

# ***“Numerical Precision and the Errors you May not be Aware Of”***

**Dr Franklin T Willmore**  **USC University of Southern California**  
[frank.willmore@usc.edu](mailto:frank.willmore@usc.edu)

28 Mar 2025



**USC**

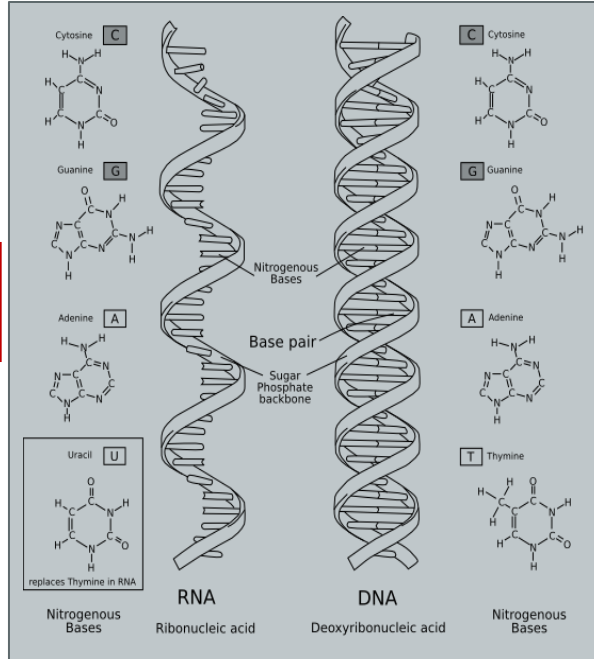
Center for Advanced Research Computing  
*Enabling scientific breakthroughs at scale*



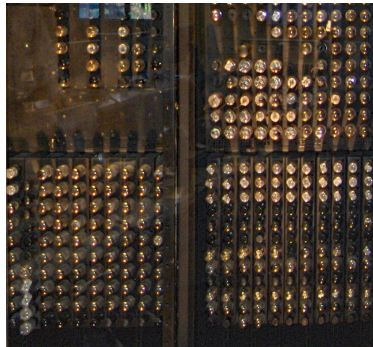
# WHAT IS A NUMBER?

- Real
- Imaginary
- Complex
- Integer
- Countable
- Random
- Finite/Infinite
- Positive/Negative
- Rational/Irrational/Transcendental
- Binary/Decimal/Octal/Hexadecimal
- Exact/Approximate
- Floating point

# WHAT IS A NUMBER? Storage and Representation



[https://en.wikipedia.org/wiki/Data\\_storage](https://en.wikipedia.org/wiki/Data_storage)



0x0	0	0	1	1	0	0	0	0
0x1	0	0	0	0	0	0	0	0
0x2	0	0	0	0	0	1	0	0
0x3	0	0	0	0	0	0	0	0
0x4	0	0	1	0	0	0	0	0
0x5	0	0	0	0	0	0	0	0
0x6	0	0	0	0	0	0	0	0
0x7	0	0	0	0	0	0	0	0
0x8	0	0	0	1	0	0	0	0
0x9	0	0	0	0	0	0	0	0
0xA	0	0	1	0	0	0	0	0
0xB	0	0	0	1	0	0	0	0
0xC	0	1	0	0	0	0	0	0
0xD	0	0	0	0	1	0	0	0
0xE	0	0	0	1	0	1	0	0
0xF	0	1	0	0	0	1	0	0

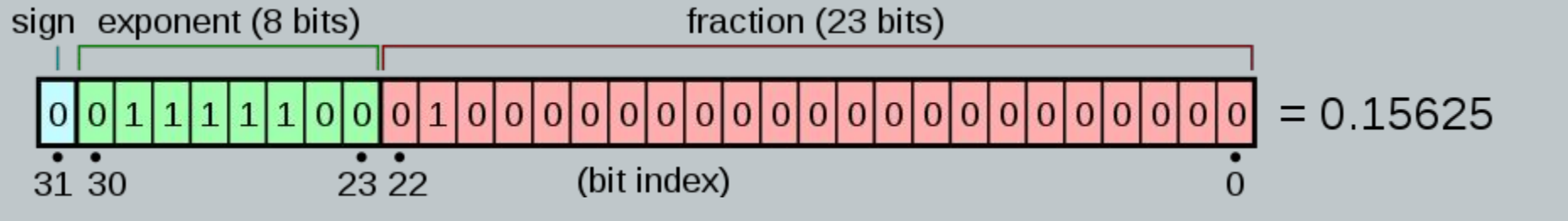


# WHAT IS A NUMBER? Storage and Representation

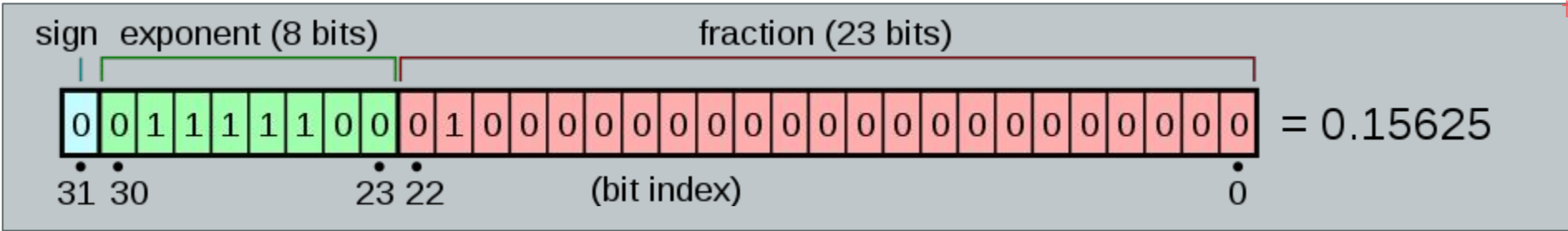


0x0	0	0	1	1	0	0	0	0
0x1	0	0	0	0	0	0	0	0
0x2	0	0	0	0	0	1	0	0
	0	0	0	0	0	0	0	0
	0	0	1	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
0x8	0	0	0	1	0	0	0	0
0x9	0	0	0	0	0	0	0	0
0xA	0	0	1	0	0	0	0	0
0xB	0	0	0	1	0	0	0	0
0xC	0	1	0	0	0	0	0	0
0xD	0	0	0	0	1	0	0	0
0xE	0	0	0	1	0	1	0	0
0xF	0	1	0	0	0	1	0	0

# WHAT IS A NUMBER? Machine representation of a floating point number

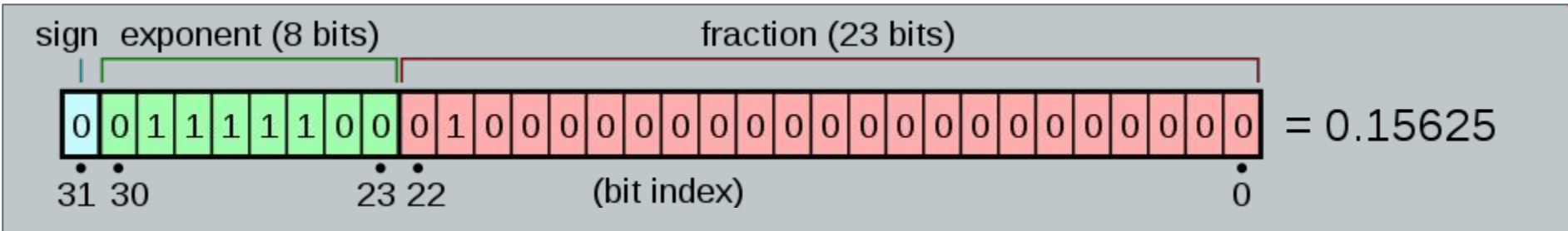


# WHAT IS A NUMBER? Machine representation of a floating point number



$-1^{\text{sign}}$   
= 1

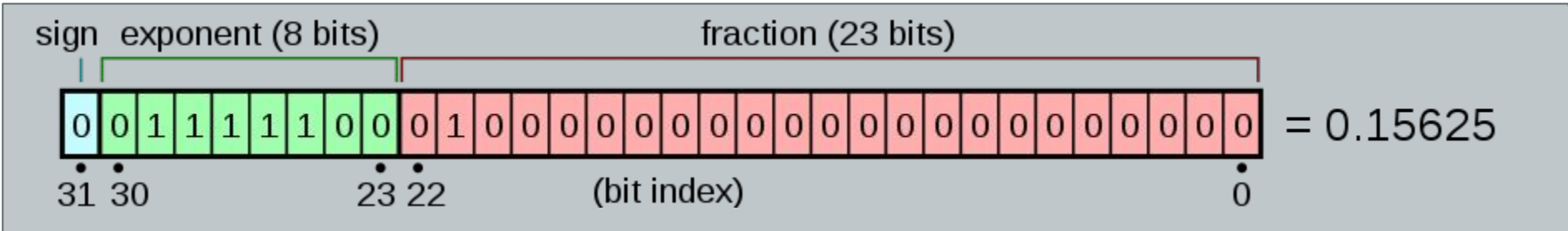
# WHAT IS A NUMBER? Machine representation of a floating point number



$$-1^{\text{sign}} = 1$$

The exponent is 'offset binary' format with a shift of -127, i.e. the exponent is  $(0 \cdot 128) + (1 \cdot 64) + (1 \cdot 32) + (1 \cdot 16) + (1 \cdot 8) + (1 \cdot 4) + (0 \cdot 2) + (0 \cdot 1) - 127$ , or **-3**. This means a factor of  $2^{-3}$ .

# WHAT IS A NUMBER? Machine representation of a floating point number



$$-1^{\text{sign}} = 1$$

The exponent is 'offset binary' format with a shift of -127, i.e. the exponent is  $(0 \cdot 128) + (1 \cdot 64) + (1 \cdot 32) + (1 \cdot 16) + (1 \cdot 8) + (1 \cdot 4) + (0 \cdot 2) + (0 \cdot 1) - 127$ , or **-3**. This means a factor of  **$2^{-3}$** .

Just like base 10 scientific notation, the exponent is shifted so that the first non-zero digit in the fraction is to the left of the decimal point. In binary, this value ( $2^0=1$ ) is implicit and not recorded. Since only the  $2^{-2}$  bit is set above, the fraction value here is  $2^0 + 2^{-2} = 1 + \frac{1}{4} = \mathbf{1.25}$ .



# WHAT IS A NUMBER? Machine representation of a floating point number



$$1 \times 2^{-3} \times 1.25 = 0.15625$$

## WHAT DOES IT MATTER? How does this affect my research?

- Machine precision is inherently limited
- This means rounding errors
- Rounding errors are a source of experimental error (in addition to modeling shortfalls) for computational science and can affect results
- Good research accounts for measurement error



## WHAT DOES IT MATTER? How does this affect my research?

- Machine precision is inherently limited
- This means rounding errors
- Rounding errors are a source of experimental error (in addition to modeling shortfalls) for computational science and can affect results
- Good research accounts for measurement error

### What can I do?

- Recognize the limitations of machine precision
- Make responsible and efficient choices regarding precision
  - AI/ML applications often use very low precision
  - GROMACS uses single precision by default
- Document choices
- Do better research

## EXAMPLE: SUMMATION AND ROUNDING

$$1 + x \rightarrow 1 + x; x > \delta$$

$$1 + x \rightarrow 1; x \leq \delta$$

$$y + x \rightarrow \max(x, y); x/y > \delta$$

$$y + x \rightarrow x + y; x/y \leq \delta$$

SO WHAT IS THE ACTUAL LIMIT OF PRECISION for float32?

$$\log_{10} 2^{24} = 24 * \log_{10} 2 = \sim 7.2247$$

Just over seven decimal places.



# EXERCISE: summing floats

```
#include <stdio.h>

int main()
{
    int ttf = 16777216;    // 2^24
    float delta = 1.0f/ttf; // 0.000000059604645
    float sum = 0.0f;

    // check inputs
    printf("Starting with sum=%1.18f, delta=%1.18f:\n\n", sum, delta);

    // should add to one
    for (int i=0; i<ttf; i++) sum += delta;
    printf("sum=%1.18f\n", sum);

    // what happens if we keep adding?
    for (int i=0; i<ttf; i++) sum += delta;
    printf("sum=%1.18f\n", sum);
}
```

# EXERCISE: summing floats

```
discovery$ salloc -p debug -n 4 -c 4
...
node$ module load gcc/13.3.0 openmpi/5.0.5
node$ git clone https://github.com/frankwillmore/machine-numbers
node$ cd machine-numbers/machine_numbers_and_MPI
node$ vi add_float.c
node$ make add_float
node$ ./add_float
Starting with sum=0.000000000000000000, delta=0.000000059604644775:

sum=1.000000000000000000
sum=1.000000000000000000
```

# EXERCISE: summing floats


- What happens if you increase the size of ttf?
  - Edit add\_float.c and recompile
  - What is the result?
- What happens if you decrease the size of ttf?
  - Edit add\_float.c and attempt to recompile
  - What is the result now?
  - Why has it changed?

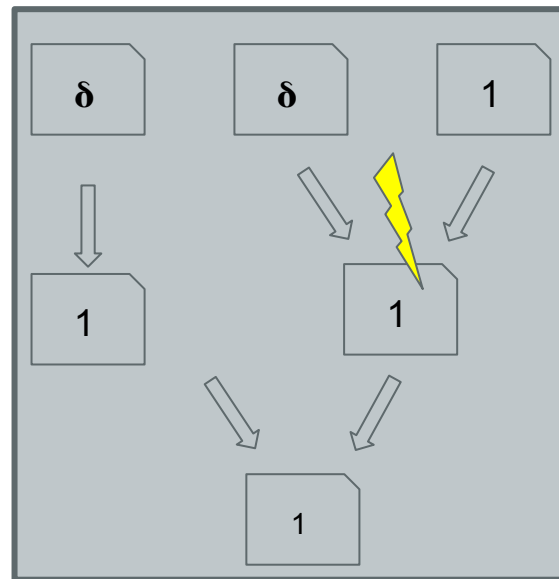
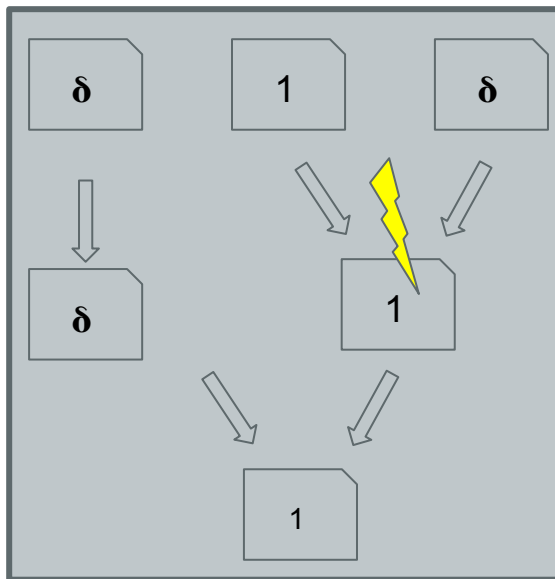
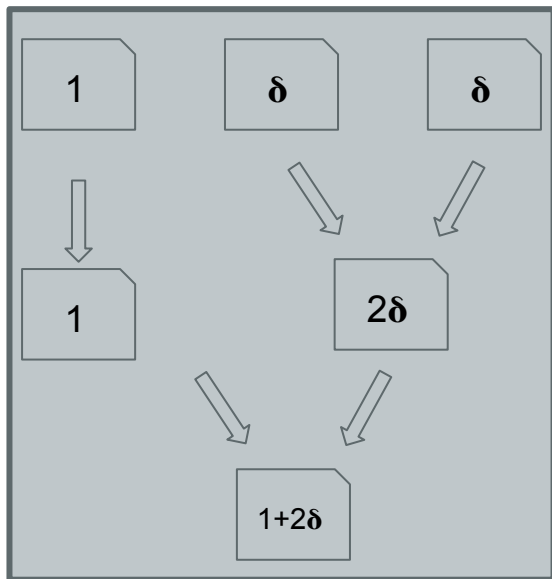
## EXAMPLE OF MPI REDUCTION

- MPI parallel processes run and complete in an indeterminate order
- Because of rounding, results of a reduction depend on the order in which operations are performed
- MPI operations can give results that vary between runs, depend on size of run, and even depend on which MPI you are using



## EXAMPLE OF MPI REDUCTION

- MPI parallel processes run and complete in an indeterminate order
- Because of rounding() results of a reduction depend on the order in which operations are performed
- MPI operations can give results that vary between runs and depend on size of run
- Different MPI implementations can also deliver different results





# EXERCISE: MPI reduction

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char* argv[])
{
    MPI_Init(NULL, NULL);

    float delta = 0.000000059604645;
    int mpi_rank, mpi_size;
    MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);

    for (int i=0; i<mpi_size; i++)
    {
        float global_sum, local_val;
        if (mpi_rank == i) local_val = 1.0f;
        else local_val = delta;
        MPI_Reduce(&local_val, &global_sum, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);
        if (!mpi_rank) printf("big val to rank %d, gives global sum = %1.10f\n", i, global_sum);
    }

    MPI_Finalize( );
}
```

# EXERCISE: MPI reduction

```
$ vi reduce.c
$ ml openmpi
$ make reduce
$ mpirun -n 3 ./reduce
rank 1/3 running.
rank 2/3 running.
rank 0/3 running.
big val to rank 0, gives global sum = 1.0000001192
big val to rank 1, gives global sum = 1.0000000000
big val to rank 2, gives global sum = 1.0000000000
```


# EXERCISE: MPI reduction

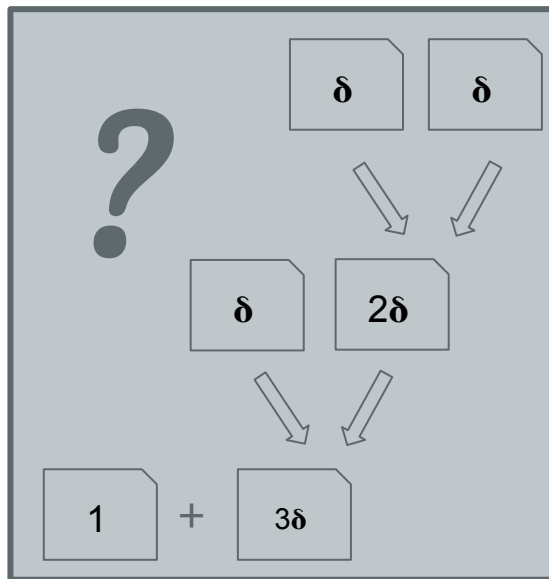
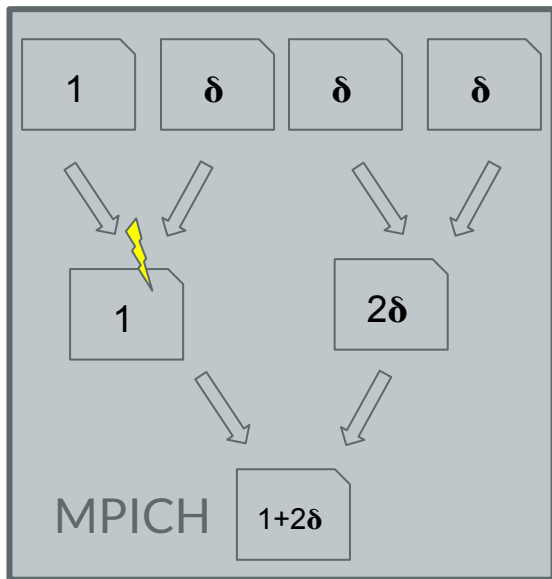
- What happens if you run it again?
  - Do the processes still return in the same order?
  - What is the result?
- What happens if you increase the number of processes, e.g. to 4?
- What happens if you load a different MPI (e.g. mpich) and build/run?
  - Do OpenMPI and MPICH give the same results?
  - Why or why not?

# EXERCISE: MPI reduction

```
$ make reduce
mpicc reduce.c -o reduce
$ mpirun -n 4 ./reduce
big val to rank 0, gives global sum = 1.0000002384
big val to rank 1, gives global sum = 1.0000000000
big val to rank 2, gives global sum = 1.0000002384
big val to rank 3, gives global sum = 1.0000000000
$ ml mpich
lmod is automatically replacing "openmpi/5.0.5" with "mpich/4.2.2".
$ make clean
rm -f a.out add_float reduce central_limit
$ make reduce
mpicc reduce.c -o reduce
$ mpirun -n 4 ./reduce
big val to rank 0, gives global sum = 1.0000001192
big val to rank 1, gives global sum = 1.0000001192
big val to rank 2, gives global sum = 1.0000001192
big val to rank 3, gives global sum = 1.0000001192
```

## EXAMPLE OF MPI REDUCTION

- MPI parallel processes run and complete in an indeterminate order
- Because of rounding() results of a reduction depend on the order in which operations are performed
- MPI operations can give results that vary between runs and depend on size of run
- Different MPI implementations can also deliver different results.



How does OpenMPI manage to return  $1+4\delta$ ?



## EXAMPLE OF CENTRAL LIMIT THEOREM

The average values of sets of samples of a *uniformly* distributed random variable will tend to be distributed *normally*:

$\mathcal{R} \in [0,1)$  ; for a *uniformly* distributed random variable  $\mathcal{R}$

$R_j \triangleq (1/N) \sum_i \mathcal{R}_i$ ; with average values  $R_j$  for sets of  $N$  samples of  $\mathcal{R}$

$p(R_j) \propto \exp\{-(R_j - \langle R_j \rangle)^2/2\}$  ; those average values  $R_j$  are distributed *normally*

## EXAMPLE OF CENTRAL LIMIT THEOREM

$\mathcal{R} \in [0,1)$  ; for a *uniformly* distributed random variable  $\mathcal{R}$

$R_j \triangleq (1/N) \sum_i \mathcal{R}_i$ ; with average values for a sets of N samples of  $\mathcal{R}$

$p(R_j) \propto \exp\{-(R_j - \langle R_j \rangle)^2/2\}$  ; those average values are distributed *normally*

The experiment:

- Using an MPI code with different random number seeds scattered to different MPI processes, sample values of  $R_j$  are generated and gathered.
- MPI parallel processes then run and complete, although in an indeterminate order
- Because of rounding, results of a reduction depend on the order in which operations are performed
- MPI operations can give results that vary between runs and depend on size of run

# EXERCISE: central limit theorem

```
#include <stdio.h>
#include <mpi.h>
#include <math.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
    MPI_Init(NULL, NULL);

    int elements_per_proc = 8;
    int initial_seed = 123456;

    int mpi_rank, mpi_size;
    MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);

    if (!mpi_rank) printf("running %d elements per task with %d tasks.\n", elements_per_proc, mpi_size);

    // create seeds
    int scatter_seeds[mpi_size];
    scatter_seeds[0] = initial_seed;
    if (!mpi_rank) for (int i=1; i<mpi_size; i++) scatter_seeds[i] = scatter_seeds[0] + i;

    // scatter the seeds
    int seed;
    MPI_Scatter(scatter_seeds, 1, MPI_INT, &seed, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

# EXERCISE: central limit theorem

```
// seed the RNG
srand(seed);

float sum = 0;
for (int i=0; i<elements_per_proc; i++)
{
    float sample = (float)rand() / (float)RAND_MAX;
    sum += sample;
}
sum /= elements_per_proc;

// gather and bin the results:
float results[mpi_size];
MPI_Gather(&sum, 1, MPI_FLOAT, results, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
```

# EXERCISE: central limit theorem

```
int bins[10];
for (int i=0; i<10; i++) bins[i] = 0;

if (!mpi_rank) for (int i=0; i<mpi_size; i++)
{
    printf("binning %f\n", results[i]);
    bins[(int) (results[i]*10)]++;
}

if (!mpi_rank) for (int i=0; i<10; i++)
{
    for (int j=0; j<bins[i]; j++) printf("X");
    printf("\n");
}

MPI_Finalize();
}
```

# EXERCISE: central limit theorem

```
node$ vi central_limit.c
node$ make central_limit
node$ mpirun -n 48 ./central_limit
running 8 elements per task with 48 tasks.
binning 0.488702
binning 0.511513
. . .

XX
XX
XXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXX
XX
X
```

# EXERCISE: central limit theorem

- What are the likely sources of error and uncertainty?
- What happens if you run it again for a different seed value?
- What happens if you increase the number of MPI processes?
- What happens if you use a different number of samples for each  $R_j$ ?
- What happens if we have very small values of  $\mathcal{R}$ ?

# EXERCISE: central limit theorem

- What are the likely sources of error and uncertainty?
- What happens if you run it again for a different seed value?
- What happens if you increase the number of MPI processes?
- What happens if you use a different number of samples for each  $R_j$ ?
- **What happens if we have very small values of  $\mathcal{R}$ ?**
  - Since our distribution is 0..1, there is rarely (1 in 4 million) a roundoff



## TAKEAWAY POINTS

- Greater precision isn't always 'better'
- Knowing when you can use a smaller type can often mean better performance
- Making good choices here depends on understanding the limits of machine numbers
- Operations can give results that vary between runs and depend on size of run
- 'Randomness' is not always an implicit source of error, it depends on how it's used.
- There is a new floating point standard in play (posits) which looks to gain traction
- Limits of machine precision are inherent, no matter the standard

# Wait, there's more!



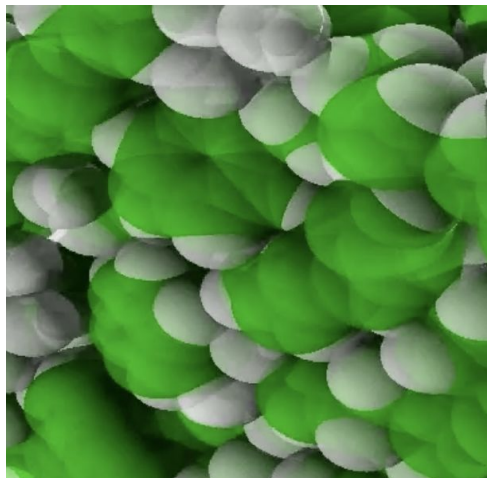
## BONUS TOPIC! CUDA/GPU application

- Q: When is it a good idea to use a GPU?
- A: For anything that looks (or can look) like a rendering problem.



## BONUS TOPIC! CUDA/GPU application

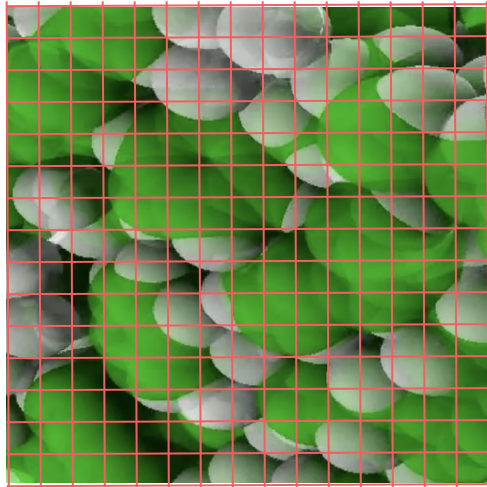
- Q: When is it a good idea to use a GPU?
- A: For anything that looks (or can look) like a rendering problem.



## BONUS TOPIC! CUDA/GPU application

- Q: When is it a good idea to use a GPU?
- A: For anything that looks (or can look) like a rendering problem.

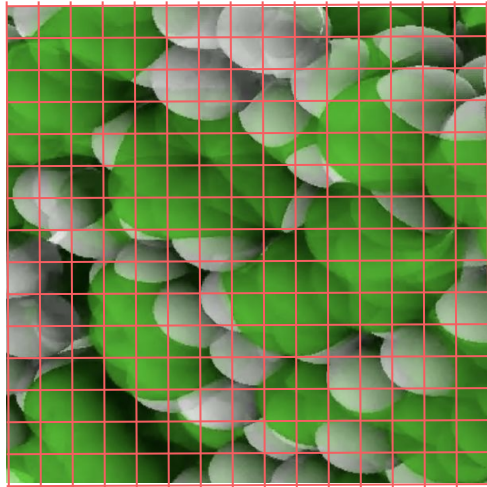
Domain decomposition



## BONUS TOPIC! CUDA/GPU application

- Q: When is it a good idea to use a GPU?
- A: For anything that looks (or can look) like a rendering problem.

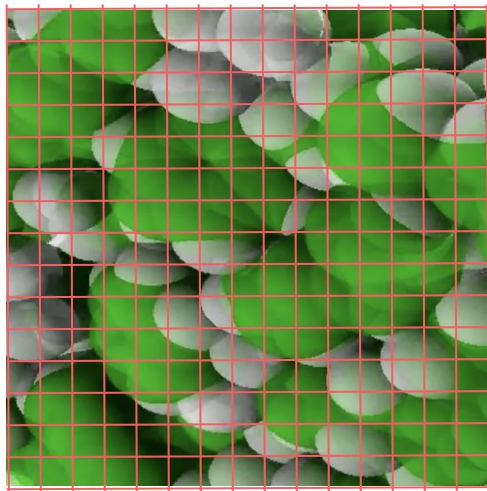
Domain decomposition



## BONUS TOPIC! CUDA/GPU application

- Q: When is it a good idea to use a GPU?
- A: For anything that looks (or can look) like a rendering problem.

Domain decomposition



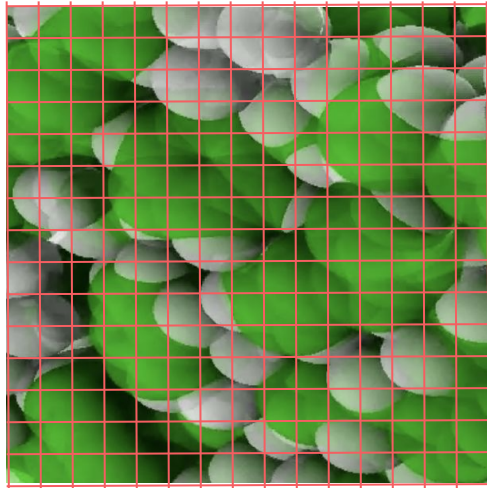
Chemical potential:

$$\mu_i = \left( \frac{\partial U}{\partial N_i} \right)_{S,V,N_{j \neq i}} \cdot \mu_i = -k_B T \ln \left( \frac{\mathbf{B}_i}{\rho_i \lambda^3} \right)$$

## BONUS TOPIC! CUDA/GPU application

- Q: When is it a good idea to use a GPU?
- A: For anything that looks (or can look) like a rendering problem.

Domain decomposition



**Chemical potential:**

$$\mu_i = \left( \frac{\partial U}{\partial N_i} \right)_{S,V,N_{j \neq i}} \cdot \mu_i = -k_B T \ln \left( \frac{\mathbf{B}_i}{\rho_i \lambda^3} \right)$$

**Widom insertion parameter:**

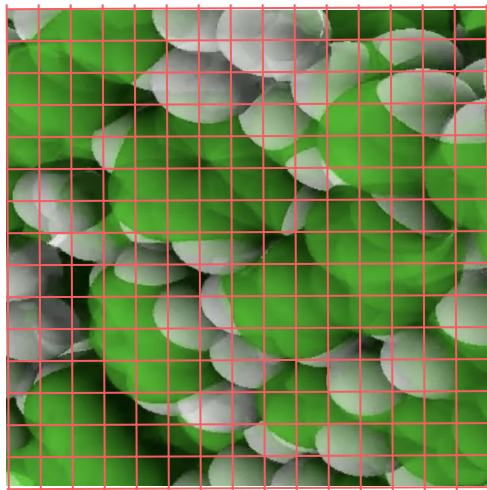
$$\mathbf{B}_i = \frac{\rho_i}{a_i} = \left\langle \exp \left( -\frac{\psi_i}{k_B T} \right) \right\rangle$$



## BONUS TOPIC! CUDA/GPU application

- Q: When is it a good idea to use a GPU?
- A: For anything that looks (or can look) like a rendering problem.

Domain decomposition



**Chemical potential:**

$$\mu_i = \left( \frac{\partial U}{\partial N_i} \right)_{S,V,N_{j \neq i}} \cdot \mu_i = -k_B T \ln \left( \frac{\mathbf{B}_i}{\rho_i \lambda^3} \right)$$

**Widom insertion parameter:**

$$\mathbf{B}_i = \frac{\rho_i}{a_i} = \left\langle \exp \left( -\frac{\psi_i}{k_B T} \right) \right\rangle$$

**Free volume index:**

$$\phi_i(x,y,z) = e^{-\beta \gamma_i}$$

# EXERCISE: free volume *via* GPU

```
discovery$ salloc --partition=gpu --gres=gpu:1 \  
               --cpus-per-task=4 --mem=32GB \  
               --time=1:00:00  
node$ module load vacuumms  
node$ cd ~/machine-numbers/GPU_fun  
node$ head PS.gfg  
33.637309    36.915309    35.467309    3.520530    0.073457  
36.121309    38.629309    34.794309    2.373410    0.028294  
34.127309    39.183309    34.275309    3.581180    0.066301  
33.785309    38.464309    35.581309    3.581180    0.066301  
39.593309    33.823309    36.362309    2.373410    0.028294  
33.439309    38.942309    33.477309    2.373410    0.028294  
35.546309    38.832309    33.826309    3.581180    0.066301  
35.729309    40.915309    33.360309    2.373410    0.028294  
34.054309    40.296309    34.378309    2.373410    0.028294  
34.658309    38.544309    36.220309    2.373410    0.028294  
node$ ./rungfg2fvi.sh &  
reading configuration  
calculating resolution = 256 for 612 potential  
using sigma = 0.000000 and epsilon = 1.000000  
node$
```

# EXERCISE: free volume *via* GPU

```
node$ nvidia-smi
Thu Mar 23 11:17:06 2023
```

+-----+									
NVIDIA-SMI 510.39.01      Driver Version: 510.39.01      CUDA Version: 11.6									
+-----+									
GPU  Name                Persistence-M  Bus-Id        Disp.A   Volatile Uncorr. ECC									
Fan  Temp  Perf  Pwr:Usage/Cap       Memory-Usage   GPU-Util  Compute M.									
+-----+-----+-----+									
0   Tesla V100-PCIE...    On        00000000:3B:00:0 Off                        Off									
N/A    47C    P0     215W / 250W     1365MiB / 16384MiB     100%      Default									
+-----+-----+-----+									
1   Tesla V100-PCIE...    On        00000000:D8:00:0 Off                        Off									
N/A    30C    P0     23W / 250W       4MiB / 16384MiB       0%      Default									
+-----+-----+-----+									
+-----+									
Processes:									
GPU   GI    CI          PID    Type    Process name                  GPU Memory									
ID  ID									
+-----+-----+-----+									
0   N/A  N/A         99430     C      gfg2fvi                      1361MiB									
+-----+									

```
username@gpuxxx$
```

## WRAP-UP

### Interesting GPU fact:

- The first GPUs didn't perform compliant arithmetic operations
- IEEE Compliant operations require:
  - padding 32 bit float to a 40 bit type
  - Performing arithmetic on 40 bit type
  - Rounding the result back to a 32 bit type
- Since they were only shading pixels, a little round-off error didn't matter much

### Additional resources:

- [https://en.wikipedia.org/wiki/IEEE\\_754](https://en.wikipedia.org/wiki/IEEE_754)
- [https://posithub.org/docs/posit\\_standard-2.pdf](https://posithub.org/docs/posit_standard-2.pdf)
- Willmore, F. T. (2016). Machine numbers and the IEEE 754 floating-point standard. In F. T. Willmore, E. Jankowski, & C. Colina (Eds.), Introduction to scientific and technical computing (pp. 19–36). CRC Press.

### Contact us!

- <https://www.carc.usc.edu/>
- [carc-support@usc.edu](mailto:carc-support@usc.edu)