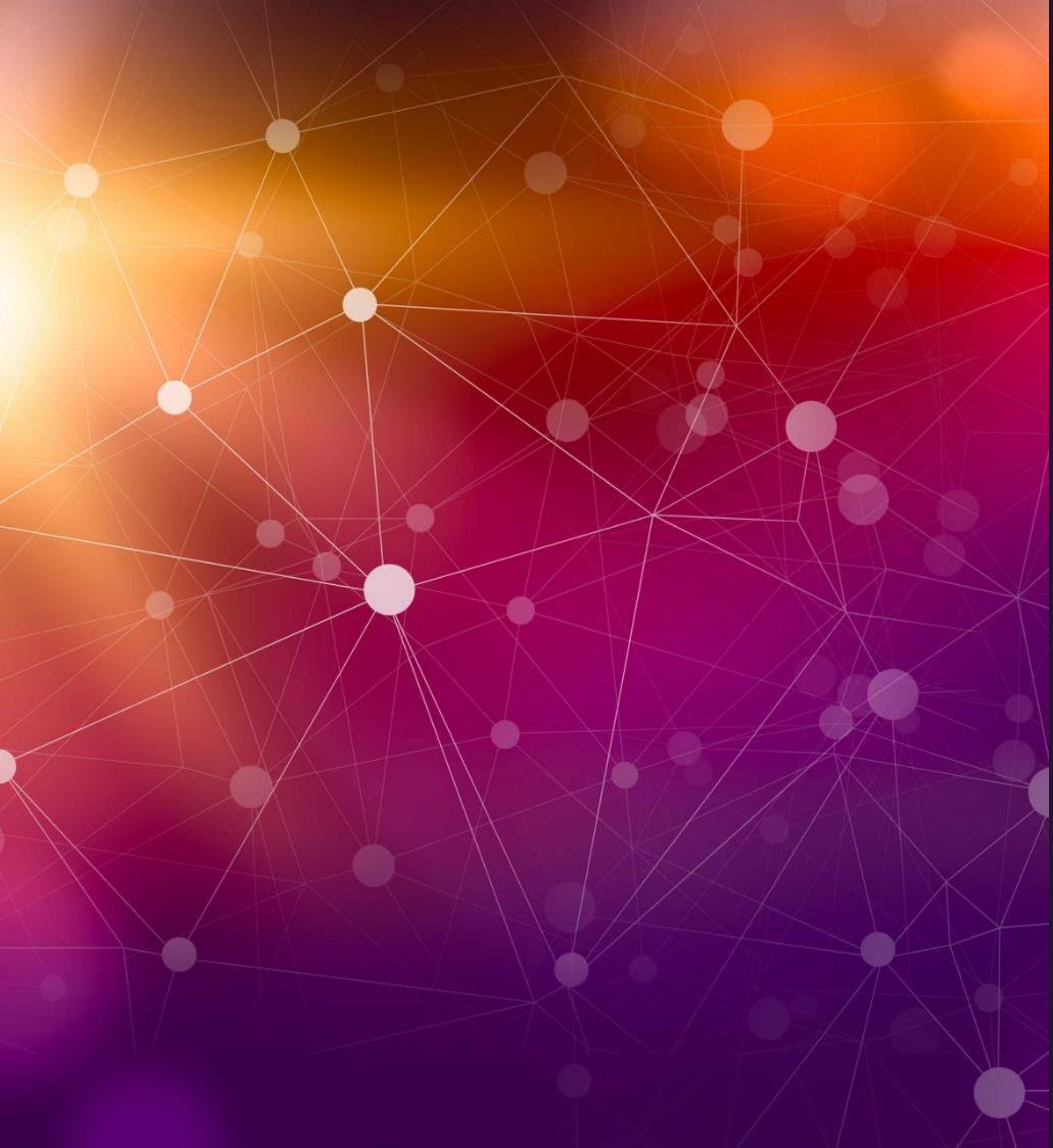
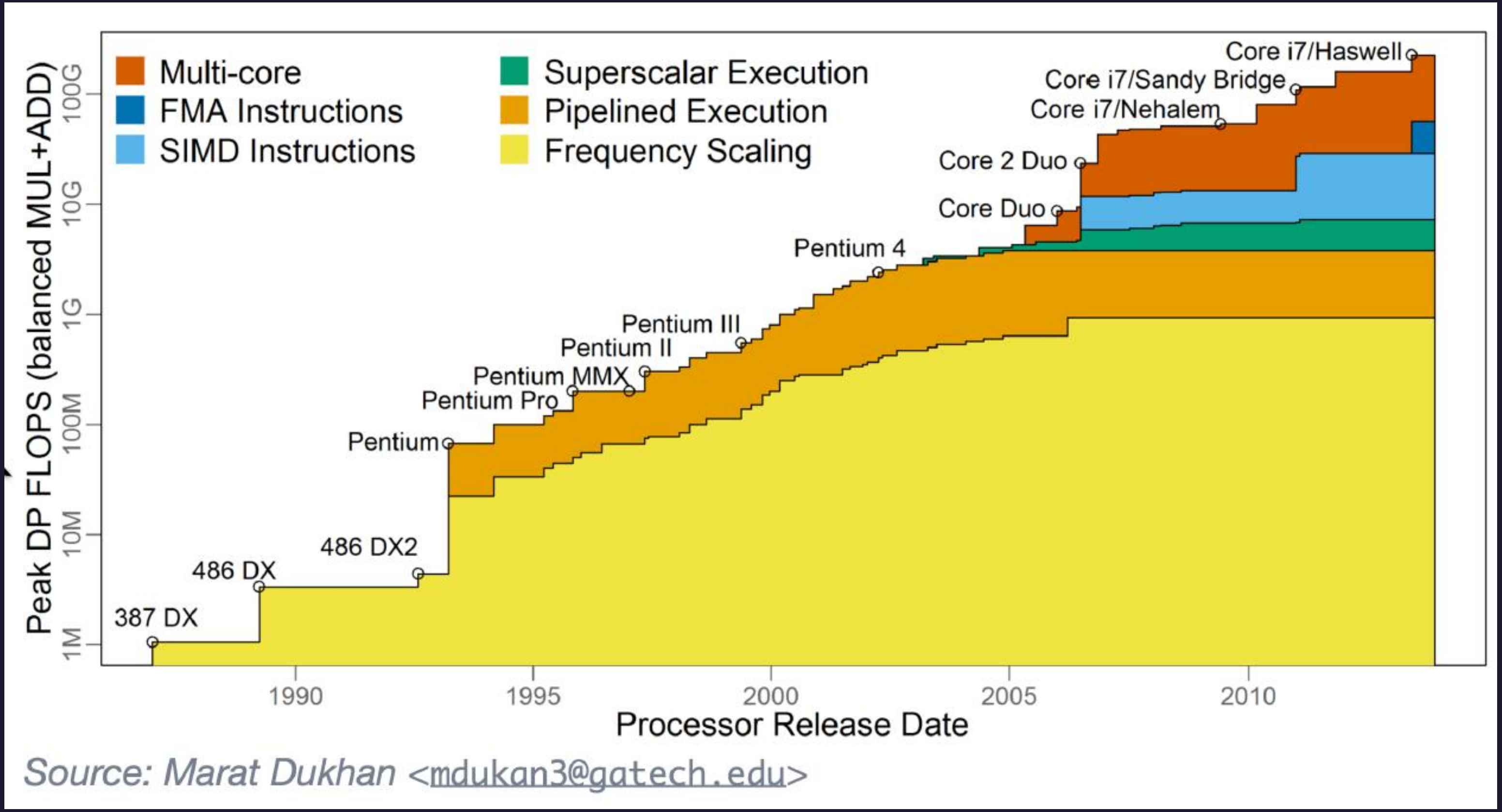


Parallel Computing with MPI

Finding meaning after Moore’s Law.

Why parallel computing?

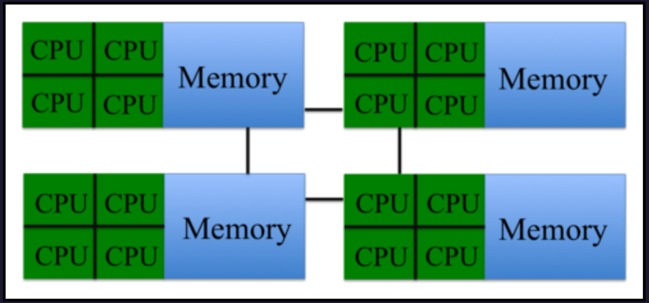
- Used to gain speed by bumping up clock rates (yellow).
- Turned out the increased heat emissions limited our ability to do that.
- Some attempts at more "parallel-like" computing (pipelining, SIMD, etc.)
- Truly parallel computing came with multi-processor, multi-core (dark orange)



Memory Organization

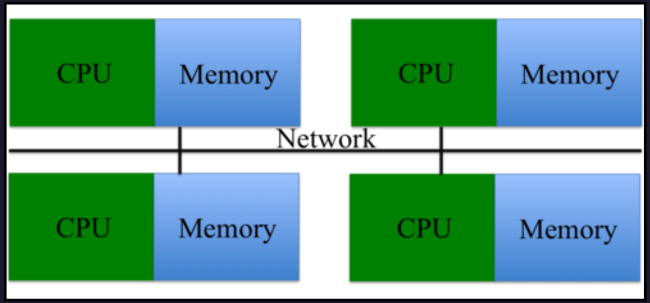
Shared Memory

- Different processors have physical access to the same memory (bus-based)
- Can share the same logical view of memory
- My array is your array
- This is what OpenMP is for (in C, C++ and Fortran)
- Julia has Threads



Distributed Memory

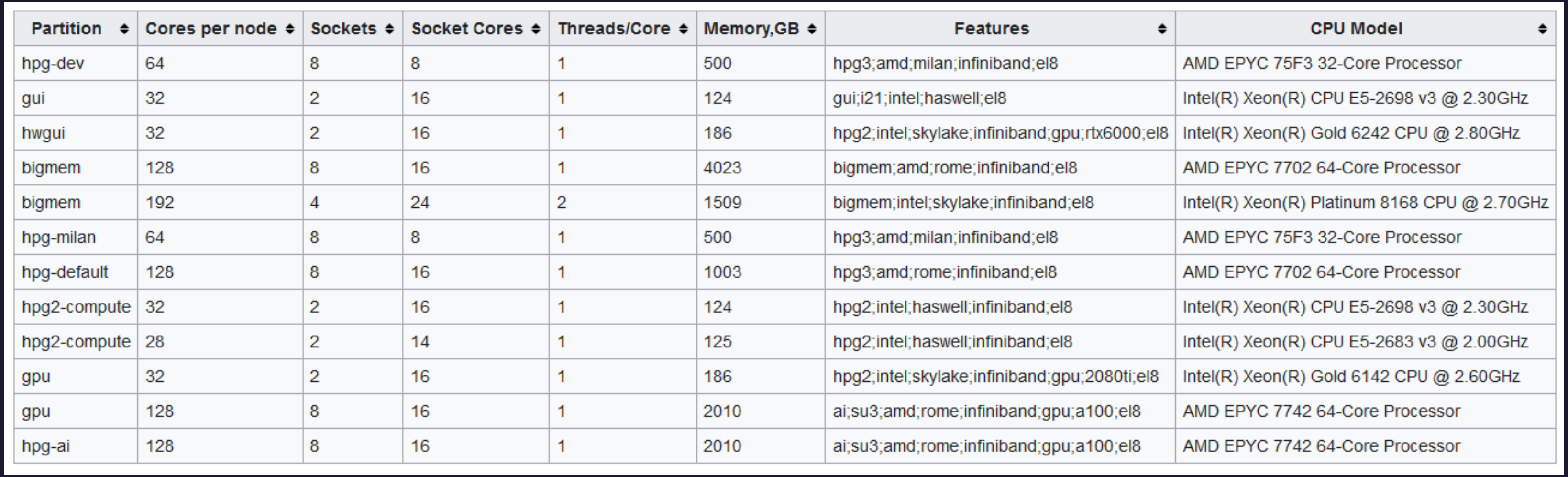
- Processors require using the network to access memory from other nodes
- Needs communications from other processes for data exchange
- I need your array results, please hand them over
- This is what MPI is for
- Julia has MPI.jl



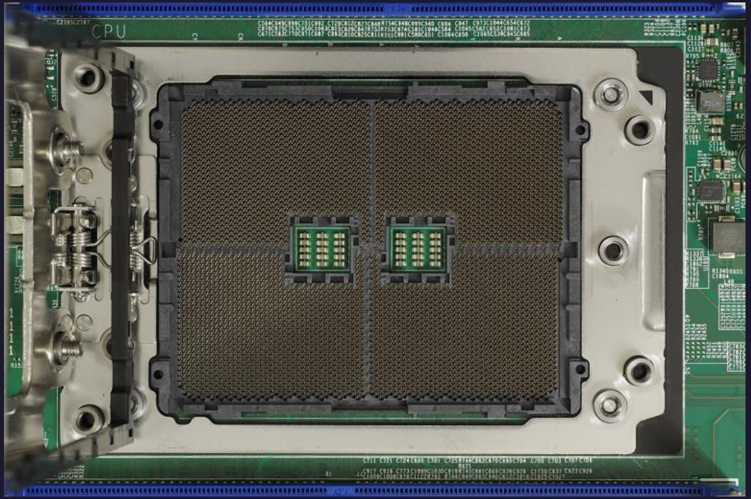
What does HiPerGator look like?

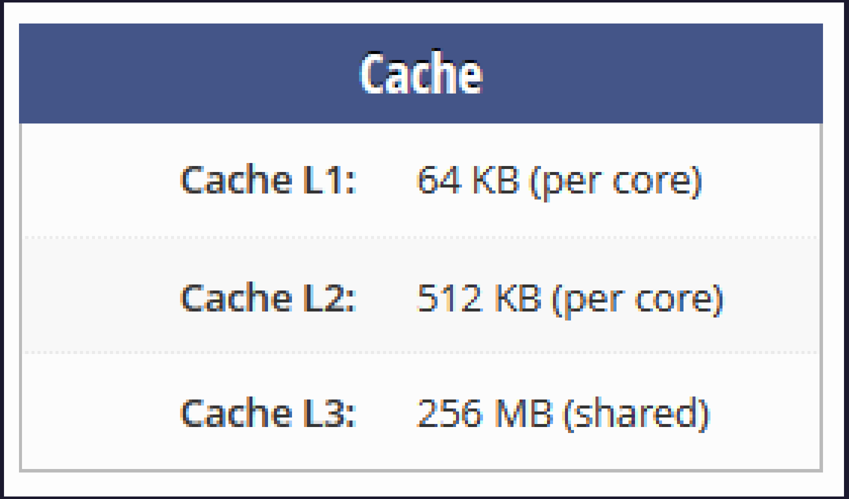
Let's look at hpg-dev:

- Each node is probably a single motherboard
- 8 Sockets for 2 CPUs
- 32 Cores per CPU
- SMT disabled (Simultaneous Multithreading or Hyperthreading)
- Each CPU has its own shared memory between cores (L3 cache)
- Each motherboard has 500GB of RAM, with the 2 CPUs connected by a memory bus
- They use SMP (Symmetric Multiprocessing) so they can communicate on RAM)



Partition	Cores per node	Sockets	Socket Cores	Threads/Core	Memory,GB	Features	CPU Model
hpg-dev	64	8	8	1	500	hpg3;amd;milan;infiniband;el8	AMD EPYC 75F3 32-Core Processor
gui	32	2	16	1	124	gui;i21;intel;haswell;el8	Intel(R) Xeon(R) CPU E5-2698 v3 @ 2.30GHz
hwgui	32	2	16	1	186	hpg2;intel;skylake;infiniband;gpu;rtx6000;el8	Intel(R) Xeon(R) Gold 6242 CPU @ 2.80GHz
bigmem	128	8	16	1	4023	bigmem;amd;rome;infiniband;el8	AMD EPYC 7702 64-Core Processor
bigmem	192	4	24	2	1509	bigmem;intel;skylake;infiniband;el8	Intel(R) Xeon(R) Platinum 8168 CPU @ 2.70GHz
hpg-milan	64	8	8	1	500	hpg3;amd;milan;infiniband;el8	AMD EPYC 75F3 32-Core Processor
hpg-default	128	8	16	1	1003	hpg3;amd;rome;infiniband;el8	AMD EPYC 7702 64-Core Processor
hpg2-compute	32	2	16	1	124	hpg2;intel;haswell;infiniband;el8	Intel(R) Xeon(R) CPU E5-2698 v3 @ 2.30GHz
hpg2-compute	28	2	14	1	125	hpg2;intel;haswell;infiniband;el8	Intel(R) Xeon(R) CPU E5-2683 v3 @ 2.00GHz
gpu	32	2	16	1	186	hpg2;intel;skylake;infiniband;gpu;2080ti;el8	Intel(R) Xeon(R) Gold 6142 CPU @ 2.60GHz
gpu	128	8	16	1	2010	ai;su3;amd;rome;infiniband;gpu;a100;el8	AMD EPYC 7742 64-Core Processor
hpg-ai	128	8	16	1	2010	ai;su3;amd;rome;infiniband;gpu;a100;el8	AMD EPYC 7742 64-Core Processor





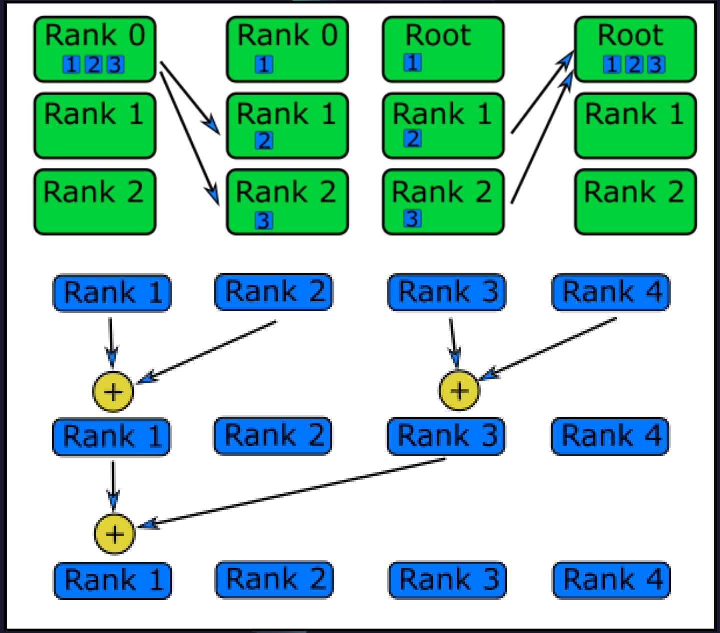
Cache	
Cache L1:	64 KB (per core)
Cache L2:	512 KB (per core)
Cache L3:	256 MB (shared)

MPI.jl

Letting you code like a C (not even C++) programmer again

Using MPI for Multiple Nodes and what it is?

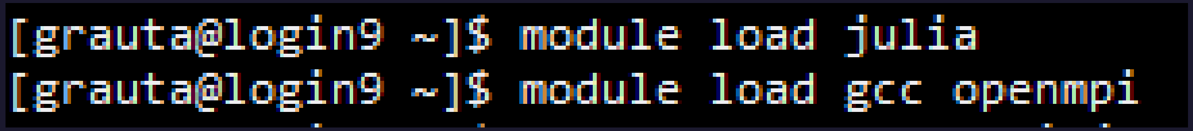
- MPI (Message Passing Interface) by itself is a standard
- This standard is implemented by different groups
- HiPerGator specifically has OpenMPI: https://www.open-mpi.org/
- Meant for allowing different processes with their own private memory spaces to communicate between each other
- Perfect for distributed memory systems since processors on one node can't see memory located on another node
- Implements message sending/receiving, data gathering/scattering and reduction algorithms



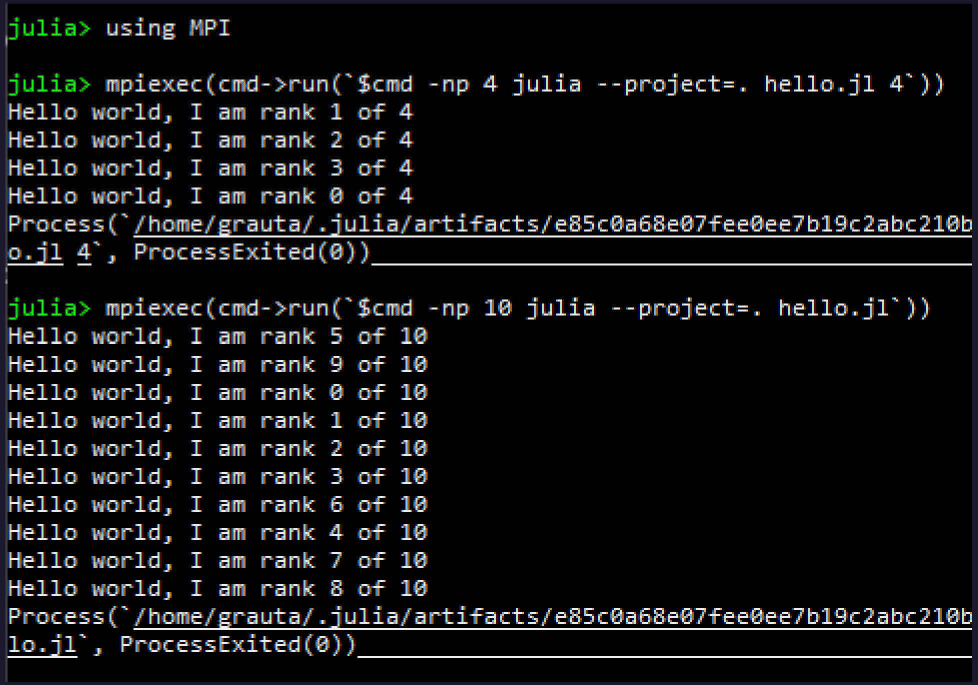
Basic MPI Example

```
using MPI

MPI.Init()

comm = MPI.COMM_WORLD

println("Hello world, I am rank $(MPI.Comm_rank(comm)) of $(MPI.Comm_size(comm))")

MPI.Barrier(comm)

MPI.Finalize() # Optional
```

https://juliaparallel.org/MPI.jl/latest/examples/01-hello/



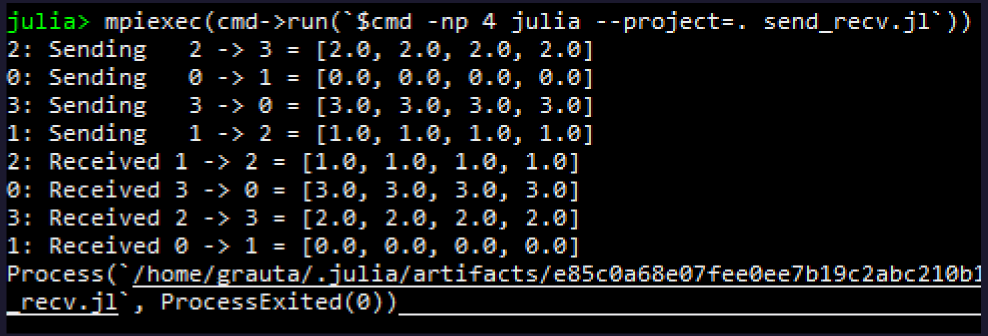
```
[grauta@login9 ~]$ module load julia
[grauta@login9 ~]$ module load gcc openmpi
```



```
julia> using MPI

julia> mpiexec(cmd->run(`$cmd -np 4 julia --project=. hello.jl 4`))
Hello world, I am rank 1 of 4
Hello world, I am rank 2 of 4
Hello world, I am rank 3 of 4
Hello world, I am rank 0 of 4
Process(`/home/grauta/.julia/artifacts/e85c0a68e07fee0ee7b19c2abc210b
o.jl 4`, ProcessExited(0))

julia> mpiexec(cmd->run(`$cmd -np 10 julia --project=. hello.jl`))
Hello world, I am rank 5 of 10
Hello world, I am rank 9 of 10
Hello world, I am rank 0 of 10
Hello world, I am rank 1 of 10
Hello world, I am rank 2 of 10
Hello world, I am rank 3 of 10
Hello world, I am rank 6 of 10
Hello world, I am rank 4 of 10
Hello world, I am rank 7 of 10
Hello world, I am rank 8 of 10
Process(`/home/grauta/.julia/artifacts/e85c0a68e07fee0ee7b19c2abc210b
lo.jl`, ProcessExited(0))
```

Sending and Receiving

- If we have different processes running, we need to communicate information between them
- We can use MPI.Send to send data out from a buffer to a specific rank
- We can use MPI.Recv to receive a message from a specific rank and store it in a buffer
- There is also MPI.Isend and MPI.Irecv for non-blocking communications (I can keep working while waiting for other communications)
- If we need a data from a non-blocking communication, we can MPI.Wait for it

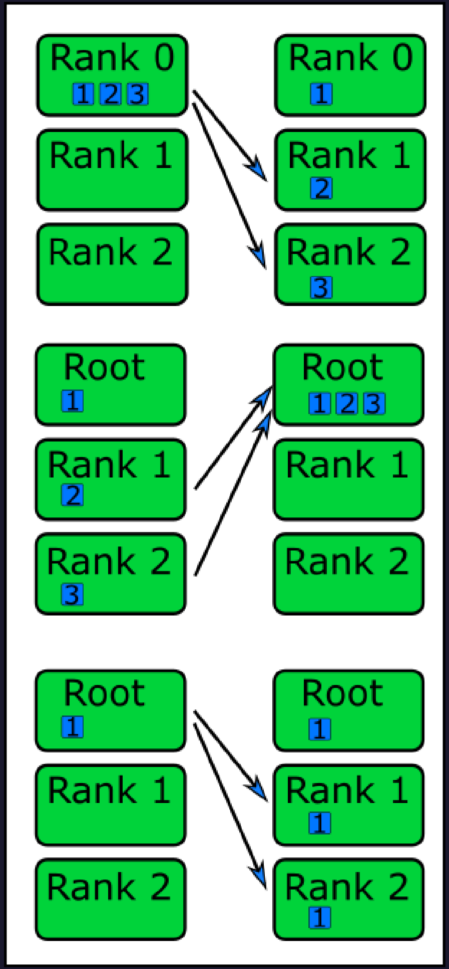


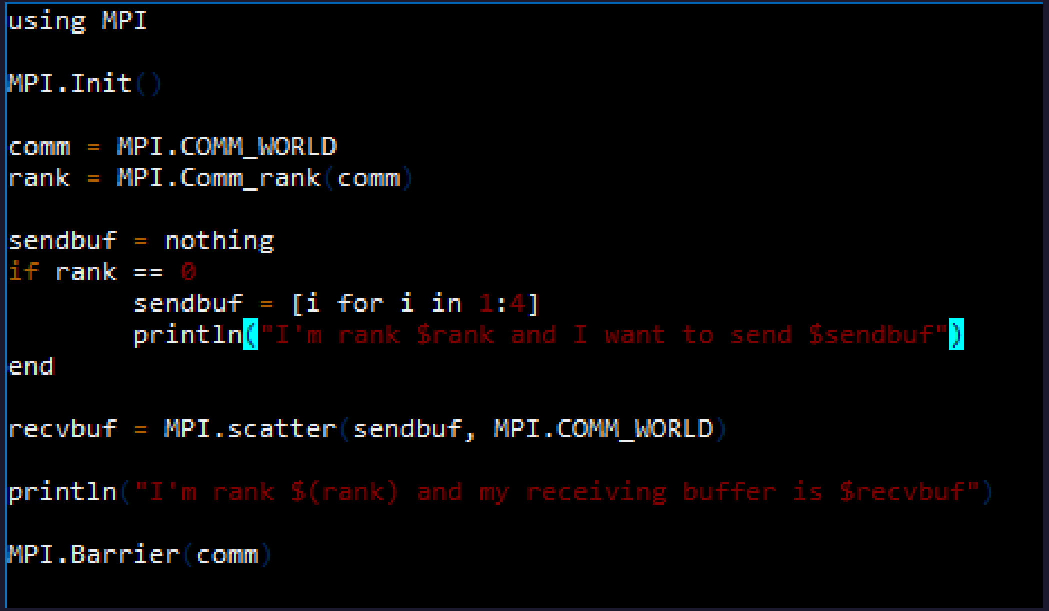
```
julia> mpiexec(cmd->run(`$cmd -np 4 julia --project=. send_recv.jl`))
2: Sending   2 -> 3 = [2.0, 2.0, 2.0, 2.0]
0: Sending   0 -> 1 = [0.0, 0.0, 0.0, 0.0]
3: Sending   3 -> 0 = [3.0, 3.0, 3.0, 3.0]
1: Sending   1 -> 2 = [1.0, 1.0, 1.0, 1.0]
2: Received 1 -> 2 = [1.0, 1.0, 1.0, 1.0]
0: Received 3 -> 0 = [3.0, 3.0, 3.0, 3.0]
3: Received 2 -> 3 = [2.0, 2.0, 2.0, 2.0]
1: Received 0 -> 1 = [0.0, 0.0, 0.0, 0.0]
Process(`/home/grauta/.julia/artifacts/e85c0a68e07fee0ee7b19c2abc210b1
_recv.jl`, ProcessExited(0))
```

https://juliaparallel.org/MPI.jl/latest/examples/04-sendrecv/

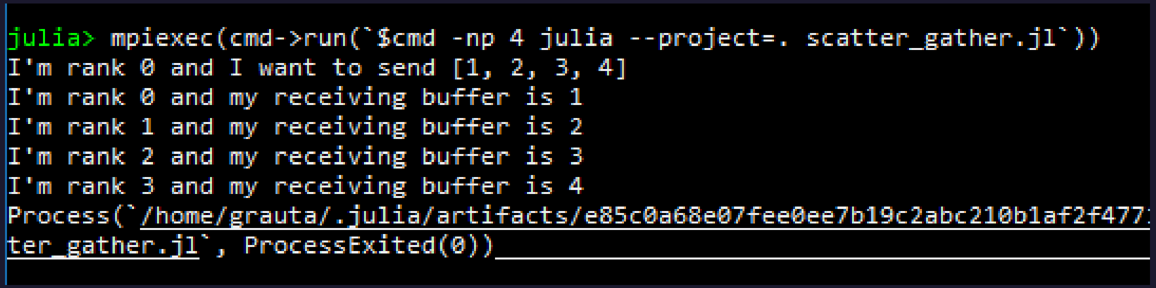
Scattering/Gathering/Broadcasting

- Can distribute data from one rank (usually 0) to many others
- Scatter splits data into chunks and sends an equal part to each rank
- Gather collects those chunks into one rank
- Broadcasting sends a single result into all ranks





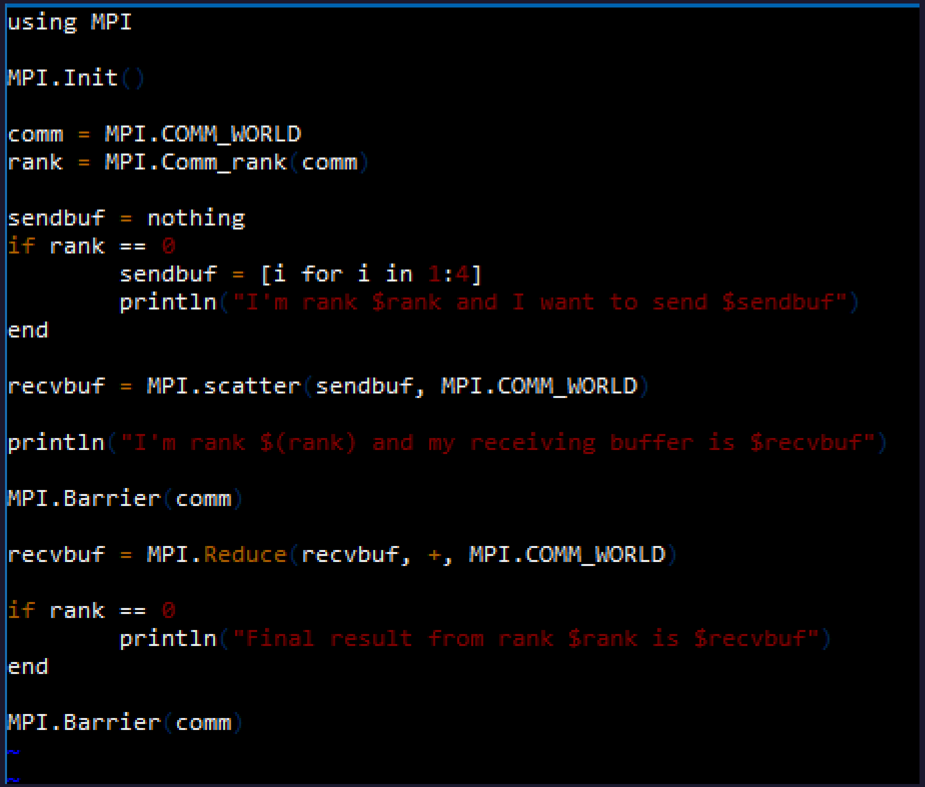
```
using MPI

MPI.Init()

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

sendbuf = nothing
if rank == 0
        sendbuf = [i for i in 1:4]
        println("I'm rank $rank and I want to send $sendbuf")
end

recvbuf = MPI.scatter(sendbuf, MPI.COMM_WORLD)

println("I'm rank $(rank) and my receiving buffer is $recvbuf")

MPI.Barrier(comm)
```



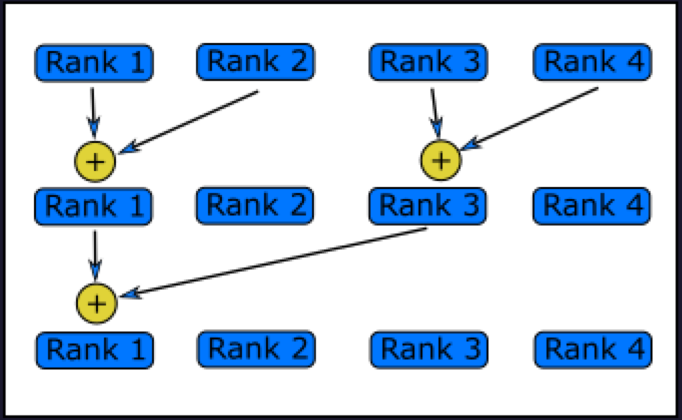
```
julia> mpiexec(cmd->run(`$cmd -np 4 julia --project=. scatter_gather.jl`))
I'm rank 0 and I want to send [1, 2, 3, 4]
I'm rank 0 and my receiving buffer is 1
I'm rank 1 and my receiving buffer is 2
I'm rank 2 and my receiving buffer is 3
I'm rank 3 and my receiving buffer is 4
Process(`/home/grauta/.julia/artifacts/e85c0a68e07fee0ee7b19c2abc210b1af2f477
ter_gather.jl`, ProcessExited(0))
```

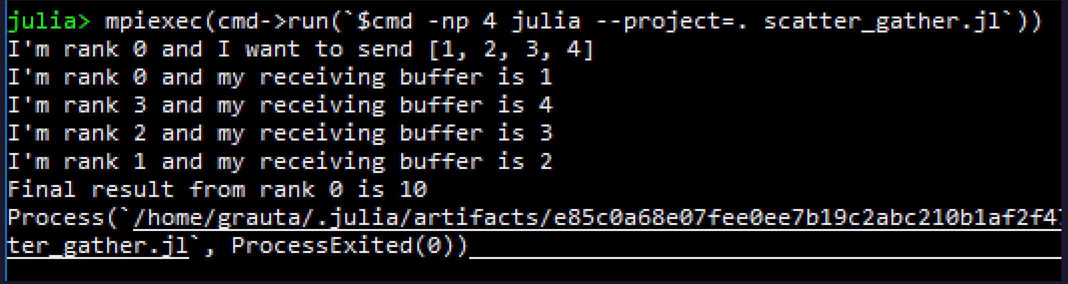
Reductions

- Parallel primitives that can efficiently “reduce” values
- Common reductions are addition, multiplication, min, max, etc.
- Gathers result into a single rank
- Can use MPI.Allreduce to Bcast result to all ranks



```julia
using MPI

MPI.Init()

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

sendbuf = nothing
if rank == 0
        sendbuf = [i for i in 1:4]
        println("I'm rank $rank and I want to send $sendbuf")
end

recvbuf = MPI.scatter(sendbuf, MPI.COMM_WORLD)

println("I'm rank $(rank) and my receiving buffer is $recvbuf")

MPI.Barrier(comm)

recvbuf = MPI.Reduce(recvbuf, +, MPI.COMM_WORLD)

if rank == 0
        println("Final result from rank $rank is $recvbuf")
end

MPI.Barrier(comm)
```





```
julia> mpiexec(cmd->run(`$cmd -np 4 julia --project=. scatter_gather.jl`))
I'm rank 0 and I want to send [1, 2, 3, 4]
I'm rank 0 and my receiving buffer is 1
I'm rank 3 and my receiving buffer is 4
I'm rank 2 and my receiving buffer is 3
I'm rank 1 and my receiving buffer is 2
Final result from rank 0 is 10
Process(`/home/grauta/.julia/artifacts/e85c0a68e07fee0ee7b19c2abc210b1af2f4
ter_gather.jl`, ProcessExited(0))
```

Many other cool things

- We’ve covered the basics but there is a lot more…
- Scans (Basically prefix sum stuff)
- Reductions as a Julia function
- Remote Memory Access (NVSHMEMRMA which is a real acronym)
- Communicating along a Graph Topology
- More communicators than just COMM_WORLD
- And probably a lot more!

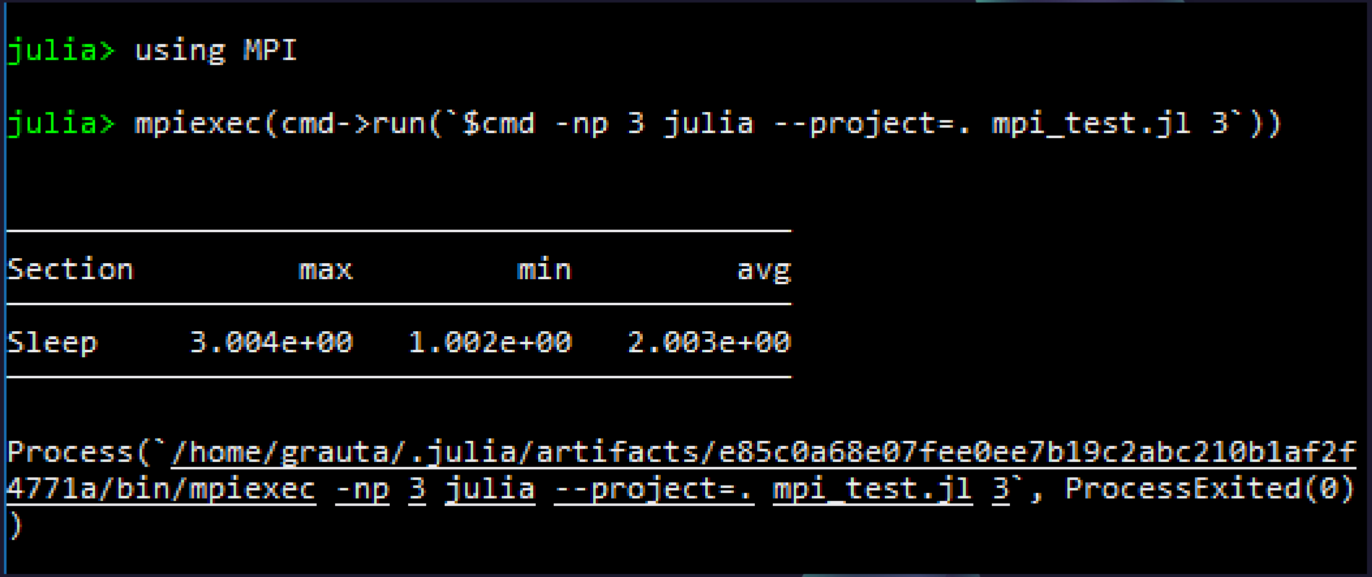
PartitionedArrays.jl

“This package provides distributed (a.k.a. partitioned) vectors and sparse matrices like the ones needed in distributed finite differences, finite volumes, or finite element computations.”

“The main objective of this package is to avoid to interface directly with MPI or MPI-based libraries when prototyping and debugging distributed parallel codes.”

PartitionedArrays Example

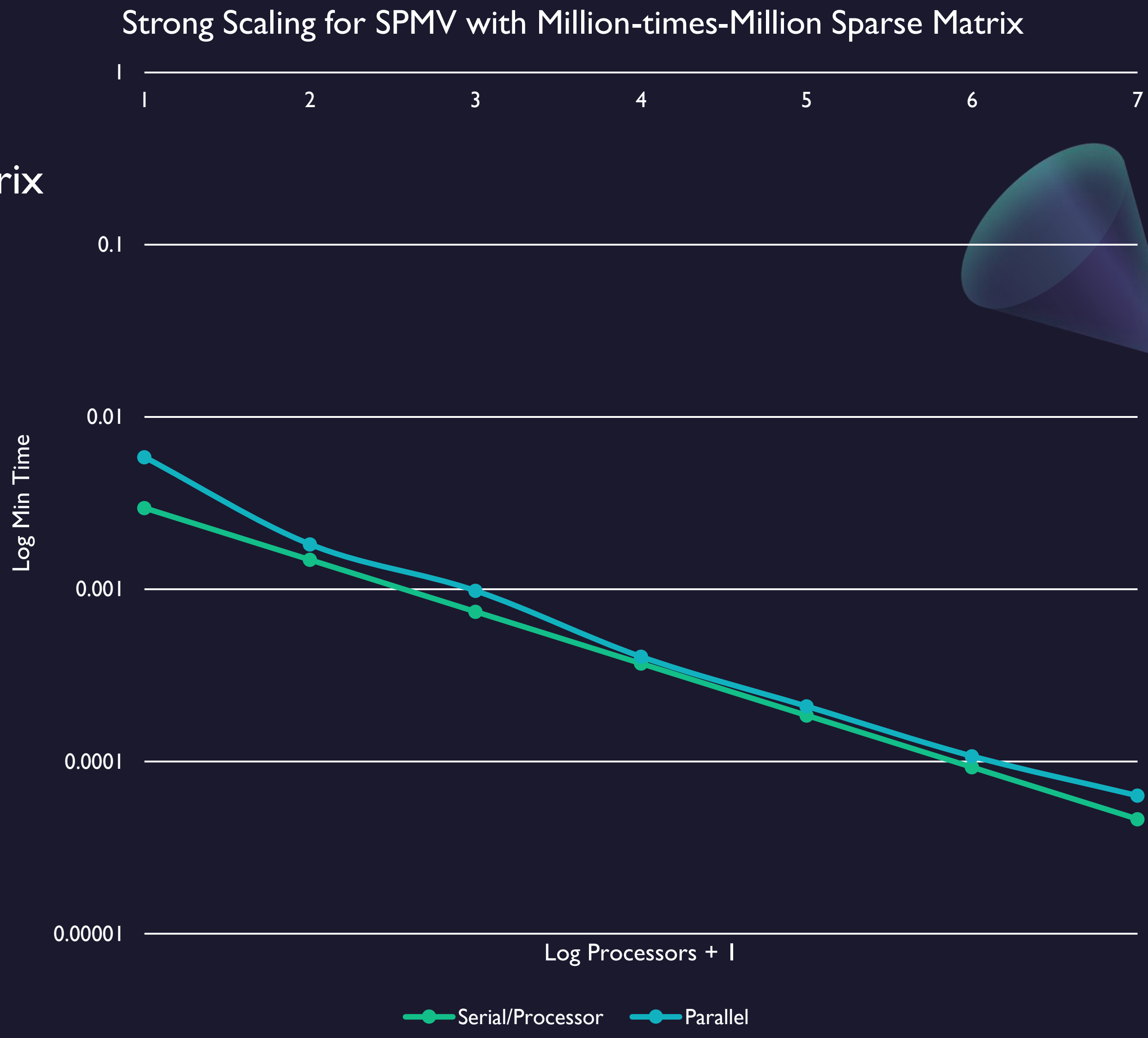
```
using PartitionedArrays

with_mpi() do distribute
    np = 3
    ranks = distribute(LinearIndices((np,)))
    t = PTimer(ranks)
    tic!(t)
    map(ranks) do rank
        sleep(rank)
    end
    toc!(t,"Sleep")
    display(t)
end
```



```
julia> using MPI

julia> mpiexec(cmd->run(`$cmd -np 3 julia --project=. mpi_test.jl 3`))

─────────────────────────────────────────────
Section            max          min          avg
─────────────────────────────────────────────
Sleep        3.004e+00    1.002e+00    2.003e+00
─────────────────────────────────────────────

Process(`/home/grauta/.julia/artifacts/e85c0a68e07fee0ee7b19c2abc210b1af2f
4771a/bin/mpiexec -np 3 julia --project=. mpi_test.jl 3`, ProcessExited(0)
)
```

Test with SPMV(SParse Matrix Vector multiplication)

- Created a million-by-million sparse tridiagonal matrix
- Distributed it among multiple MPI ranks
- Distribute the required vector information to the appropriate ranks
- Compute the product
- In PartitionedArrays.jl, easy as doing A*x



What the future holds...

Kernal Abstractions (Shared Memory)

MPI (Distributed Memory)

CUDA-aware MPI (https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/)

(https://docs.open-mpi.org/en/v5.0.x/tuning-apps/networking/cuda.html)