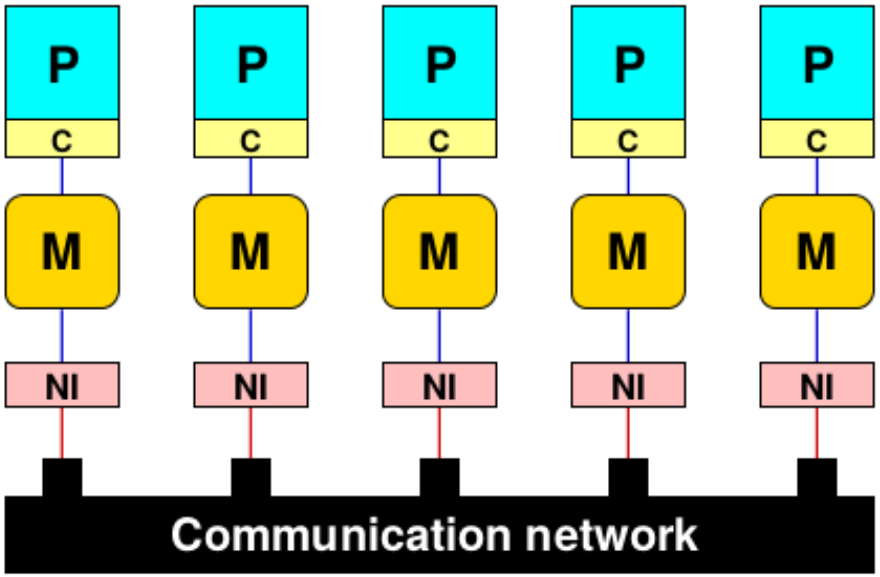
Primena super-računara u astronomiji

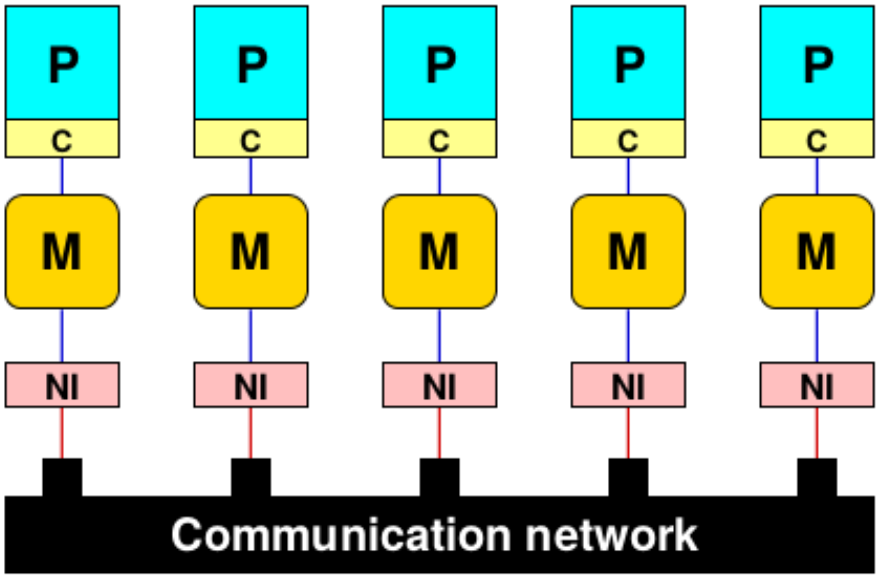
Distributed-memory computing



- **Simplified block diagram (programming model) of a distributed-memory parallel computer**

- Each processor **P** (with its own local cache **C**) is connected to exclusive local memory **M**
- Each node comprises at least one network interface (NI) that mediates the connection to a communication network
- There is a number of advanced technologies that have ten times the bandwidth and 1/10 th of the latency of Gbit Ethernet

Distributed-memory computing



- **Simplified block diagram (programming model) of a distributed-memory parallel computer**

- The most favourable design consists of a non-blocking “wirespeed” network that can switch N/2 connections between its N participants without any bottlenecks
- Problem: non-blocking switch fabrics become vastly expensive on very large installations

Message Passing

- Distributed-memory parallel programming requires the use of explicit message passing (MP), i.e. communication between processes
- There is an established standard for message passing called **MPI** (Message Passing Interface)
- MPI conforms to the following rules:
  - The same program runs on all processes (Single Program Multiple Data - SPMD). All processes taking part in a parallel calculation can be distinguished by a unique identifier called rank
  - The program is written in a sequential language like Fortran, C or C++. Data exchange, i.e. sending and receiving of messages, is done via calls to an appropriate library
  - All variables in a process are local to this process. There is no concept of shared memory

Message Passing

- In a message passing program, messages move data between processes
- For a message to be transmitted in an orderly manner, **some parameters have to be fixed in advance**:
    - Which processor is sending the message
    - Where is the data on the sending processor
    - What kind of data is being sent
    - How much data is there
    - Which process/es is/are going to receive the message
    - Where should the data be left on the receiving process(es)
    - How much data are the receiving processes prepared to accept
- All MPI calls that actually transfer data have to specify those parameters in some way

A brief glance on MPI

```
1   program mpitest
2   use MPI
3
4   integer rank, size, ierror
5
6   call MPI_Init(ierror)
7   call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
8   call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)
9
10  write(*,*) 'Hello World, I am ',rank,' of ',size
11
12  call MPI_Finalize(ierror)
13
14  end
```

- The first call in every MPI code should go to **MPI_Init** and initializes the parallel environment
- After initialization, MPI has set up a so-called communicator, called **MPI_COMM_WORLD**
- The calls to **MPI_Comm_size** and **MPI_Comm_rank** serve to determine the number of processes (size) in the parallel program and the unique identifier (rank) of the calling process

A brief glance on MPI

```
1   program mpitest
2   use MPI
3
4   integer rank, size, ierror
5
6   call MPI_Init(ierror)
7   call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
8   call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)
9
10  write(*,*) 'Hello World, I am ',rank,' of ',size
11
12  call MPI_Finalize(ierror)
13
14  end
```

- The ranks in a communicator, in this case **MPI_COMM_WORLD,** are numbered starting from zero up to N−1
- The parallel program is shut down by a call to **MPI_Finalize**
- The output of the program could look like the following:
  - ***Hello World, I am 3 of 4***
  - ***Hello World, I am 0 of 4***
  - ***Hello World, I am 2 of 4***
  - ***Hello World, I am 1 of 4***

```
   integer stat(MPI_STATUS_SIZE)
   call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
   call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)
! integration limits
   a=0.d0
   b=2.d0
   res=0.d0
! limits for "me"
   mya=a+rank*(b-a)/size
   myb=mya+(b-a)/size
! integrate f(x) over my own chunk - actual work
   psum = integrate(mya,myb)
! rank 0 collects partial results
   if(rank.eq.0) then
      res=psum
      do i=1,size-1
         call MPI_Recv(tmp, &   ! receive buffer
                       1,   &   ! array length
                            ! datatype
                       MPI_DOUBLE_PRECISION,&
                       i,   &   ! rank of source
                       0,   &   ! tag (additional label)
                            ! communicator
                       MPI_COMM_WORLD,&
                       stat,&   ! status array (msg info)
                       ierror)
         res=res+tmp
      enddo
      write (*,*) 'Result: ',res
! ranks != 0 send their results to rank 0
   else
      call MPI_Send(psum,   &   ! send buffer
                    1,      &   ! array length
                    MPI_DOUBLE_PRECISION,&
                    0,      &   ! rank of destination
                    0,      &   ! tag
                    MPI_COMM_WORLD,ierror)
   endif
```

- MPI program fragment that computes an integral over some function f(x) in parallel
- Each MPI process gets assigned a subinterval of the integration domain
- Some other function can then perform the actual integration

```
call MPI_Reduce(psum,   & ! send buffer
                res,    & ! receive buffer
                1,      & ! array length
                MPI_DOUBLE_PRECISION,&
                MPI_SUM,& ! type of operation
                0,      & ! root (accumulate res there)
                MPI_COMM_WORLD,ierror)
```

A brief glance on MPI

- All MPI functionalities described so far have the property that the call returns to the user program only after the message transfer has progressed far enough so that the send/receive buffer can be used without problems
- This is called blocking communication
- Non-blocking MPI is a way in which computation and communication may be overlapped
- Non-blocking and blocking MPI calls are mutually compatible, i.e. a message sent via a blocking send can be matched by a non-blocking receive

A brief glance on MPI - Python

- Python objects (`pickle` under the hood):

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'a': 7, 'b': 3.14}
    comm.send(data, dest=1, tag=11)
elif rank == 1:
    data = comm.recv(source=0, tag=11)
```

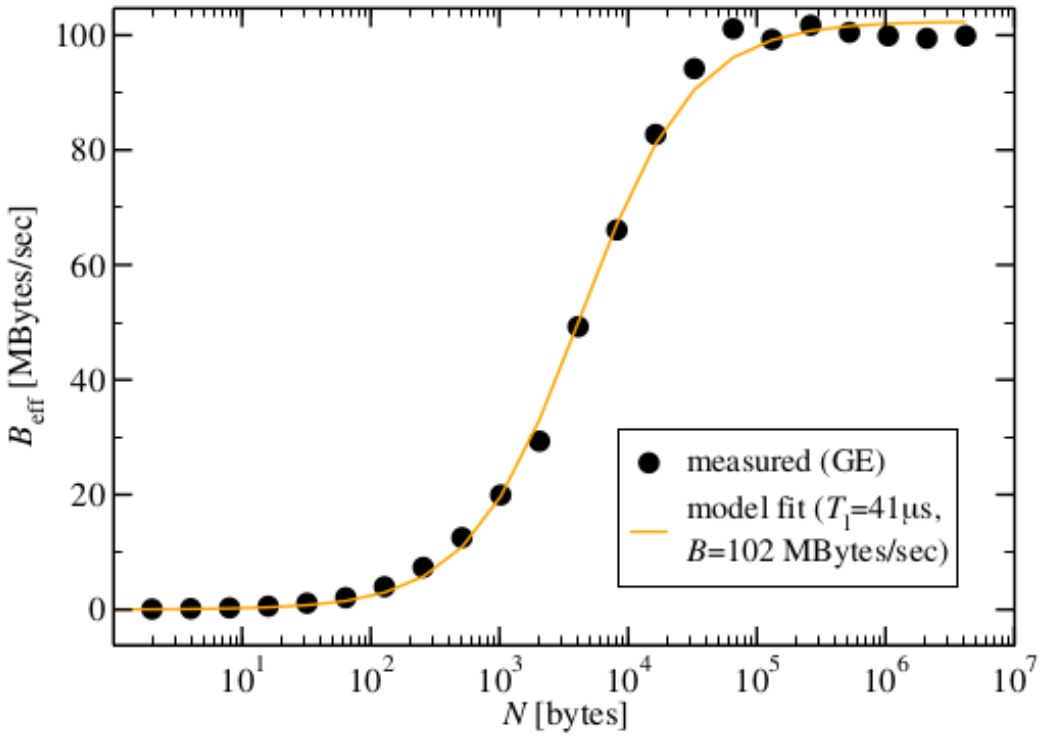
- Python objects with non-blocking communication:

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'a': 7, 'b': 3.14}
    req = comm.isend(data, dest=1, tag=11)
    req.wait()
elif rank == 1:
    req = comm.irecv(source=0, tag=11)
    data = req.wait()
```

Basic performance characteristics of networks

- There are various options for the choice of a network in a distributed-memory computer
- Assuming that the total transfer time for a message of size **N** is: $$T = T_l + \frac{N}{B}$$
- The effective bandwidth is: $$B_{\text{eff}} = \frac{N}{T_l + \frac{N}{B}}$$

Basic performance characteristics of networks



- **Fit of the model for effective bandwidth to data measured on a Gbit Ethernet network**

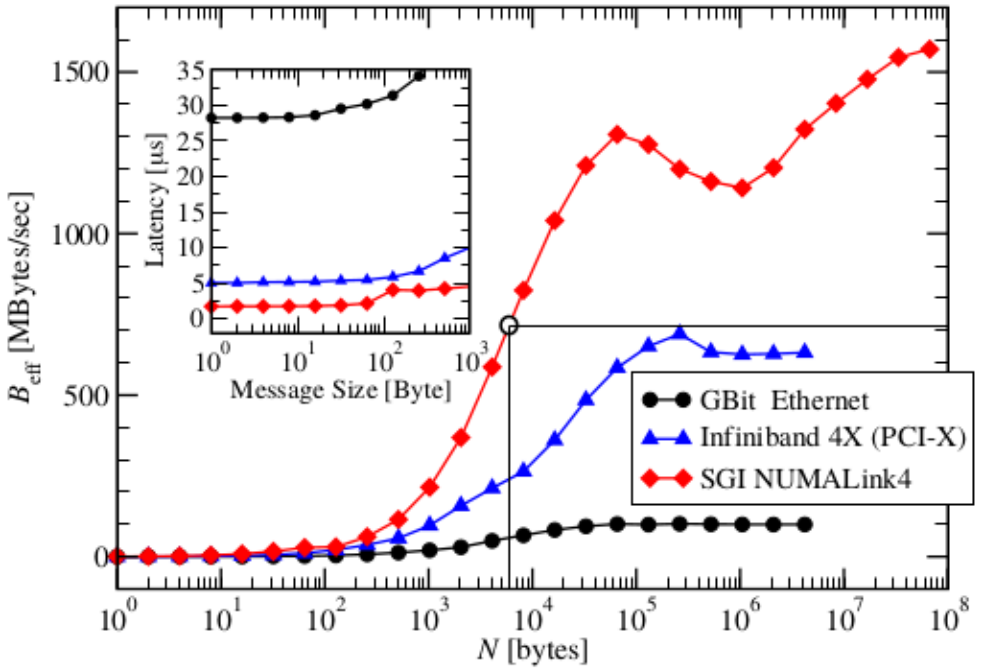
- This simple model is able to describe the gross features well

Basic performance characteristics of networks

- For the measurement of effective bandwidth the PingPong benchmark is frequently used

```
S = get_walltime()
if(rank.eq.0) then
  call MPI_Send(buf,N,MPI_BYTE,1,0,...)
  call MPI_Recv(buf,N,MPI_BYTE,1,0,...)
else
  call MPI_Recv(buf,N,MPI_BYTE,0,0,...)
  call MPI_Send(buf,N,MPI_BYTE,0,0,...)
endif
E = get_walltime()
MBYTES  = 2*N/(E-S)/1.d6     ! MByte/sec rate
TIME    = (E-S)/2*1.d6       ! transfer time in microsecs
                             ! for single message
```

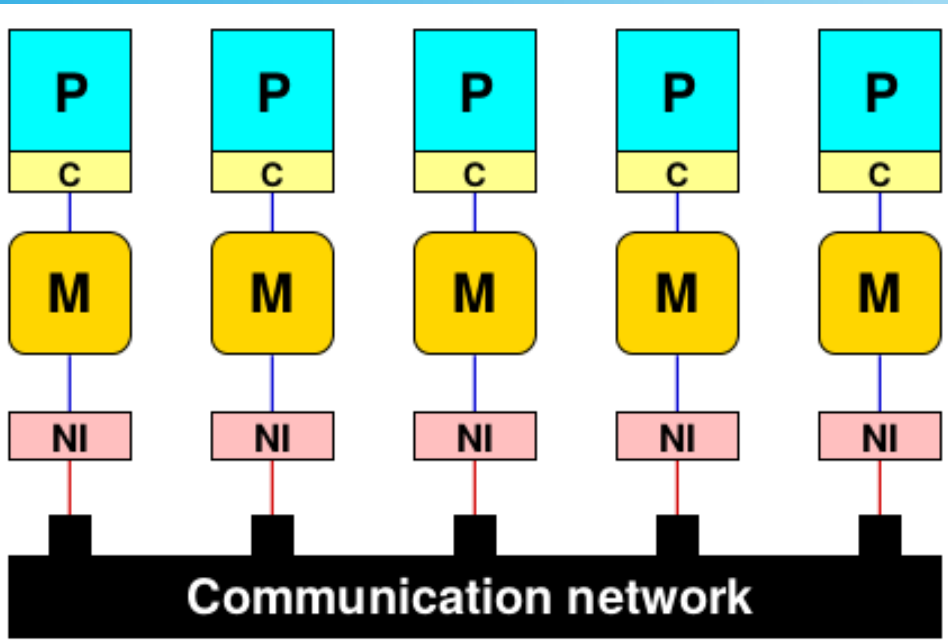
- **Result of the PingPong benchmark for three different networks**



Shared-memory computing

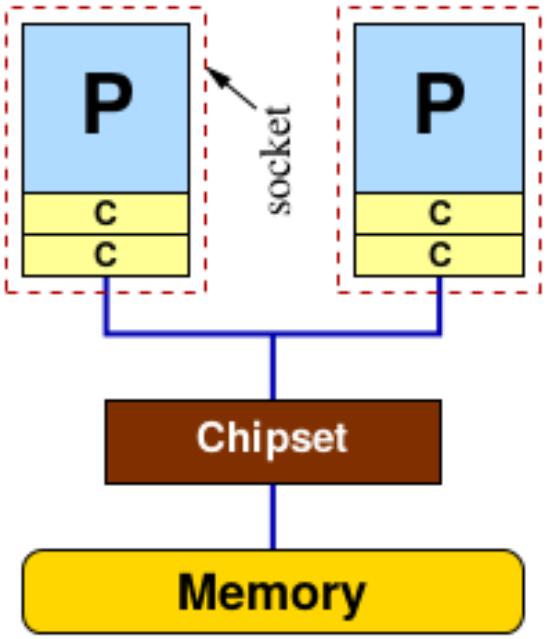
- A shared-memory parallel computer is a system in which a number of CPUs work on a common, shared physical address space
- Two main varieties of shared-memory systems:
  - **Uniform Memory Access (UMA)** systems feature a “flat” memory model: Memory bandwidth and latency are the same for all processors 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)

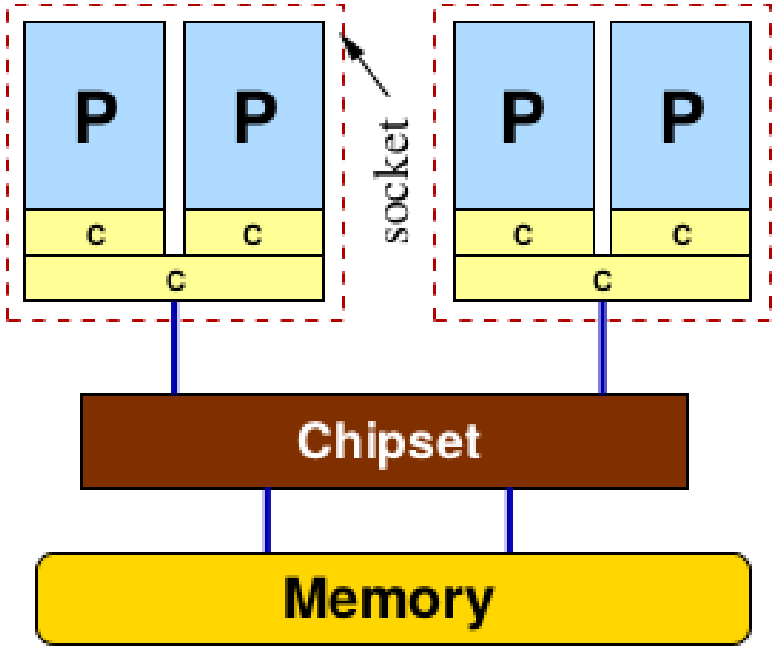
Uniform Memory Access (UMA)



- 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

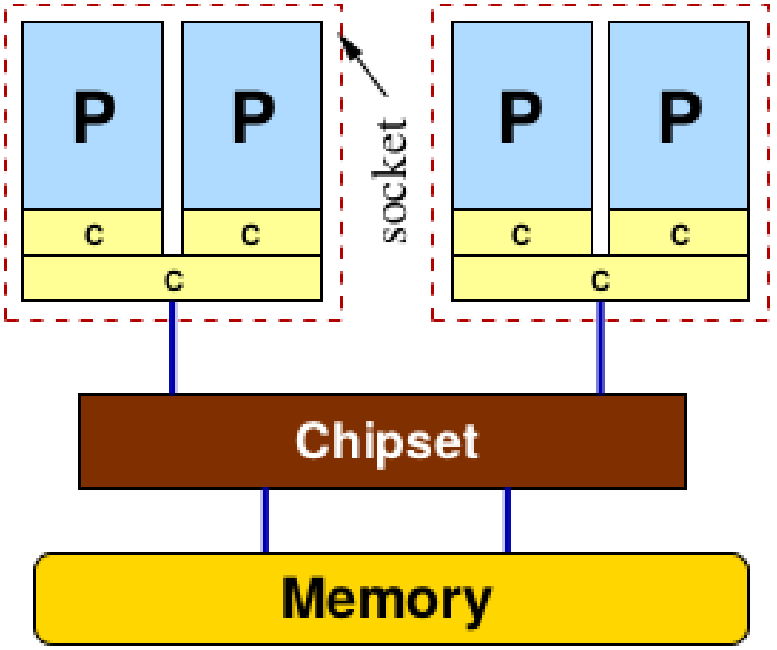
Uniform Memory Access (UMA)

- 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



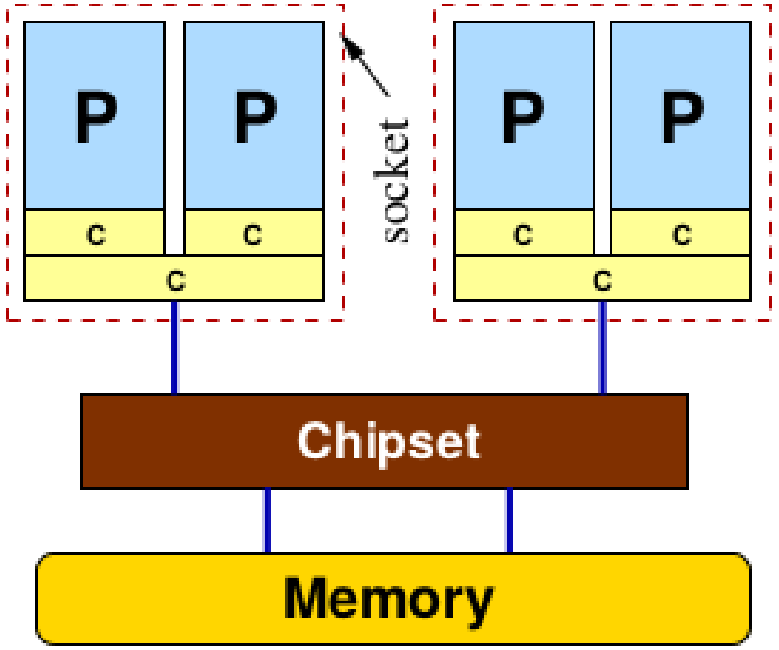
Uniform Memory Access (UMA)

- 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



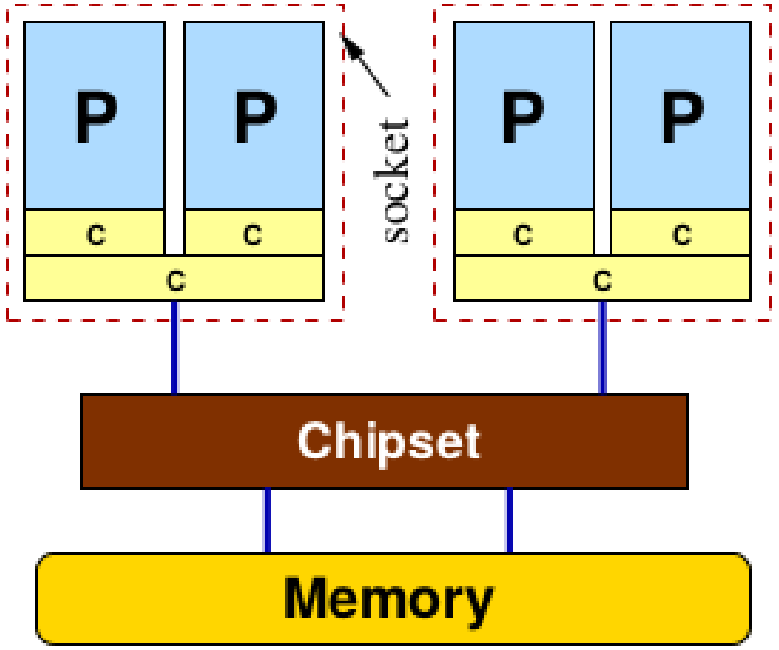
Uniform Memory Access (UMA)

- 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)

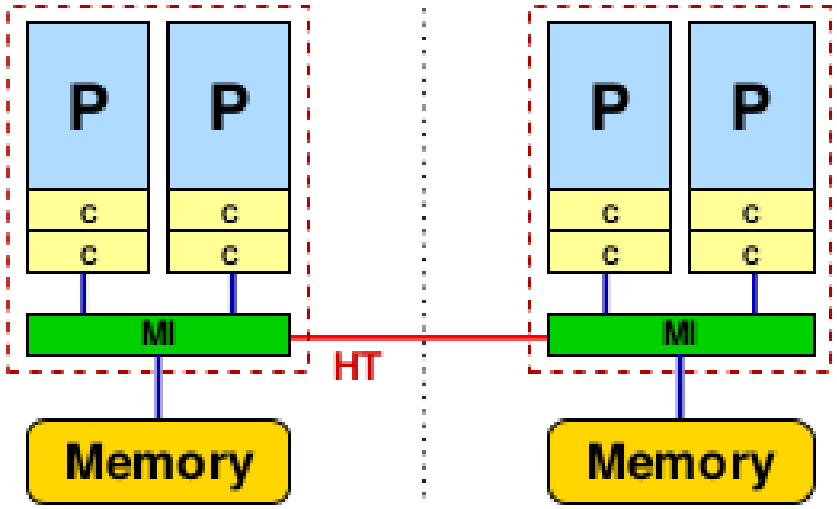
- The general problem of UMA systems is that bandwidth bottlenecks are bound to occur when the number of sockets, or FSBs, is larger than a certain limit
- In very simple designs, a common memory bus is used that can only transfer data to one CPU at a time
- In order to maintain scalability of memory 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)

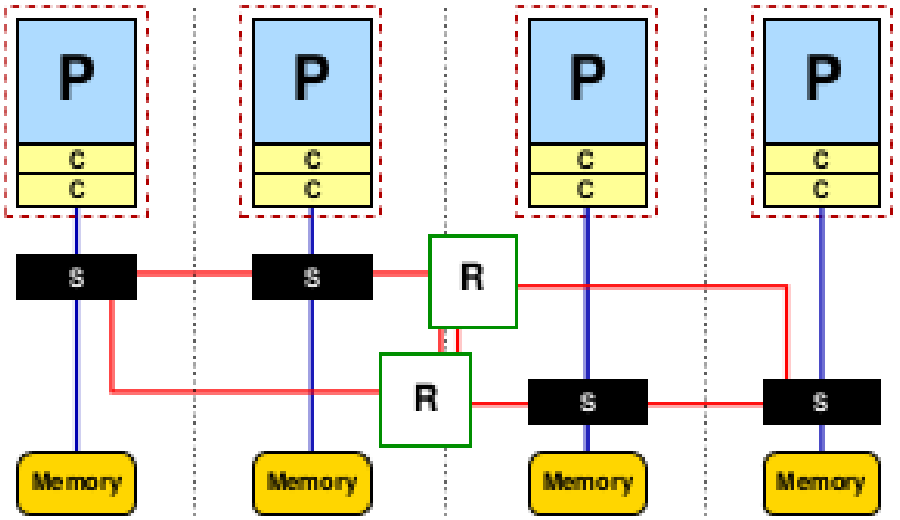
- **Example #1**: dual-core chips with separate caches and a common interface to local memory, are linked using a special high-speed connection called **HyperTransport (HT)**
- This system differs substantially from networked UMA designs in that the HT link can mediate direct coherent access from one 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)

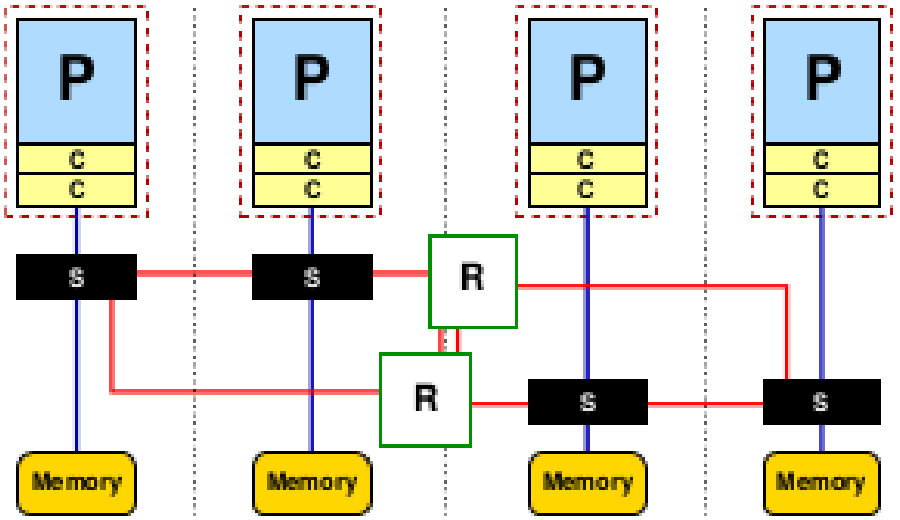
- **Example #2**: Each processor socket connects to a communication interface (S) that provides memory access as well as connectivity to the proprietary NUMALink (NL) network
- The NL network relies on routers (R) to switch connections for non-local access
- As with HT, the NL hardware allows for transparent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

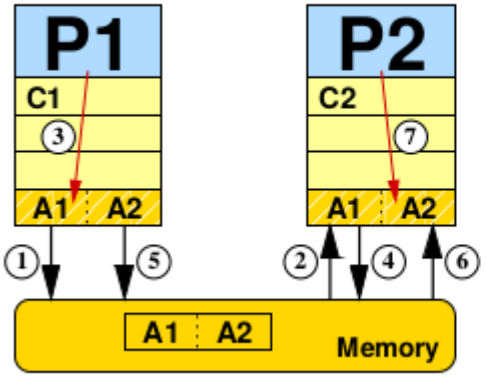


- **Figure:** ccNUMA system with routed NUMALink network and four locality domains

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



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

Shared-memory programming with OpenMP

Shared-memory programming with OpenMP