Logistics

- Hopefully, it will be RAMEN time next week
- Rob Yelle is setting up accounts on Cerberus
- All earlier lecture slides now on website
Parallelism in Shared Memory Machines

- Each processing element is constrained by what data of the application it can see natively
  - That is, what it can get to with referencing memory
  - Think of physically addressable shared memory
- Want to scale machines considerably
- Penalty for coordinating with other processing elements is now significantly higher
  - Approaches change accordingly
  - Scaling CC-NUMA shared memory is difficult
- Shift to physically distributed memory
Distributed Memory Multiprocessors

- Each processor has a local memory
  - Physically separated memory address space

- Processors must communicate to access non-local data
  - Message communication (message passing)
    - message passing architecture
  - Processor interconnection network

- Parallel applications must be partitioned across
  - Processors: execution units
  - Memory: data partitioning

- Scalable architecture
  - Small incremental cost to add hardware (cost of node)
Scalable, High-Performance Interconnect

- Interconnection network is core of parallel architecture
- Requirements and tradeoffs at many levels
  - Elegant mathematical structure
  - Deep relationship to algorithm structure
  - Hardware design sophistication
- Little consensus
  - Performance metrics?
  - Cost metrics?
  - Workload?
  - …
Network Performance Measures

Overhead: latency of interface vs. Latency: network
Interconnection Network

- Many large-scale clusters are defined by interconnection network technology and topology

Technologies

- Gigabit Ethernet (GigE) ([https://en.wikipedia.org/wiki/Gigabit_Ethernet](https://en.wikipedia.org/wiki/Gigabit_Ethernet))
  - 1 GigE, 10 GigE, 100 GigE
  - dominant high-end networking technology
  - high throughput and very low latency
  - new interconnect family
  - high throughput and very low latency

Network technologies provide advanced capabilities

- Available on RoCE (RDMA over Converged Ethernet) and IB
Topologies

- Lots of innovation in topologies in the past
- Performance of modern technologies is high enough that topology varieties are fewer now:
  - Torus
    - Generally 3 to 6 dimensions
  - Fat tree
    - branches near bottom of tree are “skinnier”
    - branches nearer the top of the hierarchy are “fatter”

- Node connection degree depends on topology and capacity
  - # “rails” = # connections from a node
  - more simultaneous communications

- Concerned about performance
  - Throughput, (bi-section) BW, latency
Performance Metrics: Latency and Bandwidth

- **Bandwidth**
  - Need high bandwidth in communication
  - Match limits in network, memory, and processor
  - Network interface speed vs. network bisection bandwidth

- **Latency**
  - Performance affected since processor may have to wait
  - Harder to overlap communication and computation
  - Overhead to communicate is a problem in many machines

- **Latency hiding**
  - Increases programming system burden
  - Examples: communication/computation overlaps, prefetch
Advantages of Shared Memory Architectures

- Compatibility with SMP hardware
- Ease of programming when communication patterns are complex or vary dynamically during execution
- Ability to develop applications using familiar SMP model, attention only on performance critical accesses
- Lower communication overhead, better use of BW for small items, due to implicit communication and memory mapping to implement protection in hardware, rather than through I/O system
- HW-controlled memory hierarch to reduce remote communication by caching all shared/private data
Advantages of Distributed Memory Architectures

- The hardware can be simpler (especially versus NUMA) and is more scalable
- Communication is explicit and simpler to understand
- Explicit communication focuses attention on costly aspect of parallel computation
- Synchronization is naturally associated with sending messages, reducing the possibility for errors introduced by incorrect synchronization
- Easier to use sender-initiated communication, which may have some advantages in performance
Types of Parallel Computing Models

- **Data parallel**
  - Simultaneous execution on multiple data items
  - Example: Single Instruction, Multiple Data (SIMD)

- **Task parallel**
  - Different instructions on different data (MIMD)

- **SPMD (Single Program, Multiple Data)**
  - Combination of data parallel and task parallel
  - Not synchronized at individual operation level

- **Message passing is for MIMD/SPMD parallelism**
  - Can be used for data parallel programming
Message Passing Model

- Hardware maintains send and receive message buffers

- Send message (synchronous)
  - Build message in local message send buffer
  - Specify receive location (processor id)
  - Initiate send and wait for receive acknowledge

- Receive message (synchronous)
  - Allocate local message receive buffer
  - Receive message byte stream into buffer
  - Verify message (e.g., checksum) and send acknowledge

- Memory to memory copy with acknowledgement and pairwise synchronization
Message-Passing Model

- A process is a program counter and address space
- Processes can have multiple threads (program counters and associated stacks) sharing a single address space

- MPI is for communication among processes (ranks)
  - More recent MPI versions allow threads to call MPI routines
- Interprocess communication consists of
  - Synchronization
  - Data movement
SPMD

- Data distributed across processes
  - Not shared

“Owner compute” rule: Process that “owns” the data (local data) performs computations on that data.
Message Passing Programming

- Defined by communication requirements
  - Data communication (necessary for algorithm)
  - Control communication (necessary for dependencies)

- Program behavior determined by communication patterns

- Message passing infrastructure attempts to support the forms of communication most often used or desired
  - Basic forms provide functional access
    - can be used most often
  - Complex forms provide higher-level abstractions
    - serve as basis for extension
    - example: graph libraries, meshing libraries, …
  - Extensions for greater programming power
Communication Types

- Two ideas for communication
  - Cooperative operations
  - One-sided operations
Cooperative Operations for Communication

- Data is cooperatively exchanged in message-passing
- Explicitly sent by one process and received by another
- Advantage of local control of memory
  - Any change in the receiving process’s memory is made with the receiver’s explicit participation
- Communication and synchronization are combined

\[
\text{Process 0} \quad \text{Send(data)} \quad \text{Process 1} \quad \text{Receive(data)}
\]
One-Sided Operations for Communication

- One-sided operations between processes
  - Include remote memory reads and writes
- Only one process needs to explicitly participate
  - There is still agreement implicit in the SPMD program

Advantages?
- Communication and synchronization are decoupled

\[ \text{Process 0} \quad \begin{array}{r} \text{Put(data)} \\ (\text{memory}) \end{array} \quad \text{Process 1} \quad \begin{array}{r} (\text{memory}) \\ \text{Get(data)} \end{array} \]
Pairwise vs. Collective Communication

- Communication between process pairs
  - Send/Receive or Put/Get
  - Synchronous or asynchronous (we’ll talk about this later)

- Collective communication between multiple processes
  - Process group (collective)
    - several processes logically grouped together
  - Communication within group
  - Collective operations
    - communication patterns
      - broadcast, multicast, subset, scatter/gather, …
    - reduction operations
What is MPI (Message Passing Interface)?

- Message-passing library (interface) specification
  - Extended message-passing model
  - Not a language or compiler specification
  - Not a specific implementation or product
- Targeted for parallel computers, clusters, and NOWs
  - NOWs = network of workstations
- Specified in C, C++, Fortran 77, F90
- Full-featured and robust
- Designed to access advanced parallel hardware
- End users, library writers, tool developers
- Message Passing Interface (MPI) Forum

http://www.mpi-forum.org/
Why Use MPI?

- Message passing is a mature parallel programming model
  - Well understood
  - Efficient match to hardware (interconnection networks)
  - Many applications
- MPI provides a powerful, efficient, and portable way to express parallel programs
- MPI was explicitly designed to enable libraries …
- … which may eliminate the need for many users to learn (much of) MPI
- Need standard, rich, and robust implementation
- Three versions: MPI-1, MPI-2, MPI-3 (new features!)
  - Robust implementations including free MPICH (ANL)
Features of MPI

- **General**
  - Communicators combine context and group for security
  - Thread safety (implementation dependent)

- **Point-to-point communication**
  - Structured buffers and derived datatypes, heterogeneity
  - Modes: normal, synchronous, ready, buffered

- **Collective**
  - Both built-in and user-defined collective operations
  - Large number of data movement routines
  - Subgroups defined directly or by topology
Features of MPI (continued)

- Application-oriented process topologies
  - Built-in support for grids and graphs (based on groups)

- Profiling
  - Hooks allow users to intercept MPI calls
  - Interposition library interface (PMPI)
  - Many tools (e.g., TAU) use PMPI

- Environmental
  - Inquiry
  - Error control
Is MPI Large or Small?

- MPI is large
  - MPI-1 is 128 functions, MPI-2 is 152 functions
  - Extensive functionality requires many functions
  - Not necessarily a measure of complexity

- MPI is small (6 functions)
  - Many parallel programs use just 6 basic functions

- “MPI is just right,” said Baby Bear Goldilocks
  - One can access flexibility when it is required
  - One need not master all parts of MPI to use it
To Use or Not Use MPI?

- **USE**
  - You need a portable parallel program
  - You are writing a parallel library
  - You have irregular or dynamic data relationships that do not fit a data parallel model
  - You care about performance

- **NOT USE**
  - You don’t need parallelism at all (Ha!)
  - You can use libraries (which may be written in MPI)
  - You use multi-threading on a shared memory machine
MPI Implementations

- **MPICH** ([https://www.mpich.org](https://www.mpich.org))
  - High-quality reference implementation of MPI standard
  - Broad network support, including IB and proprietary
  - Originated out of Argonne National Lab

- **OpenMPI** ([https://www.open-mpi.org](https://www.open-mpi.org))
  - Targets the common case
  - Consortium of academic, research, industry

- **MVAPICH** ([http://mvapich.cse.ohio-state.edu](http://mvapich.cse.ohio-state.edu))
  - Very high-performance MPI implementation
  - Supports very latest networks and functionality
MVAPICH

- Open source implementation of MPI 3.1 standard
- Delivers the best performance, scalability and fault tolerance
  - 2,875 groups in 86 countries with 465,000 downloads
- High-performance parallel programming libraries
  - MVAPICH2: InfiniBand, Omni-Path, Ethernet, RoCE
  - MVAPICH2-X: advanced MPI features, PGAS (OpenSHMEM, UPC, UPC++, CAF), MPI+PGAS with unified communication
  - MVAPICH2-GDR: MPI for clusters w/ NVIDIA GPUs
  - MVAPICH2-Virt: MPI for hypervisor / container HPC clouds
  - MVAPICH2-EA: energy aware and High-performance MPI
  - MVAPICH2-MIC: optimized MPI for clusters with Intel KNC
Communication Delay and Constraint

- Consider the speedup equation:
  \[ S(n,p) = \frac{T_{seq}(n)}{T_{par}(n,p) + \text{comm}(n,p)} \]

- If we consider the speedup equation, communications are to be limited to increase the scalability.

- If we now consider that moving \( x \) Mbits of data on a network with a bandwidth \( b \) takes (at least)
  \[ t_{tr} = \frac{x}{b} \]

  we see that a given problem requiring data-movements is bound to be limited by communication overhead.

  \[ S(n, p) \leq \frac{T_{seq}(n)}{\text{comm}(n, p)} \]
Communication Overlap

- The overlap is then a way to (partially) hide the communication cost from the application, delegating it to a third-party (often an RDMA-capable host communications adaptor (HCA))

- It also has positive advantages in terms of network noise reduction as it partially decouples the code from data-movements on the machine
Networks usually have a latency distribution with a “long tail” (10^9 Infiniband roundtrips on Tera 100)
Asynchronous Progress in MPI (1/3)

- MPI performance of various MPI messaging scenarios with focus on asynchronous progress
- Consider the following scenario:
  - Equivalent to a blocking receive

IW: Irecv — Wait
Asynchronous Progress in MPI (2/3)

- Try to overlap with computation
- Consider the following scenario:
  - Most common form of overlapping

ICW: Irecv — Compute — Wait
Asynchronous Progress in MPI (3/3)

- Overlap with computation and test
- Consider the following scenario:
  - Part of the resources involve the MPI runtime

**ITCW: Irecv — Compute / Test — Wait**
Asynchronism in Production MPIS (1/2)

Waiting time in seconds (log)

<table>
<thead>
<tr>
<th>Message size</th>
<th>OpenMPI 2.1.0 (SHM)</th>
<th>MPICH 3.3a2 (SHM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.5367e-07</td>
<td>IW</td>
<td>ITCW</td>
</tr>
<tr>
<td>3.0517e-05</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.000976562</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.03125</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Message size
Asynchronism in Production MPIs (2/2)

Waiting time in seconds (log)

<table>
<thead>
<tr>
<th>Message size</th>
<th>1024</th>
<th>32</th>
<th>1</th>
<th>0.03125</th>
<th>0.000976562</th>
<th>3.05176e-05</th>
<th>9.53674e-07</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenMPI 1.8.8 (IB)</td>
<td>ICW</td>
<td>ITCW</td>
<td>IW</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel MPI 17 (IB)</td>
<td>ICW</td>
<td>ITCW</td>
<td>IW</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MPI and Messages

- As seen on these production MPI (using default configuration) overlap is not to be taken for granted in MPI even when using “asynchronous” calls
- It is important to progress these calls
- Reasons for this are multiple (non exhaustively):
  - Optimization for latency (noise from progress)
  - No standard support for thread-multiple
    (multiple entities progressing the same requests)
  - Need for remote information when progressing messages
    (aka matching)
**MPI Matching**

- Before copying any data, MPI has to make sure that the source and target buffer are correctly identified.

- It makes “single copy” (aka “zero-copy” or “zero-recopy”) messages more difficult to implement.

- In practice, messages involve some recopy overhead:
  - Latency optimized eager (recopied on target)
  - Rendez-vous protocol requires a previous message to initiate the synchronization.
Eager Messages

- There is a recopy on the target and therefore MPI needs "polling:" to unpack the eager buffers and complete the message.
Rendezvous Read Example

- Payload movement is “zero-copy” but pinning and meta-data exchange to setup RDV needs polling.
Rendezvous Write Example

- Payload movement is “zero-copy” but pinning and meta-data exchange to setup RDV needs polling.

I would like to send you a large message, tell me when RDY

Wait for MatchingRecv

PinRecvMemory

I’m ready

Send

Init

PinSendMemory

RDY

RDMA Write

Done

Notify

Matching

RDY

Recv

I’m done
MPI Matching and Overlap

- We have seen that matching, by requiring information only available on the target process prevents modern HPC network cards to fully express their potential.
- A software mechanism has to be involved to associate two MPI buffers and therefore “meta-data” have to be exchanged prior to moving the actual message data.
- This is (one of) the reasons why MPI needs to poll its message queues to enable efficient progress.
- A possible solution to this is to implement MPI matching in the hardware.
  - Look at Portals 4 message layer.
Portals 4 and Hardware MPI

- Portals 4.1 is a low-level network interface for HPC, developed by Sandia National Laboratories since 2017.

- It provides enhanced semantics when compared to low-level Verbs (HW Matching) and features enabling the implementation of PGAS languages (such as Put notification).
The Bull Exascale Interconnect (BXI)

- Bull Exascale Interconnect (BXI) is the first implementation of Portals 4 communication model
- BXI is undergoing final validation tests in the Tera 1000 machine which gathers thousands of BXI enabled nodes

BXI card

BXI switch
Portals 4 in the MPC Runtime

- The Portals 4 communication model with the help of the BXI has been integrated to the MPC runtime.
- To do so, MPC relied on a mapping of communicators to the Portal 4 table (RR if number of comms exceed table size):

```
<table>
<thead>
<tr>
<th>EG</th>
<th>0</th>
<th>RECOVERY</th>
</tr>
</thead>
<tbody>
<tr>
<td>EG</td>
<td>1</td>
<td>CM</td>
</tr>
<tr>
<td>EG</td>
<td>2</td>
<td>RDMA</td>
</tr>
<tr>
<td>EG</td>
<td>3</td>
<td>COMM_WORLD</td>
</tr>
<tr>
<td>EG</td>
<td>4</td>
<td>COMM_SELF</td>
</tr>
<tr>
<td>EG</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
```

== ME in overflow list
Portals 4 MatchBit in MPC

- 64 bits match-bit is handled in MPC:
  - No need for communicator as it is mapped to the table entry (limits communicator number to table size)
  - 32 bits for the tag
  - 16 bits for the source rank
  - 16 bits for the sequence number

- Allows a full offload of matching in HCA matching a Send and aRecv without requiring CPU arbitration
Hardware Matching

- Portals 4 allows matching to take place in hardware
- Possible to route a message end-to-end asynchronously
- Target message address is resolved in the hardware, allowing the put event to directly target the user-space MPI buffer
- All steps are done in hardware

Diagram:

- ME (Message Event)
- REC V BUFFER
- Send Buffer
- Put_Event
- ACK_EVENT
- MD (Message Destination)

Actions:
- Append(PRIORITY_LIST)
- Zero-(re)copy
- Pt1Put()
Hardware Matching Late Receive

- For lateRecv, Portals ME has to be recopied by the CPU to create an event for the upcomingRecv.
- If earlyRecv can be offloaded, lateRecv needs special handling (here in terms of Network card MEressource) and fails at being fully offloaded.
Search for Asynchronism

- MPI does provide asynchronism when used correctly
- However, we discussed how the matching semantic still prevent a full communication offload
- MPI has primitives such as one-sided providing end-users with tools leveraging high-end network card capabilities, and avoiding synchronization
- Still most programs end-up being relatively synchronous with time-steps
- In the mean time we start hearing about …
  user-defined software stack, indigenous hardware, converging NIC, active storage, constellation of services, computer on switch, code steering, MPI sessions, Kubernetes, IO delegation, containers, burst buffers, in situ
**Convergence between Cloud and HPC (1/2)**

**Kubernetes** is an open-source system for automating deployment, scaling, and management of containerized applications.

It groups containers that make up an application into logical units for easy management and discovery. Kubernetes builds upon **15 years of experience of running production workloads at Google**, combined with best-of-breed ideas and practices from the community.

**Planet Scale**

Designed on the same principles that allows Google to run billions of containers a week, Kubernetes can scale without increasing your ops team.
With Kubernetes you can do …
  - Spawn replicas of your services and manage them
  - Compose your services using load balancing techniques
  - Expose high-availability services thanks to a TCP-level abstraction

Everything still runs on TCP ...

MPI is the standard for HPC messaging
  - Abstractions such as message queues and RPC engines
  - Becoming more current and exhibit interesting properties
  - Thinking more “world scalable”

REST and RPCs are a standard way to expose and compose scalable and resilient services
Besnard’s Accelerated RPC (ARPC) for HPC

- Consider the following goals:
  - Replace TCP for all “side” services
  - Preserve overall performance
  - Take advantage of modern HCAs
  - Enable cross-job, intra-job, and machine-wide interactions to progressively replace services
  - Use an abstraction enable the full use of high-speed networking

- With these goals eventually enabling:
  - Parallel jobs as a combination of differentiated services
  - Eventual removal of file-system dependencies
  - Parallel codes that are much lighter weight
ARPC Principles (1/3)

- RPCs are here since the dawn of computer-science
- They are a de-facto standard
  - Classical pattern for most developers
- They power most of the WEB right-now
- Are they a reasonable alternative to MPI?

- Running an RPC is simply calling a remote function with local parameters and getting back the result
- This can happen indifferently over the network or locally
ARPC Principles (2/3)

- In the code it is as simple as writing:
  
  ```
  struct question A;
  struct response B;
  /*** Start the RPC server ***/
  B = rpc.F(A);
  ```

- RPCs are not new in HPC
  - Most PGAS rely on such features internally
ARPC Principles (3/3)

- ARPC appears to have advantages over MPI:
  - No requirement for matching
  - No expectation on remote state
    - Processing comes with the data
  - Overall better use of HCA’s capabilities (see later)
  - No need for serialization (see later)
  - Convenient way to mix tasks and movement of data

- Architecture comparison
  - Contrast “linking” versus “addressing”

Current « Linking »

ARPC Constellation of Services
ARPC in Practice (1/3)

- Besnard ported the GRPC interface on top of Portals 4
- GRPC is an interface designed by Google
  - Relies on the Protobuf serialization protocol
  - Defines an RPC and parameters (protobuf objects) programmatically
- Define a key-value data-store using GRPC protobuf

```protobuf
syntax = "proto3";
package simulation;
message value_t { int32 x = 1; int32 y = 2; int32 weight = 3; }
message key_t { int32 id = 1; }
message text_t { string txt = 1; }
message size { int64 val = 1; }
message Empty {}

service SRV_HT {
  rpc get(key_t) returns (value_t) {}  
  rpc add(value_t) returns (key_t) {}  
  rpc del(key_t) returns (Empty) {}  
  rpc list(Empty) returns(text_t) {}  
  rpc get_size(Empty) returns(size) {}
}
```
ARPC in Practice (2/3)

- Complete implementation of our key-value data-store service in C++ using std::map

- Present here a slightly more compact definition than GRPC services, but such definitions are also compatible

- Objects generated by GRPC are native C++ objects which can be serialized

- Code for registering a new service:

```cpp
using namespace simulation;
using namespace ARPC;

class myReceiver : public SRV_HT::Recv {
private:
    std::map<int, simulation::value_t> datastore;
    size_t nb_keys;
public:
    myReceiver() : Recv(), nb_keys(0) {}  
    int get(Context* ctx, simulation::key_t* k, simulation::value_t* v) {
        if (k->id() < nb_keys) {
            *v = datastore.at(k->id());
        }
        return 0;
    }
    int add(Context* ctx, simulation::value_t* v, simulation::key_t* k) {
        simulation::key_t new_k;
        new_k.set_id(nb_keys++);
        datastore.insert(std::pair<int, simulation::value_t>(new_k.id(), *v));
        *k = new_k;
        return 0;
    }
    int del(Context* ctx, simulation::key_t* k, void* n) {
        if (k->id() < nb_keys) {
            datastore.erase(k->id());
        }
        return 0;
    }
    int list(Context* ctx, void* n, simulation::text_t* out) {
        std::stringstream s;
        for (std::map<int, simulation::value_t>::const_iterator it = datastore.cbegin();
            it != datastore.cend(); it++) {
            s << it->first << " -> (" << it->second.x() << ", y: " << it->second.y() << ", w: " << it->second.weight() << "); "
                << std::endl;
        }
        out->set_txt(s.str());
        return 0;
    }
    int get_size(Context* ctx, void* n, simulation::size* size) {
        size->set_val((int64_t)datastore.size());
    }
};
```

```cpp
ServicePool pool;
myReceiver recv;
ctx.cxx_pool = &pool;
pool.add(&recv);
```
ARPC in Practice (3/3)

- This is the complete client implementation
- Starting a client for the “SRV_HT”
- Sending a first value object
- Remote are addressed by “rank”
  - Working on an URI-based approach using a mechanisms to both register and resolve ressources.

```cpp
// declarations
ARPC::SRV_HT::Send s;
ARPC::Context ctx;
simulation::value_t v;
simulation::key_t k;

//consider the current process as 0 and the remote as 1
ctx.dest = 1;
v.set_x(i);
v.set_y(i*10);
v.set_weight(i*100);

// do the actual RPC as if it was local code
s.add(&ctx, &v, &k);
std::cout <<"Value inserted with key"<< k.id() << std::endl;
```
An accelerated RPC on Portals 4 works as follows:

1. Book and register a ME in response segment (fixed-sized blocks) with a unique tag.
2. Put the data to the target persistent ME (managed-local ME, no pinning as the whole memory is registered at application start).
3. When polled remotely the target RPC is called and the response sent back with the tag.
4. When polled the client RPC is now completed.

- This is a complete zero-copy chain.
- CPU only polls for valuable events.
Note that the pattern presented here for the request may happen for the response when it is larger than the response block.
**ARPC in Portals 4 (match-bit)**

- ARPC over Portals are done without sending any meta-data (apart Portals 4 ones) over the network.
- The card is then able to transparently provide all the necessary information to trigger the RPC.
- RPC are triggered when a “Put notification” is polled — this is a particularity of Portals 4.

<table>
<thead>
<tr>
<th>Service ID</th>
<th>RPC ID</th>
<th>TAG</th>
<th>Kind</th>
<th>Free</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 bits</td>
<td>16 bits</td>
<td>32 bits</td>
<td>4 bits</td>
<td>4 bits</td>
</tr>
</tbody>
</table>

- The match-bit is as follows:
  - 8 bits for the service ID (256 services in a given process)
  - 16 bits for the RPC ID (65 536 RPCs per service)
  - 32 bits tag to match a response to a request in hardware
  - 4 bits to distinguish message kind (Request, Response, Large)
  - 4 bits free
**ARPC Advantages**

- By coupling data-movements to processing RPCs solve some issues (with MPI for example)
- Main advantages
  - No expectation on remote state and improved load-balance
  - Services have dedicated resources (unlike stacked runtimes / shared-libs)
  - Network calls are regular function calls in the code
  - Client requests can be managed as tasks in a parallel task engine
  - Local RPC calls can be directly inserted in the task engine

- Interesting scenario of network-based dispatch of services by defining an RPC sending a service (as binary code) to be launched remotely
  - Reduces the pressure on IO and “spreads” services on the allocation
ARPC Lacks

- GRPC is providing HTTP2 transport layer with potential support for SSL/TLS
  - ARPC implementation does not provide such security abstraction
- ARPC is not at the level of GRPC in terms of general robustness (Virtual TCP networks) and security (TLS auth, Encryption)
  - Convinced that the hardware could eventually provide more efficient alternatives than Layer3+ approaches
  - Believe that it is usable in network-cores and HPC
- Lower endpoint level security (compared to internet standards), but should be ok in HPC systems