeneous host and graphics processing unit(s) (GPU) machine. Further, a shared memory system with limited write ordering guarantees is sufficient to implement a RPC library so this convenience can be made more widely available. This paper includes a minimal implementation of the proposed algorithm, with a real world implementation tested on x86-64, AMDGPU and NVPTX architectures on GitHub[4].

2 APPLICATION

This is a low level library implemented in freestanding C++. It provides a state machine for coordinating access to a preallocated block of shared memory between host and GPU. The interface exposed permits passing a fixed number of integers between threads running on different processors in the same machine. Since some integers can be pointers to further shared memory, this is sufficient to pass arbitrary data. Design inspiration was drawn from the Linux syscall interface. The association with LLVM is from exploiting Clang intrinsics to achieve the cross-language compilation and an intent to implement the libc related parts of the LLVM OpenMP runtime libraries with it. Some syntax choices are forced by working in the common subset of C++, CUDA, HIP, OpenCL C++ or OpenMP.

2.1 Interface

The proposed interface is approximately that of Listing 1.

```
struct callback_t {
  void operator()(char *buffer);
};

struct client {
  client(void *);

  template <typename Fill, typename Use>
  bool rpc_invoke(Fill fill, Use use) noexcept;

  template <typename Fill> bool rpc_invoke(Fill
      fill) noexcept;

private:
  void *state;
};

struct server {
  server(void *);

  template <typename Operate, typename Clear>
  bool rpc_handle(Operate op, Clear cl) noexcept;

private:
  void *state;
};
```

Listing 1: Pseudocode interface

Architecture specific details, such available atomic operators and whether calls should be batched, are encoded in the client/server type. Application specific details are provided by invocable types like callback_t, passed by value to rpc_invoke and rpc_handle.

2.2 Hello World

Hello world from a HIP kernel, implemented by passing raw Linux syscall numbers to the host application, is shown in Listing 2. This is backed by a host thread which can only allocate/free shared memory and invoke a syscall on the integers passed to it by the GPU.

LLPP '21, August 9th, 2021, Chicago (Argonne National Lab), Illinois, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
__global__ extern "C" void on_gpu(
    hostrpc::x64_gcn_type<SZ>::client_type *client
        , int, char **, int *)
{
  auto inv = [=](uint64_t x[8]) -> bool {
    return invoke<hostrpc::x64_gcn_type<SZ>::
        client_type>(client, x);
  };

  const uint64_t buffer_size = 16;
  uint64_t tmp[8];
  tmp[0] = hostrpc::allocate_op_hsa;
  tmp[1] = buffer_size;
  inv(tmp);

  char *buf = (char *)tmp[0];

  buf[0] = 'h';
  buf[1] = 'i';
  buf[2] = '\n';
  buf[3] = '\0';

  tmp[0] = hostrpc::syscall_op;
  tmp[1] = __NR_write;
  tmp[2] = 2;
  tmp[3] = (uint64_t)buf;
  tmp[4] = 3;

  inv(tmp);

  tmp[0] = hostrpc::syscall_op;
  tmp[1] = __NR_fsync;
  tmp[2] = 2;

  inv(tmp);

  tmp[0] = hostrpc::free_op_hsa;
  tmp[1] = (uint64_t)buf;
  tmp[2] = buffer_size;
  inv(tmp);
}
```

Listing 2: Hello world via syscall

2.3 GPU memory allocation in LLVM

The first real world use case for this library is intended to be providing GPU side malloc and free for LLVM's AMDGPU OpenMP implementation. This is summarised here, the details can be found in openmp_hostcall.cpp[4]. That file, when compiled into both the host and device runtime libraries, replaces weak stub symbols in the LLVM library with a working allocator. Most of the complexity left out here is matching the existing API in the host runtime, e.g. dedicating a thread to repeatedly calling rpc_handle and retrieving the RPC instance.

The client GPU callbacks, Listing 3, write an opcode to the start of the shared buffer followed by forwarding function arguments, call rpc_invoke, then copy results back. As free() returns void, the RPC returns without waiting for the round trip to the server. The type, x64_gcn_type<runtime>, manages the memory for the RPC instance. An instance of this type outlives calls from the AMDGPU client to the x86-64 server.

The #pragma omp target annotation is for compilation as part of the LLVM deviceRTL library which implemented in OpenMP. The host side is compiled as C++ to match the LLVM libomptarget plugins. Using different languages for the client and server is convenient for the build scripts.

```
#if HOSTRPC_AMDGCN
#pragma omp declare target

// overrides weak functions in target_impl.hip
extern "C" {
void *__kmpc_impl_malloc(size_t);
void __kmpc_impl_free(void *);
}

using client_type = hostrpc::x64_gcn_type<hostrpc
    ::size_runtime>::client_type;
static client_type *get_client();

void *__kmpc_impl_malloc(size_t x) {
  uint64_t data[8] = {0};
  data[0] = opcodes_malloc;
  data[1] = x;
  fill f(&data[0]);
  use u(&data[0]);
  client_type *c = get_client();
  bool success = false;
  while (!success) {
    success = c->rpc_invoke(f, u);
  }
  void *res;
  __builtin_memcpy(&res, &data[0], 8);
  return res;
}

void __kmpc_impl_free(void *x) {
  uint64_t data[8] = {0};
  data[0] = opcodes_free;
  __builtin_memcpy(&data[1], &x, 8);
  fill f(&data[0]);
  client_type *c = get_client();
  bool success = false;
  while (!success) {
    success = c->rpc_invoke(f); // async
  }
}

#pragma omp end declare target
#endif
```

Listing 3: Memory allocation, AMDGPU part

The server host callbacks, Listing 4, read the opcode from the buffer and act on it, or reset the buffer opcode to no operation. This latter is useful when different client calls are made with different GPU threads active in the warp. The threads write to offsets based on their ID and the inactive ones pick up no-op written by the server previously.

```
struct operate {
  hsa_region_t coarse_region;
  operate(hsa_region_t r) : coarse_region(r) {}
  void op(hostrpc::cacheline_t *line);

  void operator()(hostrpc::page_t *page) {
    for (unsigned c = 0; c < 64; c++)
      op(&page->cacheline[c]);
  }
};

struct clear {
  void operator()(hostrpc::page_t *page) {
    for (unsigned c = 0; c < 64; c++)
      page->cacheline[c].element[0] = opcodes_nop;
  }
};

// in a loop on a pthread,
// server->rpc_handle<operate, clear>(op, clear);

void operate::op(hostrpc::cacheline_t *line) {
  uint64_t op = line->element[0];
  switch (op) {
  case opcodes_nop: {
    break;
  }
  case opcodes_malloc: {
    uint64_t size;
    memcpy(&size, &line->element[1], 8);

    void *res;
    hsa_status_t r = hsa_memory_allocate(
        coarse_region, size, &res);
    if (r != HSA_STATUS_SUCCESS) {
      res = nullptr;
    }

    memcpy(&line->element[0], &res, 8);
    break;
  }
  case opcodes_free: {
    void *ptr;
    memcpy(&ptr, &line->element[1], 8);
    hsa_memory_free(ptr);
    break;
  }
  }
  return;
}
```

Listing 4: Memory allocation, host part

3 ALTERNATIVES

The state of the art is to use C++ derived programming languages with GPU extensions. These define kernels explicitly, possibly in the same source, possibly with language support for copying data to/from the kernels. These kernel functions are run, sometimes implicitly, on the GPU.

This proposal primarily adds the ability for those kernels to request services from the host while they are still running. For example, add this to a CUDA application, and a refinement of Listing 2 can provide file I/O.

If the client and server architectures are swapped, this provides a means of running code on the GPU, optionally asynchronously. This should be expected to be slower than the native kernel launch.

OpenMP reverse offloading may fit in the same design space as this but the author does not know of an implementation of it.

4 BACKGROUND

Terminology is not uniform in this domain. Let client be the entity that initiates the procedure call and server be the one that does the work of the call. Process will refer to either client or server. Thread will refer to a posix thread on a CPU, to a warp on NVPTX, or to a wavefront on AMDGPU.

Remote procedure calls (RPC) are a procedure call that executes on a different process. Syntactically they are usually a local function that forwards the arguments to the remote process and retrieves the result before returning. The local functions, known as stubs, are frequently generated from declarative code as the argument forwarding process is mechanical.

4.1 Known problems with RPC

This section follows those articulated by Tanenbaum and Renesse [9], written towards the decline of industry enthusiasm for RPC as a distrubuted computation strategy. "2.3[9]" notes that RPC implies a multithreaded server but in the context of GPU compute, single threaded systems are ruled out by performance requirements anyway.

4.1.1 *When RPC doesn't fit the domain.*

RPC is not a universal solution to remote computation. A calculation that can be done quicker locally, "6.2.1. Bad Assumptions[9]", or one with real time constraints "3.4 Timing Problems[9]" should be done locally. The client/server pairing doesn't map easily onto "2.5 Multicast[9]" or compute pipelines "2.1[9]", though pipelines are similar to continuation passing style which is considered in subsection 8.1.

4.1.2 *Partial failures.*

Where RPC crosses a network it is exposed to the failure mode of the network. Multiple problems listed are consequences of defining a function that does not forward failure information, "2.2 Unexpected Messages, 2.4 The Two Army Problem, all four sections of 4 Abnormal Situations[9]".

Modern RPC frameworks accept this. Apache Thrift[2] reports exceptions on infrastructure failures, to be handled by the application. Google's gRPC[6] returns a message that includes failure information alongside the call result. However, embracing the reality of network failures changes call interfaces and introduces error handling at the call site. It no longer looks like a local function call.

A single node shared memory machine uses higher reliability communication between components than an external network, e.g. PCI Express (PCIe), a common interface for GPU to host system, includes error detection and recovery. Practically, expansion cards are less prone to unreliable connections or cables being unplugged in service than external networking. Further, an error in communication between processors within a single node can be expected to crash the processor or the entire node. Error recovery is then at the user or cluster level.

The implementation suggested here does not amend the interface to propagate errors as that removes the programmer convenience. It is thus only appropriate when failures are not partial and will need handling at the system level.

4.1.3 ABI concerns. "3.1 Parameter Marshalling, 3.2 Parameter Passing and 3.3 Global Variables[9]" all require some care. The implementation associated with this paper passes N uint64_t values and expects a layer above to serialize types into those N arguments, or into shared memory to be passed by pointer. The heterogeneous machine hazard is present but largely solved on existing shared memory systems by choosing compatible representations, with the edge cases handled in serialisation.

"Lack of Parallelism" is unlikely to be a problem with both server and client multi-threaded. "Lack of Streaming", where the client waits on the server, is addressed in subsection 8.1.

4.2 Distributed computing

Sun Microsystems published a note on distributed computing[11] which offers an object orientated perspective on local and distributed computation fundamentally differing. Partial failure and inability to directly pass pointers are the invasive problems for API design. The final section of the paper describes a middle ground, where the objects are guaranteed to be on the same machine, in which case indeterminacy is largely the same as for a single process. It does not distinguish a common address space from a local computation.

The thesis of this paper is essentially that shared memory systems, where said shared memory is not subject to network failure modes, are much closer to local computation than to distributed.

5 REQUIREMENTS

The two processes require access to shared memory, implemented with sufficient write ordering that an atomic write to a flag is seen after writes to a buffer. PCIe may require the flag to be at a higher memory address than the buffer for that to be robustly true. The CUDA and HSA programming environments meet that requirement if appropriate fences are used. Atomic load and store are sufficient, compare and swap better, fetch_and/fetch_or ideal.

That is, given a shared memory system that allows control over the order in which writes are seen, one can implement remote procedure calls to make easier use of said shared memory system.

6 MOTIVATION

6.1 Host services

GPU programming is primarily based on a host processor offloading tasks to a GPU. This is the case for languages CUDA, HIP, OpenCL, OpenMP, SYCL, DPC++. Exceptions are the reverse offload work of Chen et al. [3], source unavailable, which runs on an Intel MIC chip and uses a form of RPC to execute some tasks on the host processor and the as yet unimplemented reverse offloading feature of OpenMP 5.0[1], section 4.1.6.

There are tasks that the GPU cannot do without cooperation from the host, such as file and network I/O or host memory allocation. Some functions may be special cased in the compiler for some languages, e.g. printf works from CUDA GPU code (it writes to stdout at kernel exit) but fprintf is unavailable. Allocating shared memory from code running on the GPU is generally unavailable. A library implementation of RPC can be used to fill in the gaps across all language implementations, or as a means of implementing features like OpenMP 5.0 reverse offloading.

The Linux kernel syscall interface is essentially a named function call taking a fixed number of integer arguments, albeit implemented with hardware support. If the RPC function is set up to pass integers from the GPU to the host syscall interface, the GPU is granted direct access to whatever syscalls the associated host process is able to make. For example, __NR_open, __NR_write, __NR_fsync, __NR_close in sequence can be used to write to a file on the host.

6.2 Fine grain offload

A persistent kernel launched on the GPU can act as the server process while threads on the host (or another GPU) are the client process. The client can then run functions on the GPU through the RPC infrastructure instead of through the kernel launch API. This is likely to be slower than the vendor provided API, The kernel API may use memory allocation or waiting on asynchronous signals whereas this RPC is zero syscall and based on polling as that is the lowest common denominator.

Launching a kernel, particularly across language boundaries such as an OpenMP target region run through HIP host APIs, is subtle and error prone. Using the RPC interface instead allows implementing functions in one language and calling from another without any additional complexity. The RPC implementation itself needs to work with the native kernel API for setup and completion. Once implemented in the library however, applications can add and call functions with greater convenience.

6.3 Process isolation

If both client and server run as Linux processes on the same CPU, RPC on shared memory provides a zero syscall means of communicating between the two processes. A sandbox can then be implemented for a Linux client process by using seccomp to irreversibly drop access to syscalls with the still open RPC connection used to request services from the server. This may be a reasonable way to handle just in time compilation for a memory safe language implementation.

7 UNDERLYING ALGORITHM

Implementing mutual exclusion on shared memory has known solutions, such as that attributed to Th. J. Dekker[5] and the later Peterson's algorithm[7]. Those, and more complicated subsequent solutions, are not ideal for RPC which requires strictly alternating access to the shared buffer.

	Client		Server		Client	Server	Buffer
State	In	Out	In	Out			
Quiescent	0	0	0	0	0	0	Client
Work posted	0	1	0	0	1	0	-
	0	1	1	0	1	2	Server
Server working	0	1	1	0	1	2	Server
Result posted	0	1	1	1	1	3	-
	1	1	1	1	3	3	Client
Client working	1	1	1	1	3	3	Client
Client finished	1	0	1	1	2	3	-
	1	0	0	1	2	1	Server
Server finished	1	0	0	0	2	0	-
	0	0	0	0	0	0	-
Client return	0	0	0	0	0	0	Client

Table 1: State transitions

7.1 One client, one server

The complexity is in the single client, single server case. Scaling up to multiple of each, subsection 7.3, involves multiple independent instances of the base case. This section describes the algorithm in prose, Table 1 as a state transition, subsection 7.2 as executable code.

Where boolean is the smallest integer the processes can write to atomically, the client and server each have access to, in shared memory:

- boolean outbox, to which it may atomically write 0 or 1
- boolean inbox, from which is may atomically read 0 or 1
- fixed size buffer from which it may read and write N bytes

The boolean mailboxes are strictly single writer. The client outbox is the server inbox, writable by the client. The client inbox is the server outbox, writable by the server.

The state change is strictly sequential. Table 1 shows the changes for a complete call, proceeding down the rows. After writing to the outbox, the process waits for a change to the inbox value caused by the other process writing. Read/write access to the buffer is based on the process local mailbox values, chosen such that at most one process has access at a time.

Starting from all mailboxes containing zero and leaving optimisations aside, the calling sequence from the client is:

- Write arguments to the fixed size buffer
- Write 1 to the outbox
- Wait for the inbox to change to 1
- Read results from the fixed size buffer
- Write 0 to the outbox
- Wait for the inbox to change to 0
- Return

The corresponding sequence from the server is:

- Wait for the inbox to change to 1
- Read arguments from the fixed size buffer
- Do work as specified by arguments
- Write results to the fixed size buffer
- Write 1 to the outbox
- Wait for the inbox to change to 0
- Write 0 to the outbox
- Goto start

7.2 Executable implementation

This is a minimal implementation of the one-to-one state machine, with no optimisations or cross architecture concerns, written for exposition. It will compile (as C++14) and run successfully if the listings are concatenated in order and the four external functions implemented, e.g. to do arithmetic or print to the console.

The two processes have the same fields as represented by the common base in Listing 5. Each exposes a templated function as the application hook, here shown as calls to external C functions.

```
#include <atomic>
#include <cstdint>
#include <cstdio>
#include <memory>
#include <thread>
using namespace std;

struct process_t {
  const atomic_bool *inbox;
  atomic_bool *outbox;
  uint32_t *buffer;
};

struct client_t : public process_t {
  template <typename F, typename U> void run(F
      fill, U use);
};

struct server_t : public process_t {
  // return true if a callback was invoked
  template <typename W, typename C> bool run(W
      work, C clean);
};

// Unimplemented here
void client_fill(uint32_t *);
void server_work(uint32_t *);
void client_use(uint32_t *);
void server_clean(uint32_t *);
```

Listing 5: Types

The memory allocation is not owned by the client or the server (as in C++ RAII) as both client and server access the same memory. For GPU systems, the allocation is likely to be done by the host, in which case the GPU may not be able to deallocate the corresponding memory. In the GitHub[4] implementation one type instance, separate to client and to server, owns the allocated memory and outlives the processes. Here, Listing 6 puts the state on the free store, managed by stack objects, and spawns separate C++ threads to serve as the RPC processes. The calls variable represents minimal plumbing to handle process shutdown.

```
server_t server;
client_t client;

int main() {
  const uint32_t calls = 10;
  auto box0 = make_unique<atomic_bool>();
  auto box1 = make_unique<atomic_bool>();
  auto data = make_unique<uint32_t[]>(4);

  client.inbox = server.outbox = box0.get();
  server.inbox = client.outbox = box1.get();
  client.buffer = server.buffer = data.get();

  thread st([]() -> void {
    for (uint32_t count = 0; count < 2 * calls;) {
      if (server.run(server_work, server_clean))
        count++;
    }
  });

  thread ct([]() -> void {
    for (uint32_t i = 0; i < calls; i++)
      client.run(client_fill, client_use);
  });

  st.join();
  ct.join();

  return 0;
}
```

Listing 6: Main

The client and server (Listing 7) implementations each make the handling of the four possible states explicit instead of folding the dead branches for clearer comparison to the state transitions of Table 1.

```
template <typename F, typename U> void client_t::
    run(F fill, U use) {
  bool in = inbox->load(memory_order_relaxed);
  bool out = outbox->load(memory_order_relaxed);
  atomic_thread_fence(memory_order_acquire);

  if (!in & !out) {
    // ready! write to buffer then to outbox
    fill(buffer);
    atomic_thread_fence(memory_order_release);
    outbox->store(1, memory_order_release);
    out = 1;
  }

  if (!in & out) {
    // wait for result
    while (!in)
      in = inbox->load(memory_order_relaxed);
    atomic_thread_fence(memory_order_acquire);
  }

  if (in & out) {
    // read from buffer then write to outbox
    use(buffer);
    atomic_thread_fence(memory_order_release);
    outbox->store(0, memory_order_release);
    out = 0;
  }

  if (in & !out) {
    /// wait for server to garbage collect
    while (in)
      in = inbox->load(memory_order_relaxed);
    atomic_thread_fence(memory_order_acquire);
  }
}
template <typename W, typename C> bool server_t::
    run(W work, C clean) {
  bool in = inbox->load(memory_order_relaxed);
  bool out = outbox->load(memory_order_relaxed);
  atomic_thread_fence(memory_order_acquire);

  if (in & out) {
    // work done, wait for client
    while (in)
      in = inbox->load(memory_order_relaxed);
    atomic_thread_fence(memory_order_acquire);
  }

  if (!in & out) {
    // all done, clean up buffer
    clean(buffer);
    atomic_thread_fence(memory_order_release);
    outbox->store(0, memory_order_release);
    out = 0;
    return true;
  }

  if (!in & !out) {
    // nothing to do, wait for work
    while (!in)
      in = inbox->load(memory_order_relaxed);
    atomic_thread_fence(memory_order_acquire);
  }

  if (in & !out) {
    // do work then signal client
    work(buffer);
    atomic_thread_fence(memory_order_release);
    outbox->store(1, memory_order_release);
    out = 1;
    return true;
  }

  return false;
}
```

Listing 7: Client and Server

7.3 Many clients, many servers

The one-to-one client/server state machine requires exclusive ownership of the memory used to communicate between the two. Scaling to multiple clients or multiple servers is done with multiple one-to-one state machines, each of which runs independently and as described above, with additional locking within the process to manage mutual exclusion of the scalar state machines. No additional coordination is needed between processes.

7.3.1 Thread scheduler. Linux provides a completely fair scheduler by default. A thread which takes a lock and is suspended will ultimately be rescheduled, allowing the system as a whole to make progress. CUDA does not preemptively schedule threads (warps in CUDA terminology); once one starts executing it will run to completion, modulo the program ending prematurely. This also makes simple locking code possible. OpenCL provides no forward progress guarantees, and HSA makes limited ones[8]. This implementation assumes the scheduler is not fair, such that a thread which holds a lock may be descheduled and never return. Global locks are therefore unavailable. Forward progress can be ensured on AMDGPU by using at least as many distinct state machines as there can be simultaneous wavefronts on a HSA queue.

7.3.2 Implementation limits. This implementation assumes a limit on the number of concurrent RPC calls is specified at library initialization time. For example, it may be limited by the maximum number of concurrently executing threads the hardware can support. It then allocates that many instances of the communication state up front, as a contiguous array, to avoid the complexity of reallocating concurrently accessed structures. On contemporary AMDGPU hardware it implies 8MiB of host memory reserved per instance, with some overhead from cache invalidation as a result. This may be revised in future.

7.3.3 Mutual exclusion. Each one-to-one state machine can be used by a single client and a single server at a time. Mutual exclusion, combined with the implementation choice of a fixed size array of said state machines, means picking an index which is otherwise unused.

The additional invariant relative to subsection 7.1 is that a given outbox can now only be written to while the corresponding lock is held. That is sufficient to serialize operations on the individual state machines.

The lock acquire can be very cheap for systems where the process is comprised of N threads each of which can be dedicated to a single index. For example, if the array is as wide as the maximum number of warps on an NVPTX machine, compiler intrinsics can uniquely identify that warp, and use that identifier as an index. It is also cheap if the process contains a single thread, which may be the case for a CPU server implementation, or if a feature of the surrounding infrastructure for thread management provides an ID in [0, number-threads).

In other cases, a slot can be found dynamically using a bitmap of length equal to the maximum number of calls as an array of mutual exclusion locks. This lock array is local to the process so atomic compare and swap to set a bit at index I is taking a lock at I, which can be released by fetch_and with a mask. Provided locks or a priori knowledge ensures each one-to-one state machine is only in use by one pair of processes at a time, correctness of the whole system follows from correctness of a single pair. The concept of holding a lock on an index is useful for reasoning about optimisations, whether the lock is a bit set in a bitmap or implicit.

7.4 Algorithm adaption for process locking

Both processes proceed by selecting a state machine that they can make progress on, claiming the lock for it, and then checking whether there is still work to be done. Multiple client algorithm:

- find an index that is outbox clear, inbox clear
- acquire a lock on that index
- if it is no longer outbox clear, inbox clear, release lock and return
- proceed as in the one-to-one case
- release the lock

Multiple server algorithm:

- find an index that is outbox clear, inbox set
- acquire a lock on that index
- if it is no longer outbox clear, inbox set, release lock and return
- proceed as in the one-to-one-case
- release the lock

8 OPTIMISATIONS

8.1 Asynchronous call

Some function calls have no return value, e.g. for memory deallocation. The state machine described so far requires the client to detect that the call has succeeded and set the client outbox to 0, ultimately freeing up the slot for reuse. This can be relaxed, permitting the client to return immediately after posting work by setting the outbox to 1, provided some other client call can recognise the case and clean up.

The case of a previous asynchronous call is detectable when the outbox and inbox set, indicating a result has been received, however the corresponding lock is not set (or is implicit), so no client is waiting for it. The server code does not need to be changed. A call may be split into an asynchronous one that triggers some work and a later synchronous call that retrieves the result, or multiple asynchronous calls to query whether the result is available yet. This diverges from the simple RPC model of an invisible local call though, requiring application collaboration, so is not explored further in this paper.

8.2 Bit packing

The previous assumed a boolean is stored in the smallest integer that the process can write atomically. If the process can write with fetch_or, or atomic compare and swap, the mailbox entries can be packed into fewer machine words that are written atomically. Fetch_or is ideal but not provided as part of the base PCIe specification. Atomic compare and swap is usually susceptible to the ABA problem, but in this case the bit corresponding to the current slot can only be changed by the thread holding the corresponding lock. The compare and swap can never spuriously succeed as no other thread is trying to set the same value.

8.3 Batching outbox

The processes access to shared memory may be high latency and based on atomic compare and swap (CAS), e.g. across PCIe. The failure case is then expensive, where a given thread lost the race and must try again. For a 64 bit compare and swap, 64 outbox updates can be passed with a single successful compare. This can be done by maintaining a process local bitmap for the outbox which is updated with fetch_and/fetch_or to change the index currently locked. After updating the process local bitmap, enter a loop trying to update the shared memory outbox. The cases are then:

- CAS success, have written to the outbox, return
- CAS failed, indexed bit is different to the local outbox, try again
- CAS failed, indexed bit is the same as the local outbox, return

That amounts to each competing thread trying to update multiple values and returning as soon as it, or one of the other threads, succeeds in propagating the locked value.

8.4 Exceeding fixed buffer size

Shared memory RPC can handle larger arguments by allocating memory and passing a pointer. An alternative is a variant on the asynchronous call, where the client takes a lock and issues multiple call/return sequences before dropping the lock. The server can combine the buffers at that index. This is used in a printf implementation where the data passed can exceed any fixed size buffer but an allocation round trip introduces failure modes. Notably, because the lock is an integer index, it is also usable as a unique identifier during the call which help the server reassemble associated buffers. This remains opaque to the call site.

9 LIMITATIONS AND FUTURE WORK

9.1 Syntax

Development efforts have been focused on providing a correct and performant means of calling a function on N integers. Ease of use requires a layer on top of this to handle implicit serialization of types and passing function pointers.

9.2 Combinatorial testing

Verifying that the client and server process both compile on various architectures, as various languages, can be done in linear time on a single machine with a cross compiler. Verifying that each pair executes successfully requires further infrastructure, such as a unit test framework and thread pool definition that can execute on each architecture, to reach the point where a single client/server application can be compiled repeatedly and run under various different environments.

9.3 Further architectures

The implementation is presently tested on pre-Volta NVPTX, where the remote call is made on a per-warp basis. Extending to Volta means passing a thread mask down the call stack and allowing each thread in the warp to initiate the remote call, closer to the x86-64 model. Extending to a PowerPC host may uncover little endian assumptions. Intel or ARM GPUs will require additional implementations of some platform specific code.

9.4 Performance tuning

Different pairs of architecture and different communication fabrics benefit from different optimisations. For example, batching may be worthwhile across high latency PCIe and a net loss on a faster connection. Determining the optimal set of variations for different systems will follow from benchmarks that can run across various different systems, so follows on from subsection 9.3. This should all be variations in derived template parameters, unobservable to applications that are already using the unspecialized library.

10 CONCLUSION

Remote procedure calls are a simple interface if and only if there are no network induced failure modes. This is the case on a single shared memory GPU node. The synchronisation required can be implemented on shared memory systems with two atomic booleans per RPC state instance with single writer semantics and a fixed size shared buffer for argument and return passing.

A mechanism built on shared memory such as this, or waiting for vendor support, enables file I/O and similar from GPU code. Mmap of a file into shared memory from the GPU is a particularly good fit.

Once coupled with a code generation scheme such as [2] or similar, a state machine such as this provides a convenient means of executing functions on a different processor to the caller.

REFERENCES

[1] 2018. OpenMP Application Programming Interface Version 5.0. https://www.openmp.org/spec-html/5.0/openmp.html

[2] Aditya Agarwal, Mark Slee, and Marc Kwiatkowski. 2007. *Thrift: Scalable Cross-Language Services Implementation.* Technical Report. Facebook. http://thrift.apache.org/static/files/thrift-20070401.pdf

[3] Cheng Chen, Wenxiang Yang, Fang Wang, Dan Zhao, Yang Liu, Liang Deng, and Canqun Yang. 2019. Reverse Offload Programming on Heterogeneous Systems. *IEEE Access* 7 (2019), 10787–10797. https://doi.org/10.1109/ACCESS.2019.2891740

[4] Jon Chesterfield. 2020-2021. Hostrpc. https://github.com/JonChesterfield/hostrpc.

[5] Edsger W Dijkstra. 1962. Over de sequentialiteit van procesbeschrijvingen. *Trans. by Martien van der Burgt and Heather Lawrence. In* (1962), 124.

[6] Google. 2015. gRPC. https://developers.googleblog.com/2015/02/introducing-grpc-new-open-source-http2.html

[7] Gary L. Peterson and PETERSON GL. 1981. Myths about the mutual exclusion problem. (1981).

[8] Tyler Sorensen, Hugues Evrard, and Alastair F. Donaldson. 2018. GPU Schedulers: How Fair Is Fair Enough?.. In *CONCUR (LIPIcs, Vol. 118)*, Sven Schewe and Lijun Zhang (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 23:1–23:17. http://dblp.uni-trier.de/db/conf/concur/concur2018.html#SorensenED18

[9] A. Tanenbaum and R. V. Renesse. 1988. A Critique of the Remote Procedure Call Paradigm.

[10] Steve Vinoski. 2008. Convenience Over Correctness. *IEEE Internet Computing* 12, 4 (2008), 89–92. https://doi.org/10.1109/MIC.2008.75

[11] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. 1996. A note on distributed computing. In *International Workshop on Mobile Object Systems.* Springer, 49–64.