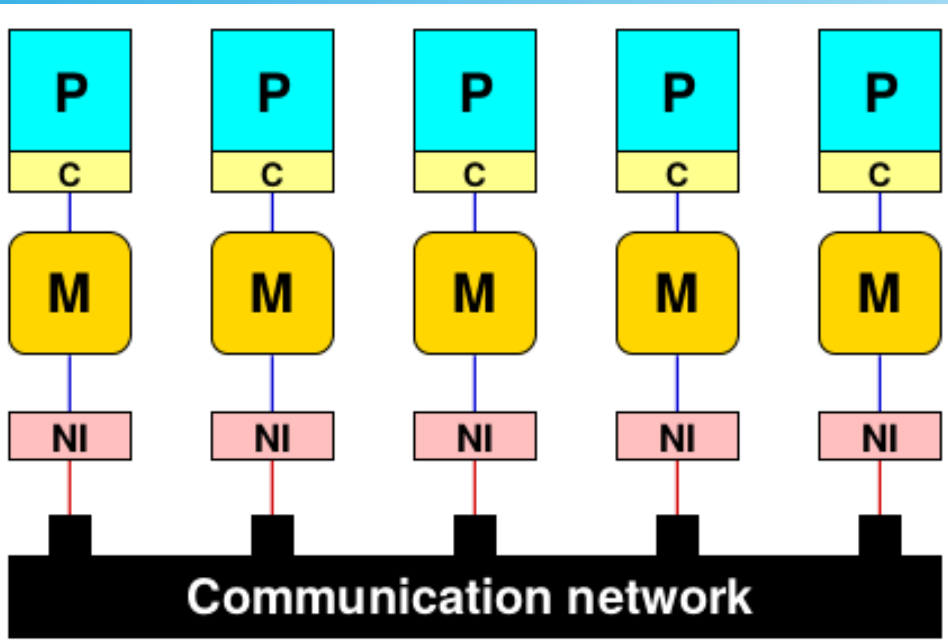
Primena super-računara u astronomiji

Shared-memory computing

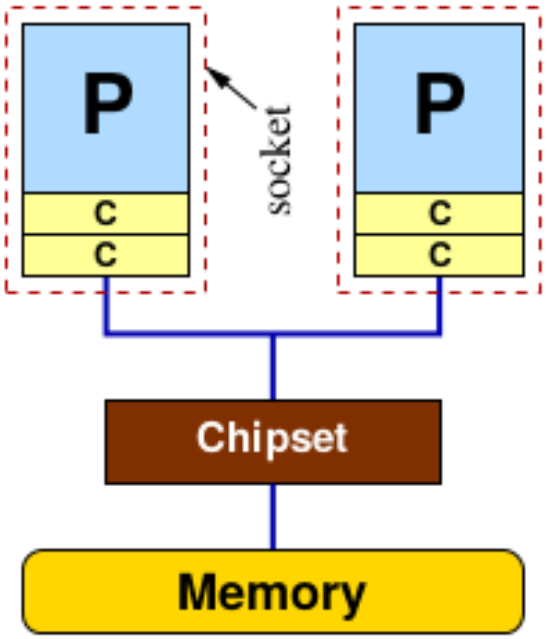
- A shared-memory parallel computer is a system in which a number of CPUs work on a common, shared physical address space
- Two main varieties of shared-memory systems:
  - **Uniform Memory Access (UMA)** systems feature a “flat” memory model: Memory bandwidth and latency are the same for all processors and all memory locations. This is also called symmetric multiprocessing (SMP)
  - On cache-coherent **Non-Uniform Memory Access (ccNUMA)** machines, memory is physically distributed, but logically shared.

Shared-memory computing



- The physical layout of such systems is quite similar to the distributed-memory case, but network logic makes the aggregated memory of the whole system appear as one single address space.
- Due to the distributed nature, memory access performance varies depending on which CPU accesses which parts of memory (“local” vs. “remote” access)

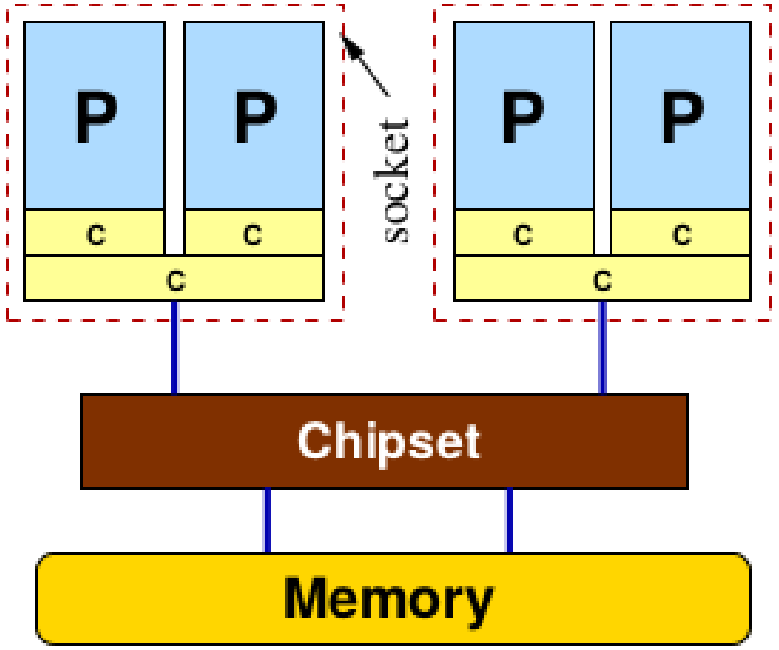
Uniform Memory Access (UMA)



- Two (single-core) processors, each in its own socket, communicate and access memory over a common bus, the **frontside bus (FSB)**.
- All arbitration protocols required to make this work are already built into the CPUs
- The chipset (aka “**northbridge**”) is responsible for driving the memory modules and connects to other parts of the node like
  I/O subsystems

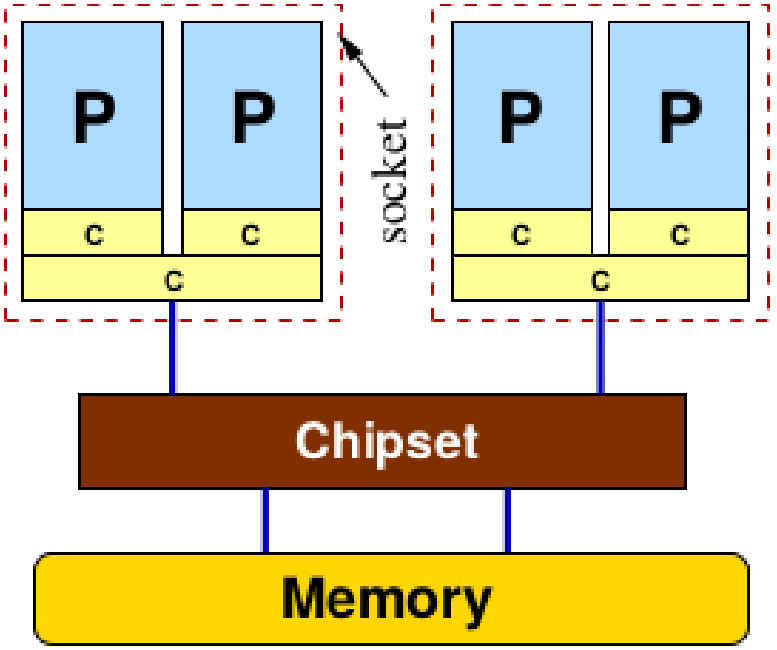
Uniform Memory Access (UMA)

- UMA system in which the FSBs of two dual-core chips are connected separately to the chipset
- The chipset plays an important role in enforcing cache coherence and also mediates the connection to memory.
- In principle, a system like this could be designed so that the bandwidth from chipset to memory matches the aggregated bandwidth of the frontside buses



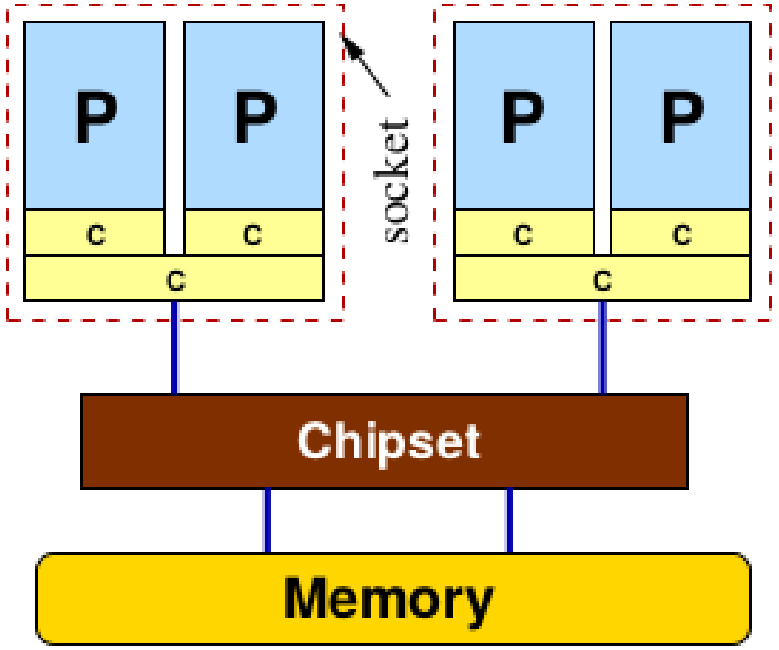
Uniform Memory Access (UMA)

- Each dual-core chip features a separate L1 on each CPU but a shared L2 cache for both
- The advantage of a shared cache is that, to an extent limited by cache size, data exchange between cores can be done there and does not have to resort to the slow frontside bus



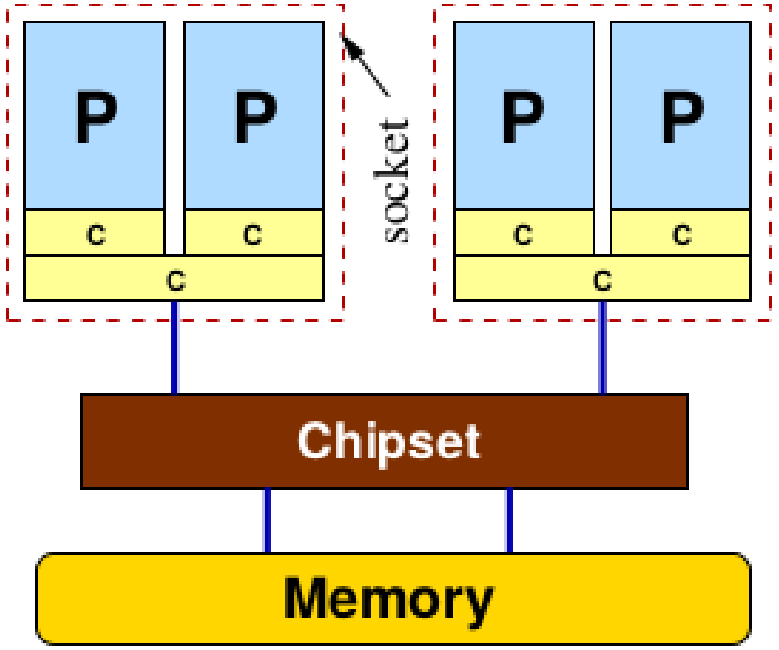
Uniform Memory Access (UMA)

- Due to the shared caches and FSB connections this kind of node is, while still a UMA system, quite sensitive to the exact placement of processes or threads on its cores.
- For instance, with only two processes it may be desirable to keep (“pin”) them on separate sockets if the memory bandwidth requirements are high.



Uniform Memory Access (UMA)

- On the other hand, processes communicating a lot via shared memory may show more performance when placed on the same socket because of the shared L2 cache.
- Operating systems as well as some modern compilers usually have tools or library functions for observing and implementing thread or process pinning.



Uniform Memory Access (UMA)

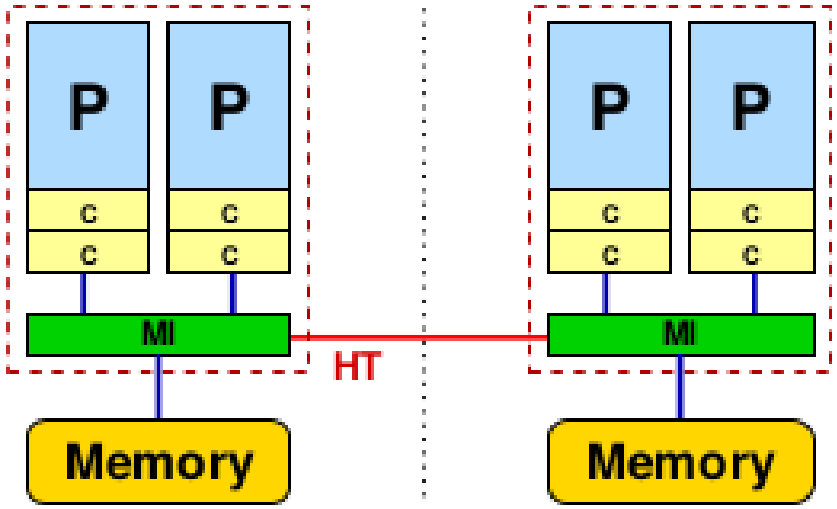
- The general problem of UMA systems is that bandwidth bottlenecks are bound to occur when the number of sockets, or FSBs, is larger than a certain limit
- In very simple designs, a common memory bus is used that can only transfer data to one CPU at a time
- In order to maintain scalability of memory bandwidth with CPU number, non-blocking crossbar switches can be built that establish point-to-point connections between FSBs and memory modules
- Due to the very large aggregated bandwidths those become very expensive for a larger number of sockets

Non-Uniform Memory Access (ccNUMA)

- **Locality domain (LD)** is a set of processor cores together with locally connected memory which can be accessed in the most efficient way, i.e. without resorting to a network of any kind
- ccNUMA principle provides scalable bandwidth for very large processor counts
- It is also found in inexpensive small two- or four-socket nodes

Non-Uniform Memory Access (ccNUMA)

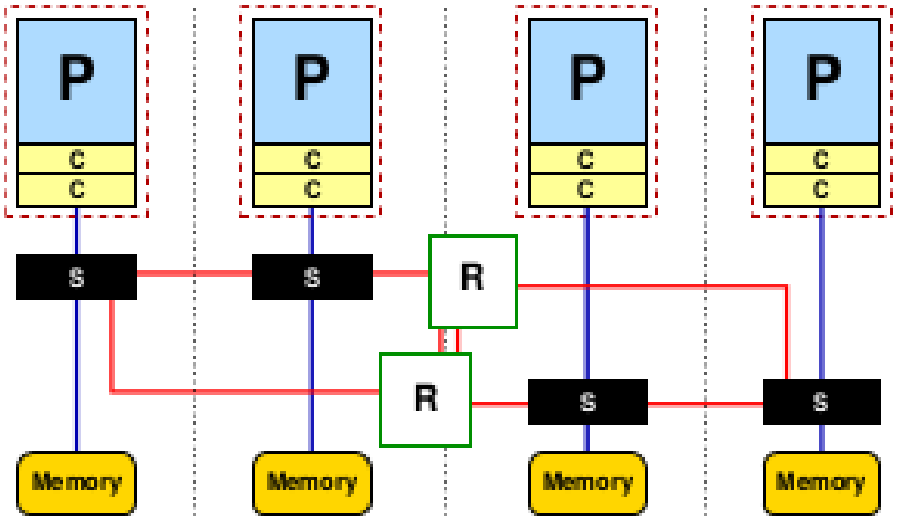
- **Example #1**: dual-core chips with separate caches and a common interface to local memory, are linked using a special high-speed connection called **HyperTransport (HT)**
- This system differs substantially from networked UMA designs in that the HT link can mediate direct coherent access from one processor to another processor's memory
- From the programmer's point of view this mechanism is transparent. All the required protocols are handled by the HT hardware



- **Figure:** HyperTransport-based cc-NUMA system with two locality domains (one per socket) and four cores

Non-Uniform Memory Access (ccNUMA)

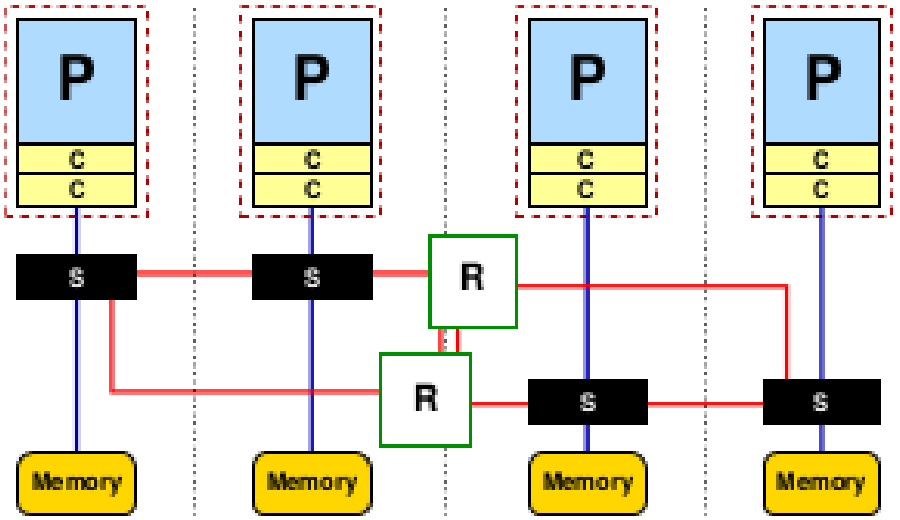
- **Example #2**: Each processor socket connects to a communication interface (S) that provides memory access as well as connectivity to the proprietary NUMALink (NL) network
- The NL network relies on routers (R) to switch connections for non-local access
- As with HT, the NL hardware allows for transparent access to the whole address space of the machine from all CPU



- **Figure:** ccNUMA system with routed NUMALink network and four locality domains

Non-Uniform Memory Access (ccNUMA)

- Multi-level router fabrics can be built that scale up to hundreds of CPUs
- It must, however, be noted that each piece of hardware inserted into a data connection (communication interfaces, routers) add to latency, making access characteristics very inhomogeneous across the system
- Furthermore, as is the case with networks for distributed-memory computers, providing wire-equivalent speed, non-blocking bandwidth in large systems is extremely expensive



- **Figure:** ccNUMA system with routed NUMALink network and four locality domains

Non-Uniform Memory Access (ccNUMA)

- In all ccNUMA designs, network connections must have bandwidth and latency characteristics that are at least the same order of magnitude as for local memory
- Although this is the case for all contemporary systems, even a penalty factor of two for non-local transfers can badly hurt application performance if access can not be restricted inside locality domains
- This locality problem is the first of two obstacles to take with high performance software on ccNUMA. It occurs even if there is only one serial program running on a ccNUMA machine
- The second problem is potential congestion if two processors from different locality domains access memory in the same locality domain, fighting for memory bandwidth
- Even if the network is non-blocking and its performance matches the bandwidth and latency of local access, congestion can occur
- Both problems can be solved by carefully observing the data access patterns of an application and restricting data access of each processor to its own locality domain

Cache coherence

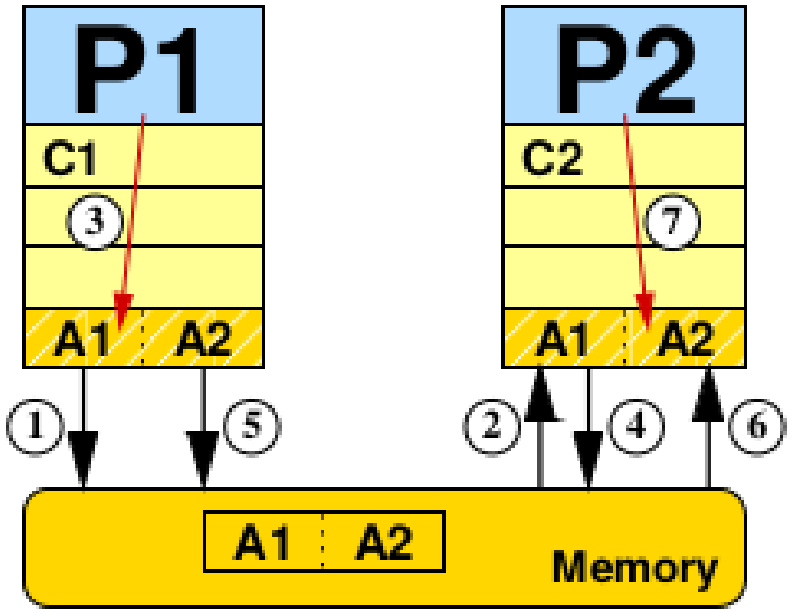
- Cache coherence mechanisms are required in all cache-based multiprocessor systems, UMA as well as ccNUMA
- This is because potentially copies of the same cache line could reside in several CPU caches
- Cache coherence protocols ensure a consistent view of memory under all circumstances
- **MESI** protocol

Cache coherence

- **MESI** protocol:
  - **M** *modified*: The cache line has been modified in this cache, and it resides in no other cache than this one. Only upon eviction will memory reflect the most current state
  - **E** *exclusive*: The cache line has been read from memory but not (yet) modified. However, it resides in no other cache
  - **S** *shared*: The cache line has been read from memory but not (yet) modified. There may be other copies in other caches of the machine
  - **I** *invalid*: The cache line does not reflect any sensible data. Under normal circumstances this happens if the cache line was in shared state and another processor has requested exclusive ownership. A cache miss occurs if and only if the chosen line is invalid

Cache coherence

- **Figure:** Two processors P1, P2 modify the two parts A1, A2 of the same cache line in caches C1 and C2. The MESI coherence protocol ensures consistency between cache and memory



1. C1 requests exclusive CL ownership
2. set CL in C2 to state I
3. CL has state E in C1 → modify A1 in C1 and set to state M
4. C2 requests exclusive CL ownership
5. evict CL from C1 and set to state I
6. load CL to C2 and set to state E
7. modify A2 in C2 and set to state M in C2

Cache coherence

- MESI protocol: simple to implement, this method has the crucial drawback that address broadcasts pollute the system buses and reduce available bandwidth for “useful” memory accesses
- A separate network for coherence traffic can alleviate this effect but is not always practicable
- A better alternative, usually applied in larger ccNUMA machines, is a directory-based protocol
- Bus logic like chipsets or memory interfaces keep track of the location and state of each cache line in the system
- This uses up a small part of main memory (usually far less than 10 %), but the advantage is that state changes of cache lines are transmitted only to those caches that actually require them
- This greatly reduces coherence traffic through the system
- Coherence traffic can severely hurt application performance if the same cache line is written to frequently by different processors - **false sharing**

Shared-memory programming with OpenMP

- Programming shared-memory systems can be done in an entirely “distributed-memory” fashion, i.e. the processes making up an MPI program can run on a UMA or ccNUMA machine
- On large “constellation” clusters systems where the number of nodes is smaller than the number of processors per node, the dominant parallelization method is often MPI due to its efficiency and flexibility
- However, the MPI is not only the most flexible but also the most tedious way of parallelization
- Shared memory opens the possibility to have immediate access to all data from all processors without explicit message passing

Shared-memory programming with OpenMP

- Shared memory opens the possibility to have immediate access to all data from all processors without explicit message passing
- The established standard in this field is **OpenMP**
- **OpenMP** is a set of compiler directives that a non-OpenMP capable compiler would just regard as comments and ignore
- The central entity in OpenMP is not a process but a **thread**
- **Threads** are also called “lightweight processes” because several of them can share a common address space and mutually access data
- Threads share everything but instruction pointer, stack pointer and register state
- Each thread can, by means of its local stack pointer, also have “private” variables, but as all data is accessible via the common address space, it is only a matter of taking the address of an item to make it accessible to all other threads as well

OpenMP worksharing and data scoping

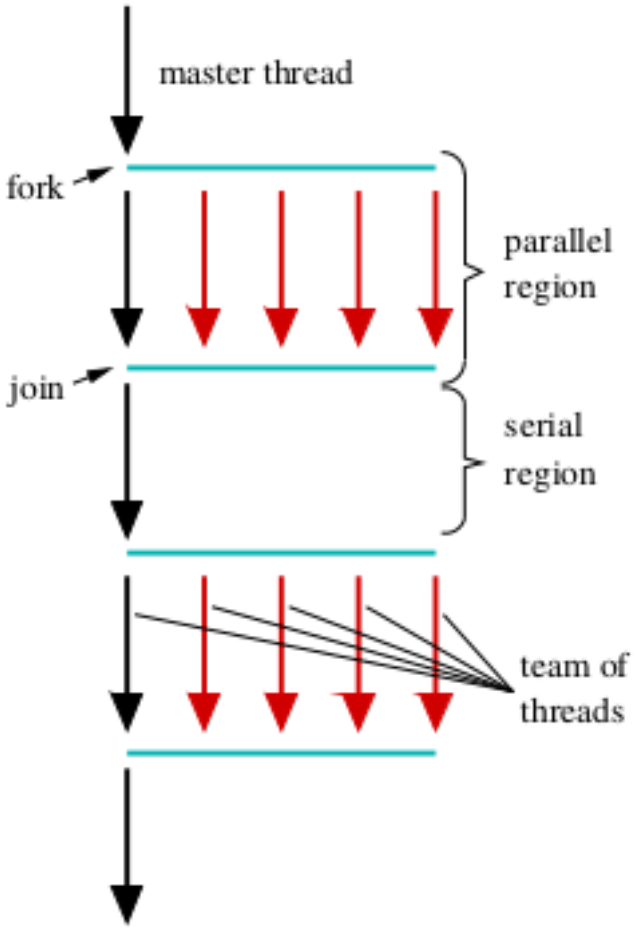
- OpenMP is a layer that adapts the raw OS thread interface to make it more usable with the typical loop structures
- Example: a parallel version of a simple integration program
- This is valid serial code, but equipping it with the comment lines starting with the sequence **!$OMP** (called a *sentinel*) and using an OpenMP-capable compiler makes it shared-memory parallel

```
1     pi=0.d0
2     w=1.d0/n
3   !$OMP PARALLEL PRIVATE(x,sum)
4     sum=0.d0
5   !$OMP DO SCHEDULE(STATIC)
6     do i=1,n
7       x=w*(i-0.5d0)
8       sum=sum+f(x)
9     enddo
10  !$OMP END DO
11  !$OMP CRITICAL
12    pi=pi+w*sum
13  !$OMP END CRITICAL
14  !$OMP END PARALLEL
```

OpenMP worksharing and data scoping

- The **PARALLEL** directive instructs the compiler to start a parallel region
- A team of threads is spawned that executes identical copies of everything up to **END PARALLEL**

```
1     pi=0.d0
2     w=1.d0/n
3   !$OMP PARALLEL PRIVATE(x,sum)
4     sum=0.d0
5   !$OMP DO SCHEDULE(STATIC)
6     do i=1,n
7       x=w*(i-0.5d0)
8       sum=sum+f(x)
9     enddo
10  !$OMP END DO
11  !$OMP CRITICAL
12    pi=pi+w*sum
13  !$OMP END CRITICAL
14  !$OMP END PARALLEL
```



OpenMP worksharing and data scoping

```
1     pi=0.d0
2     w=1.d0/n
3   !$OMP PARALLEL PRIVATE(x,sum)
4     sum=0.d0
5   !$OMP DO SCHEDULE(STATIC)
6     do i=1,n
7       x=w*(i-0.5d0)
8       sum=sum+f(x)
9     enddo
10  !$OMP END DO
11  !$OMP CRITICAL
12    pi=pi+w*sum
13  !$OMP END CRITICAL
14  !$OMP END PARALLEL
```

- By default, all variables which were present in the program before the parallel region are shared among all threads
- That would include **x** and **sum** of which we later need private versions for each thread
- OpenMP provides a way to make existing variables private by means of the **PRIVATE** clause
- If any thread in a parallel region writes to **sum** (see line 4), it will update its own private copy, leaving the other threads' untouched. Therefore, before the loop starts each thread's copy of **sum** is set to zero

OpenMP worksharing and data scoping

```
1     pi=0.d0
2     w=1.d0/n
3   !$OMP PARALLEL PRIVATE(x,sum)
4     sum=0.d0
5   !$OMP DO SCHEDULE(STATIC)
6     do i=1,n
7       x=w*(i-0.5d0)
8       sum=sum+f(x)
9     enddo
10  !$OMP END DO
11  !$OMP CRITICAL
12    pi=pi+w*sum
13  !$OMP END CRITICAL
14  !$OMP END PARALLEL
```

- In order to share some amount of work between threads and actually reduce wallclock time, work sharing directives can be applied.
- This is done using the **DO** directive with the optional **SCHEDULE** clause.
- The **DO** directive is always related to the immediately following loop (line 6) and generates code that distributes the loop iterations among the team of threads
- Note that the loop counter variable is automatically made private

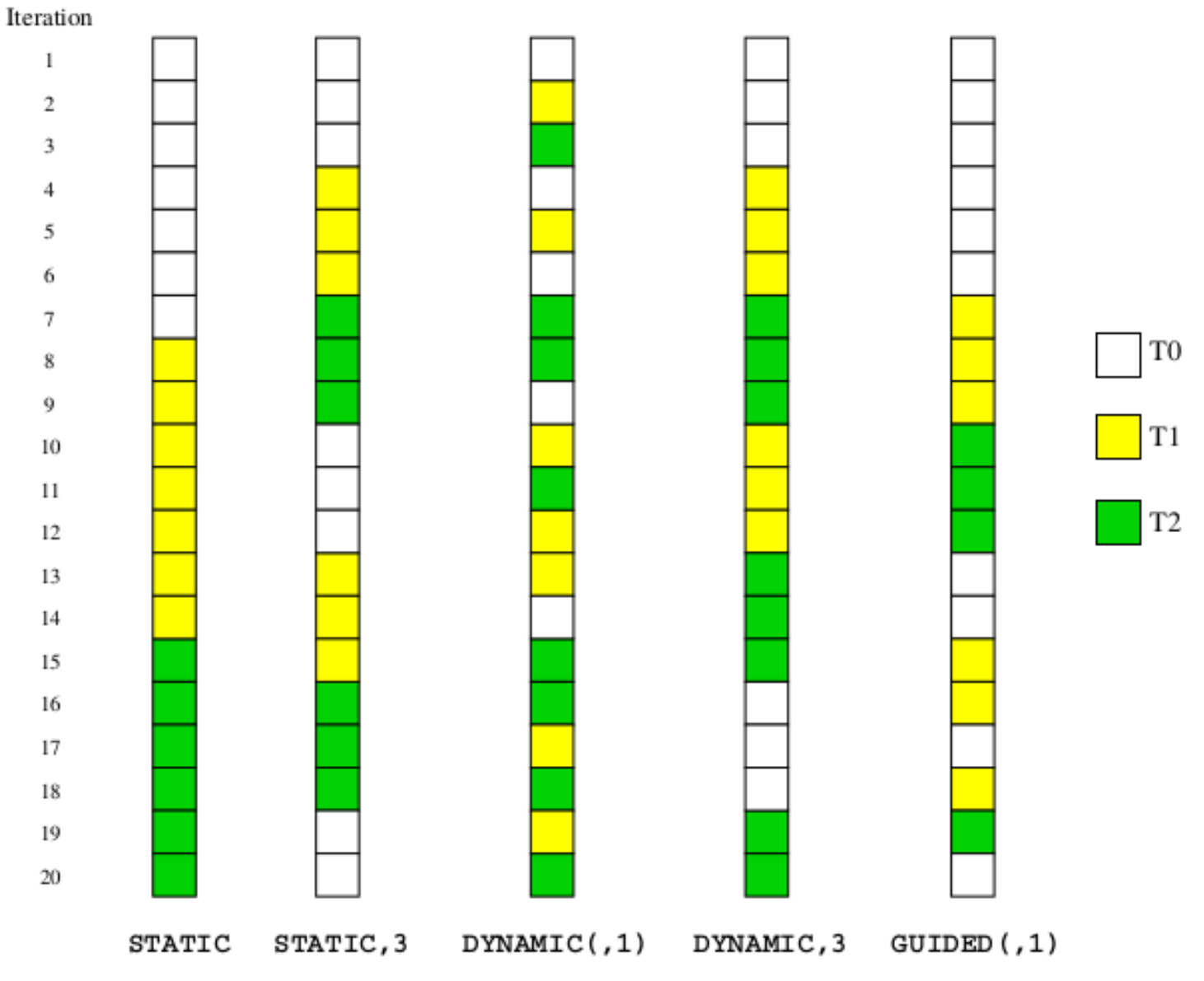
Loop scheduling

- This is controlled by the argument of **SCHEDULE**
- The simplest option is **STATIC** which divides the loop in contiguous chunks of approximately equal size
- If the amount of work per loop iteration is not constant, a solution would be to use a chunk size like in "**STATIC,1**" that dictates that chunks of size 1 should be distributed across threads in a round-robin manner
- There are alternatives to static schedule
- Dynamic scheduling assigns a chunk of work, defined by the chunk size, to the next thread that has finished its chunk
- This allows for a very flexible distribution which is usually not reproduced from run to run

Loop scheduling

- Dynamic scheduling assigns a chunk of work, defined by the chunk size, to the next thread that has finished its chunk
- This allows for a very flexible distribution which is usually not reproduced from run to run
- The downside is that dynamic scheduling generates significant overhead if the chunks are too small in terms of execution time
- In cases where this is a problem, the guided schedule may help
- Threads request new chunks dynamically, but the chunk size is always proportional to the remaining number of iterations divided by the number of threads
- Applications which are limited by memory bandwidth may suffer from insufficient access locality on ccNUMA systems

Loop scheduling



Protecting shared data

- The parallelized loop shown in the figure computes a partial **sum** in each thread's private sum variable
- To get the final result, all the partial sums must be accumulated in the global **pi** variable
- **pi** is shared so that uncontrolled updates would lead to a race condition, i.e. the exact order and timing of operations will influence the result

```
1     pi=0.d0
2     w=1.d0/n
3   !$OMP PARALLEL PRIVATE(x,sum)
4     sum=0.d0
5   !$OMP DO SCHEDULE(STATIC)
6     do i=1,n
7       x=w*(i-0.5d0)
8       sum=sum+f(x)
9     enddo
10  !$OMP END DO
11  !$OMP CRITICAL
12    pi=pi+w*sum
13  !$OMP END CRITICAL
14  !$OMP END PARALLEL
```

- In **OpenMP**, critical sections solve this problem by making sure that at most one thread at a time executes some piece of code.
- The **CRITICAL** and **END CRITICAL** directives bracket the update to **pi** so that a correct result emerges at all times

Protecting shared data

- Critical sections hold the danger of *deadlocks* when used inappropriately
- A *deadlock* arises when one or more threads wait for resources that will never become available, a situation that is generated with badly arranged **CRITICAL** directives
- When a thread encounters a **CRITICAL** directive inside a critical region, it will block forever

Protecting shared data

- OpenMP provides two solutions to the problem:

  1) A critical section may be given a name that distinguishes it from others. The name is specified in parentheses after the **CRITICAL** directive

```
!$OMP PARALLEL DO PRIVATE(x)
  do i=1,N
    x=sin(2*PI*x/N)
!$OMP CRITICAL (psum)
    sum=sum+func(x)
!$OMP END CRITICAL (psum)
  enddo
!$OMP END PARALLEL DO
  ...
  SUBROUTINE func(v)
  double precision v
!$OMP CRITICAL (prand)
  v=v+random_func()
!$OMP END CRITICAL (prand)
  END SUBROUTINE func
```

Protecting shared data

- OpenMP provides two solutions to the problem:

  2) There are OpenMP API functions that support the use of locks for protecting shared resources. The advantage of locks is that they are ordinary variables that can be arranged as arrays or in structures. That way it is possible to protect each single element of an array of resources individually

```
1   !$ use omp_lib
2     myid=0
3     numthreads=1
4   #ifdef _OPENMP
5   !$OMP PARALLEL PRIVATE(myid)
6     myid = omp_get_thread_num()
7   !$OMP SINGLE
8     numthreads = omp_get_num_threads()
9   !$OMP END SINGLE
10  !$OMP CRITICAL
11    write(*,*) 'Parallel program - this is thread ',myid,&
12                                              ' of ',numthreads
13  !$OMP END CRITICAL
14  !$OMP END PARALLEL
15  #else
16    write(*,*) 'Serial program'
17  #endif
```

Protecting shared data

- OpenMP provides two solutions to the problem:

  2) There are OpenMP API functions that support the use of locks for protecting shared resources. The advantage of locks is that they are ordinary variables that can be arranged as arrays or in structures. That way it is possible to protect each single element of an array of resources individually

```
1   !$ use omp_lib
2     myid=0
3     numthreads=1
4   #ifdef _OPENMP
5   !$OMP PARALLEL PRIVATE(myid)
6     myid = omp_get_thread_num()
7   !$OMP SINGLE
8     numthreads = omp_get_num_threads()
9   !$OMP END SINGLE
10  !$OMP CRITICAL
11    write(*,*) 'Parallel program - this is thread ',myid,&
12                                              ' of ',numthreads
13  !$OMP END CRITICAL
14  !$OMP END PARALLEL
15  #else
16    write(*,*) 'Serial program'
17  #endif
```

Protecting shared data

- OpenMP provides two solutions to the problem:

  2) There are OpenMP API functions that support the use of locks for protecting shared resources. The advantage of locks is that they are ordinary variables that can be arranged as arrays or in structures. That way it is possible to protect each single element of an array of resources individually

```
1   !$ use omp_lib
2     myid=0
3     numthreads=1
4   #ifdef _OPENMP
5   !$OMP PARALLEL PRIVATE(myid)
6     myid = omp_get_thread_num()
7   !$OMP SINGLE
8     numthreads = omp_get_num_threads()
9   !$OMP END SINGLE
10  !$OMP CRITICAL
11    write(*,*) 'Parallel program - this is thread ',myid,&
12                                              ' of ',numthreads
13  !$OMP END CRITICAL
14  !$OMP END PARALLEL
15  #else
16    write(*,*) 'Serial program'
17  #endif
```