A technology for the design of hybrid supercomputer simulation codes for relativistic particle electrodynamics*

A.V.Snytnkov[1,2], M.A.Boronina[1], E.A.Mesyats[1,2], A.A.Romanenko[2]

Institute of Computational Mathematics and Mathematical Geophysics SB RAS[1], Novosibirsk State University[2]

For the development of simulation codes for a group of similar physical problems one needs a tool of fast and flexible modification of the critical parts of the code. In the present work the modification is performed with C++ templates and inheritance. With inheritance one can transform a code for plasma simulation into a code for relativistic beam simulation. Moreover, the GPU plasma simulation code itself was built by means of C++ templates. Finally, the virtual methods is used to improve accuracy of the simulation. In such a way, C++ templates facilitate the quick and easy computation method replacement and also the transition from one supercomputer architecture to another.

1. Introduction

1.1. Languages and tools

Within the development of simulation codes, C++ provides tools of fast and flexible modification of the critical parts of the code. The examples of such critical parts are boundary conditions, field equation solvers, particle pushers, particle form factors and also the parts of the code related to the specific computer architecture (e.g. GPU kernels or OpenMP sections). In the present work the modification is performed with C++ templates and inheritance.

It should be explained why modern Fortran (Fortran 2003 or Fortran 2008) with its full-scale object-oriented tools is not used in the present work. This is because Fortran codes are much harder to debug and to make them reliable. Together with object-oriented tools and with support of hybrid architectures (CUDA, or OpenCL, or Open MP etc.) it makes the development of the code very complex. From this point of view, rewriting Fortran codes in C/C++ seems to be lesser evil compared to supporting object-oriented CUDA Fortran codes. Of course, using C++ will result in lesser performance compared to Fortran, but it will be most likely balanced by the easiness of development and debug.

"Hybrid supercomputer" in the title means only GPU-based supercomputer in the present work. Moreover, only Nvidia GPUs with CUDA will be mentioned. But why CUDA, and why CUDA only? Just because here it is an example of how the code is being ported to some other supercomputer architecture, most of the other tools like OpenCL, OpenMP, OpenACC may be used in the same way. At present we do not use OpenCL because there are not so many computers with AMD GPUs available. Moreover, there is no evidence that AMD devices with OpenCL will give much better performance than Nvidia devices with CUDA. Another tool for development of GPU codes, OpenACC is very attractive because of its relative simplicity, but the performance of the resulting codes is not satisfactory at the moment.

One more important question that should be discussed in advance is the use of virtual functions. In present work they are used to perform the same actions in some different way, for example, to implement different particle form-factors, or to implement boundary conditions on some different architecture. In such a way, virtual functions are strongly necessary for convenience of programming. On the other hand, the use of virtual functions in some cases

*Supported by RFBR under Grants 14-07-00241, 14-01-00392, 15-31-20150, and 14-01-31088.

reduces reliability of the code. In this case some simpler tools like pointers to function may be used instead.

1.2. Particle-In-Cell method

In computational electrodynamics the Particle-In-Cell [1, 2] method is widely used. At present there is a lot of object-oriented implementations of the PIC method (e.g. OOPIC library [3]), and also the template libraries for PIC method [4, 5], but mostly for CPU-based supercomputers, not for hybrid ones.

The goal of this work is to make a general purpose code that is capable of solving a number of similar physical problems. The proposed method of creating such a code is not to include all the possible functionality and all the physical effects and all the numerical methods that could be imagined. On the contrary, there must be a way to include in the code what is really necessary for the solution of a particular physical problem. Such a way is provided by the C++ templates. In this paper the consideration is limited to beam physics and plasma physics, in short, to particle electrodynamics, though it is not a principal limitation.

2. Particle-In-Cell plasma simulation code

2.1. The physical problem: beam-plasma interaction

The physical problem under study is the effect of anomalous heat conductivity observed at the GOL-3 facility in the Budker Institute of Nuclear Physics [6]. The GOL-3 facility is a long open trap where the dense plasma is heated up in a strong magnetic field during the injection of the powerful relativistic electron beam of a microsecond duration. The effect is the decrease of the plasma electron heat conductivity by 100 or 1000 times compared to the classical value for the plasma with the temperature and density observed in the experiment. Anomalous heat conductivity arises because of the turbulence that is caused by the relaxation of the relativistic electron beam in the high-temperature Maxwellian plasma. The goal of simulation is to define the origin and mechanism of the heat conductivity decrease. This is of great importance for the fusion devices because the effect of anomalous heat conductivity helps to heat the plasma and also to confine it. The problem of heat transport in fusion devices was widely discussed (e.g. [7, 8]) and some recent works [9].

2.2. Basic equations

The mathematical model employed for the solution of the problem of beam relaxation in plasma consists of the Vlasov equations for ion and electron components of the plasma and also of the Maxwell equation system. These equations in the usual notation have the following form:

$$\frac{\partial f_{i,e}}{\partial t} + \vec{v}\frac{\partial f_{i,e}}{\partial \vec{r}} + \vec{F}_{i,e}\frac{\partial f_{i,e}}{\partial \vec{p}} = 0, \qquad \vec{F}_{i,e} = q_{i,e}\left(\vec{E} + \frac{1}{c}[\vec{v}, \vec{B}]\right)$$

$$rot\vec{B} = \frac{4\pi}{c}\vec{j} + \frac{1}{c}\frac{\partial \vec{E}}{\partial t}$$

$$rot\vec{E} = -\frac{1}{c}\frac{\partial \vec{B}}{\partial t} \tag{1}$$

$$div\vec{E} = 4\pi\rho$$

$$div\vec{B} = 0$$

In the present work this equation system is solved by the method described in [10]. All the equations will be further given in the non-dimensional form. The following basic quantities are used for the transition to the non-dimensional form:

- characteristic velocity is the velocity of light $\tilde{v} = c = 3 \times 10^{10}$ cm/sec
- characteristic plasma density $\tilde{n} = 10^{14}$ cm^{-3}
- characteristic time $\tilde{t}$ is the plasma period (a value inverse to the electron plasma frequency) $\tilde{t} = \omega_p^{-1} = \left(\dfrac{4\pi n_0 e^2}{m_e}\right)^{-0.5} = 5.3 \times 10^{-12}$ sec

2.3. Numerical methods

The Vlasov equations are solved by the PIC method. This method implies the solution of the equation of movement for model particles, or superparticles. The quantities with the subscript i are related to ions and with the subscript e to electrons.

$$\frac{\partial \vec{p}_e}{\partial t} = -\left(\vec{E} + [\vec{v}_e, \vec{B}]\right)$$

$$\frac{\partial \vec{p}_i}{\partial t} = \kappa\left(\vec{E} + [\vec{v}_i, \vec{B}]\right)$$

$$\frac{\partial \vec{r}_{i,e}}{\partial t} = \vec{v}_{i,e}, \qquad \kappa = \frac{m_e}{m_i}, \qquad \vec{p}_{i,e} = \gamma \vec{v}_{i,e}, \gamma^{-1} = \sqrt{1 - v^2}$$

The leapfrog scheme is employed to solve these equations.

$$\frac{\vec{p}_{i,e}^{\,m+1/2} - \vec{p}_{i,e}^{\,m-1/2}}{\tau} = q_i\left(\vec{E}^m + \left[\frac{\vec{v}_{i,e}^{\,m+1/2} - \vec{v}_{i,e}^{\,m-1/2}}{2}, \vec{B}^m\right]\right)$$

$$\frac{\vec{r}_{i,e}^{\,m+1} - \vec{r}_{i,e}^{\,m}}{\tau} = \vec{v}_{i,e}^{\,m+1/2},$$

here τ is the timestep. The scheme proposed by Langdon and Lasinski [11] is used to obtain the values of electric and magnetic fields. The scheme employs the finite-difference form of the Faraday and Ampere laws. A detailed description of the scheme can be found in [10]. The scheme gives the second order of approximation with respect to space and time.

2.4. The implementation

The methods described in section 2.3 were first implemented as a Fortran code for general-purpose computers and then the code was parallelized. The parallel implementation is given in [12]. This code was tested and proved to be producing physically correct results [13].

In order to make the code more flexible and suitable for simulation of a wide range of physical phenomena it was rewritten in C++. The precision of computations remain the same, the results of computations with the Fortran and C++ code correspond exactly, bit to bit.

In C++ implementation the classes for cell and particle were introduced for the Particle-In-Cell method as well as the "simulation domain" class. In order to use special types of particle and special computational methods within a cell, the "simulation domain" class and the cell class were implemented as templates.

```
template <template <class Particle> class Cell >
class GPUPlasma: public Plasma
{
public:
//electric field evaluation
void ElectricField(double *locEx,double *locEy,double *locEz,
                   int nt,double *locHx,double *locHy,double *locHz,
                   double *loc_npJx,double *loc_npJy,double *loc_npJz);

// magnetic field evaluation
virtual void MagneticField(double *locQx,double *locQy,double *locQz,
                  double *locHx,double *locHy,double *locHz,
                            int nt,double *locEx,double *locEy,double *locEz);

//copy the cells from CPU memory to GPU memory
void CopyCells(Cell<Particle> **cp);

virtual void SetInitialConditions();

virtual void SetBoundaryConditions();

//comparing GPU results to the basic Fortran code results: debug only
double CheckGPUArraySilent()
}
```

Figure 1. The main components of the ”GPU simulation domain” class

3. Transition from the CPU plasma simulation code to the GPU plasma simulation code

The GPU plasma simulation code was built by means of C++ template tools. The ”simulation domain” class was previously implemented as a plain CPU code. The class has a set of methods for electromagnetic field evaluation. In order to achieve good GPU performance these field evaluation methods should be rewritten using CUDA kernels. It was performed through the creation of the derived class ”GPU simulation domain” (GPUPlasma, Fig.1) and the virtual methods implementing field evaluation. The bottleneck of PIC codes is the particle push. With the CPUs it takes up to 90 % of runtime. It means that the methods of the cell class performing the particle push were also rewritten using CUDA kernels.

On the basis of the PIC method template (class Plasma) a derived class was created (class GPUPlasma), which is also a template. In the GPUPlasma the following methods were added:

- copying of the domain to GPU
- comparison of CPU and GPU results
- invocation of GPU kernels for field evaluation

The implementation of the ”cell” for GPU (GPUCell class) was inherited from Cell class. It is important that particle storage within a cell must be optimized in terms of GPU memory, thus structure or class arrays are not suitable, thus it is done through a simple pointer-to-double *doubParticleArray*, Fig. 2. The GPUCell class, Fig. 3 also includes copying to and from device and comparison of GPU and CPU cells. ”particle” implementation for GPU here is exactly the same as for CPU. Here it is necessary for debug that computation method are implemented just once for both CPU and GPU.

This gives the speedup of about 8 for Nvidia Tesla M2090 compared to 4 Xeon cores and 40 for Nvidia Kepler compared to 4 Intel Xeon cores .

```
template<class Particle>
class Cell
{
public:
  int i,l,k;      //number of the cell along each dimension
  double hx,hy,hz,tau;   //cell size and time-step
  double *doubleParticleArray;  //particles

//evaluating current produced by the particle "p"
__host__ __device__
void CurrentToMesh(double3 x,double3 x1,double mass,double q_m,
                   double tau,
                        int *cells,CurrentTensor *t1,
                        CurrentTensor *t2,Particle *p);

//form-factor funcion of the PIC1 method: used by CurrentToMesh
__host__ __device__ virtual void  Kernel(CellDouble& Jx,
                   int i11,int i12,int i13,
                   int i21,int i22,int i23,
                   int i31,int i32,int i33,
                   int i41,int i42,int i43,
                   double su,double dy,double dz,
                   double dy1,double dz1,double s1);

//pushing the particles in this cell: the main function of the PIC method
__host__ __device__ virtual
int Move(unsigned int i,int *cells,CurrentTensor *t1,CurrentTensor *t2,
double mass,double q_mass,
                 double *p_control,int jmp_control,
                 CellDouble *Ex1,CellDouble *Ey1,CellDouble *Ez1,
                 CellDouble *Hx1,CellDouble *Hy1,CellDouble *Hz1)

}
```

Figure 2. The main components of the "Cell" class

4. Transition from plasma simulation code to beam simulation code

With inheritance one can transform a code for plasma simulation into a code for relativistic beam simulation. The parts of code being changed are the boundary conditions and initial distribution.

Let us start with the *GPUPlasma* class that implements the PIC method for GPU in application to relativistic plasma physics. The class has special method for field evaluation and particle pushing and initial distribution setting. Moreover, it is already parallel and already ported to GPU. In order to use all this staff with beam simulation a derived class *GPUBeam* is made, as shown in Fig.4. As it can be seen from the picture, the constructor of the GPUBeam class does nothing on its own, it just calls the constructor of the basic class. The parameters of the constructor represent the number of grid nodes, domain size, number of particles per cell, average density, charge-to-mass ratio and time-step.

The use of the defined class *GPUBeam* is quite simple, as shown in Fig.5. The *Initialize* function is called, which actually does all memory allocations and initializations, and then *Step* performs the timestep. All the differences between the plasma simulation and beam interaction simulation are automatically taken into consideration through the mechanism of virtual functions.

4.1. Initial distribution

Since the initial distribution for the beam-beam interaction problem is entirely different, it is Gaussian instead of uniform, and, which is even more important, the particles occupy just a

```
template <class Particle >
class GPUCell: public Cell<Particle>
{
public:

__host__ __device__
    GPUCell(int i1,int l1,int k1,double Lx,double Ly,
    double Lz,int Nx1, int Ny1, int Nz1,double tau1):
       Cell<Particle>(i1,l1,k1,Lx,Ly,Lz,Nx1,Ny1,Nz1,tau1){}

GPUCell<Particle>* copyCellFromDevice(Cell<Particle>* d_src);

//redefined here
int Move(unsigned int i,int *cells,CurrentTensor *t1,
CurrentTensor *t2,double mass,double q_mass,
                 double *p_control,int jmp_control,
                 CellDouble *Ex1,CellDouble *Ey1,CellDouble *Ez1,
                 CellDouble *Hx1,CellDouble *Hy1,CellDouble *Hz1)       ;

}
```

Figure 3. The main components of the "GPUCell" class

```
template <template <class Particle> class Cell >
class GPUBeam: public GPUPlasma<Cell>
{

public:
        GPUBeam(){}
        ~GPUBeam(){}

        GPUBeam(int nx,int ny,int nz,double lx,double ly,double lz,
        double ni1,int n_per_cell1,double q_m,double TAU):
               GPUPlasma<Cell>(nx,ny,nz,lx,ly,lz,ni1,
               n_per_cell1,q_m,TAU){}

        void InitParticles(thrust::host_vector<Particle> & vp);

        void BoundaryConditions();
};
```

Figure 4. The definition of the "simulation domain" (GPUBeam) class for beam simulation.

part of simulation domain, a new function should be made to implement the new distribution. In fact, the *InitParticles* function is made *virtual* in the basic class. The function is called from the *Initialize* method of the basic class. Here, in the derived class, *InitParticles* is redefined.

4.2. Boundary conditions

$$E_x(x,y,z) = \frac{2q(x-x_0)}{h_z((x-x_0)^2+(y-y_0)^2)}, \tag{2}$$

$$E_y(x,y,z) = \frac{2q(y-y_0)}{h_z((x-x_0)^2+(y-y_0)^2)}, \quad E_z = 0, \qquad \vec{H} = [\vec{v},\vec{E}]. \tag{3}$$

The boundary conditions are evaluated according to the formulae (2-3). In the plasma physics problem which was the test suite for the *GPUPlasma* class, the boundary conditions are periodic, thus a very simple CUDA kernel is called for just a few microseconds. Nevertheless, there is a special wrapper functions that invokes this kernel. In the GPUBeam class it is replaced but a more complex, still very fast kernel invoked by the *BoundaryConditions* function. he *BoundaryConditions* function is made *virtual* in the basic class. The function is called from the

```
   GPUBeam<GPUCell> *plasma;

//number of grid nodes, domain size, number of particles per cell,
//    average density, charge-to-mass ratio and time-step
   plasma = new GPUBeam<GPUCell>(100,100,100,1.5,
                          0.01,0.1,1.0,100,1.0,5e-5);

   plasma->Initialize();


   plasma->Step();
```

Figure 5. An example of the GPUBeam class object definition.

field evaluation method of the basic class. Here, in the derived class, *BoundaryConditions* is redefined.

4.3. Performance

The program that is the template implementation of PIC algorithm for beam-beam interaction is called GPU-Beam here. The timings for one timestep with one single Nvidia Tesla GPU are given in table 1 in comparison with the time for one timstep with 1 Intel Xeon core. The run here involves 1 million model particles and a grid of 100^3 nodes. The GPU code for beam simulation made with templates is compared here to the genuine Fortran code (denoted Fortran-Beam) made with the same numerical methods for beam simulation by M.A.Boronina et al. [14]

Table 1. Computation time, ms

Program name	GPU-Beam	Fortran-Beam
Particle pushing time	429.193	0552.253
Field evaluation time	536.250	0180.426
Boundary condition computing time	05.43	2893.723

Thus, with this derived class one can obtain a code for beam simulation. And since the parallel and GPU-related parts of the code were not touched, the parallelization efficiency remains at high level. At present, the GPU beam simulation code works 24 times faster compared to the CPU version (a single Intel Xeon core) [14].

5. The easy replacement of the numerical methods in the plasma simulation code

Finally, inheritance is used to improve accuracy of the simulation. In order to reduce the noise level the different particle form-factors of different shapes were being used.

The form-factor function (sometimes also called the shape function) here have the following form, as shown in figure 6.

1. S1 or PIC form-factor (classical form-factor of the Particle-In-Cell method)

$$R_{S1}(x) = \begin{cases} \dfrac{1}{h}\left(1 - \dfrac{|x|}{h}\right), & |x| \le h, \\ 0, & |x| > h. \end{cases} \tag{4}$$

2. S2 or the parabolic form-factor

$$R_{S2}(x) = \begin{cases} \dfrac{1}{h} - \dfrac{1}{h^3}\left(x^2 + \dfrac{h^2}{4}\right), & |x| \le \frac{1}{2}h, \\ \dfrac{1}{2h^3}\left(\dfrac{3}{2}h - |x|\right)^2, & \frac{1}{2}h < |x| \le \frac{3}{2}h, \\ 0, & |x| > \frac{3}{2}h. \end{cases} \tag{5}$$

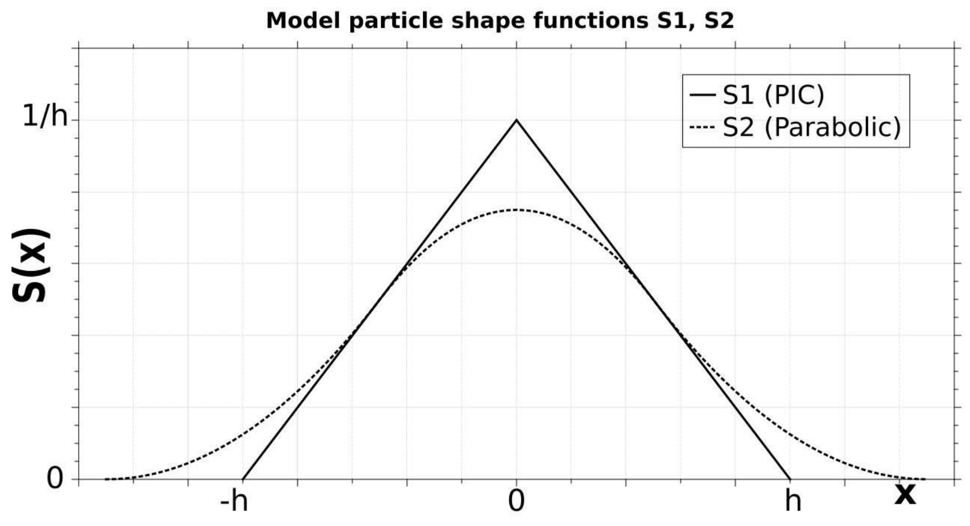

Figure 6. Particle form factors.

The form factors are quickly and simply replaced in the code by the use of C++ templates. The simulation domain is implemented as a template class with the Cell class being template parameter. One of the Cell class methods, *Kernel* in Fig. 2 is the form factor. Its is called by the *CurrentToMesh* function, that computes the impact of the particle to all the components of the current and to the density. The *CurrentToMesh* itself is called by the *Move* function, which is the pusher. It should be noticed here, that the *Kernel* function only should be replaced to introduce the new particle form-factor, all the other Cell class method remain intact.

This method is a virtual one. In such a way, this form factor can be easily replaced by a different one when a derived class is made on the basis of Cell class. Then, when this derived class with the new form factor method is given as a parameter to the simulation domain class, it results in a new code employing the new form factor.

The same could be also done by passing the pointer to the form-factor function to an instance of the Cell class. It may be necessary if virtual functions are not fully supported as it happened with older CUDA versions.

6. Conclusion

In such a way, C++ templates facilitate the quick and easy form-factor replacement and also the transition from one supercomputer architecture to another. At present the physical problems under study are restricted to electrodynamics of charged particles, but in principle the described technology can be applied to all particle method applications.

Still, there is one more question that should be answered. What is the sense of rewriting Fortran codes with some sophisticated C++ tools? First of all, the existing Fortran codes for both problems mentioned in the paper can not give the result with the resolution high enough in reasonable time. By turning them into C++ GPU codes such a resolution can be achieved. Second, there is a lot of other problems of computational electrodynamics that can be solved by the same or similar methods, like glow discharge simulation, solar wind simulation etc. Thus one needs a technology to facilitate creating codes for new physical problems on the basis of the existing codes instead of just writing the code every time from the very beginning for each new physical problem. Such a technology is proposed in the present paper.

References

1. Bidsall Ch., Langdon B. Plasma Physics via Computer Simulation. — McGraw-Hill, 1985.
2. R.W Hockney, J.W Eastwood. Computer Simulation Using Particles CRC Press, 1988
3. Verboncoeur, J.P. OOPIC: object oriented particle-in-cell code. //Plasma Science, 1995. IEEE Conference Record - Abstracts., 1995 IEEE International Conference.
4. Viktor K. Decyk. Skeleton PIC codes for parallel computers. //Computer Physics Communications. Volume 87, Issues 1–2, 2 May 1995, Pages 87–94
5. V.E.Malyshkin, A.A. Tsigulin, Avtometriya, 2003 -in Russian.
6. V.T.Astrelin,A.V.Burdakov,V.V.Postupaev. Generation of Ion-Acoustic Waves and Suppresion of Heat Transport during Plasma Heating by an Electron Beam. Plasma Physics Reports. Vol24, No. 5 pp.414-425.
7. B. I. Cohen, D. C. Barnes, J. M. Dawson, G. W. Hammett, W. W. Lee, G. D. Kerbel, J. -N. Leboeuf, P. C. Liewer, T. Tajima, R. E. Waltz The numerical tokamak project: simulation of turbulent transport. Computer Physics Communications, Volume 87, Issues 1-2, 2 May 1995, Pages 1-15.
8. A. Jaun, K. Appert, J. Vaclavik, L. Villard. Global waves in resistive and hot tokamak plasmas. Computer Physics Communications, Volume 92, Issues 2-3, December 1995, Pages 153-187.
9. J.-L. Gardarein, R.Reichle, F.Rigollet, C. Le Niliot, C. Pochea Calculation of heat flux and evolution of equivalent thermal contact resistance of carbon deposits on Tore Supra neutralizer Fusion Engineering and Design, Volume 83, Issues 5-6, October 2008, Pages 759-765.
10. Vshivkov V.A., Grigoryev Yu.N.,Fedoruk M.P. Numerical "Particle-in-Cell" Methods. Theory and applications VSP, Utrecht-Boston, 249 p., 2002.
11. Langdon, A.B.and B. Lasinski, 1976, Meth. Comp. Phys.16, 327, ed. B. Alder et al.
12. A.V.Snytnikov. Supercomputer simulation of plasma electron heat conductivity decrease due to relativistic electron beam relaxation.// Procedia Computer Science, Volume 1, Issue 1, May 2010, Pages 607-615
13. Note on quantitatively correct simulations of the kinetic beam-plasma instability K.V.Lotov, I.V.Timofeev, E.A.Mesyats, A.V.Snytnikov, V.A.Vshivkov //Phys. Plasmas 22, 024502 (2015)
14. Vshivkov V.A., Boronina M.A. Three-dimensional simulation of ultrarelativistic charged beams dynamics: study of initial and boundary conditions // Math. Mod. 2012. Vol. 24, N 2. P. 67–83.