So if symmetric, i.e., uniform access, shared memory does not scale, we can try managing memory in other ways.

Each processor has a chunk of memory, but can also access memory of other processors, perhaps arranged in a tree.
A processor will have fast access to its closest chunk of memory, but slower access to more remote memory.
A processor will have fast access to its closest chunk of memory, but slower access to more remote memory. And different chunks of remote memory will have different access speeds.
Of course many other topologies have been tried: star, ring, hypercube, full interconnect, and so on.
Of course many other topologies have been tried: star, ring, hypercube, full interconnect, and so on.

This architecture evens out the access time to different chunks of memory a little.
This is *non-uniform memory access*
This is *non-uniform memory access*

NUMA shared memory scales much better than symmetric shared memory
This is *non-uniform memory access*

NUMA shared memory scales much better than symmetric shared memory

By scaling here we mean you can build larger machines with more processors cost effectively
This is *non-uniform memory access*

NUMA shared memory scales much better than symmetric shared memory

By scaling here we mean you can build larger machines with more processors cost effectively

But there is a downside: now programs and programmers (and the OS) have to worry about *data locality*: data a processor needs should be kept close to that processor
This is *non-uniform memory access*

NUMA shared memory scales much better than symmetric shared memory

By scaling here we mean you can build larger machines with more processors cost effectively

But there is a downside: now programs and programmers (and the OS) have to worry about *data locality*: data a processor needs should be kept close to that processor

It can make a huge difference to the speed of a program if the data is not where it should be
If data is close to the processor that is using it, it will go faster than if the data has to be fetched from further away.
Classifications
NUMA

If data is close to the processor that is using it, it will go faster than if the data has to be fetched from further away.

So you try to keep data near the relevant processor.
If data is close to the processor that is using it, it will go faster than if the data has to be fetched from further away.

So you try to keep data near the relevant processor.

Of course, if data needs to be used by several processors, this becomes a very difficult scheduling problem.
NUMA implementations stratify the memory in terms of “distance”
NUMA implementations stratify the memory in terms of “distance”

For example:

- direct connection on the local memory bus
- on the same node
- one hop away
- two hops away
- and so on
Classifications
NUMA

Though this is often simplified to: local, remote, and “far away”
Classifications

NUMA

Though this is often simplified to: local, remote, and “far away”

The OS or system libraries or the programmer will try their best to place data in appropriate memory to minimise latency, using their knowledge of the NUMA hierarchy and their knowledge of the program’s needs.
Though this is often simplified to: local, remote, and “far away”

The OS or system libraries or the programmer will try their best to place data in appropriate memory to minimise latency, using their knowledge of the NUMA hierarchy and their knowledge of the program’s needs

So the programmer ideally would have a good idea of the architecture of a machine before writing code for it
Classifications
NUMA

Though this is often simplified to: local, remote, and “far away”

The OS or system libraries or the programmer will try their best to place data in appropriate memory to minimise latency, using their knowledge of the NUMA hierarchy and their knowledge of the program’s needs.

So the programmer ideally would have a good idea of the architecture of a machine before writing code for it.

And the portability of a program is in question.
Classifications
NUMA

Though this is often simplified to: local, remote, and “far away”

The OS or system libraries or the programmer will try their best to place data in appropriate memory to minimise latency, using their knowledge of the NUMA hierarchy and their knowledge of the program’s needs

So the programmer ideally would have a good idea of the architecture of a machine before writing code for it

And the portability of a program is in question

This is still a matter of great research and development!
And, of course, there are hybrids where CPUs share some memory symmetrically and some memory NUMA
2010: Sgi made a 2048 core shared memory machine running a standard Linux distribution using an unmodified kernel (Altix UV)
2010: Sgi made a 2048 core shared memory machine running a standard Linux distribution using an unmodified kernel (Altix UV)

They claimed the hardware scaled to 262144 cores sharing 16TB of memory
Classifications
NUMA

2010: Sgi made a 2048 core shared memory machine running a standard Linux distribution using an unmodified kernel (Altix UV)

They claimed the hardware scaled to 262144 cores sharing 16TB of memory

It was a ccNUMA architecture: cache coherent NUMA
2010: Sgi made a 2048 core shared memory machine running a standard Linux distribution using an unmodified kernel (Altix UV)

They claimed the hardware scaled to 262144 cores sharing 16TB of memory

It was a ccNUMA architecture: *cache coherent NUMA*

Memory could be connected in a variety of ways, such as a fat tree or a torus, with a claim of a maximum $2\mu$sec maximum latency
Classifications
NUMA

2010: Sgi made a 2048 core shared memory machine running a standard Linux distribution using an unmodified kernel (Altix UV)

They claimed the hardware scaled to 262144 cores sharing 16TB of memory

It was a ccNUMA architecture: *cache coherent NUMA*

Memory could be connected in a variety of ways, such as a fat tree or a torus, with a claim of a maximum $2\mu$sec maximum latency

It wasn’t successful