How Can MPI Fit Into Today's Big Computing

Jonathan Dursi
Senior Research Associate
Centre for Computational Medicine
The Hospital for Sick Children
https://github.com/ljdursi/EuroMPI2016

Healthier Children. A Better World.

SickKids RESEARCH INSTITUTE

Who Am I?

Old HPC Hand...

Ex-astrophysicist turned large-scale computing.

- Large-scale high-speed adaptive reactive fluid fluids
- DOE ASCI Center at Chicago
  - ASCI Red
  - ASCI Blue
  - ASCI White
- FORTRAN, MPI, Oct-tree regular adaptive mesh

Who Am I?

Old HPC Hand...

Ex-astrophysicist turned large-scale computing.

- Large-scale high-speed adaptive reactive fluid fluids
- DOE ASCI Center at Chicago
    - ASCI Red
    - ASCI Blue
    - ASCI White
- FORTRAN, MPI, Oct-tree regular adaptive mesh
- Joined HPC centre after postdoc
- Worked with researchers on wide variety of problems
- Got interested in genomics:
    - Large computing needs
    - Very interesting algorithmic challenges

Who Am I?

Old HPC Hand...

Gone Into Genomics

Started looking into Genomics in ~2013, made move in 2014

- Ontario Institute for Cancer Research
- Working with Jared Simpson, author of ABySS (amongst other things)
    - First open-source human-scale de novo genome assembler
    - MPI-based

Who Am I?

Old HPC Hand...

Gone Into Genomics

Started looking into Genomics in ~2013, made move in 2014

- Ontario Institute for Cancer Research
- Working with Jared Simpson, author of ABySS (amongst other things)
    - First open-source human-scale de novo genome assembler
    - MPI-based
- ABySS 2.0 just came out, with a new non-MPI mode

Who Am I?

Old HPC Hand...

Gone Into Genomics

In the meantime, one of the de-facto standards for genome analysis, GATK, has just announced that version 4 will support distributed cluster computing — using Apache Spark.

MPI's Place In Big Computing Is Small, and Getting Smaller

MPI is For...

1990s	All multi-node cluster computing
2000s	All multi-node **scientific** computing
2006ish	All multi-node scientific **simulation**
2010s	All multi-node **physical sciences** simulation
2016	**Much** multi-node physical sciences simulation

MPI's Place In Big Computing Is Small, and Getting Smaller

but it doesn't have to be this way

Outline

- A tour of some common big-data computing problems
  - Genomics and otherwise
  - Not so different from complex simulations
- A tour of programming models to tackle them
  - Spark
  - Dask
  - Distributed TensorFlow
  - Chapel
- A tour of data layers the models are built on
  - Akka
  - GASNET
  - TCP/UDP/Ethernet: Data Plane Development Kit
  - Libfabric/UCX
- Where MPI is: programming model at the data layer
- Where MPI can be

The Problems Big Data Frameworks Aim To Solve

Big Data Problems

Big Data problems same as HPC, if in different context

- Large scale network problems
  - Graph operations
- Similarity computations, clustering, optimization,
  - Linear algebra
  - Tree methods
- Time series
  - FFTs, smoothing, ...

Main difference: hit highly irregular, unstructured, dyanamic problems earlier in data analysis than simulation (but Exascale...)

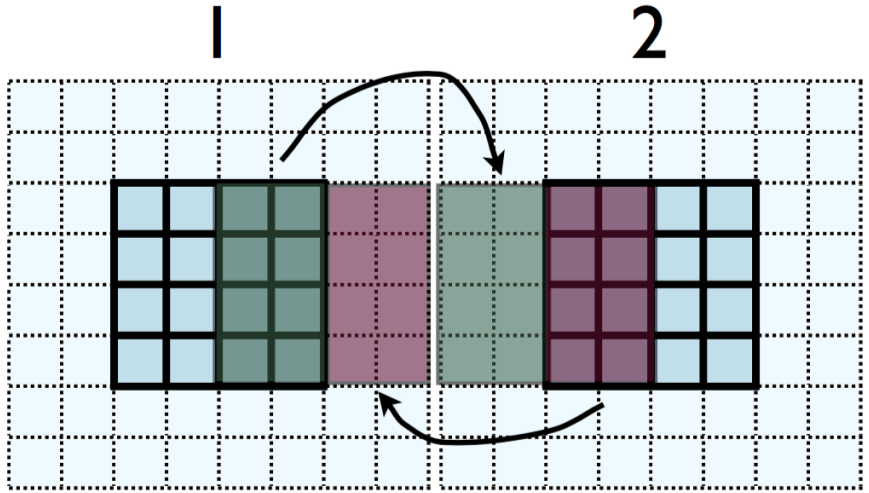
- Irregular data lookups:
  - Key-Value Stores

Big Data Problems

Key-value stores

In simulation, you normally have the luxury of knowing exactly where needed data is, because you put it there.

Maintained with global state, distributed state, or implicitly (structured mesh).



Big Data Problems

Key-value stores

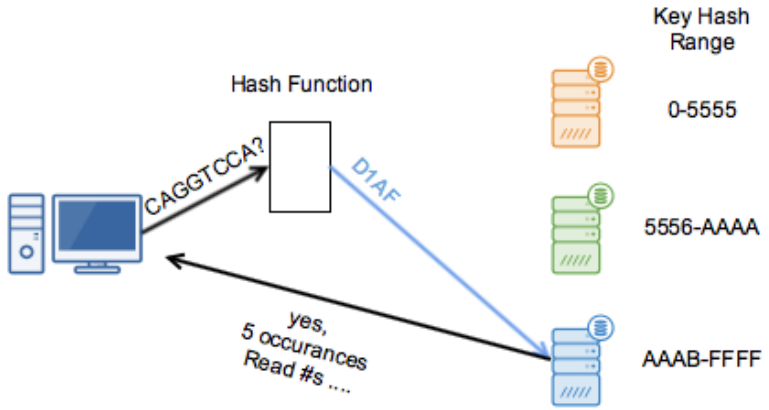
With faults, or rapid dynamism, or too-large scale to keep all state nearby, less true.

Data analysis: much of genomics involves lookups in indices too large for one node's memory

- Good/simple/robust distributed hash tables within an application would be extremely valuable.

Begins to look a lot like distributed key-value stores in big data.

Highly irregular access (don't know where you're sending to, receiving from, when requests are coming, or how many messages you will receive/send) is not MPI's strong suit.



Big Data Problems

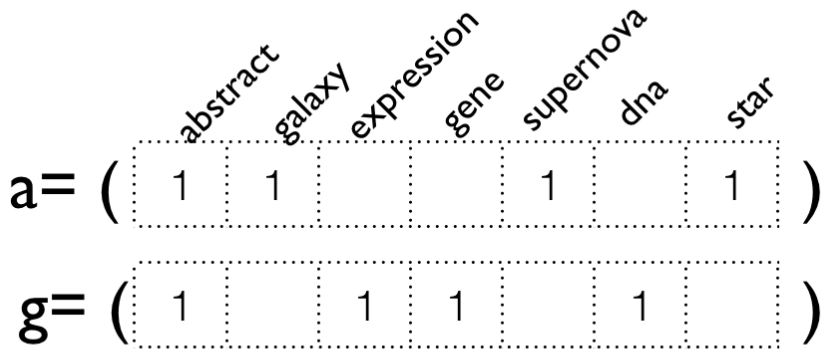
Key-value stores

Linear algebra

Almost any sort of numeric computation requires linear algebra.

In many big-data applications, the linear algebra is *extremely* sparse and unstructured; say doing similarity calculations of documents, using a bag-of-words model.

If looking at ngrams, cardinality can be enormous, no real pattern to sparsity



$$S_{a,g} = \frac{\mathbf{w_a} \cdot \mathbf{w_g}}{||w_a|| \cdot ||w_g||}$$
$$= \frac{1}{2 \cdot 2}$$
$$= \frac{1}{4}$$

Big Data Problems

Key-value stores

Linear algebra

Graph problems

As with other problems - big data graphs are like HPC graphs, but more so.

Very sparse, very irregular: nodes can have enormously varying degrees, *e.g.* social graphs

Big Data Problems

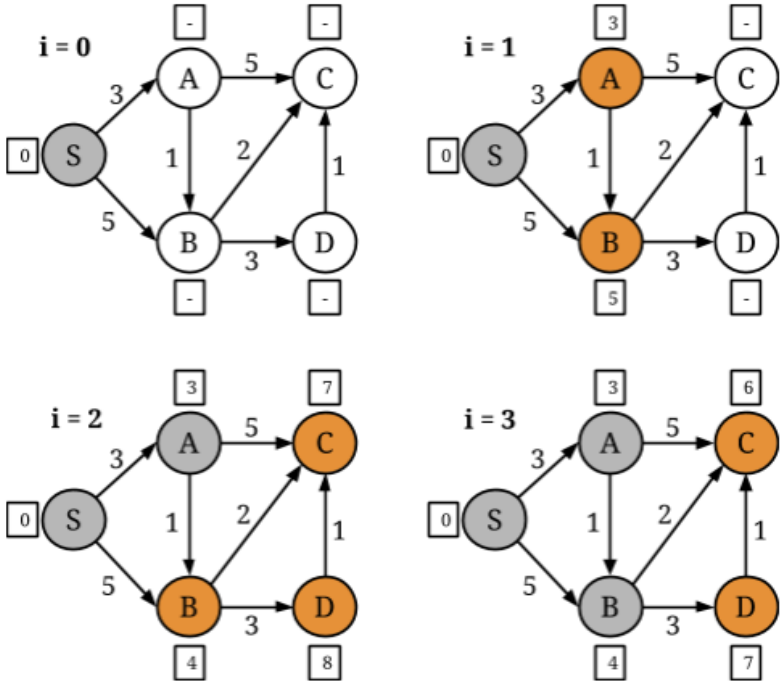
Key-value stores

Linear algebra

Graph problems

Generally decomposed in similar ways.

Processing looks very much like neighbour exchange on an unstructured mesh; can map unstructured mesh computations onto (very regular) graph problems.



https://flink.apache.org/news/2015/08/24/introducing-flink-gelly.html

Big Data Problems

Key-value stores

Linear algebra

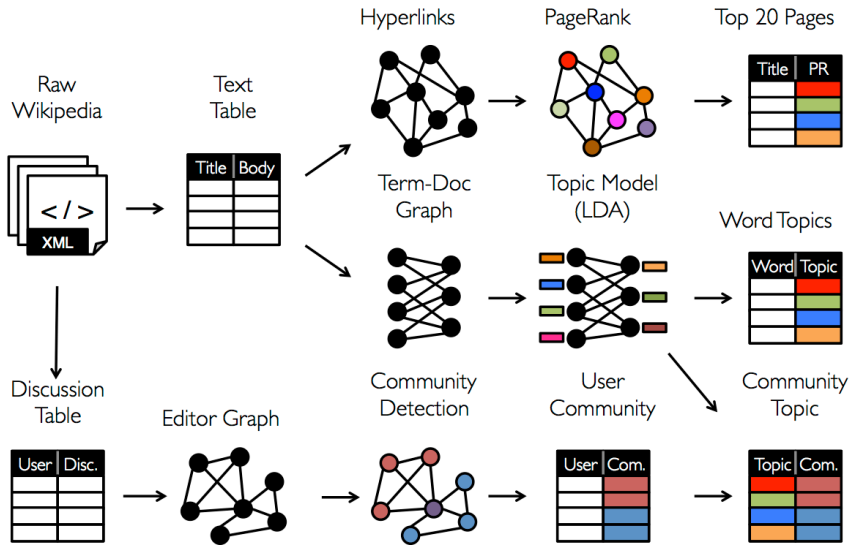
Graph problems

Calculations on (*e.g.*) social graphs are typically very low-compute intensity:

- Sum
- Min/Max/Mean

So that big-data graph computations are often *more* latency sensitive than more compute-intensive technical computations

⇒ lots of work done and in progresss to reduce communication/framework overhead



https://spark.apache.org/docs/1.2.1/graphx-programming-guide.html

Big Data Problems

Key-value stores

Linear algebra

Graph problems

Commonalities

The problems big-data practitioners face are either:

- The same as in HPC
- The same as HPCish large-scale scientific data analysis
- Or what data analysis/HPC will be facing towards exascale
  - Less regular/structured
  - More dynamic

So it's worth examining the variety of tools that community has built,

- To see if they can benefit our community
- To see what makes them tick
- To see if we can improve them

Big Data Programming Models

Spark: http://spark.apache.com

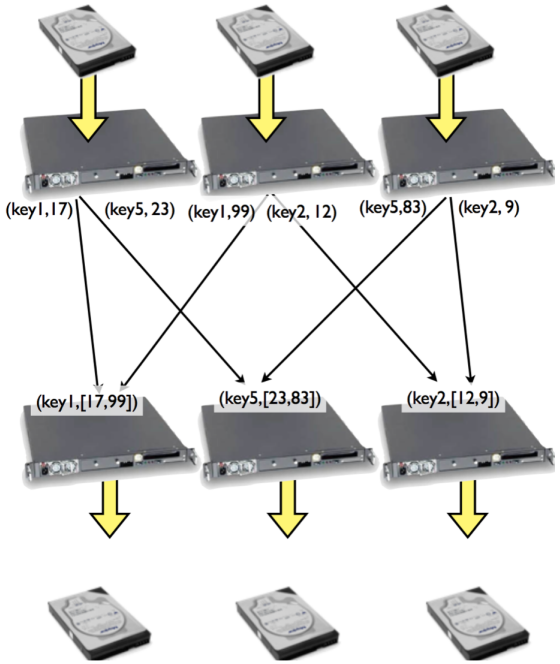
Spark

Overview

Hadoop came out in ~2006 with MapReduce as a computational engine, which wasn't that useful for scientific computation.

- One pass through data
- Going back to disk every iteration

However, the ecosystem flourished, particularly around the Hadoop file system (HDFS) and new databases and processing packages that grew up around it.



Spark

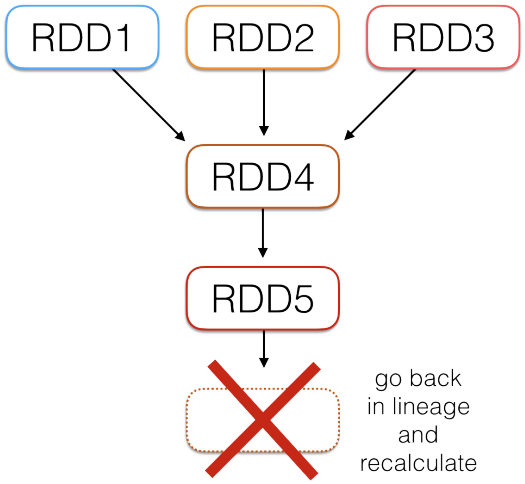
Overview

Spark (2012) is in some ways "post-Hadoop"; it can happily interact with the Hadoop stack but doesn't require it.

Built around concept of in-memry resilient distributed datasets

- Tables of rows, distributed across the job, normally in-memory
- Immutable
- Restricted to certain transformations

Used for database, machine learning (linear algebra, graph, tree methods), *etc.*



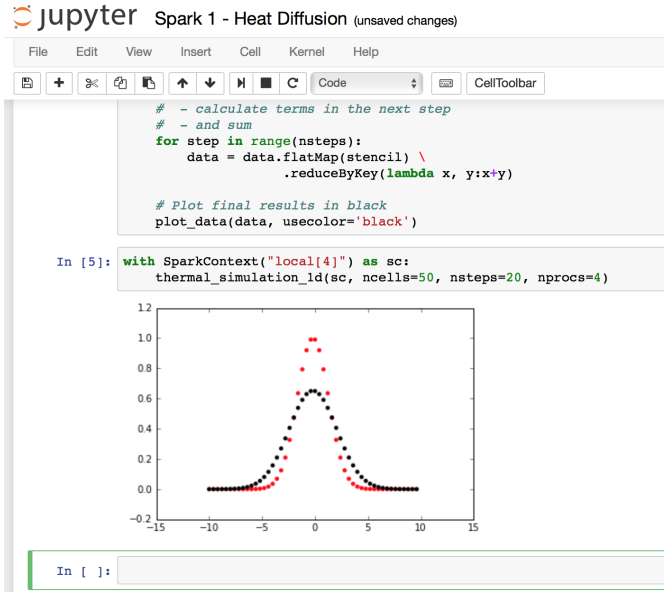
Spark RDDs prove to be a very powerful abstraction.

Key-Value RDDs are a special case - a pair of values, first is key, second is value associated with.

Linda tuple spaces, which underly Gaussian.

Can easily use join, *etc.* to bring all values associated with a key together:

- Like all stencil terms that are contribute at a particular grid point



Operations on Spark RDDs can be:

- Transformations, like map, filter, join...
- Actions like collect, foreach, ..

You build a Spark computation by chaining together transformations; but no data starts moving until part of the computation is materialized with an action.

Delayed computation + view of entire algorithm allows optimizations over the entire computation graph.

So for instance here, nothing starts happening in earnest until the `plot_data()` (Spark notebook 1)

```
    # Main loop: For each iteration,
    #  - calculate terms in the next step
    #  - and sum
    for step in range(nsteps):
        data = data.flatMap(stencil) \
                   .reduceByKey(lambda x, y:x+y)

    # Plot final results in black
    plot_data(data, usecolor='black')
```

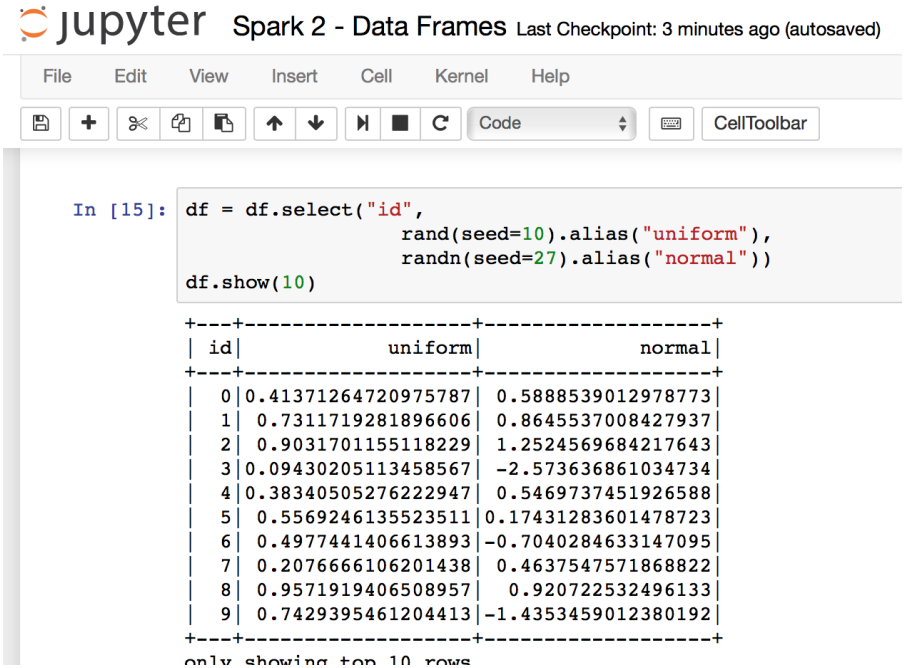
Knowledge of lineage of every shard of data also means recomputation is straightforward in case of node failure

But RDDs are also building blocks.

Spark Dataframes are lists of columns, like pandas or R data frames.

Can use SQL-like queries to perform calculations. But this allows bringing the entire mature machinery of SQL query optimizers to bear, allowing further automated optimization of data movement, and computation.

(Spark Notebook 2)



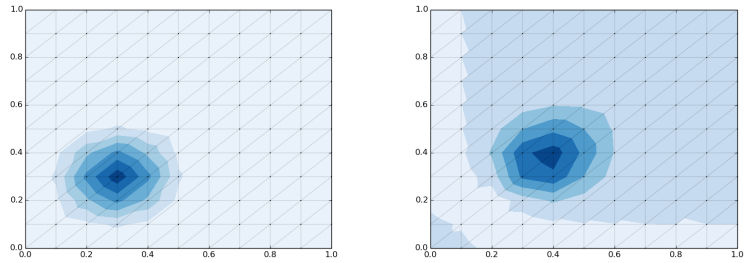
Graph library — GraphX — has also been implemented on top of RDDs.

Many interesting features, but for us: Pregel-like algorithms on graphs.

Nodes passes messages to their neighbours along edges.

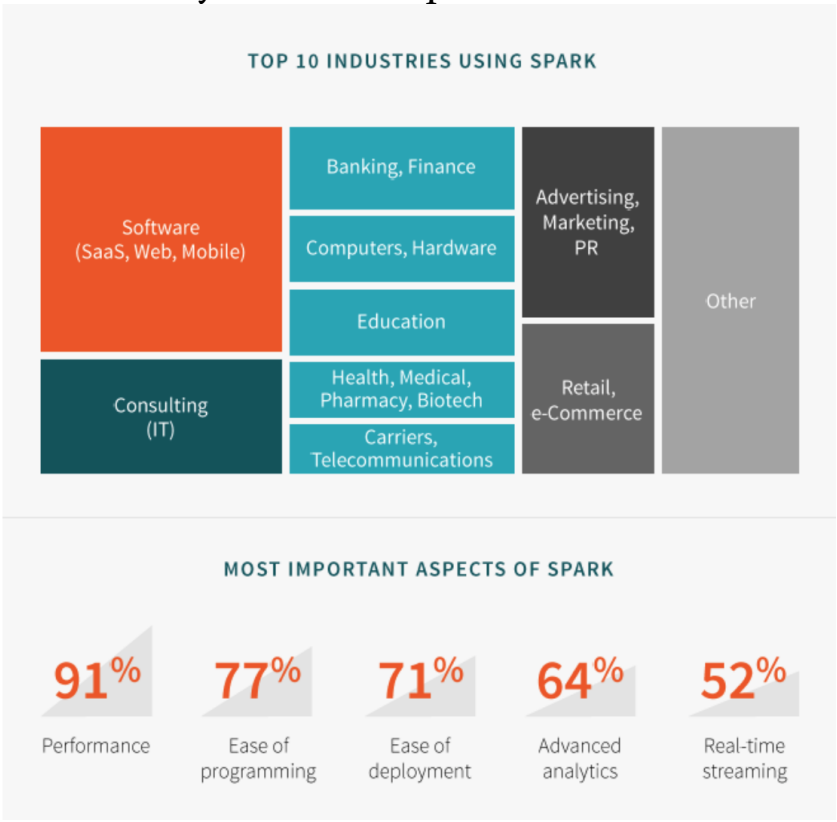
This makes implementing unstructured mesh methods extremely straightforward (Spark notebook 4):

```
def step(g:Graph[nodetype, edgetype]) : Graph[nodetype, edgety
    val terms = g.aggregateMessages[msgtype](
        // Map
        triplet => {
            triplet.sendToSrc(src_msg(triplet.attr, triplet.sr
            triplet.sendToDst(dest_msg(triplet.attr, triplet.s
          },
        // Reduce
        (a, b) => (a._1, a._2, a._3 + b._3, a._4 + b._4, a._5
    )

    val new_nodes = terms.mapValues((id, attr) => apply_update

    return Graph(new_nodes, graph.edges)
}
```



Adoption has been rapid and substantial

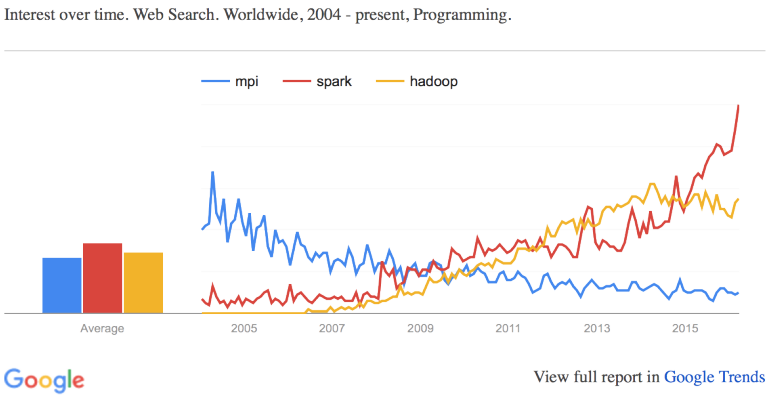
- Biggest open-source big-data platform
- Huge private-sector adoption, which has been hard for HPC
- Selected in industry because of performance and ease of use



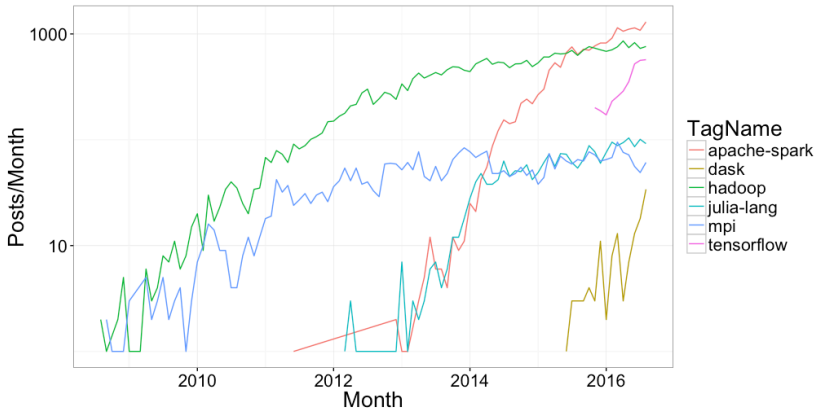
http://insidebigdata.com/2015/11/17/why-is-apache-spark-so-hot/

Adoption has been enormous

- Huge interest by programmers



Google Search



Questions on Stack Overflow

Cons

- JVM Based (Scala) means C interoperability always fraught.
- Not much support for high-performance interconnects (although that's coming from third parties - HiBD group at OSU)
- Very little explicit support for multicore yet, which leaves some performance on the ground.
- Doesn't scale *down* very well; very heavyweight

Pros

- Very rapidly growing
- Performance improvements version to version
- Easy to find people willing to learn

Dask: http://dask.pydata.org/

Dask

Overview

Dask is a python parallel computing package

- Very new - 2015
- As small as possible
- Scales down very nicely
- Works very nicely with NumPy, Pandas, Scikit-Learn
- Is definitely nibbling into MPI "market share"
  - For traditional numerical computing on few nodes
  - For less regular data analysis/machine learning on large scale
  - (likely siphoning off a little uptake of Spark, too)

Used for very general data analysis (linear algebra, trees, tables, stats, graphs...) and machine learning

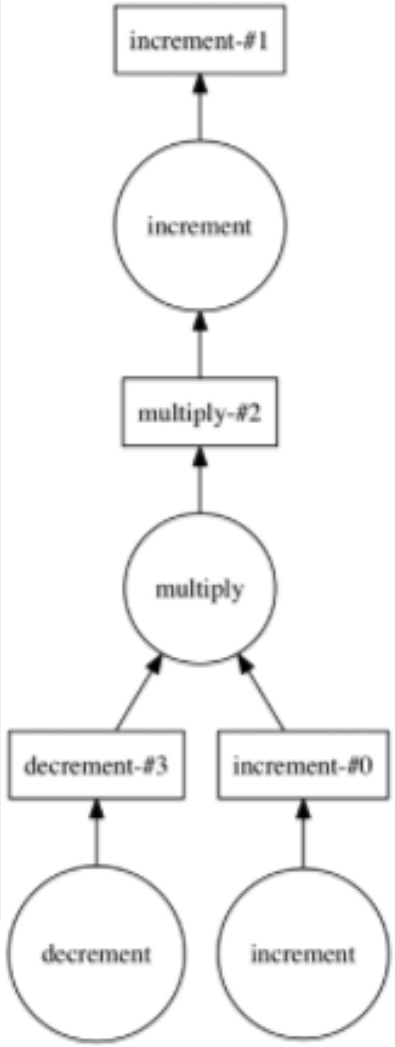
Dask

Overview

Task Graphs

Allows manual creation of quite general parallel computing data flows (making it a great way to prototype parallel numerical algorithms):

```
from dask import delayed, value

@delayed
def increment(x, inc=1):
    return x + inc

@delayed
def decrement(x, dec=1):
    return x - dec

@delayed
def multiply(x, factor):
    return x*factor

w = increment(1)
x = decrement(5)
y = multiply(w, x)
z = increment(y, 3)

from dask.dot import dot_graph
dot_graph(z.dask)

z.compute()
```



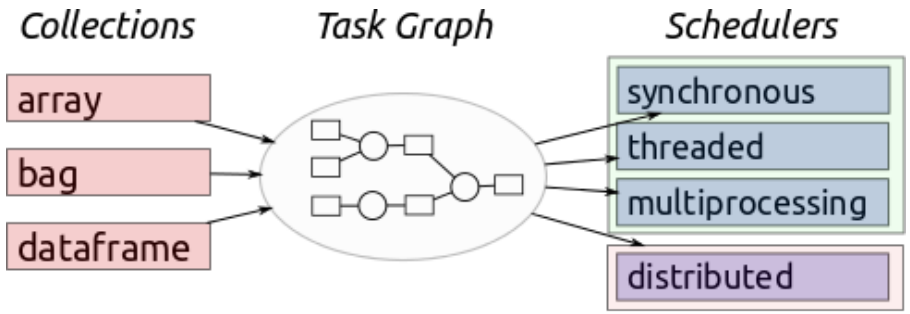
Dask

Overview

Task Graphs

Once the graph is constructed, computing means scheduling either across threads, processes, or nodes

- Redundant tasks (recomputation) pruned
- Intermediate tasks discarded after use
- Memory use kept low
- If guesses wrong, task dies, scheduler retries
  - Fault tolerance



http://dask.pydata.org/en/latest/index.html

Dask

Overview

Task Graphs

Dask Arrays

Array support also includes a small but growing number of linear algebra routines

Dask allows out-of-core computation on arrays (or dataframes, or bags of objects): will be increasingly important in NVM era

- Graph scheduler automatically pulls only chunks necessary for any task into memory
- New: intermediate results can be spilled to disk

```
file = h5py.File(hdf_filename,'r')
mtx = da.from_array(file['/M'], chunks=(1000, 1000))
u, s, v = da.linalg.svd(mtx)
u.compute()
```

Dask

Overview

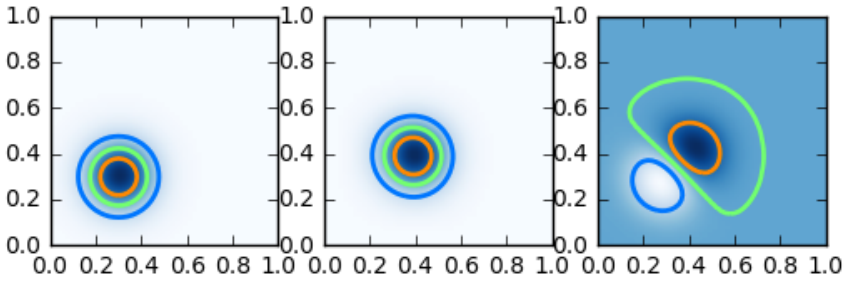
Task Graphs

Dask Arrays

Arrays have support for guardcells, which make certain sorts of calculations trivial to parallelize (but lots of copying right now):

(From dask notebook)

```
subdomain_init = da.from_array(dens_init, chunks=((npts+1)//2

def dask_step(subdomain, nguard=2):
    # `advect` is just operator on a numpy array
    return subdomain.map_overlap(advect, depth=nguard, bounda

with ResourceProfiler(0.5) as rprof, Profiler() as prof:
    subdomain = subdomain_init
    nsteps = 100
    for step in range(0, nsteps//2):
        subdomain = dask_step(subdomain)
    subdomain = subdomain.compute(num_workers=2, get=mp_get)
```



Dask

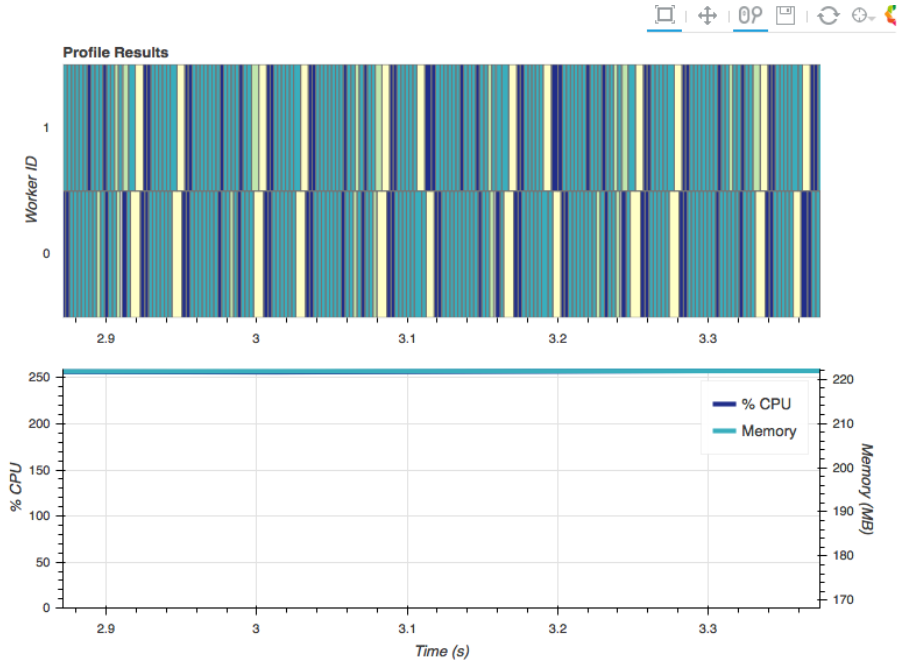
Overview

Task Graphs

Dask Arrays

Diagnostics

Comes with several very useful performance profiling tools which will be instantly famiilar to HPC community members



Cons

- Performance: Aimed at analysis tasks (big, more loosely coupled) rather than simulation
  - Scheduler+TCP: 200µs per-task overhead, orders of magnitude larger than an MPI message
  - Not a replacement in general for large-scale tightly-coupled computing
  - Python

Pros

- Trivial to install, start using
- Outstanding for prototyping parallel algorithms
- Out-of-core support baked in
- With Numba+Numpy, very reasonable single-core performance
- Automatically overlaps communication with computation: 200µs might not be so bad for some methods
- Scheduler, communications all in pure python right now, rapidly evolving:
  - **Much** scope for speedup

TensorFlow: http://tensorflow.org

TensorFlow

Overview

TensorFlow is an open-source dataflow for numerical computation with dataflow graphs, where the data is always in the form of "tensors" (n-d arrays).

Very new: Released November 2015

From Google, who uses it for machine learning.

Lots of BLAS operations and function evaluations but also general numpy-type operations, can use GPUs or CPUs.

TensorFlow

Overview

Graphs

As an example of how a computation is set up, here is a linear regression example.

TensorFlow notebook 1

In [11]:
```
# Try to find values for W and b that compute y_data = W * x_data + b
# (We know that W should be 0.1 and b 0.3, but Tensorflow will
# figure that out for us.)
W = tf.Variable(tf.random_uniform([1], -1.0, 1.0))
b = tf.Variable(tf.zeros([1]))
y = W * x_data + b

# Minimize the mean squared errors.
loss = tf.reduce_mean(tf.square(y - y_data))
optimizer = tf.train.GradientDescentOptimizer(0.5)
train = optimizer.minimize(loss)

# Before starting, initialize the variables.  We will 'run' this first.
init = tf.initialize_all_variables()

# Launch the graph.
sess = tf.Session()
sess.run(init)

# Fit the line.
for step in range(201):
    sess.run(train)
    if step % 20 == 0:
        print(step, sess.run(W), sess.run(b))
```

TensorFlow

Overview

Graphs

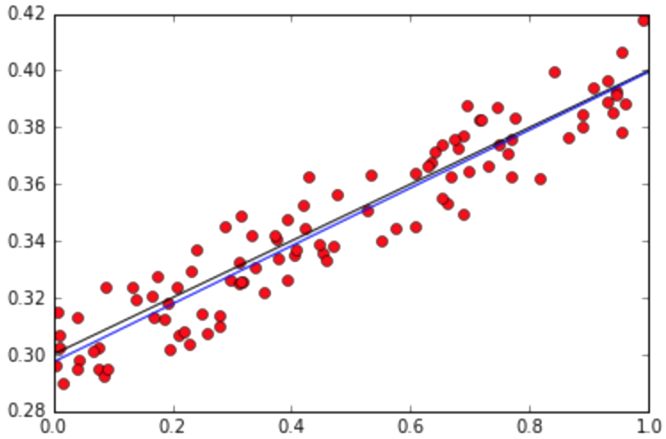
Linear regression is already built in, and doesn't need to be iterative, but this example is quite general and shows how it works.

Variables are explicitly introduced to the TensorFlow runtime, and a series of transformations on the variables are defined.

When the entire flowgraph is set up, the system can be run.

The integration of tensorflow tensors and numpy arrays is very nice.

```
Out[12]: [<matplotlib.lines.Line2D at 0x7f9fc1d3c358>]
```



TensorFlow

Overview

Graphs

Mandelbrot

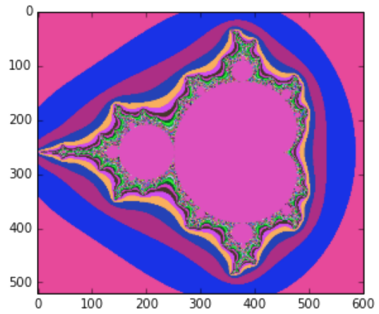
All sorts of computations on regular arrays can be performed.

Some computations can be split across GPUs, or (eventually) even nodes.

All are multi-threaded.

In [15]:
```
# Compute the new values of z: z^2 + x
zs_ = zs*zs + xs

# Have we diverged with this new value?
not_diverged = tf.complex_abs(zs_) < 4

# Operation to update the zs and the iteration count.
#
# Note: We keep computing zs after they diverge! This
#       is very wasteful! There are better, if a little
#       less simple, ways to do this.
#
step = tf.group(zs.assign(zs_),
                n_iters.assign_add(tf.cast(not_diverged, tf.float32)))

for i in range(200):
    step.run()
```

In [16]:
```
display_fractal(n_iters.eval())
```

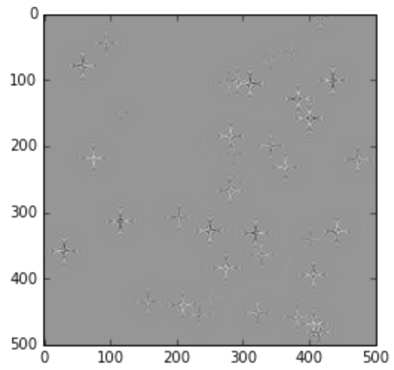


All sorts of computations on regular arrays can be performed.

Some computations can be split across GPUs, or (eventually) even nodes.

All are multi-threaded.

```
In [87]: # Initialize state to initial conditions
         tf.initialize_all_variables().run()

         # Run 1000 steps of PDE
         for i in range(1000):
           # Step simulation
           step.run({eps: 0.03, damping: 0.04})
           # Visualize every 50 steps
           if i % 50 == 0:
             display_array(U.eval())
```



As with laying out the computations, distributing the computations is still quite manual:

```
with tf.device("/job:ps/task:0"):
  weights_1 = tf.Variable(...)
  biases_1 = tf.Variable(...)

with tf.device("/job:ps/task:1"):
  weights_2 = tf.Variable(...)
  biases_2 = tf.Variable(...)

with tf.device("/job:worker/task:7"):
  input, labels = ...
  layer_1 = tf.nn.relu(tf.matmul(input, weights_1) + biases_1)
  logits = tf.nn.relu(tf.matmul(layer_1, weights_2) + biases_2
  # ...
  train_op = ...

with tf.Session("grpc://worker7.example.com:2222") as sess:
  for _ in range(10000):
    sess.run(train_op)
```

Communications is done using gRPC, a high-performance RPC library based on what Google uses internally.

TensorFlow

Overview

Graphs

Mandelbrot

Wave Eqn

Distributed

Adoption

Very rapid adoption, even though targetted very narrowly: deep learning

All threaded number crunching on arrays and communication of results of those array calculations

Cons

- N-d arrays only means limited support for, e.g., unstructured meshes, hash tables (bioinformatics)
- Distribution of work remains limited and manual (but is expected to improve - Google uses this)

Pros

- C++ - interfacing is much simpler than Spark
- Fast
- GPU, CPU support, not unreasonable to expect Phi support shortly
- Great for data processing, image processing, or computations on n-d arrays

Chapel: http://chapel.cray.com

Chapel

Overview

Chapel was one of several languages funded through DARPA HPCS (High Productivity Computing Systems) project. Successor of ZPL.

A PGAS language with global view; that is, code can be written as if there was only one thread (think OpenMP)

```
config const m = 1000, alpha = 3.0;

const ProblemSpace = {1..m} dmapped Block({1..m});

var A, B, C: [ProblemSpace] real;

B = 2.0;
C = 3.0;

A = B + C;
```

```
$ ./a.out --numLocales=8 --m=50000
```

Chapel

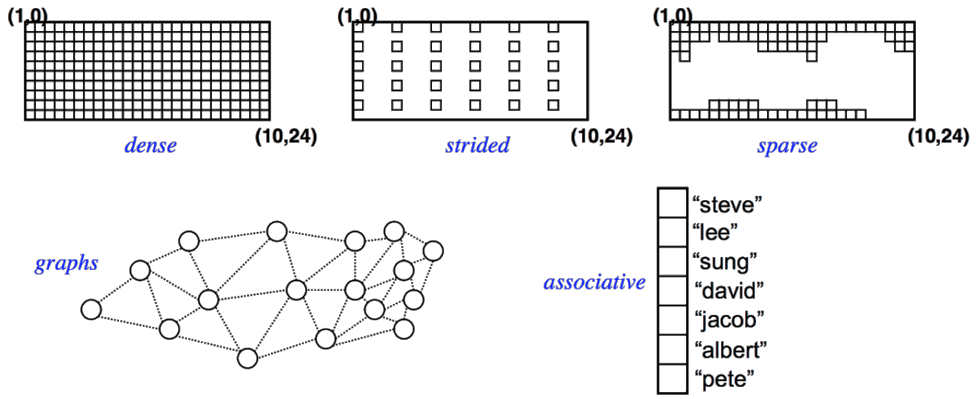
Overview

Domain Maps

Chapel, and ZPL before it:

- Separate the expression of the concurrency from that of the locality.
- Encapsulate *layout* of data in a "Domain Map"
- Express the currency directly in the code - programmer can take control
- Allows "what ifs", different layouts of different variables.

What distinguishes Chapel from HPL (say) is that it has these maps for other structures - and user can supply domain maps:



http://chapel.cray.com/tutorials/SC09/Part4_CHAPEL.pdf

Running the Jacobi example shows a standard stencil-on-regular grid calculation:

```
$ cd ~/examples/chapel_examples
$ chpl jacobi.chpl -o jacobi
$ ./jacobi
Jacobi computation complete.
Delta is 9.92124e-06 (< epsilon = 1e-05)
no of iterations: 60
```

```
var iteration = 0,                    // iteration counter
    delta: real;                      // measure of convergence

const north = (-1,0), south = (1,0), east = (0,1), west = (0,-1);

do {
  // compute next approximation using Jacobi method and store in XNew
  forall ij in ProblemSpace do
    XNew(ij) = (X(ij+north) + X(ij+south) + X(ij+east) + X(ij+west)) / 4.0;

  // compute difference between next and current approximations
  delta = max reduce abs(XNew[ProblemSpace] - X[ProblemSpace]);

  // update X with next approximation
  X[ProblemSpace] = XNew[ProblemSpace];

  // advance iteration counter
  iteration += 1;

  if (verbose) {
    writeln("iteration: ", iteration);
    writeln(X);
    writeln("delta: ", delta, "\n");
  }
} while (delta > epsilon);
```

Lots of things do stencils on fixed rectangular grids well; maybe more impressively, concurrency primitives allow things like distributed tree walks simply, too:

```
proc buildTree(height: uint = treeHeight, id: int = 1): node {
  var newNode = new node(id);

  if height > 1 {
    cobegin {
      newNode.left  = buildTree(height-1, id + 1);
      newNode.right = buildTree(height-1, id + (1 << (height-1)));
    }
  }

  return newNode;
}
// sum() walks the tree in parallel using a cobegin, computing the sum
// of the node IDs using a postorder traversal.
//
proc sum(n: node): int {
  var total = n.id;

  if n.left != nil {
    var sumLeft, sumRight: int;
    cobegin with (ref sumLeft, ref sumRight) {
      sumLeft = sum(n.left);
      sumRight = sum(n.right);
    }
    total += (sumLeft + sumRight);
  }
  return total;
}
```

Cons

- Compiler still quite slow
- Domain maps are static, making (say) AMR a ways away.
  - (dynamic associative arrays would be a *huge* win in bioinformatics)
- Irregular domain maps are not as mature

Pros

- Growing community
- Developers very interested in "onboarding" new projects
- Open source, very portable
- Using mature approach (PGAS) in interesting ways

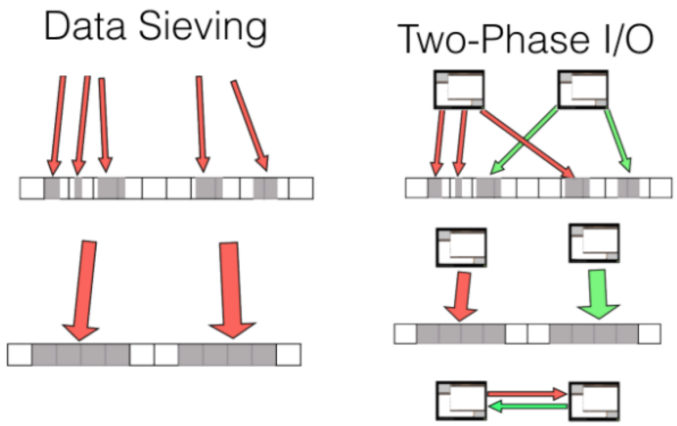
Common Themes

Higher-Level Abstractions

MPI — with collective operations and MPI-IO — has taught us well-chosen higher-level abstractions provide:

- User: **both**:
  - higher performance (performance portability, autotuning)
  - higher productivity (less code, fewer errors)
- Toolbuilder: Clear, interesting targets for:
  - Algorithmic development (research)
  - Implementation tuning (development)

Better deal for both.



Common Themes

Higher-Level Abstractions

- Spark: Resilient distributed data set (table), upon which:
  - Graphs
  - Dataframes/Datasets
  - Machine learning algorithms (which require linear algebra)
  - Mark of a good abstraction is you can build lots atop it!
- Dask:
  - Task Graph
  - Dataframe, array, bag operations
- TensorFlow:
  - Data flow
  - Certain kinds of “Tensor” operations
- Chapel:
  - Domains
  - Locales

Common Themes

Higher-Level Abstractions

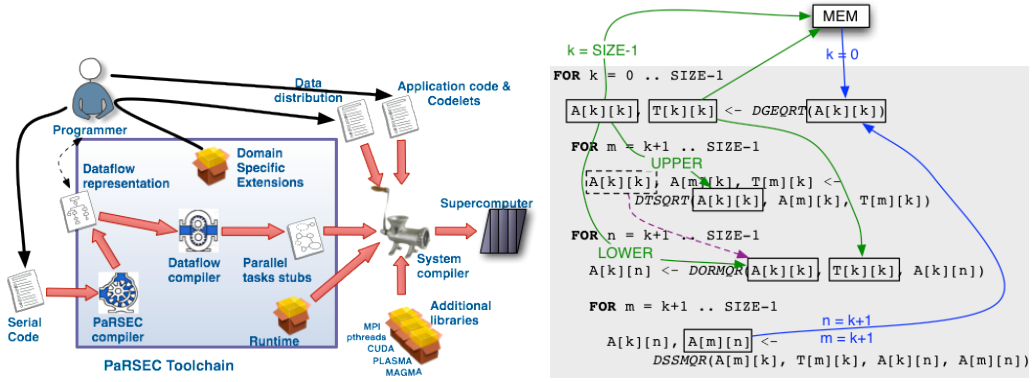
Data Flow

All of the approaches we've seen implicitly or explicitly constructed dataflow graphs to describe where data needs to move.

Then can build optimization on top of that to improve data flow, movement

These approaches are extremely promising, and already completely usable at scale for some sorts of tasks.

Already starting to attract attention in HPC, e.g. PaRSEC at ICL:



Common Themes

Higher-Level Abstractions

Data Flow

Data Layers

There has been a large amount of growth of these parallel programming environments for data crunching in the past few years

None of the big data models described here are as old as 4 years of age as public projects.

For all of them, performance is a selling point for their audience.

None use MPI.

Data Layers

Data layers

Even if you don't like the particular ones listed, programming models/environments like above would be useful for our scientists!

The ones above are tuned for iterative data analysis

- Less tightly coupled than much HPC simulation
- But that's a design choice, not a law of physics
- And science has a lot of data analysis to do anyway

They aren't competitors for MPI, they're the sort of things we'd like to have implemented atop MPI, even if we didn't use them ourselves.

⇒ Worth examining the data-movement layers of these stacks to see what features they require in a communications framework.

Spark, Flink: Akka http://akka.io

Akka

Overview

Akka is modeled after Erlang:

```
class Ping(pong: ActorRef) extends Actor {
  var count = 0
  def incrementAndPrint { count += 1; println("ping") }
  def receive = {
    case StartMessage =>
        incrementAndPrint
        pong ! PingMessage
    case PongMessage =>
        incrementAndPrint
        if (count > 99) {
          sender ! StopMessage
          context.stop(self)
        } else {
          sender ! PingMessage
        }
  }
}

object PingPongTest extends App {
  val system = ActorSystem("PingPongSystem")
  val pong = system.actorOf(Props[Pong], name = "pong")
  val ping = system.actorOf(Props(new Ping(pong)), name = "pi

  ping ! StartMessage
}
```

http://alvinalexander.com/scala/scala-akka-actors-ping-pong-simple-example

Akka

Overview

Akka is a Scala based concurrency package that powers Spark, Flink:

Actors are message passing

- Active messages (RPC)
  - Messages trigger code
- Asynchronous communication and synchronous (w/ timeouts)
- Unreliable transport:
  - At most once
  - But in-order guarantee for successful messages
- Failure detection
- Several backends
  - HTTP, TCP, or UDP
- Support for persistence/checkpointing
- Support for migration
- Support for streaming: persistent data flow

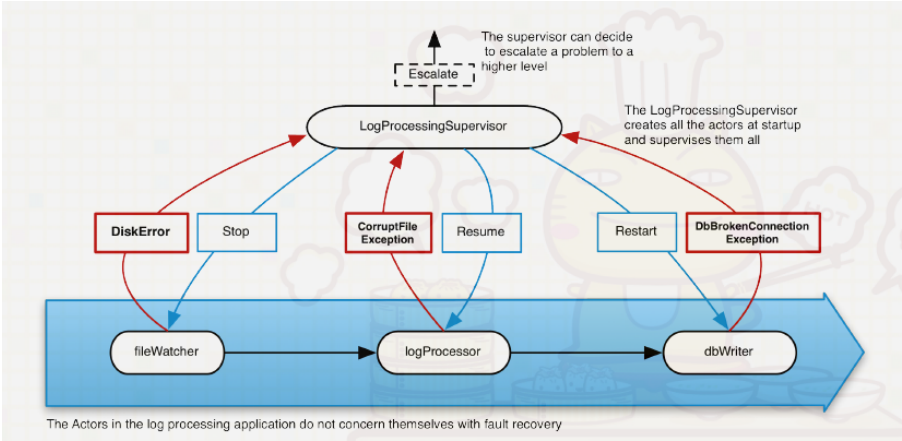
Also support for futures, etc.

Akka

Overview

Benefits for Spark

- Active Messages/Actors support very irregular executions discovered at runtime
- Akka + JVM allows easy deployment of new code (à la Erlang)
- Fault tolerance supports the resiliency goal



http://zhpooer.github.io/2014/08/03/akka-in-action-testing-actors/

UPC, Chapel, CAF, *etc.*: GASNET https://gasnet.lbl.gov

GASNet

Overview

History:

- 2002: Course Project (Dan Bonachea, Jaein Jeong)
- Refactored UPC network stack into library for other applications

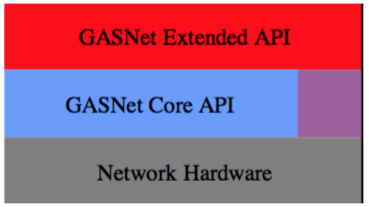
Features:

- Core API: Active messages
- Extended API: RMA, Synchronization
- Guaranteed delivery but unordered
- Wide range of backends

Used by UPC, Coarray Fortran, OpenSHMEM reference implementation, Legion, Chapel...

Blocking	Non-blocking with explicit handle	Non-blocking with implicit handle
gasnet_get(…) gasnet_get(…) … gasnet_get(…)	h = gasnet_get_nb(…) gasnet_wait_syncnb(h) h = gasnet_get_nb(…) gasnet_wait_syncnb(h) … h = gasnet_get_nb(…) gasnet_wait_syncnb(h)	gasnet_get_nbi(…) gasnet_wait_syncnbi_all() gasnet_get_nbi(…) gasnet_wait_syncnbi_all() … gasnet_get_nbi(…) gasnet_wait_syncnbi_all()

Figure 4: Message and synchronization pattern for ping-pong test



https://people.eecs.berkeley.edu/~bonachea/upc/gasnet-project.html

GASNet

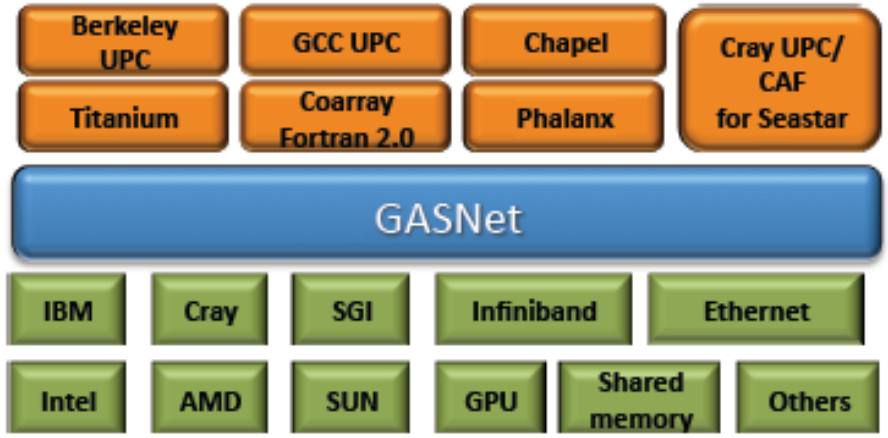
Overview

Benefits for PGAS languages

RMA is very efficient for random access to large distributed mutable state

- So not immediately helpful for Spark RDDs
- Very useful for HPC
- Compilers are very capable at reordering slow memory access to hide latency

Active messages greatly simplify irregular communications patterns, starting tasks remotely



https://xstackwiki.modelado.org/DEGAS

TensorFlow, Dask, and one or two other things:
TCP/UDP/Ethernet

TensorFlow, Dask, and one or two other things:

TCP/UDP/Ethernet

No, seriously

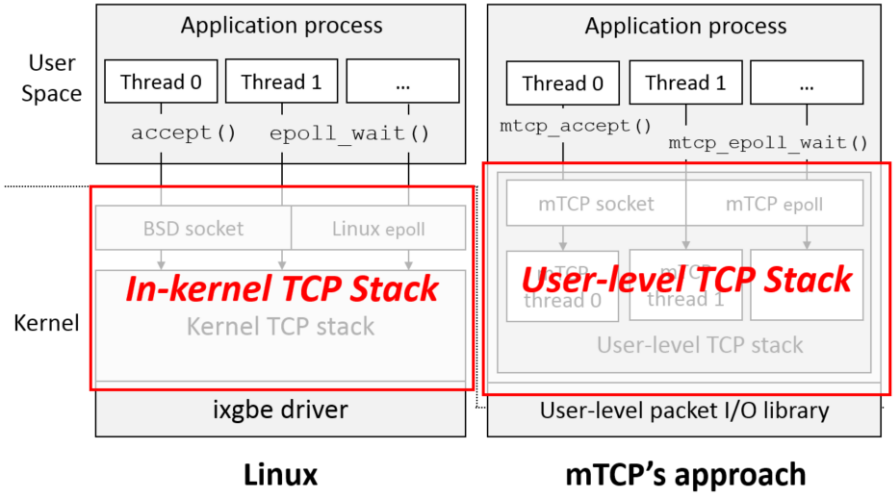
TCP/UDP Ethernet

High-Frequency traders who *lay undersea cables* to avoid latency use TCP, often even internally.

User-space networking stacks can have true zero-copies, avoid context switches, and so can be very performant, eg:

- Cisco UCS usNIC
- Teclo
- mTCP
- lkl, libuinet

Can be specialized to LAN only, optimized



http://github.com/eunyoung14/mtcp

TCP/UDP

Very mature

Ethernet

Lots of experience with RPC, eg gRPC from Google (used in TensorFlow); unreliable transport available

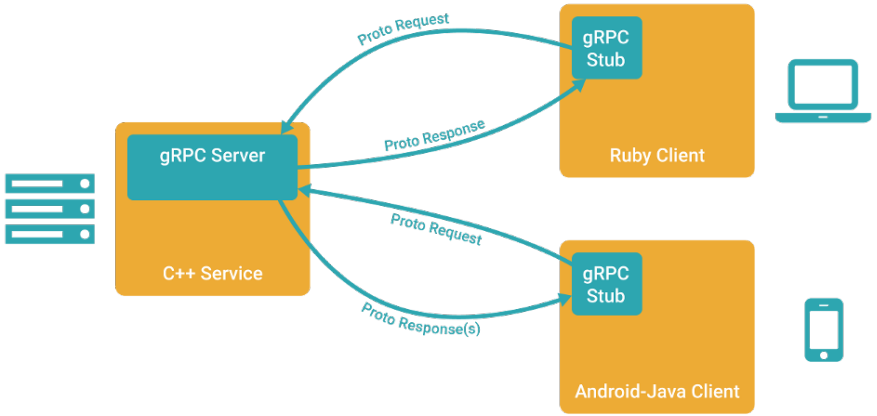
Benefits

Fault-tolerant

Data Plane Development Kit (DPDK) - userspace ethernet; ~80 cycles/packet

~45% of current top 500 is 10G or 1G ethernet

But no RDMA (or slower).



https://grcp.io

TCP/UDP Ethernet

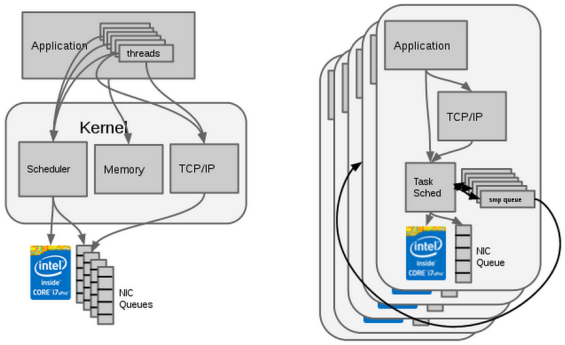
Benefits

Example of this: ScyllaDB NoSQL database, much faster than competitors

- C++
- Sockets/Ethernet based,
  - Their own user-space TCP
  - and/or DPDK

Based on underlying SeaStar framework

- Very interesting C++ concurrent/parallel tools
- Futures/Promises: Continuations
- On-node message passing
- Off-node with explicit networking



http://scylladb.com

Upcoming: LibFabric, UCX

Libfabric, UCX

Overview

Many different types of projects have to re-invent the wheel of network-agnostic layer

- MPI Implementations
- GASNet
- High performance file systems
- Control plane

Projects like Libfabric, UCX, and CCI (now less active) aim to package up this layer and expose it to a variety of consumers

Support

- Active messages
- Passive messages
- RMA
- Atomics (CAS,...)

Libfabric, UCX

Overview

Libfabric

Grew out of the OpenFabrics alliance

Hardware abstraction, not network: quite low-level

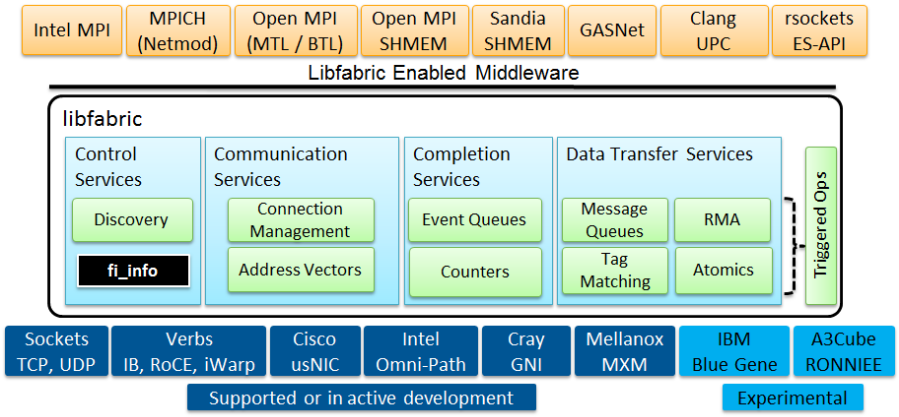
Scalable addressing support (minimal memory usage)

Lightweight

Reliable or unreliable transport

Substantial support: DOE, DOD, NASA, Intel, Cray, Cisco, Mellanox, IBM, UNH

Some OpenSHMEM implementations, OpenMPI OFI MTL, MPICH channel, GASNet OFI conduit...



Libfabric, UCX

Overview

Libfabric

UCX

Started as UCCS, based on OpenMPI BTL/MTLs

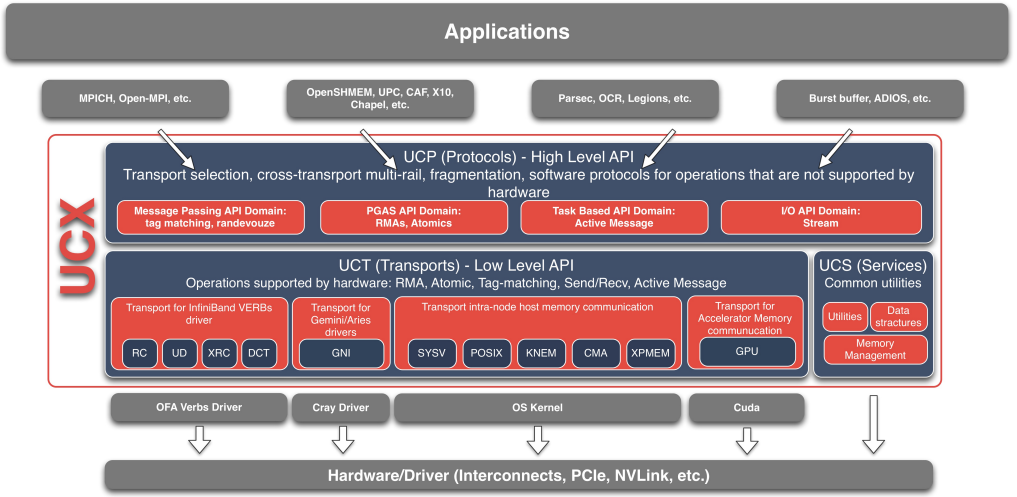
Aims as being higher-level API than Libfabric but in truth there's much overlap

Model is more that a single job/task has a UCX "universe"

IBM, UT, Mellanox, NVIDIA, ORNL, Pathscale, UH

Reliable but out-of-order delivery

MPICH, An OpenSHEMEM implementation, OpenMPI..



Two solid, rapidly maturing projects

Clearly capable of supporting higher-level data layers (MPI, GASNet, OpenSHMEM), likely high-enough level for programming models to compile down to

Will greatly reduce the barrier to writing high-performance cluster applications

There are slight differences of focus, but is there room for both efforts?

- Time will tell
- Competition is good

Data layers

Summary

The data layers that support these richer programming models have a few things in common.

- Relaxed transport reliability semantics
  - Relaxing one or both of in-order and arrival guarantees, at least optionally
  - Sometimes it's more natural to handle that at higher levels
  - In some other cases, not necessary
  - In these cases, can get higher performance

Data layers

Summary

The data layers that support these richer programming models have a few things in common.

- Relaxed transport reliability semantics
  - Relaxing one or both of in-order and arrival guarantees, at least optionally
  - Sometimes it's more natural to handle that at higher levels
  - In some other cases, not necessary
  - In these cases, can get higher performance
- Fault Tolerance
  - More natural with relaxed semantics
  - Crucial for commodity hardware, large-scale

Data layers

Summary

The data layers that support these richer programming models have a few things in common.

- Relaxed transport reliability semantics
  - Relaxing one or both of in-order and arrival guarantees, at least optionally
  - Sometimes it's more natural to handle that at higher levels
  - In some other cases, not necessary
  - In these cases, can get higher performance
- Fault Tolerance
  - More natural with relaxed semantics
  - Crucial for commodity hardware, large-scale
- Active messages/RPC
  - Makes more dynamic and/or irregular communication patterns much more natural
  - Allows much more complicated problems

Data layers

Summary

The data layers that support these richer programming models have a few things in common.

- Relaxed transport reliability semantics
  - Relaxing one or both of in-order and arrival guarantees, at least optionally
  - Sometimes it's more natural to handle that at higher levels
  - In some other cases, not necessary
  - In these cases, can get higher performance
- Fault Tolerance
  - More natural with relaxed semantics
  - Crucial for commodity hardware, large-scale
- Active messages/RPC
  - Makes more dynamic and/or irregular communication patterns much more natural
  - Allows much more complicated problems
- RMA
  - Used in some but not all
  - Very useful for handling large distributed mutable state
  - Not all problem domains above require this, but very useful for simulation

Where does (and could) MPI fit in

Whither MPI

There's so much going on!

Most exciting time in large-scale technical computing maybe ever.

“Cambrian Explosion” of new problems, tools, hardware:

- This is what we signed up for!

MPI has much to contribute:

- Great implementations
- Great algorithmic work
- Dedicated community which “eats its dog food”

But neither MPI implmentations nor expertise is being used

How Can MPI Fit Into Today's Big Computing?

Whither MPI

There's so much going on!

"MPI" is a lot of things:

- Standard
- Implementations
- Algorithm development community
- Software development community

Lots of places to fit in!

Whither MPI

There's so much going on!

Standards

MPI 4.0 is an enormous opportunity

- It's becoming clear what the needs are for
  - Large scale scientific data analysis
  - Large scale "big data"-type analysis
  - Towards exascale
- x.0 releases are precious things
  - Breaking backwards compatibility is allowed
  - Breaking backwards compatibility is expected

Whither MPI

There's so much going on!

Standards

For the API, it's fairly clear what the broader community needs:

Relaxed reliability

- Strict is primarily useful for scientists writing MPI code
- Scientists shouldn't be writing MPI code
- Allow applications that can handle out-of-order or dropped messages the performance win by doing so

Whither MPI

There's so much going on!

Standards

For the API, it's fairly clear what the broader community needs:

Relaxed reliability

- Strict is primarily useful for scientists writing MPI code
- Scientists shouldn't be writing MPI code
- Allow applications that can handle out-of-order or dropped messages the performance win by doing so

Fault tolerance

- Heroic attempts at this for years
- Increasingly, vitally necessary
- Maybe only allow at relaxed reliability levels?

Whither MPI

There's so much going on!

Standards

For the API, it's fairly clear what the broader community needs:

Relaxed reliability

- Strict is primarily useful for scientists writing MPI code
- Scientists shouldn't be writing MPI code
- Allow applications that can handle out-of-order or dropped messages the performance win by doing so

Fault tolerance

- Heroic attempts at this for years
- Increasingly, vitally necessary
- Maybe only allow at relaxed reliability levels?

Active messages

- Really really important for dynamic, irregular problems (all at large enough scale)

Whither MPI

There's so much going on!

Standards

More Access to MPI Runtimes

- Implementations have fantistic, intelligent runtimes
- Already make many decisions for the user
- Some bits are exposed through tools interface
- Embrace the runtime, allow users more interaction

Whither MPI

There's so much going on!

Standards

Such an MPI could easily live between (say) Libfabric/UCX and rich programming models

Delivering real value:

- Through API
- Through Datatypes
- Through Algorithms
  - Collectives
  - IO
  - Other higher-level primitives?
- Through intelligent runtimes

Could be a network-agnostic standard not just in HPC but elsewhere

Whither MPI

There's so much going on!

Standards

Algorithms

Collectives over nonreliable transport

What does something like Dask or Spark — maybe built on MPI-4 — look like when dealing with NVM?

- External-memory algorithms become cool again!
- Migrating pages of NVM with RDMA?

Increased interest in execution graph scheduling

- But centralized scheduler is bottleneck
- What does work-stealing look like between schedulers?
- Google Omega

Spark has put a lot of work into graph and machine-learning primitives

- But for RDDs
- What can you improve on with mutability?

Whither MPI

There's so much going on!

Standards

Algorithms

Coding

Projects exist that could greatly reduce time-to-science immediately, while MPI-4 is sorting out, and eventually take advantage of an MPI-4's capabilties

Dask:

- Try a high-performance network backend (libfabric/UCX?)
- Distributed scheduling

TensorFlow:

- Higher-level DSL that generates the data flow graphs
- Extend support to other accellerators

Chapel:

- Port code over, help find performance issues
- Help with partial collectives (*eg* not `MPI_COMM_WORLD`)
- Lots of compiler work to be done: LLVM, optimizations, IDE

The Future is Wide Open

A possible future exists where every scientist — and most data scientists and big data practitioners — rely on MPI every day while only a few write MPI code

The Future is Wide Open

A possible future exists where every scientist — and most data scientists and big data practitioners — rely on MPI every day while only a few write MPI code

Sockets for high-performance technical computing

The Future is Wide Open

A possible future exists where every scientist — and most data scientists and big data practitioners — rely on MPI every day while only a few write MPI code

Sockets for high-performance technical computing

The platform to build important communications and numerical algorithms, rich programming and analysis enviroments

The Future is Wide Open

A possible future exists where every scientist — and most data scientists and big data practitioners — rely on MPI every day while only a few write MPI code

Sockets for high-performance technical computing

The platform to build important communications and numerical algorithms, rich programming and analysis enviroments

There's much to be done, and the opportunities are everywhere

The Future is Wide Open

A possible future exists where every scientist — and most data scientists and big data practitioners — rely on MPI every day while only a few write MPI code

Sockets for high-performance technical computing

The platform to build important communications and numerical algorithms, rich programming and analysis enviroments

There's much to be done, and the opportunities are everywhere

But the world won't wait

The Future is Wide Open

A possible future exists where every scientist — and most data scientists and big data practitioners — rely on MPI every day while only a few write MPI code

Sockets for high-performance technical computing

The platform to build important communications and numerical algorithms, rich programming and analysis enviroments

There's much to be done, and the opportunities are everywhere

But the world won't wait

The world isn't waiting

The Future is Wide Open

A possible future exists where every scientist — and most data scientists and big data practitioners — rely on MPI every day while only a few write MPI code

Sockets for high-performance technical computing

The platform to build important communications and numerical algorithms, rich programming and analysis enviroments

There's much to be done, and the opportunities are everywhere

But the world won't wait

The world isn't waiting

Good Luck!