

An Introduction to Parallel Scientific Computing For Mathematicians

Chris Rackauckas

University of California, Irvine

April 11, 2015

Overview

- Parallel computing has become the main method for solving the most complex computational problems.
- The purpose of this presentation is to introduce:
 - The types of architectures used for parallel computing
 - The main APIs/protocols of parallel computing
 - Broad overview of how to utilize this knowledge for scientific computing
 - Introduction to concurrency in algorithms
 - Establish numerical linear algebra as the center of practical parallel scientific computing

Outlook

- I will be presenting an overview of parallel computing, the architectures and tools.
- Jiancheng will present an in depth introduction to MPI / BLAS
- Tao will present “practical parallel computing”, how to write parallel scientific computing code.

Why Parallel Computing?

The technology that promises to keep Moore's Law going after 2013 is known as extreme ultraviolet (EUV) lithography. It uses light to write a pattern into a chemical layer on top of a silicon wafer, which is then chemically etched into the silicon to make chip components. EUV lithography uses very high energy ultraviolet light rays that are closer to X-rays than visible light. That's attractive because EUV light has a short wavelength—around 13 nanometers—which allows for making smaller details than the 193-nanometer ultraviolet light used in lithography today. But EUV has proved surprisingly difficult to perfect.

-MIT Technology Review

Answer to the “end of Moore's Law”: Parallel Computing.

Basic Computing Language

- An API is an abstract description of how to use a protocol / application. It describes the basic functions that are used to compute in this abstract model. It may also describe function syntax.
- A library is an implementation of an API. It contained complied code, functions, protocols adhering to a specific syntax (and is usually part of a language)
- A toolkit is a set of libraries grouped together for solving a wide range of related problems.
- A framework has “inversion of control” from a toolkit. The software already does something, and to use you you “insert your behavior” into various places of the framework.

Example

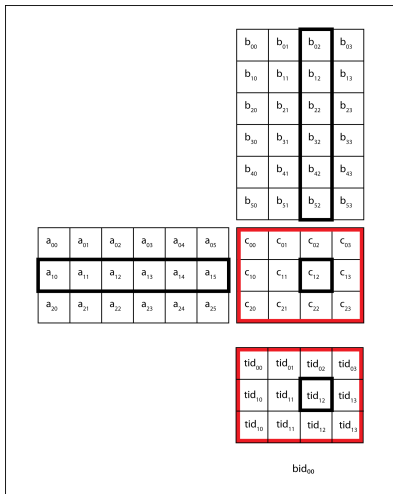
Example: Numerical Linear Algebra. Let's say you are making a Gaussian Elimination software.

- Your “theoretical methods” written in pseudocode are the API. You design it by saying there should be a function that takes in a (lower triangular matrix,vector) pair and spits out the solution to $Ax + b$, and there should be a function which takes in a matrix A and spits out a lower triangular matrix A' , etc.
- You implement these functions in MATLAB. This is a library for Gaussian Elimination.
- Your friend Bill then adds your functions to his collection of QR-factorization algorithms, eigenvalue solvers, etc. to make a Numerical Linear Algebra Toolkit for MATLAB.

Parallel Computing Architectures

- Parallel code is highly dependent on the chosen architecture
 - These correspond to what portions of the hardware are shared and what portions are physically separated.
- Different APIs / Protocols have been developed for different architectures
 - They give commands for controlling and linking each separate entity.
 - These are language independent!

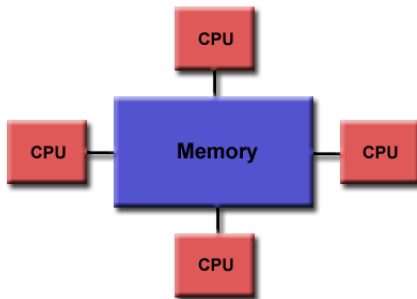
Basic Concurrency Example: Matrix Multiplication



How could you implement this?

- 1 Master sends out rows and columns, gets back answer, puts in place
- 2 Each node holds a portion of the matrix, they send each other the pieces as needed to update
- 3 Etc.

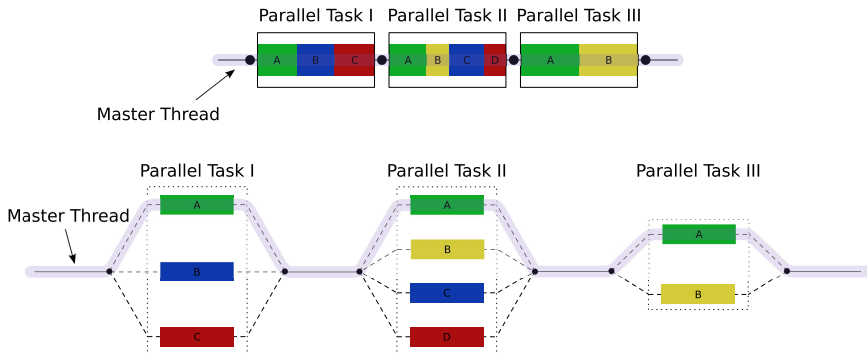
Introduction to Shared Memory Computing



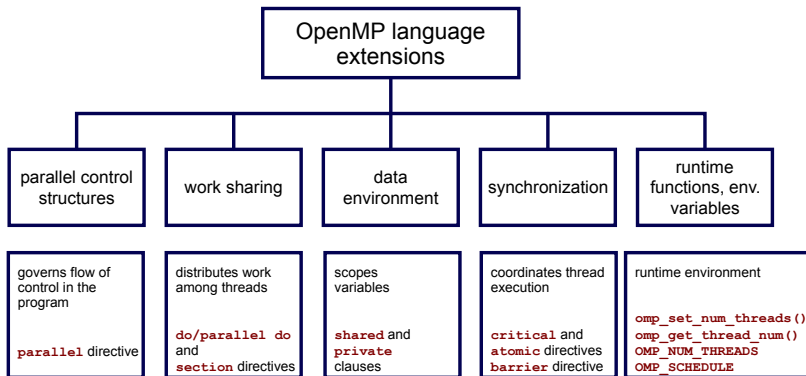
- Most basic parallel computing model.
- Multi-core PCs and Multi-chip HPC nodes follow this model.
- Usually share I/O bus and have a “master” controller.
- Usually based on some threading idea
- APIs are OpenMP or POSIX Threads.
- Sufficient for most applications

OpenMP Overview

Idea: Parallel computing via “threads”



OpenMP Directives



Shared Memory Computing Example: Threads

Example: Ensemble Model in Machine Learning

- An ensemble model uses many different statistical models and aggregates the results
- Each model takes the dataset and is “independent” from one another.

Parallelize using threads:

- 1 Write a function for running each model.
- 2 Create an array for the processes/threads.
- 3 Use a loop to call the function on each process.
- 4 Give a directive to wait for all to finish.

Shared Memory Computing Example: Threads

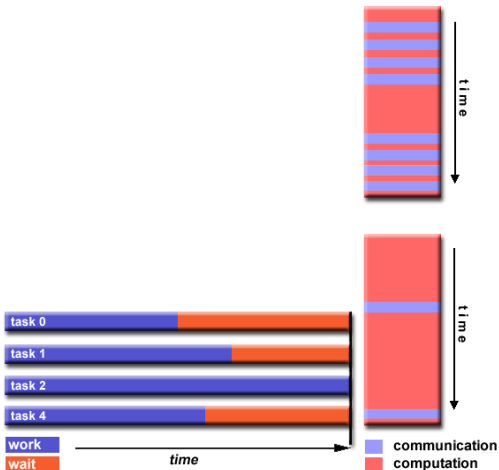
```
1 subprocesses = []
2 for trial in range(0, trials):
3     for model in modelList:
4         print("Running Model " + model.tag)
5         model.run(sproc, subprocesses)
6     for p in subprocesses:
7         p.wait()
```

Program: TBEEF Recommendation Algorithm

Purpose: Machine Learning

Language: Python, Multiprocessing Library

Potential Problems With Design



Potential Problems:

- As slow as the slowest process
 - Should double up some computationally easier models?
- Manage communication vs computation time
 - Should send multiple models to the same process at a time?
- No “Compile time optimizations”

Shared Memory Computing Example: Parallel Vectorization

Example: Many Random Walks

- Want to create a large matrix where each row is a separate random walk.

Parallelize using threads:

- 1 Create the matrix.
- 2 Write the function for the random walk.
- 3 Call parallel vectorization function.

Shared Memory Computing Example: Parallel Vectorization

```
1 clusterExport(cluster,"rows")
2 walkFunc <- function(p){
3   n <- 100
4   walk <- matrix(rbern(n*rows,p),nrow=n, ncol=rows)
5   walk <- parApply(cluster,walk,c(1,2),walker)
6   for (i in 2:n){walk[i,] <- walk[i,] + walk[i-1,]}
7   means <- parApply(cluster,walk,1,mean)
8   ans <- c()
9   ans[[1]] <- walk
10  ans[[2]] <- means
11  return(ans)}
12 walks <- c()
13 means <- c()
14 for (i in 1:length(ps)){
15   ans <- walkFunc(ps[i])
16   walks[[i]] <- ans[[1]]
17   means[[i]] <- ans[[2]]}
```

Language: R, Snow Library

Shared Memory Computing Example: Parallel Looping

Example: Monte Carlo Simulation of SDEs

- Want to understand how “knocking down a parameter” changes the mean and variance of the solution for many different parameters
 - Each parameter I want to run an independent solution.
 - All I want to know at the end is what the mean and variance was.
- 1 Call a parallel loop.
 - 2 Have the first part of the loop calculate the value.
 - 3 Save those values in a non-temporary (shared) array.

Shared Memory Computing Example: Parallel Looping

```
1 parfor k=1:K
2     %Make Random Parameters
3     set(stream,'Substream',k);
4     RA      = simMotif4(vars,crabp,cypmax,N,dt,stream);
5     mean1   = mean(RA);
6     var1    = var(RA);
7     %Next Run: Reset RNG and Knockdown
8     set(stream,'Substream',k);
9     crabp   = crabp*knock;
10    RA      = simMotif4(vars,crabp,cypmax,N,dt,stream);
11    mean2   = mean(RA);
12    var2    = var(RA);
13    %Calculate Conclusions
14    perChangeMean=abs((mean2-mean1)/max(mean1,mean2) *100);
15    perChangeVar =abs((var2-var1)/max(var1,var2) *100);
16    DeltaMeans(k)=perChangeMean;
17    DeltaVars(k) =perChangeVar;
18 end
```

Language: MATLAB, Parallel Computing Toolbox

Shared Memory Computing Example: Linear Algebra

```
1 A = rand(40000,40000);  
2 B = rand(40000,40000);  
3 A*B;
```

Language: Julia

Linear Algebra is naively parallel in many languages!

Introduction to BLAS / LINPACK

- Basic Linear Algebra Subprograms (BLAS) is a Fortran library first published in 1979.
- Contains functions for dot product, cross product, matrix multiplication.
- Modern implementations include OpenBLAS, Intel MKL, cuBLAS, etc.
 - Modern implementations are shared memory parallelized via OpenMP and others.
- LINPACK is a Fortran library for numerical linear algebra which uses BLAS.
- Most numerical programming languages use BLAS and LINPACK for matrix algebra, including MATLAB, Mathematica, NumPy, R, Julia, etc.

Never code your own basic functions! These are highly optimized!

Use of BLAS: dgemv

```

1 DGEMV - perform one of the matrix-vector operations
2 y := alpha*A*x + beta*y, or y := alpha*A'*x + beta*y,
3
4 SUBROUTINE DGEMV ( TRANS, M, N, ALPHA, A, LDA, X, INCX,
5                   BETA, Y, INCY )
6 DOUBLE          PRECISION ALPHA, BETA
7 INTEGER          INCX, INCY, LDA, M, N
8 CHARACTER*1      TRANS
9 DOUBLE          PRECISION A( LDA, * ), X( * ), Y( * )

```

MATLAB, Julia, NumPy, etc. all do this so you don't have to!

Practical Parallel Computing: Language Binding

The most practical (and most common) solution for parallel computing is then to employ these libraries from within a language.

- Interpreted languages are at a severe disadvantage speed-wise because they won't have pre-compiled functions to repeatedly call! Thus in many languages (such as MATLAB) each core will remake the function each time it's called!
- The solution is to “bind” libraries from other languages.
Common solutions are:
 - MATLAB: MEX (C bindings) and CUDA kernel bindings
 - Python: Jython (Java + Python), Cython (C + Python)
 - R: Rpp (R + C++)
- Common packages to use bound:
 - PETSc
 - BLAS / LAPACK / FFTW
 - Your own!

Language Binding Example: TBEEF

```
1 subprocesses = []
2 for trial in range(0, trials):
3     for model in modelList:
4         print("Running Model " + model.tag)
5         model.run(sproc, subprocesses)
6     for p in subprocesses:
7         p.wait()
```

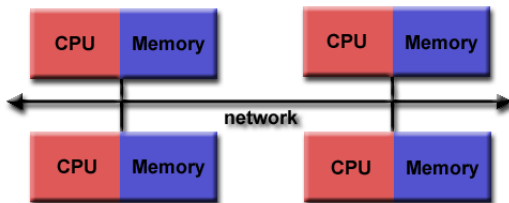
Program: TBEEF Recommendation Algorithm

Purpose: Machine Learning

Language: Python, Multiprocessing Library

The code for running the models is a function in C with OpenMP

Introduction to Distributed Memory Computing



- Commonly referred to as “Cluster Computing”
- Each computing node has its own memory, messages must be send between them.
- May share I/O / Disks, may not.
- When many nodes are involved it's known as “Grid Computing”

Distributed Memory APIs / Frameworks

Different programming models work better/worse for different implementations.

There are many different APIs for programming distributed memory systems.

Because of this, most programming requires the use of libraries built on top of these APIs:

- The most common for scientific/numerical computing is MPI. It is used in the design of numerical libraries in Fortran and C.
- Hadoop is a framework for a distributed file system (HDFS) and a common processing API known as MapReduce. This framework is designed for large datasets and is commonly used in extremely big data applications like machine learning and in internet servers.

Message Passing Interface (MPI)

- MPI is the most common distributed memory API for scientific computing.
- It is implemented in many languages including C, C++, Java, R, Python, etc.
- It has been falling out of fashion in recent years due to being “too close to the metal” and not allowing many compile-time enhancements.
- Used to build most of the “fundamental libraries” (ScaLAPACK, P-FFTW, PETSc, etc.)

MPI Example

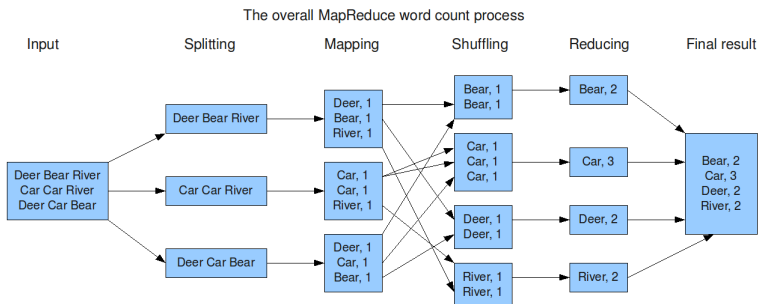
```
1 void Ax(double *v, double *u, int l_n, int l_N, int N, ...
2     int      i,j;
3     double   temp;
4     MPI_Request requests[4];
5     MPI_Status statuses[4];
6     //Communications
7     MPI_Irecv(gl,N,MPI_DOUBLE,idleft,1,MPI_COMM_WORLD,& ...
8     MPI_Irecv(gr,N,MPI_DOUBLE,idright,2,MPI_COMM_WORLD,&...
9     MPI_Isend(&(u[(N)*(l_N-1)]),N,MPI_DOUBLE,idright,1 ...
10    MPI_Isend(u,N,MPI_DOUBLE,idleft,2,MPI_COMM_WORLD,&...
11    for(j=1;j<l_N-1;j++){
12        for(i=0;i<N;i++){
13            temp = 4.0*u[i+N*j];
14            temp -= u[i + N*(j-1)];
15            if(i>0 ){ temp -= u[(i-1)+N* j    ];}
16            if(i<N-1){ temp -= u[(i+1)+N* j    ];}
17            temp -= u[i + N*(j+1)];
18            v[i+N*j] = temp;}}
19    MPI_Waitall(4,requests,statuses);
```

MapReduce

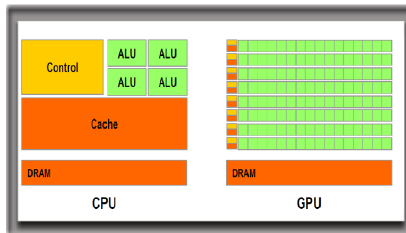
MapReduce is a framework where you “plug in” two functions:

- A mapping function for doing calculations / processing
- A reducing function for putting outputs together

Used in big data applications where the data cannot fit on any single computer alone (distributed file system)



Introduction to GPGPU Computing



Dilemma of GPGPU Computing: Fast shared memory with thousands of cheap dumb cores.

- Each slower than normal CPU cores, but can only execute a small amount of commands.
- Limited memory (Currently the Tesla K80 has 24GB of memory)

GPGPU Computing is the cheap solution for grid computing “mid-sized” massively parallel problems.

CUDA vs OpenCL

- CUDA is the most common way of doing GPGPU computing
 - C library written and maintained by Nvidia
 - Bindings exist to many different languages, so all one has to do is write critical functions in CUDA and link to Python, R, MATLAB, etc.
 - MATLAB has native “easy CUDA”
 - Can only be used with Nvidia graphics cards
 - Much faster than OpenCL (as of 2015)
- OpenCL is an open API for doing GPGPU computing
 - Based off of the OpenGL (Open Graphics Language) API
 - Implemented in many different languages
 - Works on all graphics cards / chips

Easy MATLAB CUDA

```

1  %Setup Initial Value Matrix
2  RAout = zeros(n,m,'gpuArray');
3  RAin  = zeros(n,m,'gpuArray'); ...
4  %Vectorize Parameters
5  sigma = sigma*ones(n,m,'gpuArray');
6  alphaGPU=alpha*ones(n,m,'gpuArray'); ...
7  %Automatic CuBLAS Calls (on GPGPU) for Linear Algebra
8  DiffRA = DiffX*RAout + RAout*DiffY;
9  %Parallel Vectorization Function More Efficient
10 [RAout,RAin,R,RAR,BP,RABP]=arrayfun(@RAREactions,RAout,...

```

Program: SPDE Solver

Language: MATLAB with Parallel Computing Toolbox

Pros/Cons of GPGPU Computing

Pros

- Cheap massively parallel solution.
- Tons and tons of cores with shared memory.
- Fast memory buses (DDR5).
- Perfect for “Embarrassingly parallel” applications.
- Rapidly getting better.

Cons

- Slow cores, not faster for “non-super-parallel” iterated problems.
- Low memory limits applications.
- This architecture is optimized in a very different way from traditional parallel programs, and thus **REQUIRES** a completely different implementation to get good performance.
- Cheap cards artificially slower for fp64 calculations.

Example CUDA Card



Click to open expanded view

Nvidia Tesla K80 24GB GPU Accelerator passive cooling 2x Kepler GK210 900-22080-0000-000

by [NVIDIA](#)



18 customer reviews

| [4 answered questions](#)

Price: **\$4,885.00** & **FREE Shipping**

In Stock.

Ships from and sold by [Safe Harbor Computers, NLE, Post, 3D & Broadcast.](#)

Estimated Delivery Date: April 16 - 21 when you choose Standard at checkout.

- Nvidia Tesla K80 GPU: 2x Kepler GK210
- Memory size (GDDR5) : 24GB (12GB per GPU)
- CUDA cores: 4992 (2496 per GPU)
- Memory bandwidth: 480 GB/sec (240 GB/sec per GPU)
- 2.91 Tflops double precision performance with NVIDIA GPU Boost - See more at: <http://www.nvidia.com/object/tesla-servers.html#sthash.IF5LVwFq.dpuf>

6 new from **\$4,855.00**

Xeon Phi Co-Processors

- “Cluster on a card” - chips are low-power Atom chips.
- Designed by Intel for the same purpose as GPGPUs
 - Many slow but cheap processors
- Rapidly increasing power
 - New architecture, “Knights Landing”, increases the speed 3x to ~3 Flops (~Tesla K80)
 - Knights Hill already announced and will be much faster as well!

Xeon Phi Pros / Cons

Pros

- Newest cards as fast as fastest GPGPUs
- Standard x86 architecture, i.e. your code already works for it
 - Numerical linear algebra already optimized via Intel MKL (BLAS)
 - Uses all the same libraries as CPUs for coding (OpenMP, MPI, etc)
- Showing lots of promise

Cons

- Relatively new
- Can be hard to optimize
 - Not a problem for numerical code written as linear operations!

Xeon Phi Card Example



Click to open expanded view

Intel Xeon Phi 7120P Coprocessor

by [Intel](#)



1 customer review

Price: **\$3,883.00** + \$8.00 shipping

In stock.

Usually ships within 4 to 5 days.

Ships from and sold by [8anet online store](#).


Estimated Delivery Date: April 17 - 22 when you choose Expedited at checkout.

- 61 Cores
- 1.238 GHz Clock Speed
- 300W Max TDP
- 16GB Memory
- Passively Cooled

1 used from **\$2,750.00**

Xeon Phi - New Fastest Super Computer

2018 Aurora Supercomputer to Reach 180 Petaflops

posted by [martyb](#) on Friday April 10, @06:12AM 
from the [what-90-quadrillion-rabbit's-ears-look-like](#) dept.

[takyon](#) writes:

The Register's new sister site, The Platform, [broke news](#) of an upcoming 180 petaflops supercomputer named "Aurora" to be installed at the [Argonne National Laboratory](#). The system will reportedly use 2.7x the power (from 4.8 megawatts to 13 megawatts) to deliver 18x the peak performance of Argonne's existing [Mira supercomputer](#) (more detail [here](#)).

[Aurora](#) will use Intel's upcoming 10nm "Knights Hill" [Xeon Phi](#) processors and a second-generation [Omni-Path](#) optical interconnect with far greater bandwidth than current designs. The storage capacity will exceed 150 petabytes. Cray Inc. will manufacture the system, which will cost \$200 million and round out the [CORAL trio](#) of supercomputers, including the 150-300 PFLOPS Summit at Oak Ridge National Laboratory and the 100+ PFLOPS Sierra at Lawrence Livermore National Laboratory. The other two systems will use [IBM Power9 and NVIDIA Volta chips](#).



Xeon Phi - So Fast It's a Weapon

10 Apr 2015 at 18:08, Iain Thomson



85



19



55

The US government has blocked Intel from shipping high-end Xeon processors to China's supercomputer builders – and other American chip giants are banned, too.

Intel confirmed to *The Register* last night it was refused permission to sell the chips to the Middle Kingdom's defense labs and other parts of its supercomputing industry.

"Intel was informed in August by the US Department of Commerce that an export license was required for the shipment of Xeon and Xeon Phi parts for use in specific previously disclosed supercomputer projects with Chinese customer [INSPUR](#)," a spokesperson for the Santa Clara-based biz said, adding:

Intel complied with the notification and applied for the license which was denied. We are in compliance with the US law.

Those Xeon chips are vital to high-performance computing needed for scientific research and similar work: they will be used to power [the 50,000-node, 180-petaFLOPS Aurora supercomputer](#) Intel and Cray are building for the US Department of Energy, due to go live in 2018. China's [Tianhe-2 computer](#), today the world's fastest publicly known supercomputer, uses 3.1 million Intel Xeon E5 cores to hit 54 petaFLOPS in peak performance.

The decision to deny China's boffins access to the powerful processors [emerged this week](#) – but was formalized on February 18 [in rules \[PDF\]](#) set by the End-User Review Committee (ERC) in the Bureau of Industry & Security (BIS) at the US Department of Commerce.

The ERC is a joint operation run by the Departments of Commerce, State, Defense, and Energy, and occasionally the Treasury, to decide who American companies can and can't sell to.

As a result of the change, the ERC added the National Supercomputing Center Changsha in

Cloud Computing

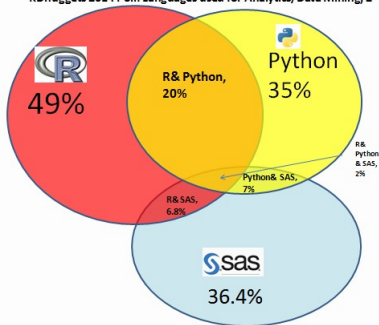
Cloud computing is non-local parallel computing.

Two distinct types:

- “Service Cloud Computing”
 - Using mainframes/servers to do the heavy computing
 - Low resource interface interacts with the user
 - Architecture is the same as previous mainframes / servers
 - This is the model for modern Software as a Service (SaaS)
 - Also useful for large scale virtualization platforms (for enterprise)
- Major services: Microsoft Azure, Amazon EC2, etc.
- “Distributed Computing”
 - Computing using different computers connected over the cloud
 - Examples: rendering, bitcoin mining, Monte Carlo simulations
 - Problems: Slow interconnects makes it not useful for many HPC applications (though can be useful for large-scale machine learning)
 - Often used with MapReduce-type frameworks.

Current Trends in Languages

KDNuggets 2014 Poll: Languages used for Analytics/Data Mining, 2



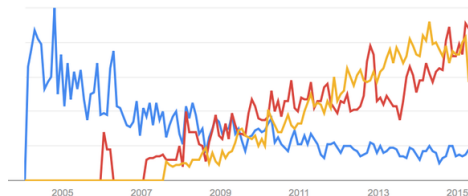
3. Languages with the highest growth in 2014 were

- Julia, 316% growth, from 0.7% share in 2013 to 2.9% in 2014
- SAS, 76% growth, from 20.8% in 2013 to 36.4% in 2014
- Scala, 74% growth, from 2.2% in 2013 to 3.9% in 2014

4. The languages with the largest decline in share of usage were

- F#, 100% decline, from 1.7% share in 2013 to zero in 2014
- C++/C, 60% decline, from 9.3% in 2013 to 3.6% in 2014
- GNU Octave, 57% decline, from 5.6% in 2013 to 2.4% in 2014
- MATLAB, 50% decline, from 12.5% in 2013 to 6.3% in 2014
- Ruby, 44% decline, from 2.2% in 2013 to 1.3% in 2014
- Perl, 41% decline, from 4.5% in 2013 to 2.6% in 2014

■ mpi ■ spark ■ hadoop



Current Trends in Languages/Protocols/Platforms

- One large trend is that “computing for everyone” languages are increasing
 - Clusters are “good enough” so people simply use SAS, Scala, Python, R, etc. to lower development time.
 - Language binding is a viable solution for many problems.
- Simple Map/Reduce Frameworks like Spark and Hadoop are growing rapidly.
 - Used in data mining, machine learning, and web applications.
 - These are too “lightweight” and problem-specific to be MPI alternatives.
- MPI is dying?
 - MPI is mostly for writing libraries for other languages.
 - People are looking for more fault-tolerant computing.

MPI Alternatives

The following are MPI alternatives that one should keep an eye on:

- Some people point to Chapel, developed in 2009. However, it has notorious problems with long compile times and does not match the speed of MPI.
- Fortran 2008 has made major improvements and added new libraries, but has “legacy” issues.
- HPX and Charm++ are two that get mentioned, but not much adoption has taken place and there are questions as to its speed.

My take: MPI is here to stay and will still be the king of low-level protocols for a long time. But very few will write at this low of a level (like Assembly). As co-processor use increases, there will also be an increase in the use of Fortran 2008 due to its co-array structures.

Higher-Level Parallel Languages

- With a large number of libraries written in lower level languages that can be bound, many have turned to less efficient but easier to work with higher-level languages.
- Currently the most common: R, Python, and MapReduce-type frameworks (Hadoop/Spark). Also Stata and SAS for statistics/machine learning. MATLAB is falling out of favor.
- There was an early experiment with concurrency through functional programming (Clojure), but that trend has died down.

However, there is a platform that is continuing to gain steam that is quite useful for numerical computations...

Julia

Julia is a high-level language that was designed for concurrent numerical computing and data analysis

- Its syntax is almost exactly like MATLAB and is interpreted
- Just In Time (JIT) compiler for C-like performance
- Uses common libraries (BLAS, LINPACK, FFTW, PETSc, etc)
- Language bindings for Python, C, R, etc.
- Lisp-like metaprogramming for genetic algorithms
- Interfaces to Intel MKL for Xeon Phi Co-processors
- Less than 3 years old, not even version 1 yet.

	Fortran	Julia	Python	R	Matlab	Octave	Mathe- matica	JavaScript	Go	LuaJIT	Java
	gcc 4.8.2	0.3.7	2.7.9	3.1.3	R2014a	3.8.1	10.0	V8 3.14.5.9	go1.2.1	gsl-shell 2.3.1	1.7.0_75
fib	0.57	2.14	95.45	528.85	4258.12	9211.59	166.64	3.68	2.20	2.02	0.96
parse_int	4.67	1.57	20.48	54.30	1525.88	7568.38	17.70	2.29	3.78	6.09	5.43
quicksort	1.10	1.21	46.70	248.28	55.87	1532.54	48.47	2.91	1.09	2.00	1.65
mandel	0.87	0.87	18.83	58.97	60.09	393.91	6.12	1.86	1.17	0.71	0.68
pi_sum	0.83	1.00	21.07	14.45	1.28	260.28	1.27	2.15	1.23	1.00	1.00
rand_mat_stat	0.99	1.74	22.29	16.88	9.82	30.44	6.20	2.61	8.23	3.71	4.01
rand_mat_mul	4.05	1.09	1.08	1.63	1.12	1.06	1.13	14.58	8.45	1.23	2.35

Figure: benchmark times relative to C (smaller is better, C performance = 1.0).