

Outro To Parallel Computing

John Urbanic
Pittsburgh Supercomputing Center
Parallel Computing Scientist

Purpose of this talk

Now that you know how to do some real parallel programming, you may wonder how much you *don't* know. With your newly informed perspective we will take a look at the parallel software landscape so that you can see how much of it you are equipped to traverse.

A quick outline...

- An example
- Scaling
- Amdahl's Law
- Languages and Paradigms
 - Message Passing
 - Data Parallel
 - Threads
 - PGAS
 - Hybrid
- Data Decomposition
- Load Balancing
- Summary

How parallel is a code?

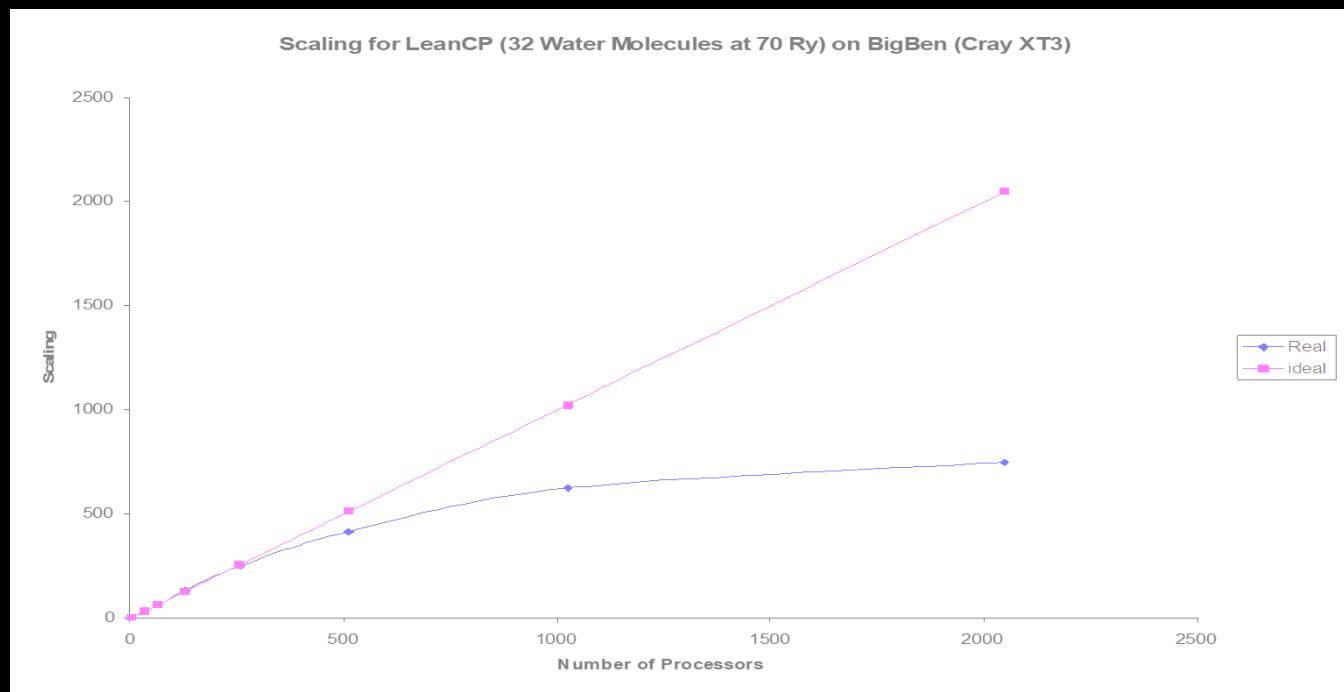
- Parallel performance is defined in terms of scalability

Strong Scalability

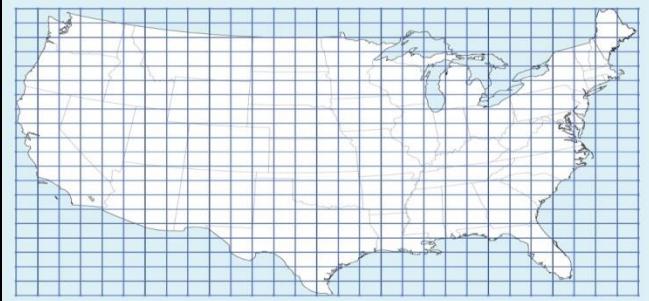
Can we get faster for a given problem size?

Weak Scalability

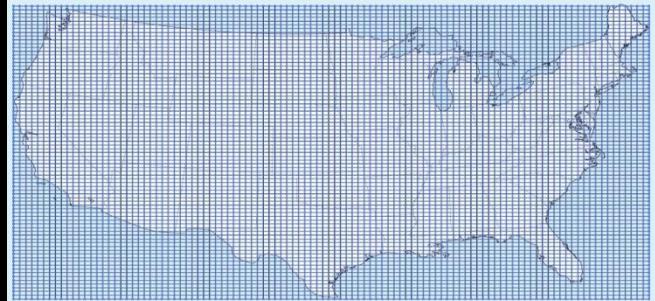
Can we maintain runtime as we scale up the problem?



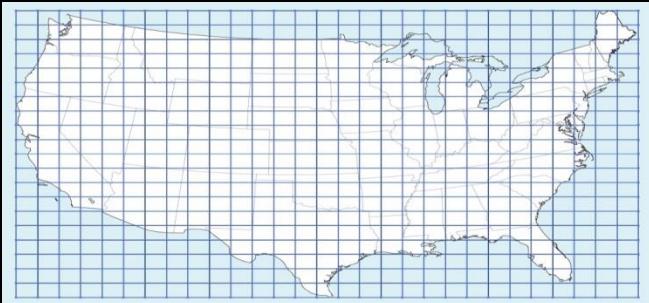
Weak vs. Strong scaling



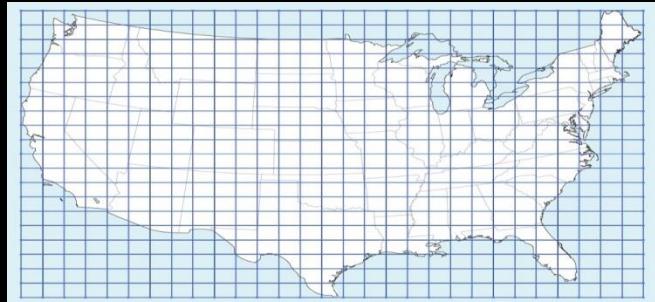
Weak Scaling



More accurate results



Strong Scaling

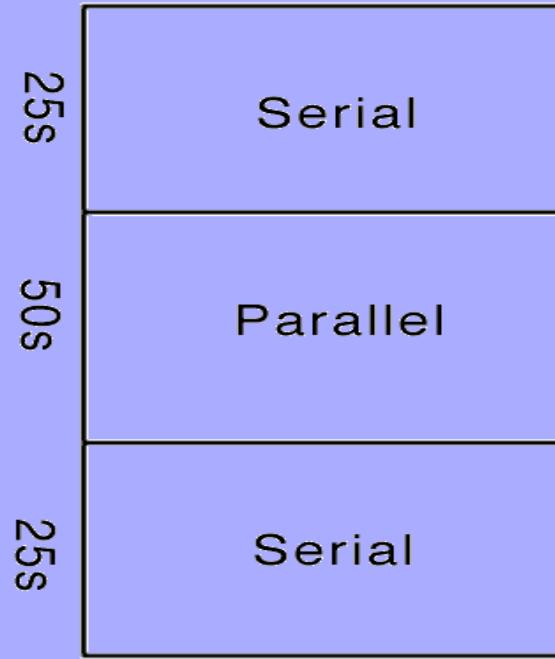


Faster results
(Tornado on way!)

Your Scaling Enemy: Amdahl's Law

How many processors can we really use?

Let's say we have a legacy code such that it is only feasible to convert half of the heavily used routines to parallel:



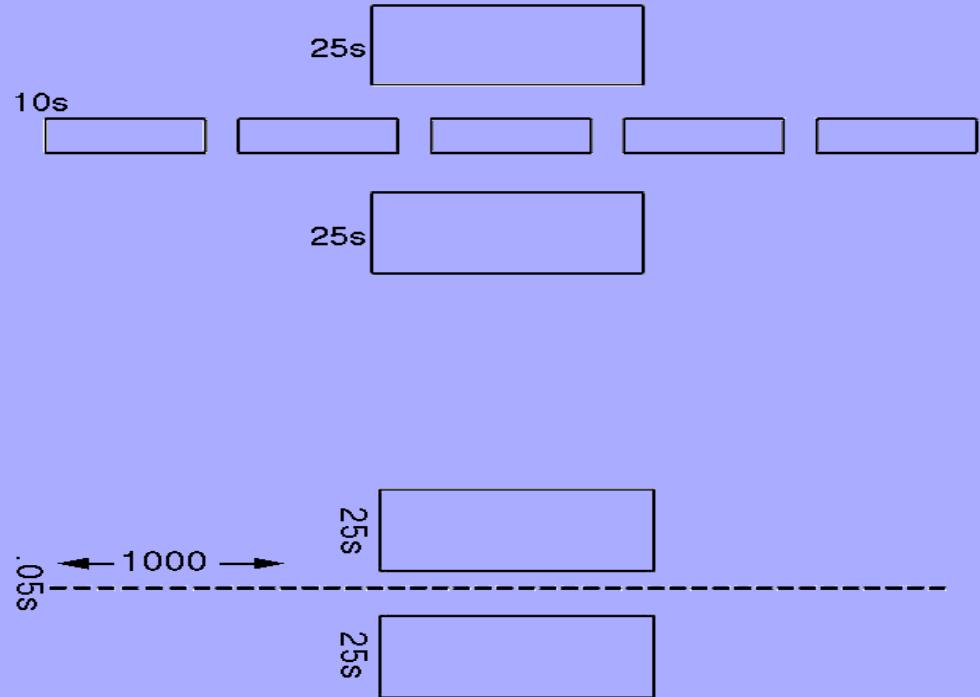
Amdahl's Law

If we run this on a parallel machine with five processors:

Our code now takes about 60s.
We have sped it up by about 40%.

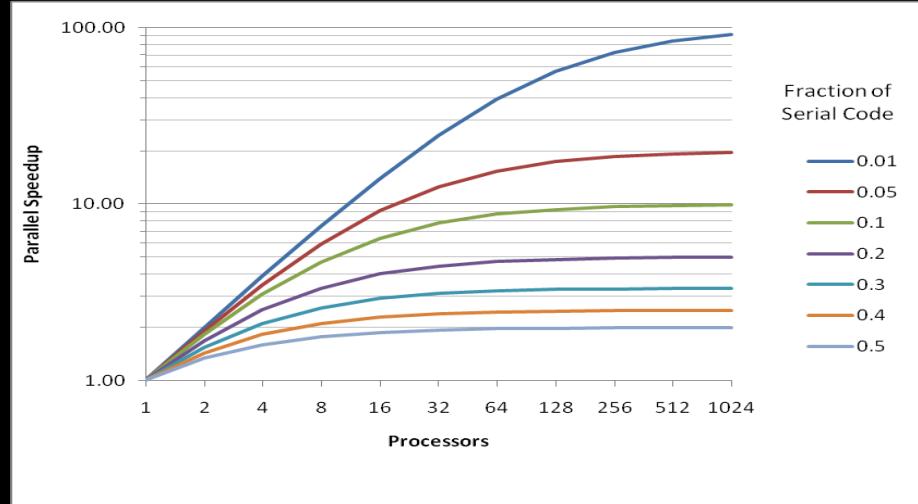
Let's say we use a thousand processors:

We have now sped our code by about a factor of two. Is this a big enough win?



Amdahl's Law

- If there is x% of serial component, speedup cannot be better than $100/x$.
- If you decompose a problem into many parts, then the parallel time cannot be less than the largest of the parts.
- If the critical path through a computation is T, you cannot complete in less time than T, no matter how many processors you use .



- Amdahl's law used to be cited by the knowledgeable as a limitation.
- These days it is mostly raised by the uninformed.
- Massive scaling is commonplace:
 - Science Literature
 - Web (map reduce everywhere)
 - Data Centers (Spark, etc.)
 - Machine Learning (GPUs and others)

Need to write some scalable code?

First Choice:

Pick a language - or maybe a library, or paradigm
(whatever that is)?

Languages: Pick One *(Hint: MPI + ?)*

Parallel Programming environments since the 90's

ABCPL	CORRELATE	GLU	Mentat	Paraphrase2	pC++
ACE	CPS	GUARD	Legion	Paralation	SCHEDULE
ACT++	CRL	HasL	Meta Chaos	Parallel-C++	SciTL
Active messages	CSP	Haskell	Midway	Parallaxis	POET
Adl	Cthreads	HPC++	Millipede	ParC	SDDA.
Adsmith	CUMULVS	JAVAR	CparPar	ParLib++	SHMEM
ADDAP	DAGGER	HORUS	Mirage	ParLin	SIMPLE
AFAPI	DAPPLE	HPC	MpC	Parmacs	Sina
ALWAN	Data Parallel C	IMPACT	MOSIX	Parti	SISAL.
AM	DC++	ISIS.	Modula-P	pC	distributed smalltalk
AMDC	DCE++	JAVAR	Modula-2*	pC++	SML.
AppLeS	DDD	JADE	Multipol	PCN	SONIC
Amoeba	DICE.	Java RMI	MPI	PCP:	Split-C.
ARTS	DIPC	javaPG	MPC++	PH	SR
Athapascan-0b	DOLIB	JavaSpace	Munin	PEACE	SThreads
Aurora	DOME	JIDL	Nano-Threads	PCU	Strand.
Automap	DOSMOS.	Joyce	NESL	PET	SUIF.
bb_threads	DRL	Khoros	NetClasses++	PETSc	Synergy
Blaze	DSM-Threads	Karma	Nexus	PENNY	Telegrphos
BSP	Ease .	KOAN/Fortran-S	Nimrod	Phosphorus	SuperPascal
BlockComm	ECO	LAM	NOW	POET.	TCGMSG.
C*.	Eiffel	Lilac	Objective Linda	Polaris	Threads.h++.
"C* in C	Eilean	Linda	Occam	POOMA	TreadMarks
C**	Emerald	JADA	Omega	POOL-T	TRAPPER
CarlOS	EPL	WWWinda	OpenMP	PRESTO	uC++
Cashmere	Excalibur	ISETL-Linda	Orca	P-RIO	UNITY
C4	Express	ParLin	OOF90	Prospero	UC
CC++	Falcon	Eilean	P++	Proteus	V
Chu	Filaments	P4-Linda	P3L	QPC++	ViC*
Charlotte	FM	Glenda	p4-Linda	PVM	Visifold V-NUS
Charm	FLASH	POSYBL	Pablo	PSI	VPE
Charm++	The FORCE	Objective-Linda	PADE	PSDM	Win32 threads
Cid	Fork	LiPS	PADRE	Quake	WinPar
Cilk	Fortran-M	Locust	Panda	Quark	WWWinda
CM-Fortran	FX	Lparx	Papers	Quick Threads	XENOOPS
Converse	GA	Lucid	AFAPI.	Sage++	XPC
Code	GAMMA	Maisie	Para++	SCANDAL	Zounds
COOL	Glenda	Manifold	Paradigm	SAM	ZPL

Paradigm?

- Message Passing
 - MPI
- Data Parallel
 - Fortran90
- Threads
 - OpenMP, OpenACC, CUDA
- PGAS
 - UPC, Coarray Fortran
- Frameworks
 - Charm++
- Hybrid
 - MPI + OpenMP

Message Passing: MPI in particular

Pros

- Has been around a longtime (~20 years inc. PVM)
- Dominant
- Will be around a longtime (on all new platforms/roadmaps)
- Lots of libraries
- Lots of algorithms
- Very scalable (100K+ cores right now)
- Portable
- Works with hybrid models
- We teach MPI in two days also
- **This is the only route to massive scalability today!**

Cons

- Lower level means more detail for the coder
- Debugging requires more attention to detail
- Domain decomposition and memory management must be explicit
- Students leaving our MPI workshop may face months of work before they are able to actually run their production code
- **Development usually requires a “start from scratch” approach**

Data Parallel – Fortran90

Computation in FORTRAN 90

P	P	P	P
P	P	P	P
P	P	P	P
P	P	P	P

```
Real A(4,4), B(4,4), C(4,4)  
  
A=2.0  
FORALL (I=1:4, J=1:4)  
    B(I,J)=I+J  
C=A+B
```

P = Processor

$$C = \begin{matrix} & \mathbf{A} & + & \mathbf{B} & \\ \begin{matrix} 4 & 5 & 6 & 7 \\ 5 & 6 & 7 & 8 \\ 6 & 7 & 8 & 9 \\ 7 & 8 & 9 & 10 \end{matrix} & \begin{matrix} 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \end{matrix} & + & \begin{matrix} 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 \\ 4 & 5 & 6 & 7 \\ 5 & 6 & 7 & 8 \end{matrix} \end{matrix}$$

Data Parallel

Communication in FORTRAN 90

P	P	P	P
P	P	P	P
P	P	P	P
P	P	P	P

P = Processor

```
Real A(4,4), B(4,4)  
  
FORALL (I=1:4, J=1:4)  
  B(I,J)=I+J  
A=CSHIFT (B, DIM=2, 1)
```

A=

3	4	5	6
4	5	6	7
5	6	7	8
2	3	4	5

CSHIFT (B, 2, 1)

2	3	4	5
3	4	5	6
4	5	6	7
5	6	7	8

Data Parallel

Pros

- So simple you just learned some of it
- ...or already knew it from using Fortran
- Easy to debug

Cons

- If your code doesn't totally, completely express itself as nice array operations, you are left without a flexible alternative.
 - Image processing: Great
 - Irregular meshes: Not so great

Threads in OpenMP

Fortran:

```
!$omp parallel do
do i = 1, n
    a(i) = b(i) + c(i)
enddo
```

C/C++:

```
#pragma omp parallel for
for(i=1; i<=n; i++)
    a[i] = b[i] + c[i];
```

Threads in OpenACC

SAXPY in C

```
void saxpy(int n,
           float a,
           float *x,
           float *restrict y)
{
#pragma acc kernels
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}

...
// Somewhere in main
// call SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
...
```

SAXPY in Fortran

```
subroutine saxpy(n, a, x, y)
    real :: x(:), y(:), a
    integer :: n, i
!$acc kernels
    do i=1,n
        y(i) = a*x(i)+y(i)
    enddo
!$acc end kernels
end subroutine saxpy

...
$ From main program
$ call SAXPY on 1M elements
call saxpy(2**20, 2.0, x_d, y_d)
...
```

Threads without directives: CUDA

```
// Host code
int main(int argc, char** argv)
{
    // Allocate input vectors in host memory
    h_A = (float*)malloc(size);
    if (h_A == 0) Cleanup();
    h_B = (float*)malloc(size);
    if (h_B == 0) Cleanup();
    h_C = (float*)malloc(size);
    if (h_C == 0) Cleanup();

    // Initialize input vectors
    Init(h_A, N);
    Init(h_B, N);

    // Allocate vectors in device memory
    cudaMalloc((void**)& d_A, size);
    cudaMalloc((void**)& d_B, size);
    cudaMalloc((void**)& d_C, size);

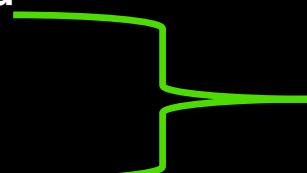
    // Copy vectors to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Run kernel
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

    // Copy results from device memory to host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
}

// GPU Code
__global__ void VecAdd(const float* A, const float* B, float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}
```

Some things we did not mention

- OpenCL (Khronos Group)
 - Everyone supports, but not as a primary focus
 - Intel – OpenMP
 - NVIDIA – CUDA, OpenACC
 - AMD – now HSA (hUMA/APU oriented)
 - Fortran 2008+ threads (sophisticated but not consistently implemented)
 - C++11 threads are basic (no loops) but better than POSIX
 - Python threads are fake (due to Global Interpreter Lock)
 - DirectCompute (Microsoft) is not HPC oriented
 - C++ AMP (MS/AMD)
 - TBB (Intel C++ template library)
 - Cilk (Intel, now in a gcc branch)
- 
- Very C++ for threads

Threads

Splits up tasks (as opposed to arrays in data parallel) such as loops amongst separate processors.

Do communication as a side effect of data loop distribution. Not an big issue on shared memory machines. Impossible on distributed memory.

Common Implementations:

 pthreads (original Unix standard)

 OpenMP

 OpenACC

 OpenCL (Khronos Group)

 DirectCompute (Microsoft)

Very C++ oriented:

- C++ AMP (MS/AMD)
- TBB (Intel C++ template library)
- Cilk (Intel, now in a gcc branch)

Pros:

1. Doesn't perturb data structures, so can be incrementally added to existing serial codes.
2. Becoming fairly standard for compilers.

Cons:

1. Serial code left behind will be hit by Amdahl's Law
2. Forget about taking this to the next level of scalability. You can not do this on MPPs at the machine wide level.

PGAS with Co-Array Fortran (now Fortran 2008)

**Co-array synchronization is at the heart of the typical Co-Array Fortran program.
Here is how to exchange an array with your north and south neighbors:**

```
COMMON/XCTILB4/ B(N,4) [*]  
SAVE /XCTILB4/  
  
CALL SYNC_ALL( WAIT=(/IMG_S,IMG_N/) )  
B(:,3) = B(:,1)[IMG_S]  
B(:,4) = B(:,2)[IMG_N]  
CALL SYNC_ALL( WAIT=(/IMG_S,IMG_N/) )
```

Lots more examples at co-array.org.

Partitioned Global Address Space: (PGAS)

Multiple threads share at least a part of a global address space.

Can access local and remote data with same mechanisms.

Can distinguish between local and remote data with some sort of typing.

Variants:

Co-Array Fortran (CAF)

Fortran 2008

Unified Parallel C (UPC)

Pros:

1. Looks like SMP on a distributed memory machine.
2. Currently translates code into an underlying message passing version for efficiency.

Cons:

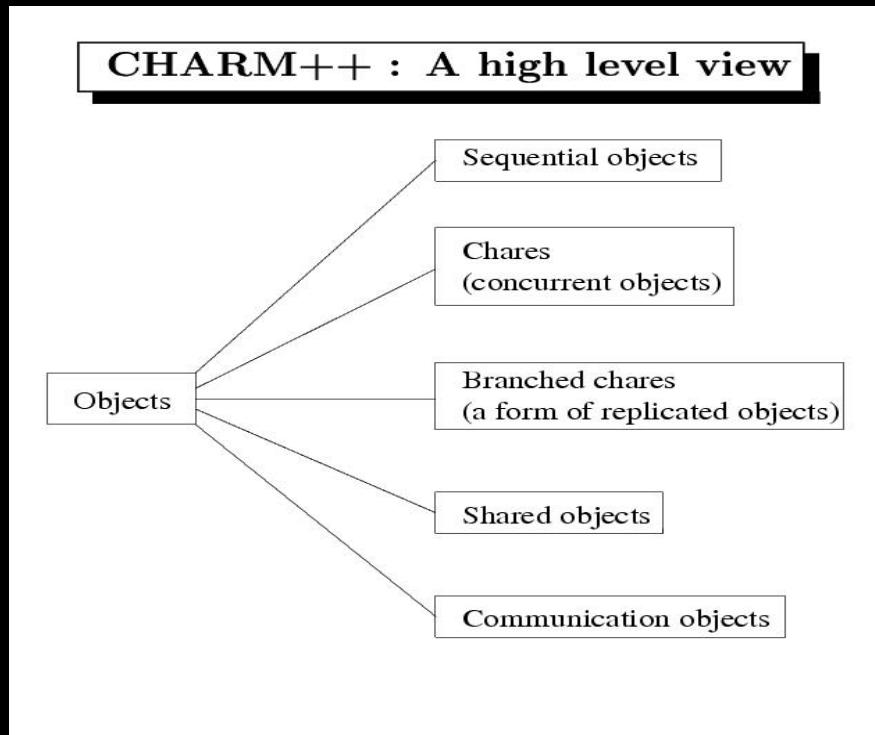
1. Depends on (2) to be efficient.
2. Can easily write lots of expensive remote memory access without paying attention.
3. Currently immature.

Frameworks

One of the more experimental approaches that was gaining some traction was to use a parallel framework that handle the load balancing and messaging while you “fill in” the science. Charm++ is the most popular example:

Charm++

- Object-oriented parallel extension to C++
- Run-time engine allows work to be “scheduled” on the computer.
- Highly-dynamic, extreme load-balancing capabilities.
- Completely asynchronous.
- NAMD, a very popular MD simulation engine is written in Charm++



Frameworks (Newsflash!)

- After a long time with no positive reports in this space, I can definitely say that the Machine Learning (Artificial Intelligence) community has embraced this in an effective manner.
- The most popular frameworks/toolkits/packages used for deep learning (aka Neural Nets) are very much in this philosophy.
- Caffe, TensorFlow and others use a high level descriptive approach to arrange other components, often themselves a higher level layer in Python or whatnot, to invoke libraries written in C++ (and actually Fortran is hidden in there more often than those groups would believe in the form of BLAS type libraries).
- These frameworks use threads, GPUs and distributed nodes very heavily.
- You could say that the math library nature of this work makes this unique, but the innovation in arranging these codeflows is not at all rote.

Hybrid Coding

- Problem: given the engineering constraint of a machine made up of a large collection of multi-core processors, how do we use message passing at the wide level while still taking advantage of the local shared memory?
- Similar Problem: given a large machine with accelerators on each node (GPU or MIC), how do we exploit this architecture?
- Solution: Hybrid Coding. Technically, this could be any mix of paradigms. Currently, this is likely MPI with a directive based approach mixed in.
- At the node level, you may find OpenMP or OpenACC directives most usable.
- But, ***one must design the MPI layer first, and then apply the OpenMP/ACC code at the node level.*** The reverse is rarely a viable option.

Counterintuitive: MPI vs. OpenMP on a node

It might seem obvious that, since OpenMP is created to deal with SMP code, you would ideally like to use that at the node level, even if you use MPI for big scalability across an MPP.

Very often, it turns out that the MPI-to-the-core (pun completely intended) version is faster. This indeed seems odd.

The answer is that after going to the trouble of doing a proper MPI data decomposition, you have also greatly aided the caching mechanism (by moving concurrently accessed data into different regions of memory). Hence the win.

However, if you are only interested in node-level scaling, this would be a lot of effort.

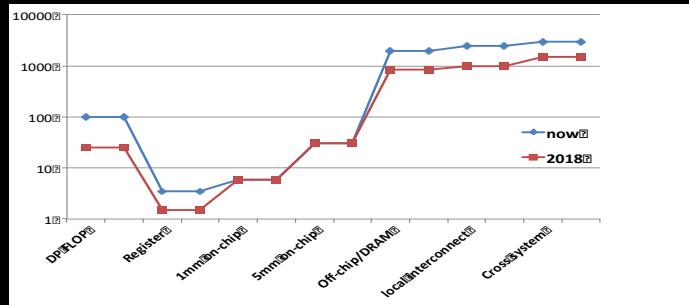
MPI as an answer to an emerging problem ?!

In the Intro we mentioned that we are at a historic crossover where the cost of even on-chip data movement is surpassing the cost of computation.

MPI codes explicitly acknowledge and manage data locality and movement (communication).

Both this paradigm, and quite possibly MPI, offer the only immediate solution. You and your programs may find a comfortable future in the world of massive multi-core.

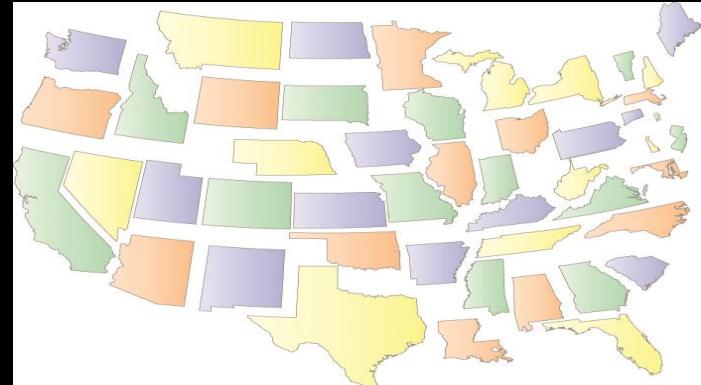
This is a somewhat recent realization in the most avant-garde HPC circles. Amaze your friends with your insightful observations!



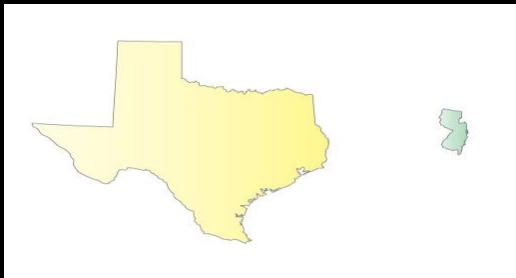
Parallel Programming in a Nutshell

Assuming you just took our workshop

- You have to spread *something* out.
- These can theoretically be many types of abstractions: work, threads, tasks, processes, data,...
- But what they *will* be is your data. And then you will use MPI, and possibly OpenMP/ACC, to operate on that data.

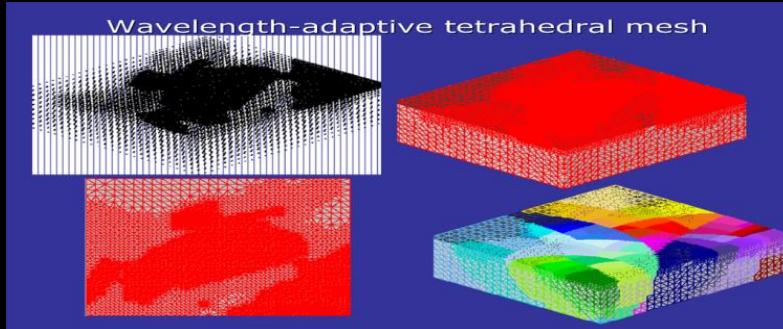


Domain Decomposition Done Well: Load Balanced



Is Texas vs. New Jersey a good idea?

- A parallel algorithm can only be as fast as the slowest chunk.
 - Might be dynamic (hurricane moving up coast)
- Communication will take time
 - Usually orders of magnitude difference between registers, cache, memory, network/remote memory, disk
 - Data locality and “neighborly-ness” matters very much.



A Few Scalability Hints

- ~~Minimize~~ Eliminate serial sections of code
 - Only Way To Beat Amdahl's law
- Minimize communication overhead
 - Choose algorithms that emphasize nearest neighbor communication
 - Possibly Overlap computation and communication with asynchronous communication models
- Dynamic load balancing (at least be aware of issue)
- Minimize I/O and learn how to use parallel I/O
 - Very expensive time wise, so use sparingly (and always binary)
- Choose the right language for the job!
- *Plan out your code beforehand.*
 - *Because the above won't just happen late in development*
 - *Transforming a serial code to parallel is rarely the best strategy*

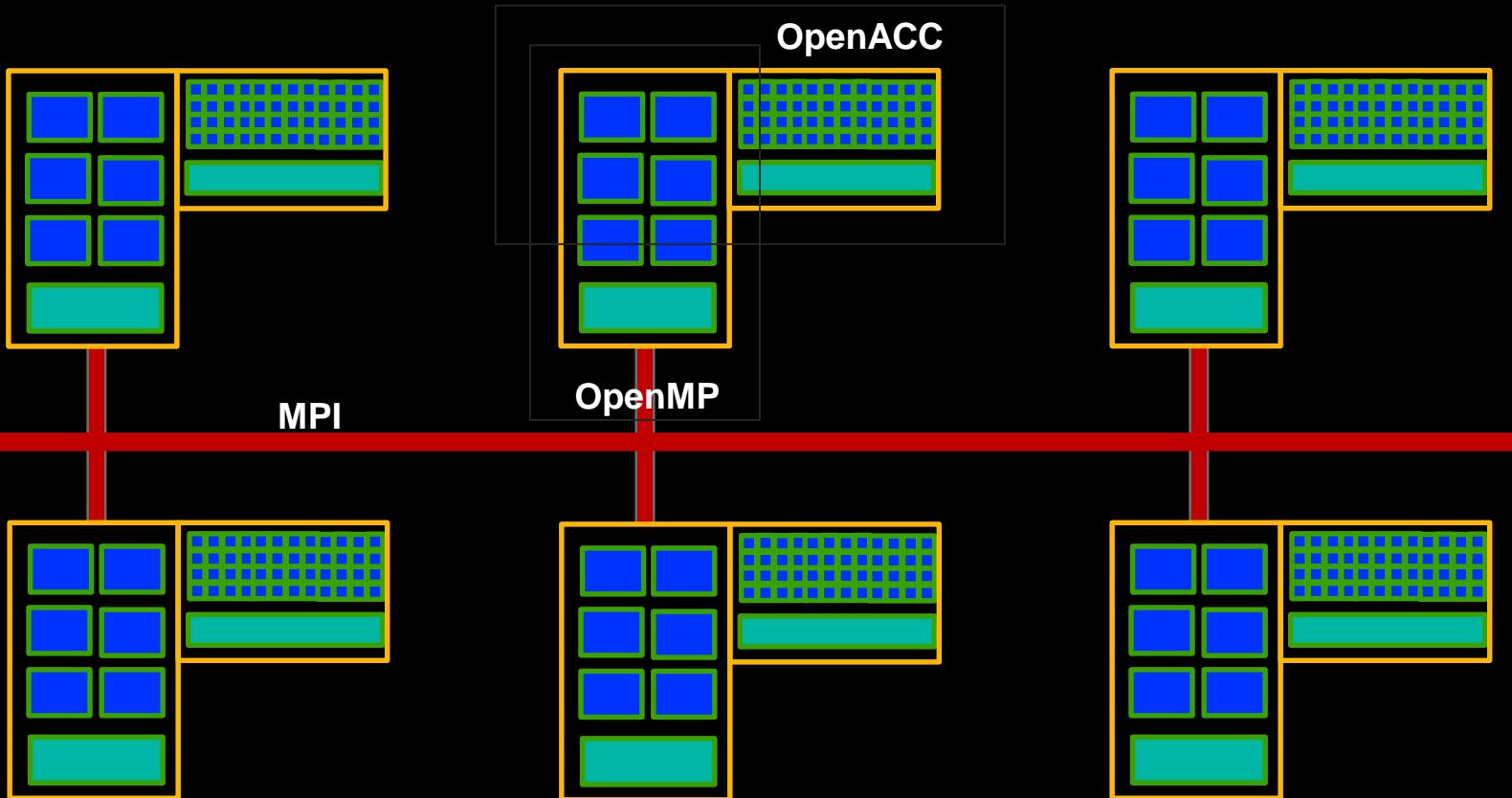
Summary

- **Hardware drives our software options:**
 - Serial boxes can't get to petaFLOPs (let alone exaFLOPS)
 - Moore's Law OK, but resulting power dissipation issue is the major limiting factor
 - Multiple processors are the one current end-run around this issue
 - This won't change any time in the foreseeable future
 - So parallel programming we will go...
- **Software options are many:**
 - Reality has chosen a few winners
 - You have learned the important ones

Summary (of entire workshop, really)

- Still mostly up to you *if you want to scale beyond a few processors*
 - Automatic parallelization has been “a few years away” for the past 30 years.
- Dozens of choices
 - But really only MPI (*with maybe OpenMP/OpenACC*)

In Conclusion...



The Future and where you fit.

While the need is great, there is only a short list of serious contenders for 2020 exascale computing usability.

MPI 3.0 +X

(MPI 3.0 specifically addresses exascale computing issues)

PGAS (partitioned global address space)

CAF (now in Fortran 2008!), UPC

APGAS

X10, Chapel

Frameworks

Charm++

Functional

Haskell

