Outro To Parallel Computing

John Urbanic
Pittsburgh Supercomputing Center
Parallel Computing Specialist

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…

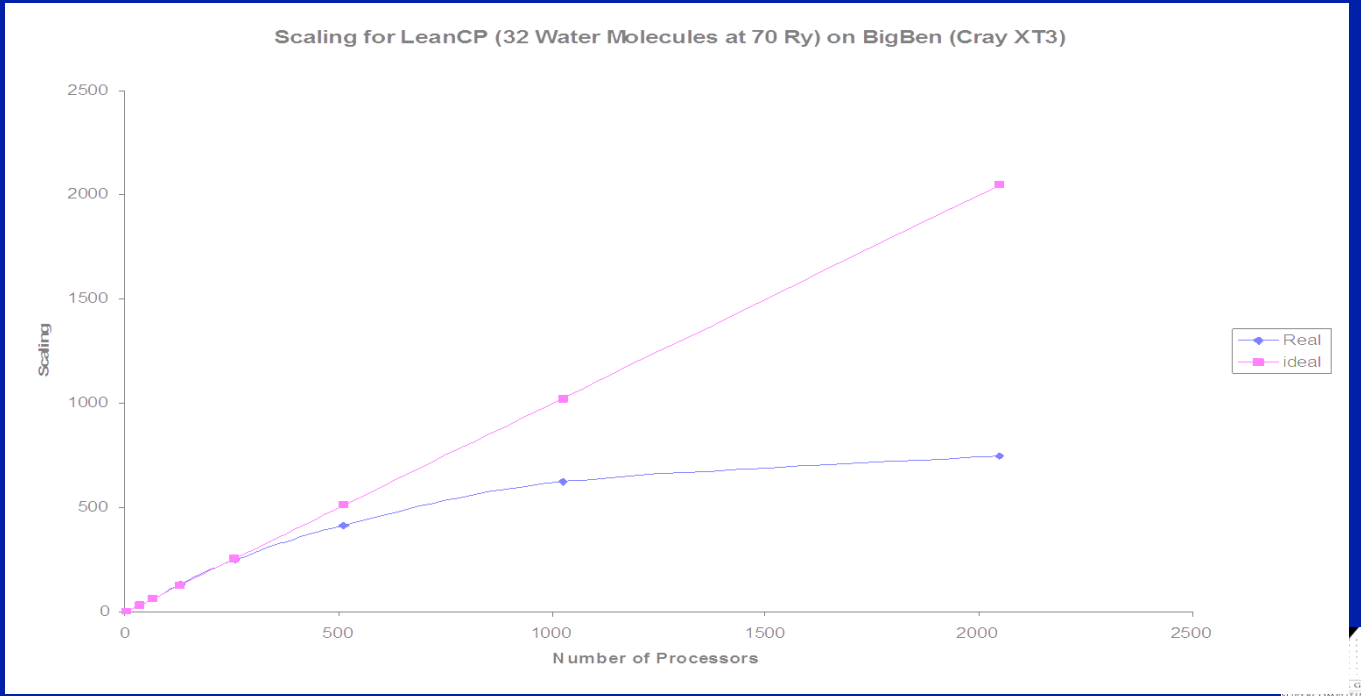
- An example
- Scaling
- Amdahl's Law
- Languages and Paradigms
  - Message Passing
  - Data Parallel
  - Threads
  - PGAS
  - Hybrid
- Data Decomposition
- Load Balancing
- Summary

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

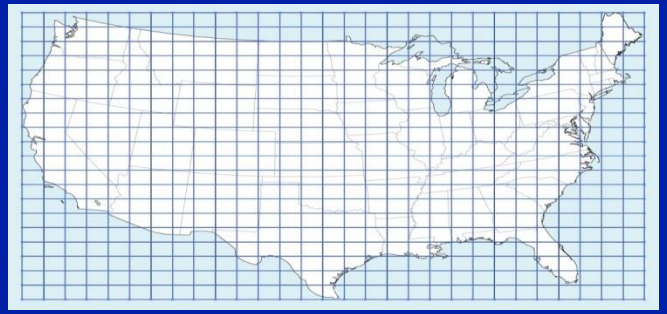
Parallel scaling of liquid water* as a function of system size on the Blue Gene/L installation at YKT:

CO Mode Native Layer with Optimizations											
Nodes	32	64	128	256	512	1024	2048	4096	8192	16384	20480
Processors	32	64	128	256	512	1024	2048	4096	8192	16384	20480
W8 Time s/step	0.22	0.10	0.082	0.071	0.046	0.026	0.020				
W16 Time s/step	0.73	0.40	0.23	0.15	0.106	0.061	0.041	0.035			
W32 Time s/step	2.71	1.52	0.95	0.44	0.26	0.15	0.11	0.081	0.063		
W64 Time s/step		6.72	3.77	1.88	0.87	0.51	0.31	0.21	0.15		
W128 Time s/step					7.4	3.31	1.57	1.09	0.581	0.425	
W256 Time s/step						21.1	14.3	7.64	3.54	2.09	1.90

*Liquid water has 4 states per molecule.

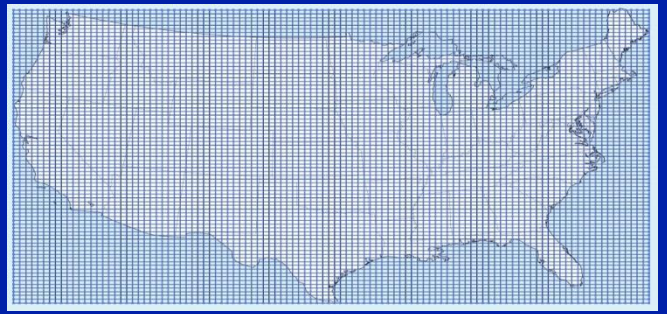
- Weak scaling is observed!
- Strong scaling on processor numbers up to ~60x the number of states!

Weak vs. Strong scaling

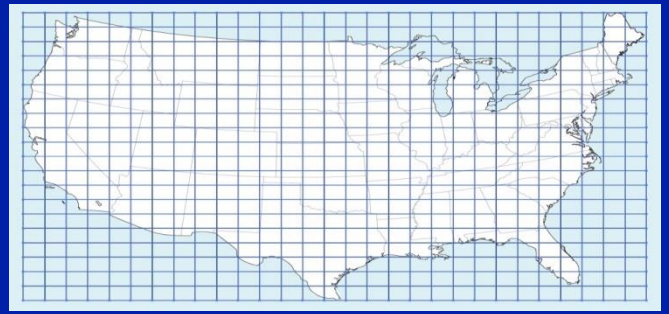


More Processors

Weak Scaling

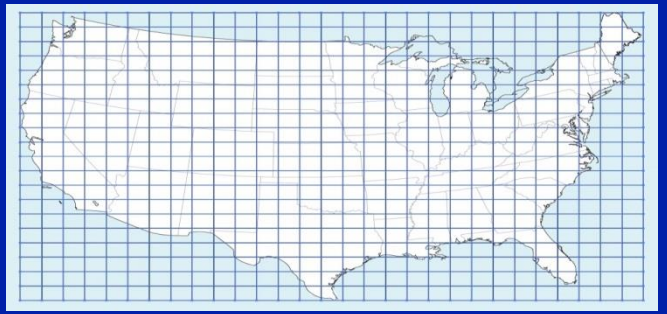


More accurate results



More Processors

Strong Scaling

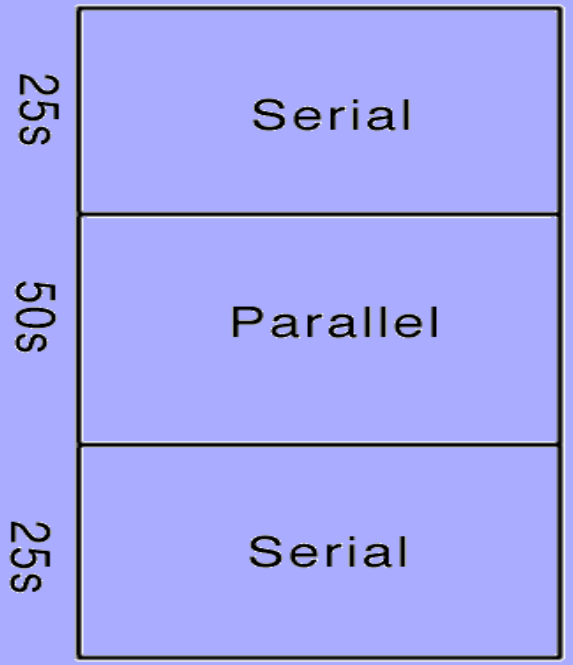


Faster results
(Tornado on way!)

Your Scaling Enemy: Amdahl's Law

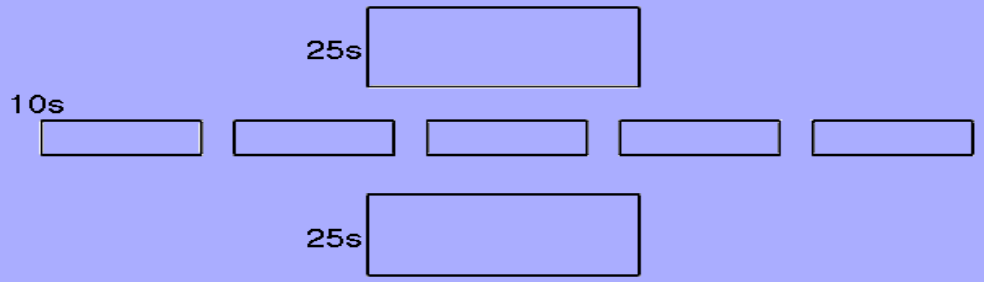
How many processors can we really use?

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



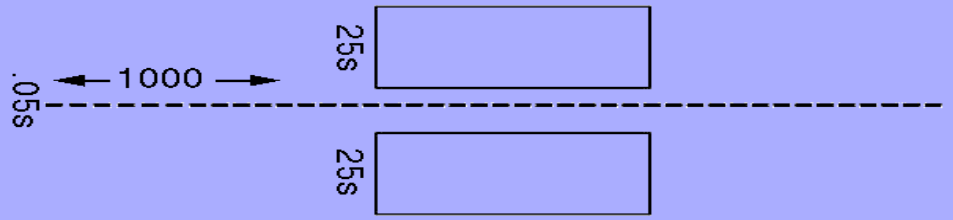
Amdahl's Law

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%.

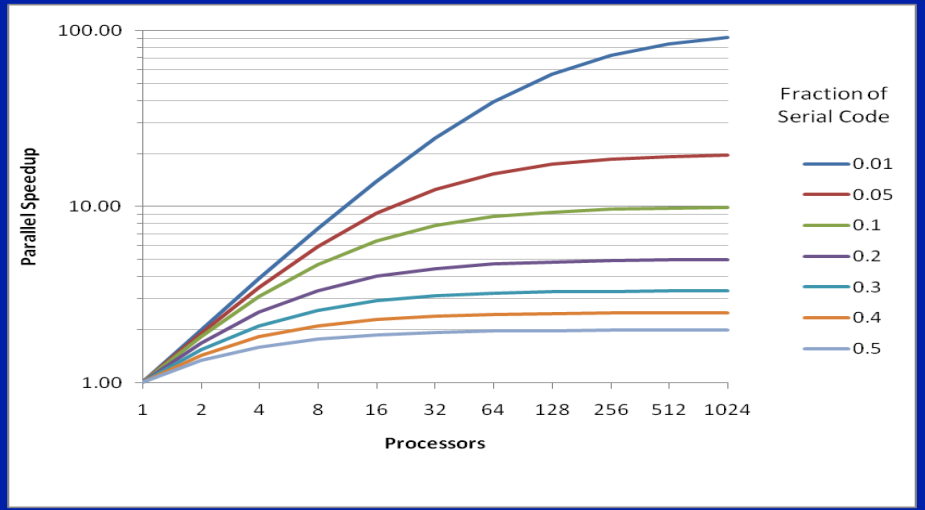
Let's say we use a thousand processors:



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 .



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
ACE
ACT++
Active messages
Adl
Adsmith
ADDAP
AFAPI
ALWAN
AM
AMDC
AppLeS
Amoeba
ARTS
Athapascan-0b
Aurora
Automap
bb_threads
Blaze
BSP
BlockComm
C*.
"C* in C
C**
CarlOS
Cashmere
C4
CC++
Chu
Charlotte
Charm
Charm++
Cid
Cilk
CM-Fortran
Converse
Code
COOL

CORRELATE
CPS
CRL
CSP
Cthreads
CUMULVS
DAGGER
DAPPLE
Data Parallel C
DC++
DCE++
DDD
DICE.
DIPC
DOLIB
DOME
DOSMOS.
DRL
DSM-Threads
Ease .
ECO
Eiffel
Eilean
Emerald
EPL
Excalibur
Express
Falcon
Filaments
FM
FLASH
The FORCE
Fork
Fortran-M
FX
GA
GAMMA
Glenda

GLU
GUARD
HAsL.
Haskell
HPC++
JAVAR.
HORUS
HPC
IMPACT
ISIS.
JAVAR
JADE
Java RMI
javaPG
JavaSpace
JIDL
Joyce
Khoros
Karma
KOAN/Fortran-S
LAM
Lilac
Linda
JADA
WWWinda
ISETL-Linda
ParLin
Eilean
P4-Linda
Glenda
POSYBL
Objective-Linda
LiPS
Locust
Lparx
Lucid
Maisie
Manifold

Mentat
Legion
Meta Chaos
Midway
Millipede
CparPar
Mirage
MpC
MOSIX
Modula-P
Modula-2*
Multipol
MPI
MPC++
Munin
Nano-Threads
NESL
NetClasses++
Nexus
Nimrod
NOW
Objective Linda
Occam
Omega
OpenMP
Orca
OOF90
P++
P3L
p4-Linda
Pablo
PADE
PADRE
Panda
Papers
AFAPI.
Para++
Paradigm

Parafrase2
Paralation
Parallel-C++
Parallaxis
ParC
ParLib++
ParLin
Parmacs
Parti
pC
pC++
PCN
PCP:
PH
PEACE
PCU
PET
PETSc
PENNY
Phosphorus
POET.
Polaris
POOMA
POOL-T
PRESTO
P-RIO
Prospero
Proteus
QPC++
PVM
PSI
PSDM
Quake
Quark
Quick Threads
Sage++
SCANDAL
SAM

pC++
SCHEDULE
SciTL
POET
SDDA.
SHMEM
SIMPLE
Sina
SISAL.
distributed smalltalk
SMI.
SONiC
Split-C.
SR
Sthreads
Strand.
SUIF.
Synergy
Telegrphos
SuperPascal
TCGMSG.
Threads.h++.
TreadMarks
TRAPPER
uC++
UNITY
UC
V
ViC*
Visifold V-NUS
VPE
Win32 threads
WinPar
WWWinda
XENOOPS
XPC
Zounds
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

P = Processor

```
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
```

C=

4	5	6	7
5	6	7	8
6	7	8	9
7	8	9	10

A

2	2	2	2
2	2	2	2
2	2	2	2
2	2	2	2

+

B

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

Data Parallel

Communication in FORTRAN 90

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];
}
```

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:

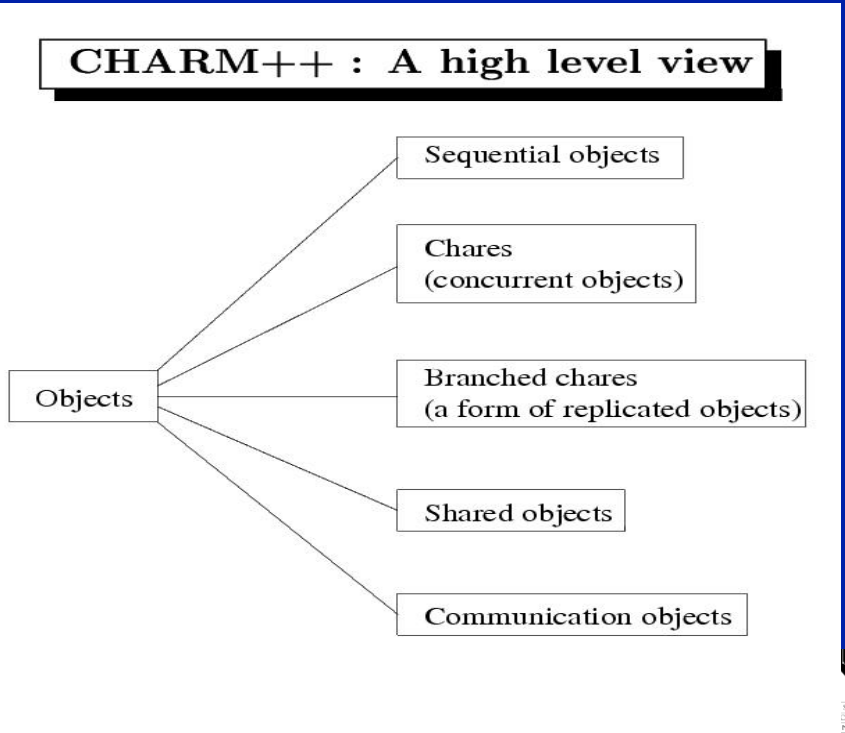
1. Depends on (2) to be efficient.
2. Can easily write lots of expensive remote memory access without paying attention.
3. Currently immature.

Frameworks

One of the more experimental approaches that is gaining some traction is to use a parallel framework that handles the load balancing and messaging while you “fill in” the science. Charm++ is a particularly 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++**

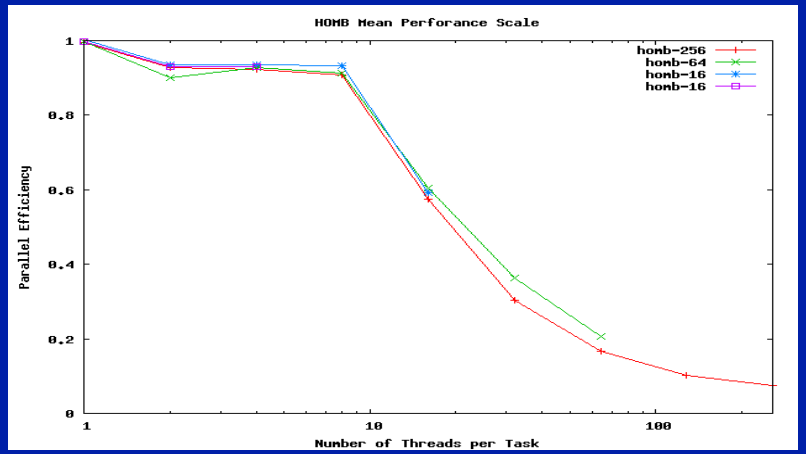


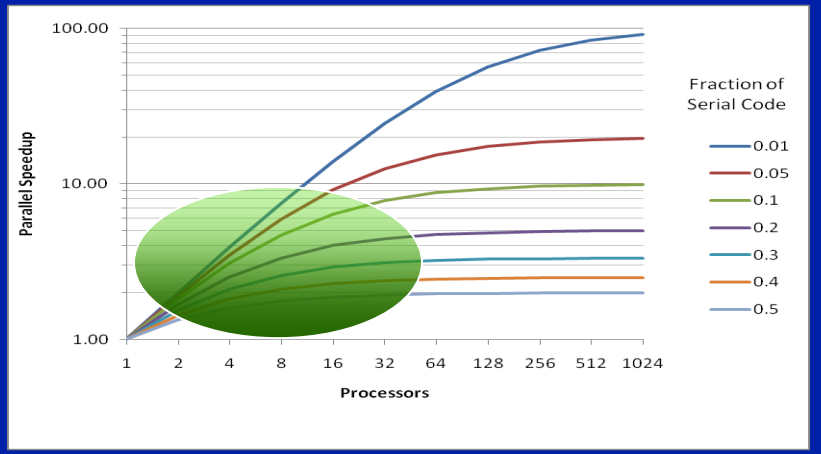
Hybrid Coding

- 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 them apply the OpenMP/ACC code at the node level***. The reverse is rarely a viable option.

Hybrid Expectations

- NUMA (or SMP node size) will impose a wall on the OpenMP/ACC border:

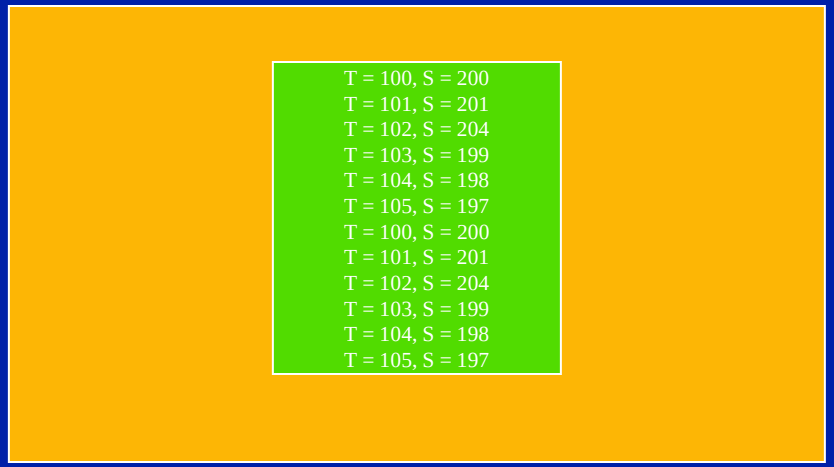




Candidate Hybrid Applications

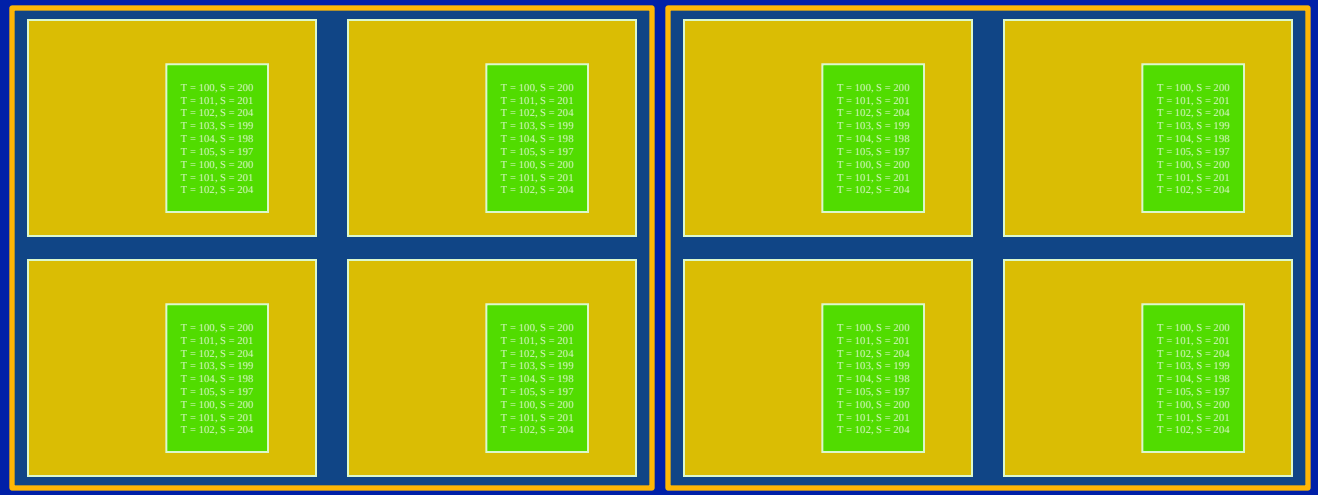
- Applications that need to use accelerators.
- Codes that benefit from the largest memory available per node.

Example: Code with a large lookup table, like an Equation of State table. Global variables are always evil, but we really need this large data structure accessible to every node.



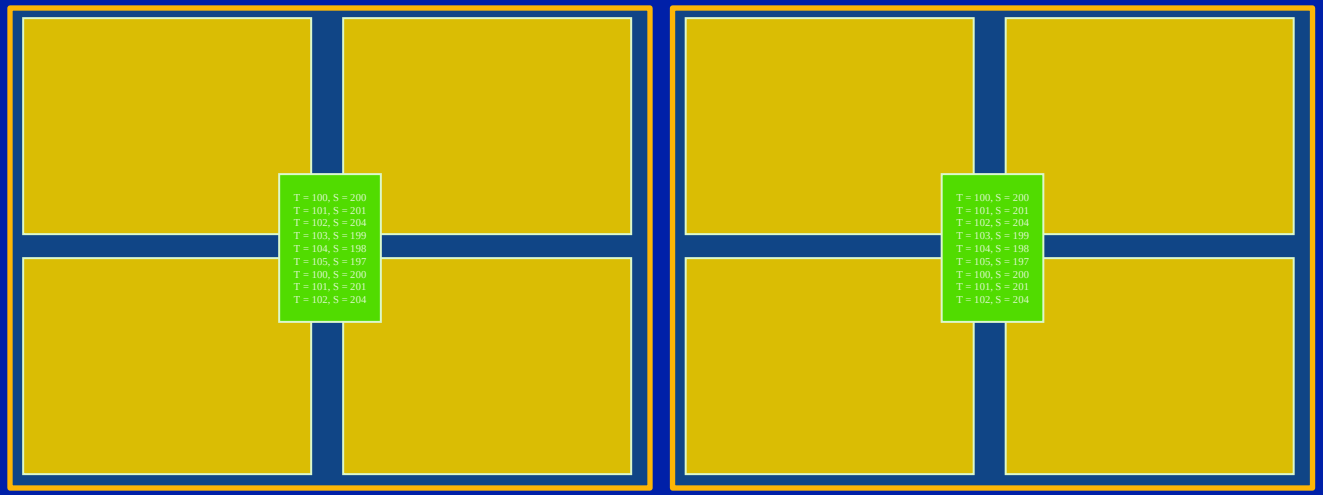
Good Hybrid Application

If we use straight MPI, then we end up duplicating this table on every PE.



Good Hybrid Application

With a hybrid approach, we can reduce this to one copy per node. A big win if the table size is significant.



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.

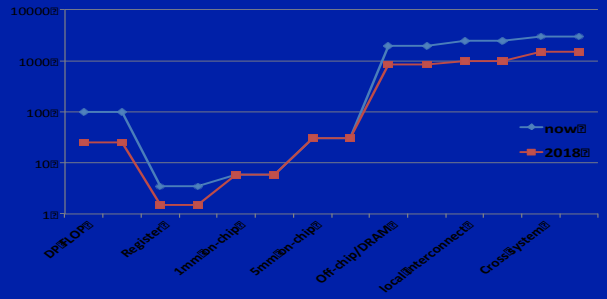
MPI as an answer to an emerging problem ?!

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 possible outright MPI, offer the only immediate solution. You and your programs may find a comfortable future in the world of massive multi-core.

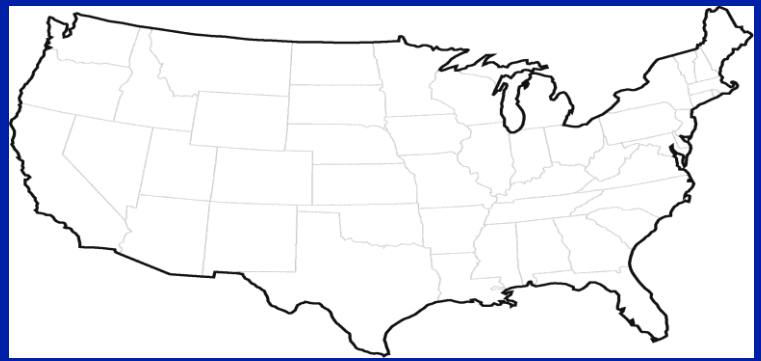
This is a somewhat recent realization in the most *avant-garde* HPC circles. Amaze your friends with your insightful observations!

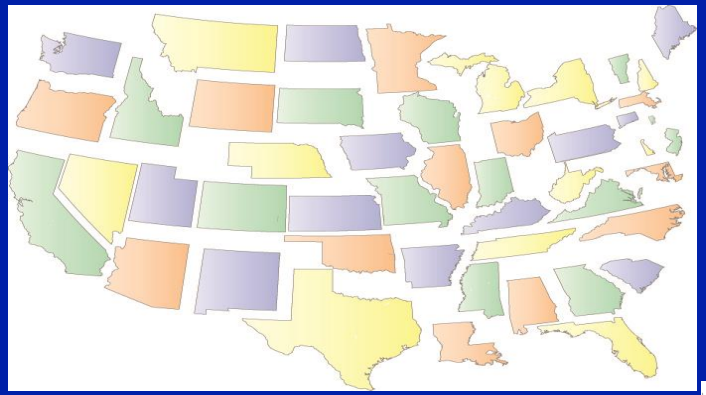


Parallel Programming in a Nutshell

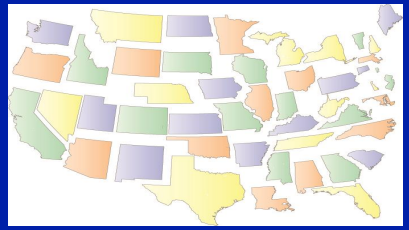
Assuming you just took our workshop

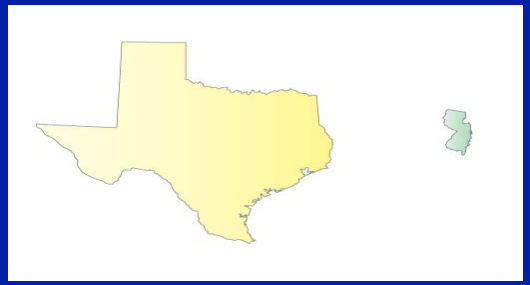
- You have to spread *something* out.
- These can theoretically be many types of abstractions: work, threads, tasks, processes, data,…
- But what they *will* be is your data. And then you will use MPI, and possibly OpenMP/ACC, to operate on that data.





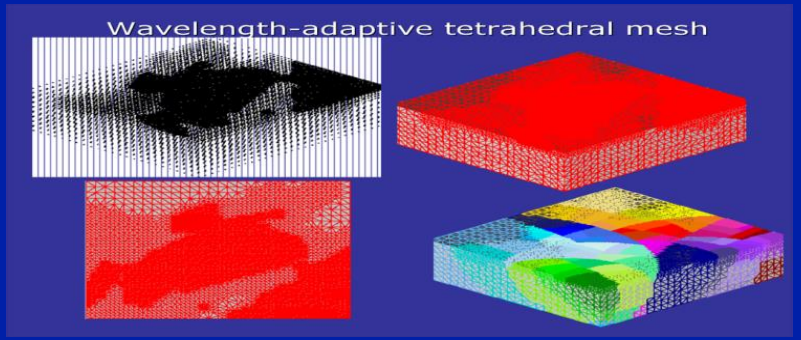
Domain Decomposition Done Well: Load Balanced





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 Parting Coding 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 are learning an important one of them
  - After we are done here, you will be able to explicitly understand the benefits and limitations compared to those other options

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 20 years.
- Dozens of choices
  - But really only MPI *(with maybe OpenMP/OpenACC)*

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