

Introduction to Parallel Programming

Overview

- Parallel programming allows the user to use multiple cpus concurrently
- Reasons for parallel execution:
 - shorten execution time by spreading the computational cycles across multiple cpus.
 - permit a larger problem by access to the combined memory of multiple cpus.
- The days of waiting for the next-generation chip to improve your serial code throughput are over.

Amdahl's Law

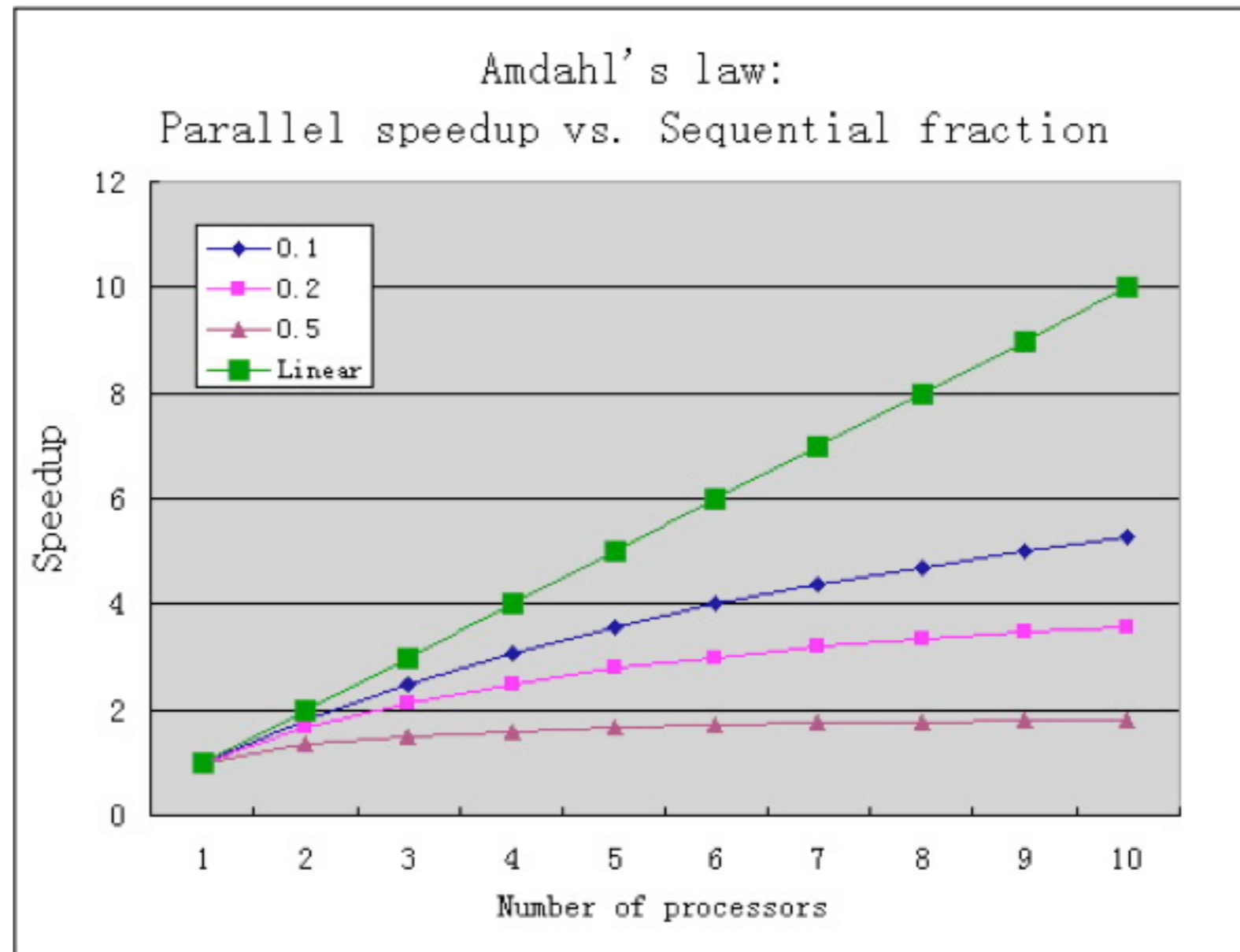
- Describes the speedup one can expect as a function of the number of processors (N) used and the code fraction that is parallel (p).

$$T(I) = T(I) * (1-p) + T(I) * p$$

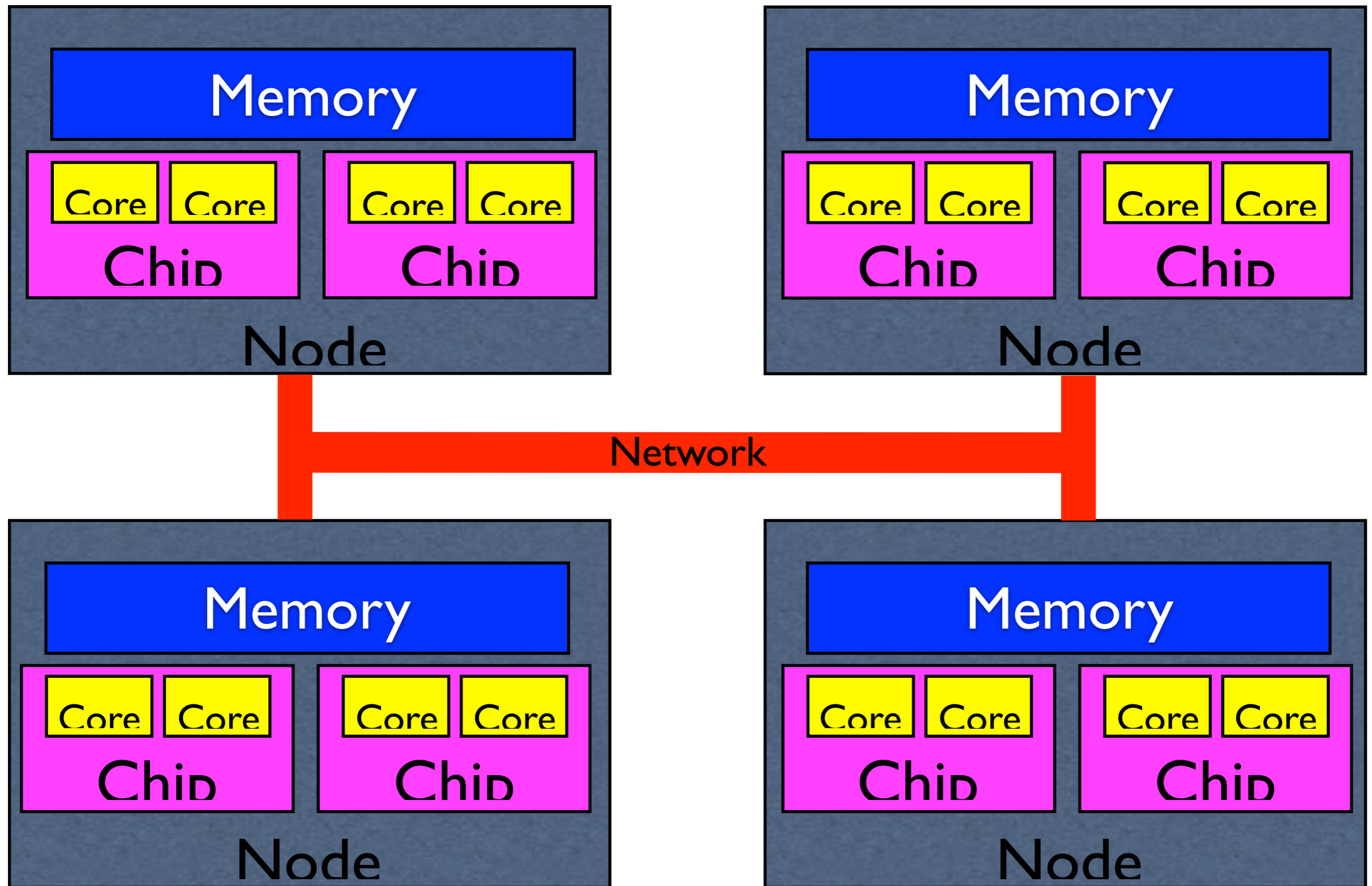
$$T(N) = T(I) * (1-p) + \frac{T(I) * p}{N}$$

$$\begin{aligned} \text{Speedup} &= T(I) / T(N) \\ &= 1 / ((1-p) + p/N) \end{aligned}$$

$$\text{Max Speedup} = 1 / (1-p)$$



General Parallel Architecture



Shared memory / Distributed memory

Types of Parallelism

- Process Parallelism (MPMD) - a code may contain different segments that can be computed concurrently. Example: ocean, land, atmosphere and ice parts of a climate model.
- Data Parallelism (SPMD) - the same code works on different datastreams. For example, dividing a global domain into subdomains - each processor executes all the code for an individual subdomain.
- Data and process parallelism may be employed together.

Parallel Programming Concepts

- Synchronization - making sure all code gets to a certain point before proceeding.
- Load balancing - trying to keep processes or threads from being idle while others are computing.
- Granularity - how large a chunk of work is in each parallel section - alleviates the overhead implementing parallel constructs.

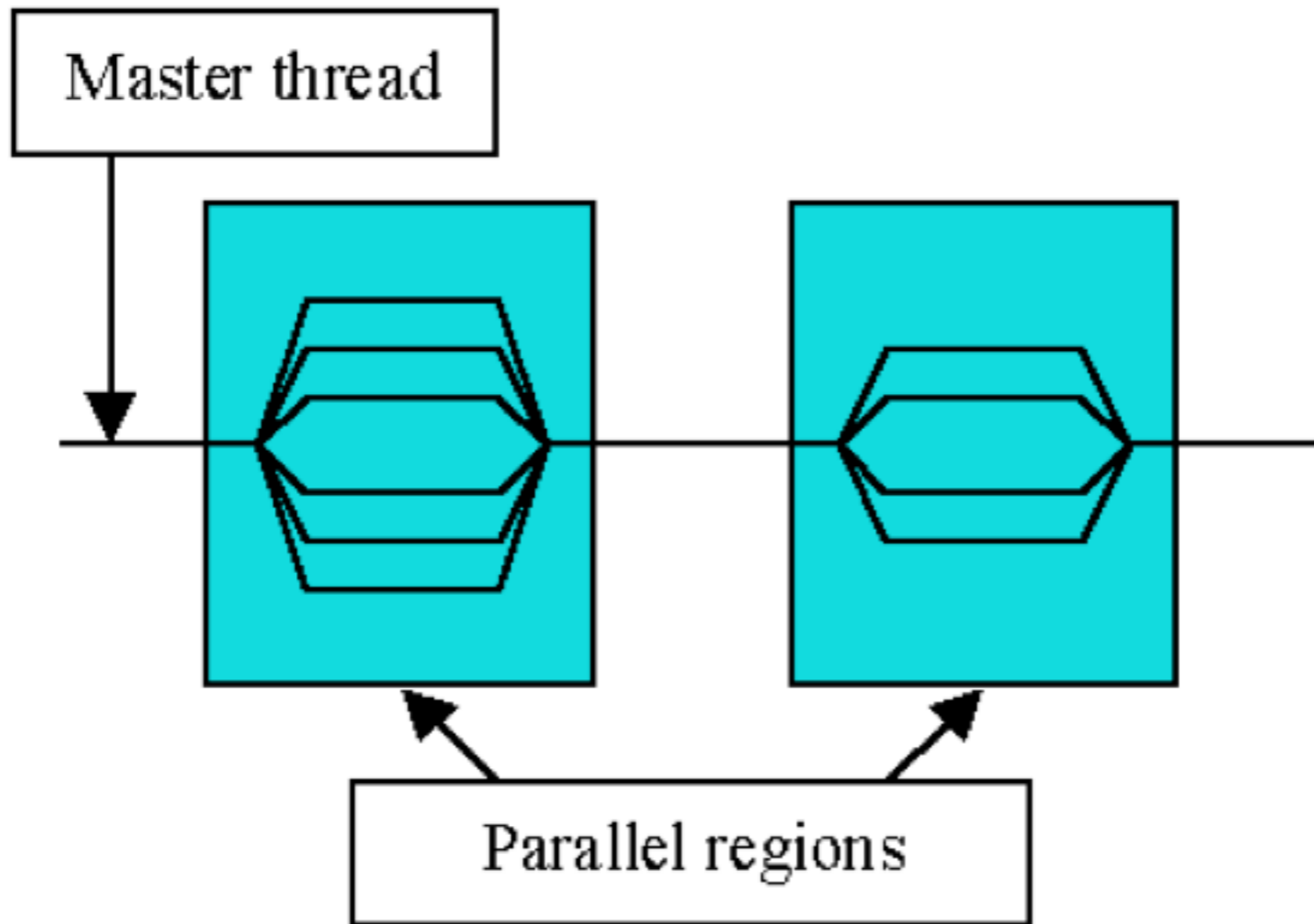
Parallel Programming Paradigms:

Shared Memory

- Shared memory techniques launch threads during execution
- Automatic Parallelizers - just turn on the compiler switch - it finds the do loops that can be done in parallel.
- Compiler Directives - **OpenMP** is the current standard. User inserts 'comments' in code that compiler recognizes as parallelization instructions. Only modest changes to code necessary.
- Only works with shared memory architecture.

See [hello_omp.f90](#)

Open MP - Overview



Tutorial:

<http://www.osc.edu/supercomputing/training/openmp/big/fslid.001.html>

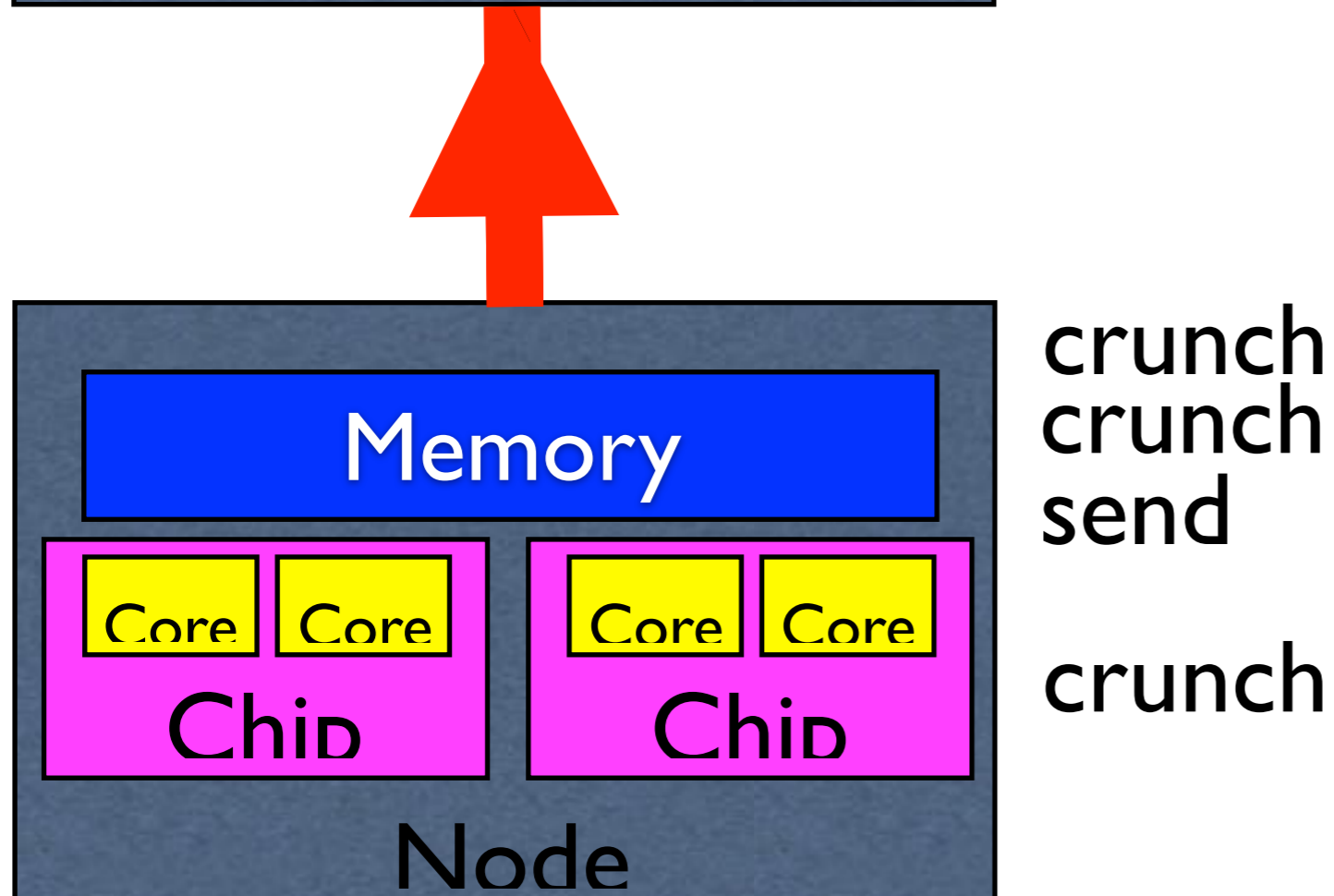
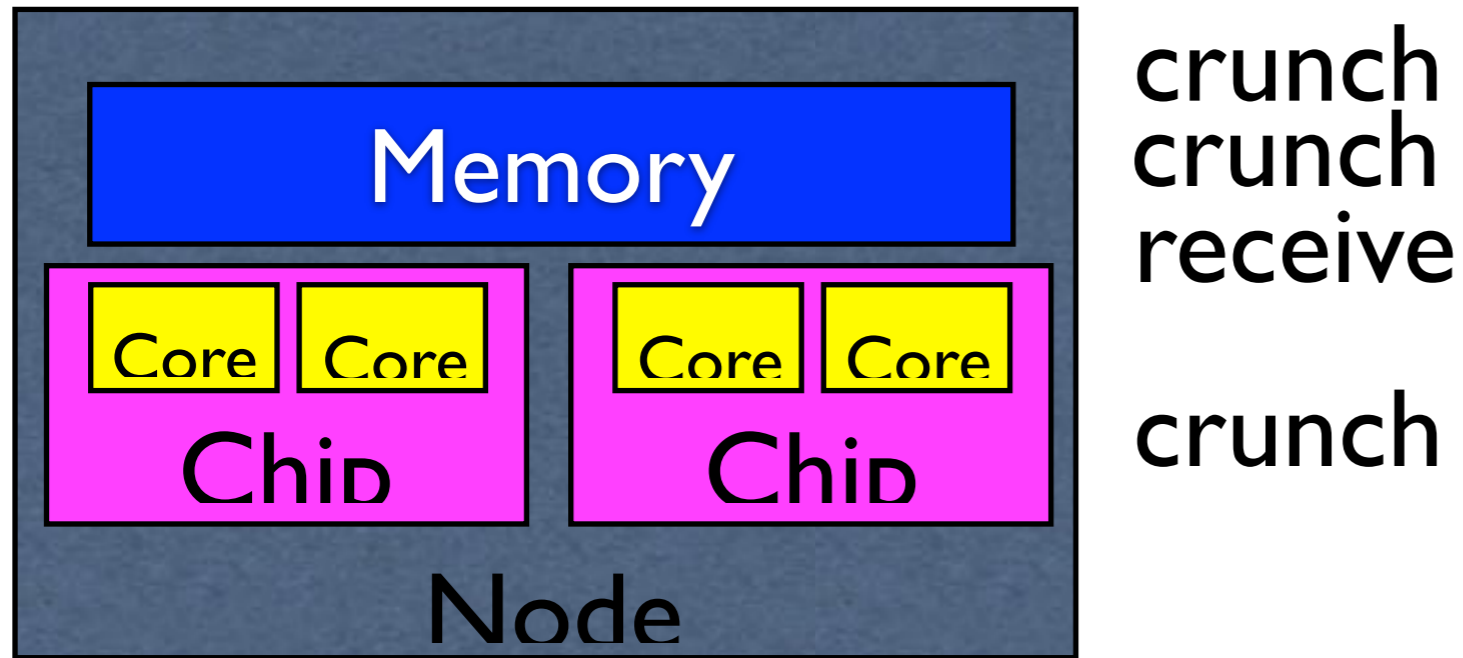
Openmp: <http://www.openmp.org/>

Parallel Programming Paradigms: Message Passing

- Can work with both distributed and shared memory architectures.
- **MPI** is the standard - comes in several implementations: MPICH2, open-mpi.
- Library calls explicitly control the parallel behavior - extensive user rewrite of code. Code is explicitly instructed to send and receive messages from other processes.
- Message passing and shared memory techniques and be used in a hybrid mode.

See [hello_mpi.f90](#)

Message Passing - Overview



Message Passing - MPI examples

- <https://computing.llnl.gov/tutorials/mpi/>

- Barriers

`MPI_BARRIER (comm, ierr)`

- Collective operations

`MPI_BCAST (buffer, count, datatype, root, comm, ierr)`

`MPI_REDUCE (sendbuf, recvbuf, count, datatype, op, root, comm, ierr)`

- Sends/receives (blocking and non blocking)

- `MPI_SEND (buf, count, datatype, dest, tag, comm, ierr)`

- `MPI_RECV (buf, count, datatype, source, tag, comm, status, ierr)`

- `MPI_ISEND (buf, count, datatype, dest, tag, comm, request, ierr)`

`MPI_Irecv (buf, count, datatype, source, tag, comm, request, ierr)`

`MPI_WAITALL (count, array_of_requests, array_of_statuses, ierr)`

- much more...

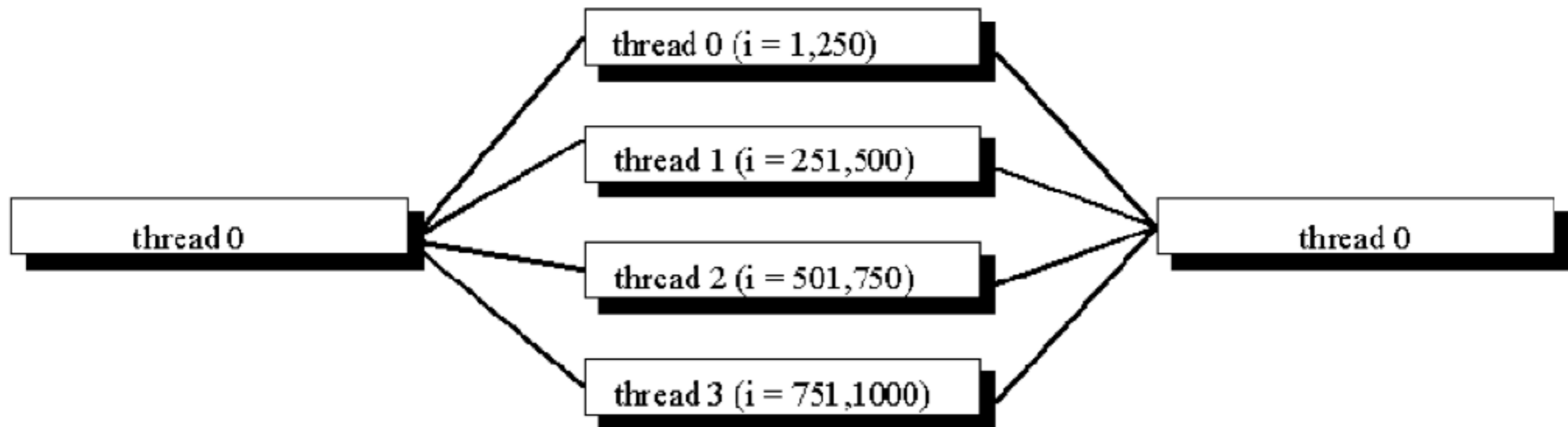
- compile with `mpif90`, run with `mpirun -np xx`

Open MP - First Steps

- Identify parallel do-loops. Each do loop carries overhead so it can be helpful to have a larger outer do-loop for parallelism.
- Identify functionally parallel regions.
- Identify **shared** and **private** data (scoping).
- Identify **race conditions** where shared data can change program output unexpectedly.
- Compile with extra flag (-fopenmp for gfortran)

Open MP - parallel do loop

```
c$omp do shared(x) private(i)
c$omp& schedule(static)
      do i = 1, 1000
        x(i) = a
      enddo
```



Open MP - reduction and sections

```
c$omp do shared(x) private(i)
c$omp&  reduction(+:sum)
    do i = 1, N
        sum = sum + x(i)
    enddo

c$omp do shared(x) private(i)
c$omp&  reduction(min:gmin)
    do i = 1, N
        gmin = min(gmin, x(i))
    end do
```

```
c$omp parallel
c$omp  sections

c$omp  section
    call computeXpart()
c$omp  section
    call computeYpart()
c$omp  section
    call computeZpart()

c$omp  end sections
c$omp end parallel

    call sum()
```

Open MP - data dependency

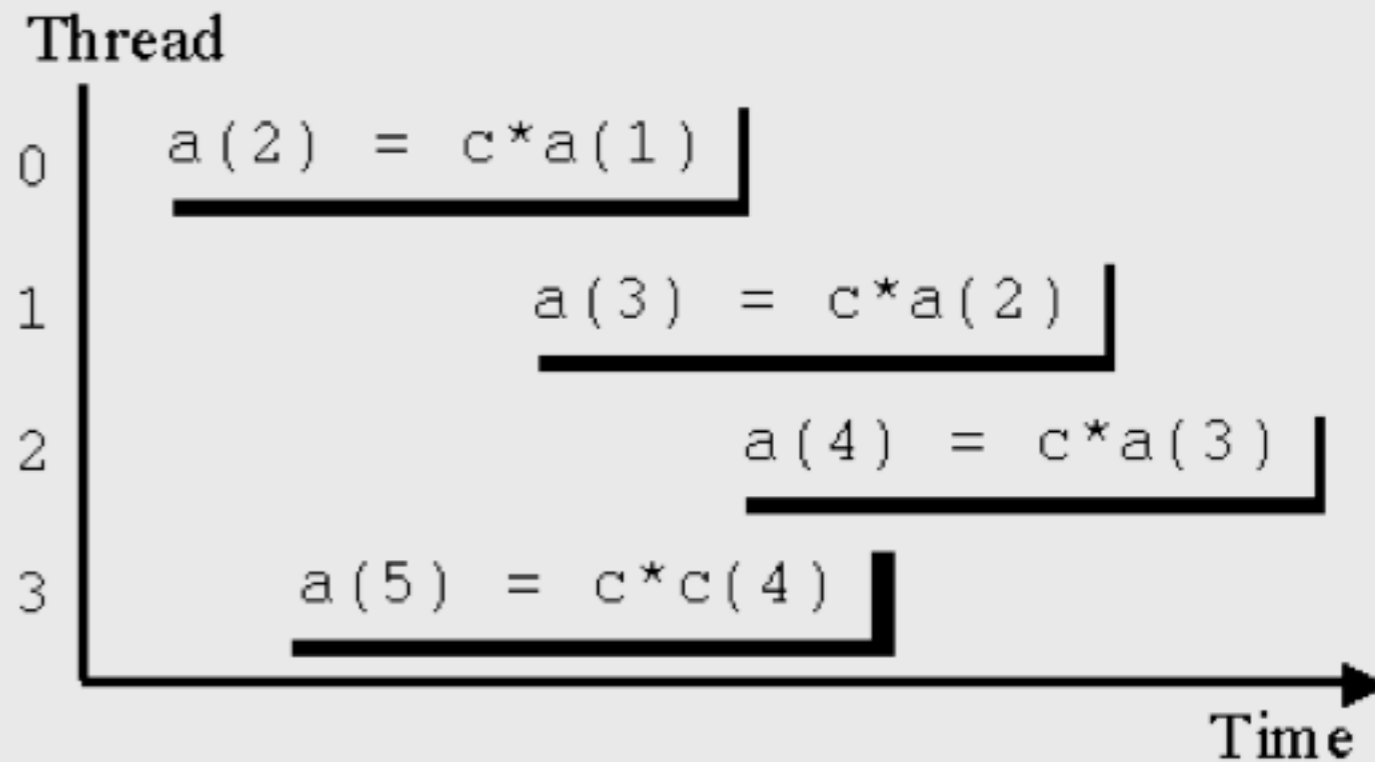
- Only variables that are **written** in one iteration and **read** in another iteration will create data dependencies.
- A variable cannot create a dependency unless it is **shared**.
- Often data dependencies are difficult to identify. **APO** can help by identifying the dependencies automatically.

Recurrence:

```
do i = 2, 5
  a(i) = c*a(i-1)
enddo
```

Is there a dependency here?

```
do i = 2, N, 2
  a(i) = c*a(i-1)
enddo
```



Open MP - Run time

- OpenMP execution can be controlled with environment variables
- **OMP_NUM_THREADS** - sets the number of threads requested for parallel execution.
- **OMP_DYNAMIC** - enables or disables dynamic adjustment of the number of threads used in a parallel region (due to system load).

Coarrays - Fortran 2008

- compilers just now getting it implemented
- extends array syntax of fortran with trailing subscripts in square brackets to denote the image (process)
- evolving rapidly - Fortran 2015 standard will be considerably more advanced.