Track B: Particle Methods – Part 1

PRACE Spring School 2012

Maciej Cytowski (ICM UW)

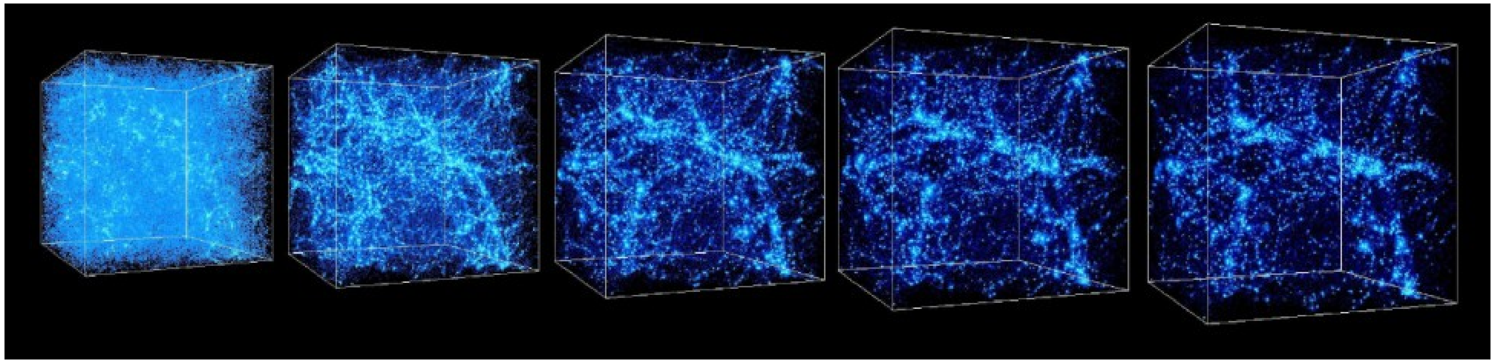
PART1: Particle methods

- Computational problem and implementation ~ 20 min.
- First look at the code and execution on Notos ~ 30 min.
- Parallel Random Number Generation (SPRNG) ~ 30 min.
- Visualization with VisNow ~ 30 min.

Computational problem and implementation

Particle methods

- **Simulation of particles interacting with each other in 3D space**
- **Scientific areas:**
  - Cosmology (e.g. GADGET-3),
  - Molecular modelling (e.g. GROMACS),
  - Plasma physics (e.g. PEPC),
  - other..
- **Large scale simulations (e.g. 4096^3 particles)**



Simulation steps – scalar version

1. Random generation of positions of N particles
2. Computing interaction forces between particles
3. Movement of particles with respect to force field
4. Write current state to file
5. Repeat steps 2-4 (or stop if conditions are fulfilled)

Simulation steps – parallel version

1. Parallel random generation of positions of N particles
2. **Assigning particles to processors (load-balancing)**
3. Computing interaction forces between particles (**local and remote**)
4. Movement of particles with respect to force field
5. Write current state to file **with the use of parallel I/O**
6. Repeat steps 2-5 (or stop if conditions are fulfilled)

Load-balancing

- **Load balancing** = data partitioning between available processes
- In order to minimize the execution time and enable good scalability we need to provide proper load-balancing (each process should have a similar amount of work assigned)

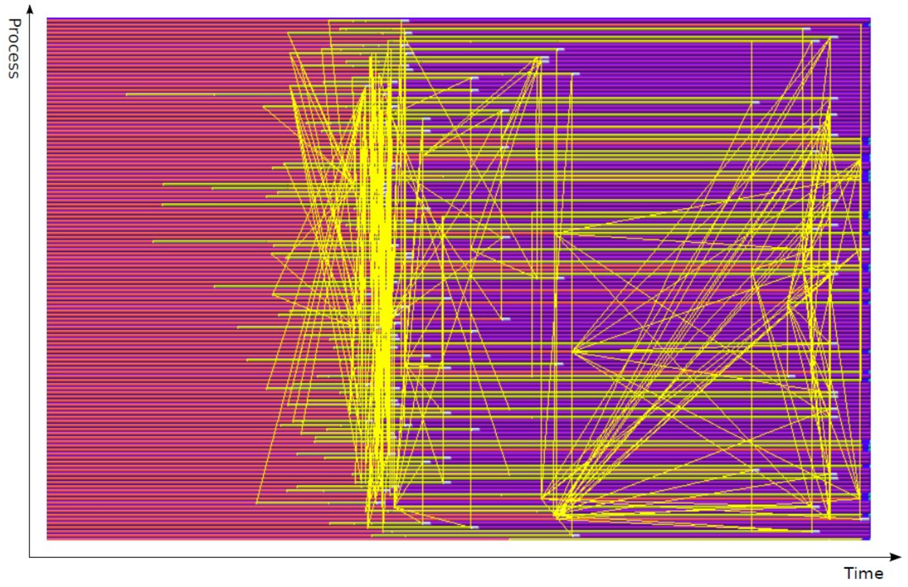
How to provide efficient load-balancing in our case (particle system)?

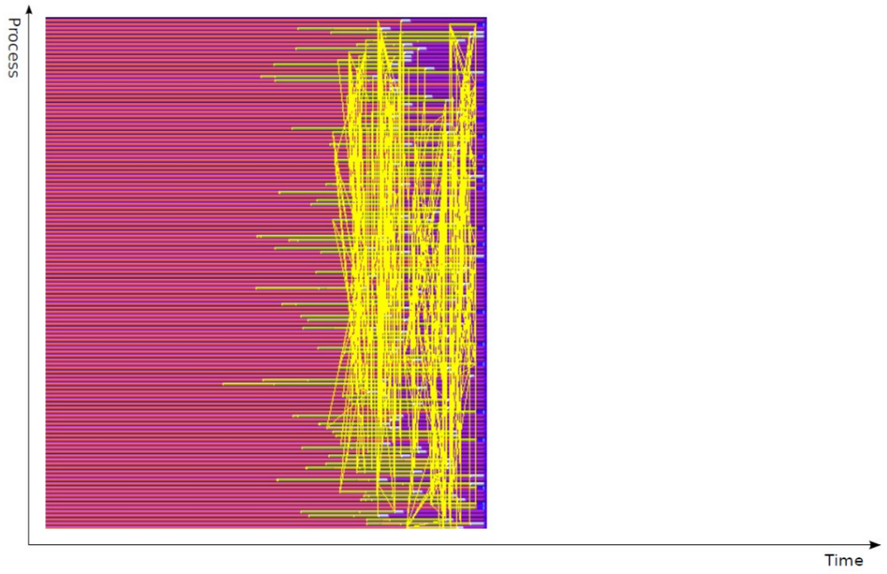
- Naive solution:
  - Partition the 3D box to P equal size sub-boxes and assign each sub-box to distinct processes
  - **Remark:** this may lead to unequal work distribution between processes

What about scalability?

Load-balancing

- **Unequal work distribution between processes leads to longer computation times and limited scalability**

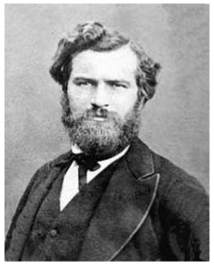




Other ideas?

Curves in topology

- In 1887 **Camille Jordan** defined a curve as follows (so called Jordan curves): **The curve is a continuous mapping of [0,1] interval**



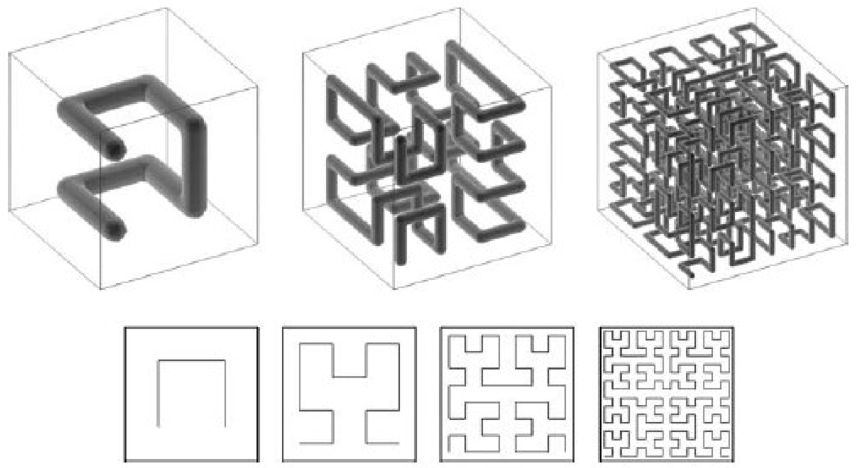
- The definition turned out to be very wide.

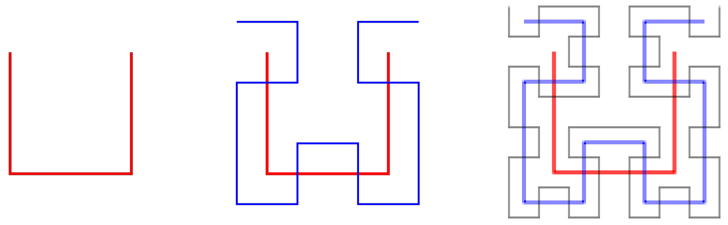
- Three years later Italian mathematician **Giuseppe Peano** discovered an example of Jordan curve whose range contains the entire 2-dimensional unit square.

- A different curve of the same kind was discovered and described by **David Hilbert**

Space-filling curves

- **In mathematical analysis, a space-filling curve is a curve whose range contains the entire 2-dimensional unit square (or more generally an N-dimensional hypercube)**
- **Hilbert curve in 2D:**
  - Start with „U" shape in 2x2 grid
  - Base of the „U" shape is replaced with two „U" shapes
  - We put two rotated „U" shapes on both sides of the initial „U" shape
  - In this way we approach 4x4 grid
  - We iterate this process and approach to 2^nx2^n grids
- For each point in N-dimensional hypercube we are able to find its corresponding value in the [0,1] interval

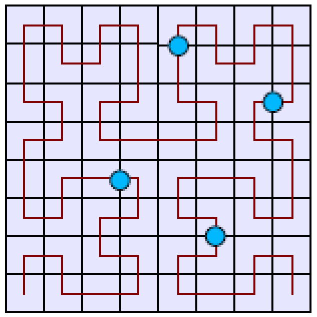

V.Springel „The cosmological simulation code GADGET-2", Mon. Not. R. Astron. Soc. 364, 1105–1134 (2005)

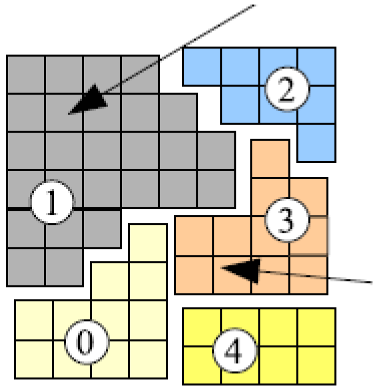


Load-balancing with Space-filling curves

- For each particle in the 3D box we can find its corresponding value in the [0,1] interval
- We cut the [0,1] interval into fragments containing equal numbers of particles
- We map fragments to parallel processes

- This kind of load-balancing methods (geometrical load-balancing methods) have very nice property of geometrical locality which is very important when computing particle interactions with other particles in neighbourhood





Our particle simulation code

- We will develop computer code that simulates interations between particles in 3D space
- Usually such codes are based on efficient tree algorithms to compute interations between particles (e.g. Barnes-Hut algorithm)
- Tree based algorithms are out of the scope of this training
- We will use a simple Particle-Particle method (N^2 complexity)
- We will focus mainly on load-balancing and hybrid parallelization

Parallelization – MPI & OpenMP

- We will use technology standards for parallelization:
  - **MPI**: Message Passing Interface library for parallel programming on distributed memory systems
  - **OpenMP**: pragma based programming model for parallel programming on shared memory systems
  - **MPI + OpenMP hybrid model**:
    - very high number of MPI processes (hundreds of thousands)
    - programming massively parallel systems with multi-core nodes.

Load-balancing – Zoltan library

- The **Zoltan** library is a collection of data management services for parallel, unstructured, adaptive, and dynamic applications,
- Sandia National Laboratories
- It simplifies the load-balancing, data movement, unstructured communication, and memory usage difficulties
- Applications: adaptive finite-element methods, particle methods, crash simulations,
- Zoltan implements **dynamic load-balancing**
- Website: http://www.cs.sandia.gov/zoltan
- User's guide: http://www.cs.sandia.gov/zoltan/ug_html/ug.html
- License: LGPL

Parallel Random Number Generation – SPRNG library

- **SPRNG** = Scalable Parallel Random Number Generators Library
- Florida State University
- The goal of the SPRNG project was to develop, implement and test a scalable package for parallel pseudo random number generation which will be easy to use on a variety of architectures
- SPRNG 2.0 – various SPRNG random number generators each in one library
- Uses GNU Multiple Precision (GMP) package
- Website: http://sprng.cs.fsu.edu/
- Documentation: http://sprng.cs.fsu.edu/ -> SPRNG -> Version 2.0 -> User's Guide

First look at the code

First look at the code

- Location on Notos system: **/opt/prace/particles**
- Files and directories:
  - **main.c** – main() function (MPI initialization, main simulation loop)
  - **particles.h** – structures, global variables, defines
  - **init.c** – initialization of the particles system
  - **decompose.c, decompose.h** – Zoltan decompisition functions
  - **dynamics.c** – particle interactions and movement
  - **io.c, io.h** – particle I/O (for visualization with VisNow)
  - **Makefile** – makefile (Notos settings)
  - **particles.ll** – LoadLeveler script for execution on Notos
  - **particles.out.example** – example output file
  - **results/** - directory with simulation output files

main.c

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<mpi.h>
#include<particles.h>

int main(int argc,char **argv) {

  int i;
  int iter;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  if(argc<3) {
    if(rank==0) {
      fprintf(stderr,"Parameters are missing.\n");
    }
    exit(1);
  } else {
    np = atoi(argv[1]);
    niter = atoi(argv[2]);
  }

  generateParticles(np);

  //decompositionInit(argc,argv);

 for(iter=0;iter<niter;iter++) {

    if(rank==0) printf("Iteration %d\n",iter);

    write_particles(iter);
    //decompose();
    //compute_forces();
    //move();

  }

  MPI_Barrier(MPI_COMM_WORLD);
  MPI_Finalize();

  return 0;
}
```

main simulation loop

particles.h

```
#include <zoltan.h>

struct vector3d{
  float x;
  float y;
  float z;
};

struct particle_data{
  ZOLTAN_ID_TYPE gid;
  struct vector3d position;
  struct vector3d velocity;
  struct vector3d force;
};

struct particle_data *particles;

struct export_list_data{
  int particle;
  int proc;
};
```

particle data structure

```
int rank;        // MPI rank of process
int size;        // number of MPI processes

int niter;
int np;          // global number of particles
int lnp;         // local (per process) number of particles

float boxsize;
float epsilon;

#define G 6.674
```

init.c

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

#define SEED 985456376

/* This function generates particle data and sets up few simulation
        parameters */

int generateParticles(int np) {

  int p;

  boxsize=64.0;
  epsilon=8.0;

  lnp=np/size;

  particles = (struct particle_data*) malloc(lnp*2*sizeof(struct
        particle_data));

for(p=0;p<lnp;p++) {

   particles[p].gid=(ZOLTAN_ID_TYPE)(rank*lnp+p);

   particles[p].position.x = 0.0;
   particles[p].position.y = 0.0;
   particles[p].position.z = 0.0;

   particles[p].velocity.x = 0.0;
   particles[p].velocity.y = 0.0;
   particles[p].velocity.z = 0.0;

   particles[p].force.x=0.0;
   particles[p].force.y=0.0;
   particles[p].force.z=0.0;

  }

  return 0;

}
```

initial positions

array of particles

decompose.c

Decomposition driving functions – to be described later

Zoltan query functions – to be described later

```
#include <stdio.h>
#include <stdlib.h>
#include <particles.h>
#include <mpi.h>
#include <zoltan.h>
#include <decompose.h>


/* This function performs the initialization of Zoltan decomposition
    */
int decompositionInit(int argc,char **argv) {
…
}

/* This function performes the decomposition */
int decompose() {
…
}

int ztn_return_dimension(void *data, int *ierr) {
…}

void ztn_return_coords(…) {
…}

int ztn_return_num_node(…) {
…}

void ztn_return_owned_nodes(…) {
…}

int ztn_return_particle_data_size(…) {
…}

void ztn_pack(…) {
…}

void ztn_pre(…) {
…}

void ztn_mid(…) {
…}

void ztn_post(…) {
…}

void ztn_unpack(…) {
…}
```

dynamics.c

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>
#include<mpi.h>
#include<zoltan.h>
#include "particles.h"
#include "decompose.h"

/* This function computes forces between two given particles */
int force(struct particle_data *p1,struct particle_data *p2) {
…}

/* This function computes interactions between all particles in the
         simulation */
int compute_forces(){
…}

/* This function moves particles with respect to forces acting on
         them */
int move() {
…}

/* This function computes interactions between imported and local
         particles */
int compute_remote_forces(int nexp,struct export_list_data
         *export_list) {
…}
```

compute_remote_forces() will not be described in detail, but will be rather treated as a blackbox for computing remote interactions

Functions that computes interactions between particles – to be described later

Makefile

```
SYSTYPE = "Notos"
#SYSTYPE = "GNU"
ZOLTANDIR = /opt/zoltan/XL
SPRNGDIR = /opt/sprng
GMPDIR = /opt/gmp

ifeq ($(SYSTYPE),"GNU")
CC = mpicc
OPT = -O3 -ffast-math -ftree-vectorize –fopenmp
INCLUDES = -I. -I$(ZOLTANDIR)/include -I$(SPRNGDIR)/include
LIBS = -lm -L$(ZOLTANDIR)/lib -lzoltan -lsprng -lgmp
endif

ifeq ($(SYSTYPE),"Notos")
CC = mpixlc_r
OPT = -O3 -qstrict -qarch=450d -qtune=450 -qsmp=omp -g
INCLUDES = -I. -I$(ZOLTANDIR)/include -I$(SPRNGDIR)/include
LIBS = -lm -L$(ZOLTANDIR)/lib -L$(SPRNGDIR)/lib -L$
        (GMPDIR)/lib -lzoltan -lsprng -lgmp
endif

EXEC = particles

OPTIONS = $(OPT)

OBJS = main.o init.o decompose.o io.o dynamics.o

INCL = particles.h

CFLAGS = $(OPTIONS) $(INCLUDES)

$(EXEC): $(OBJS)
        $(CC) $(OPTIONS) $(OBJS) $(LIBS) -o $(EXEC)

$(OBJS): $(INCL)

clean:
        rm -f $(OBJS) $(EXEC) results/*
```

particles.ll

```
# @ job_name = particles
# @ account_no = G47-17
# @ class = prace
# @ error = particles.err
# @ output = particles.out
# @ environment = COPY_ALL
# @ wall_clock_limit = 00:20:00
# @ job_type = bluegene
# @ bg_size = 4
# @ queue
mpirun -exe particles -args "32768 1" -env "OMP_NUM_THREADS=1" -mode VN -np 16
```

Program has two arguments:
- global number of particles
- number of iterations

First compilation

1. ssh login@atol.icm.edu.pl
2. ssh notos
3. cp –r /opt/prace/particles ~/
4. cd particles
5. module load mpi_default
6. make

First execution

1. llsubmit particles.ll
2. llq
3. less particles.out

Parallel Random Number Generation – SPRNG library

SPRNG basics

- Given a RNG and its initial state, each call to the generator produces the next random number in the sequence.
- Each sequence of random numbers is called a **random number stream**.
- **SPRNG** can be used to produce a **totally reproducible** stream of parallel pseudorandom numbers, independent of the number of processors used in the computation and of the loading produced by sharing of the parallel computer (important during debugging and porting)
- Seed does not refer precisely to the starting state of the random number stream. **Distinct streams initialized with the same seed will have different starting states**.

SPRNG functions

- We will use three SPRNG functions:
  - **init_sprng()** – initializes the SPRNG library
  - **sprng()** – generates random numbers
  - **free_sprng()** – frees the memory used by SPRNG

init_sprng()

```
int *init_sprng(int rng_type, int streamnum, int nstreams, int
seed, int param)
```

•**init_sprng** initializes random number streams

•**nstreams** is the number of distinct streams that will be initialized across all the processes and must be greater than 0

•**streamnum** is the stream number, typically the processor number, and must be in [0,nstreams-1]

•**seed** is the seed to the random number generator. It is acceptable (and recommended) to use the same seed for all the streams

•The argument **param** selects the appropriate parameters of the RNG. The macro SPRNG_DEFAULT, defined in the SPRNG header files, can be used to choose the default parameters

•**init_sprng** returns the ID of the stream when it completes successfully

•**rng_type = DEFAULT_RNG_TYPE;**

sprng()

```
double sprng(int *stream)
```

- **sprng** generates random number in [0,1)
- **stream** is the ID of the stream from which the next random number in [0,1) is returned by this function
- **stream** argument must have been obtained by a prior call to **init_sprng**, **spawn_sprng** or **unpack_sprng**

free_sprng()

```
int free_sprng(int *stream)
```

- this function frees the memory used to store information concerning the random number stream identified by the stream ID **stream**.

Hands-on (1) – Exercise 1

- Edit **init.c** file

- Include header file:
```
#include <sprng.h>
```

- Define the seed for the RNG:
```
#define SEED 985456376
```

Hands-on (2) – Exercise 1

- Edit **init.c** file
- Inside **generateParticles()** function:
  - Declare variables for generator properties (int) and for the random stream (int*)
  - Set the properties of RNG (use the default settings)
  - Initialize the SPRNG library
  - Use the sprng function to generate random X,Y and Z coordinates (each in the range of [0,boxsize)) of each particle in the loop
  - Free the memory used by SPRNG

Hands-on (3) – Exercise 1

- Compile and run the code
- It should produce a random placement in 3D box
- In order to see the placement we will use the visualization package called VisNow (see the next presentation)

- Copy the file **results/step0000** to your laptop:
  - On Notos: **scp results/step0000 delta:~/**
  - On your laptop: **scp login@atol.icm.edu.pl:step0000 directory/**