12.950 wrapup

Parallel Programming: MPI with OpenMP, MPI tuning, parallelization concepts and libraries

Final day agenda

- Hybrid MPI+OpenMP programming
- MPI Performance Tuning & Portable Performance
- Performance concepts and Scalability
- Different modes of parallelism
- Parallelizing an existing code using MPI
- Using 3rd party libraries or writing your own library

Hybrid Programming: Combining MPI with OpenMP

Acknowledgements

- Lorna Smith, Mark Bull (EPCC)
- Rolf Rabenseifner, Mathias Muller (HLRS)
- Yun He and Chris Ding (LBNL)
- The IBM, LLNL, NERSC, NCAR, NCSA, SDSC and PSC documentation and training teams.
- The MPI Forum
- I try to attribute all graphs; please forgive any mistakes or omissions.

Questions

- What is Hybrid Programming?
- Why would we care about it?
- How do we do it?
- When do we attempt it?
- Is it not delivering the performance promised?
- Alternatives

Recap

- Distributed memory programming:
  - Distinct processes, explicitly partaking in the pairwise and collective exchange of control and data messages (implicit synchronization)
  - No way to directly access the variables in the memory of another process
- Shared memory programming:
  - Multiple threads or processes sharing data in the same address space/shared memory arena and explicity synchronizing when needed

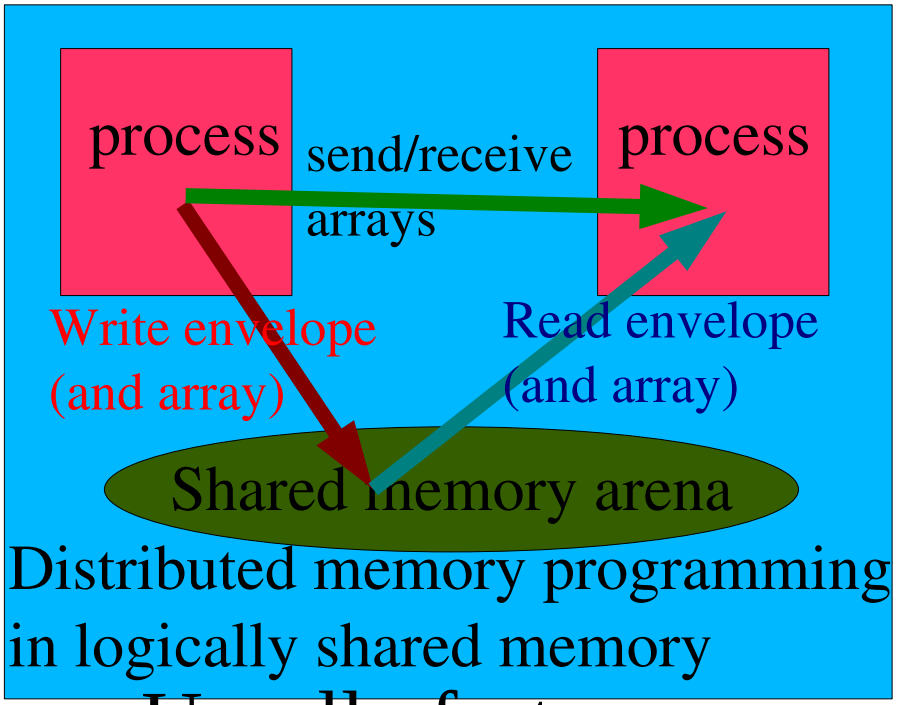
Hybrid Programming

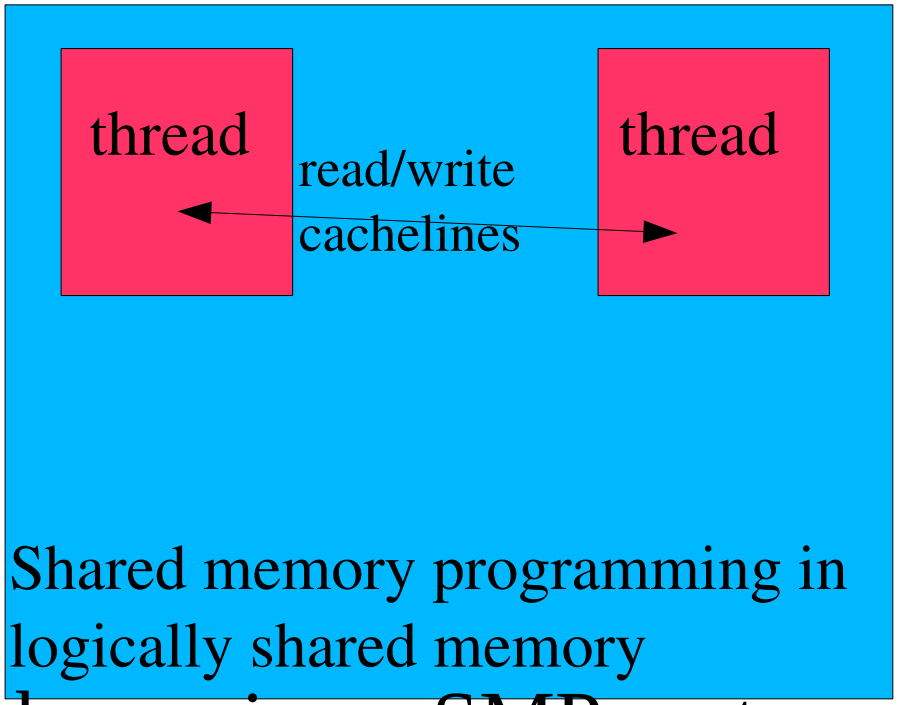
- So, on a cluster of *N* SMP nodes, each having *M* processors can we combine the two paradigms?
  - Shared memory for calculations within the node
    - *1* process, *M* threads per node
    - *P* processes, *M/P* threads per node
    - *M*N* threads, implicitly communicating when outside the node
  - Distributed memory for problems spread over many nodes.
  - This can be done explicitly (directed by the user) or implicitly (handled behind the scenes by a runtime library).

Why care?

- You have a threaded code and you'd prefer to have some code reuse scaling up to the cluster
- The MPI code has scalability problems due to communications scaling as $O(P^2)$ and cutting P by half or even to $O(P)$ is seen as helping
- *It appears as a more efficient way to exchange information between processes on the same node*
- For algorithms with two-level load-balancing, the inner level may be easier to do with threads

Why copy on an SMP/ccNUMA?





- Usually fast message exchange in an SMP system makes use of a shared memory area which processes can access to read/write data
- Sometimes memory protection is overridden using a memory bypass... (kernel copy mode)

Topology mapping & network saturation

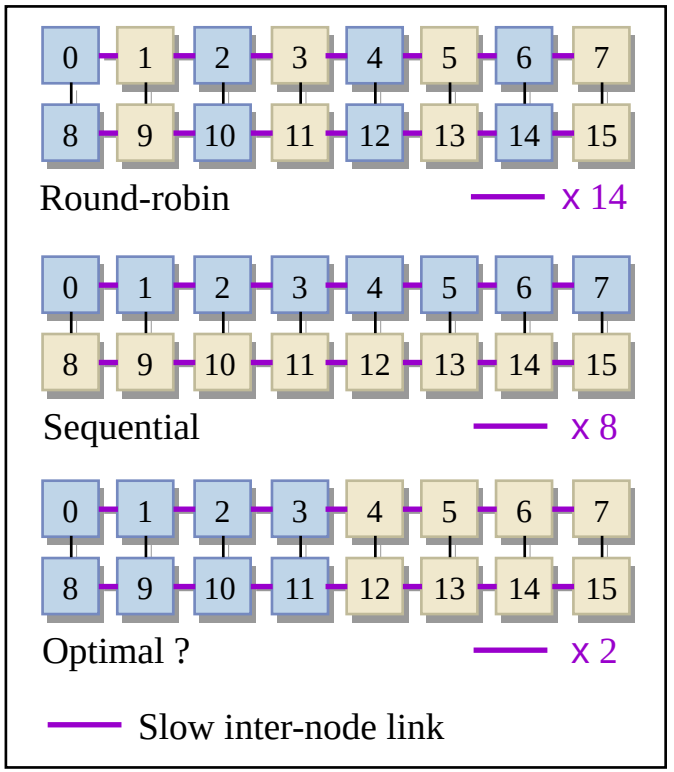


Figure by MIT OpenCourseWare.

- Fitting application topology onto hardware topology
  - Topologies for most MPI implementations do not provide useful re-mapping
  - Wanted: Minimizing communications over slower links

src:HLRS

- If many (or even more than one) processes need to communicate over the slower link network is saturated

Motivations

- Multicore machines abound:
  - Seems far more natural to share memory between cores on the same chip avoiding message passing.
- Waste of system resources:
  - Even with Domain Decomposition some memory gets replicated (plus MPI system buffer memory)
  - Memory bandwidth waste from extra copies
  - Cache pollution from message passing
  - O/S overhead managing processes (heavyweight)

Motivations (cont)

- MPI scaling issues:
  - Collective communication scalability issues
  - Load imbalance with larger # of processes
  - Increasing contention for network resources:
    - Sharing of interface, network link bandwidth limit
    - Latency becomes less predictable
  - For constant problem size (weak scaling) message sizes get smaller:
    - More latency dependence
    - Less efficient use of the network

Basic concept

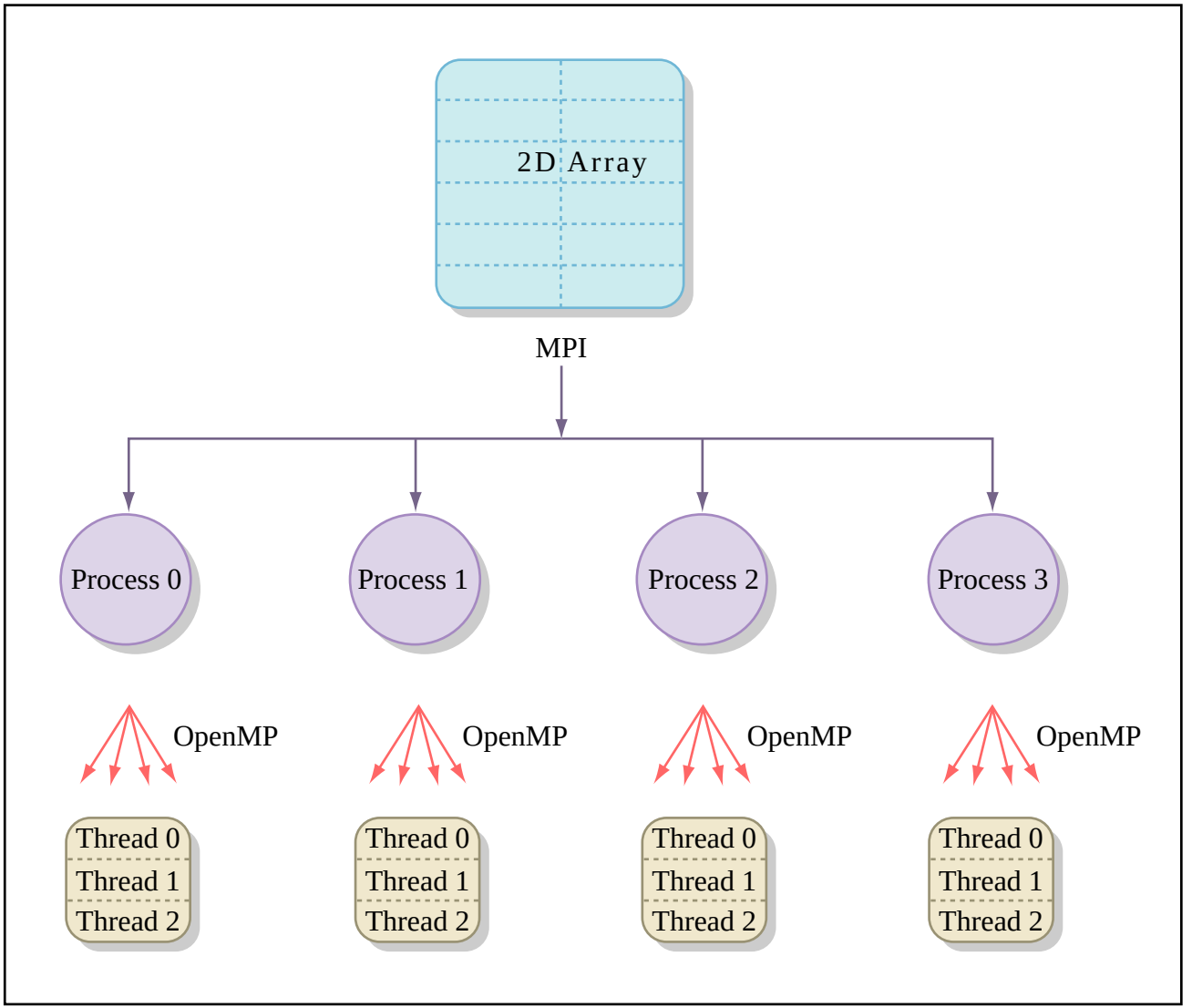


Figure by MIT OpenCourseWare.

src:EPCC

Pros and Cons

- Memory savings
- Better load balancing opportunities
- Less (larger?) off-node MPI messages
- Scaling opportunities
  - N/M processes
- Hardware speeds
- Performance varies
  - Optimization is harder
- MPI communication may not be threaded
- Thread woes: false sharing, etc.
- Extra synchronization introduced with OpenMP
  - Fork/Join & barrier

Basic shared memory options

- OpenMP
  - Evolving standard
  - Easier to use
    - Can be combined with autoparallelization
    - Restricted flexibility
  - Designed for HPC
  - Easier to combine with 3rd party code
- Pthreads
  - POSIX Standard
  - Cumbersome to use
    - But full flexibility
  - Heavyweight
  - Designed for systems apps and not HPC
    - But on Linux systems at least, it lies below the OpenMP layer.

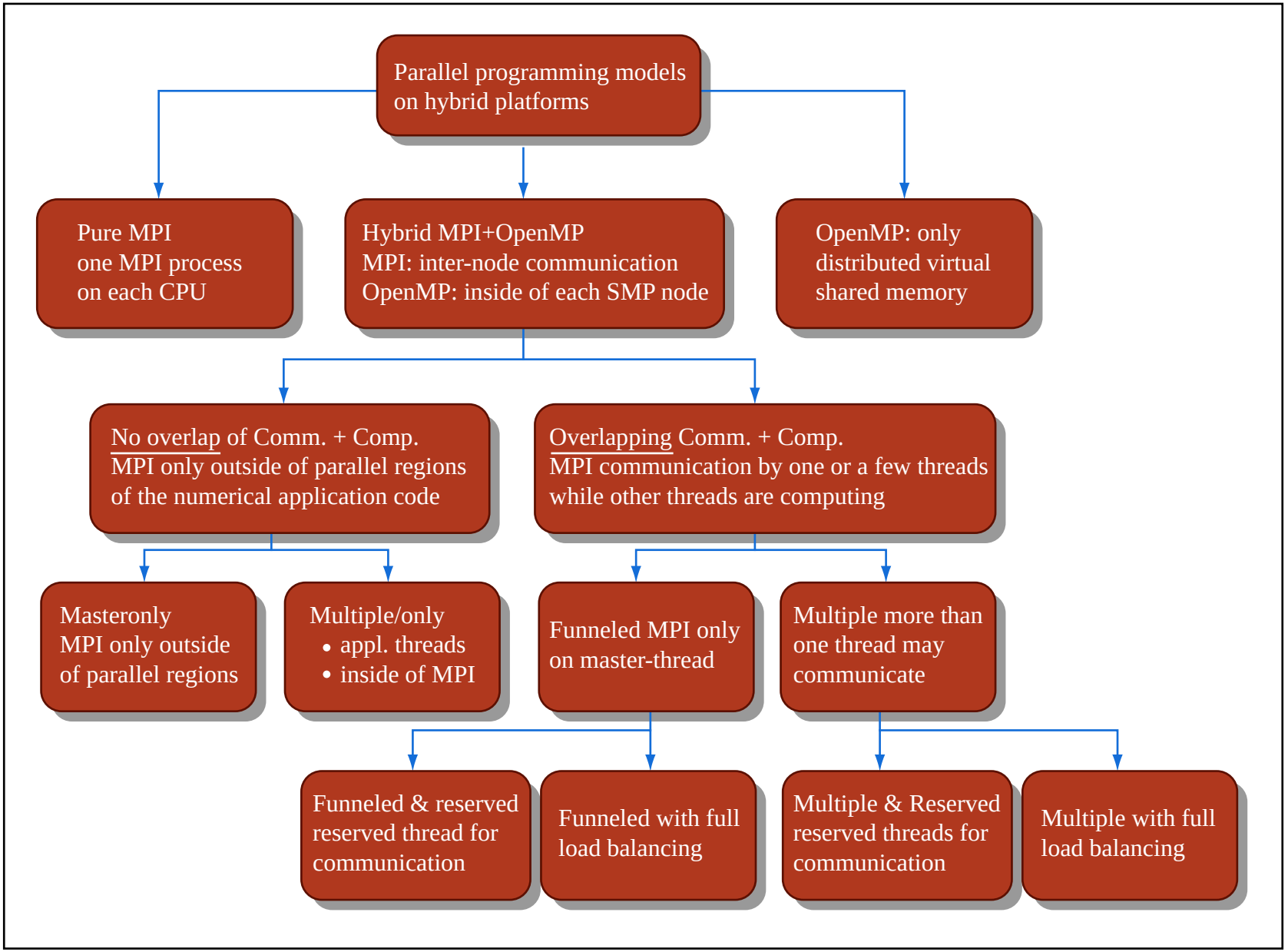
So when to do it?

- Codes where MPI scaling suffers when OpenMP does not for small # of procs
  - If both scale bad but for different reasons it may still be beneficial to go hybrid as one may compensate for the other one's deficiencies (improbable but possible)
- Severe load imbalance *might* become less important for a hybrid code.
  - If oversubscription and dynamic increase of threads are allowed, load imbalance handled for large SMPs.
- Algorithm has finite scalability for MPI

When to do it cont.

- If the algorithm is by design fine-grained, a two-level design can relegate the coarse grain to MPI
- Replicated data are a bottleneck to solving larger problems (replication reduced within the node)
- Existing OpenMP code can be easily extended to MPI at the outermost loop level. (Rare IMHO)
- MPI implementation issues:
  - Restricted # of processes in each node for fast comms
  - Slow intra-node MPI performance

A multitude of options



src: HLRS

Writing hybrid code

- From a sequential code,
  - first parallelize with MPI
  - and then add OpenMP (at the same or different level)
- From an MPI code add OpenMP
- From an OpenMP code, think of a higher level parallelization strategy (as if it were sequential)
- Consider different combinations of OpenMP threads per MPI task and test various OpenMP scheduling options

Master-only hybrid code

- Most obvious strategy is Master-only:
  - Take an MPI code and parallelize using OpenMP the code in between MPI calls:
    - Usually done at a fine (loop) level – the easiest approach
    - Better still if done at a SPMD level with fewer parallel regions and synchronization points
    - We assume that the potential for parallelism within each MPI process is substantial:
      - Large computational work in loops
      - Natural two-level domain decomposition can be extracted
      - Little communication/synchronization between threads needed

Matrix-vector multiplication

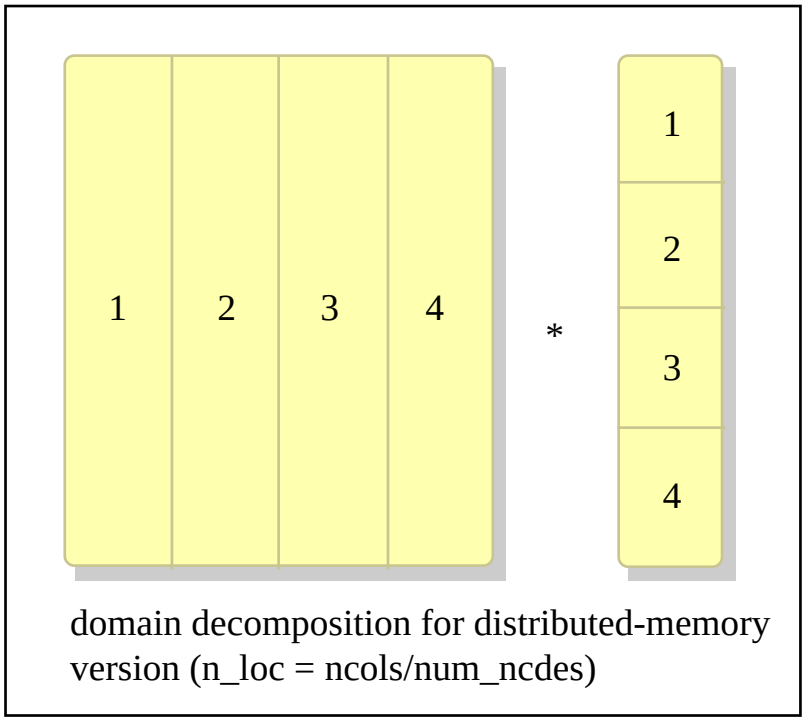
Consider matrix-vector multiplication: A.b = c

- First, the serial loop:

```
DO j=1,ncols
   DO i=1,nrows
      c(i)=c(i)+a(i,j)*b(j)
   END DO
END DO
```

- Second, the distributed-memory version:

```
DO j=1,n_loc ! My local part
   DO i=1,nrows
      c(i)=c(i)+a(i,j)*b(i)
   END DO
END DO
CALL MPI_REDUCE_SCATTER(c) ! Update c (global sum and broadcast)
```



domain decomposition for distributed-memory version (n_loc = ncols/num_ncdes)

Figure by MIT OpenCourseWare.

src:IBM

Matrix-vector multiplication cont.

```
c=0.0
!$OMP PARALLEL shared (c), private (c_loc)
c_loc=0.0
DO j=1,n_loc
!$OMP DO PRIVATE (i)
   DO i=1,nrows
      c_loc(i)=c_loc(i)+a(i,j)*b(j)
   END DO
!$OMP END DO NOWAIT
END DO

!$OMP CRITICAL
DO i=1,nrows
   c(i)=c(i)+c_loc(i)
END DO
!$OMP END CRITICAL
!$OMP END PARALLEL
CALL MPI_REDUCE_SCATTER(c)
```

sets global copy of c to zero

creates local copy of c and sets it to zero

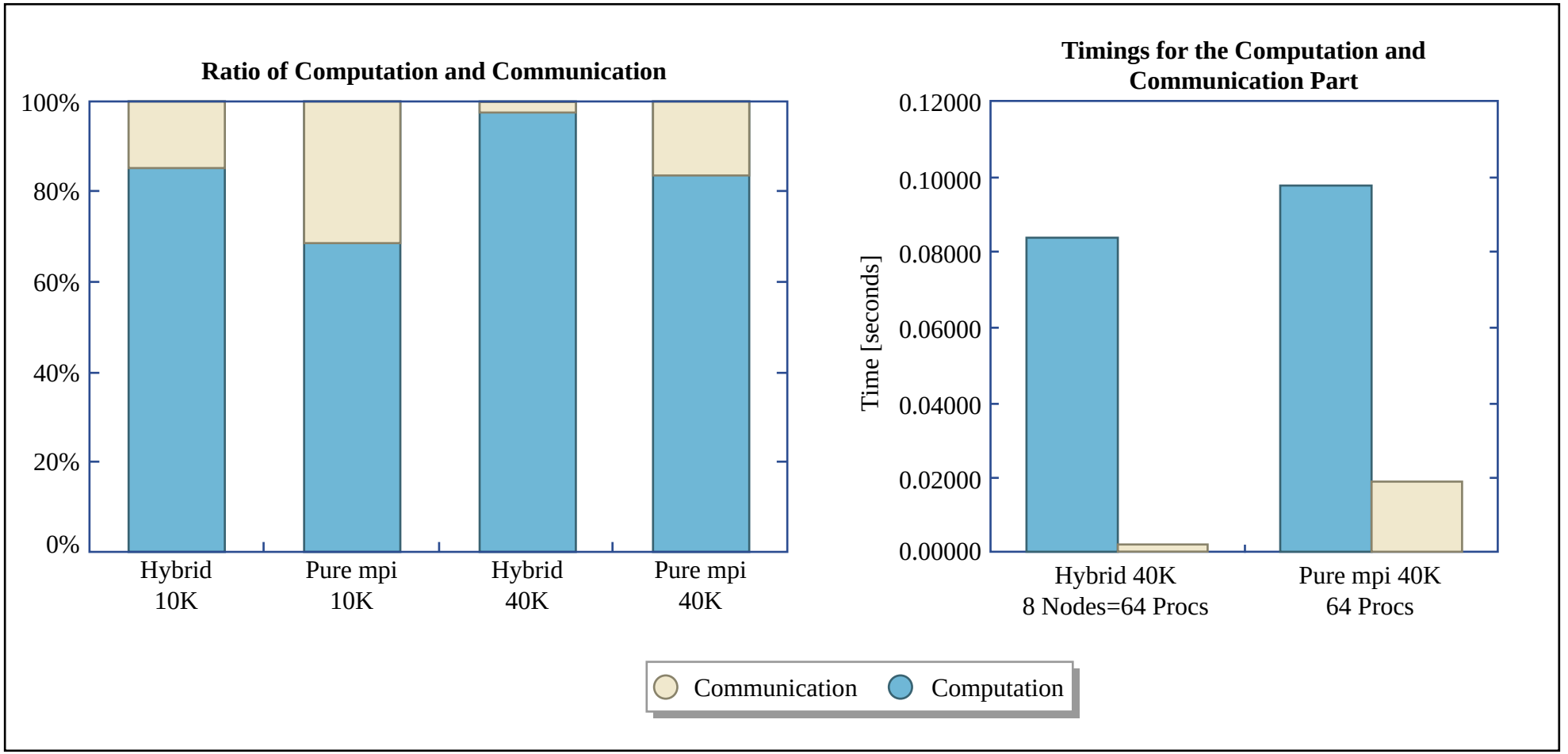
In the critical section, a single thread updates the global copy of c.

src:IBM

Example use

- Make sure you propagate OMP_NUM_THREADS to all your processes (via mpirun/mpiexec or dotfile)
  - Or use OMP_SET_NUM_THREADS or hardcode them
- For production do not oversubscribe a node:
  - (# of MPI procs per node) x (# of OpenMP threads per MPI proc) <= (number of processors in a node).
  - For “fat” SMP nodes both numbers above are likely to be > 1 and you may want to leave a processor free to handle O/S tasks.

Savings in communications



src:Hitachi&LRZ/RRZE

Working with master-only

- MPI communications outside parallel regions is always safe.
  - High thread startup-winddown overhead
- Master region on the other hand needs a barrier:

```
#pragma omp barrier /* this may not be necessary */
#pragma omp master /* other threads idle waiting for the master */
    MPI_Send(...)
#pragma omp barrier /* needed to ensure other threads can modify
   the send buffer */
```

Common Master-only Problems

- Idle (and/or sleeping) cpus
- Utilizing the full inter-node bandwidth
  - Less but larger inter-node messages
  - Contrast with the saturation problem for MPI
    - Both are undesirable scenarios
- Fine grain OpenMP problems
  - False sharing
  - Synchronization (that also causes cache flushes)
- On the other hand: minimize programming effort

Beyond master only

- If the MPI implementation is thread-safe for use by multiple threads at the same time, all threads can partake in message passing
- Otherwise less restrictive variants of master-only for portable & correct code- call MPI_* from:
  - Within MASTER regions: same as Master-only with quite possibly less fork-join overhead, explicit barrier
  - Within SINGLE regions: dangerous, implicit barrier,
  - Within CRITICAL regions: safe for all to partake

MPI-2 support

- MPI-2 Initialization for threaded MPI processes
  - MPI_Init_thread(argc, argv, required, provided)
    - Test for required, check value in provided
    - MPI_THREAD_SINGLE: one user thread
    - MPI_THREAD_FUNNELED: master-only
    - MPI_THREAD_SERIALIZED: serialized MPI calls
    - MPI_THREAD_MULTIPLE: many concurrent calls.
    - Use instead of MPI_Init() from the “main thread”
    - Match with MPI_Finalize() from the same thread
    - Provided depends on how library used, runtime args etc.

Single vs. Funnelled vs. Serialized

```
!$OMP PARALLEL DO
        do i=1,1000
                a(i) = b(i)
        end do
!$OMP END PARALLEL DO

        call MPI_RECV(b,...)

!$OMP PARALLEL
!$OMP DO
        do i=1,1000
                c(i) = b(i)
         end do
!$OMP END DO NOWAIT
! do more work
!$OMP END PARALLEL
```

```
!$OMP PARALLEL
!$OMP DO
        do i=1,1000
                a(i) = b(i)
        end do
!$OMP END DO
!$OMP MASTER
        call MPI_RECV(b,...)
!$OMP END MASTER
!$OMP BARRIER
!$OMP DO
        do i=1,1000
                c(i) = b(i)
         end do
!$OMP END DO NOWAIT
! do more work
!$OMP END PARALLEL
```

```
!$OMP PARALLEL
!$OMP DO
        do i=1,1000
                a(i) = b(i)
        end do
!$OMP END DO
!$OMP SINGLE
        call MPI_RECV(b,...)
!$OMP END SINGLE

!$OMP DO
        do i=1,1000
                c(i) = b(i)
         end do
!$OMP END DO NOWAIT
! do more work
!$OMP END PARALLEL
```

Overlapping computation with communication

```
!$OMP PARALLEL
!$OMP DO
        do i=1,1000
                a(i) = b(i)
        end do
!$OMP END DO NOWAIT
!$OMP SINGLE
        call MPI_RECV(d,...)
!$OMP END SINGLE NOWAIT
!$OMP DO SCHEDULE (GUIDED,n)
        do i=1,1000
                c(i) = b(i)
         end do
! synchronize at the end
!$OMP END DO
! do more work that involves array d
!$OMP END PARALLEL
```

- Designate “main” or first thread to reach the communication point as the MPI thread.
- Dynamic or guided schedule
- Other solution: Spread work that has no dependency on communications among threads statically

Thread reservation

- Easier way to overlap communication with computation
- Either reserve the master (funnelled) or several threads for communication
- Worksharing directives break down – need to distribute the work among the remaining threads manually
- If the ratio of Tcomm/Tcomp doesn't match the distribution of thread roles end up with idle time

Static work scheduling

```
#pragma omp parallel private (i, mythread, numthreads, myrange, mylow)
{
      mythread = omp_get_thread_num();
      numthreads = omp_get_num_threads();
      if (mythread < Nreserved) {
            /* communication work */
      } else {
            myrange = (high-low+1)/(numthreads-Nreserved);
            extras = (high-low+1)%(numthreads-Nreserved);
            if (mythread-Nreserved < extras) {
                  myrange++;
                  mylow = low + (mythread-Nreserved)*myrange;
             } else {
                  mylow = low + (mythread-Nreserved)*myrange + extras;
            }
            for (i=mylow; i<mylow + myrange; i++)
                  { /* computational work */ }
      }
```

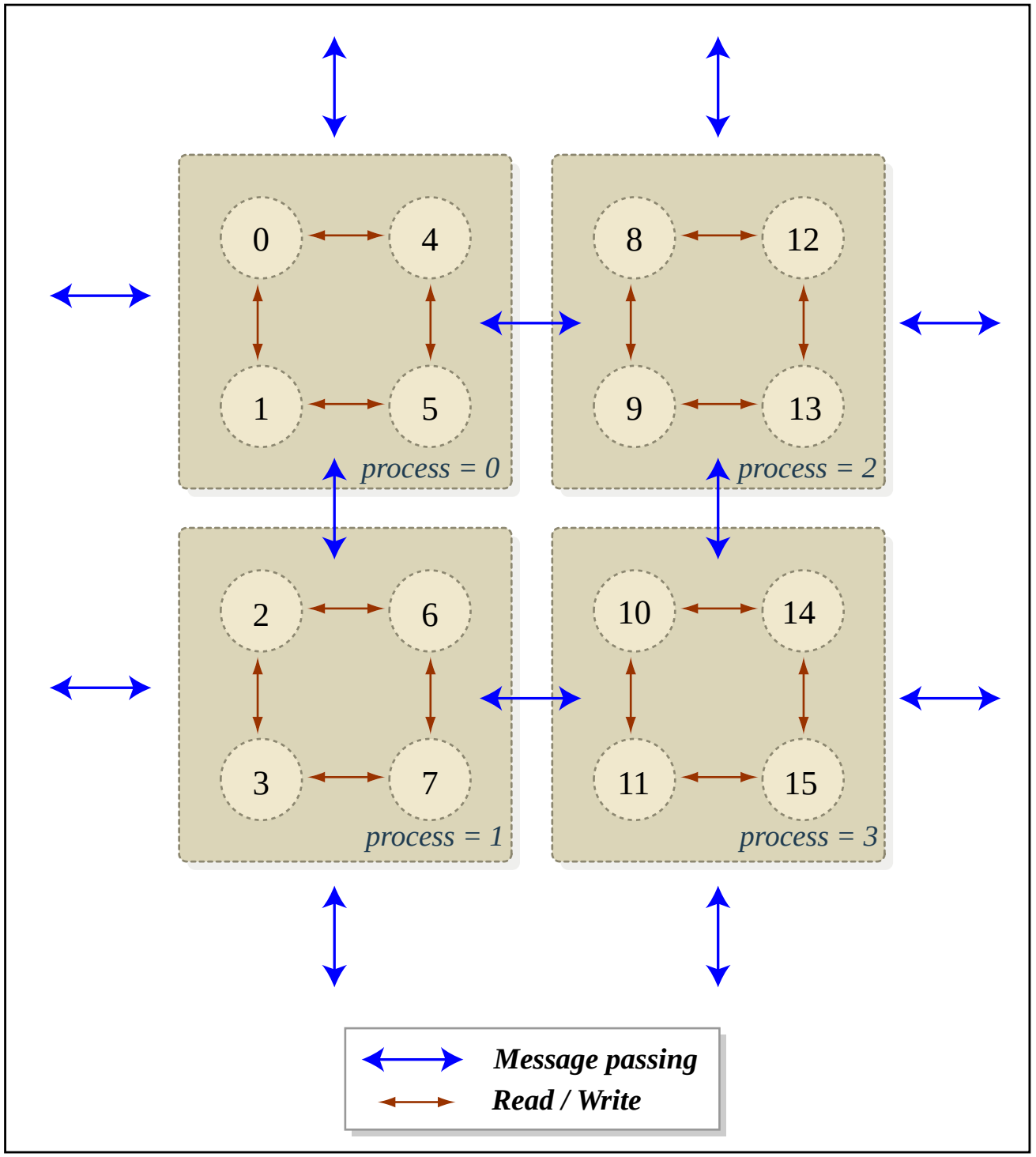
Overlapping Challenges

- an application problem
  - separation of local or halo-based code (**hard**)
- a programming problem
  - thread-ranks-based vs. OpenMP work-sharing
  - Beware of race conditions
- a load balancing problem,
  - if only some threads communicate / compute
- no ideal solution – alternatively avoid it and try:
  - SPMD-model & MPI_THREAD_MULTIPLE

MPI_THREAD_MULTIPLE details

- Make sure that different threads communicating are using different communicators or clearly different tags (as the source process would always be the same).
- When using collective operations make sure that on any given communicator all threads call the MPI routines in the same order.
- Bi-directional exchanges are unlikely to benefit much from this model.

Halo exchanges



src:EPCC

Figure by MIT OpenCourseWare.

Deciding for hybrid

- There is an obvious way to have two-level //
  - Single level parallel codes can be converted to dual level but usually the performance is not any better
- The resulting code should not be a nightmare to code and maintain
  - Master-only usually clean but not always performing
  - Pthread code can be very ugly to upkeep
- Early investigations at least should show that it is similar in performance to an all-MPI code

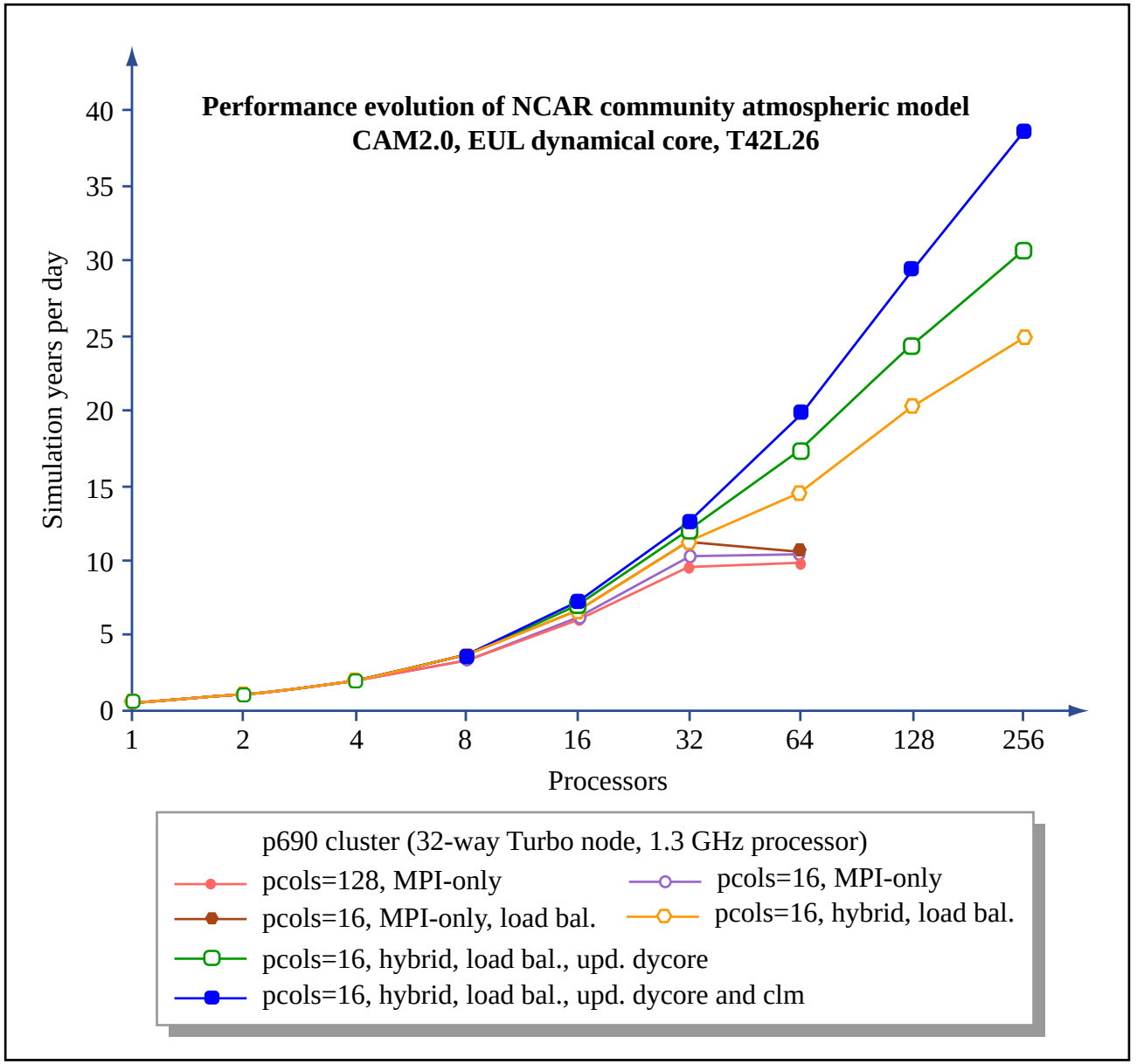
Tests for hybrid

- Test small scale OpenMP (2 or 4 processor) vs. all MPI to see difference in performance.
  - We cannot expect OpenMP to scale well beyond a small number of processors, but if it doesn't scale even for that many it's probably not worth it.
- Test the network to see whether one set of MPI processes can saturate the bandwidth between two nodes
  - Master-only allows for cleaner code usually.

Performance issues

- Can be a success story:
  - MM5 weather forecasting mode
    - 332MHz IBM SP Silver (PPC604e) nodes:
      - Very imbalanced, slow network for the processors
    - 64 MPI procs: 1755 secs (494 comm secs)
    - 16(x4) MPI procs: 1505 secs (281 comm secs)
- Very frequently however hybrid ends up being slower than pure MPI for the same number of processors. Easy to be discouraged by following the literature.

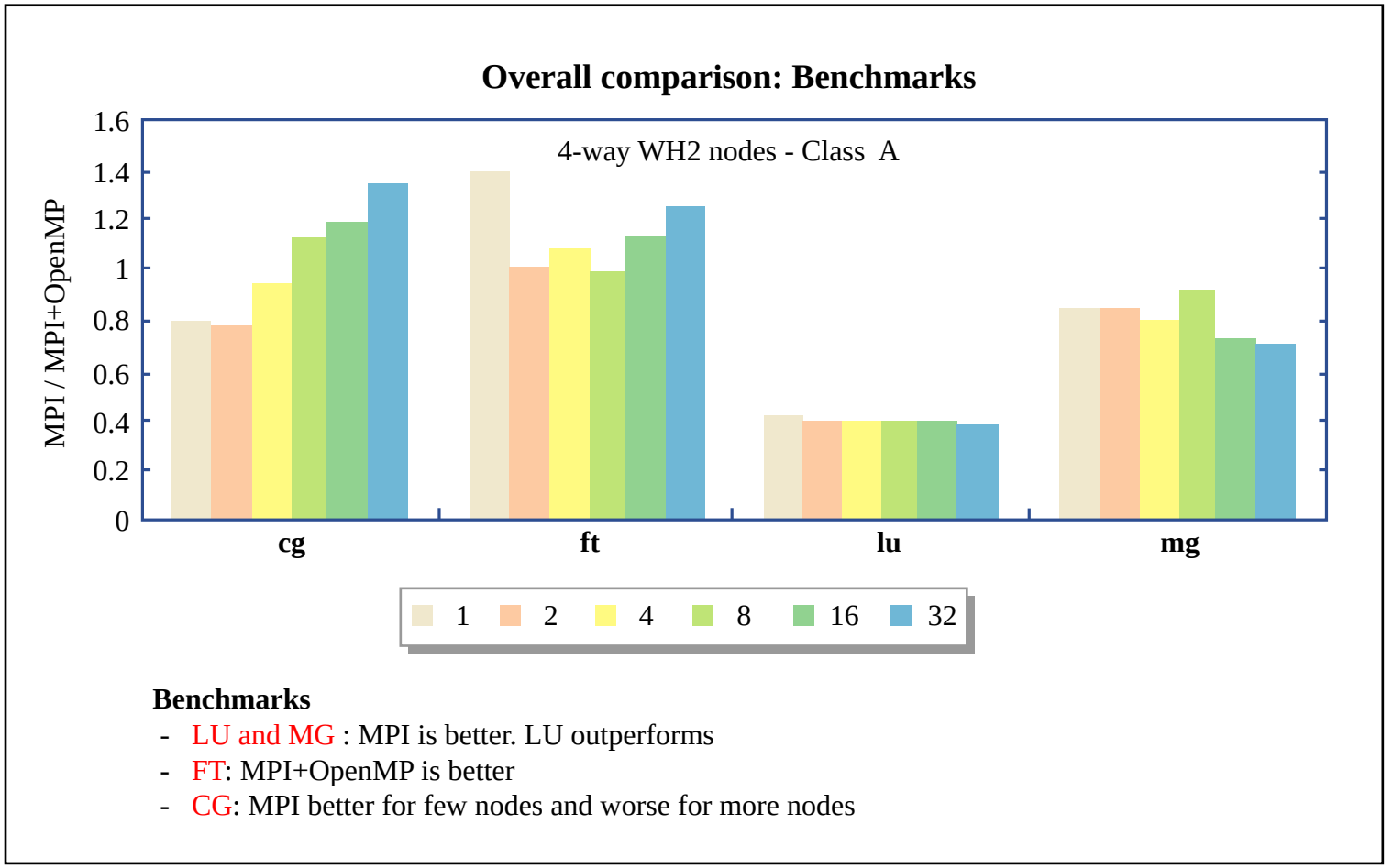
Extending scalability



Pat Worley ORNL
T42L26 grid
128 longitude
64 latitude
26 vertical
MPI latitude-only decomposition

Figure by MIT OpenCourseWare.

NAS Parallel Benchmarks



Benchmarks

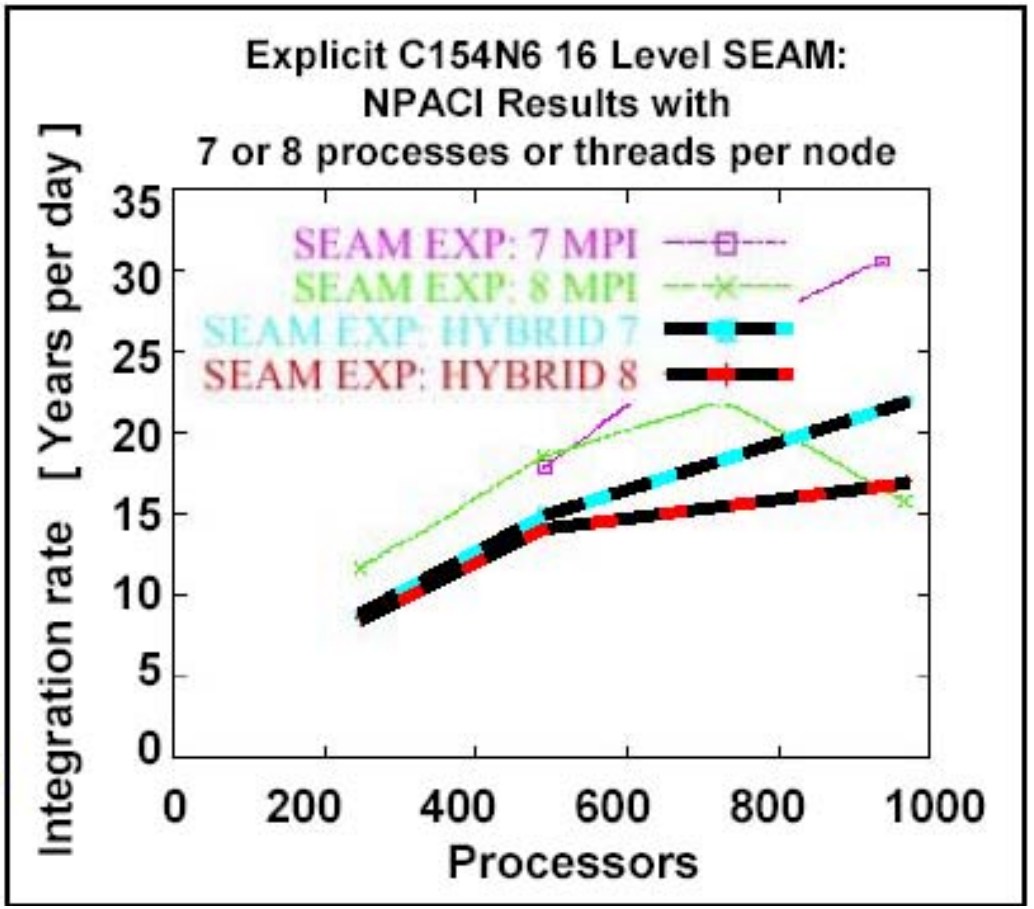
- LU and MG : MPI is better. LU outperforms
- FT: MPI+OpenMP is better
- CG: MPI better for few nodes and worse for more nodes

src: Franck Cappello & Daniel Etiemble @lri.fr

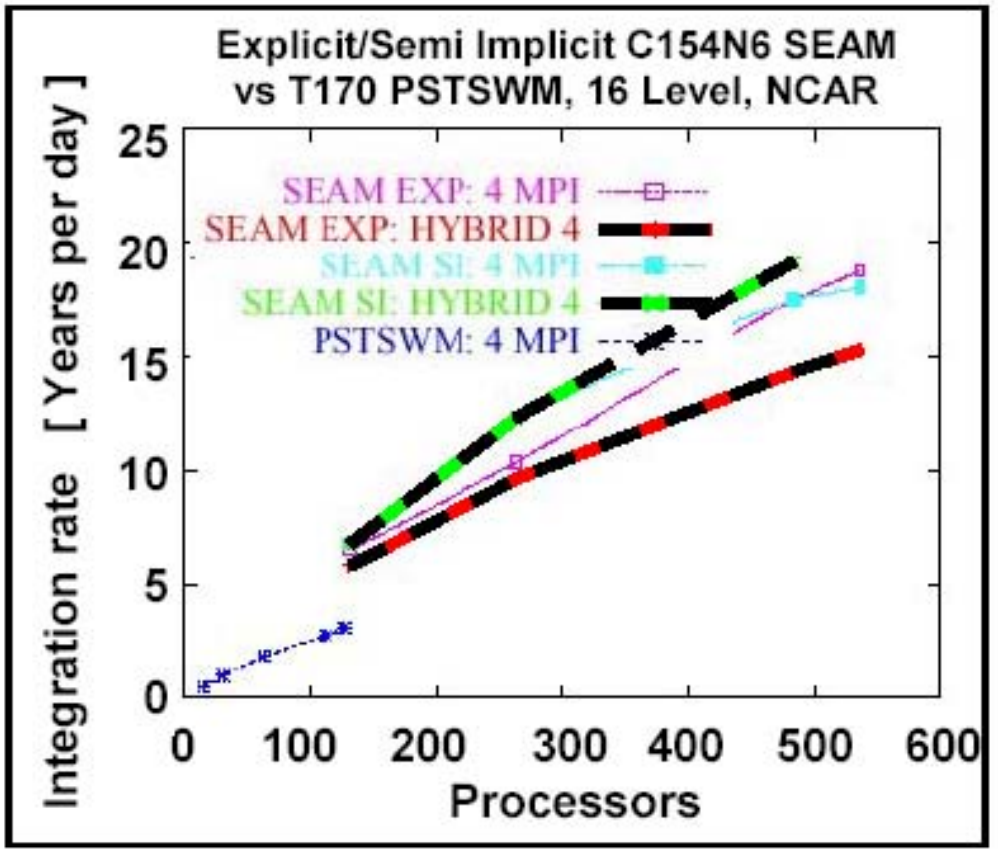
Figure by MIT OpenCourseWare.

Which approach is faster?

- Pure MPI versus Hybrid MPI+OpenMP (Masteronly)
- What's better?
  → it depends on?



Courtesy of Rolf Rabenseifner, HLRS, University of Stuttgart. Used with permission.



Figures: Richard D. Loft, Stephen J. Thomas, John M. Dennis:
Terascale Spectral Element Dynamical Core for Atmospheric General Circulation Models.
Proceedings of SC2001, Denver, USA, Nov. 2001.
http://www.sc2001.org/papers/pap.pap189.pdf
Fig. 9 and 10.

Courtesy of Rolf Rabenseifner, HLRS, University of Stuttgart. Used with permission.

src:HLRS

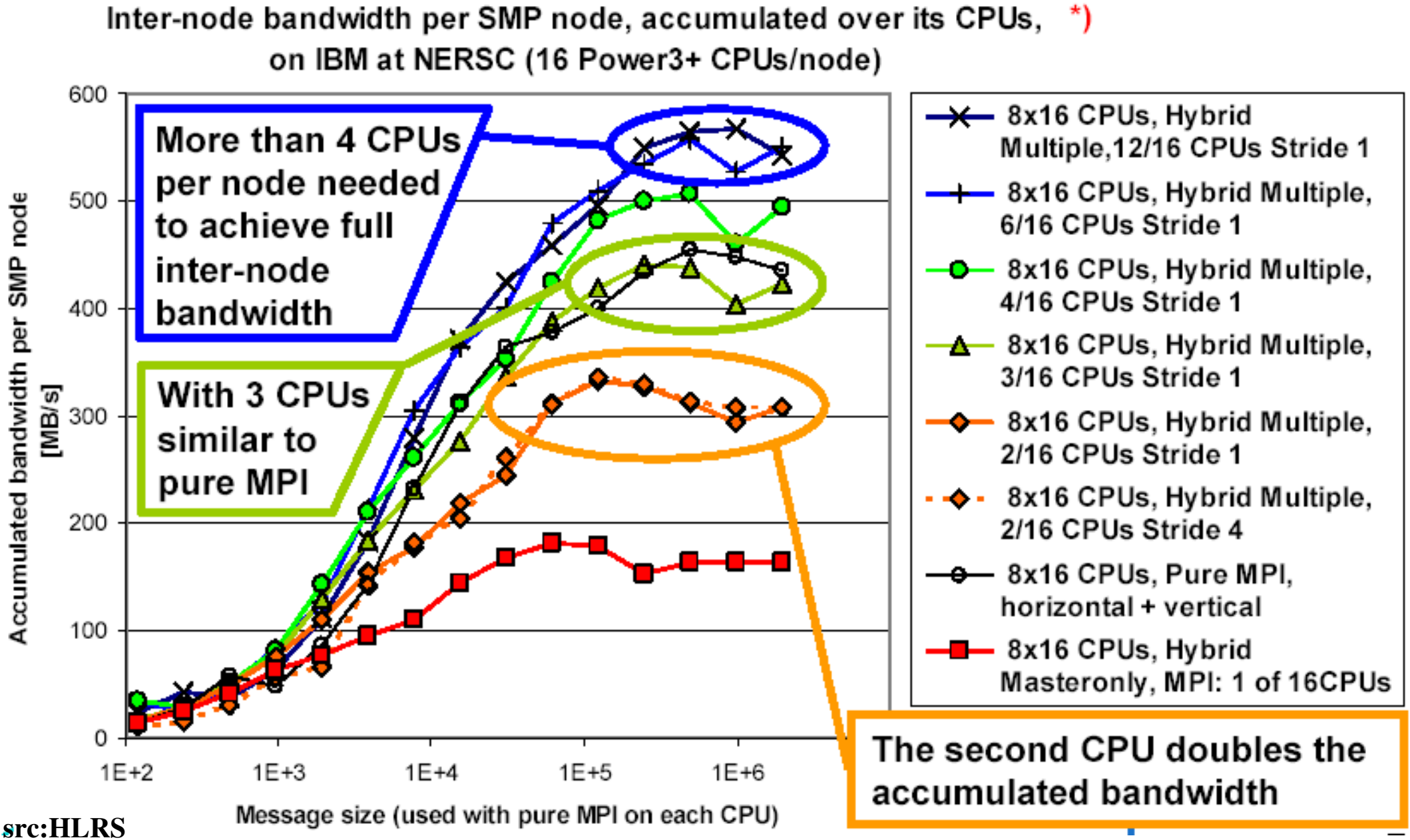
No panacea (perfect solution)

- MPI only problems
  - Topology mapping
  - Unnecessary intra-node communication
  - Network link saturation
- MPI-OpenMP problems
  - Sleeping threads (master-only)
  - OpenMP overhead
    - Thread fork/join
    - Cache flushing on synchronization
    - Worse spatial locality
  - Utilizing the full inter-node bandwidth

Tuning Opportunities

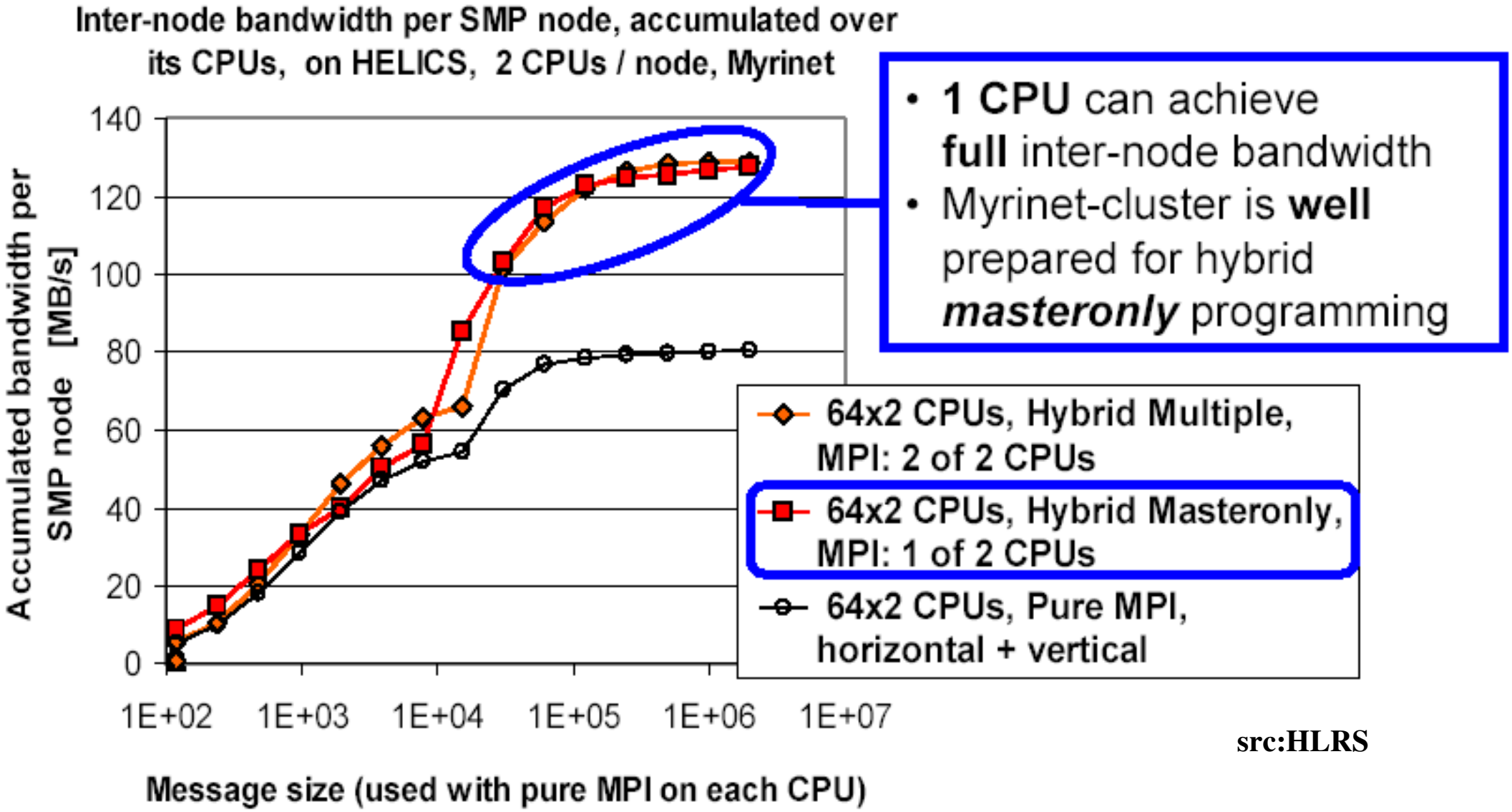
- Speed up MPI routines:
  - Threads may copy non-contiguous data into contiguous buffers (instead of derived datatypes)
  - Use multiple threads communicating to utilize inter-node bandwidth
  - Better still employ multi-threaded MPI library.
  - Otherwise use only hardware that can saturate network with 1 thread
- For throughput use idling CPUs for “niced” apps

The Rabenseifner tests



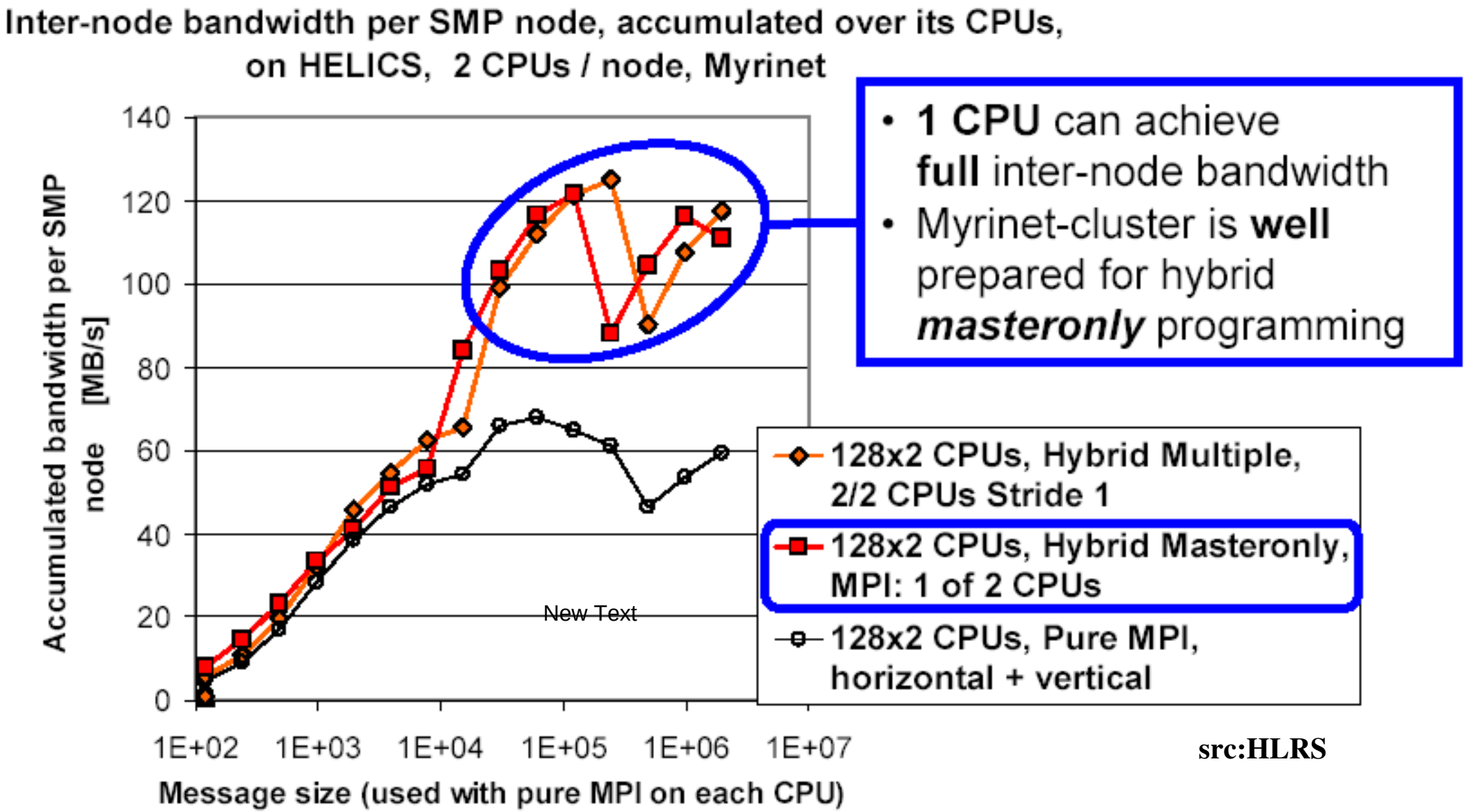
Courtesy of Rolf Rabenseifner, HLRS, University of Stuttgart. Used with permission.

Myrinet clusters for Master-only



Courtesy of Rolf Rabenseifner, HLRS, University of Stuttgart. Used with permission.

Myrinet clusters for Master-only cont



Courtesy of Rolf Rabenseifner, HLRS, University of Stuttgart. Used with permission.

src:HLRS

Comparing inter-node bandwidth with CPU performance

*) Bandwidth per node:
totally transferred bytes on the network
/ number of nodes / wall clock time

Courtesy of Rolf Rabenseifner, HLRS, University of Stuttgart. Used with permission.

All values: aggregated over one SMP nodes. *) mess. size: 16 MB +) 2 MB	Master -only, inter-node [GB/s]	pure MPI, inter-node [GB/s]	Master-only bw / max. intra-node bw	pure MPI, intra-node [GB/s]	memo-ry band-width [GB/s]	**Peak &** *Linpack* perfor-mance Gflop/s	max.inter -node bw / peak & *Linpack* perf. B/Flop	nodes*CPUs
Cray X1,shmem_put preliminary results	9.27	**12.34**	75 %	33.0	136	**51.2** *45.03*	0.241 *0.274*	**8 * 4 MSPs**
Cray X1, MPI preliminary results	4.52	**5.52**	82 %	19.5	136	**51.2** *45.03*	0.108 *0.123*	**8 * 4 MSPs**
NEC SX-6 global memory	**7.56**	4.98	100 %	78.7 93.7+)	256	**64** *61.83*	0.118 *0.122*	**4 * 8 CPUs**
NEC SX-5Be local memory	2.27	**2.50 a)**	91 %	35.1	512	**64** *60.50*	0.039 *0.041*	**2 *16 CPUs** a) only with 8
Hitachi SR8000	0.45	**0.91**	49 %	5.0	32 store 32 load	**8** *6.82*	0.114 *0.133*	**8 * 8 CPUs**
IBM SP Power3+	0.16	**0.57+)**	28 %	2.0	16	**24** *14.27*	0.023 *0.040*	**8 *16 CPUs**
SGI O3000, 600MHz	0.43+)	**1.74+)**	25 %	1.73+)		**4.8** *3.64*	0.363 *0.478*	**16 *4 CPUs**
SUN-fire (prelimi.)	0.15	**0.85**	18 %	1.68				**4 *24 CPUs**
HELICS Dual-PC cluster with Myrinet	0.118 +)	**0.119 +)**	100 %	0.104 +)		**2.80** *1.61*	0.043 *0.074*	**128 *2 CPUs**

Alternatives

- So coding hybrid codes involves:

  1)More complicated, two-level logic

  2)Further opportunities for bugs with sloppy coding

  3)Unexpected performance pitfalls

  4)Extra performance variation platform to platform

- What alternatives are there for clusters?
  - MPI + multithreaded parallel libraries
  - Scalable OpenMP: If one cares the most about (1) & (2) and one already has an OpenMP code available

Multi-threaded parallel libraries

- Master-only paradigm only the drudgery of the OpenMP code moves inside the library
- Example: MPI code calling multithreaded LAPACK/BLAS library on each process
  - Useful provided there is enough work to go around
  - Allows the use of fast direct solvers
- Hybrid code with master-only segment calling multithreaded library. Problems when other multithreaded parts call serial version of library.

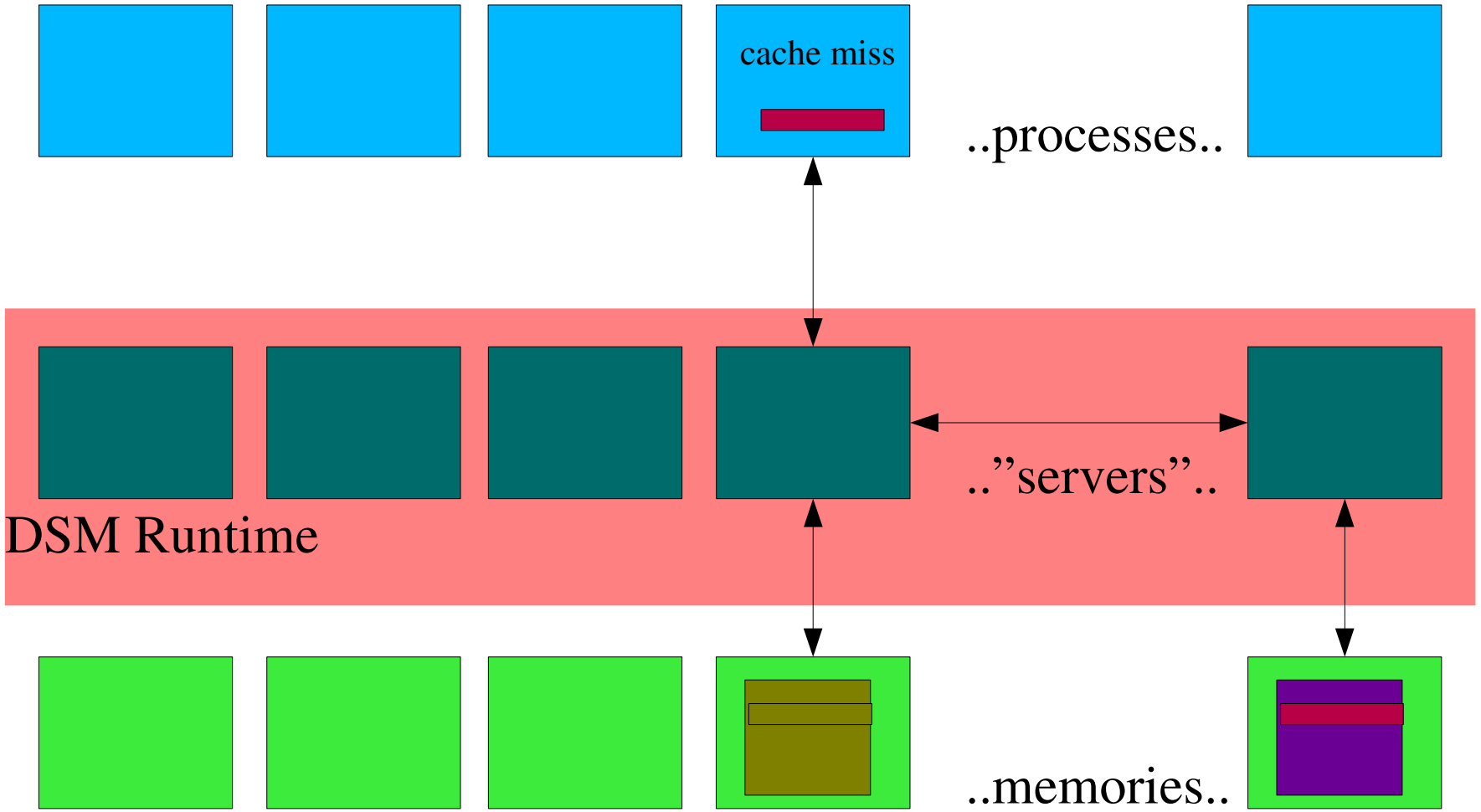
SDSM

- Software Distributed Shared Memory:
  - A user-level runtime system that creates the illusion of a shared memory system on a set of distributed memory machines .
  - aka VDSM or SVM (V for Virtual), constrast with HDSM (Hardware - aka simply as DSM, cc-NUMA etc.) as is the case of most modern Unix servers (IBM Regatta & Squadron, SGI Altix, HP Superdome, Sun Sunfire etc) and multiprocessor AMD Opteron servers.

Enhanced OpenMP runtimes

- Enhanced OpenMP runtime, usually built on top of some distributed shared memory library that:
  - detects accesses to memory at remote nodes and
  - behind the scenes fetches a local copy of that memory location.
  - Handles all cache-coherence issues
  - Essentially replaces user-level message passing with remote gets and puts of variables, usually at a large granularity (most commonly memory page level).

How it usually works



Disadvantages

- Depending on how relaxed the cache coherence protocol is (and the locality of memory accesses), SDSM systems may suffer from much elevated amounts of network traffic
  - Many research SDSM systems have experimented with various tricks (and coherence protocol variations) to help minimize this.
- In general the remote access is triggered by a page miss (which is costly to begin with)

Disadvantages (cont)

- If the code does not walk through memory sequentially then whole memory pages (4KB or more) are fetched for a few cachelines' worth of useful data. Essentially like wasted prefetching.
  - Cache-coherence performance problems are much worse for SDSM systems: False sharing...
  - Generic cache-coherence in hardware is much slower.
  - Need code with each thread working on a distinctly unique part of the dataset for performance. Even then, dataset boundaries should be at page boundaries.

Advantages

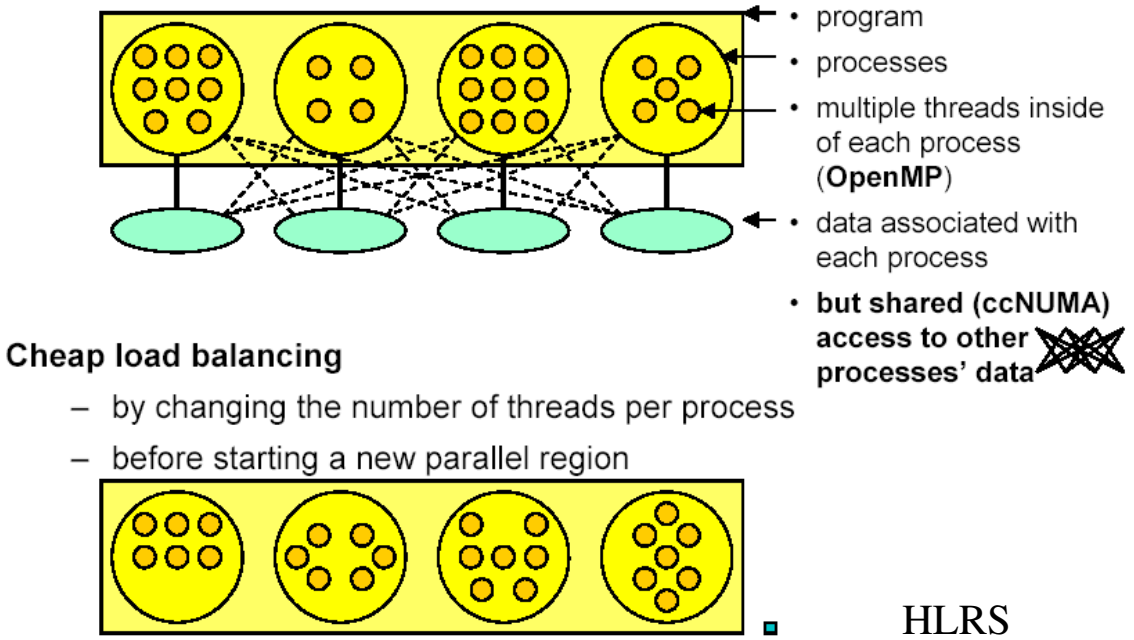
- OpenMP (or in some cases Pthread) code transfers over cleanly.
  - Minimal coding effort, maximum reuse
- Certainly allows memory scaling of codes
- A lot of ground for enhanced performance
- But still for a limited family of codes:
  - SPMD-like OpenMP codes would perform the best
  - Would also be the easiest for MPI+OpenMP

Current SDSM Options

- Last few versions of the Intel compilers include this capability from an evolved version of Threadmarks from Rice.
  - Additional SHAREABLE directive for across-node memory consistency
  - Compile with -cluster-openmp (& -clomp-* options)
- Omni-SCASH (with Score)
  - Part of a complete cluster environment with MPI, distributed C++ libraries and checkpointing.

Multilevel Parallelism

- MLP: OpenMP inside processes, Unix System V shared memory between processes
- Developed by NASA (NAS) and SGI for the Origin series of DSM machines, works on Altix



Courtesy of Rolf Rabenseifner, HLRS, University of Stuttgart. Used with permission.

Summary

- Hybrid programming merges the shared and the distributed memory programming paradigms
- Potentialy superior on dual-cpu SMP clusters
- Care in coding to avoid performance pitfalls
- MPI-2 added support for mixing threads and MPI
- Easiest to use OpenMP when for master-comms
- Alternatives hiding the message exchange from the user exist but have performance issues

Summary cont.

- Master-only paradigm useful only on certain platforms including Myrinet clusters
  - Internally multi-threaded MPI library useful
- Other platforms need extra optimizations to saturate inter-node bandwidth
- Master-only with single-threaded MPI suffers from idle processors during communications
- Difficult to load-balance, hide communication with computation.

Amdahl's Law

Serial time on 1 proc: $T_s = T_{ser} + T_{par}$

Parallel time on P procs: $T_p = T_{ser} + T_{par}/P + T_{pover}(P)$

Parallel overhead:

$$T_{pover}(P) = T_{over}(P) + T_{comm}(P) + T_{sync}(P)$$

Parallel time on 1 proc: $T_p(1) = T_s + T_{over}(1)$

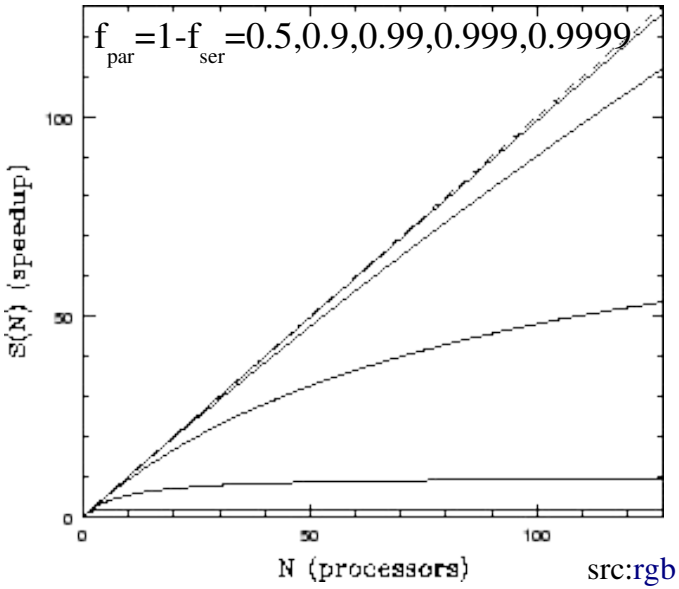
Best case scenario: $T_{pover}(P) = 0$

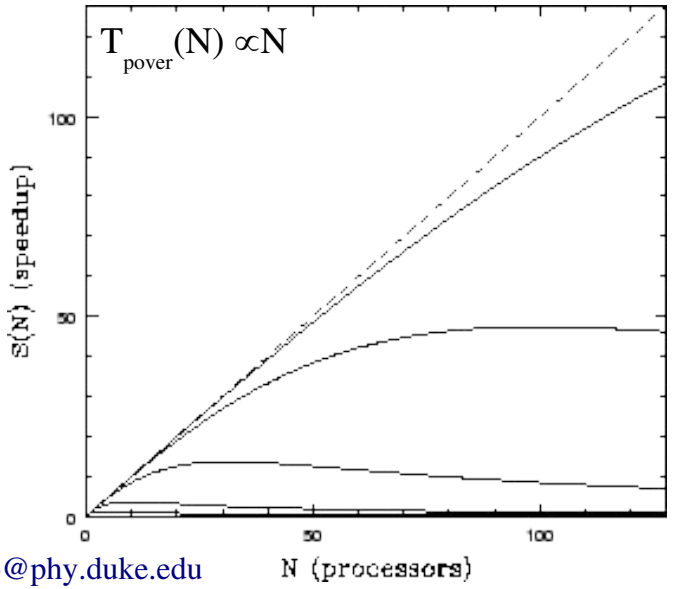
Speedup, $S_p = T_s / T_p$, serial fraction $f_{ser} = T_{ser}/T_s$

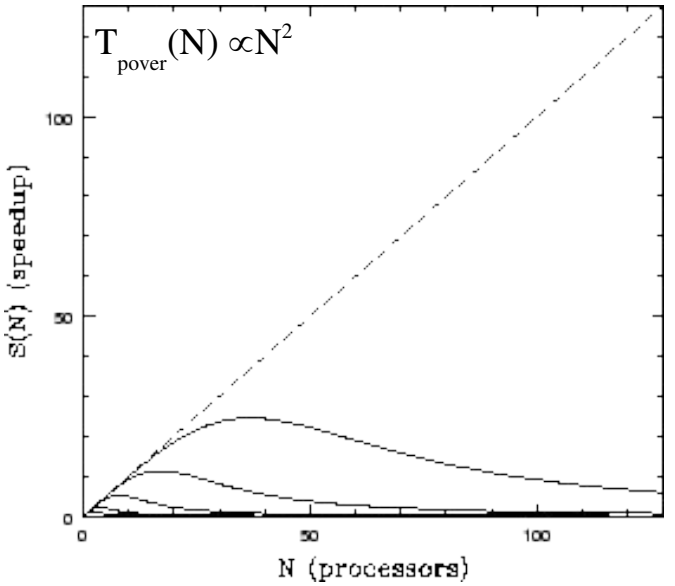
then $\mathbf{S_p = \{f_{ser} + [1 - f_{ser}] / P\}^{-1}}$ so $\mathbf{S_p \leqslant 1 / f_{ser}}$

There's no free lunch!

Courtesy of Robert G. Brown, Duke University. Used with permission.





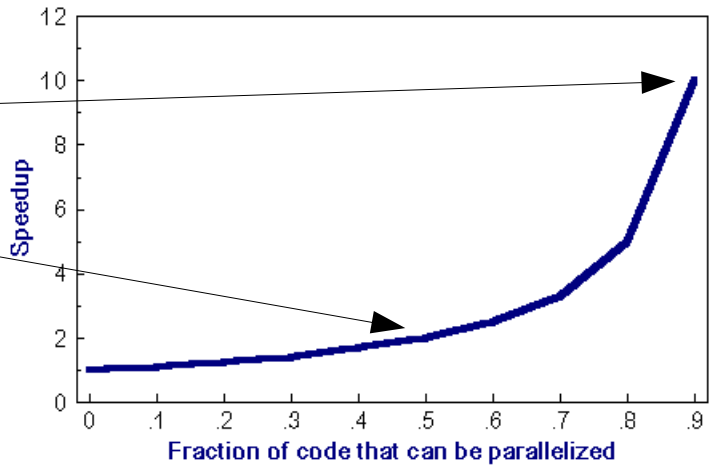


src:rgb@phy.duke.edu

Amdahl's Law

	speedup		
N	f_{par} = .50	f_{par} = .90	f_{par} = .99
10	1.82	5.26	9.17
100	1.98	9.17	50.25
1000	1.99	9.91	90.99
10000	1.99	9.91	99.02

Small problem size:
2D Grid Calculations 85 seconds 85%
Serial fraction 15 seconds 15%
Larger problem size:
2D Grid Calculations 680 seconds 97.84%
Serial fraction 15 seconds 2.16%



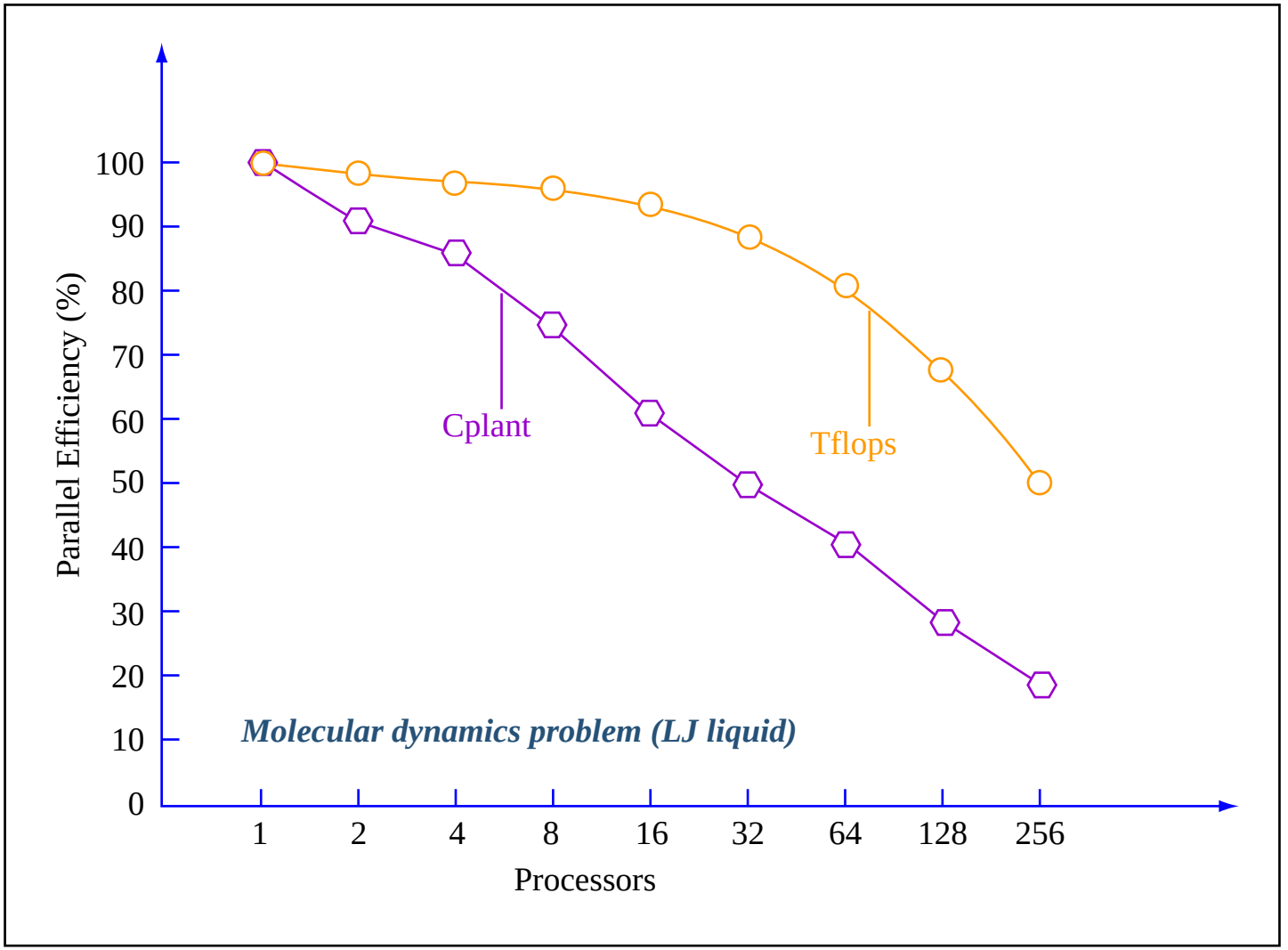
src:LLNL

Courtesy of Lawrence Livermore National Laboratory. Used with permission.

Parallel Scaling & Efficiency

- Two kinds of Scaling:
  - Strong: fixed problem size
    - $S_{Sp} = T_s / T_p$, calculated sometimes as
      - $S_{Sp}(P) = T_p(1) / T_p(P)$ or $S_{Sp}(P) = (T_s(P_N)/P_N)/T_p(P)$
    - Ideally scales linearly, sometimes superlinearly
      - Parallel Efficiency: **E = S/P** (expressed as a %)
  - Weak: problem size W that scales with P so as to keep the effective problem size per processor constant
    - $S_{Wp}(P) = T_p(PW,P) / T(W)_s$, calculated sometimes as $S_{Wp}(P) = T_p(PW,P) / T(W,Q)_p$. Should be close to 1. **E=1/S**

Parallel Efficiency E_{Sp} Example



src: Sandia National Lab

Weak Scaling S_{Wp} Example

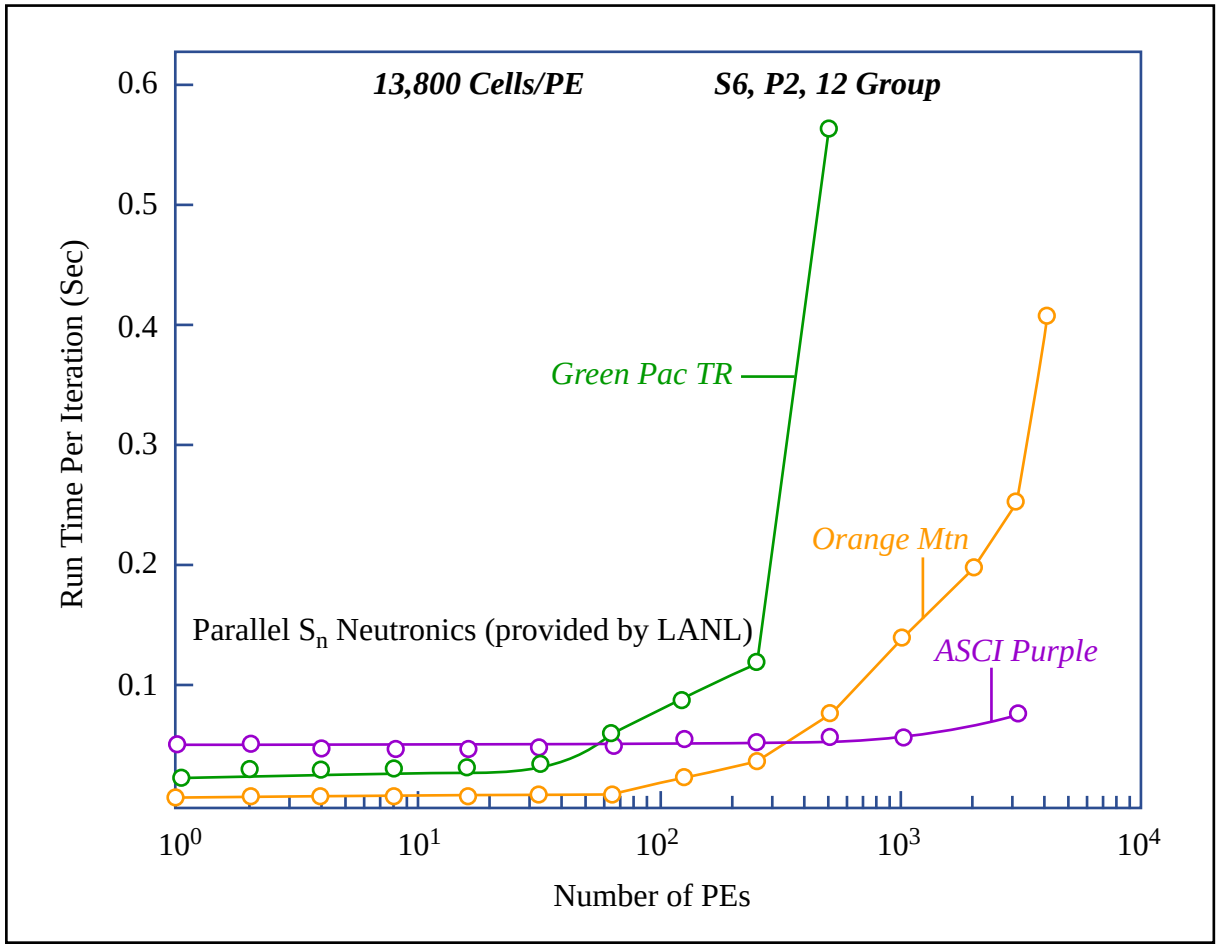


Figure by MIT OpenCourseWare.

src:Sandia National Lab

MPI Performance Modeling

- Ideally one would like a complete performance model.
- Basic performance model for Pt2Pt comms:
  - T = latency + message_size/bandwidth
  - In fact multiple ranges (buffered, synchronous, other)
  - In fact multiple latencies and bandwidth
  - More complicated models: logP, logGP,C^3 etc.
- Even more involved models for collective comms
- Non-blocking comms and overlap of communication with computation further complicates models
- Given a model, one uses the machine and runtime parameters to get an estimated wallclock time.

Performance Rules

However, even in the absence of a detailed or even a rudimentary performance model we can still come up with important rules of thumb!

1. Minimize the ratio of communication to computation
2. Between lots of small messages and one large one choose the latter (less overhead, message aggregation)
3. (At least) for anything more than a few processors collective communications should be a gain.
4. Avoid synchronizations (explicit & implicit)
5. Balance the work (load balance)
6. Overlap communication with computation
7. Perform redundant work if it's cheaper!

Performance Tuning

- Time your code, as a function of processors, problem
  - Important to decide on weak vs. strong scaling
- See at what np bottleneck starts appearing
- Use profiling/tracing tools for that np and over
  - Look at the graphical picture and find hotspots
    - Look for time spent idling (load imbalance)
    - Look for too many small message exchanges
    - Look for obvious collective subroutine patterns
    - Look for too many messages to/from one process at a time
  - Look at time spent in routines and find culprits
  - Try applying the rules (routine, code, algorithm changes)
- Iterate

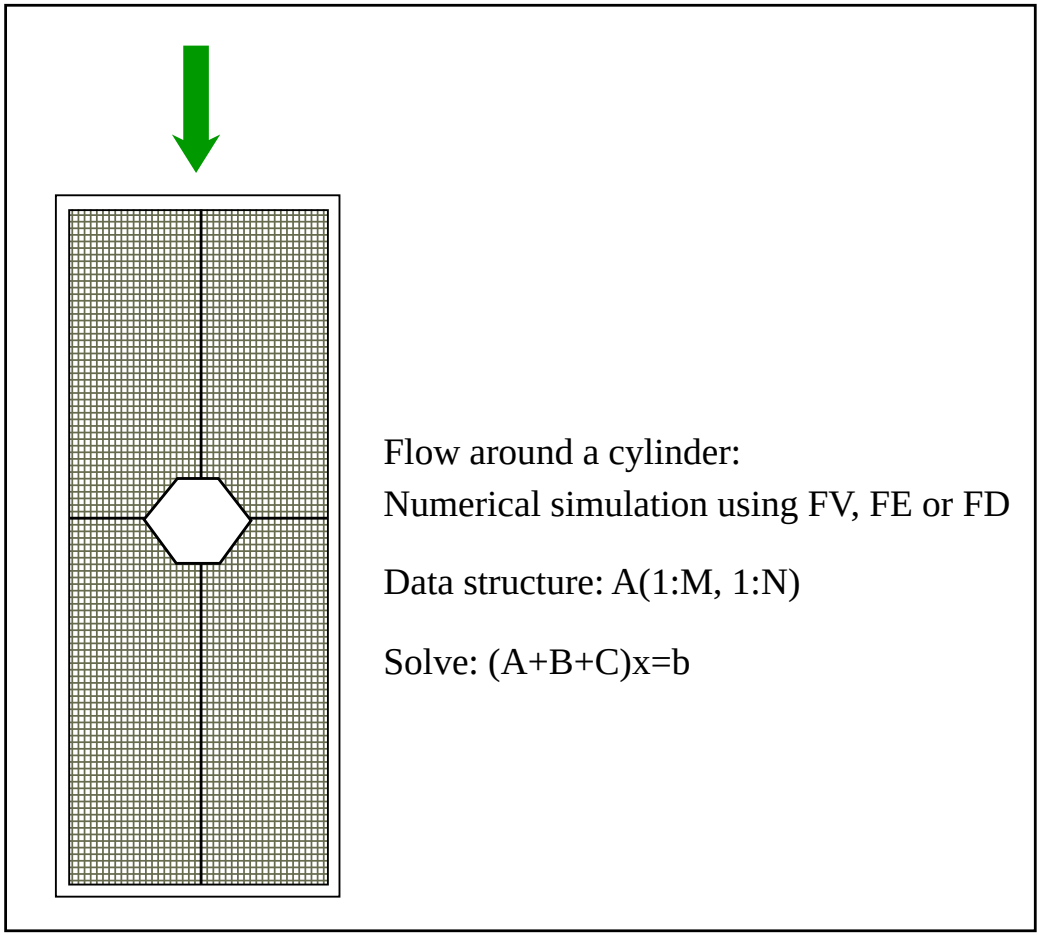
Portable Performance

- Unfortunately, in practice a contradiction in terms!
- Architecture/MPI implementation dependent
- Sometimes problem size dependent!
- So a **very well tuned** code may (preferably under the cover) employ different routines/algorithms for different machines and even problem sizes. Hide complexity with libraries.
- A **well tuned** code on the other hand can be expected to be reasonably performing in a portable manner.
- Employing 3rd party libraries makes this more of the library provider's problem and less of yours.

Programming in Parallel

- Decompose a problem into parts that can be solved concurrently.
  - If no communication is needed, then problem is EP (Embarrassingly Parallel)! :-)
- The decomposition can be in terms of mapping to data structures, mapping to physical entities (regions of space), mapping to different tasks.
- Algorithms and decompositions need to be compatible, requiring as little communication overhead as possible. Use smart partitioners!
- Best serial algorithm is not best parallel one!
  - Think Gauss Seidel vs. Jacobi

Problem Statement



Flow around a cylinder:
Numerical simulation using FV, FE or FD

Data structure: A(1:M, 1:N)

Solve: (A+B+C)x=b

Figure by MIT OpenCourseWare.

Major Options

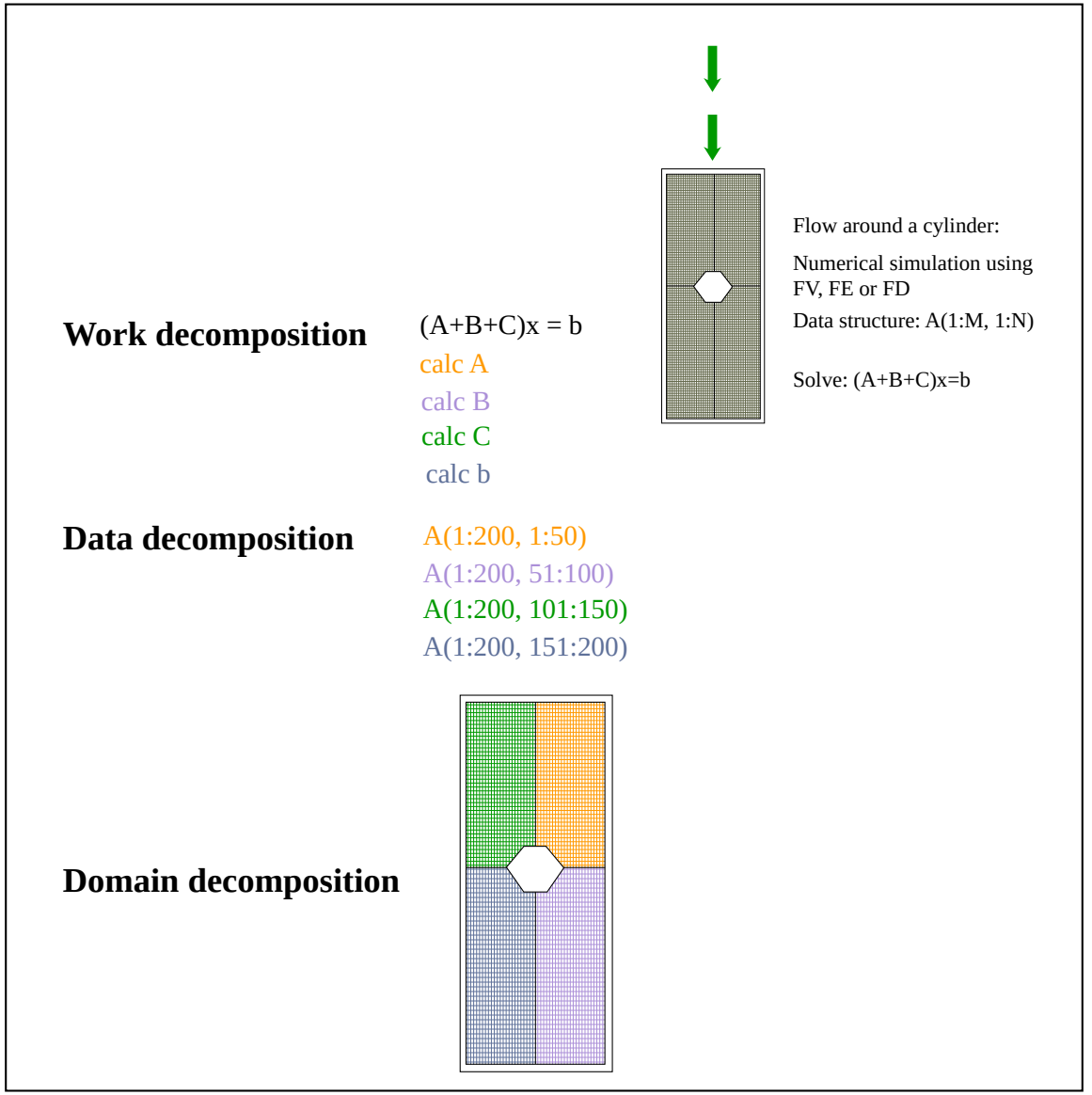
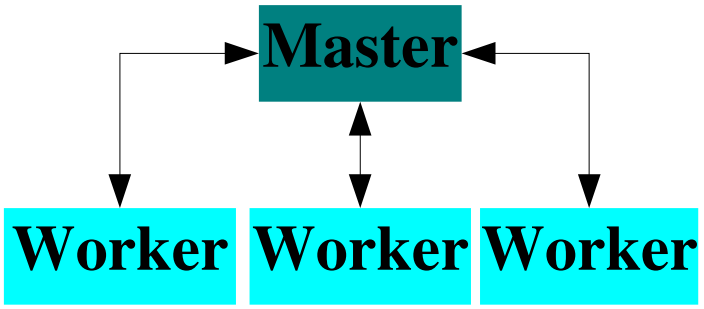


Figure by MIT OpenCourseWare.

Major modes of parallelism

- Master-worker (or task-farming)
  - Embarrassingly parallel
- Domain decomposition
- Array distribution
- Particle/Space distribution
- Pipelined dataflows
- Concurrent workflow execution
- Weakly coupled interdisciplinary models

Master-Worker or (Slave :-)



- Also known as task farming
- Automatic load balance
- Loosely coupled calculations
- Easy implementation

- One master coordinates work, many workers execute it.
  - Worker is in a loop, waiting for a new task, executing it and then sending the result to master.
  - Master is in a loop, sending tasks to the workers, receiving results and operating on them, adjusting the queue of tasks continously.
  - Master notifies termination.

Advanced Task Farming

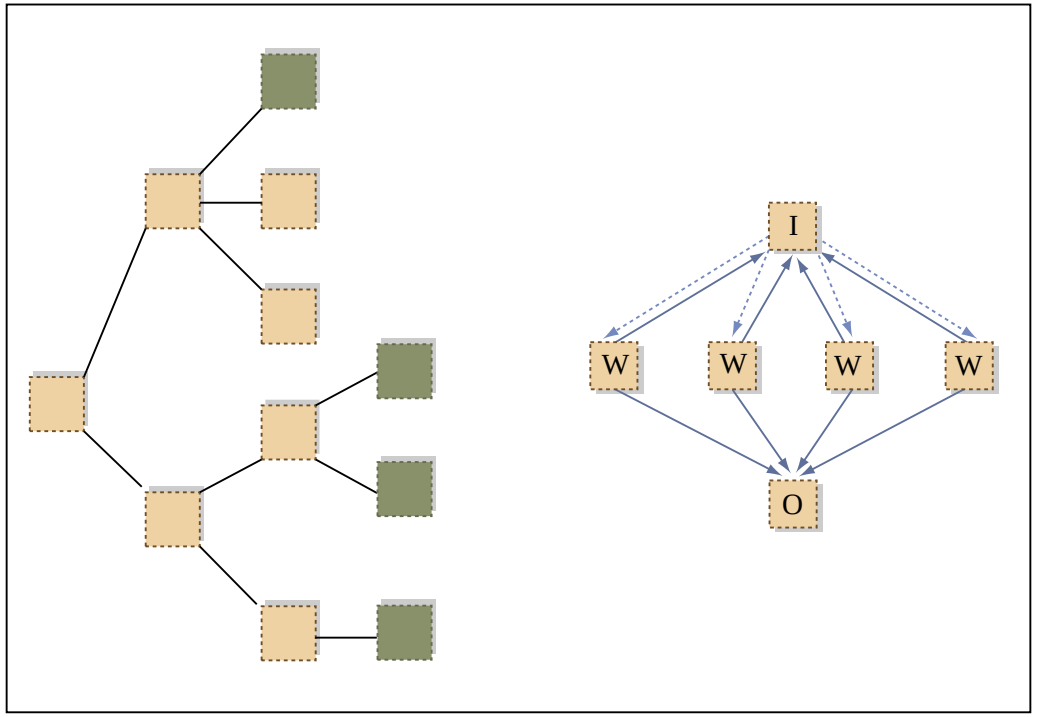


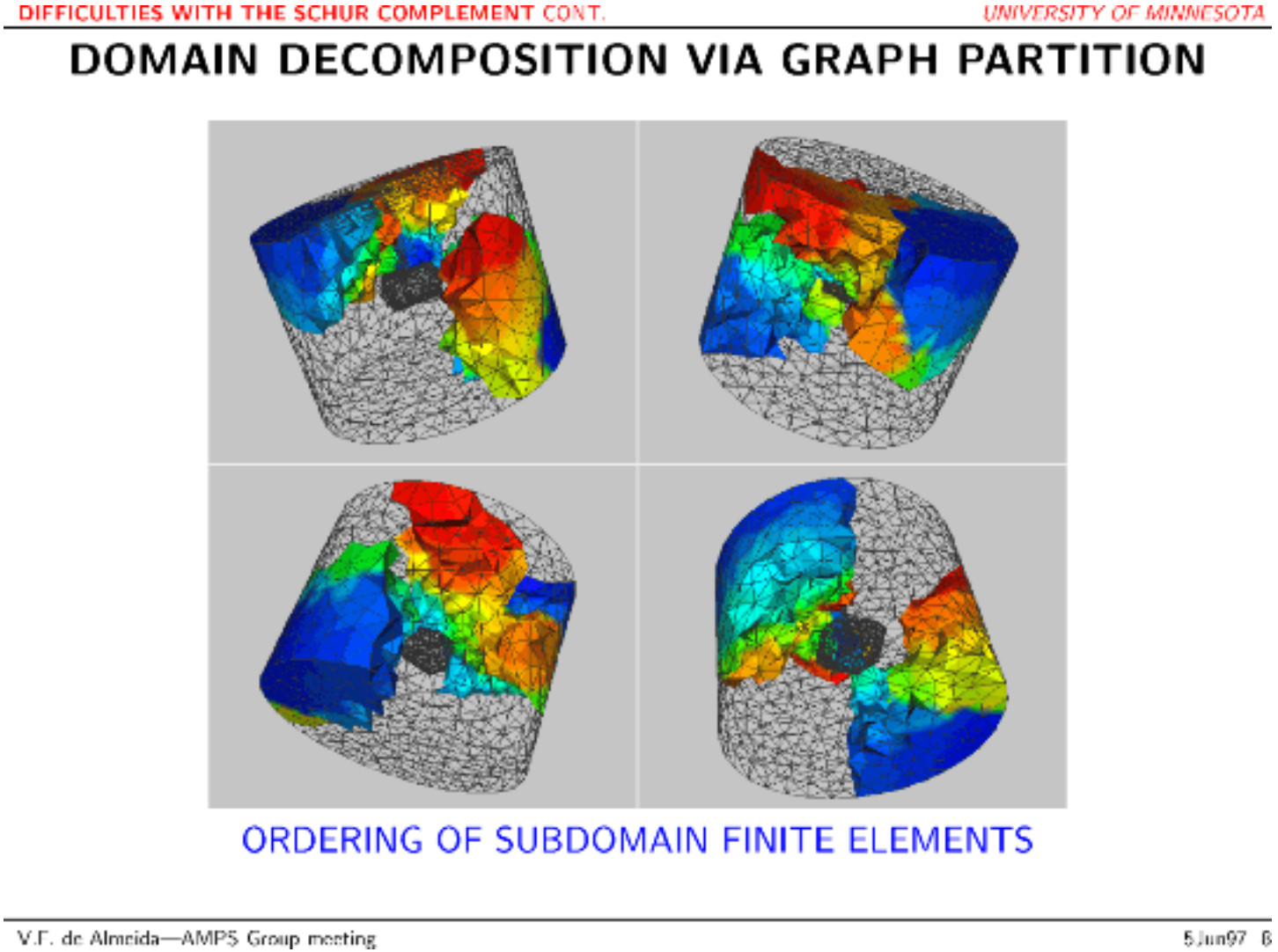
Figure by MIT OpenCourseWare.

- Search algorithms:
  - e.g. Interval solvers
  - Recursive M-W
  - Master is also Worker
- Master Slave with two Masters: Input/Output.
  - Reduces the load on the coordinating (input) Master.

Embarrassingly Parallel (EP)

- A misnomer: “Pleasantly Parallel” is better name
  - That is you wish you had an EP application! :-)
  - Very loosely coupled independent subtasks, requiring minimal *(infrequent)* point-to-point communications
  - Monte Carlo calculations, parameter searches etc.
  - Usually implemented as M-W, with the master also possibly doubling up as a worker due to low load
  - Negligible serial fraction, great scalability
  - Great for slow interconnects, unbalanced systems

Domain Decomposition

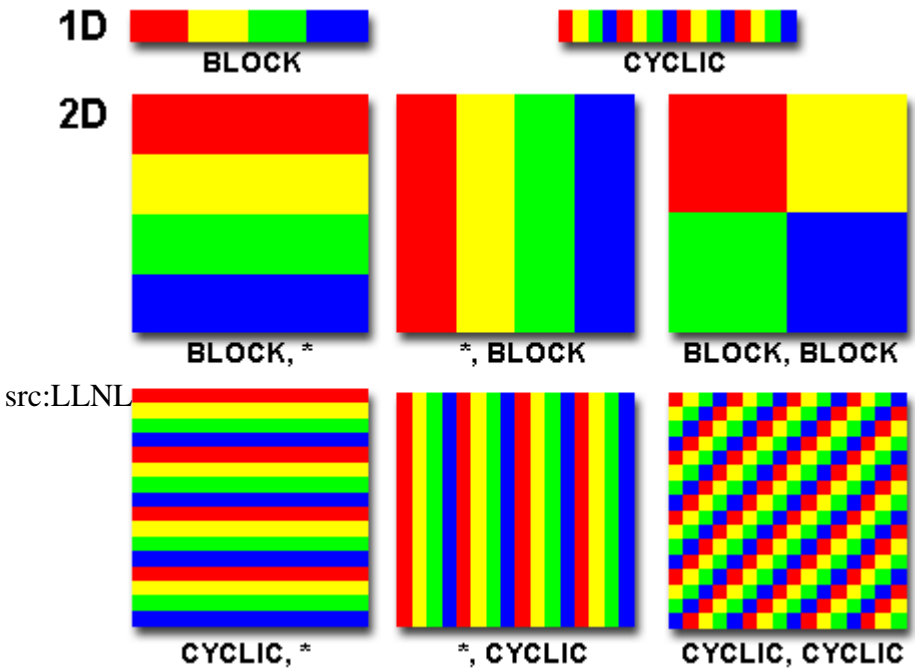


(Courtesy Dr. Valmor de Almeida, Oak Ridge National Laboratory)

- Distribute data structures to load balance, minimize comm volume and number of neighbors.
- Local problem +global update

Courtesy of Valmor de Almeida, Oak Ridge National Laboratory. Used with permission.

Array Distribution



src:LLNL

Courtesy of Lawrence Livermore National Laboratory. Used with permission.

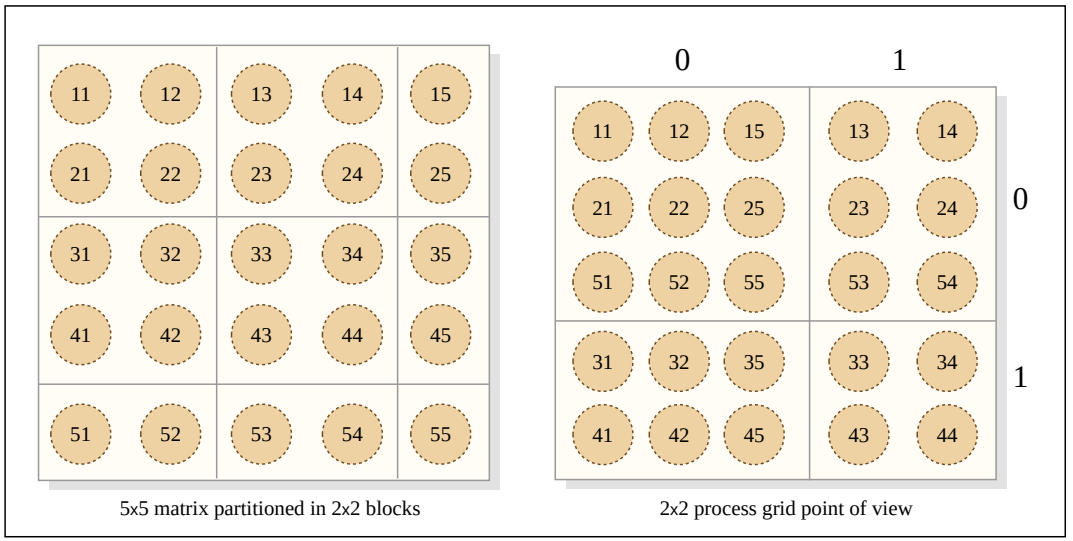


Figure by MIT OpenCourseWare.

- Given say, $\mathbf{A}\boldsymbol{x}=\boldsymbol{b}$
- The arrays in the code are distributed among the processes/threads as a block (better locality) or cyclically (better load-balance)
- Or block-cyclic
- *Owner computes...*

Particle/Space decomposition

- N-body problems in statistical mechanics, astrophysics, molecular dynamics etc.
  - N particle trajectories evolving coupled with each other with long and/or short range interactions.
    - Direct algorithm is $O(N^2)$, for load balance distribute particles to processors as evenly as possible.
    - Algorithms employing particle aggregation (hierarchical tree structures) are O(N log(N)) or O(N)
    - Decomposition is then based on space partitions; some may be empty at any given time.

Pipelined Dataflows

src:LLNL

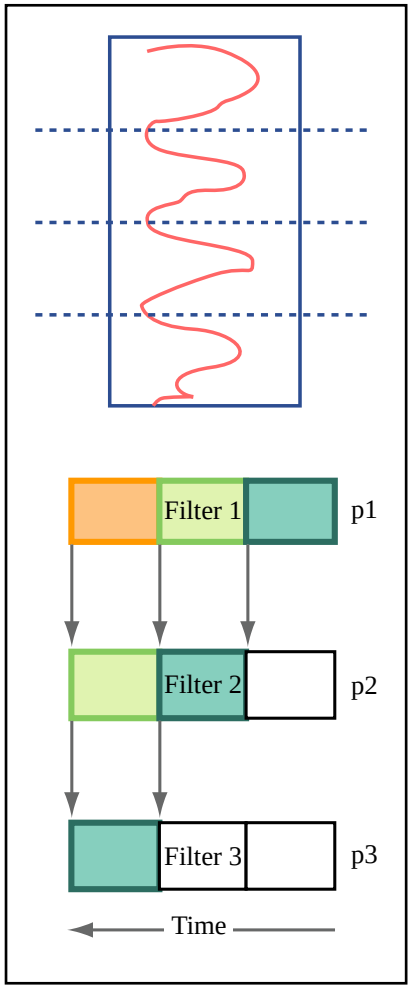


Figure by MIT OpenCourseWare.

- Like a processor pipeline, only at the level of individual CPUs or machines
- There is a corresponding pipeline start-up and wind-down cost. The longer the pipeline, the higher the cost, the more the parallelism.
  - For N>2(P-1) slices & P pipeline stages
    - *floor((N-2(P-1))/P)+2(P-1)* total stages
  - Each stage can be parallel in itself

Concurrent Workflow Execution

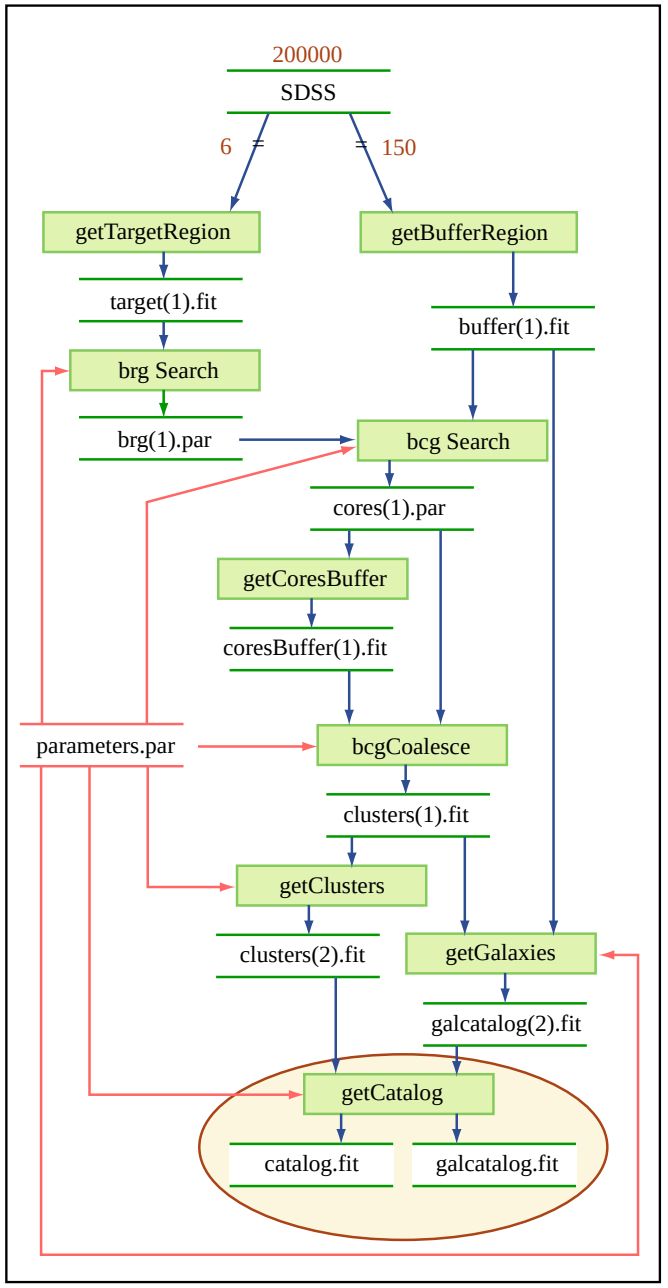


Figure by MIT OpenCourseWare.

src: Sloan Digital Sky Survey (SDSS)

- Traditional distributed computing with tighter coupling and loops
- Ideal type of Grid Computing applications. Example from:
  - the Sloan Galaxy Cluster Finder
- Parallelism depends on the “width” of the workflow, coupled with any pipelined dataflow parallelism.
- Multiple data dependencies

Interdisciplinary Models

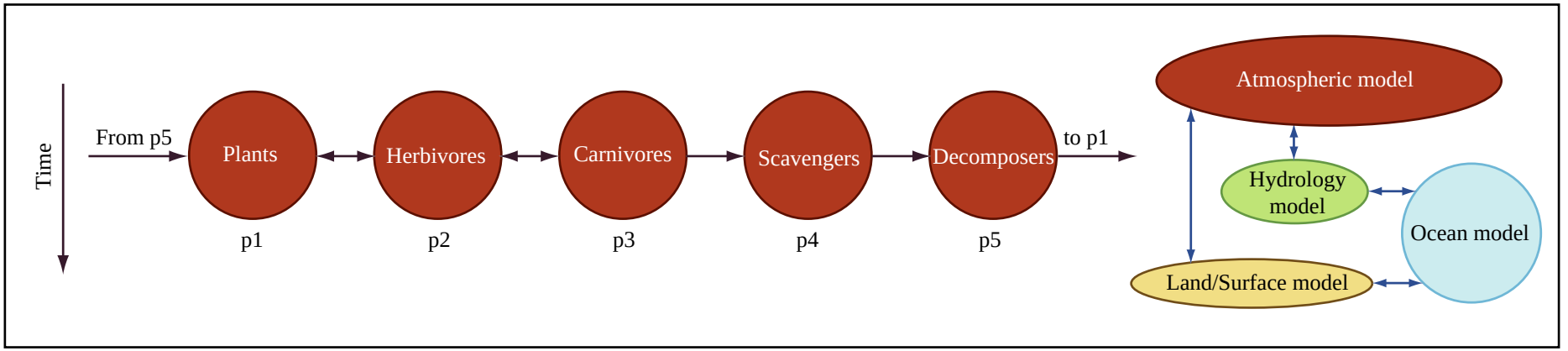


Figure by MIT OpenCourseWare.

src: LLNL

- Multiple physical/biological/chemical processes
- working in different domains, evolving in distinct timescales,
- Each process can be internally parallelized.
- *Component* approach
- Load balancing issues
- *weakly* interacting with each other (through fields, fluxes etc.) every n ($\geqslant 1$) common timesteps.
- 1 or 2-way interactions

Parallelizing an Existing Application

- Study and understand the inherent parallelism in your algorithm – you need to employ it to the maximum.
- Profile your code to find expensive parts
- Try to parallelize at as coarse a level as possible
- Among natural (for your algorithm) decompositions choose the decomposition that is the best compromise:
  - For little communication overhead
  - For load balance
  - For requiring as little change to the code as possible
- Code up, test and either tune or try another parallelization approach if too slow.
- Use 3rd party libraries!

Parallel Libraries

- Vendor, ISV or research/community codes
  - Many times free!
- They contain a lot of domain expertise
  - There is usually good documentation & support
- They offer a higher level programming abstraction
  - Cleaner, easier to understand, debug & maintain code
  - The performance issues are moved to the library
  - Changes in algorithms often possible internally, without changing the API.
- Bottom line: USE THEM if you can!

Linear Algebra

- Dense Linear Algebra:
  - ScaLAPACK/PBLAS & BLACS (on MPI)
  - PLAPACK/sB BLAS
- Sparse Linear Algebra:
  - Direct algorithms:
    - CAPS, MFACT
    - WSMP, SuperLU_DIST, PSPACES, MP_SOLVE, MUMPS
  - Iterative algorithms
    - PARPRE
    - PIM, PSPARSLIB, Aztec, Blocksolve,
  - Eigensolvers
    - PARPACK, SYISDA

Other Libraries

- Multi Solvers
  - PETSc
- Mesh & Graph partitioners:
  - CHACO, Jostle, (Par)Metis, PARTY, Scotch
- FFTs, random number generators
  - FFTW, UHFFT, SPRNG
- Dynamic Load Balancers
  - Zoltan, DRAMA
- Coupling libraries
  - MpCCI, MCT

Your own library!

- You can try and hide the functionality of your parallel solver behind a library interface
- You may have to use your own communicator
- You should assume that MPI gets initialized outside the library
- You should provide a clean API and a data structure definition to interface with
- You cannot make too many assumptions about the code that will call your library. If you do, document them well so that you and other remember them!

12.950 Parallel Programming for Multicore Machines Using OpenMP and MPI

IAP 2010