Brief Overview of Parallel Programming

M. D. Jones, Ph.D.

Center for Computational Research
University at Buffalo
State University of New York

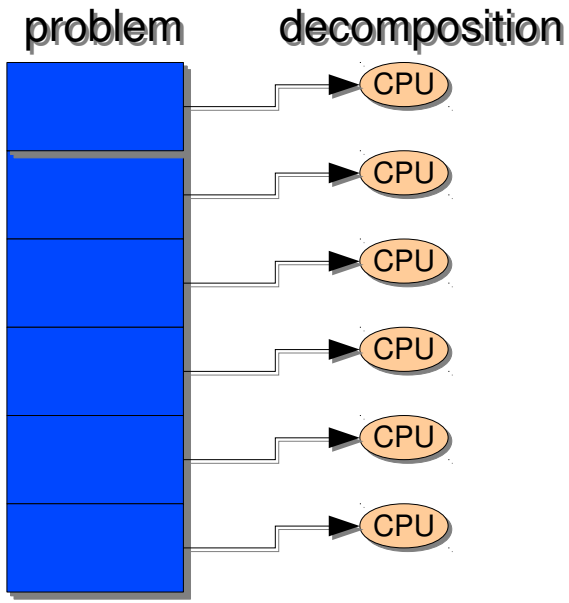
High Performance Computing I, 2013

A Little Motivation

- Why do we parallel process in the first place?
  - **Performance** - either solve a bigger problem, or to reach solution faster (or both)
- Hardware is becoming (actually it has been for a while now) intrinsically parallel
- Parallel programming is **not** easy. How easy is sequential programming? Now add another layer (a sizable one, at that) for parallelism ...

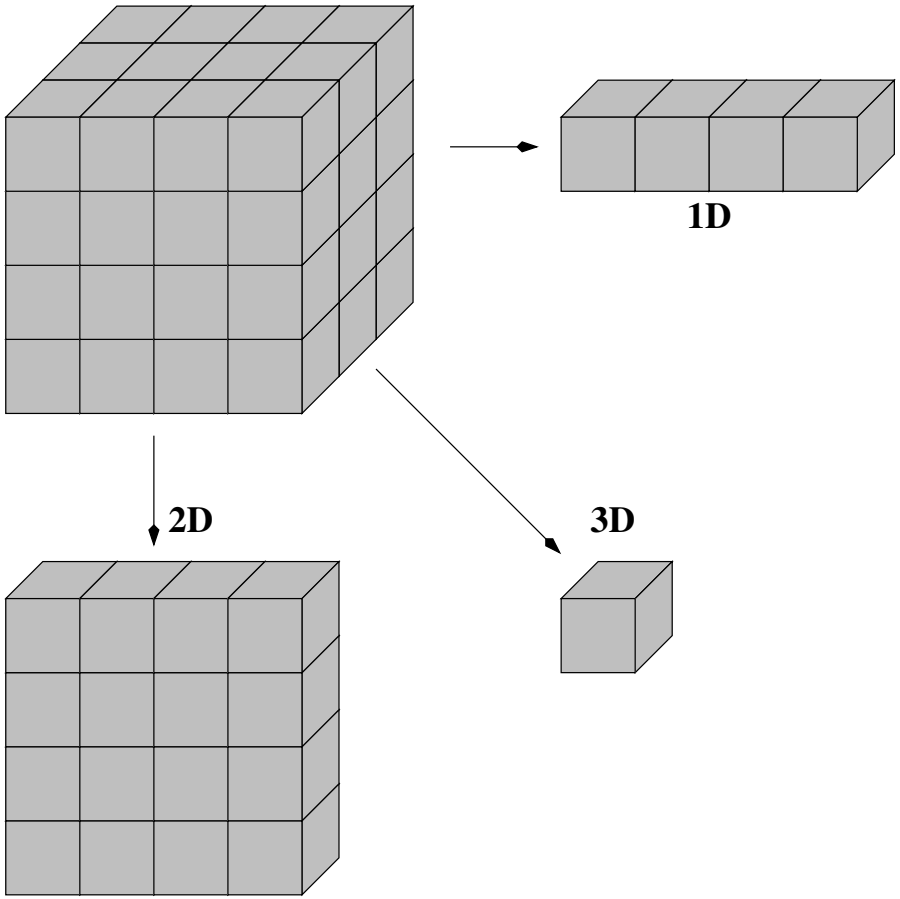
Big Picture

Basic idea of parallel programming is a simple one:



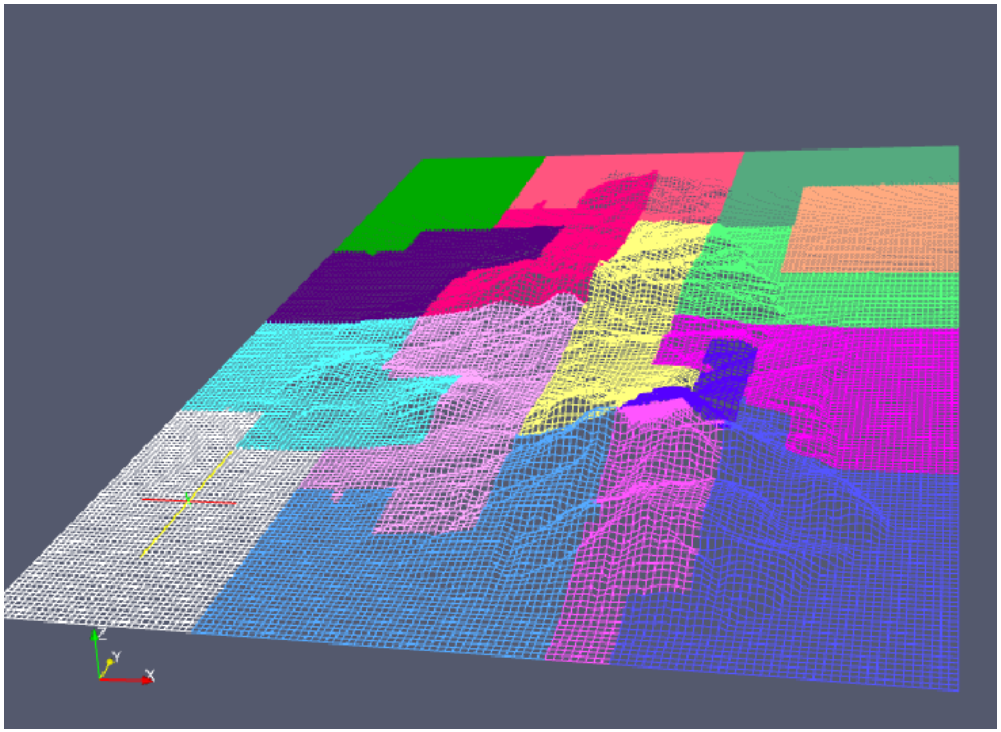
in which we decompose a large computational problem into available processing elements (CPUs). How you achieve that decomposition is where all the fun lies ...

Decomposition



Uniform volume example - decomposition in 1D, 2D, and 3D. Note that load balancing in this case is simpler if the work load per cell is the same ...

Decomposition (continued)



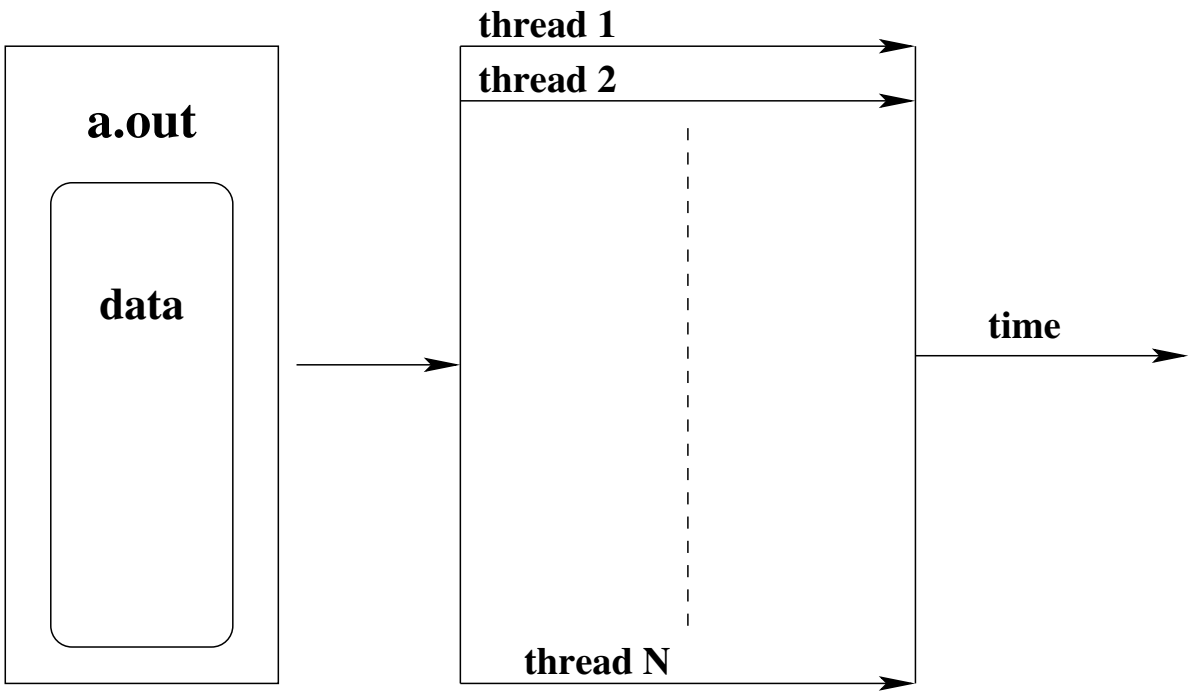
Nonuniform example - decomposition in 3D for a 16-way parallel adaptive finite element calculation (in this case using a terrain elevation). Load balancing in this case is much more complex.

The Parallel Zoo

There are many parallel programming models, but roughly they break down into the following categories:

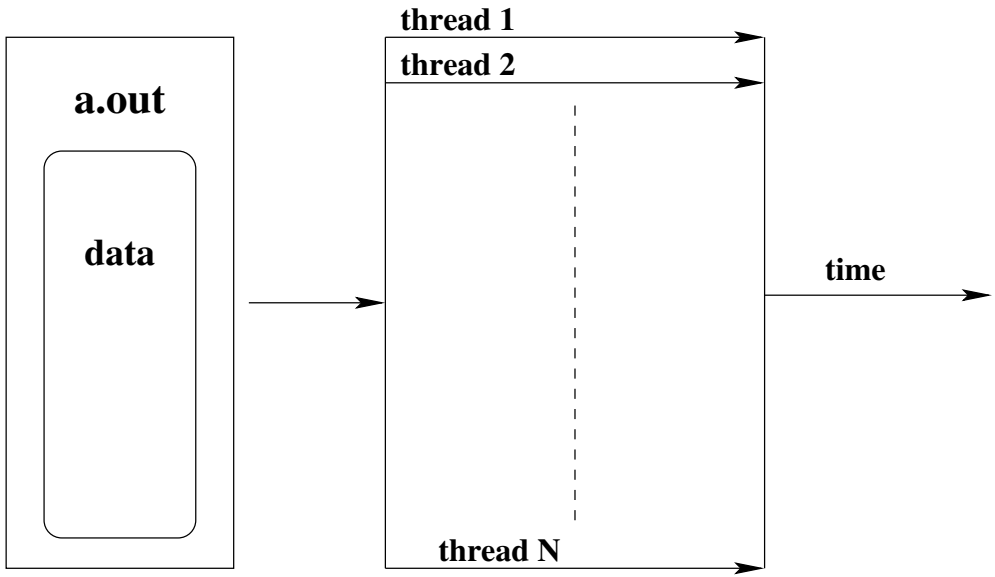
- Threaded models: e.g., **OpenMP** or `Posix` threads as the application programming interface (API)
- Message passing: e.g., **MPI** = Message Passing Interface
- Hybrid methods: combine elements of other models

Thread Models



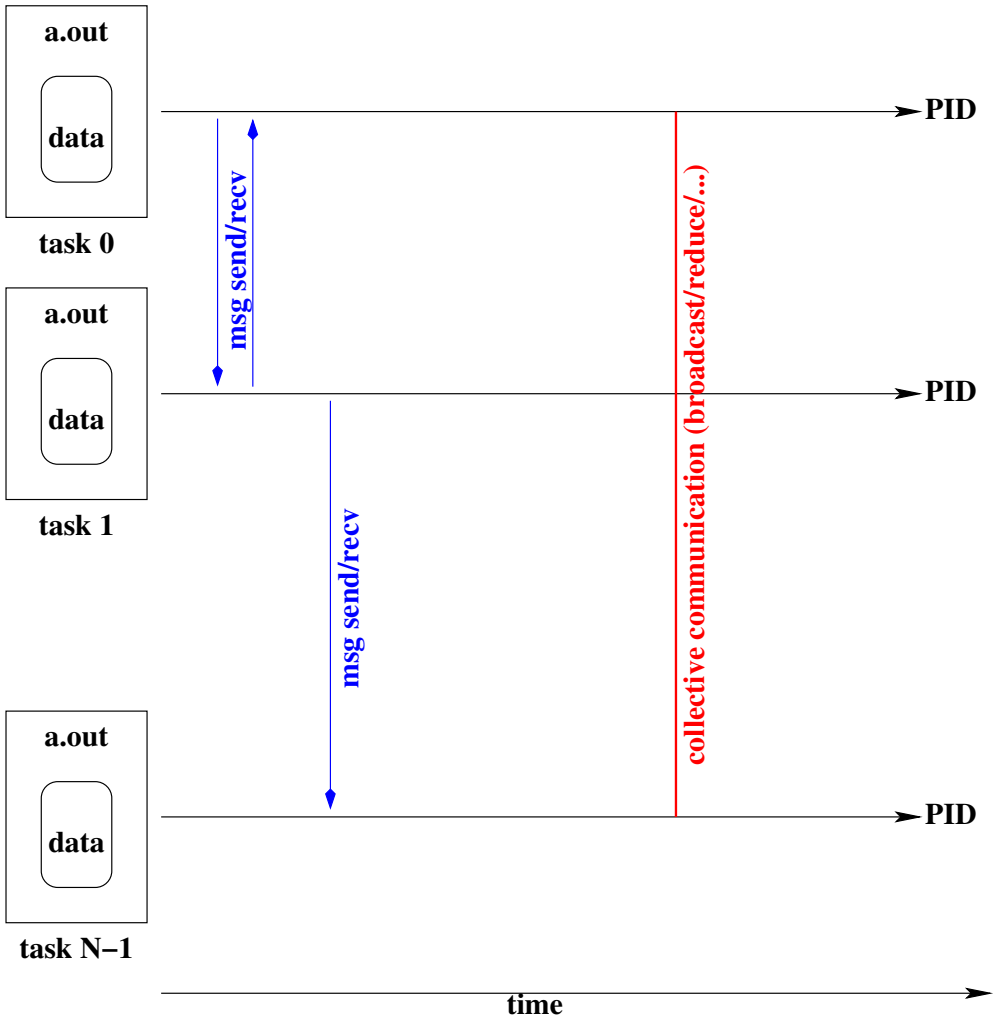
Typical thread model (e.g., **OpenMP**)

Thread Models



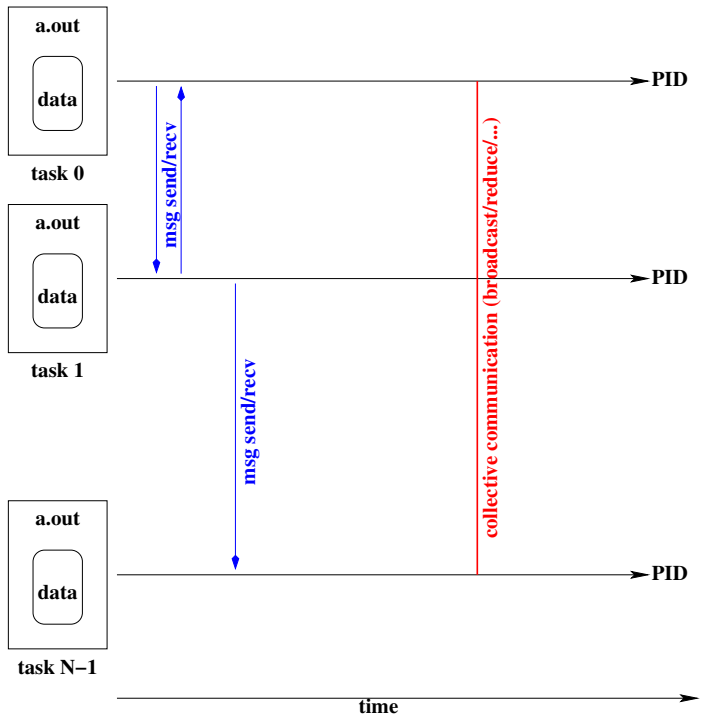
- **Shared** address space (all threads can access data of program **a.out**).
- Generally limited to single machine except in very specialized cases.
- GPGPU (general purpose GPU computing) uses thread model (with a large number of threads on the host machine.

Message Passing Models



Typical message passing model (e.g., **MPI**).

Message Passing Models



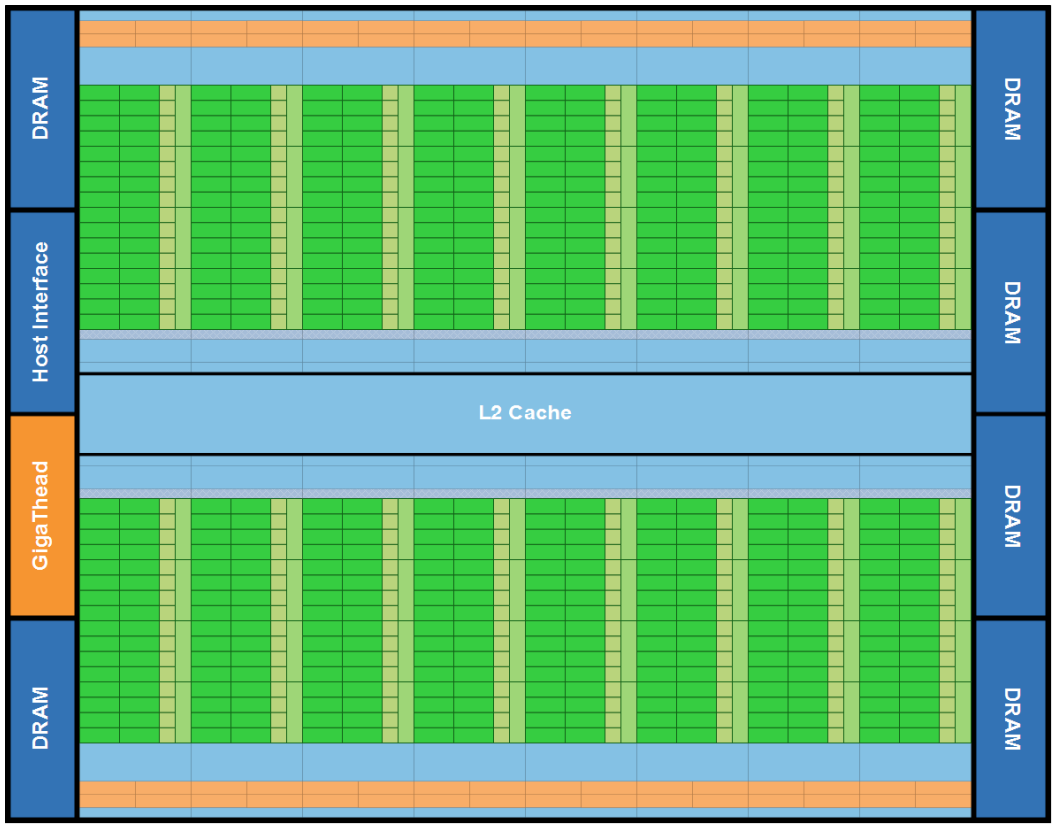
- **Separate** address space (tasks can not access data of other tasks without sending messages or participating in collective communications).
- General purpose model - messages can be sent over network, shared memory, etc.
- Managing communication is generally up to the programmer.

GPUs

Since the advent of programmable vertex/shader models in OpenGL (and then in DirectX), GPUs have become interesting from the point of view of scientific computation as general purpose compute engines (so called GPGPU computing):

- Massively data parallel (generally using thread model).
- May not support IEEE 754-2008 floating-point arithmetic standard (significant shortcoming for scientific computation).
- GPUs have been optimized for single precision arithmetic, double generally performs much worse (another shortcoming).
- ECC memory has not been generally used (ouch).
- All of these shortcomings are gradually being rectified ...

NVIDIA Fermi/GF100



Fermi's 16 SM are positioned around a common L2 cache. Each SM is a vertical rectangular strip that contain an orange portion (scheduler and dispatch), a green portion (execution units), and light blue portions (register file and L1 cache).

16 SMs, 32 Cuda cores/SM, full IEEE 754-2008, ECC memory

GPGPU Programming

With the hardware specifications still in a state of flux, it should be no surprise that the software situation is even worse:

- OpenGL and DirectX are graphics APIs, so are not compatible with scientific computing.
- Major GPU vendors have their own APIs for GPGPU programming, with little to no overlap or compatibility (CUDA for NVIDIA, CtoM for ATI).
- Some efforts to standardize, e.g. OpenCL, still very early and unproven.
- GPGPU programming is very similar to parallel HPC prior to PVM/MPI.

Parallelism at Three Levels

Task - macroscopic view in which an algorithm is organized around separate concurrent tasks

Data - structure data in (separately) update-able "chunks"

Instruction - ILP through the "flow" of data in a predictable fashion

Approaching the Problem

The first step in deciding where to add parallelism to an algorithm or application is usually to analyze from the point of **tasks** and **data**:

Tasks How do I reduce this problem into a set of tasks that can be executed concurrently?

Data How do I take the key data and represent it in such a way that large chunks can be operated on independently (and thus concurrently)?

Dependencies

Bear in mind that you always want to minimize the **dependencies** between your concurrent tasks and data:

- On the task level, design your tasks to be as independent as possible - this can also be **temporal**, namely take into account any necessity for ordering the tasks and their execution
- Sharing of data will require synchronization and thus introduce data dependencies, if not race conditions or contention

An Hour of Planning ...

Taking the time to carefully analyze an existing application, or plan a new one, can really pay off later:

Plan for parallel programming by keeping data and task parallel constructs/opportunities in mind

Analyze an existing or new program to discover/confirm the locations of the time-consuming portions

Optimize by implementing a strategy using data or task parallel constructs

Auto-Parallelization

Often called “implicit” parallelism, some compilers (usally commercial ones) have the ability to:

- automatically parallelize simple (outer) loops (thread-level parallelism, or TLP)
- automatically vectorize (usually innermost loops) using ILP

Note that thread level parallelism is only usable on an SMP architecture

Auto-Vectorization

Like auto-parallelization, a feature of high-performance (often commercial) compilers on hardware that supports at least some level of vectorization:

- compilers finds low-level operations that can operate simultaneously on multiple data elements using a single instruction
- Intel/AMD processors with SSE/SSE2/SSE3 usually limited to 2^1 - 2^3 vector length (hardware limitation that true "vector" processors do not share)
- User can exert some control through compiler directives (usually vendor specific) and data organization focused on vector operations

Downside of Automatic Methods

The automatic parallelization schemes have a number of shortfalls:

- Very limited in scope - parallelizes only a very small subset of code
- Without directives from the programmer, compiler is very limited in identifying parallelizable regions of code
- Performance is rather easily degraded rather than sped up
- Wrong results are easy to (automatically) produce

Shared Memory Parallelism

Here we consider "explicit" modes of parallel programming on shared memory architectures:

- `HPF`, High Performance Fortran
- `SHMEM`, the old Cray shared memory library
- `Pthreads`, UNIX ANSI standard (available on Windows as well)
- `OpenMP`, common specification for compiler directives and API

Distributed Memory Parallel Programming

In this case, mainly message passing, with a few other (compiler directives again) possibilities:

- `HPF`, High Performance Fortran
- `Cluster OpenMP`, a new product from Intel to push OpenMP into a distributed memory regime
- `PVM`, Parallel Virtual Machine
- `MPI`, Message Passing Interface, has become the dominant API for general purpose parallel programming
- `UPC`, Unified Parallel C, extensions to ISO 99 C for parallel programming (new)
- `CAF`, Co-Array Fortran, parallel extensions to Fortran 95/2003 (new, officially part of Fortran 2008)

OpenMP

Synopsis: designed for multi-platform shared memory parallel programming in C/C++/FORTRAN primarily through the use of compiler directives and environmental variables (library routines also available).

Target: Shared memory systems.

Current Status: Specification 1.0 released in 1997, 2.0 in 2002, 2.5 in 2005, 3.0 in 2008, 3.1 in 2011. Widely implemented by commercial compiler vendors, also now in most open-source compilers (4.0 specification released in 2013-07, not yet available in production compilers).

More Info: The OpenMP home page:
http://www.openmp.org

OpenMP Availability

Platform	Version	Invocation (example)
Linux x86_64	3.1	ifort -openmp -openmp_report2 ...
	3.0	pgf90 -mp ...
GNU (>=4.2)	2.5	gfortran -fopenmp ...
GNU (>=4.4)	3.0	gfortran -fopenmp ...
GNU (>=4.7)	3.1	gfortran -fopenmp ...

MPI

Message **P**assing **I**nterface

Synopsis: Message passing library specification proposed as a standard by committee of vendors, implementors, and users.

Target: Distributed and shared memory systems.

Current Status: Most popular (and most portable) of the message passing APIs.

More Info: ANL's main MPI site:
`www-unix.mcs.anl.gov/mpi`

MPI Availability at CCR

Platform	Version(+MPI-2)
Linux IA64	1.2+(C++,MPI-I/O)
Linux x86_64	1.2+(C++,MPI-I/O),2.x(various)

I am starting to favor the commercial `Intel MPI` for its ease of use, expecially in terms of supporting multiple networks/protocols.

Unified Parallel C (UPC)

Unified Parallel C (UPC) is a project based at Lawrence Berkeley Laboratory to extend C (ISO 99) and provide large-scale parallel computing on both shared and distributed memory hardware:

- Explicit parallel execution (SPMD)
- Shared address space (shared/private data like OpenMP), but exploits data locality
- Primitives for memory management
- BSD license (free download)
- `http://upc.lbl.gov/`, community website at `http://upc.gwu.edu/`

Simple example of some UPC syntax:

```
1   shared int all_hits[THREADS];
2  ...
3  ...
4      for (i=0; i < my_trials; i++) my_hits += hit();
5      all_hits[MYTHREAD] = my_hits;
6      upc_barrier;
7      if (MYTHREAD == 0) {
8        total_hits = 0;
9        for (i=0; i < THREADS; i++) {
10           total_hits += all_hits[i];
11       }
12       pi = 4.0*((double) total_hits)/((double) trials);
13       printf("PI estimated to %10.7f from %d trials on %d threads.\n",
14        pi, trials, THREADS);
15     }
```

Co-Array Fortran

Co-Array Fortran (CAF) is an extension to Fortran (Fortran 95/2003) to provide data decomposition for parallel programs (somewhat akin to UPC and HPF):

- Original specification by Numrich and Reid, ACM Fortran Forum, **17**, no. 2, pp 1-31 (1998).
- ISO Fortran committee included co-arrays in next revision to Fortran standard (c.f. Numrich and Reid, ACM Fortran Forum, **24**, no 2, pp 4-17 (2005), Fortran 2008.
- Does not require shared memory (more applicable than OpenMP)
- Early compiler work at Rice:

  `http://www.hipersoft.rice.edu/caf/index.html`

- Simple examples of CAF usage:

```
1  REAL, DIMENSION(N)[*] :: X,Y
2  X        = Y[PE]   ! get from Y[PE]
3  Y[PE]    = X       ! put into Y[PE]
4  Y[:]     = X       ! broadcast X
5  Y[LIST]  = X       ! broadcast X over subset of PE's in array LIST
6  Z(:)     = Y[:]    ! collect all Y
7  S = MINVAL(Y[:])   ! min (reduce) all Y
8  B(1:M)[1:N] = S    ! S scalar, promoted to array of shape (1:M,1:N)
```

- UPC and CAF are examples of partitioned global address space (PGAS) language, for which a large push is being made by AHPCRC/DARPA as part of the **Petascale** computing initiative (also one based on java called Titanium)

Leveraging Existing Parallel Libraries

One “easy” way to utilize parallel programming resources is through the use of existing parallel libraries. Most common examples (note orientation around standard mathematical routines):

BLAS - Basic Linear Algebra Subroutines

LAPACK - Linear Algebra PACKage (uses BLAS)

ScaLAPACK - distributed memory (MPI) solvers for common LAPACK functions

PetSC - Portable, Extensible Toolkit for Scientific Computation

FFTW - Fastest Fourier Transforms in the West

BLAS

The Basic Linear Algebra Subroutines (BLAS) form a standard set of library functions for vector and matrix operations:

- Level 1 Vector-Vector (e.g. **xdot**, where x=s,d,c,z)
- Level 2 Matrix-Vector (e.g. **xaxpy**, where x=s,d,c,z)
- Level 3 Matrix-Matrix (e.g. **xgemm**, where x=s,d,c,z)

These routines are generally provided by vendors hand-tuned at the level of assembly code for optimum performance on a particular processor. Shared memory versions (multithreaded) and distributed memory versions available.

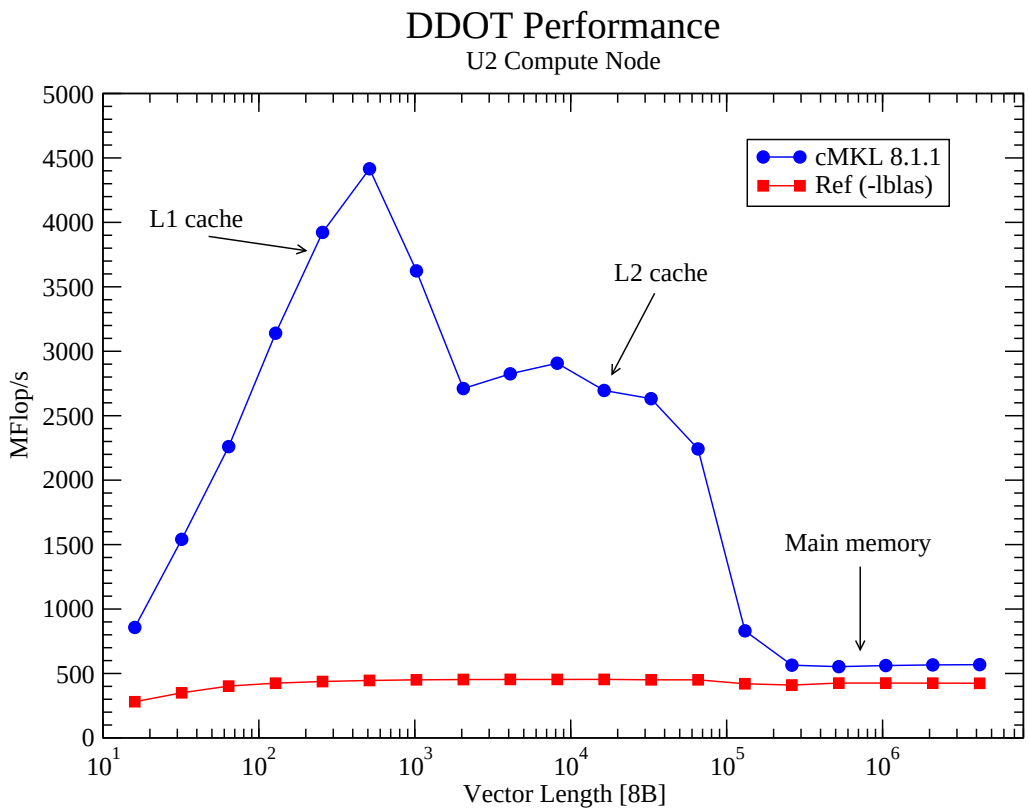
www.netlib.org/blas

Vendor BLAS

Vendor implementations of the BLAS:

AMD	ACML
Apple	Velocity Engine
Compaq	CXML
Cray	libsci
HP	MLIB
IBM	ESSL
Intel	MKL
NEC	PDLIB/SX
SGI	SCSL
SUN	Sun Performance Library

Performance Example



Performance advantage of Intel MKL **ddot** versus reference (system) version. Level 3 BLAS routines have even more significant gains.

LAPACK

The Linear Algebra PACKage (LAPACK) is a library that lies on top of the BLAS (for optimium performance and parallelism) and provides:

- solvers for systems of simultaneous linear equations
- least-squares solutions
- eigenvalue problems
- singular value problems
- On CCR systems, the `Intel MKL` is generally preferred (includes optimized BLAS and LAPACK)

www.netlib.org/lapack

ScaLAPACK

The Scalable LAPACK library, ScaLAPACK, is designed for use on the largest of problems (typically implemented on distributed memory systems):

- subset of LAPACK routines redesigned for distributed memory MIMD parallel computers
- explicit message passing for interprocessor communication
- assumes matrices are laid out in a two-dimensional block cyclic decomposition
- On CCR systems, the `Intel Cluster MKL` includes ScaLAPACK libraries (includes optimized BLAS, LAPACK, and ScaLAPACK)

`www.netlib.org/scalapack`

Programming Costs

Consider:

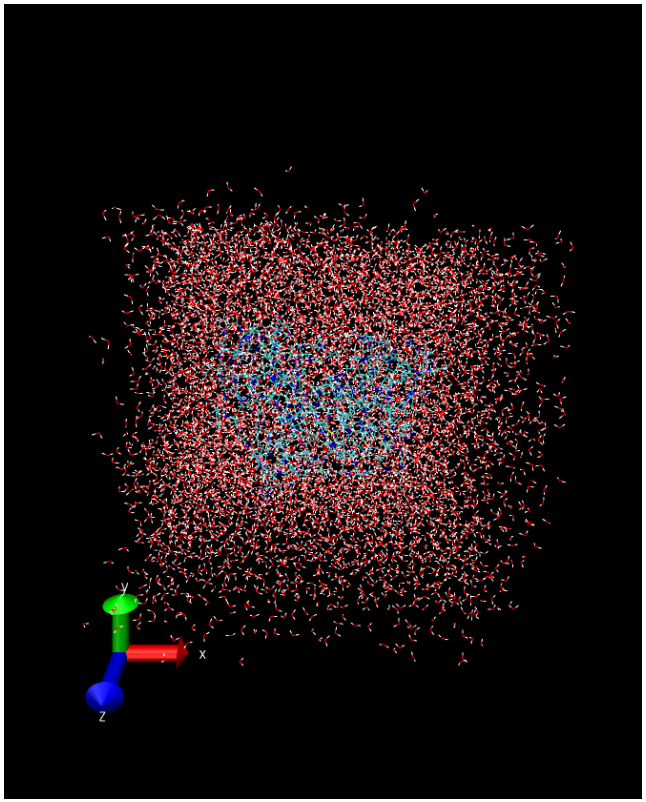
Complexity: parallel codes can be orders of magnitude more complex (especially those using message passing) - you have to plan for and deal with multiple instruction/data streams

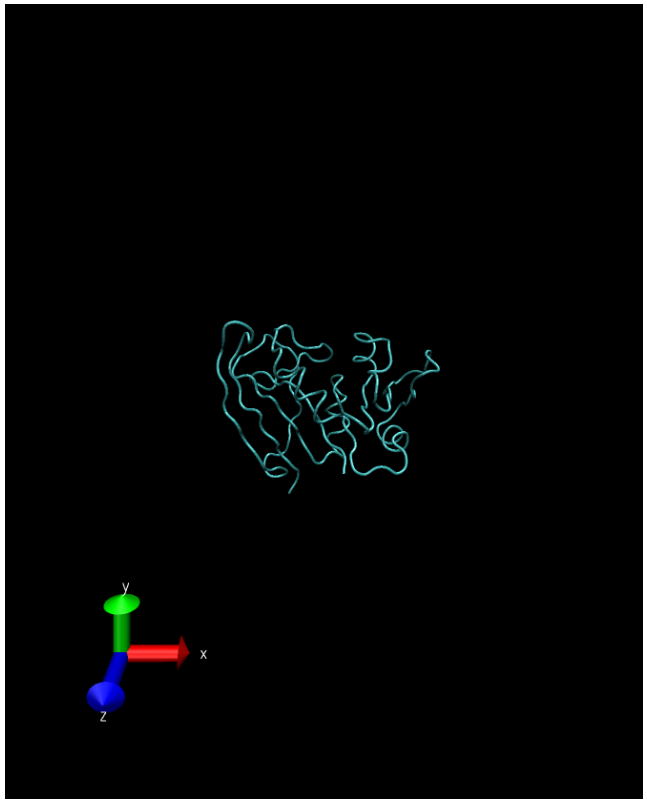
Portability: parallel codes frequently have long lifetimes (proportional to the amount of effort invested in them) - all of the serial application porting issues apply, plus the choice of parallel API (MPI, OpenMP, and POSIX threads are currently good portable choices, but implementations of them can differ from platform to platform)

Resources: overhead for parallel computation can be significant for smaller calculations

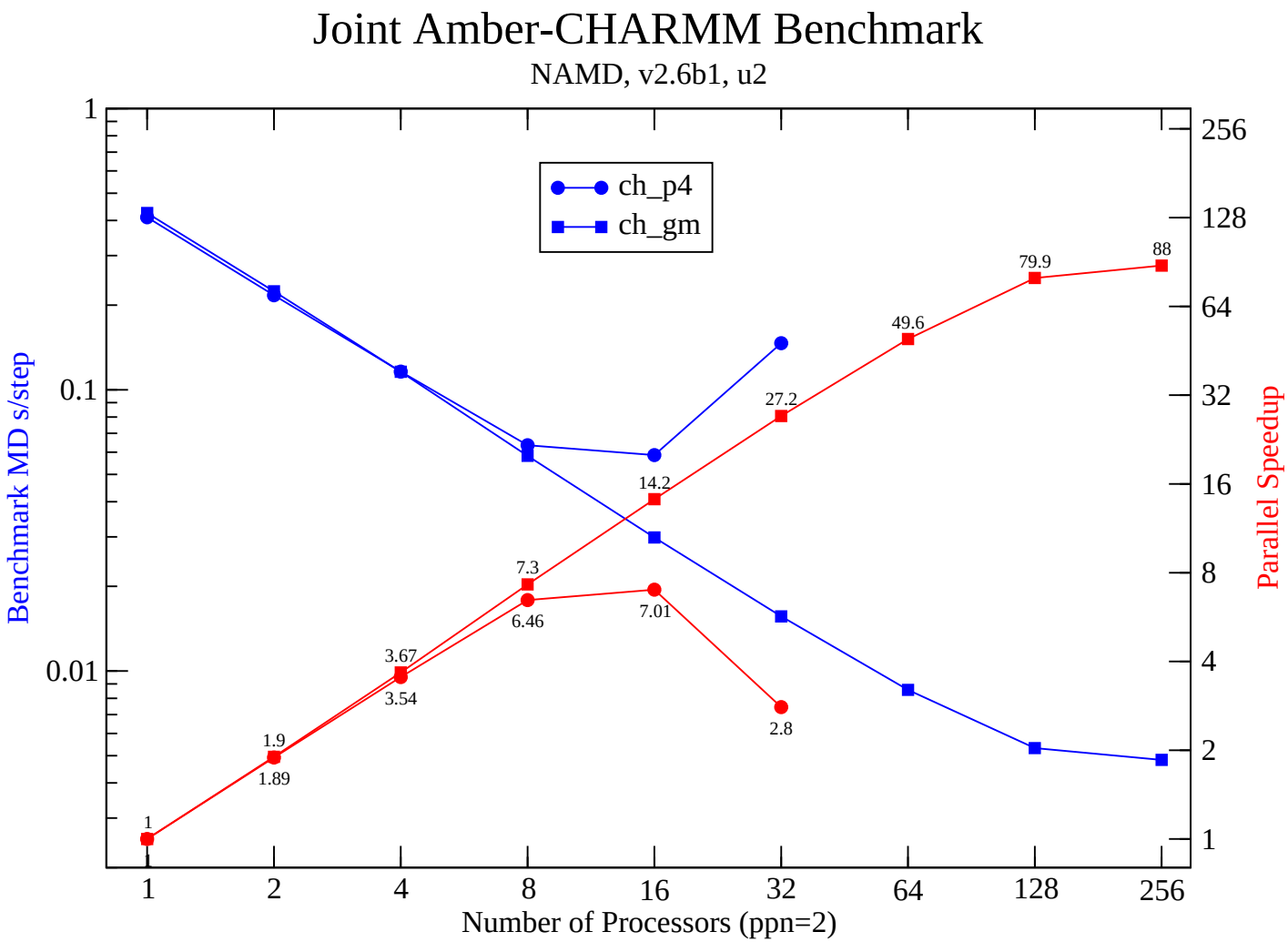
Scalability: limitations in hardware (CPU-memory speed and contention, for one example) and the parallel algorithms will limit speedups. All codes will eventually reach a state of decreasing returns at some point

Speedup Limitation Example (MD/NAMD)





JAC (Joint Amber-CHARMM) Benchmark: DHFR Protein, 7182 residues, 23558 atoms, 7023 TIP3 waters, PME (9Åcutoff)


Joint Amber-CHARMM Benchmark
NAMD, v2.6b1, u2
ch_p4
ch_gm
Benchmark MD s/step
Parallel Speedup
Number of Processors (ppn=2)
1
0.1
0.01
256
128
64
32
16
8
4
2
1
1
2
4
8
16
32
64
128
256
1
1.9
1.89
3.67
3.54
7.3
6.46
14.2
7.01
27.2
2.8
49.6
79.9
88

Communications

All parallel programs will need some communication between tasks, but the amount varies considerably:

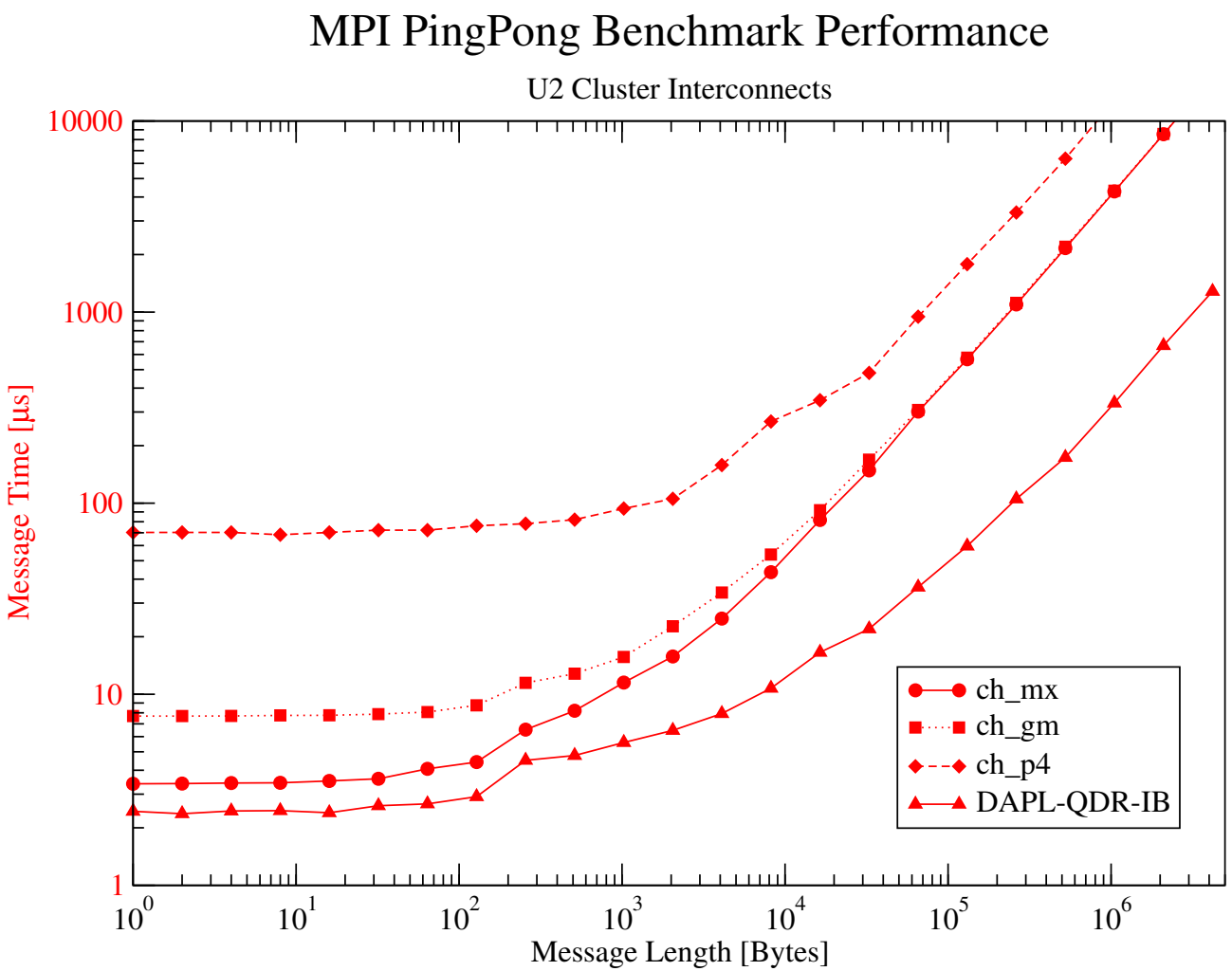
- **minimal** communication, also known as “embarrassingly” parallel - think Monte Carlo calculations
- **robust** communication, in which processes must exchange or update information frequently - time evolution codes, domain decomposed finite difference/element applications

Communication Considerations

Communication needs between parallel processes affect parallel programs in several important ways:

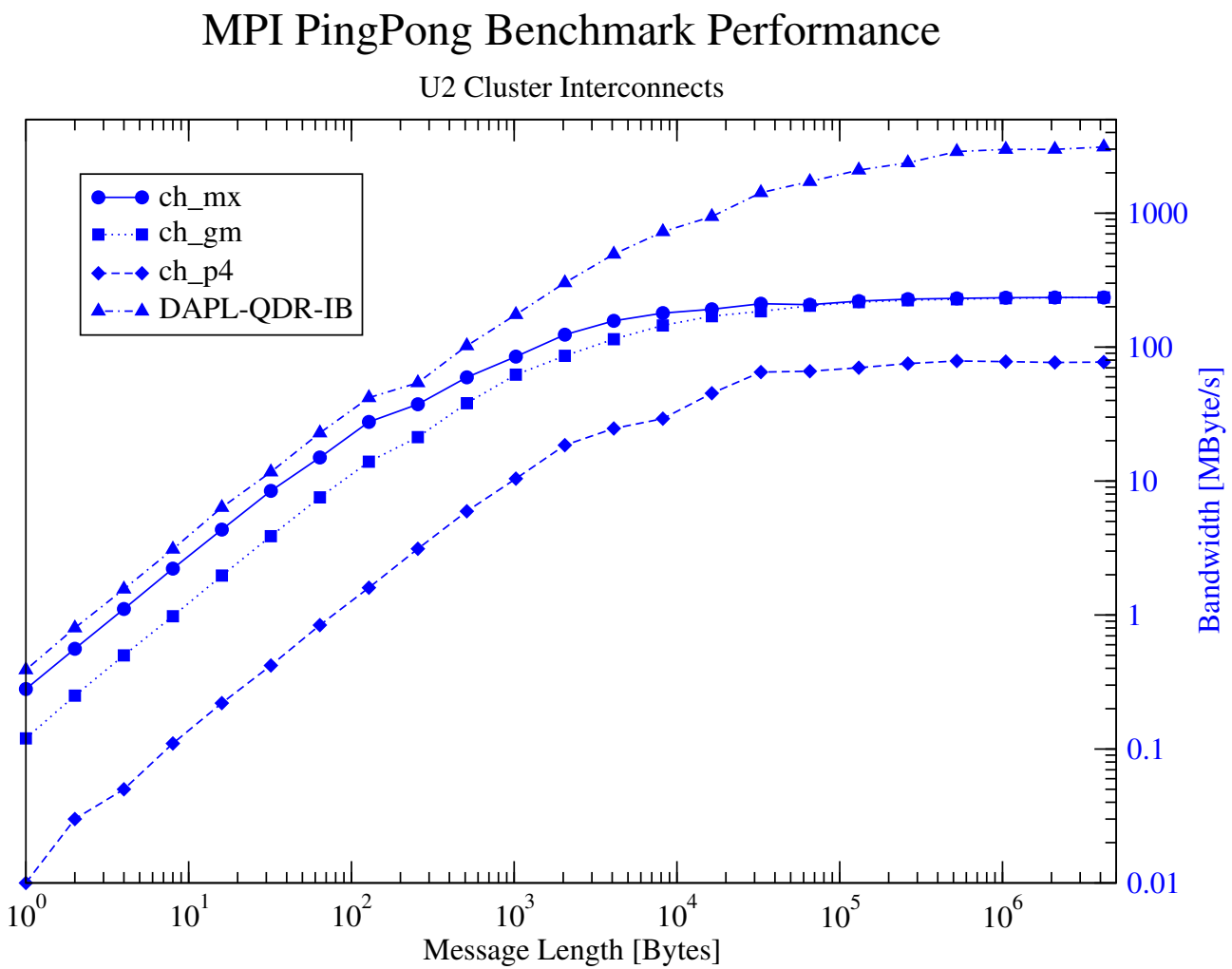
- **Cost**: there is always overhead when communicating:
  - *latency* - the cost in overhead for a zero length message (microseconds)
  - *bandwidth* - the amount of data that can be sent per unit time (MBytes/second); many applications utilize many small messages and are *latency-bound*
- **Scope**: *point-to-point* communication is always faster than *collective* communication (that involves a large subset or all processes)
- **Efficiency**: are you using the best available resource (network) for communicating?

Latency Example - MPI/U2



Latency on U2 - TCP/IP Ethernet/ch_p4, Myrinet/ch_mx, and QDR Infiniband.

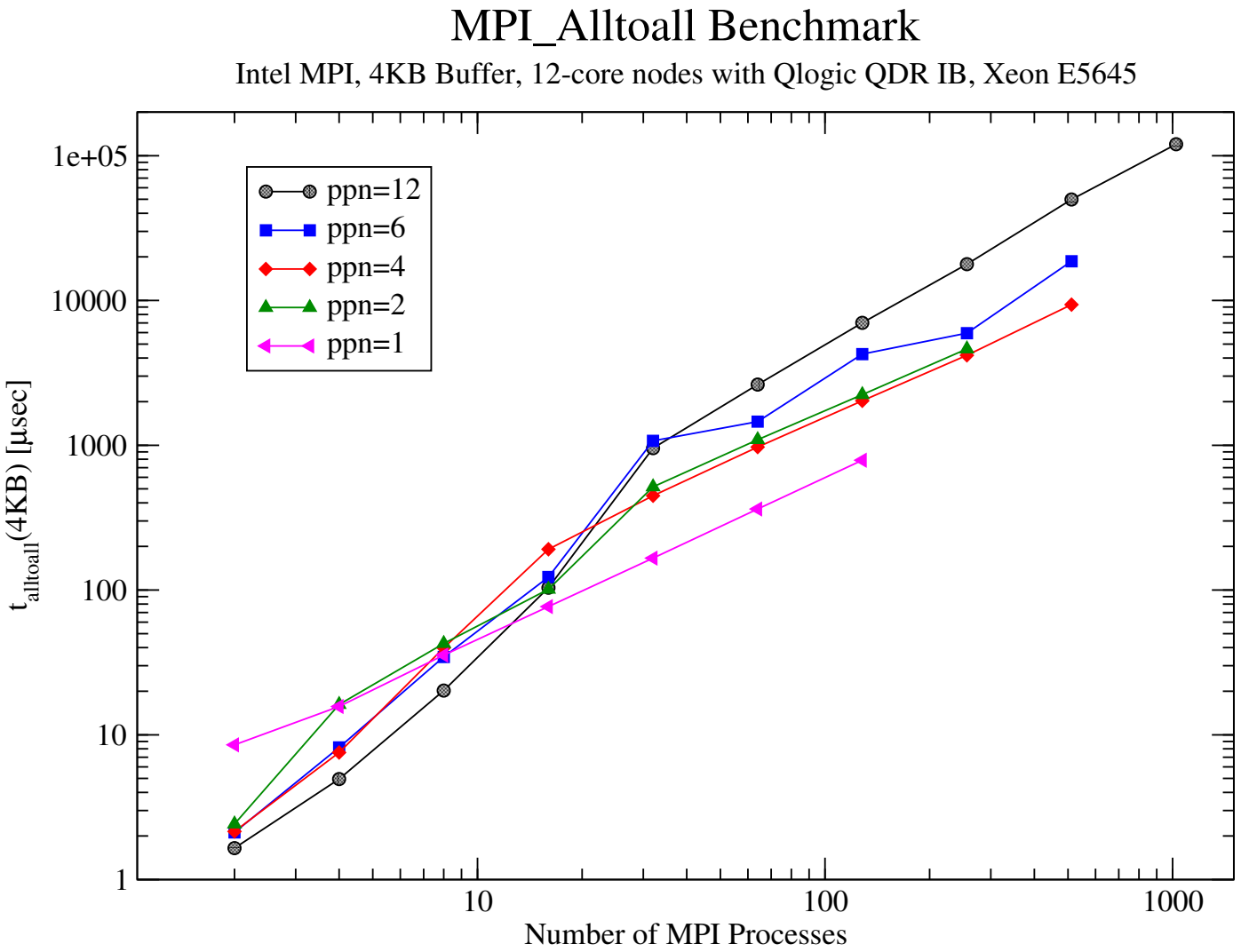
Bandwidth Example - MPI/U2



Bandwidth on U2 - TCP/IP Ethernet/ch_p4, Myrinet/ch_mx, and QDR Infiniband.

- **Visibility**: message-passing codes require the programmer to explicitly manage all communications, but many data-parallel codes do not, masking the underlying data transfers
- **Synchronicity**: Synchronous communications are called *blocking* as they require an explicit "handshake" between parallel processes - asynchronous communications (*non-blocking* offer the ability to overlap computation with communication, but place an extra burden on the programmer. Examples include:
  - Barriers - force collective synchronization; often implied, and always expensive
  - Locks/Semaphores - typically protect memory locations from simultaneous/conflicting updates to prevent race conditions

Collective Example - MPI/U2



Cost of `MPI_Alltoall` for 4KB buffer on U2/QDR IB.