Distributed Memory Implementation Strategies for the kinetic Monte Carlo Algorithm

António Esteves
Centro ALGORITMI

Alfredo Moura
Institute of Polymers and Composites

University of Minho | Braga, Portugal
School of Engineering

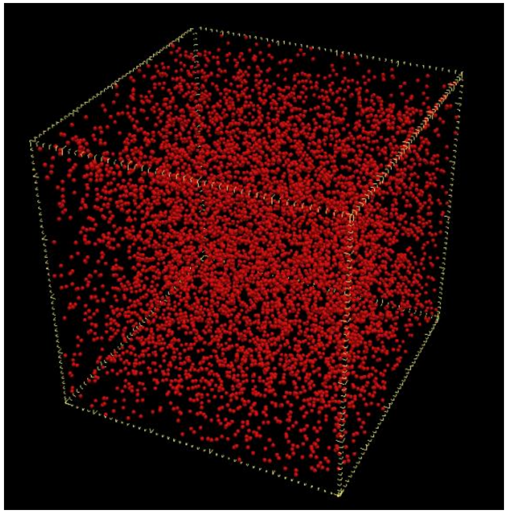
EuroMPI 2016 | Edinburgh | September 27th 2016

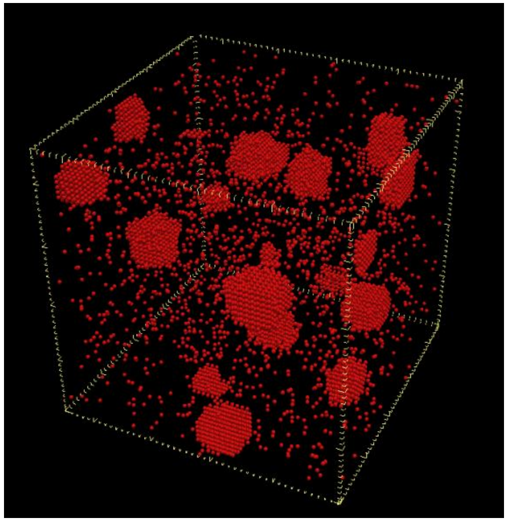
Summary

- Objectives
- Al_3Sc precipitation
- kinetic Monte Carlo method - kMC
- synchronous parallel kinetic Monte Carlo method - spkMC
- Parallelization of spkMC with MPI
- Results
- Conclusions

Objectives

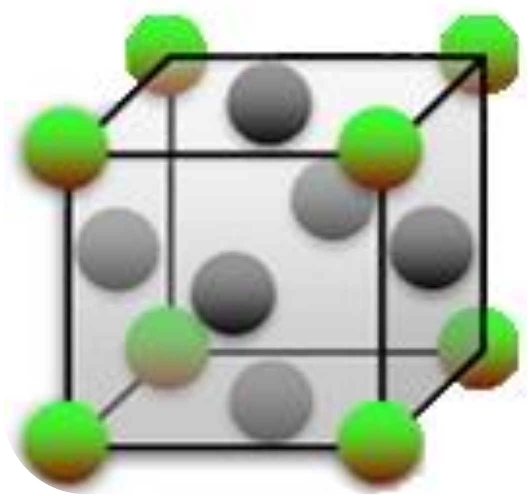
- Parallelize kMC with spkMC algorithm to speedup its execution
- Use distributed memory architecture and MPI
- Explore different computation vs. communication strategies
- Evaluate spkMC results by comparing them with kMC:
  - number of precipitates
  - dimension of precipitates
  - precipitates normalized by lattice sites, etc.
- Compare and assess spkMC implementations performance and scalability

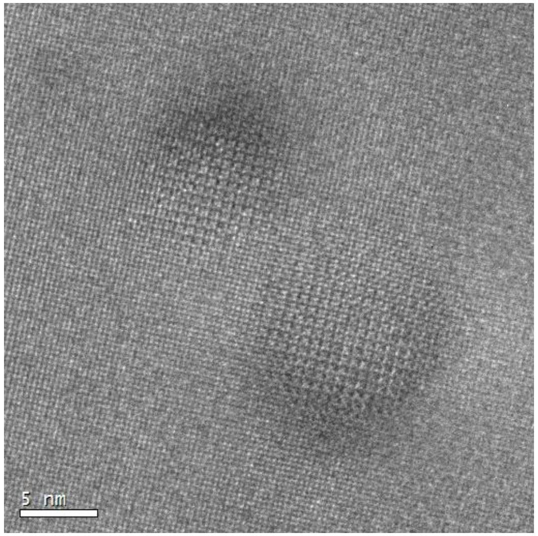




Al_3Sc precipitation

- **Precipitation** of Al_3Sc in Aluminum is the formation of clusters of atoms with an Al_3Sc structure
- **Precipitates** alter significantly the Al properties
- Precipitates have a **F**ace-**C**entered **C**ubic **crystalline** structure
- **Sc atoms** on the vertices and **Al atoms** on the faces
- Atoms **move** in the lattice structure by means of:
  - **vacancy diffusion**: jump to a neighbor vacant site
  - interstitial diffusion





TEM image of Al_3Sc precipitates

kinetic Monte Carlo method

- Used to model the **temporal evolution** of a system by stochastically exploring sequences of transitions

- Calculates the **transitions rates** for all trial **configurations** → $\boldsymbol{\Gamma_{i,j}}$

- Selects a new configuration $\boldsymbol{j}$ with a **probability proportional to** $\boldsymbol{\Gamma_{i,j}}$

kinetic Monte Carlo method

- $\boldsymbol{\Gamma_{i,V}}$ is called **vacancy exchange frequency**

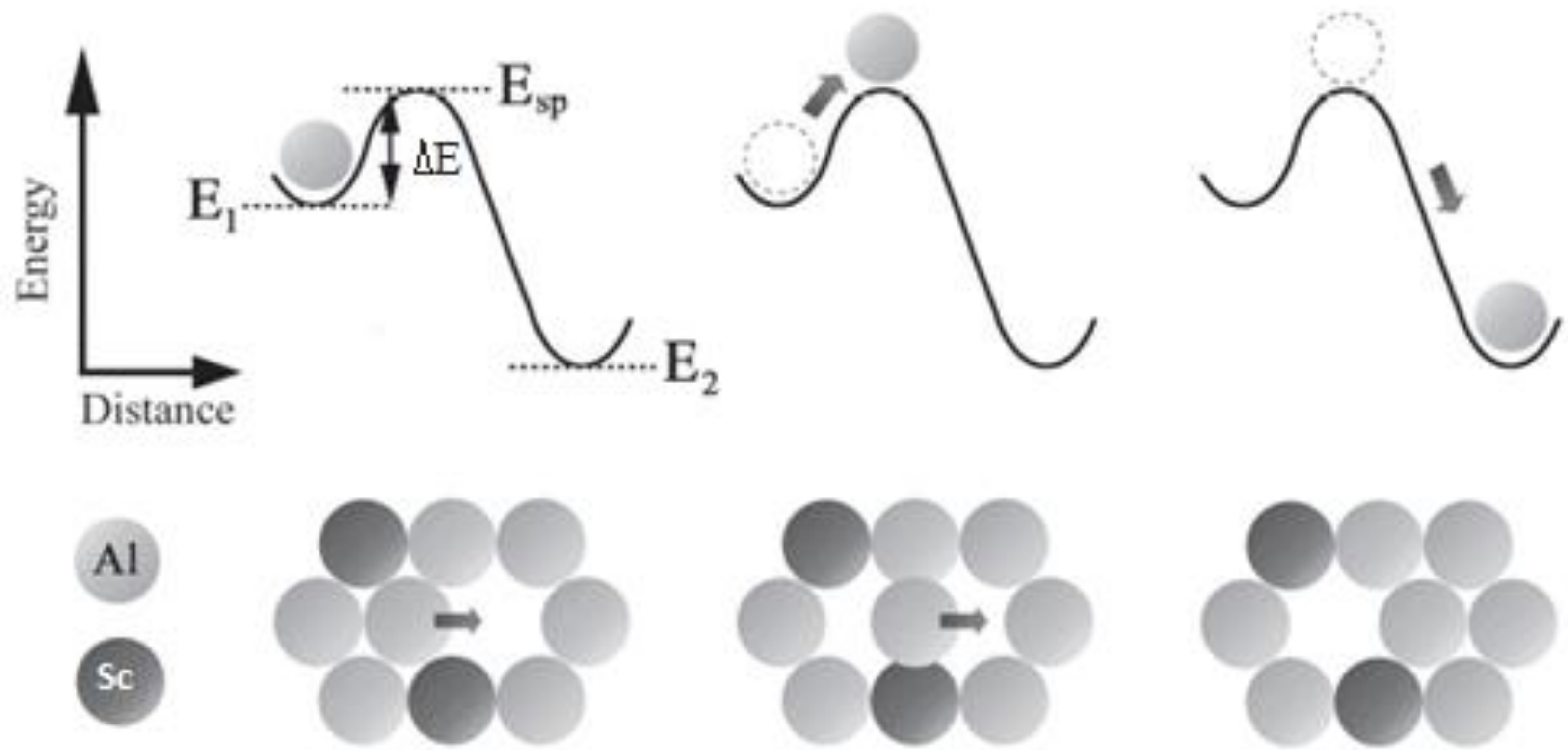
$$\Gamma_{i,V} = v_i * e^{-\frac{\Delta E_{i,V}}{k_B T}}$$

- $\boldsymbol{v_i}$ ≡ **attempt frequency** for an *Al*/*Sc* atom

- $\boldsymbol{\Delta E_{i,V}}$ ≡ **activation energy** required to move an *Al*/*Sc* atom into a vacancy

kinetic Monte Carlo method

- Moving an *Al* atom through **vacancy diffusion**



kinetic Monte Carlo method

- **Selecting a move**
  - A vacancy is surrounded by 12 first nearest neighbors
  - Calculate 12 jump frequencies → $\boldsymbol{\Gamma_1} \ldots \boldsymbol{\Gamma_{12}}$
  - Generate a **random number** between **0** and **1**
  - Select the **n**-th jump frequency that verifies the relation:

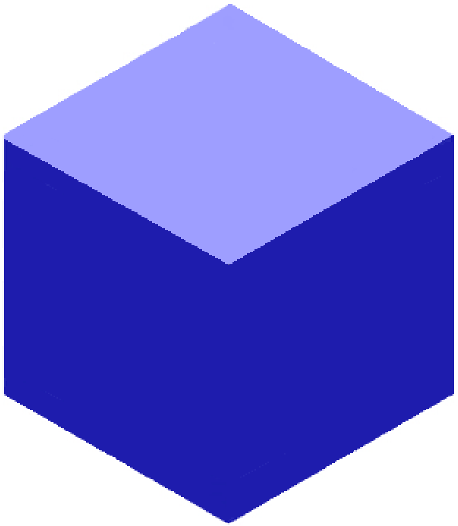
$$\sum_{i=1}^{n} \Gamma_i \leq \text{random number} \leq \sum_{i=1}^{n+1} \Gamma_i$$

synchronous parallel kinetic Monte Carlo method

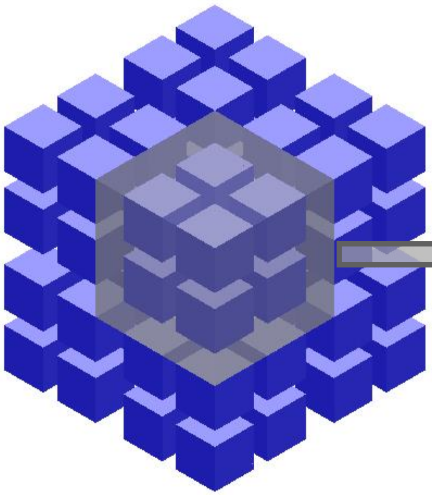
- Perform a **spatial decomposition** into subdomains
- Obtain the **accumulated frequency** for each subdomain → $$\Gamma_k = \sum_i^{n_k} \Gamma_{ik}$$
- Define the **maximum frequency** → $$\Gamma_{max} \geq \max_{k=1,\ldots,K} \{\Gamma_k\}$$
- Assign a **null event frequency** to the subdomains → $$\gamma_{0k} = \Gamma_{max} - \Gamma_k$$
- Define the spkMC **time step** increment → $$\delta t_p = -\frac{\ln \xi}{\Gamma_{max}}$$

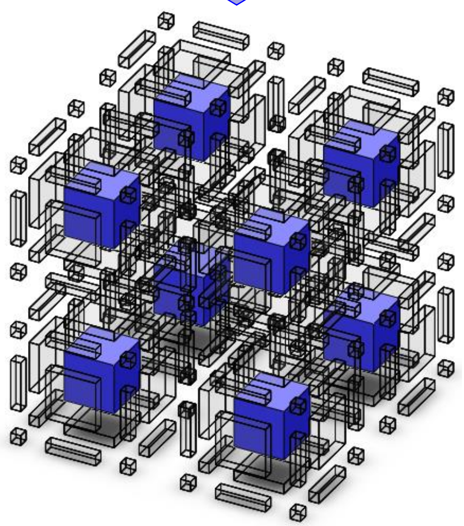
spkMC implementation - spatial domain decomposition

1 Lattice

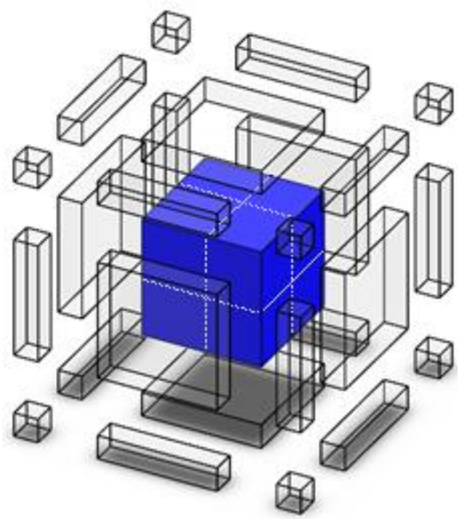


1 Lattice | P=8 Subdomains | 8x8 Sectors





A subdomain and its 26 boundary regions



spkMC implementation – major decisions

- At simulation start, a **vacancy** is placed on every **sector**
- Vacancies are allowed to **migrate** out of their original sector
- **Sprint** is a sequence of MCS, performed on a sector, without communication
- At end of sprint, each process communicates boundary moves to its neighbors
- **Boundary region** is as large as possible and we keep track of the changes that occurred on it during the sprints
- Avoid conflicts with a **checker board** scheme

spkMC implementation – inter-process communication

- **MPI point-to-point** communication:
  - Both processes participate actively
  - Complications:
    - when a process has multiples messages to be received
    - when the strong synchronization associated with blocking communication is unsuitable
    - possibility of deadlock
  - Alternatives:
    - MPI nonblocking point-to-point pattern
    - MPI-2/3 one-sided communication (or RMA)

spkMC implementation - one-sided communication

- Allows **remote memory access** (RMA) to a region called **window**
- **Access epoch**: RMA synchronization call on the window → 1+ RMA communication calls → RMA synchronization call
- **Advantage**: asynchronous, or at least, less synchronous
- **Transfer routines**: `MPI_Put`, `MPI_Get`, `MPI_Accumulate`
- **Synchronization mechanisms**:
  - fence, post-start-complete-wait, lock-unlock

spkMC implementation - one-sided communication

- **Fence** synchronization**:**
  - It is collective over the entire communicator associated with the window
  - It may result in communication overhead.
- **Post-start-complete-wait** (PSCW) synchronization:
  - Restricts synchronization to the minimum
  - Programmer selects the groups of processes that synchronize.
- **Lock-unlock** synchronization:
  - The origin process calls `MPI_Win_lock` to access the target window → calls transfer routines → calls `MPI_Win_unlock`
  - Emulates a shared memory model.

spkMC implementation - prototypes

- **PSCW** and **lock-unlock** prototypes:
  - Use RMA
  - A trimmed list of boundary and ghost moves is communicated to the adequate neighbor processes at the end of the sprints
  - Lock-unlock proved to be 3x faster than PSCW
  - Performance was not satisfactory → code profiling proved that a significant percentage of the execution time was spent in MPI barriers

spkMC implementation - prototypes

- **Send-receive** prototype**:**
  - Uses point-to-point communication
  - The communication pattern is simpler, regular and similar to the one used by SPPARKS/LAMMPS simulators
  - The communication runs in 3 steps: send and receive moves to/from nearest neighbor in +X (or -X) direction, in +Y (or -Y) direction, and in +Z (or -Z) direction
  - Due to checker board scheme we do not have to send and receive from both '+' and '-' directions in each step
  - Send (or receive) the variable **number** of moves and the **moves**
  - Initiate a non-blocking receive (MPI_Irecv) → do a blocking send (MPI_Send) → wait for receiving to complete (MPI_Wait).

spkMC implementation - prototypes

- **optimized send-receive** prototype:
  - The tasks done during each MCS were optimized, mainly to simplify the analysis of the vacancy moves
  - The data structures were simplified.

spkMC implementation – *opt-send-receive* algorithm

Read simulation, lattice, energy, and parallelization parameters
if (this is master process) **then**
    Send and extended subdomain to all processes
Receive the extended subdomain from master process
Compute the 1st and 2nd nearest neighbors for all subdomain
for (each *sprint* of the simulation) **do**
    for (each *sector* in subdomain) **do**
        for (each *MCS* of a sprint) **do**
            for (each *vacancy* in current sector) **do**

kMC core

                Calculate the activation energy associated with the 12 1st nearest neighbors of the vacancy
                Calculate vacancy exchange frequency and real time for this MCS
                Select randomly a 1st neighbor for new position of the vacancy
                Swap the vacancy with the selected neighbor
                Store the vacancy move in the array ***moves***
            endFor
        endFor
        Eliminate false moves, convert coordinates, and generate ***movesX|Y|Z***
        Send and receive ***movesX*** to/from the neighbor process in X direction
        Send and receive ***movesY*** to/from the neighbor process in Y direction
        Send and receive ***movesZ*** to/from the neighbor process in Z direction
    endFor
    if (this sprint is a snapshot point) **then**
        Master process gathers subdomains from all processes and writes configuration to file
endFor

Simulations setup

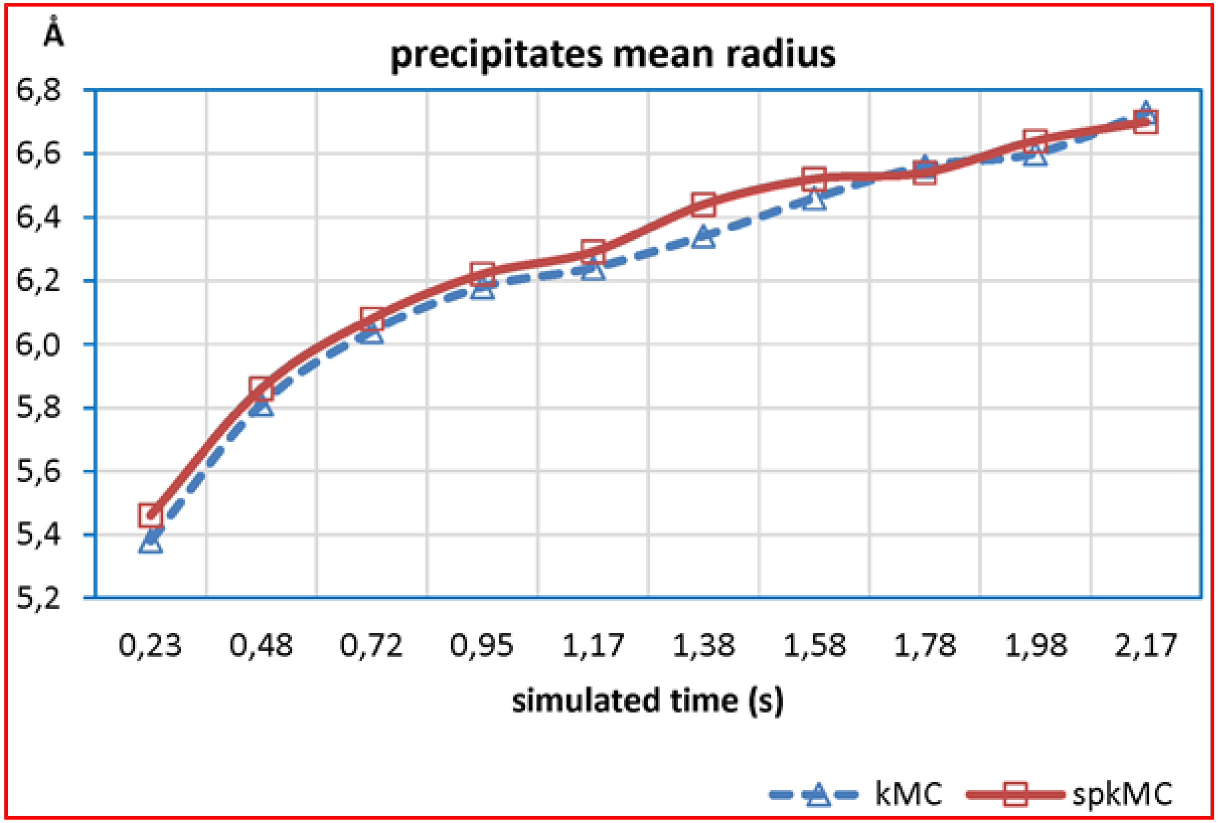
- The simulations were run on the University of Minho **SeARCH cluster**
- Cluster nodes run **Linux x86_64**
- The code was compiled with **gcc 4.9.0** and **Open MPI 1.8.4**
- Hardware configuration of each **node**:
  - 2 processors/sockets
  - Processors: Intel Xeon E5-2650 v2, with ivy bridge microarchitecture, 2.6 GHz, 8 physical cores, and 16 cores with hyper-threading
  - 64GB of RAM
  - 20MB of L3 cache
  - 256KB of L2 cache per core
  - 32KB of L1D and 32KB of L1I cache per core.
- Inter-node **communication**: Ethernet and Myrinet

Comparing kMC and spkMC output based on 4 metrics

Metrics:

T=673 K , 1% Sc

- **Mean radius (Å)**
  + −
- Mean size (atoms)
  + −
- Number of precipitates
  + −
- Precipitates/*Lattice sites*
  + −



Comparing kMC and spkMC output based on 4 metrics

T=673 K , 1% Sc

Metrics:

- Mean radius (Å)

  + −

- **Mean size (atoms)**

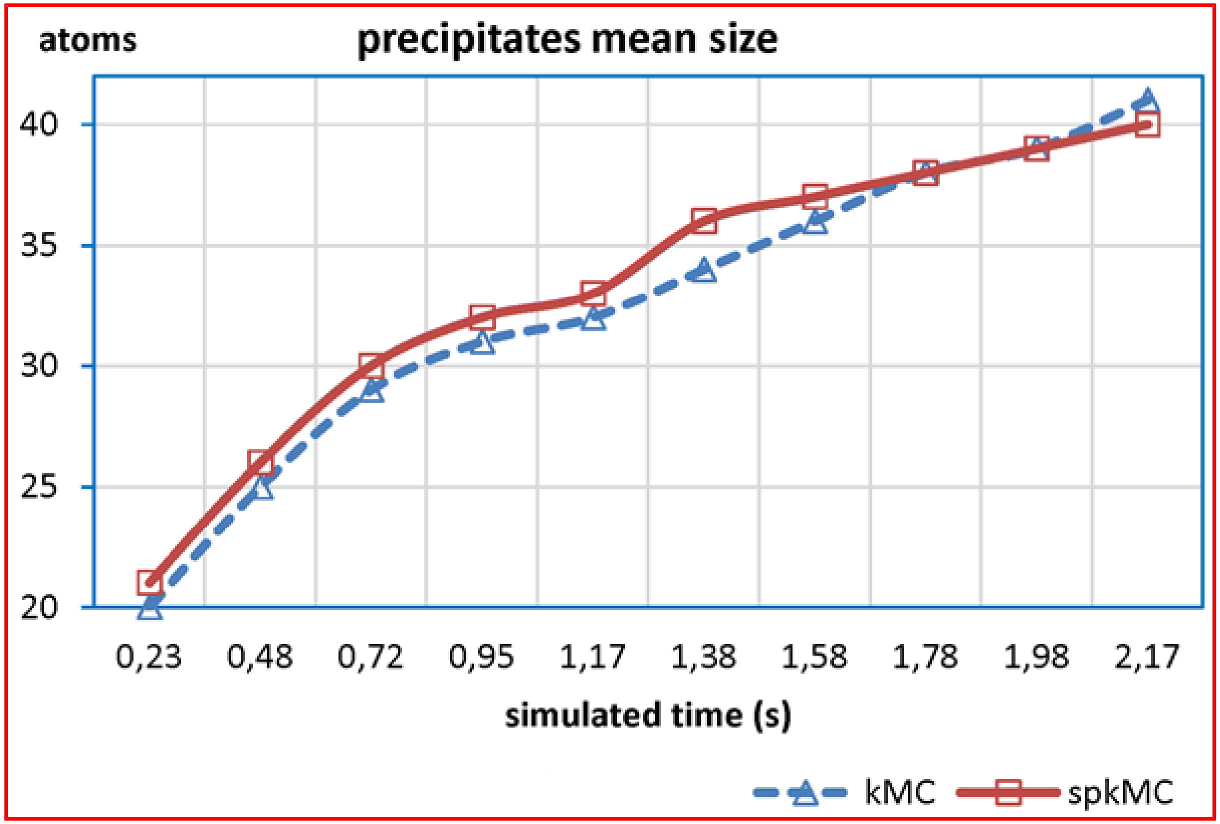
  + −

- Number of precipitates

  + −

- Precipitates/*Lattice sites*

  + −



Comparing kMC and spkMC output based on 4 metrics

Metrics:

T=673 K , 1% Sc

- Mean radius (Å)

  \+ −

- Mean size (atoms)

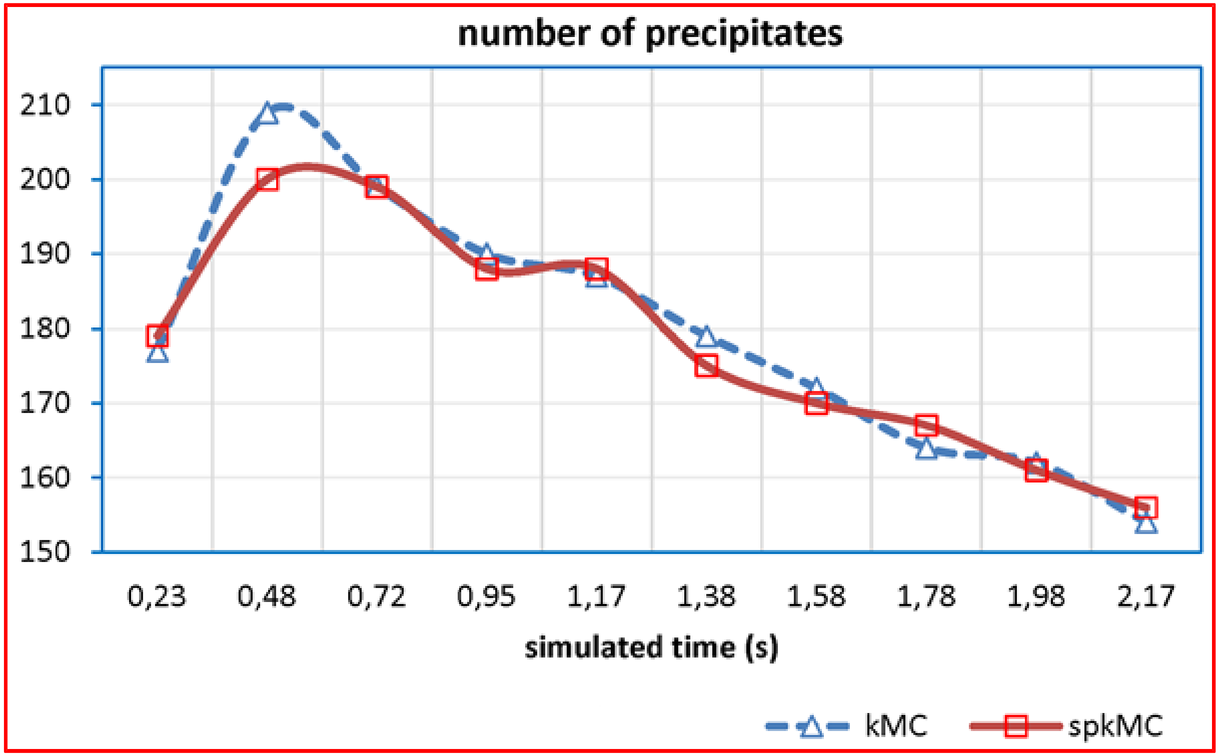
  \+ −

- **Number of precipitates**

  \+ −

- Precipitates/*Lattice sites*

  \+ −

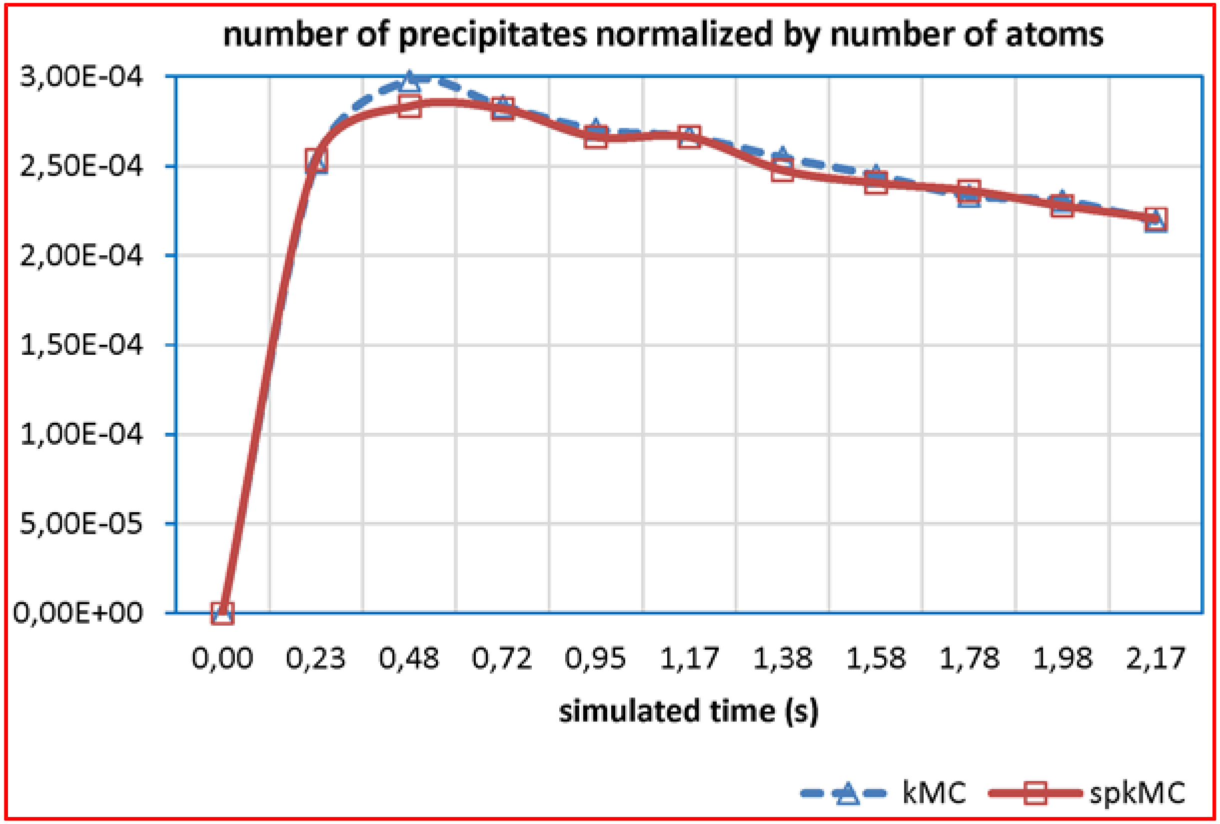


Comparing kMC and spkMC output based on 4 metrics

T=673 K , 1% Sc

Metrics:

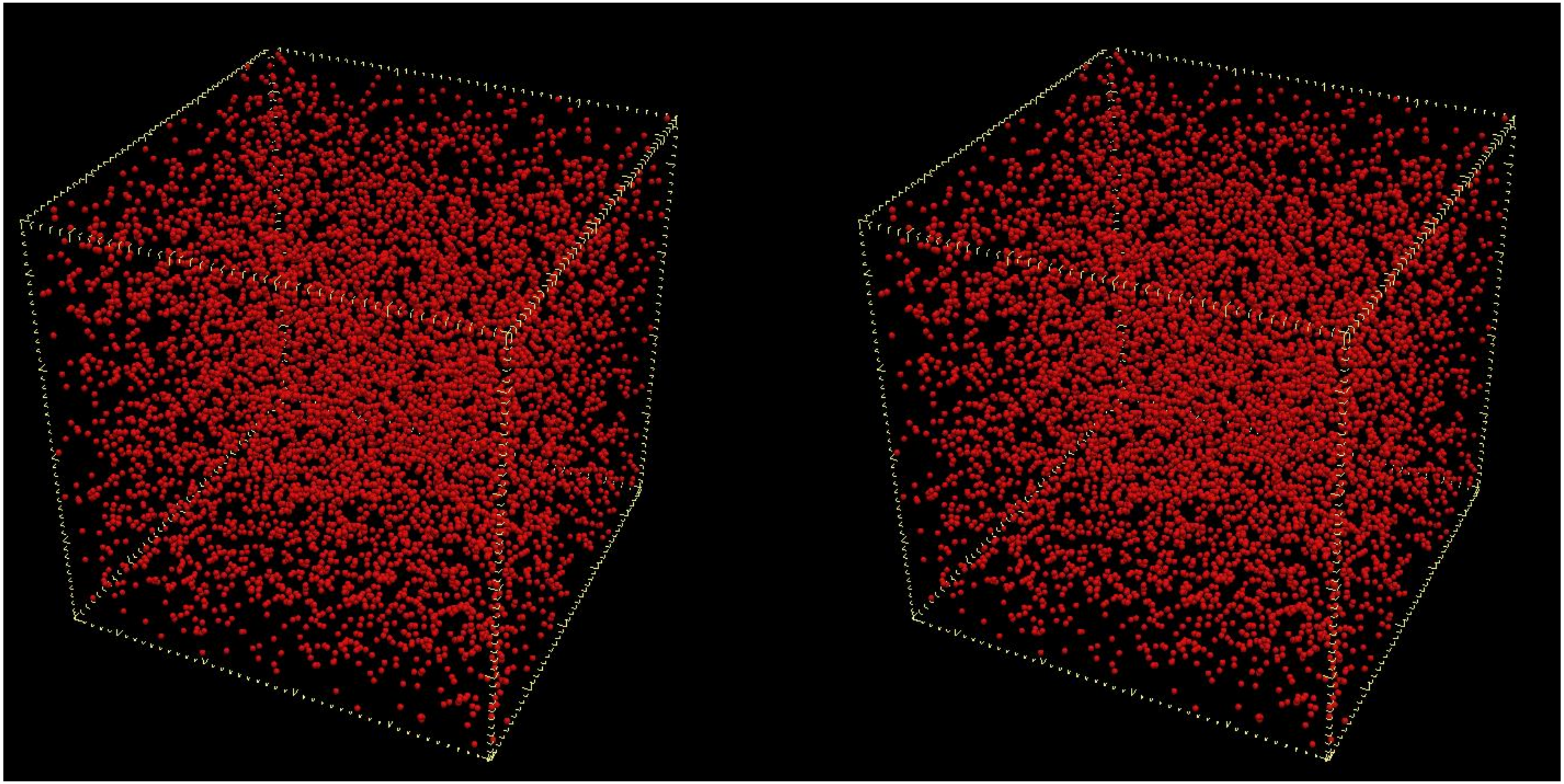
- Mean radius (Å)
  + −
- Mean size (atoms)
  + −
- Number of precipitates
  + −
- **Precipitates/*Lattice sites***
  + −



spkMC simulation evolution :: T=873 K

output from spkMC simulation

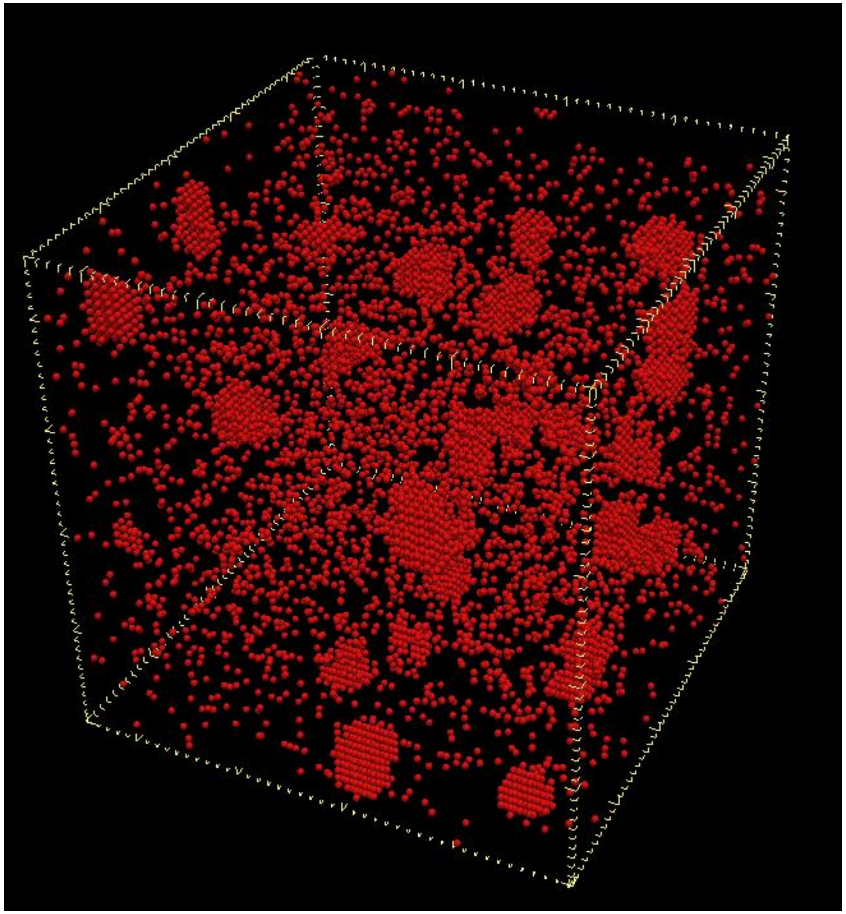
output from DBSCAN clustering



0 ms

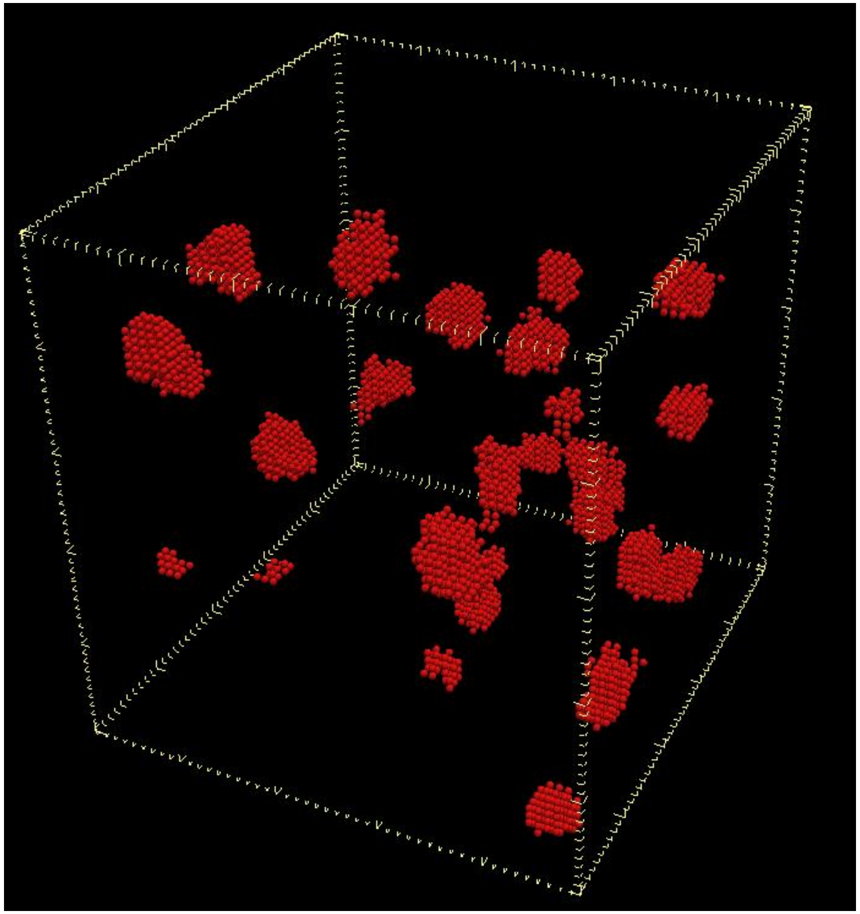
spkMC simulation evolution :: T=873 K

output from spkMC simulation



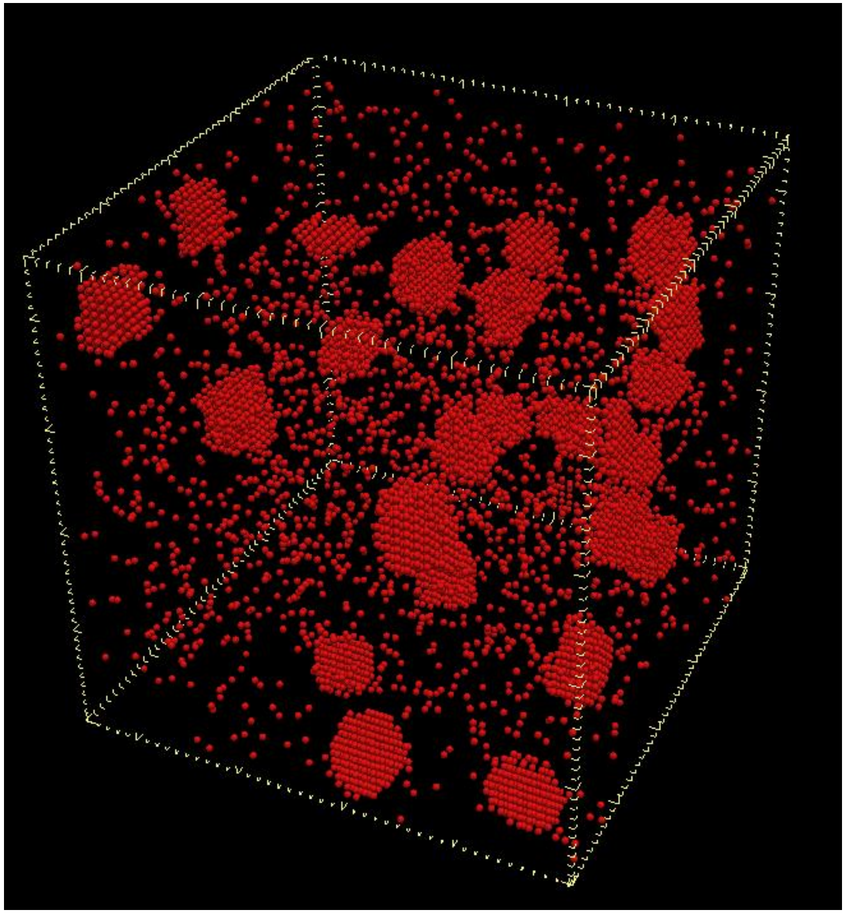
0.54 ms

output from DBSCAN clustering



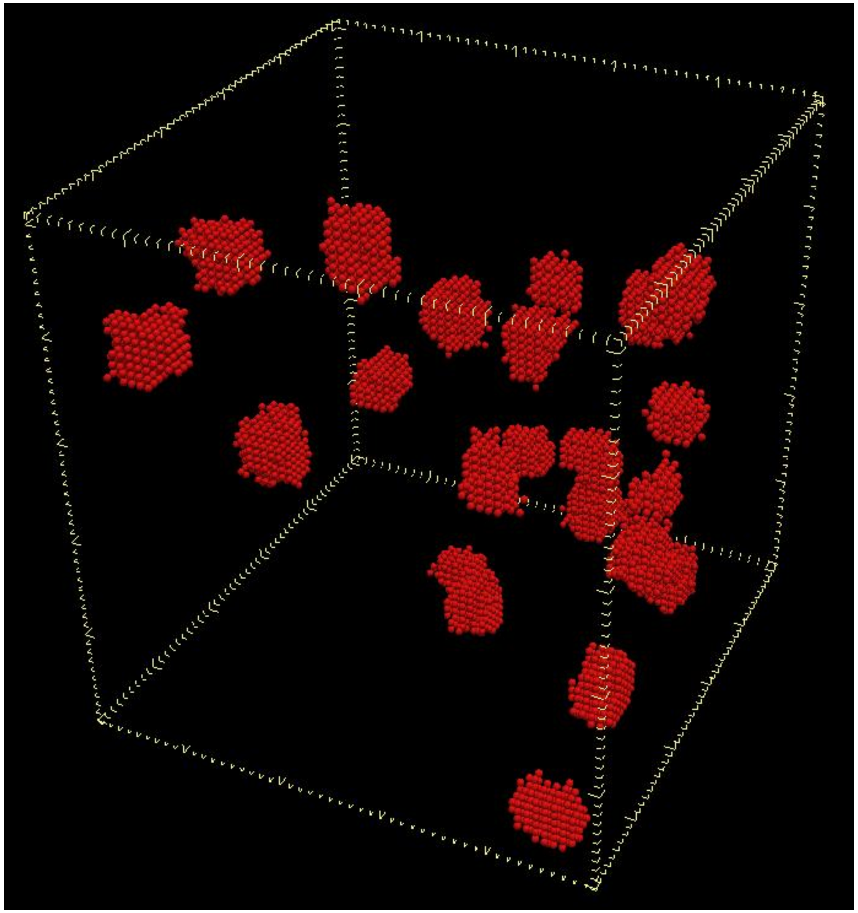
spkMC simulation evolution :: T=873 K

output from spkMC simulation



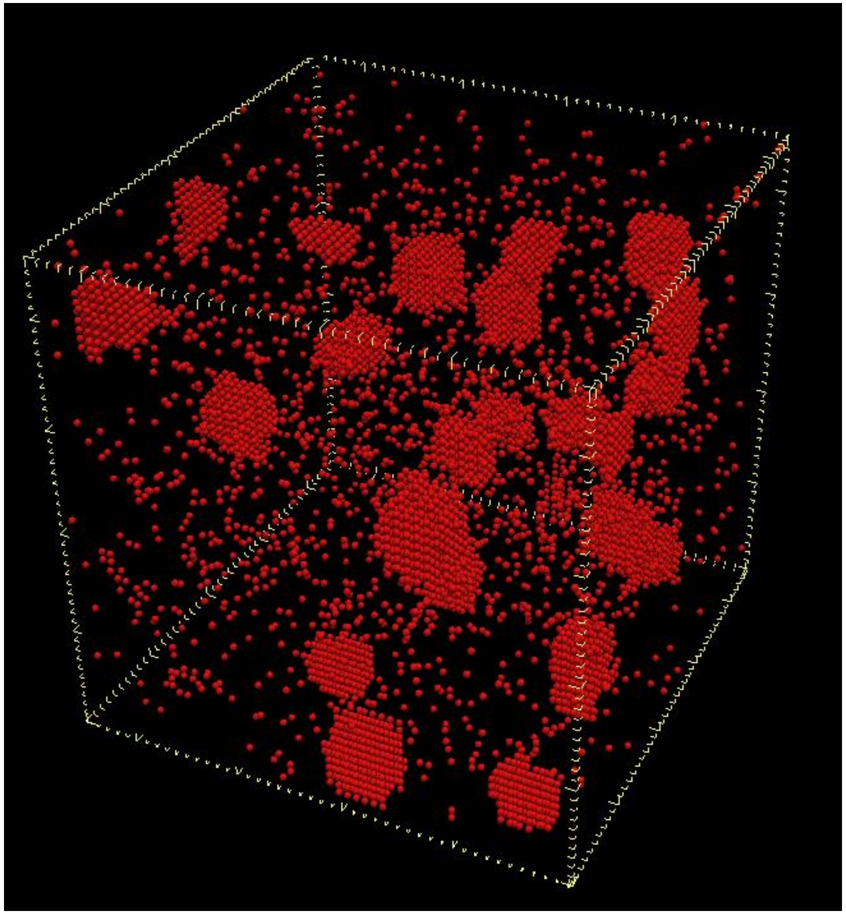
1.00 ms

output from DBSCAN clustering



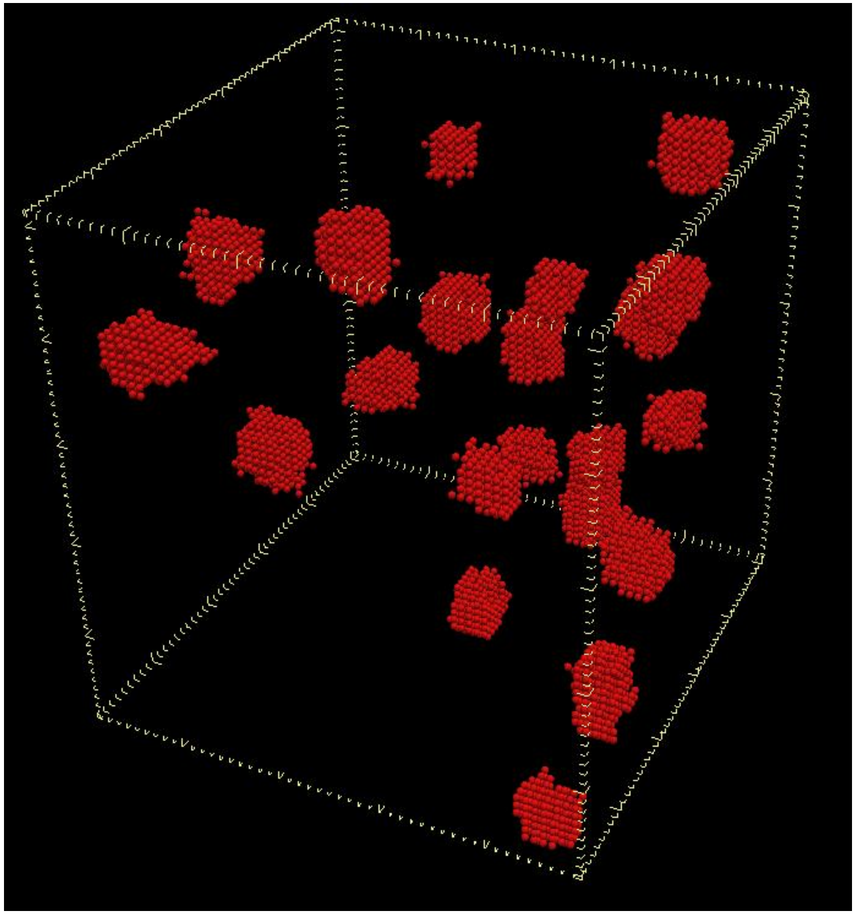
spkMC simulation evolution :: T=873 K

output from spkMC simulation



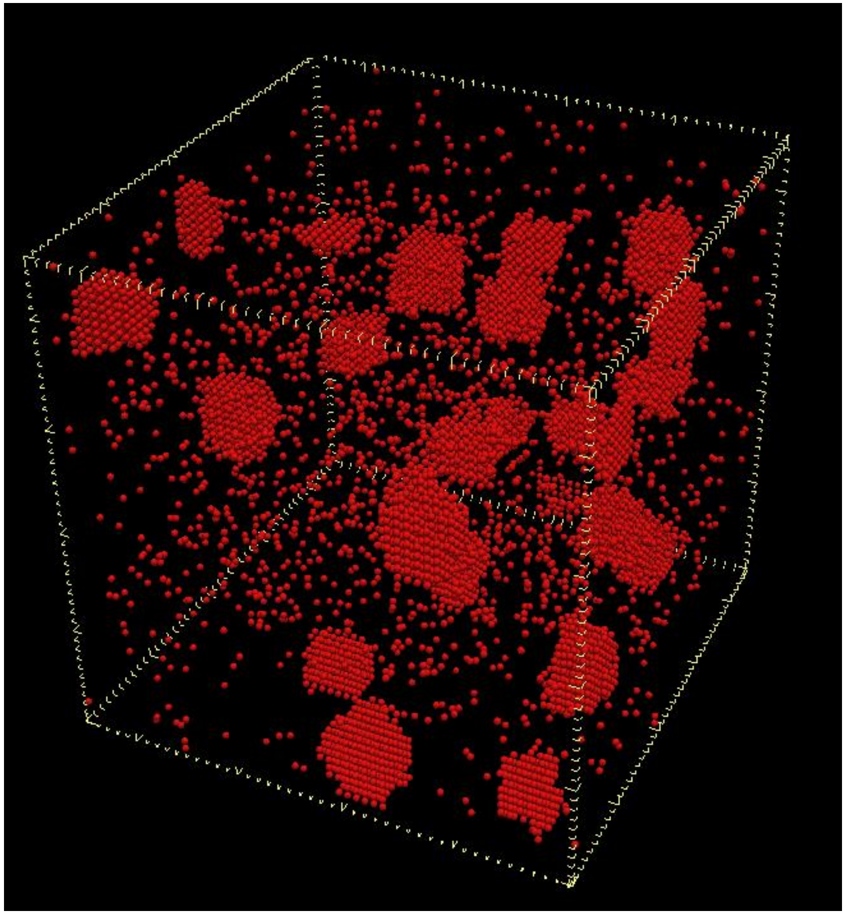
1.44 ms

output from DBSCAN clustering

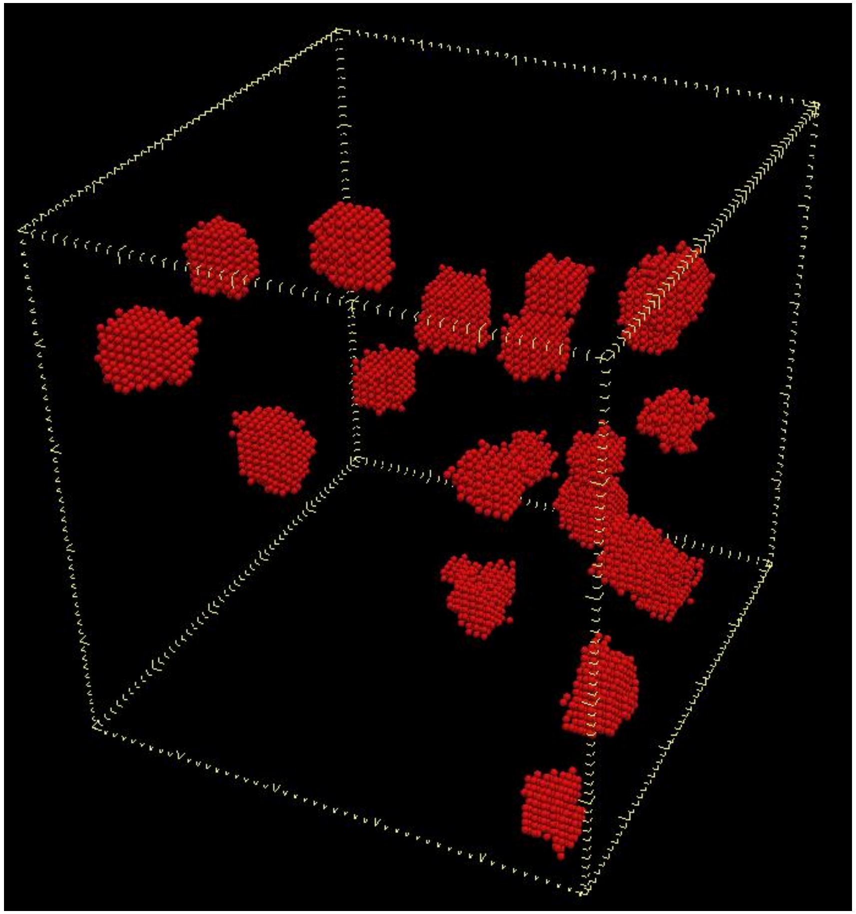


spkMC simulation evolution :: T=873 K

output from spkMC simulation



output from DBSCAN clustering

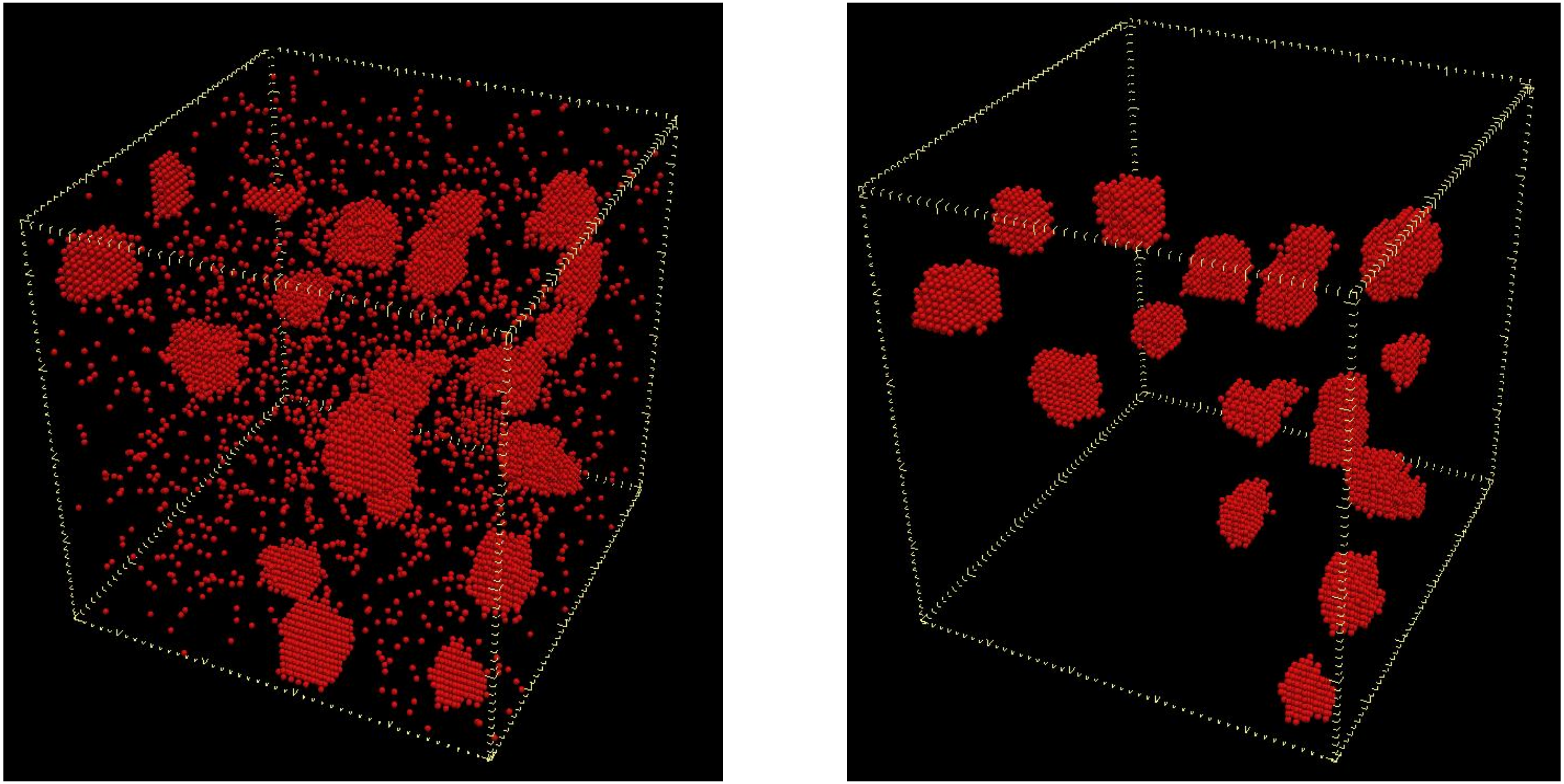


1.86 ms

spkMC simulation evolution :: T=873 K

output from spkMC simulation

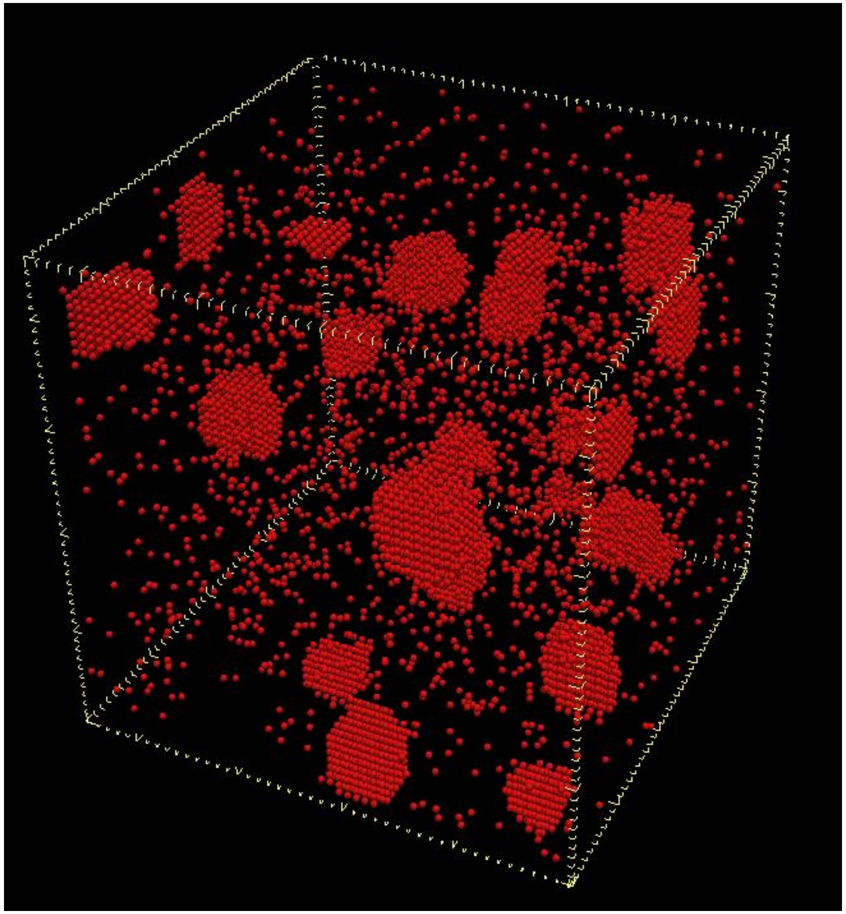
output from DBSCAN clustering



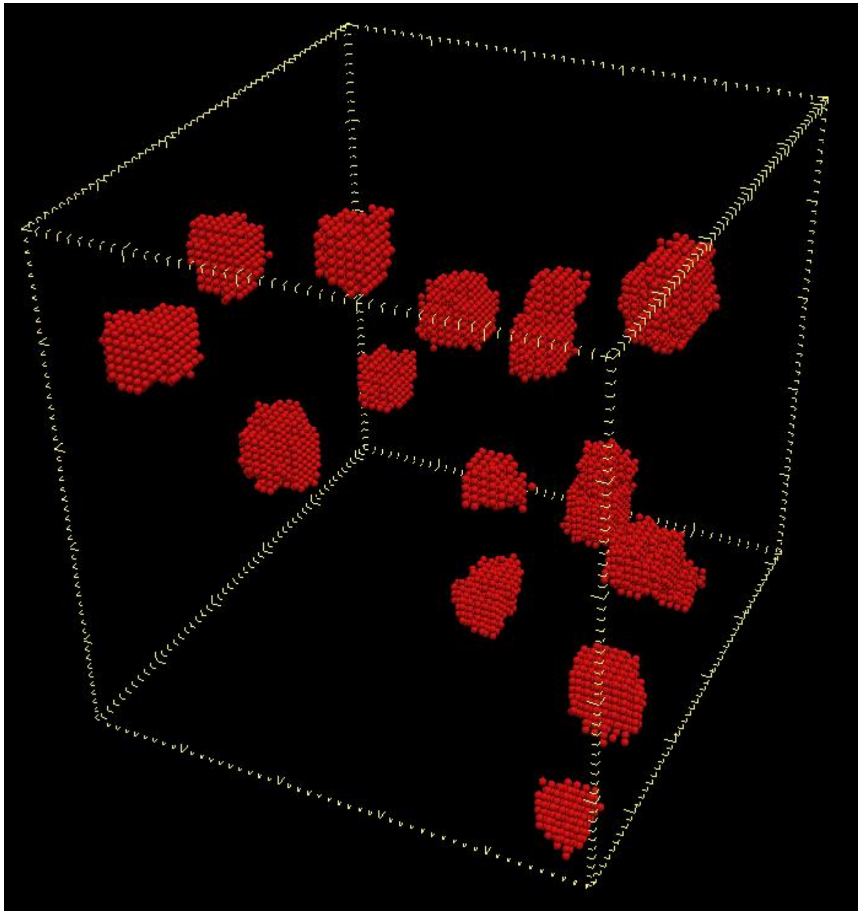
2.27 ms

spkMC simulation evolution :: T=873 K

output from spkMC simulation



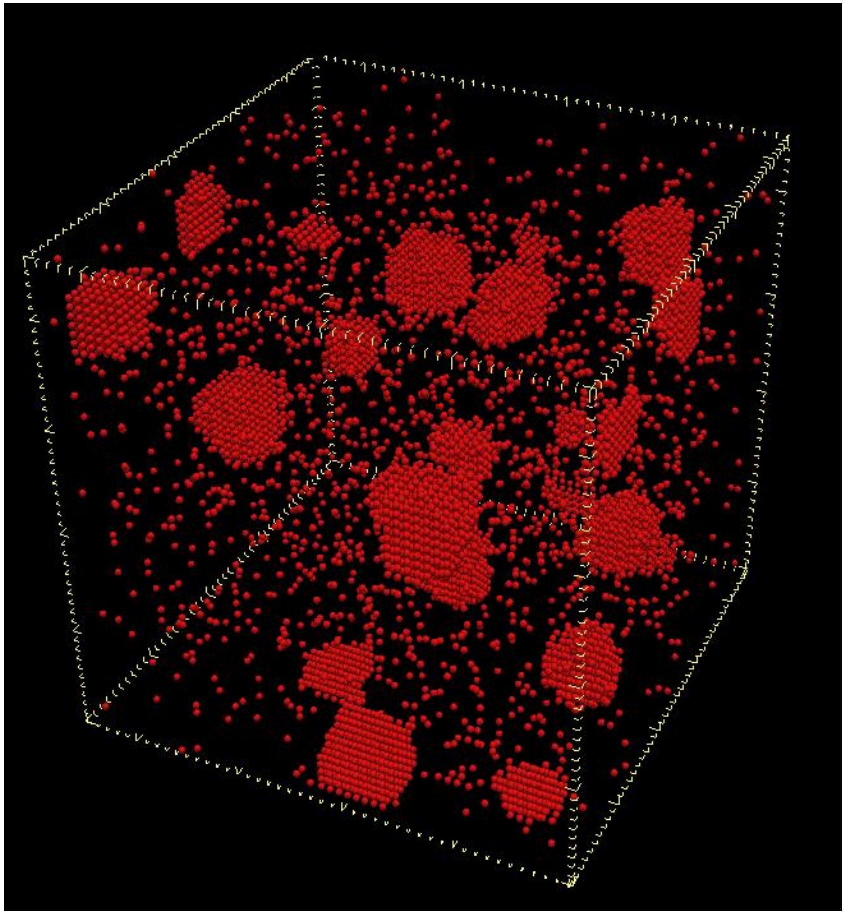
output from DBSCAN clustering



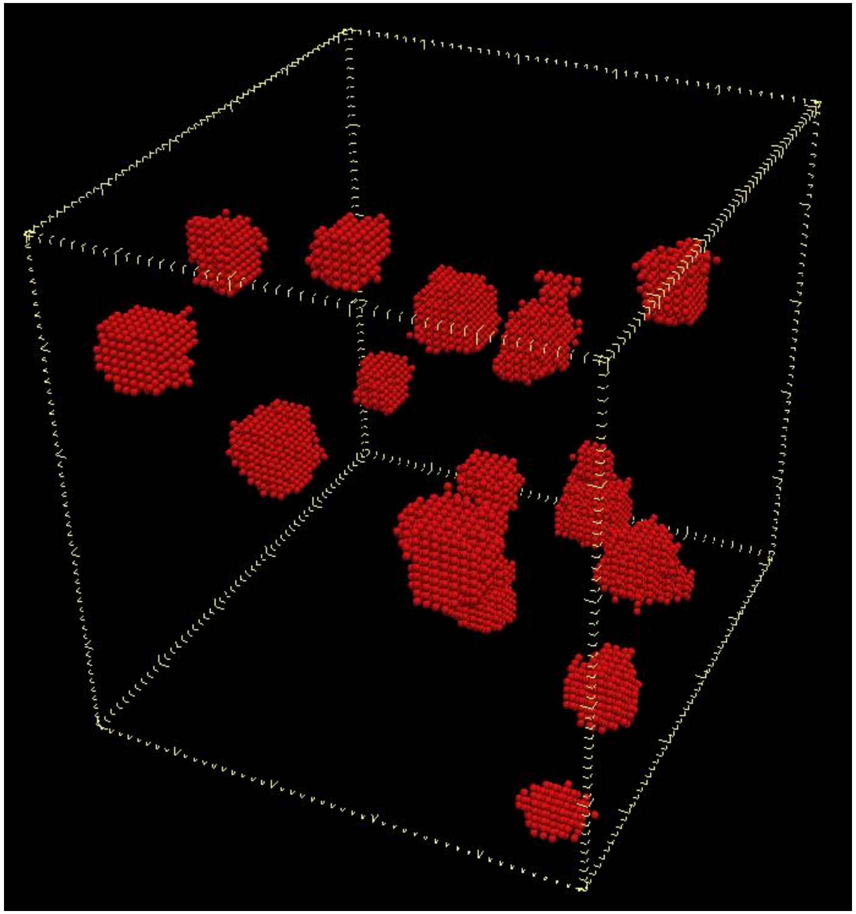
2.68 ms

spkMC simulation evolution :: T=873 K

output from spkMC simulation



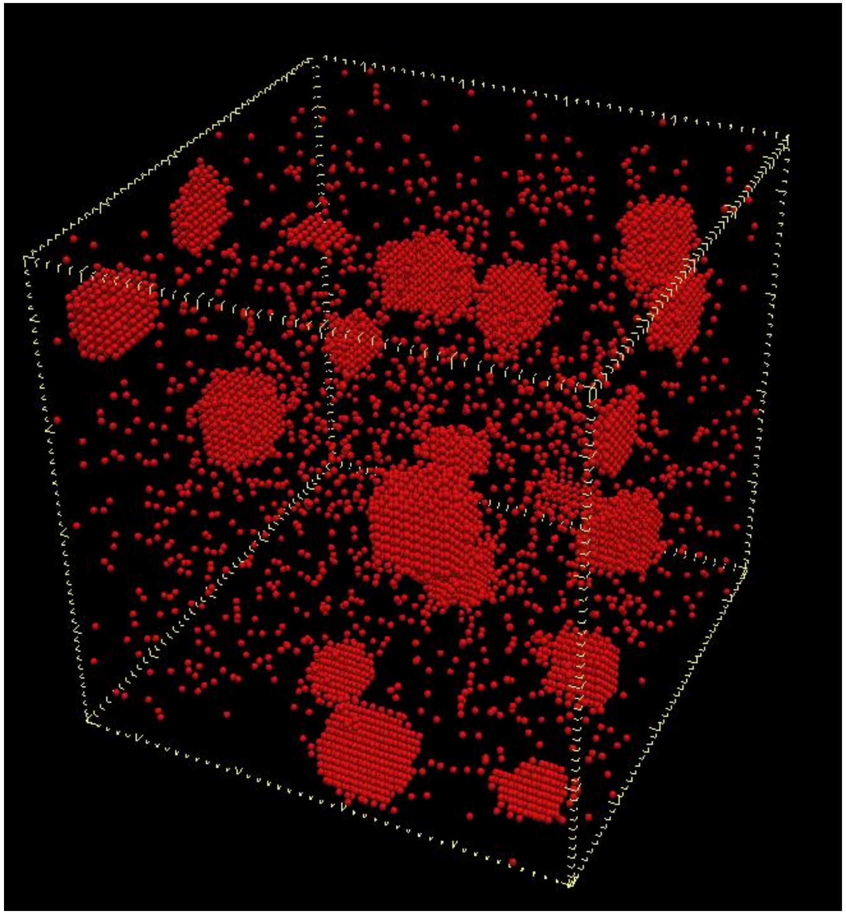
output from DBSCAN clustering



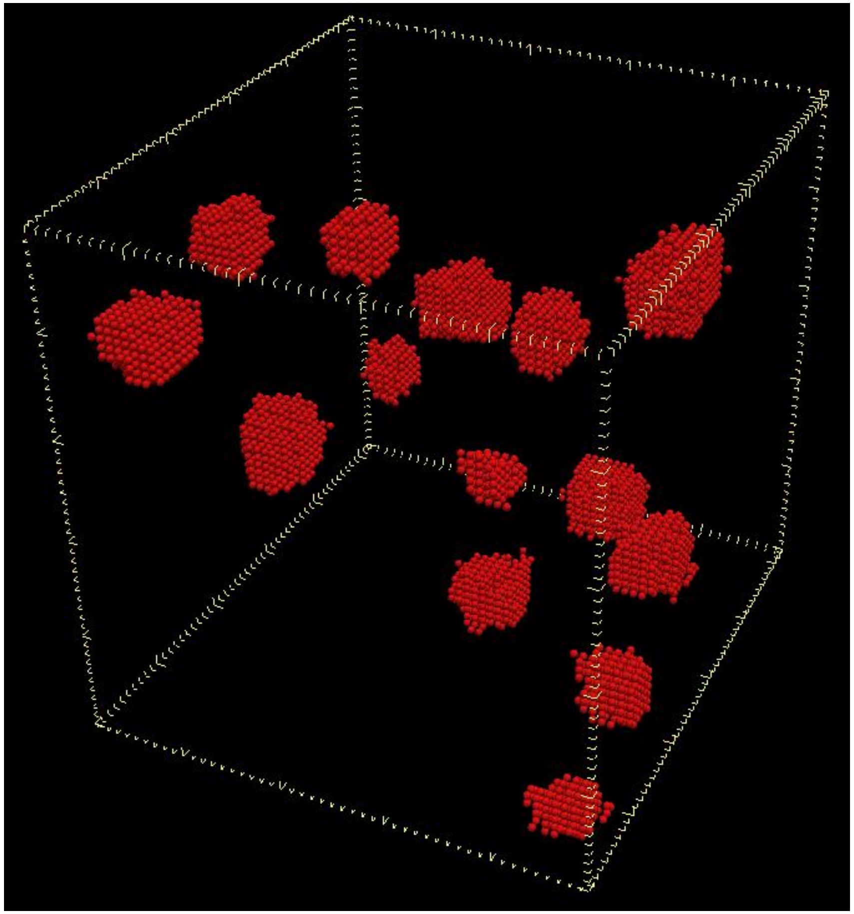
3.09 ms

spkMC simulation evolution :: T=873 K

output from spkMC simulation



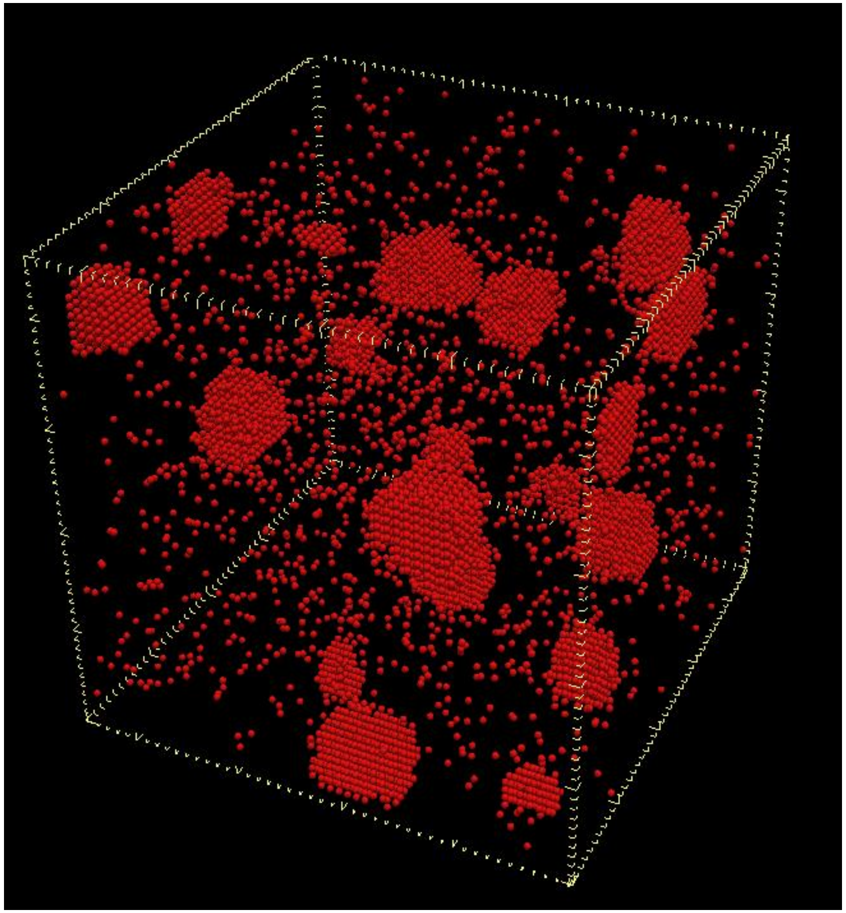
output from DBSCAN clustering



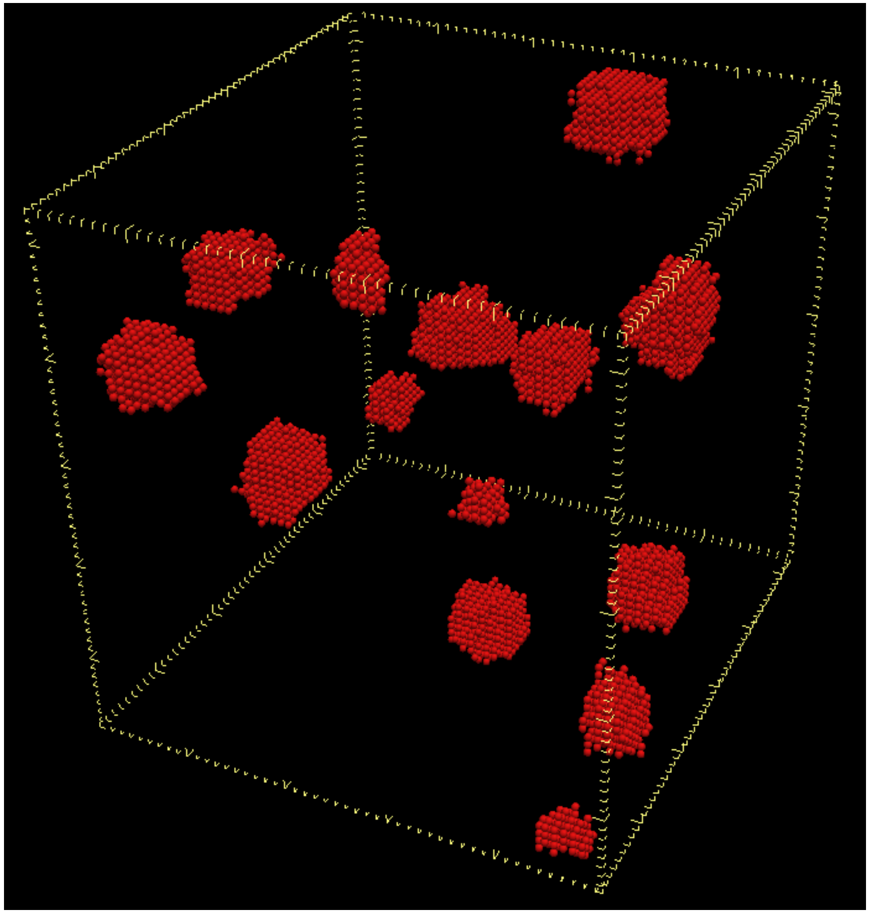
3.49 ms

spkMC simulation evolution :: T=873 K

output from spkMC simulation



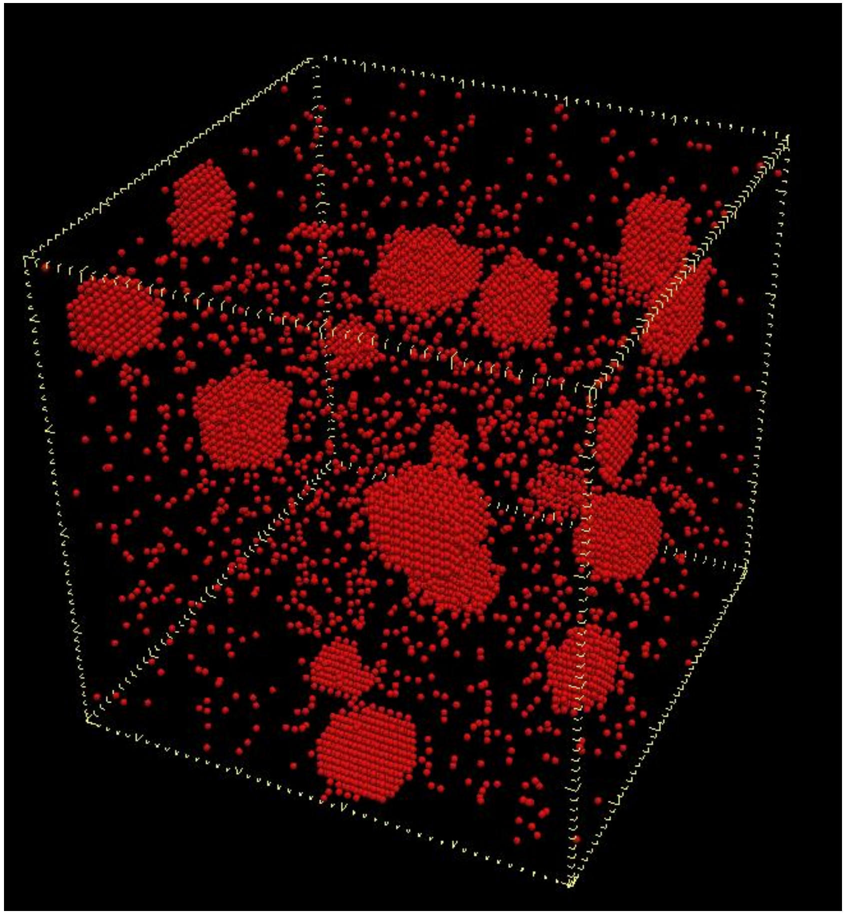
output from DBSCAN clustering



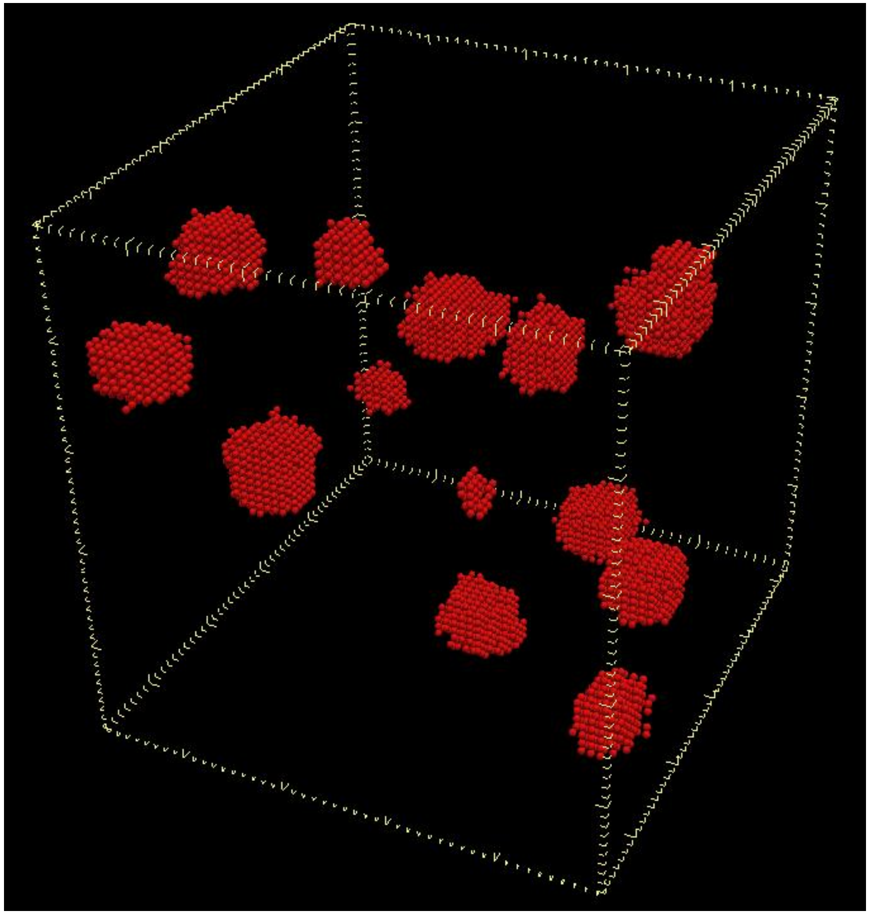
3.89 ms

spkMC simulation evolution :: T=873 K

output from spkMC simulation



output from DBSCAN clustering

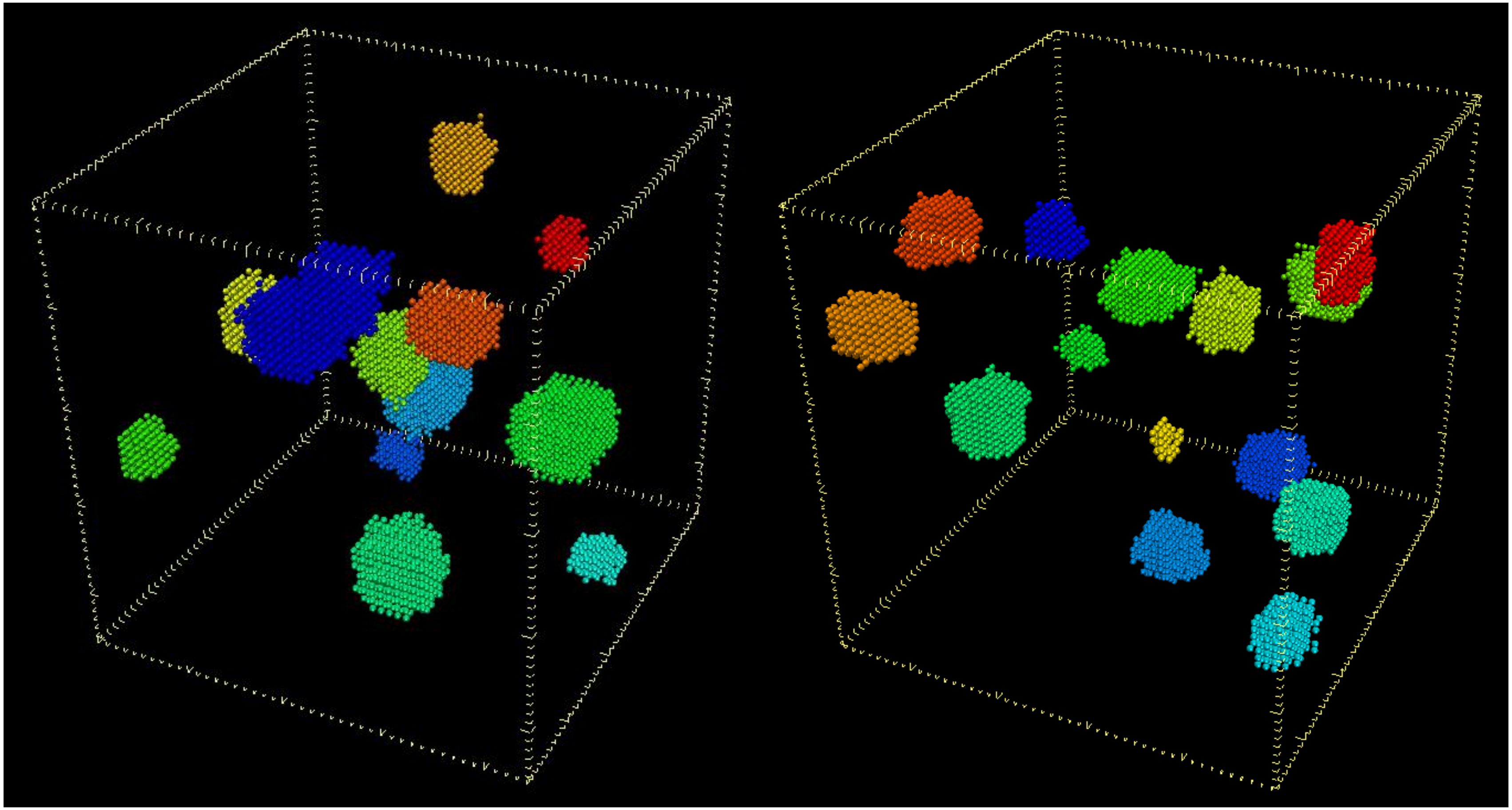


4.29 ms

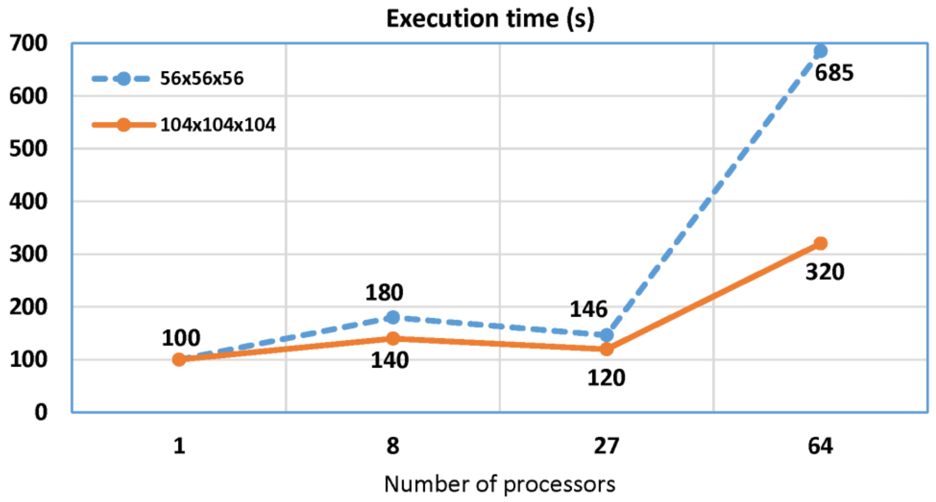
Comparing kMC and spkMC final configuration :: T=873K

1.0% Sc - kMC

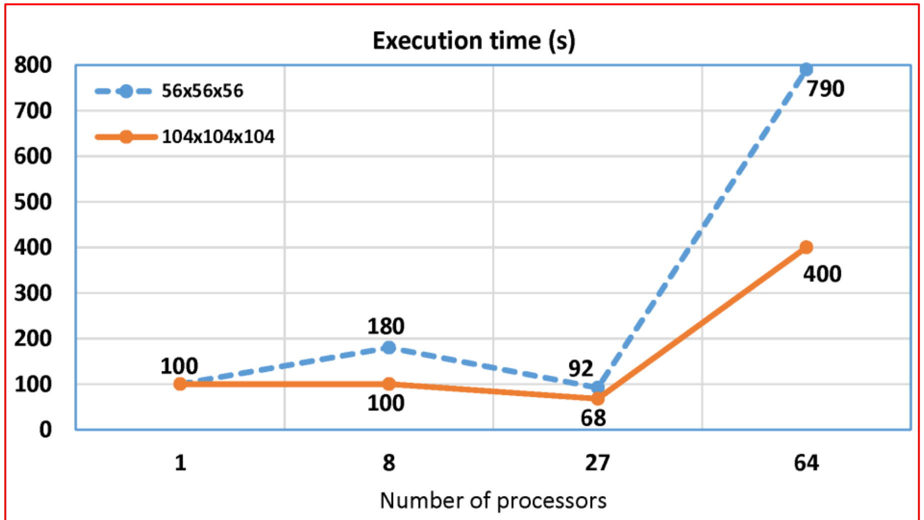
1.0% Sc - spkMC



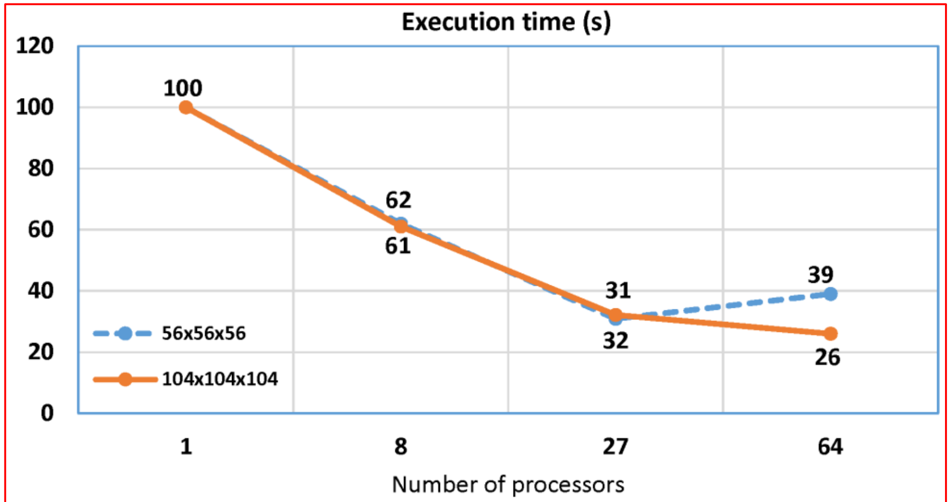
Performance of the different spkMC prototypes



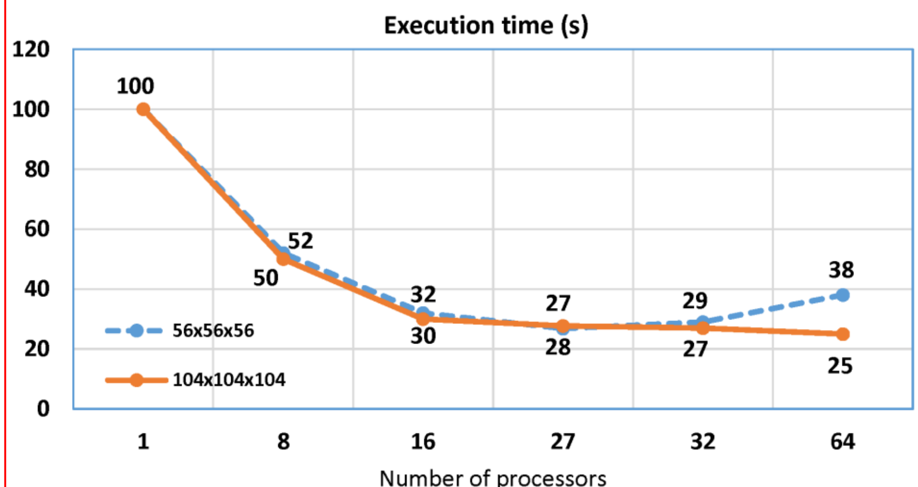
(a) lock-unlock version, ethernet



(b) lock-unlock version, myrinet

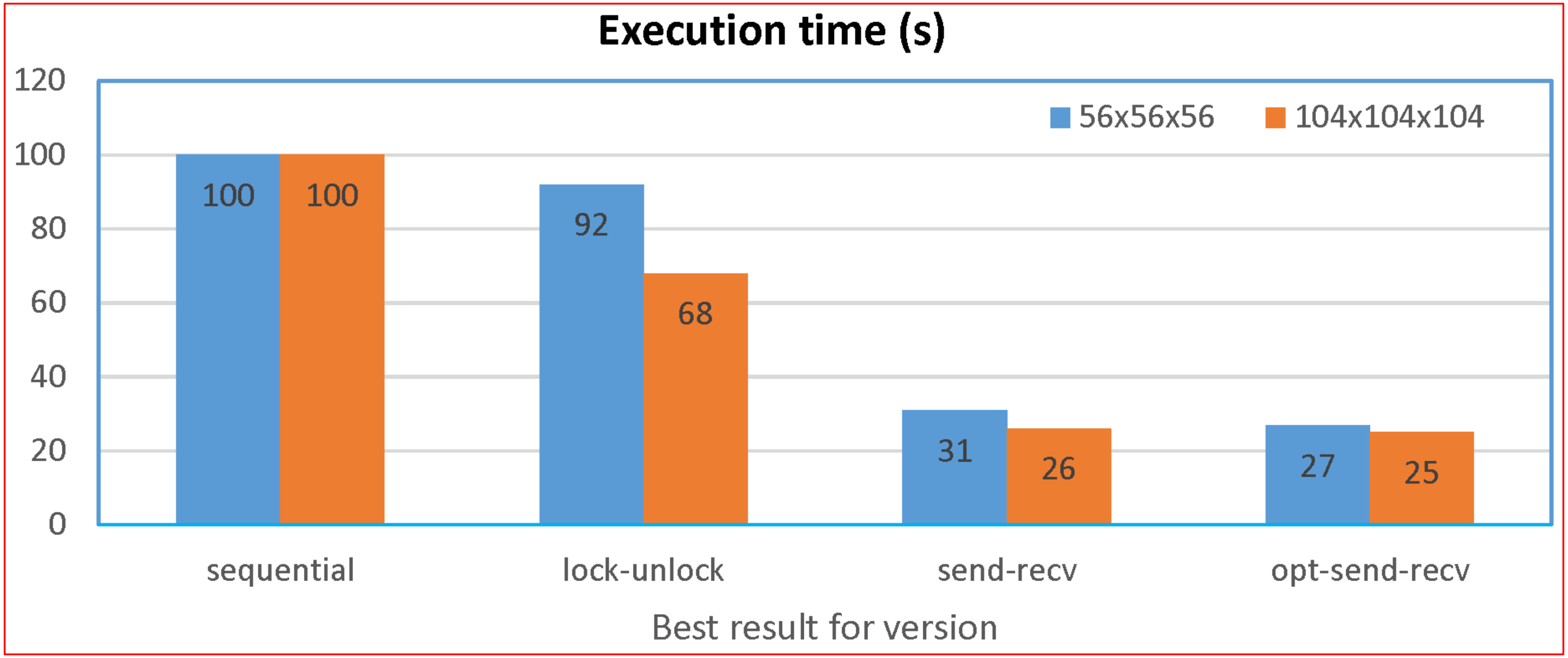


(c) send-recv version



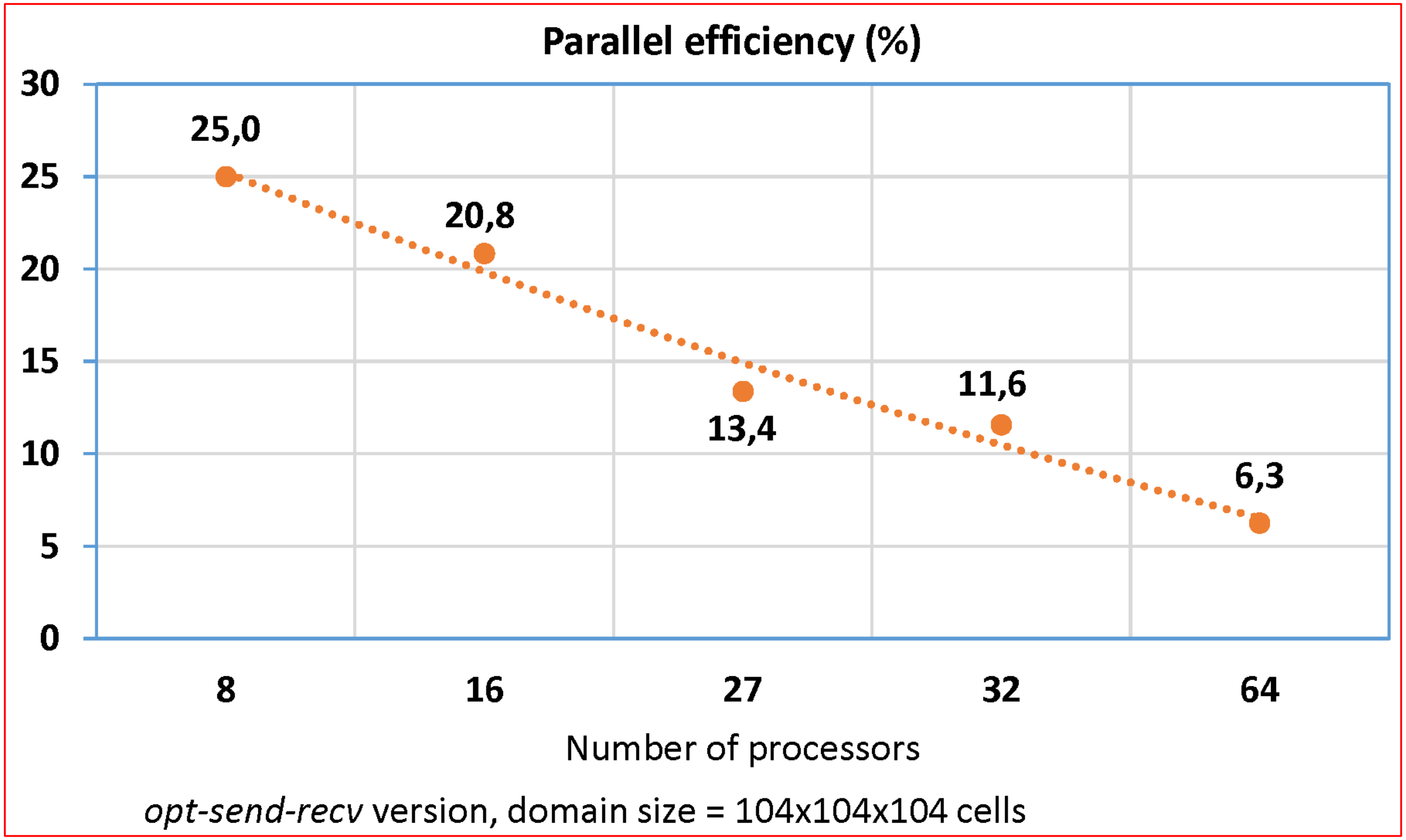
(d) opt-send-recv version

Comparing the best performance of different prototypes



- **Speedup** of the best spkMC prototype in relation to sequential kMC is **4**

Parallel efficiency of the *opt-send-receive* prototype



- spkMC presents a **low parallel efficiency**

Conclusions

- spkMC reproduces accurately the statistical behavior of the sequential kMC
- The precipitation problem is not embarrassingly parallel → spkMC only presents a 4x speedup when compared to kMC
- Open MPI 1.8.4 does not support RMA natively → RMA did not disclosed its potential in the lock-unlock prototype

Future Work

- Improve the parallel simulation performance and scalability
  - Prevent the migration of vacancies between sectors
    - eliminates the **iterations complexity** associated with multiple vacancies
    - improves the **load balancing** between processes
  - Overlap communication with computation
  - Use a hybrid **MPI-OpenMP** implementation to improve intra-node computation performance and still allow more parallelism than a single node
  - Take advantage of the improved **RMA** support allowed by **Open MPI 2.0.0**