More on Parallelism

William Gropp

Dealing with Interconnect Topology

- As we've seen, messages from different processes can interact within the network, impacting performance
- What role should algorithms take?
- What role should the programming model take in addressing this?
  - ♦ MPI provides some support but implementations are uneven…

Using MPI For Process Placement

- MPI provides "process topology" routines to create a new communicator with a "better" layout
- When using a regular grid, consider using these routines:

```
int dims[2], periodic[2];
for (i=0; i<2; i++)  { dims[i] = 0; periodic[i] = 0; }
MPI_Dims_create( size, 2, dims );
MPI_Cart_create( MPI_COMM_WORLD, 2, dims, periodic, 1,
&newComm );
```

- The "1" tells MPI_Cart_create to reorder the mapping of processes to create a "better" communicator for neighbor communication.
- Use newComm instead of MPI_COMM_WORLD in neighbor communication
- There's also an MPI_Graph_create, but it isn't very useful (too general). You can use MPI_Comm_split to create your very own reordering.

Experiments with Topology and Halo Communication on "Leadership Class" Machines

- The following slides show some results for a simple halo exchange program (halocompare) that tries several MPI-1 approaches and several different communicators:
  - ♦ MPI_COMM_WORLD
  - ♦ Dup of MPI_COMM_WORLD
    - Is MPI_COMM_WORLD special in terms of performance?
  - ♦ Reordered communicator - all even ranks in MPI_COMM_WORLD first, then the odd ranks
    - Is ordering of processes important?
  - ♦ Communicator from MPI_Dims_create/MPI_Cart_create
    - Does MPI Implementation support these, and do they help
- Communication choices are
  - ♦ Send/Irecv
  - ♦ Isend/Irecv
  - ♦ "Phased"

Method 1: Use Irecv and Send

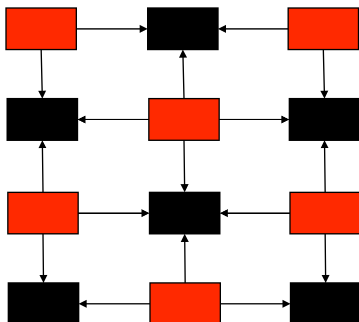
- Do i=1,n_neighbors
  Call MPI_Irecv(inedge(1,i), len, MPI_REAL, nbr(i), tag,&
  comm, requests(i), ierr)
  Enddo
  Do i=1,n_neighbors
  Call MPI_Send(edge(1,i), len, MPI_REAL, nbr(i), tag,&
  comm, ierr)
  Enddo
  Call MPI_Waitall(n_neighbors, requests, statuses, ierr)
- Does not perform well in practice (at least on BG, SP).
  - Quiz for the audience: Why?

Method 2: Use Isend and Irecv

- Do i=1,n_neighbors
  Call MPI_Irecv(inedge(1,i),len,MPI_REAL,nbr(i),tag,&
  comm, requests(i),ierr)
  Enddo
  Do i=1,n_neighbors
  Call MPI_Isend(edge(1,i), len, MPI_REAL, nbr(i), tag,&
  comm, requests(n_neighbors+i), ierr)
  Enddo
  Call MPI_Waitall(2*n_neighbors, requests, statuses, ierr)

Phased Communication

- It may be easier for the MPI implementation to either send or receive
- Color the nodes so that all senders are of one color and all receivers of the other. Then use two phases
  - Just a "Red-Black" partitioning of nodes
  - For more complex patterns, more colors may be necessary



This is an example of manual scheduling a communication step. Only consider this if there is evidence of inefficient communication.

Halo Exchange on BG/L

	4 Neighbors		8 Neighbors	
	Irecv/Send	Irecv/Isend	Irecv/Send	Irecv/Isend
World	112	199	94	133
Even/Odd	81	114	71	93
Cart_create	107	218	104	194

- 64 processes, co-processor mode, 2048 doubles to each neighbor
- Rate is MB/Sec (for all tables)

Halo Exchange on BG/L

	4 Neighbors		8 Neighbors	
	Irecv/Send	Irecv/Isend	Irecv/Send	Irecv/Isend
World	64	120	63	72
Even/Odd	48	64	41	47
Cart_create	103	201	103	132

- 128 processes, virtual-node mode, 2048 doubles to each neighbor
- Same number of *nodes* as previous table

Halo Exchange on BG/P

	4 Neighbors		8 Neighbors	
	Irecv/Send	Irecv/Isend	Irecv/Send	Irecv/Isend
World	208	328	184	237
Even/Odd	219	327	172	243
Cart_create	301	581	242	410

- 64 processes, co-processor mode, 2048 doubles to each neighbor
- Rate is MB/Sec (for all tables)

Halo Exchange on BG/P

	4 Neighbors		8 Neighbors	
	Irecv/Send	Irecv/Isend	Irecv/Send	Irecv/Isend
World	112	132	81	89
Even/Odd	74	84	50	50
Cart_create	109	132	84	89
World(txyz)	132	177	92	108
Even/Odd(txyz)	78	84	55	55
Cart_create(txyz)	132	177	84	108

- 256 processes, virtual-node mode, 2048 doubles to each neighbor
- Same number of *nodes* as previous table

Halo Exchange on Cray XT4

	4 Neighbors			8 Neighbors	
	Irecv/Send	Irecv/Isend	Phased	Irecv/Send	Irecv/Isend
World	153	153	165	133	136
Even/Odd	128	126	137	114	111
Cart_create	133	137	143	117	117

(Periodic)	4 Neighbors			8 Neighbors	
	Irecv/Send	Irecv/Isend	Phased	Irecv/Send	Irecv/Isend
World	131	131	139	115	114
Even/Odd	113	116	119	104	104
Cart_create	151	151	164	129	128

- 1024 processes, 2000 doubles to each neighbor

Halo Exchange on Cray XT4

	4 Neighbors			8 Neighbors	
	Irecv/Send	Irecv/Isend	Phased	Irecv/Send	Irecv/Isend
World	311	306	331	262	269
Even/Odd	257	247	279	212	206
Cart_create	265	275	266	236	232

(Periodic)	4 Neighbors			8 Neighbors	
	Irecv/Send	Irecv/Isend	Phased	Irecv/Send	Irecv/Isend
World	264	268	262	230	233
Even/Odd	217	217	220	192	197
Cart_create	300	306	319	256	254

- 1024 processes, SN mode, 2000 doubles to each neighbor

Observations on Halo Exchange

- Topology is important (again)
- For these tests, MPI_Cart_create always a good idea for BG/L; often a good idea for periodic meshes on Cray XT3/4
  - Not clear if MPI_Cart_create creates optimal map on Cray
  - On BG/P, there is an environment variable controlling mapping. MPI_Cart_create has no effect in vn mode :(
- Cray performance is significantly under what the "ping-pong" performance test would predict
  - The success of the "phased" approach on the Cray suggests that some communication contention may be contributing to the slow-down
    - Either contention along links (which should not happen when MPI_Cart_create is used) or contention at the destination node.
  - To see this, consider the performance of a single process sending to four neighbors

Discovering Performance Opportunities

- Lets look at a single process sending to its neighbors. We *expect* the rate to be roughly twice that for the halo (since this test is only sending, not sending and receiving)

System	4 neighbors		8 Neighbors	
		Periodic		Periodic
BG/L	488	490	389	389
BG/L, VN	294	294	239	239
BG/P	1139	1136	892	892
BG/P, VN	468	468	600	601
XT3	1005	1007	1053	1045
XT4	1634	1620	1773	1770
XT4 SN	1701	1701	1811	1808

- BG gives roughly double the halo rate. XTn is much higher
  - It should be possible to improve the halo exchange on the XT by scheduling the communication
  - Or improving the MPI implementation

Discovering Performance Opportunities

- Ratios of a single sender to all processes sending (in rate)
- *Expect* a factor of roughly 2 in non-VN mode (since processes must also receive)

System	4 neighbors		8 Neighbors	
		Periodic		Periodic
BG/L	2.2		2.0	
BG/L, VN	1.5		1.8	
BG/P	3.8		2.2	
BG/P, VN	2.6		5.5	
XT3	7.5	8.1	9.1	9.4
XT4	11	11	13	14
XT4 SN	5.5	5.6	6.7	7.1

- BG/L gives roughly double the halo rate. XTn is much higher
  - It should be possible to improve the halo exchange on the XT by scheduling the communication
  - Or improving the MPI implementation (But is it topology routines or point-to-point communication? How would you test each hypothesis?)

What Does All of This Mean?

- The simple s+rn model is a good starting point
  - ♦ But it can *overestimate* performance when there is contention on the links between processes
    - One "fix" is to reduce 1/r (rate)
  - ♦ And it can *underestimate* performance when concurrency is possible on the links between processes
  - ♦ It also ignores the possibility of concurrency between computation and communication
    - Much as the Cache-Oblivious model ignores the possibility of overlap with cache misses and work

Programming Model Considerations

- Process(or) topology influences performance
  - ♦ What role should the programming model take with respect to processor topology?
    - None: Assume complete interconnect
    - Total: Reflect underlying physical interconnect
    - Abstract: Assume some generic model
      - MPI sort of does this for the special case of regular grid (Cartesian) communication
- Other performance models include PRAM (assume infinite communication speeds) and Bulk Synchronous Programming (communication in phases, with overlap of computation)
  - ♦ BSP designed in concert with a program model (library) to make performance more predictable

A Parallel Performance Model With Overlap

- The logP model
  - ♦ L – Latency (or delay) for a single short message (typically a word)
  - ♦ o – Overhead (time processor must devote to sending message)
  - ♦ g – "Gap" or time between consecutive message transmissions (basically controls bandwidth through L/g messages inflight)
  - ♦ P – number of processors
- Better separates time that it takes a message to move from one processor to another (L) from the time it takes a processor to manage that communication (o)

Comments

- Programming Models for Distributed Memory
  - ♦ Focus on moving data between separate address spaces
  - ♦ The programmer is responsible for managing the partitioning of data structures among the separate address spaces
    - A major reason that these programming models are considered hard to use
  - ♦ Two principle models for communicating data:
    - Two-sided or send-receive message passing
    - One-sided or put/get/accumulate

Two Sided Communication

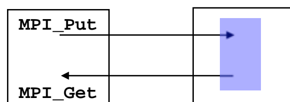
- Source and destination actively participate in communicating data
- The memory locations being accessed (read or written) are explicitly specified in both sender and receiver
- The period of time in which data may be changed is clearly marked in the code
- A consequence (almost) of these features is that det*erministic algorithms have provably deterministic implementations*

One Sided Communication

- One process initiates data transfer
- Completion may be either:
  - Collective: All (involved) processes agree to complete all pending one-sided communication
    - In MPI terms, this is active *target* synchronization
  - Independent: Only the initiating process marks the completion
    - In MPI terms, this is passive *target synchronization*
    - The target (remote) process may need to perform some action, but this is not represented in the program
- Lets consider the halo exchange with one-sided operations

MPI One-Sided Communication

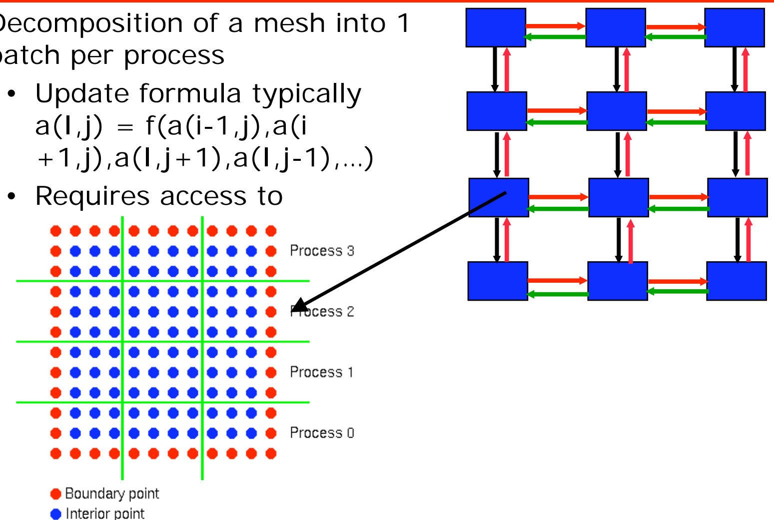
- Three data transfer functions
  - Put, get, accumulate



- Three synchronization methods
  - Fence
  - Post-start-complete-wait
  - Lock-unlock
- A natural choice for implementing halo exchanges
  - Multiple communication per synchronization

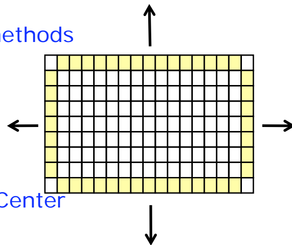
Halo Exchange

- Decomposition of a mesh into 1 patch per process
  - Update formula typically a(I,j) = f(a(i-1,j),a(i+1,j),a(I,j+1),a(I,j-1),...)
  - Requires access to

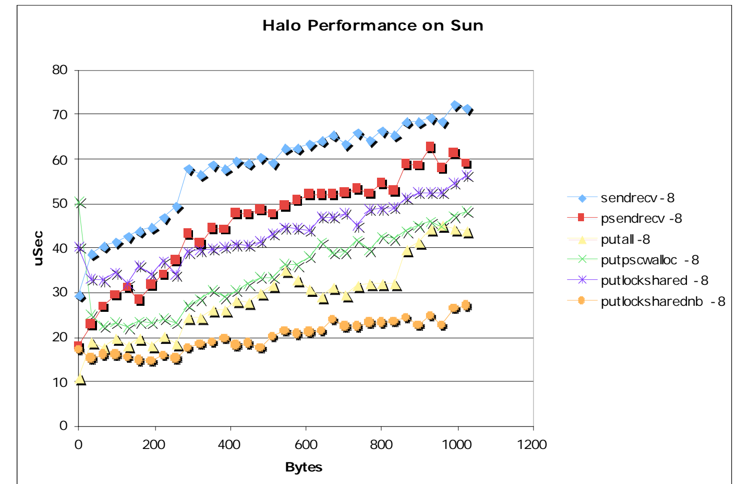


Performance Tests

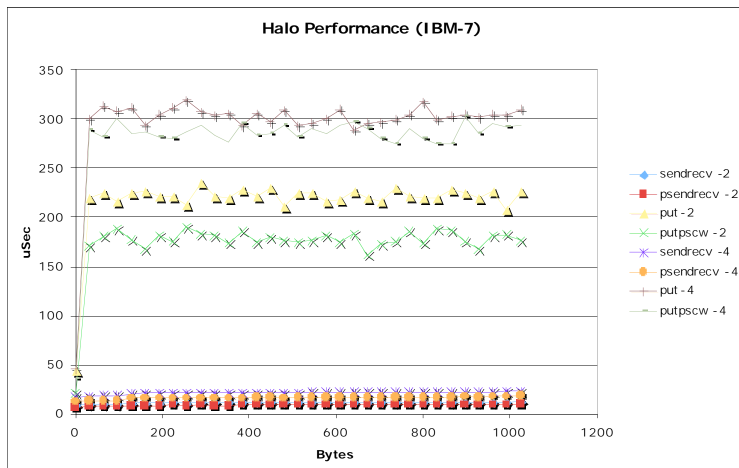
- "Halo" exchange or ghost-cell exchange operation
  - Each process exchanges data with its nearest neighbors
  - Part of the mpptest benchmark; works with any MPI implementation
    - Even handles implementations that only provide a subset of MPI-2 RMA functionality
    - Similar code to that in halocompare, but doesn't use process topologies (yet)
  - One-sided version uses all 3 synchronization methods
- Available from
- http://www.mcs.anl.gov/mpi/mpptest
- Ran on
  - Sun Fire SMP at RWTH, Aachen, Germany
  - IBM p655+ SMP at San Diego Supercomputer Center



One-Sided Communication on Sun SMP with Sun MPI



One-Sided Communication on IBM SMP with IBM MPI



Observations on MPI RMA and Halo Exchange

- With a good implementation and appropriate hardware, MPI RMA can provide a performance benefit over MPI point-to-point
- A key feature is separating initiation from completion and permitting a single operation (e.g., MPI_Win_fence) to complete multiple data transfers (MPI_Put)
- Architectural note:
  - A fast (hardware) barrier can be critical
  - If "fence" is implemented with point-to-point communication, $\log_2 p$ communication steps are required
  - If a bast barrier/fence is provided, how is the set of processes or nodes defined? Can multiple sets be defined?

Adapting the Algorithm to Architecture

- Problem:
  - $u \in \Re^n$, $F(u) = 0$, representing nonlinear PDE on Domain Ω. Discretize.
- Typical Algorithmic decomposition:
  - Nonlinear problem →
    Newton method →
    Linear system involving Jacobian matrix →
    Solve linear system in parallel
- However, limited temporal locality for linear solves, particularly for solvers such as multigrid
- One Solution: Cross - iteration algorithms
  - Think in blocks involving time/iteration, not just slices
  - Examples - CG methods, Nonlinear Schwarz

Nonlinear Schwarz Brings Back Memory Reuse

- An alternate approach:
  - Divide Ω into overlapping domains Ω_i, boundaries $\partial\Omega_i$ Let u_i be u restricted to Ω_i. $\partial\Omega_i \cap \Omega$ set from Ω_j
  - For k=0, …
    Solve $F(u_1^{k+1},u_2^k) = 0$ for u_1^{k+1}on Ω_1,
    Solve $F(u_1^k,u_2^{k+1}) = 0$ for u_2^{k+1} on Ω_2, …
- Each subdomain involves local (cache resident) solve
  - Choose Ω_i to fit in fast memory
  - Nonlinear methods are not (yet) O(1)
    - Permit temporal locality
  - Linear solvers used are O(1)
- Memory hierarchy handled through multilevel version
  - Solve $F(u_1^{k+1},u_2^k) = 0$ for u_1^{k+1}on $\Omega_1$with nonlinear Schwarz, etc.
- For an intro to memory issues for algorithm designers, see
  - Karp in SIAM Review 1996
  - McGeoch, AMS Notices March 2001

The Dimensions of a Typical Cluster

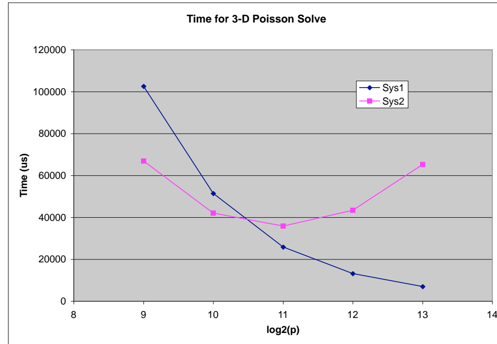
- 6.1 m x 2.1 m x 1m
- 1-norm size (airline baggage norm) = 9.2m
- At 2.4Ghz, =
  74 cycles
  (49 x 17 x 8)
- Real distance is greater
  - Routes longer
  - Signals slower than light in a vacuum
- PRAM (Parallel Random Access Memory) model is not helpful
  - Like using Newtonian mechanics for predicting behavior of near lightspeed particles
  - It is *too* simple

Two Challenges for Scalable Computing

- Amdahl's law
  - Maximum speed up = $1/(1-(T_s/(T_s+T_p)) = 1/(1\text{-serial_fraction})$
  - For applications that require every bit of available memory (so called weak scaling), the serial fraction is very small
  - For applications with fixed problem size (strong scaling), this is often already a problem
- Little's law
  - From queuing theory
    - In a stable system, the arrival rate * the residency time equals the number in the queue
  - For memory, we have
    - Residency time = memory latency
    - Arrival rate = 1/clock
    - Thus number = memory latency in clocks
  - This number is the number of outstanding operations, such as loads, or the number of concurrent operations needed to avoid waiting on memory
  - Typical values are 100-250

Achieving Good Scaling

- Solve a 3-D Poisson Problem as part of a larger application
- Algorithm is Multigrid preconditioned CG
- One system shows good (predicted) scaling
- The other eventually shows a slowdown
- Why?
  - System 1 has a special network for MPI_Allreduce. Cost is low and nearly constant
  - System 2 does not. Cost is (relatively) high
- What else could we do?



Reorganizing Conjugate Gradient for High Performance

- Problem:
  - Solve a linear system Ax = b
- Conjugate Gradient Method:
  - Iterate, computing $Ap^{(n)}$ at the nth step
  - Form new approximate solution using dot products and vector operations
- Performance Problems
  - Dot products cause synchronization
  - Sparse matrix-vector products strain memory system

Effect of Inner Products

- Typical Krylov method

$\beta = r^T z$
$\rho = \beta / \beta_{old}$
$\beta_{old} = \beta$

$p = z + \rho p$
$z = Ap$
$\alpha = \beta / p^T z$
$x = x + \alpha p$
$r = r - \alpha z$
$z = Mr$

- Rearange to

start $\beta = r^T z$
$z_1 = Az$
end $\beta = r^T z$
$\rho = \beta / \beta_{old}$
$\beta_{old} = \beta$
$p = z + \rho p$
$z = z_1 + \rho z$ (=$Az + \rho Ap_{old} = Ap$)
$\alpha = \beta / p^T z$
$x = x + \alpha p$
$r = r - \alpha z$
$z = Mr$

- ***Not*** numerically identical (compiler *must not* do this)
- Deeper pipelining possible by further loop unrolling (also not numerically identical)

Comments

- No claim that this is the right thing to do
  - Illustrates the opportunities
- Must analyze tradeoffs
  - More floating point operations
  - More data motion
  - Less waiting for inner product
- General Ideas
  - Overlap communication with useful work
  - Consider cross (sub)step and cross iteration transformations
  - Initiate early, wait late
- This is a simple algebraic approach
  - The real solution is to apply these principles at the algorithmic level to gain much greater benefit

Distributed Memory code

- Single node performance is clearly a problem.
- What about parallel performance?
  - Many successes at scale (e.g., Gordon Bell Prizes for >100TF on 64K BG nodes
  - Some difficulties with load-balancing, designing code and algorithms for latency, but skilled programmers and applications scientists have been remarkably successful
- Is there a problem?
  - There is the issue of **productivity**. Consider the NAS parallel benchmark code for Multigrid (mg.f):

mg.f Fri Feb 10 15:42:41 2006 2

mg.f Fri Feb 10 15:42:41 2006 3

What is the problem? The user is responsible for all steps in the decomposition of the data structures across the processors

Note that this does give the user (or someone) a great deal of flexibility, as the data structure can be distributed in arbitrary ways across arbitrary sets of processors

Another example...

Manual Decomposition of Data Structures

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

0	1	4	5	8	9	12	13
2	3	6	7	10	11	14	15
16	17	20	21	24	25	28	29
18	19	22	23	26	27	30	31
32	33	36	37	40	41	44	45
34	35	38	39	42	43	46	47
48	49	52	53	56	57	60	61
50	51	54	55	58	59	62	63

0	1	4	5	16	17	20	21
2	3	6	7	18	19	22	23
8	9	12	13	24	25	28	29
10	11	14	15	26	27	30	31
32	33	36	37	48	49	52	53
34	35	38	39	50	51	54	55
40	41	44	45	56	57	60	61
42	43	46	47	58	59	62	63

- Trick!
  - This is from a paper on dense matrix tiling for uniprocessors!
- This suggests that managing data decompositions is a common problem for real machines, whether they are parallel or not
  - *Not just an artifact of MPI-style programming*
  - Aiding programmers in data structure decomposition is an important part of solving the productivity puzzle

Possible solutions

- Single, integrated system
  - Best choice when it works
    - E.g., Matlab, R
- Current Terascale systems and many proposed petascale systems exploit hierarchy
  - Successful at many levels
    - Cluster hardware
    - OS scalability
  - We should apply this to productivity software
    - The problem is hard
    - Apply classic and very successful Computer Science strategies to address the complexity of generating fast code for a wide range of user-defined data structures.
- How can we apply hierarchies?
  - One approach is to use libraries
    - Limited by the operations envisioned by the library designer
  - Another is to enhance the users ability to express the problem in source code

Observations

- Much use of mechanical transformations of code to achieve better performance
  - Compilers do not this well
    - Too many other demands on the compiler
- Use of carefully crafted algorithms for specific operations such as allreduce, matrix-matrix multiply
  - Far more challenging than the performance transformations
- Increasing acceptance of some degree of automation in creating code
  - ATLAS, PhiPAC, TCE
  - Source-to-source transformation systems
    - E.g., ROSE, Aspect Oriented Programming (asod.net)

Getting Performance Out of Source Code

- Let the compiler do it
  - Too difficult (the compiler has too much to worry about)
- Improve the language so that the compiler can do it
- Build better tools to work with the language and the compiler

Potential challenges faced by languages

1. Time to develop the language.
2. Divergence from mainstream compiler and language development.
3. Mismatch with application needs.
4. Performance.
5. Performance portability.
6. Concern of application developers about the success of the language.

- Understanding these provides insights into potential solutions
- Annotations can *complement* language research by using the principle of *separation of concerns*
- The annotation approach can be applied to *new* languages, as well

Challenges Faced by Languages

- Time to develop the language.
  - ♦ New languages take a long time to reach the level of maturity demanded by scientific applications
  - ♦ HPF is a good example — One designer has said that HPF needed at least 10 years and didn't get them
    - Even though JHPF has had some success on the Earth Simulator, performance costs of HPF still limit use
  - ♦ Applications need help now — this effort will provide help in the near term

Challenges Faced by Languages

- Divergence from mainstream compiler and language development.
  - ♦ Many of the productivity problems with current languages will be addressed faster and more effectively by mainstream programming environments (languages + debuggers + development environments)
  - ♦ The scientific market is not large enough to duplicate or even match these advances.
  - ♦ Reflected in the language work that seeks to *extend* rather than replace languages used by science applicatoins

Challenges Faced by Languages

- Mismatch with application needs.
  - ♦ Data structures and operations are often low-level and general or higher level but with loss of generality (Sparse matrix implementations with regular arrays is very painful). Productivity gains are shown on examples that closely match added data structures; lost on other structures
  - ♦ Domain-specific languages are really "data-structure-specific languages," emphasizing a particular set of operations rather than abstractions directly related to the science
  - ♦ I/O usually neglected
    - And rarely independent of processor count, leaving the programmer to once again explicitly manage the data decompositions to achieve a canonical output order suitable for other tools

Challenges Faced by Languages

- Performance and Performance Portability
  - ♦ Major reason for using parallelism at all is performance (the other is memory)
  - ♦ "Elegance" is irrelevant if too much performance is sacrificed
    - Of course, performance is irrelevant if it is too hard to program
  - ♦ Many applications suffer their largest performance loss within a single node
    - Languages that focus only on the parallel part of the program are missing one of the major sources of problems
    - Further, any focus on single-node performance helps a far greater number of applications
  - ♦ The success of tools that generate better code for single-processor dense matrix-matrix multiplication demonstrates the need for even simple tools

Challenges Faced by Languages

- Concern of application developers about the success of the language.
  - Applications in computational science can take decades to develop. An application developer must ask, ``Will a new language persist?'' Experience is sobering; languages such as CM Fortran and HPF had substantial efforts behind them and yet either have disappeared completely or remain a small (and not well supported) niche language.

Key Observations

- 90/10 rule
  - current languages adequate for 90% of code
  - 10% of code causes 90% of trouble
- Memory hierarchy issues a major source of problems
  - Significant effort is put into relatively mechanical transformations of code
  - Other transformations are avoided because of their negative impact on the readability and maintainability of the code.
    - Example is loop fusion for routines that sweep over a mesh to apply different physics. Fusion, needed to reduce memory bandwidth requirements, breaks modularity of routines written by different groups.
- Coordination of distributed data structures another major source of problems
  - But the need for performance encourages a global/local separation
    - Reflected in PGAS languages
- New languages may help, but not anytime soon
  - New languages will never be the entire solution
  - Applications need help now

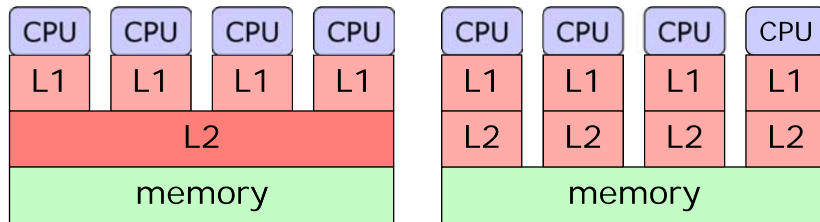
Massive Scale

- Load Balancing
  - Work virtualization
    - Give the system flexibility to allocate resources
- Fault Management
  - Fault detection
    - Check, do not impose, conservation properties
    - Symmetries imply conservation principles - exploit them
    - "Assessing Fault Sensitivity in MPI Applications," SC2004 Best Technical Paper
  - Fault Recovery
    - Compact representations for checkpointing
      - E.g., do you need every bit for a valid simulation, or could you store a set of coefficients for a FE representation to the computational accuracy? Is there a natural different representation that could be stored?
    - What is the minimum amount of information that is needed?
      - Don't forget to cost-weight the information!
    - Is there redundant information that can be used to reconstruct lost data?

Shared Memory Systems

- Loads and stores are supported in hardware
- Permits a (apparently) simple, low-overhead programming model: multiple threads, each executing standard assignment/reference operations
- Many correctness and performance hazards, however....

Two Architectural Models



- Left: Shared variables can stay in fast memory
  - But ensuring correctness complicates design, can impact performance
- Right: Updates to shared variables requires store to memory (slow)
  - But easier to design; faster for partitioned address space models

Complications

- Consistency
  - When does one thread see the results of an update to memory made by another thread?
- Sequential consistency
  - Execution is as if the execution is some interleaving of the *statements* (not the hardware instructions)
  - Code then executes "the way it looks"
- Sequential consistency is hard to make fast
  - Other consistency models trade simplicity for performance
  - Release consistency requires separate ac*quire* and *release* actions on an object

More Complications

- Writes may be completed in an order that is different than the were issued. Consider this code:

Thread 0	Thread 1
A=1;	B=3;
B=2;	While (A);
A=0;	Printf("%d\n", B);

What value is printed?
Does it matter if A and B are declared volatile?

If sequential consistency is provided, then the value printed is known.

False Sharing

- Consider this code:

Thread 0	Thread 1
N=100000;	M = 100000;
While (N--) a++;	While (M--) b++;

How many cache misses occur?
1 Model: 4: N, M, A, B.

False Sharing (2)

- Consider this case
  - A, B, N, M are all in the same cache line
  - A processor may only write to a value if it is in that cores L1 cache
  - A and B are written to memory (store), not just updated in register
- Then instead of 4 cache misses, there are as many as 200000 (one for each access to either A or B)
- This is not a correctness problem; it is a performance problem
  - The programming language *hides* the hardware-defined associating between variables