COLLEGE OF ENGINEERING
NUCLEAR ENGINEERING & RADIOLOGICAL SCIENCES
UNIVERSITY OF MICHIGAN

Lecture 13
Parallel Algorithms & Performance

Prof. Brendan Kochunas
10/21/2019

NERS 590-004

MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

Outline

- Motivation and Big Picture
- Quick Review of Parallel Architectures
- Types of parallelism and their algorithms/programming models
- Ingredients of parallel algorithms
- Common problems with debugging parallel code
- Parallel Performance Metrics
- Algorithm Performance Example:
  - Neutron transport

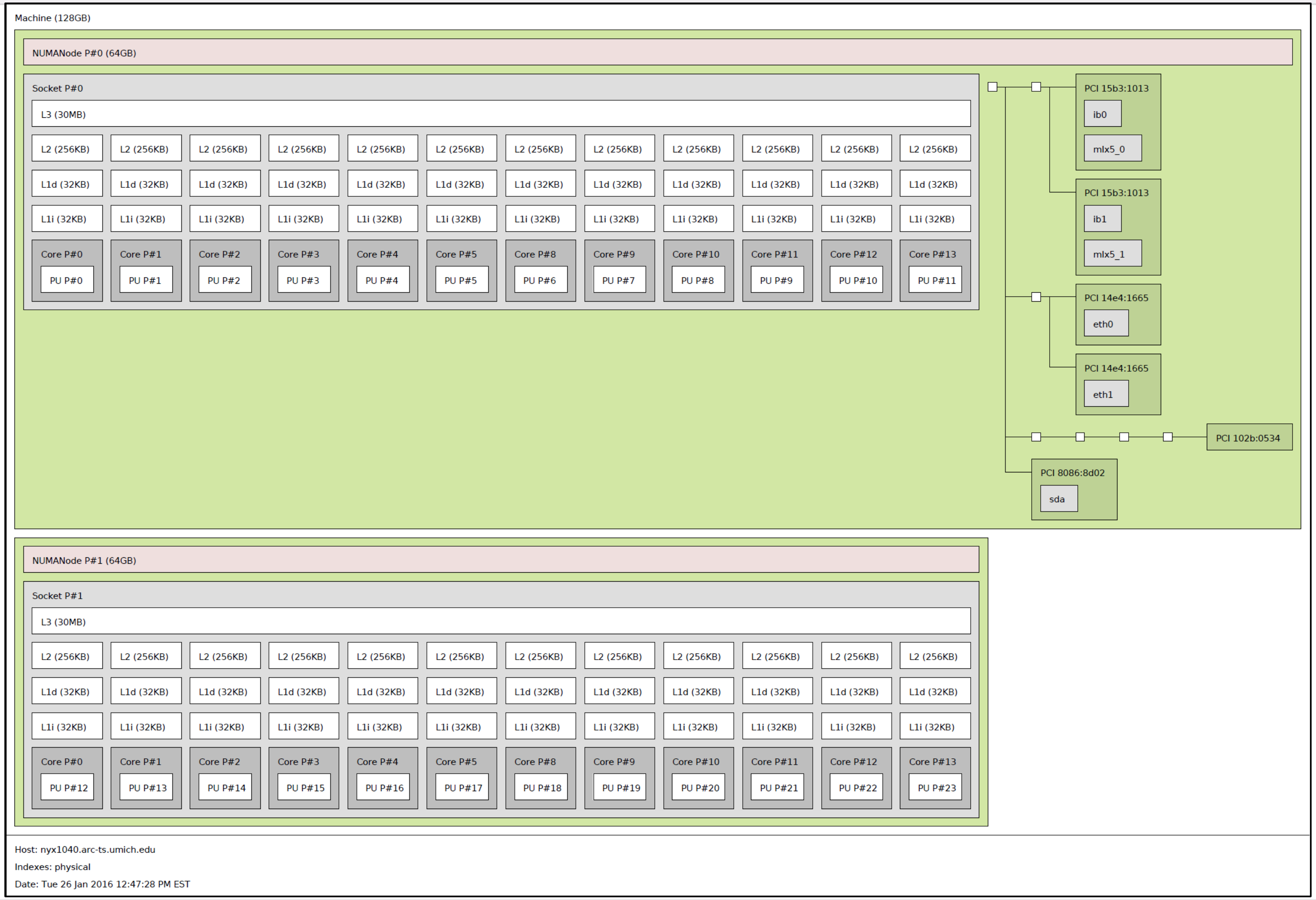
Motivation

- We may understand the mathematical formulations of our problems.
- We may understand “simple” ways to implement these on a computer.
- We’re interested in advancing our algorithms to perform simulations (and conduct science) at a faster pace.
- This requires knowledge of how to take our algorithms and devise parallel models for use on today’s supercomputers
    - (and tomorrow’s desktops)

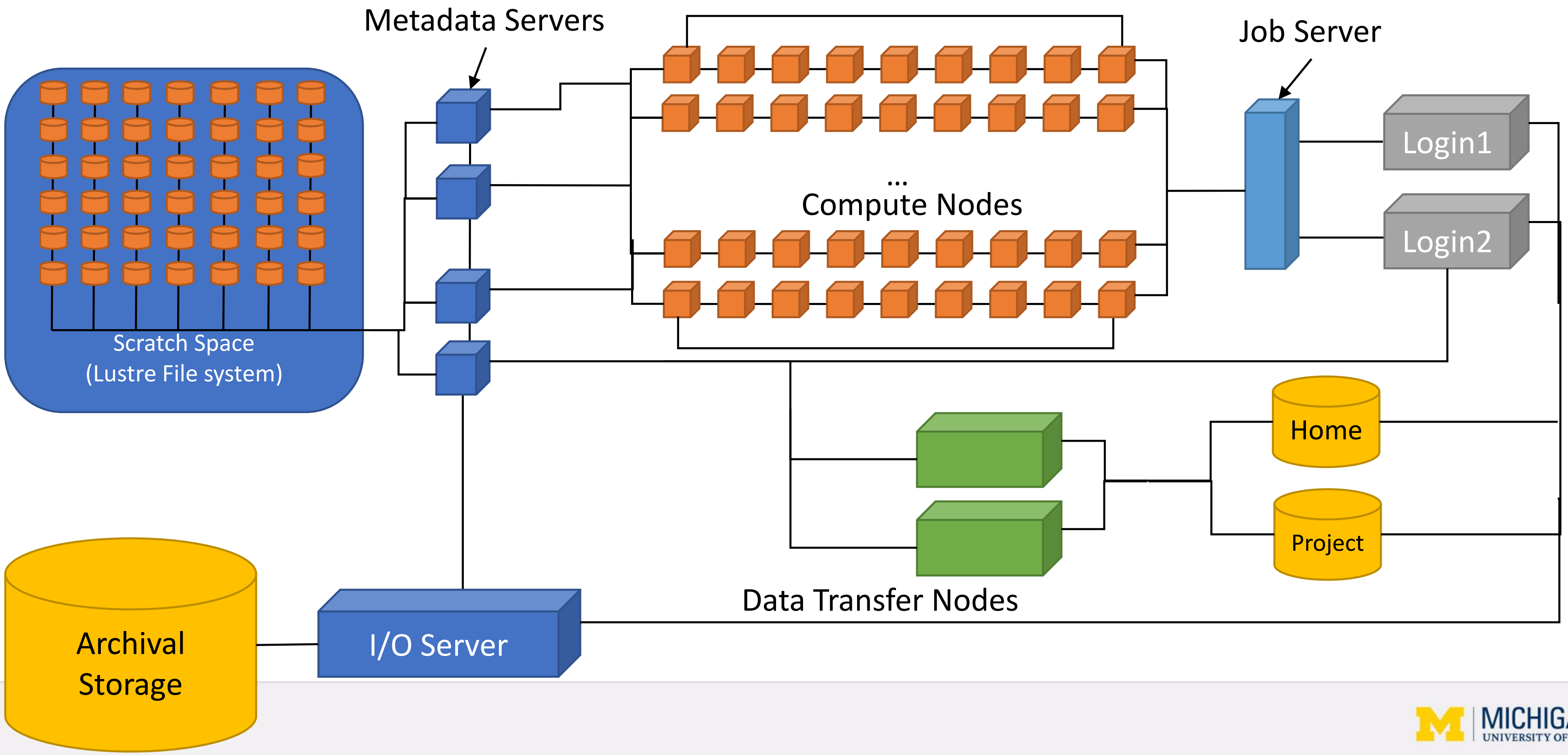
Today's Learning Objectives

- Knowledge of what techniques are used to develop parallel algorithms
  - And the types of problems this creates
- How do parallel algorithms map to hardware?
- What are the problems I should expect to encounter in parallel programming?
- What metrics are useful for evaluating the quality of a parallel algorithm?
- Performance models can be developed for complex serial and parallel algorithms,
  - and analysis of these models can give us insight into our algorithms.

Flux node Architecture



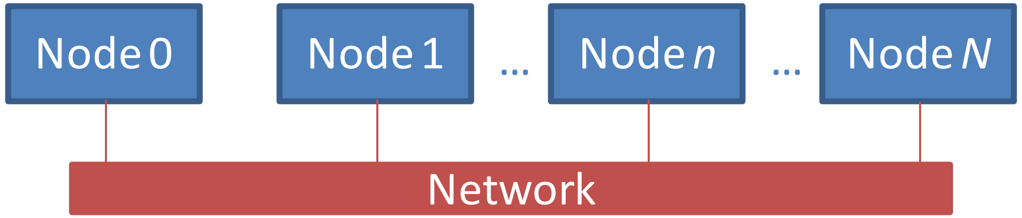
Contemporary HPC Platforms



Types of Parallel Algorithms

Distributed Memory Parallelism

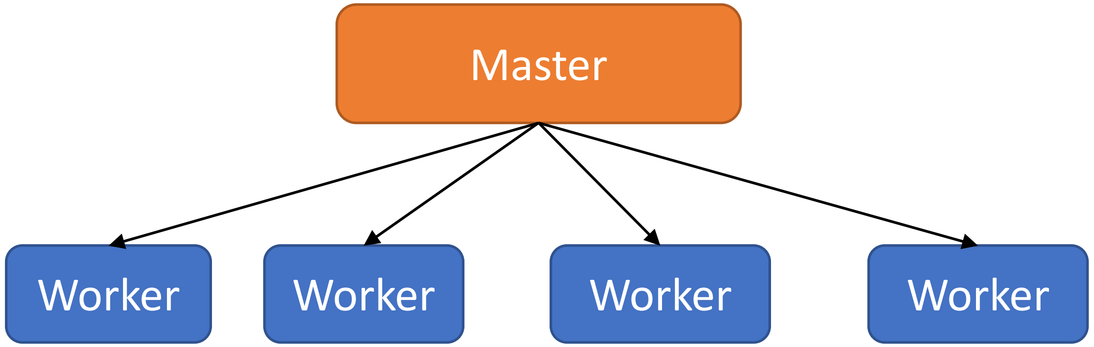
- Each process has its own memory.
  - Data between processes must be explicitly communicated.
- Usually more difficult to convert serial programs to distributed memory execution models
- Generally much easier to design software from ground up to run with distributed memory
- Common programming models
  - MPI
  - Unified Parallel C (UPC), Fortran Co-arrays



Typical Algorithms for Distributed Memory Parallelism

Master/Worker

- Master usually does more variety of work (e.g. I/O)
- Master controls execution of workers. Sends workload to workers



Bulk Synchronous

- Periodic synchronization
- Large workloads on processors between synchronization.

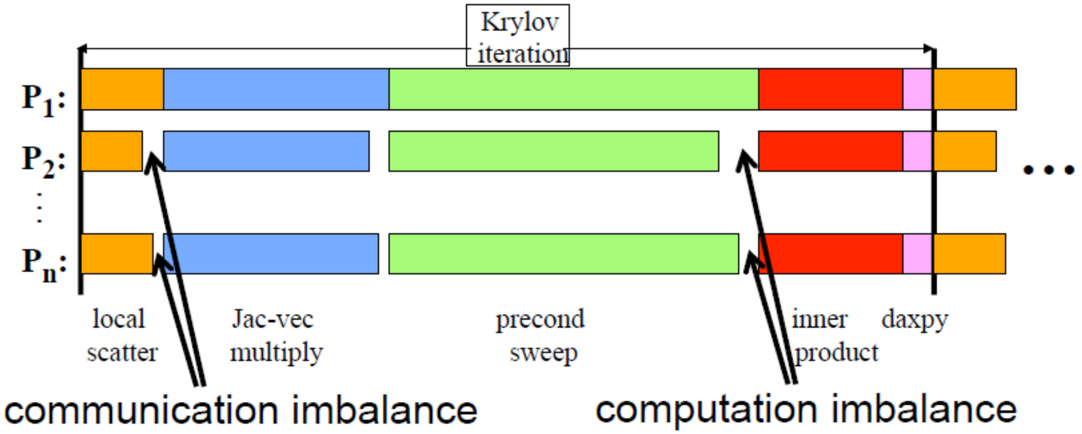
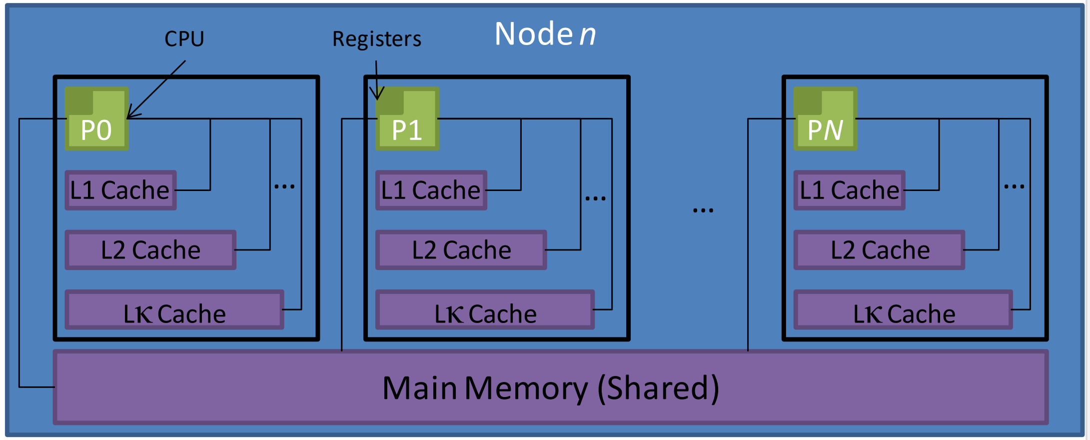


Figure from: D. Keyes, "Algorithmic Adaptations to Extreme Scale Computing", ATPESC Workshop Presentation, (2013).

Shared Memory Parallelism

- All processes "see" the same memory.
  - Changes by one process to main memory are visible to all processes
- Usually low overhead to implement with current programming models
  - Not always easy to get good performance
- Common programming models
  - pthreads (POSIX)
  - OpenMP
  - Kokkos



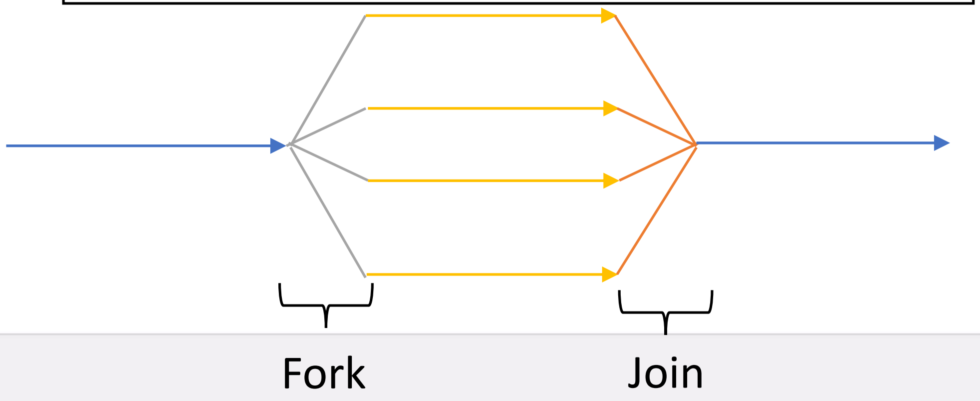
Typical Algorithms in Shared Memory Parallelism

Fork/Join

- Simple loop parallelization

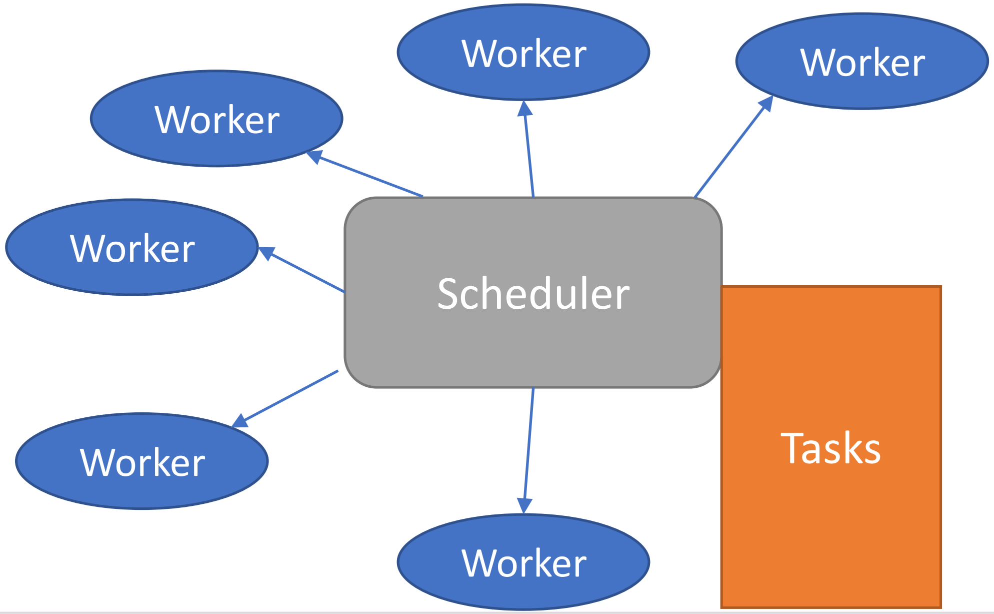
```
!Serial Section

DO PARALLEL i=1,n !implied fork
  !some operations for i
END DO PARALLEL !implied join
```



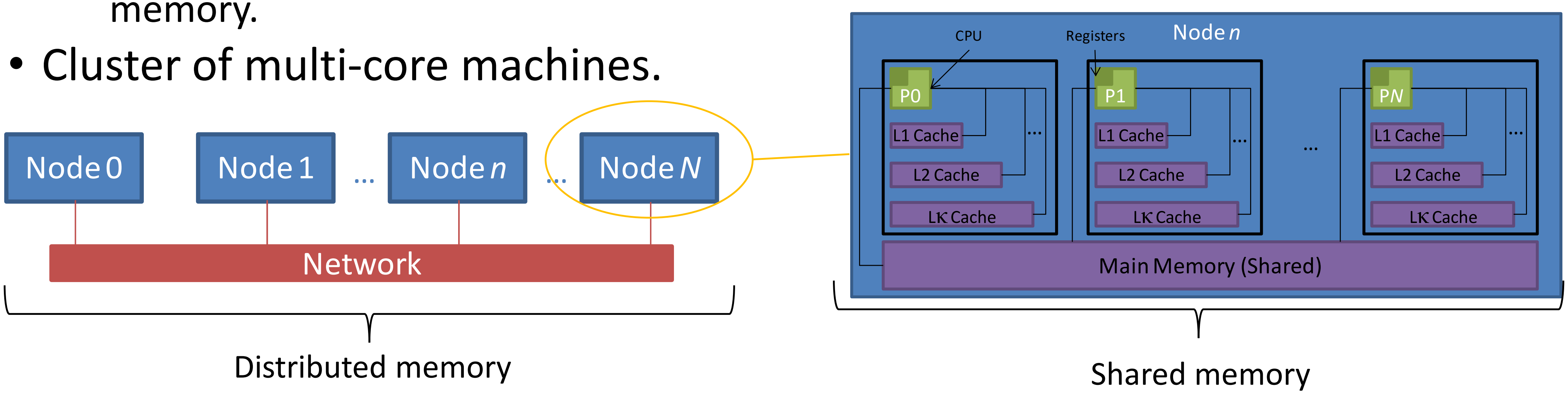
Pool of Tasks

- Tasks and work assignment are usually dynamic.



Hybrid Parallelism

- You guessed it, combines distributed and shared memory.
- This is representative of most modern compute clusters.
    - But remember these machines are configured to be able to run flexibly as either purely distributed, hybrid, or (if the programming model exists) purely shared memory.
- Cluster of multi-core machines.



A few closing points

- Distributed memory algorithms and shared memory algorithms are not necessarily mutually exclusive
  - e.g. your code may make use of some combination of these
- There are other types of algorithms, but these are the "most common"
- Generally, parallel algorithms typically require some definition of how the memory is treated between the parallel processes
  - This can be abstracted away from the hardware.

Parallel Algorithm "Ingredients"

Parallel Algorithm Ingredients

- What is the *programming model*? (distributed, shared, both)
  - If distributed, what is the communication model?
- What should the *granularity* of the parallelism be?
- How are you going to *decompose* the problem in parallel?
- How are you going *partition* the problem to obtain a balanced decomposition?
- Can all this be done once for a single simulation?
- What synchronizations are required?

Coarse Grained vs. Fine Grained

Coarse Grained

- Divide work into large tasks
  - Example: executing several functions
- Coarse grained parallelism usually has better strong scaling than fine-grained parallelism.
  - Although smaller limits to the maximum parallelism
- More susceptible to load imbalance.

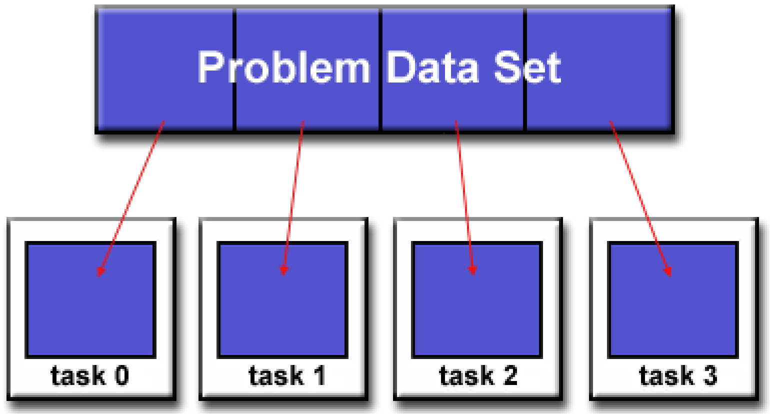
Fine Grained

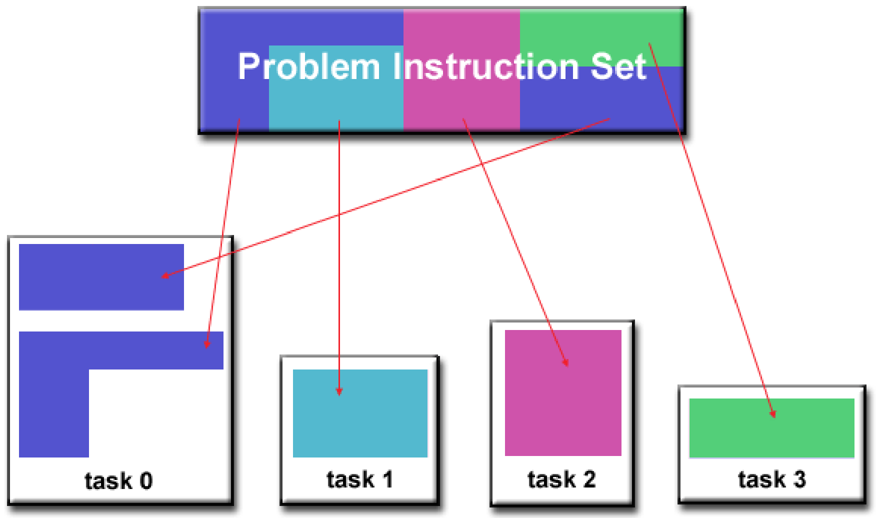
- Divide work into many small tasks
  - Example: iterations of a loop
- Usually has good load balance
- Difficult to hide overhead from parallelism
- Works well for things like SIMD & vector computing

Algorithm & Hardware will ultimately determine which is better. However, coarse-grained will usually be better

Decomposition

- What is being divided into parallel work?
- Most typical is domain decomposition
  - Divide up part of your equation "phase space"
    - Phase space = dependent variables of unknown (e.g. Cartesian space)
  - Slightly different is data decomposition
    - e.g. decompose a matrix in parallel
      - Matrix is usually a discretization of the phase space(s)
- Also have functional decomposition
  - Decompose by computation or operation
    - e.g. fluid on one process, solid on another for convective/conductive heat transfer

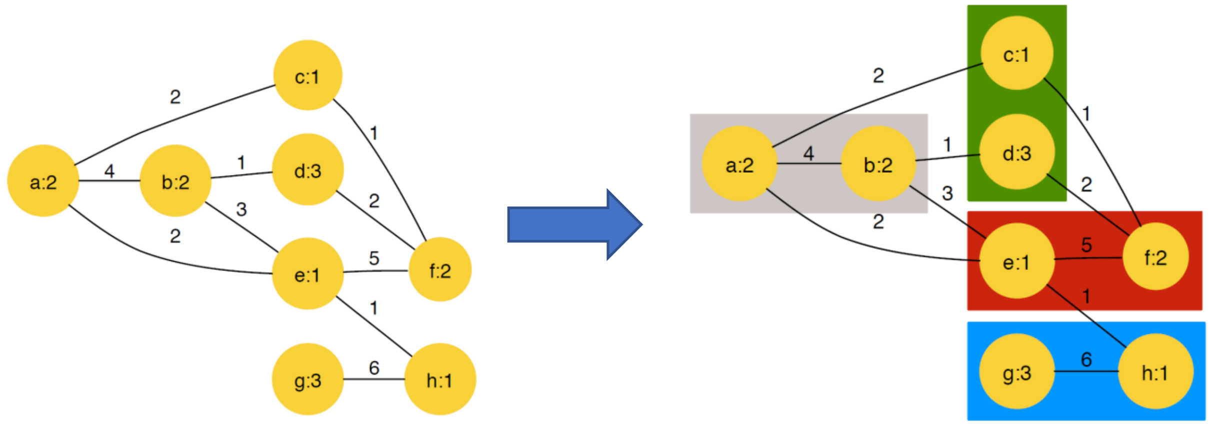




Figures from: https://computing.llnl.gov/tutorials/parallel_comp/

Partitioning

- How do you decompose the problem in parallel?
  - Example: Matrix partitioning
- In general this is a much harder problem.
  - Especially for the general case.
    - Involves a lot of graph theory



Figures from: R. Vuduc, “Graph Partitioning,” Lecture in CSE/CS 8803, Georgia Institute of Technology, April 2008

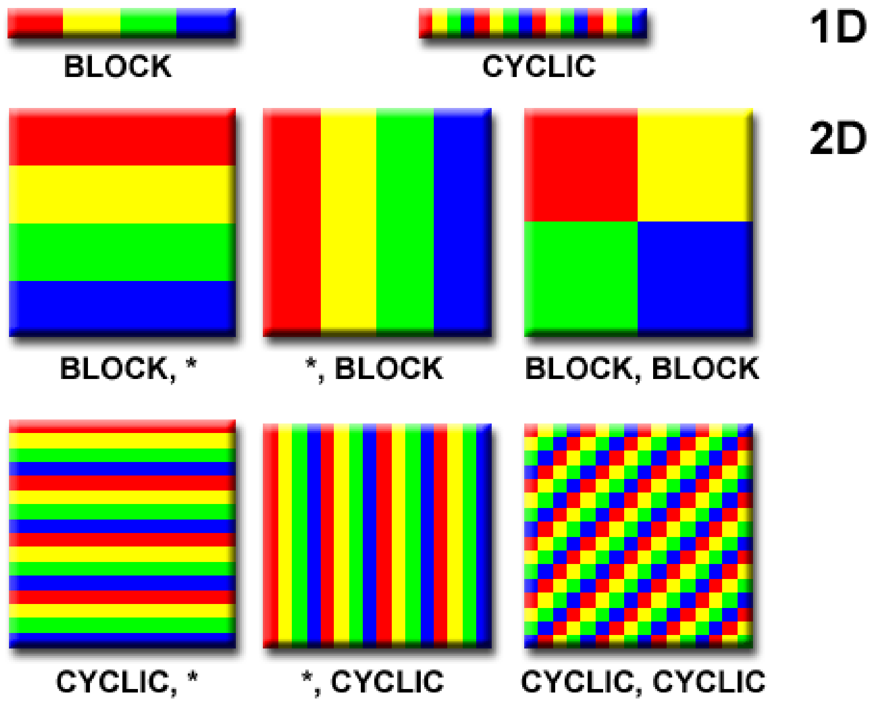


Figure from: https://computing.llnl.gov/tutorials/parallel_comp/

- Libraries exist to do this for us: METIS & ParMETIS

Dynamic vs. Static

Static

- Determine decomposition and partitioning once up-front prior to execution.
- Execute without changing number of processors or decomposition or partitioning
  - Fork/Join is not considered dynamic if the number of threads always the same
- More likely you will encounter this case

Dynamic

- Necessary to achieve better performance if computation load changes during run time.
- Change number of processors during run time.
- Change partitioning during run time.

Synchronization

- Generally, best to avoid as much as possible
  - In practice, never completely avoidable.
- In shared memory parallelism this includes the fork and join operations.
- Synchronization usually occurs whenever you encounter an integral.
  - More generally it occurs with "reduction" operations.
  - In a reduction operation you reduce parallel data to a single process
    - E.g. computing a sum, finding a max, computing a product, logical operators
- In distributed memory parallelism (more specifically MPI), it is any collective operation (not just reduce)
- Critically important to be aware of collective operations

Illustrations of collective operations

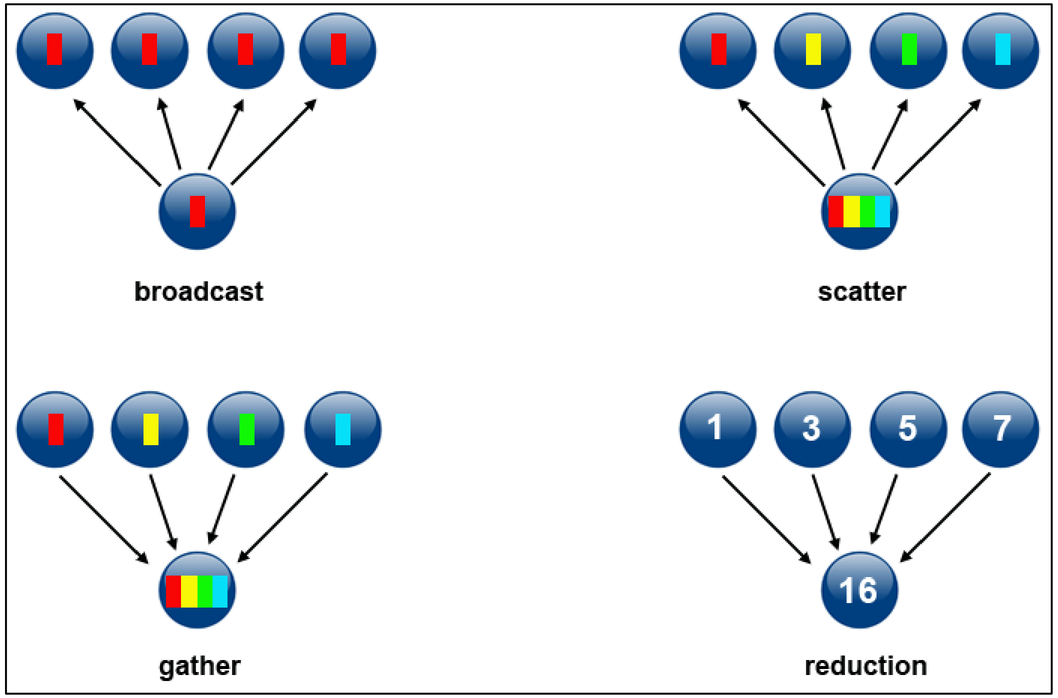


Figure from: https://computing.llnl.gov/tutorials/parallel_comp/

Parallel Programming Pitfalls

Parallel Programming Sounds Easy… but

- But it is much harder than programming in serial
- There is a whole new world of bugs that you can encounter
  - Deadlocks and Race conditions

- Efficiency is more difficult to achieve
- Generally have to be more aware of what's going on…

Deadlock

Problem

- Symptoms
  - Code will run for a while
  - Then code will "hang".
  - Code just sits... and sits... and sits.

```
IF(MOD(myRank,2) == 0) THEN
  CALL MPI_Send(sbuffer, n, MPI_DOUBLE_PRECISION, &
    myRank+1, 0, MPI_COMM_WORLD, mpierr)
  CALL MPI_Recv(rbuffer, n, MPI_DOUBLE_PRECISION, &
    myRank+1, 0, MPI_COMM_WORLD, MPI_STATUS_NULL, mpierr)
ELSE
  CALL MPI_Send(sbuffer, n, MPI_DOUBLE_PRECISION, &
    myRank-1, 0, MPI_COMM_WORLD, mpierr)
  CALL MPI_Recv(rbuffer, n, MPI_DOUBLE_PRECISION, &
    myRank-1, 0, MPI_COMM_WORLD, MPI_STATUS_NULL, mpierr)
ENDIF
IF(MOD(myRank,2) == 0) &
  CALL MPI_Reduce(sbuf,rbuf,n,MPI_DOUBLE_PRECISION, MPI_SUM, &
    0, MPI_COMM_WORLD, mpierr)
```

Solution

- Investigate where your calls to communication are made.
  - Usually will happen around branching constructs.
- Think about how it would execute with 2 processors.

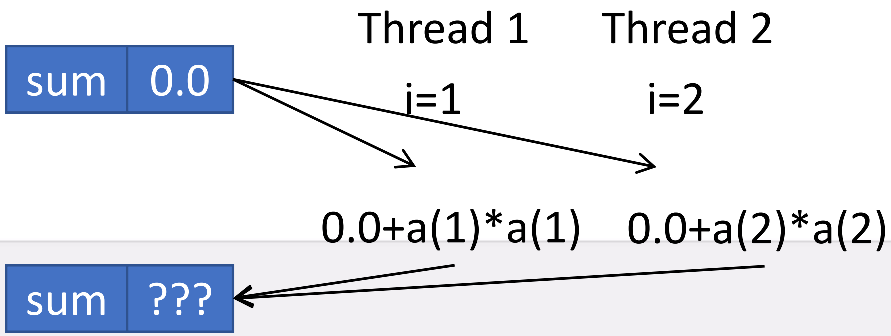
```
IF(MOD(myRank,2) == 0) THEN
  CALL MPI_Send(sbuffer, n, MPI_DOUBLE_PRECISION, &
    myRank+1, 0, MPI_COMM_WORLD, mpierr)
  CALL MPI_Recv(rbuffer, n, MPI_DOUBLE_PRECISION, &
    myRank+1, 0, MPI_COMM_WORLD, MPI_STATUS_NULL, mpierr)
ELSE
  CALL MPI_Recv(rbuffer, n, MPI_DOUBLE_PRECISION, &
    myRank-1, 0, MPI_COMM_WORLD, MPI_STATUS_NULL, mpierr)
  CALL MPI_Send(sbuffer, n, MPI_DOUBLE_PRECISION, &
    myRank-1, 0, MPI_COMM_WORLD, mpierr)
ENDIF
```

Race Conditions

Problem

- Symptoms
  - Indeterminate behavior.
  - Seemingly random values are produced
  - Shared memory parallelism

```
sum=0.0
PARALLEL DO i=1,n
  sum=sum+a(i)*a(i)
END PARALLEL DO
```



Solution

- Create separate storage for each thread
  - Reduce values among threads at end of parallel execution

```
s=0.0
PARALLEL DO i=1,n
  s(t)=s(t)+a(i)*a(i)
END PARALLEL DO
sum=SUM(s(1:nthreads))
```

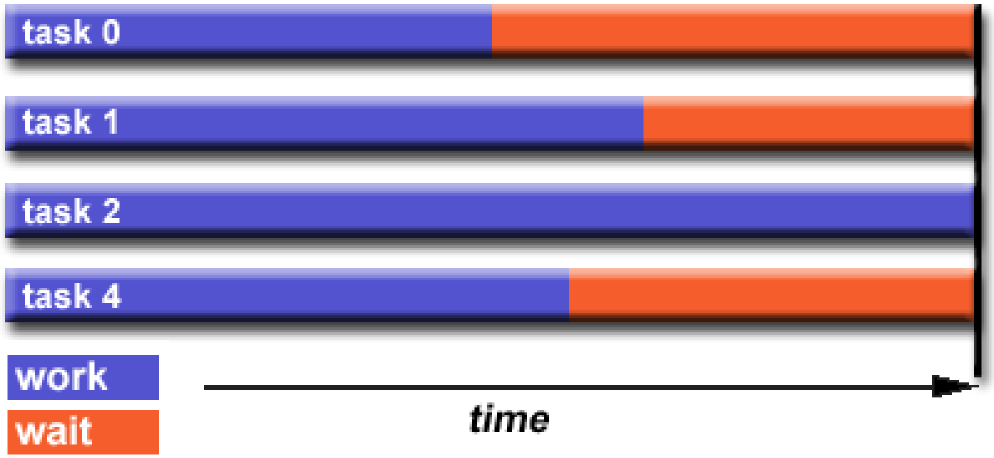
- Introduce a serialization/lock/critical section on variable

```
sum=0.0
PARALLEL DO i=1,n
  !Critical
  sum=sum+a(i)*a(i) !One thread at a time
END PARALLEL DO
```

Load Balance & Idle Time

Problem

- Poor strong scaling
- Poor parallel efficiency
- Increase cores by factor of 2x, do not observe 2x speedup.

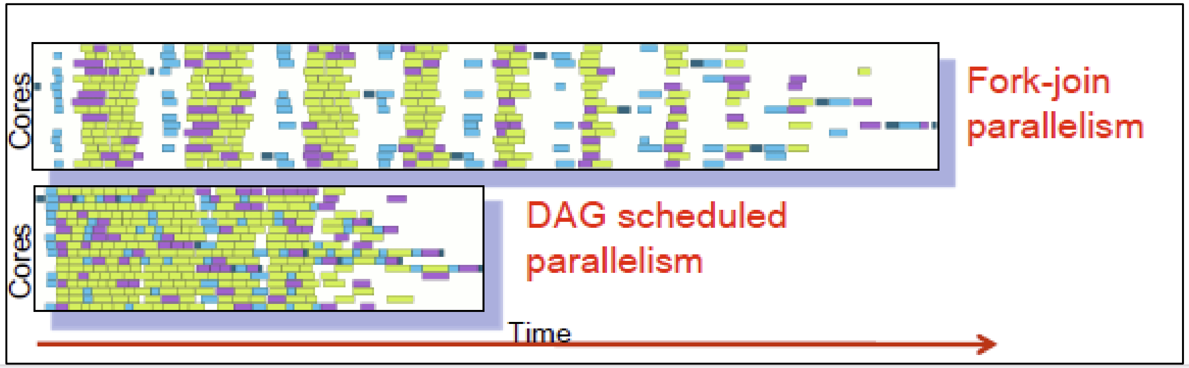


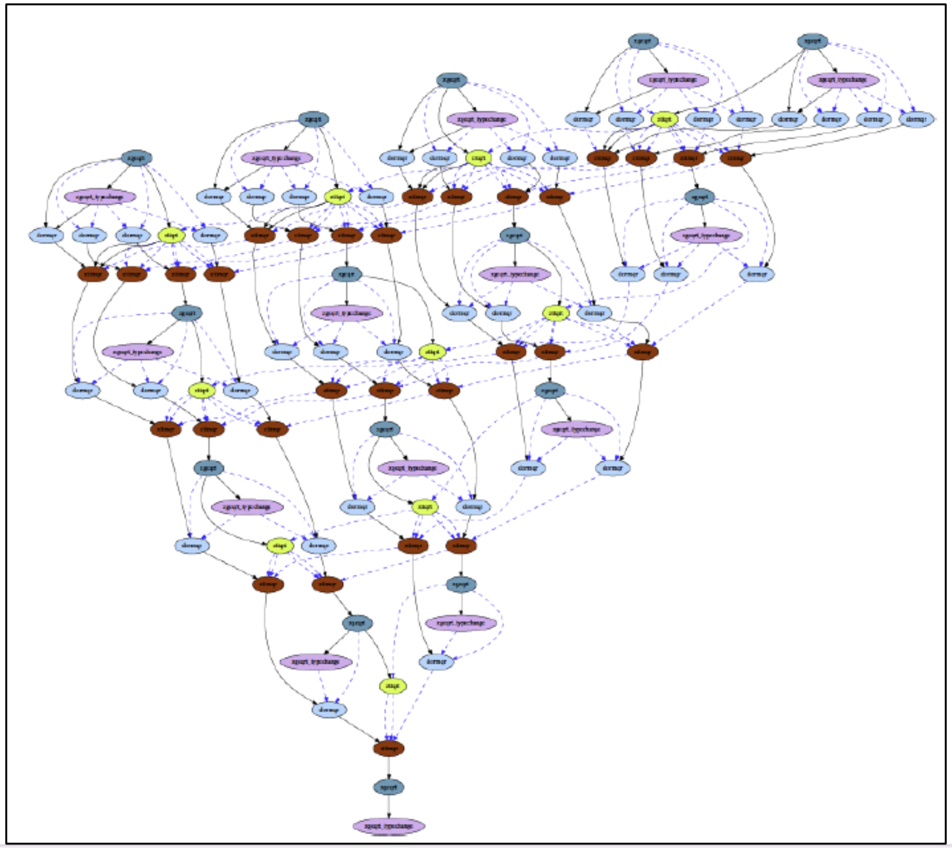
Solution

- Determine what the load balance/imbalance is
  - Need to assign a value of "work" to each subdomain.
  - What is the maximum to minimum workload for all domains.
- Change partitioning to improve load balancing
- Change parallel algorithm

State of the Art Techniques in Load Balancing and Scheduling

- Utilize Directed Acyclic Graphs (DAGs)
- Decompose your algorithm into a DAG
- Use algorithms for dynamic or optimal scheduling on the DAG
- Improves performance on multicore/hybrid architectures





Figures from: J. Dongarra, “Adaptive Linear Solvers and Eigensolvers”,ATPESC Workshop Presentation, (2013).

Summary of Programming Pitfalls

- If you think debugging serial programs is difficult, debugging parallel programs is often exponentially harder.
- Before you get a bruise from banging your head against the wall
  - Take a step back and ask yourself what’s the behavior you are observing
  - Then think like a (medical) doctor, and try to diagnose the problem based on the symptoms.
- Race conditions and deadlocks are really easy to implement accidentally.
- Resolving load imbalance often requires a lot of effort.

Parallel Performance

Parallel Performance Metrics

- Strong Scaling: fixes problem size and increases number of processors.
  - Provides insight into how finely grained an algorithm can be parallelized and how much parallel overhead there is relative to useful computation
- Weak Scaling: fixes problem size *per process* and increases number of processors.
  - Provides insight into whether the parallel overhead varies faster or slower than the amount of work as the problem size is increased.
- Speedup and Efficiency: $S\left(P_{size}, N_p\right) \equiv \frac{T\left(P_{size}, 1\right)}{T\left(P_{size}, N_p\right)}$
- Good efficiency does not necessarily mean you have fast code
  - It could mean you have terrible serial performance

$$E_{strong}\left(P_{size}, N_p\right) \equiv \frac{T\left(P_{size}, 1\right)}{N_p \times T\left(P_{size}, N_p\right)}$$

$$E_{weak}\left(P_{size}, N_p\right) \equiv \frac{T\left(P_{size}, 1\right)}{T\left(P_{size} \times N_p, N_p\right)}$$

Parallel Execution Time Models

Moving from serial to parallel

- Serial Latency based model

$$T_{serial} = Ft_F + \alpha_1 L + \sum_{j=1}^{\kappa-1} \left(\alpha_{j+1} - \alpha_j\right) M_j + \left(\alpha_{mem} - \alpha_\kappa\right) M_\kappa$$

- Parallel Model

$$T_{parallel}\left(N_p\right) = \frac{T_{serial}}{N_p} + T_{overhead}\left(N_p\right)$$

- Difficult to develop exact expressions,
  - Alternatively measure realistic average values based on microbenchmarks.

Canonical Execution Time Models

- Distributed Memory Computing
  - Point-to-Point Communication Time

$$T_{comm} = \underbrace{\alpha_{network}}_{\text{Latency}} + \underbrace{\beta_{network}}_{\text{Bandwidth}} \underbrace{N}_{\text{Amount of data}}$$

  - Collective operations have their own (depends on algorithm implemented in library)

$$T_{\text{All\_reduce,small}} = \lceil \log p \rceil \left(\alpha_{network} + \beta_{network} \times N + \gamma \times N\right)$$

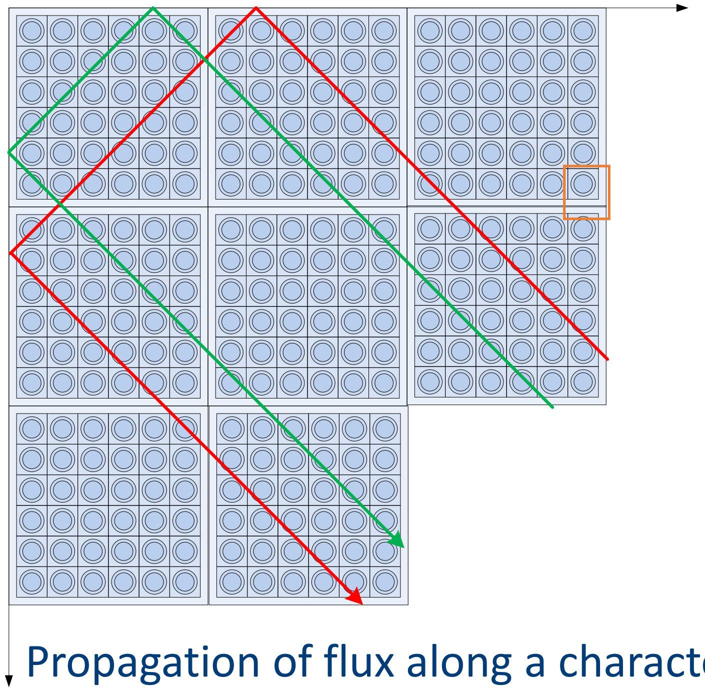
$$T_{\text{All\_reduce,large}} = 2 \log p \alpha_{network} + \frac{p-1}{p}\left(2\beta_{network} \times N + \underbrace{\gamma}_{\text{Time to perform reduce operation (e.g. sum, max, multiply, etc.)}} \times N\right)$$

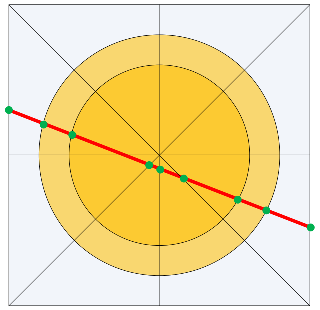
Fundamentals of getting good parallel performance

- Maximize amount of work that can be parallelized.
- Minimize overhead.
- Usually this means
  - Balance work loads among processors
  - Avoid synchronization
    - Especially for shared memory
  - use non-blocking communication
    - Primarily in distributed memory models
- Make sure the serial code is optimized.

$$T_{parallel}\left(N_p\right)=\underbrace{T_{non-parallel}}_{\text{Minimize}}+\overbrace{\frac{T_{serial}}{N_p}}^{\text{Assumes perfect load balance}}+\underbrace{T_{overhead}\left(N_p\right)}_{\text{Minimize}}$$

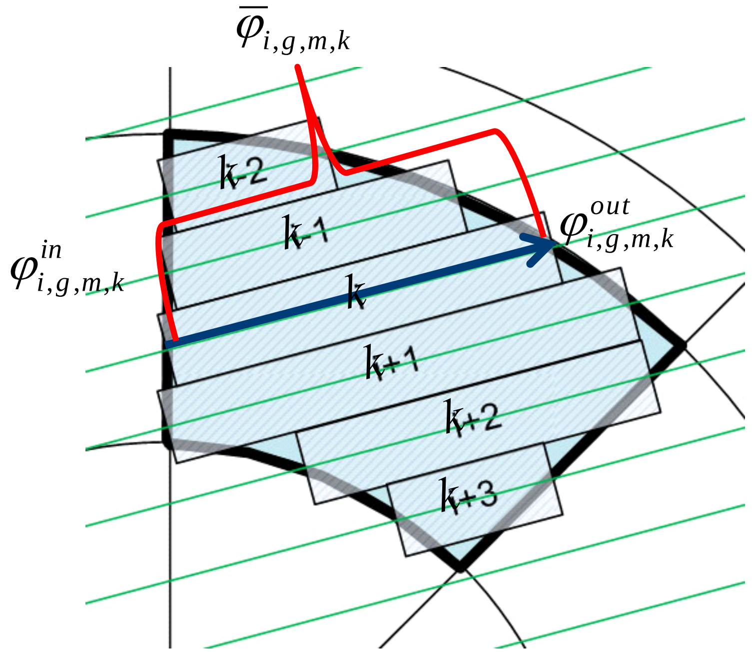
Example of More Complex Performance Models: Neutron Transport





Segment average angular flux

$$\bar{\varphi}_{i,g,m,k} = \frac{\varphi^{in}_{i,g,m,k} - \varphi^{out}_{i,g,m,k}}{\Sigma_{t,i,g} s_{i,k,m}} + \frac{q_{i,g,m}}{\Sigma_{t,i,g}}$$



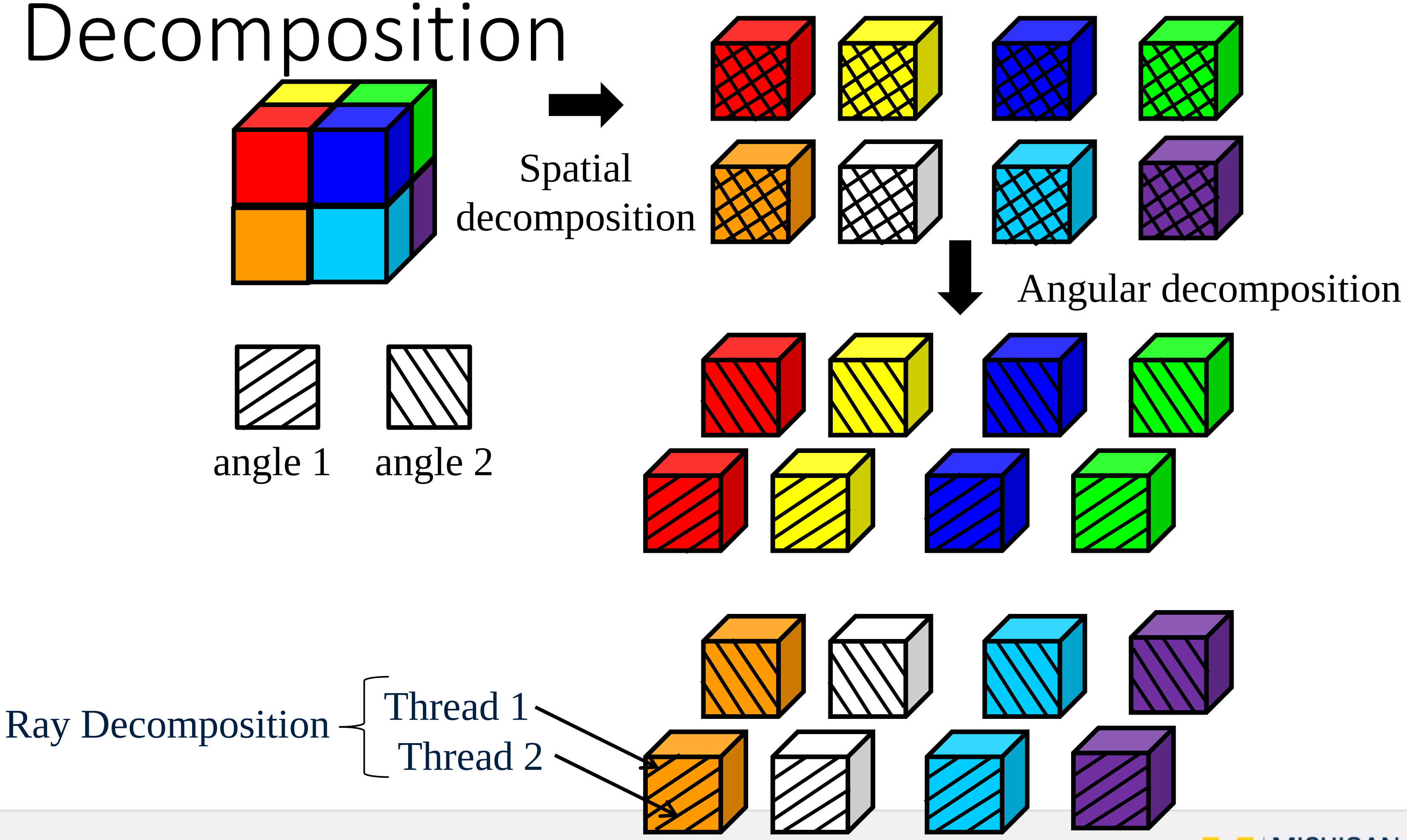
Region average angular flux

$$\bar{\varphi}_{i,g,m} = \frac{\sum_k \bar{\varphi}_{i,g,m,k} s_{i,k,m} \delta A_{k,m}}{\sum_k s_{i,k,m} \delta A_{k,m}}$$

Propagation of flux along a characteristic segment

$$\varphi^{out}_{i,g,m,k} = \varphi^{in}_{i,g,m,k} \exp\left(-\Sigma_{t,i,g} s_{i,k,m}\right) + \frac{q_{i,g,m}}{\Sigma_{t,i,g}}\left[1 - \exp\left(-\Sigma_{t,i,g} s_{i,k,m}\right)\right]$$

Parallel Decomposition



Summary of Parallel Kernel

1. Loop over all angles in angle subdomain
   a. Loop over all long rays in angle in parallel with threads
      i. Order the modular ray data for a complete long ray — *build*
      ii. Evaluate the exponential function for all segments in the long ray — *exp*
      iii. Load incoming boundary conditions for each end of ray
      iv. Loop over all segments in the long ray — *MOC*
         1. Evaluate Eq. 3.5 in forward direction
         2. Accumulate $\varphi^{d}_{i,g,m,k}$ into temporary for Eq. 3.12 for forward direction.
         3. Evaluate Eq. 3.5 in backward direction
         4. Accumulate $\varphi^{d}_{i,g,m,k}$ into temporary for Eq. 3.12 for backward direction.
      v. Store outgoing boundary conditions for each end of ray
   b. Accumulate $w_m \delta A_m \hat{\varphi}_{i,g,m}$ into temporary for Eq. 3.13 — *scal*
   c. Wait for all threads to finish loop over long rays
   d. Send outgoing boundary conditions to neighbor subdomains in space — *BCUp*
2. Add partial sums of Eq. 3.13 from all threads
3. Add partial sums of Eq. 3.13 from all angular domains
4. Evaluate Eq. 3.14 for all regions
5. Wait for all incoming boundary conditions from neighbor subdomains in space — *flux*

Serial Performance Model

$$T_{sweep} = T_{build} + T_{\exp} + T_{MOC} + T_{BC} + T_{scal} + T_{BCUp} + T_{flux}$$
$$F_{sweep} = F_{build} + F_{\exp} + F_{MOC} + F_{BC} + F_{scal} + F_{BCUp} + F_{flux}$$

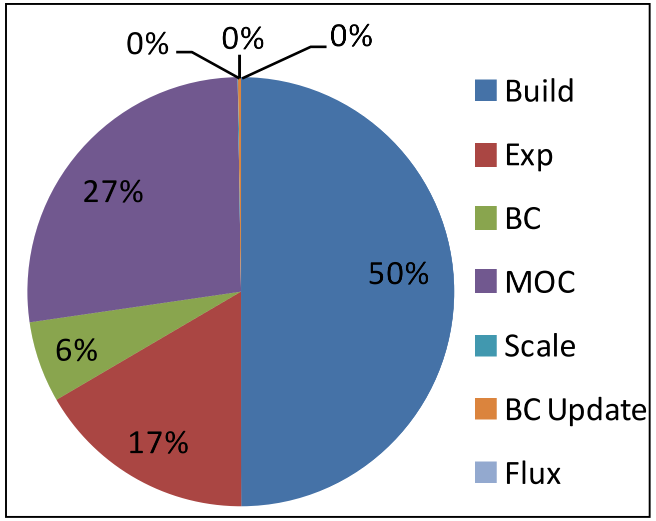
- The time to perform a sweep is represented by the different components of the algorithm.
- Number of FLOPs and Loads as function of the problem size

Component	FLOPs	Loads	Computational Intensity (FLOPs/Loads)
build	nseg	$c_{build} \times$ nseg	$1/c_{build}$
exp	$3\times$ nseg	$6\times$ nseg	0.5
MOC	$8\times$ nseg	$12\times$ nseg	0.75
BC	0	$8\times$ nlongray	0
BCUp	0	$4\times$ nlongray	0
scal	$8\times$ nreg $\times$ nangoct	$8\times$ nreg $\times$ nangoct $+ 4\times$ nangoct	~1.0
flux	$4\times$ nreg	$4\times$ nreg	1.0
Sweep (Total)	$12\times$ nseg $+ 8\times$ nangoct $\times$ nreg $+ 4\times$ nreg	$(18 + c_{build})\times$ nseg $+ 12\times$ nlongray $+ 8\times$ nangoct $\times$ nreg $+ 4\times$ nreg $+ 4\times$ nangoct	$0.0 < C.I. <\sim 0.5$

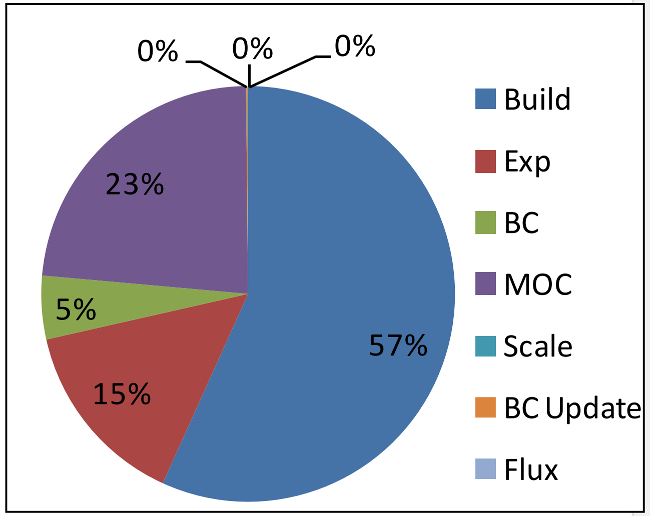
Validation of Serial Performance Model

Relative Difference Between Performance Model and Experimentally Measured Performance

Case	FLOPs	Loads	Exec. Time
Default	0.0002%	-5.54%	-7.41%
Fine Angles	0.0002%	-6.89%	-7.38%
Fine Rays	0.00007%	-12.19%	4.45%
Fine Space	0.0001%	0.70%	-1.89%



Performance Model Prediction



Measured

Parallel Performance Model

- Serial $T_{sweep} = T_{build} + T_{\exp} + T_{MOC} + T_{BC} + T_{scal} + T_{BCUp} + T_{flux}$
- Parallel

$$T_{sweep} = \frac{T_{build} + T_{\exp} + T_{MOC} + T_{BC}}{p_{space} p_{ang} p_{ray}} + \frac{T_{scal}}{p_{space} p_{ang}} + T_{OMP}(p_{ray}) + \max\left(\frac{T_{flux} + T_{ray}}{p_{space}} + T_{ang}, T_{space} \right)$$

- Spatial Decomposition

$$T_{space} = \mathrm{nface} \times (\alpha_{network} + \beta_{network} \times \mathrm{nlongray}(iang, iface)) + \frac{T_{BCUp}}{(p_{space})^{2/3}}$$

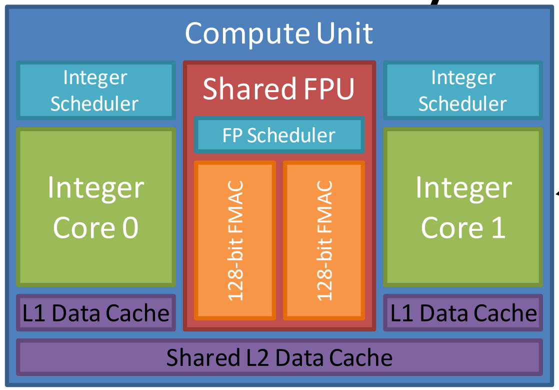
- Angular Decomposition

$$T_{ang} = c_1(p_{ang})\alpha_{network} + c_2(p_{ang})(\beta_{network} + \gamma')N$$

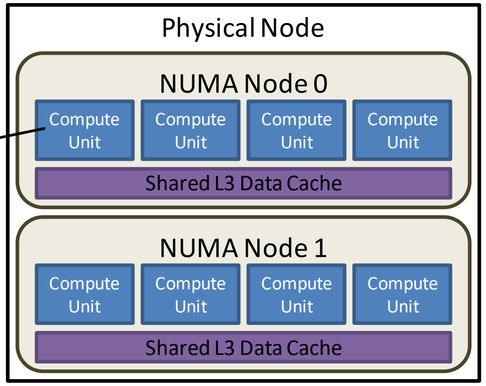
- Ray Decomposition

$$T_{OMP}(p_{ray}) = T_{PARALLEL}(p_{ray}) + (2T_{BARRIER}(p_{ray}) + T_{SINGLE}(p_{ray})) \times 4\mathrm{nangoct} + \left\lceil \frac{\mathrm{nlongray}}{p_{ray} \times \mathrm{chunk}} \right\rceil T_{SCHEDULE}(p_{ray})$$

Comparison of Measured and Predicted Parallel Efficiency for Strong Scaling

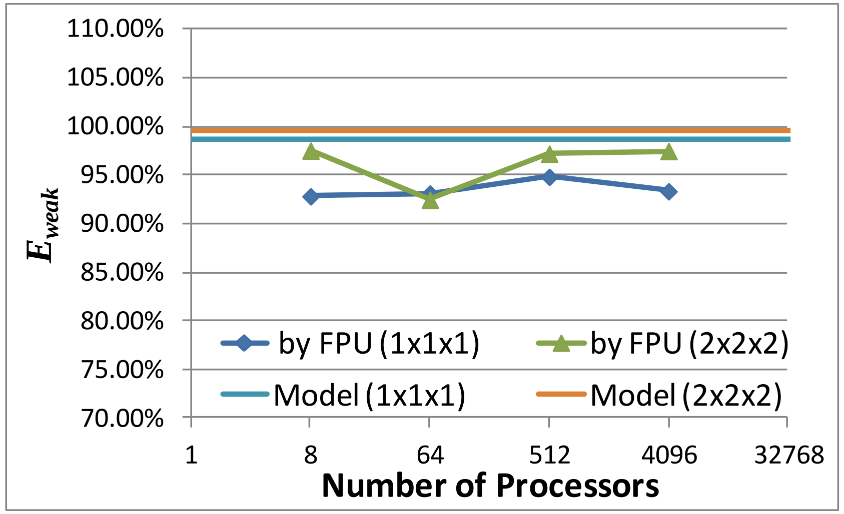


AMD Opteron 6200 Series
“Interlagos” Processor Micro-architecture

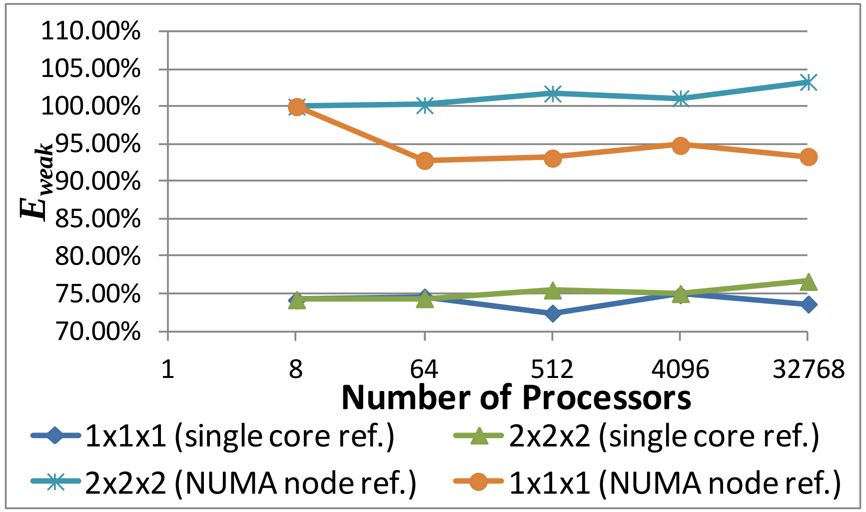


N_p	Ray Decomposition			Angle Decomposition			Space Decomposition		
	by FPU	by core	Model	by FPU	by core	Model	by FPU	by core	Model
2	104.9%	53.5%	99.40%	101.8%	80.00%	99.03%	N/A	N/A	N/A
4	100.5%	67.04%	97.89%	98.18%	78.88%	97.15%	N/A	N/A	N/A
8	87.49%	67.33%	94.22%	96.05%	71.03%	93.60%	98.91%	76.32%	99.52%
16	N/A	62.17%	86.78%	89.30%	63.85%	87.22%	N/A	N/A	N/A
64	N/A	N/A	N/A	N/A	N/A	N/A	93.14%	71.66%	98.69%

Parallel Efficiency of Spatial Weak Scaling

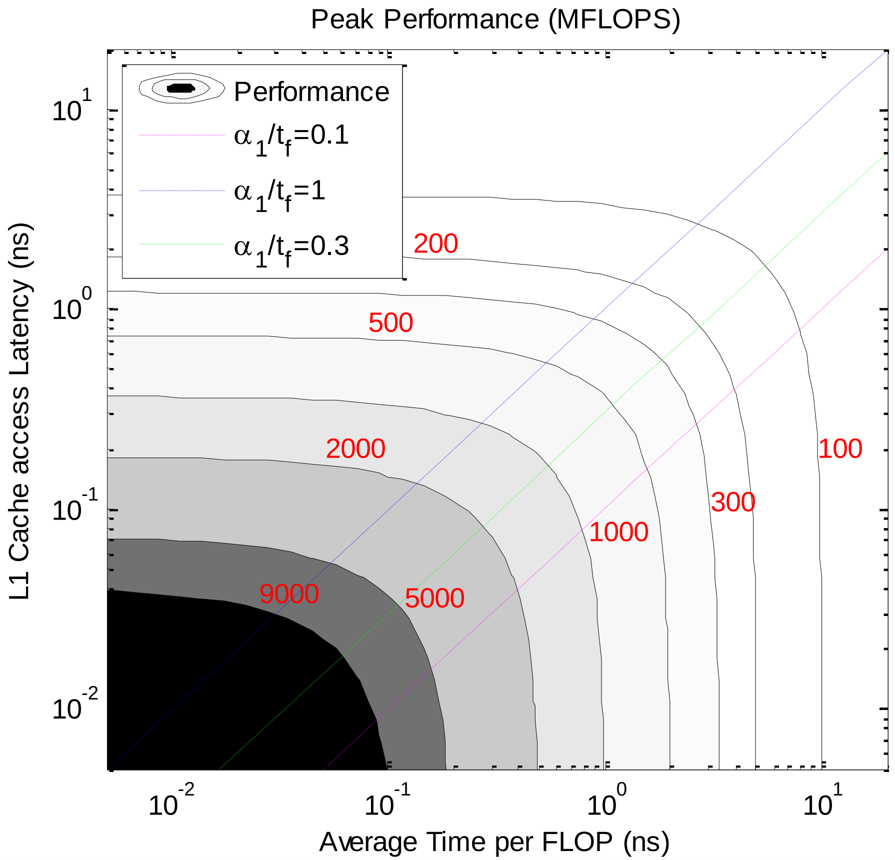


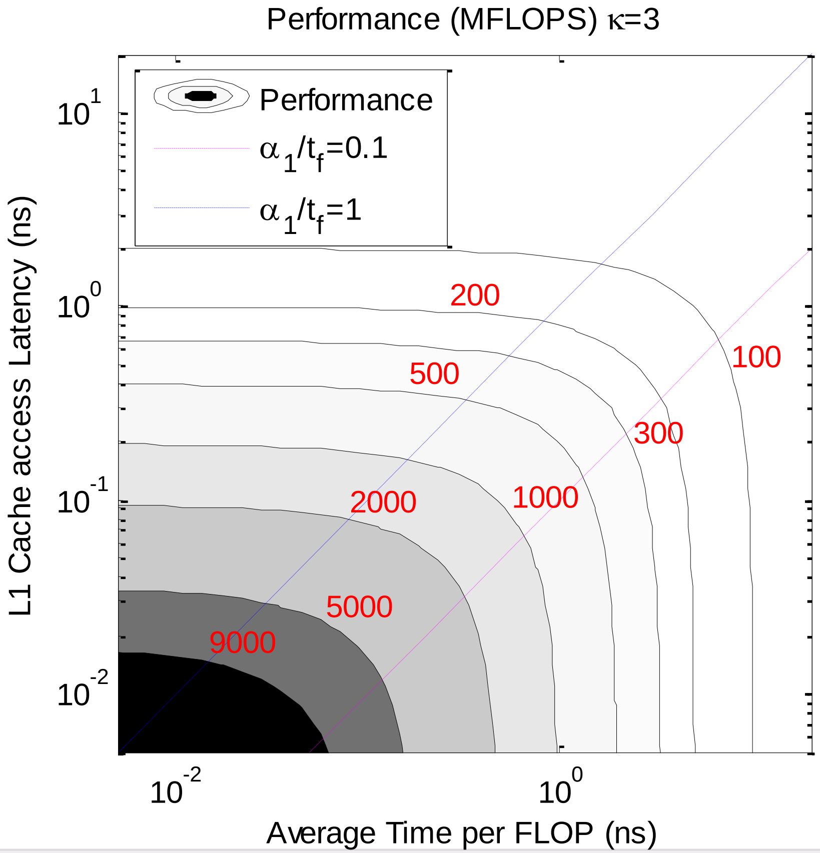
Comparison of Predicted and Measured Weak Scaling Efficiency for Spatial Decomposition



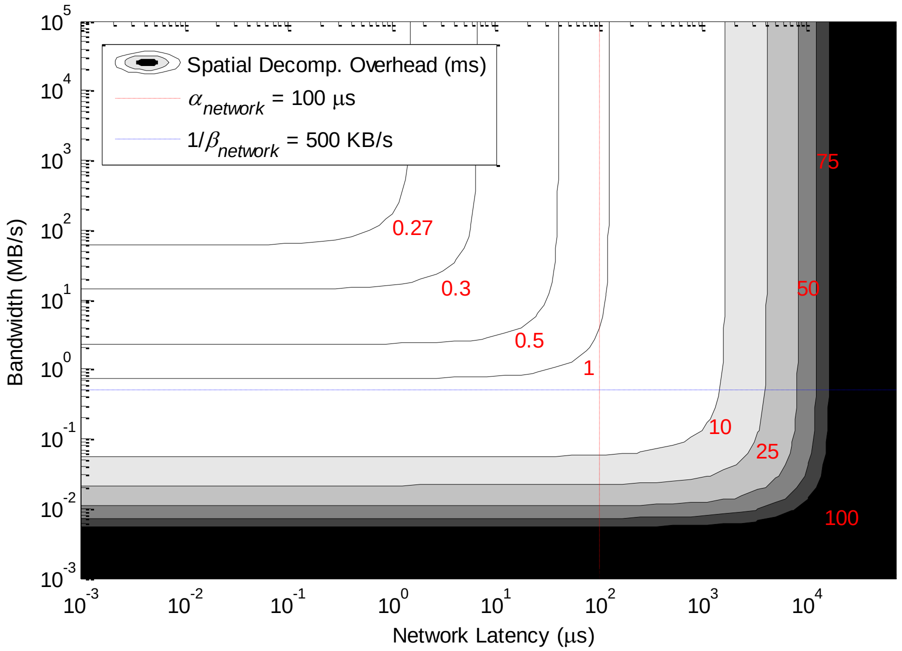
Weak Scaling Efficiency for Spatial Decomposition with Different Reference Cases

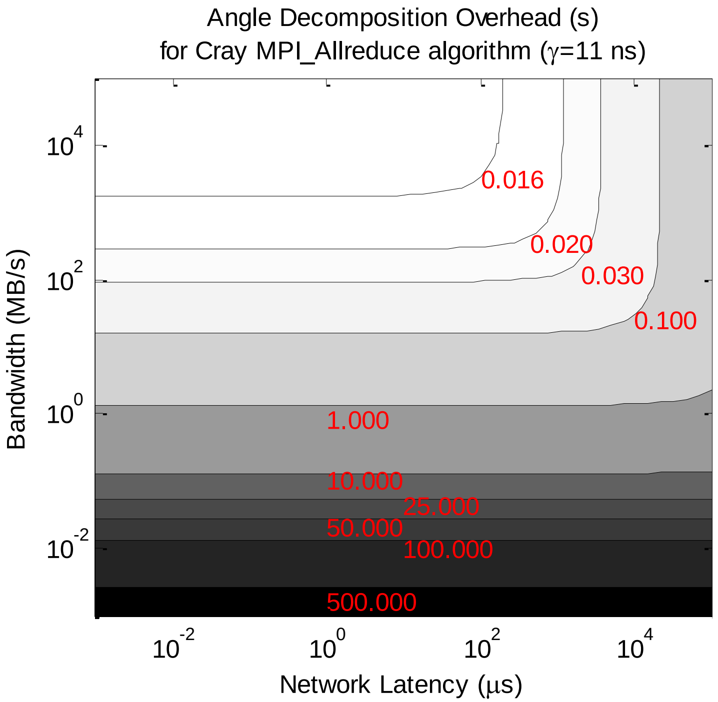
Serial Performance Model Hardware Sensitivities





Parallel Performance Model Network Hardware Sensitivity





Parallel Performance Model Sensitivity to Number of Domains