PNBsolver: A Domain-Specific Language for Modeling Parallel N-body Problems

Ferosh Jacob*, Md Ashfakul Islam

Department of Computer Science, University of Alabama

Weihua Geng

Department of Mathematics, University of Alabama

Jeff Gray, Susan Vrbsky, Brandon Dixon

Department of Computer Science, University of Alabama

Purushotham Bangalore

Computer and Information Sciences,University of Alabama at Birmingham

Abstract

With the advent of multicore processors, parallel computation has become a necessity for next generation applications. It is often a tedious task for domain users to optimize their programs for a specific platform, algorithm, and problem size. We believe that domain users should be freed from this task and they should be equipped with tool support to reuse the optimized solutions written by expert parallel programmers. In this paper, we introduce a two-stage modeling approach that allows domain users to express the problem using domain constructs and reuse the available optimized solutions. This approach has been applied successfully to N-body problems using a domain-specific language called PNBsolver, which allows domain users to specify the computations in an N-body problem without any implementation or platform-specific details. Using the PNBsolver, the domain users are allowed to control the platform and implementation of the generated code. The accuracy and execution time of the generated code can also be fine-tuned based on the parameters provided in PNBsolver. We also show how common N-body interactions are implemented using PNBSolver.

*Corresponding author

Email addresses: `fjacob@crimson.ua.edu`, `mislam@crimson.ua.edu` (Ferosh Jacob*, Md Ashfakul Islam), `wgeng@as.ua.edu` (Weihua Geng), `gray@cs.ua.edu`, `vrbsky@cs.ua.edu`, `dixon@cs.ua.edu` (Jeff Gray, Susan Vrbsky, Brandon Dixon), `puri@cis.uab.edu` (Purushotham Bangalore)

1. Introduction

Even though the first computing machine built by Charles Babbage is believed to be a parallel machine [1], until the last few years parallel computing was reserved for the scientific community. Recently, parallel programming has emerged as an essential skill for software developers, primarily due to two reasons: 1) As the chip industry hit the power wall, manufactures shifted to chip multiprocessors (CMPs), and 2) Introduction of Graphical Processing Units (GPUs) for general-purpose computation (GPGPU) [2]. To utilize the increased computation power available in desktops and powerful GPUs, parallel computing is an essential skill for the next generation of software engineers. Because of the nature of parallel applications, programmers will have to adapt to a new style of programming to use the resources, as well as to maintain and evolve for future applications [3].

Popular parallel programming paradigms include: 1) OpenMP [4] (Shared memory), 2) Message Passing Interface[1] (Distributed memory), and 3) CUDA[2] and OpenCL[3] for GPU platforms. All of these parallel programming paradigms have their advantages and disadvantages for a given problem, based on the execution environment, implementation, and even size of the problem. In the current scenario, to find the optimized execution time for a problem of a given size in a specific platform, a programmer must manually write a parallel version, optimize it and compare the execution time with other platforms. It is often hard for programmers to optimize these programs because their domain of expertise is the problem domain and not the High Performance Computing (HPC) domain. To solve this problem, programmers should be freed from implementation and optimization of parallel versions, but should have the flexibility to express the problem and switch between execution environments (optimal execution time is also a function of problem size). There is no magical tool to convert any sequential program to an optimized parallel program. However, for a restricted domain this can be achieved [5, 6, 7]. In this paper, we address two research questions that are summarized in the following subsections.

1.1. Q1: Separating specification and implementation

Which algorithm on a specific platform can provide the optimized execution for a problem of fixed size? To answer this question, the best way is to develop and execute programs implementing various algorithms in that platform for a fixed size. This leads to the task of rewriting thousands of lines of code. Can we reuse the implementation for another similar problem? This is possible only if the implementation details are separated from the problem specification. Is there a better algorithm which can give the optimized execution for the specific-problem? This cannot be linked with the existing code even if the problem specification is separated from the implementation.

---

[1]MPI, http://www.open-mpi.org/

[2]CUDA, http://developer.nvidia.com/category/zone/cuda-zone

[3]OpenCL, http://www.khronos.org/opencl/

1.2. Q2: Abstract problem specification

How can we maintain and switch between many program versions to provide the optimized execution time? As mentioned earlier, there can be many implementations for a given problem specification. If all such implementations are based on an abstract problem specification, any of these implementations can be used with the given problem specification. If the domain is limited, we can provide optimized solutions in required platforms for each of these implementations.

We used modeling techniques from software engineering to provide separation of specification and implementation and abstraction in a restricted domain. Our solution approach is explained in the following subsection.

1.3. Solution approach: Modeling parallel programs

Models in the context of software modeling can be defined as the abstract representation of a system. Model-driven Engineering [8] is a methodology that makes use of a Domain-Specific Modeling Language (DSML) [9, 10] such that it can: 1) define a system using domain information, 2) analyze the domain, and 3) automatically generate source code or other artifacts by code generation techniques. A model is defined by a set of entities, associations and constraints, commonly called a metamodel [11]. In the case of a Domain-Specific Language (DSL) [12], a metamodel is the grammar defining the DSL. The domain experts define domain problems in a DSL file (an instance of the model), which is capable of capturing information to represent the whole system.

We developed a two-stage technique for modeling parallel programs. In the first stage, the domain experts create a model (DSL file) specifying the problem (this model has no platform-specific or implementation-specific details) and in the second stage, this model is mapped to different platforms and implementations based on user preferences. The main advantage of this two-step modeling technique is that it can be extended to include additional platforms and implementations and is well-suited for a domain like parallel programming.

In this paper, we introduce PNBsolver, which is a DSL based on our two-stage modeling technique. PNBsolver can represent N-body problems without any implementation details and also has the flexibility to allow users to switch between implementations (algorithms) and platforms. Section 2 overviews N-body problems and analyzes a commonly used algorithm in such problems. Section 3 describes PNBsolver from a user's perspective and the implementation details are explained in Section 4. Section 5 gives details of a comparison study of PNBsolver solutions with existing implementations. Additional examples using PNBsolver are shown in Section 6. Related works are reviewed in Section 7 and the paper is concluded in Section 8.

2. N-body problems and tree code algorithm

The N-body problem is a generalized concept of problems related to the physical properties of a system. The system can contain billions of interacting bodies. The classic N-body problem of celestial mechanics originated from

Newton's Principia [13]. The bodies can be giant celestial objects in a solar system or minuscule particles in an atomic system. The physical properties can be motion, energy, charge, momentum, or force. There are many areas in science that use N-body problems (e.g., astrophysics, plasma physics, molecular physics, fluid dynamics, quantum chemistry and quantum chromo-dynamics) [14, 15, 16, 17]. Barnes and Hut [18] introduced a hierarchical $\mathcal{O}(N \log N)$ algorithm to calculate an N-body force equation and was later used in different N-body problems [19, 20, 21]. In this section, the tree code algorithm is reviewed, explaining how the efficient implementations of this algorithm can be reused in N-body computations.

2.1. Tree code algorithm

In the Tree code algorithm for computing N-body interactions, the 3-D space is divided into clusters in a hierarchical way. If the target particle is well-separated from a cluster satisfying the multipole acceptance criterion (MAC), the particle-particle interactions between the target particle and all source targets in the cluster are replaced by a particle-cluster interaction. To this end, a truncated (specified by ORDER, the number of terms of the expansion) Taylor expansion of all particles in the cluster about the center of the cluster is performed. The computation involves two parts: 1) the Taylor coefficients as functions of the distances between the target particle and the center of the cluster, which is computed by an efficient recurrence relation, and 2) the moments as functions of the distance between the center and all particles of the cluster, which is pre-computed and stored. Please refer to [19] for more details.

The tree code algorithm has proved its efficiency for N-body problems [18]. It reduces the computational cost of these problem from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log(N))$. The tree code algorithm is explained in two perspectives: 1) Mathematical formulation; and 2) Source code. In the mathematical formulation, the algorithm is explained with the help of a 3D cube, which itself is divided in all dimensions into subunits recursively. In the source code, this is represented as a tree structure.

2.1.1. Tree code algorithm: A mathematical perspective

$$I_i = \sum_{j=1, j \neq i}^{N} M_j G(x_i, y_j) \tag{1}$$

In a system consisting of N bodies, the interaction of body i with other $N-1$ bodies is given in Equation 1. According to the tree code algorithm, for bodies within a cell/cluster (one of the subunits of the 3-D cube) satisfying the MAC (θ) [22] in Equation 2, their interaction with body i can be calculated more efficiently with controllable errors. In Equation 2, r_c represents the location of the cell center and R represents the distance between the ith body and the cell center. On satisfying the MAC, the more efficient particle-cluster interaction

between body i and bodies inside a cell is given as $I_{i,c}$ in Equation 3 from [19, 23, 24]. Please note that k, x, and y are vectors with three components.

$$\frac{r_c}{R} < \theta \tag{2}$$

$$I_{i,c} \approx \sum_{||k||=0}^{p} \frac{1}{k!} D_x^k G(x_i, y_c) \sum_{y_j \in c} M_j (y_j - y_c)^k \tag{3}$$

$$I_{i,c} \approx \sum_{||k||=0}^{p} TCOEFF(k, G) \times MOMENTS(c) \tag{4}$$

In Equation 3, for each 3D value k, the first term computes the Taylor coefficients for every target body (using recurrence relations) and the second term computes the moment of the cluster, which is only evaluated once for each cluster. In general, Equation 3 can be rewritten as Equation 4. From an implementation perspective, to implement this algorithm for any interaction, programmers only need to choose one of our included interactions or provide the Taylor coefficients for user-specified interactions. In Equations 3 and 4, p represents the ORDER of the Taylor coefficients.

2.1.2. Tree code algorithm: A source code perspective

A general implementation of the tree code has three stages: 1) Tree creation, 2) node values calculation, and 3) Tree traversal. Details of these stages are explained as follows.

Tree creation. At this stage, a tree-based structure is used to maintain a list of bodies. Bodies of the system are added to the root node and if any node has more than a fixed number of bodies (MAXPARNODE), the node is further branched to eight child nodes. This is equivalent to dividing a 3D space recursively until each of the existing subunits has at most MAXPARNODE number of bodies.

Node values calculation. At this stage (usually implemented along with tree creation), the node properties are calculated. This includes calculating the resultant center of the node and arranging inputs such that bodies in the same node are arranged closer for faster access. Note that each intermediate node of the tree contains the approximate interaction information (the second term, moments, in Equation 3) of the sub-units under that intermediate node. Hence, the root node can approximate the entire system.

Tree traversal. In this stage, the interactions are calculated. The bodies are traversed from the root node. If a node satisfies the MAC, traversal is stopped and a node's approximated value is used as the interaction. If MAC is not satisfied, the traversal is continued until it reaches the leaves. At the leaves, interaction is computed by direct summation.

A summary of five tree code implementations collected from different sources [25, 26, 27] implementing different interactions is shown in Table 1. We leave

Table 1: An analysis of existing tree code implementations

Name	Equation	Language	LOC (files)
Total PE	$\sum_{i=1}^{N-1} Q_i \sum_{j=i+1}^{N} \frac{Q_j}{|R_i-R_j|}$	FORTRAN	966 (3)
PE_i	$\sum_{j=i,j\neq i}^{N} \frac{Q_j}{|R_i-R_j|}$	FORTRAN	1006 (3)
$Screened_i$	$Q_i \sum_{j=i,j\neq i}^{N} \frac{Q_j e^{-|R_i-R_j|}}{|R_i-R_j|}$	FORTRAN	1023 (3)
Grav. F_i	$M_i \sum_{j=i,j\neq i}^{N} \frac{M_j(R_i-R_j)}{|R_i-R_j|^3}$	C	1921 (15)
Grav. F_i	$M_i \sum_{j=i,j\neq i}^{N} \frac{M_j(R_i-R_j)}{|R_i-R_j|^3}$	Javascript	908 (6)

out some constants such as dielectric or gravitational constants for simplicity. As shown in the table, all the implementations had roughly 1000 lines of code (LOC).

2.2. Analysis summary

There are parallel implementations of tree code available [28, 29, 30]. In general, there are two parallel implementations: 1) Parallelizing the node interactions, interactions are distributed equally among the parallel instances (threads or processes) and every instance creates their own tree (same tree) for computations; 2) Parallelizing tree creation and node interactions, the tree creation, as well as tree walking, is distributed among the instances. In most cases, direct computation is embarrassingly parallel. Hence, when such a computation is executed in a highly parallelized device like a GPU, the direct computation can outperform the tree code algorithm for a range of size. For a larger size problem, the $\mathcal{O}(N\log(N))$ tree code can outperform the $\mathcal{O}(N^2)$ direct summation. This justifies our combination of parallel direct summation implementation in a GPU, parallel implementation of tree code in a CPU, and parallel implementation of tree code in a GPU. The research questions introduced in Sections 1.1 and 1.2 are revisited in the context of N-body problems in the following subsections.

1. **Separating specification and implementation in N-body problems:** The domain users should be able to specify the N-body problems without any implementation details (e.g., which algorithm to use). They should also be able to provide the implementation details (e.g., desired accuracy, allowed error, platform). However, the problem specification should be logically separated from the other details.

2. **Abstract problem specification in N-body problems:** The problem should be represented at the correct abstraction level. It should not be too high such that code optimizations cannot be applied (e.g., users should be allowed to configure parameters like MAXPARNODE, ORDER, THETA)

```
1  kernel Force in R
2
3  // Kernel declarations
4    vector F
5    scalar M
6    constant K=1
7
8    /*
9     * Read positions and mass from file
10    * formatted as <x y z>m
11    *
12    */
13    read "<R_1,R_2,R_3>M", "data.dat"
14
15    //Actual computation
16    F=K*M SUM(M*R/(R_*R_*R_))
17
18    // Write force to file
19    write "F_1,F_2,F_3", "out.dat"
20
21 endkernel
22
23 // Generate CUDA code for force kernel
24 generate CUDA ACCURATE Force.
```

Figure 1: Gravitational force kernel in space R using PNBsolver

and not too low level (e.g., users might not want to specify the bodies as charge or mass).

We designed PNBsolver to address these questions. PNBsolver is introduced in the following section from a user's perspective.

3. Working with PNBsolver

The equation to calculate the gravitational force for each body in an N-body system with masses M_i and positions R_i is shown in Equation 5. An equivalent representation of the force using PNBsolver is shown in Figure 1. In the figure, the Force kernel is defined in space R, more specifically a three dimensional position vector $\mathbb{R}^3$. A "pnb" (parallel N-body) file can be logically divided into three sections: 1) Declarations, 2) Calculations, and 3) Generation. Each file section is explained in the following subsections.

$$F_i = KM_i \sum_{j=1, j \neq i}^{N} \frac{M_j(R_i - R_j)}{|R_i - R_j|^3} \qquad (5)$$

3.1. Declarations

Every variable used in the calculation section should be declared before usage. PNBsolver identifies three types of variables: 1) vector, 2) scalar, and 3) constants. Vectors and scalars are linked to every body in the system. The total force that acts on a body or position of a body at an instance are examples of vectors; the mass or charge of a body are example of scalars. The constant

7

```
1    //A cluster not satisfying MAC
2
3    //Variable declarations
4    double dx, dy, dz, dist, distsq, temp1;
5
6    //Calculating distance variables
7    dx = R[i] - tarpos[0];
8    dy = R[i] - tarpos[1];
9    dz = R[i] - tarpos[2];
10   distsq = dx * dx + dy * dy + dz * dz;
11
12   //Avoiding i==j
13   if(distsq > 0.0f){
14           dist = sqrt(distsq);
15
16           //Calculating specified variables
17           temp1 = dist * dist * dist;
18
19           //Actual calculation
20           Forcex = Forcex + M[i] * dx / temp1;
21           Forcey = Forcey + M[i] * dy / temp1;
22           Forcez = Forcez + M[i] * dz / temp1;
23   }
```

Figure 2: Code generated for the tree code algorithm with MPI/OMP

```
1    //Variable declarations
2    TYPE dx, dy, dz, distsq, invdist, temp1;
3    //Calculating distance variables
4    dx = Ri.x - tarpos.x;
5    dy = Ri.y - tarpos.y;
6    dz = Ri.z - tarpos.z;
7    distsq = dx * dx + dy * dy + dz * dz;
8
9    if(distsq > 0.0f){
10           //Using CUDA Fast functions
11           if(ACCURATE)
12               invdist = rsqrt(distsq);
13           else
14               invdist = rsqrtf(distsq);
15
16           //Calculating specified variables
17           temp1   = invdist * invDist * invDist;
18
19           //Actual calculation
20           Force.x += dx * tarpos.w * temp1;
21           Force.y += dy * tarpos.w * temp1;
22           Force.z += dz * tarpos.w * temp1;
23   }
```

Figure 3: Kernel code generated for direct summation with CUDA

data type is used to specify the constants in the calculation and also the global properties of the system. The gravitational constant or the dielectric constant are examples of constant data types. The scalar and the constant variables can be initialized along with their declaration, as shown in Figure 1 (line 6). In the case of a scalar, all the N bodies will be initialized with the given value.

3.2. Calculations

Actual calculation for the kernel is specified in this section. As shown in Figure 1 (line 16), the calculation involves two expressions. The first expression, $K * M$ corresponds to the KM_i in Equation 5. For discussion, this expression is called the outer expression and the second expression is called the inner expression. PNBsolver expects the inner equation to perform the SUM operation. In addition to the variables declared, PNBsolver identifies the position vector in both inner and outer expressions. For the Force kernel, the position vector is R (line 1). In the outer expression, R represents the actual position, and in the inner expression R gives the relative position with the iteration value. For calculations, the magnitude of a vector can be obtained by adding the "_" to the name of the variable (R_ in line 16). PNBsolver achieves separation of calculations within the expression statement with "(" brackets. The expression used in the brackets are computed to a temporary variable before the final computation and the temporary variable is replaced with the occurrences of the expression in the final expression statement. This is done to optimize the computation.

Read and write statements: This section helps PNBsolver to integrate with existing programs or functions. In this section, the vectors and scalars that are declared are initialized. The read statements read the values from a file to the variables and write statements write the computed results to a file. Both of these commands take two parameters: 1) Format and 2) File name. The format argument specifies how values for each body are formatted in a line. A single kernel can have many read and write statements. As an example, if masses were specified in a different file, another read statement could be added before the calculation statement. For reading and writing formats, the three co-ordinates of vectors can be accessed by adding "_1, _2," and "_3," respectively, to their names (line 13 and line 19).

3.3. Generation

The first two sections can express the N-body problem efficiently without any implementation details. This section controls the implementation and code generation. The default code generation language is "C" and the algorithm is selected based on the mode and target parallel programming paradigms. PNBsolver supports CUDA, OMP, and MPI parallel programming paradigms. There are three modes for each paradigm: 1) ACCURATE, 2) AVERAGE, 3) FAST. The program runs faster when a mode is changed from ACCURATE to FAST and a program gives more accurate results when the mode is changed from FAST to ACCURATE. More details about how algorithms are selected based on the mode and paradigms are explained in the implementation section.

A section of code generated for the tree code algorithm and CUDA direct implementation is shown in Figures 2 and 3. This code computes the Force for the body located at R_i due to another body located at tarpos. Both implementations have pre-defined variables, dx, dy, dz, distsq. The distsq calculates the magnitude of the distance. The if statement could be avoided if

9

we set $M_i = 0.0$ before the computation, but our current implementation generates the `if` statement. However, if there is a softening factor to the magnitude, the PNBsolver parser identifies this and removes the `if` statement. An example for such a case is explained in Figure 15.

In Figures 2 and 3, `temp1` (line 4 and line 2, respectively) is a variable created by using brackets in the expression statement. As shown in the figure, the calculation is implemented in two different ways for the two versions (line 17 in Figures 2 and 3). For the CUDA implementation, the type of the variable is defined based on the mode, for faster implementations `float` is used and for accurate implementations `double` is used. Hence, the type of the variable is defined as a C++ template variable `TYPE` (variable is removed by actual code during compilation). `ACCURATE` is another template variable to tune optimization and speed. For distance, because CUDA has a fast math function, to perform the square root and inverse, we use that function (Figure 3), instead of the square root followed by the division operator (Figure 2) as in the CPU code. Use of the `float4` data type for the position and mass is another GPU optimization. If the variable `FAST` is set, CUDA fast functions (e.g., `rsqrtf,__expf(x)`) are used.

4. Implementation details of PNBsolver

A block digram showing the implementation of PNBsolver is illustrated in Figure 4. The "pnb" file is passed to the parser and the parser identifies the mode and platform for selecting the proper template. The parser also generates the kernel code which is further optimized and given to the code integrator. The optimizations applied at this stage are very general and they are not coupled to any specific platform. Removing the brackets with new temporary variables, finding expressions that are repeated, and checking whether softening is applied are examples of optimizations at this stage. The code integrator merges the kernel code to the template and fills the template values. This includes updating function declarations and parameters, function calls and arguments, and declaring type variables and initialization. After this, platform-specific optimizations are applied. Rewriting functions with CUDA fast math functions is an example of such an optimization.

The project is implemented in Java and ANTLR[4] is used for implementing the "pnb" file parser. The template store is implemented using StringTemplate[5]. Two important parts of the implementation are "pnb" file parsing and code generation using templates. The code generation varies based on the mode and platform selected in the "pnb" file. However, the code generated will be either a parallel direct summation implementation or a parallel tree code implementation. The PNBsolver Parser, PNBsolver code generator, and Modes of operation are explained in the following subsections.

---

[4]ANTLR, `http://www.antlr.org/`
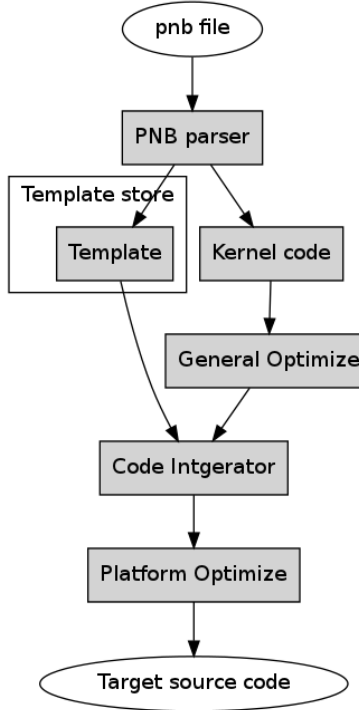[5]StringTemplate, `http://www.stringtemplate.org`

Figure 4: Block diagram showing the implementation of PNBsolver

4.1. PNBSolver Parser

The simplified EBNF grammar for the PNBsolver language is shown in Figure 5. As shown in the figure, the file allows one or more kernels, but code is generated for only one kernel. This is designed to support future extensions, where the output of a kernel can be passed to the input of another kernel. After parsing, the expression statement is captured in two expression objects, the inner expression object and the outer expression object. All expressions in a "pnb" file are made of variables defined in the declaration section. To use a constant inside the expression statement, it should be declared as a constant and use the variable name in the expression statement. As a rule, the read statements should be defined before the expression statement, and write statements after the expression statement. This seems logical as a program usually requires reading before the calculation and writing after the calculation. The EBNF rules for `variabledeclaration`, `writestmts`, and `parameters` are not shown in the figure to provide clarity.

4.2. PNBsolver code generator

The PNBsolver parser can identify the inputs and outputs for the kernel. The code generator inserts declarations and initialization (if any) into the program.

```
1
2  grammar PNBsolver;
3
4  content
5    : kernels   execute "." EOF
6    ;
7
8  kernels
9    : ("kernel"  ID "in" ID declarations readstmts
10            expressionstmt writestmts "endkernel")+
11   ;
12
13 declarations
14   : (type variabledeclaration)+
15   ;
16
17 execute
18   : "generate" platform ("[" parameters "]")? mode ID
19   ;
20
21 platform
22   : "CUDA"|"OMP"|"MPI"|"OCL"
23   ;
24
25 mode
26   : "ACCURATE"|"AVERAGE"|"FAST"
27   ;
28 type
29   : "vector"|"scalar"|"constant"
30   ;
31
32 expressionstmt
33   : IDENTIFIER "=" (expression)? "SUM" expression
34   ;
35
36 expression
37   : multidiv( "+" multidiv|   "-" multidiv )*
38   ;
39
40 multidiv
41   : atom ("*" atom |"/" atom)*
42   ;
43
44 atom
45   : IDENTIFIER
46   | "(" expression ")"
47   | "exp" expression
48   | "pow" "("expression "," NUMBER ")"
49   ;
50
51 readstmts
52   : ("read" STRING "," STRING)*
53   ;
```

Figure 5: Simplified EBNF grammar of PNBsolver

The core computation code is generated as shown in Figures 2 and 3. These two code sections are inserted into the template identified by mode and platform. To add a new algorithm, we have defined a new template and configured the parser to route through a different code generator. The same approach can be used to support a language other than C. In Section 5, we generate FORTRAN code using a FORTRAN code generator. There are four "C" templates available

```
1 struct tnode {
2     int numbodies, begin, end;
3     double x_min, y_min, z_min, x_max, y_max, z_max;
4     double x_mid, y_mid, z_mid, radius;
5     int level, children, exist_ms;
6     double ***moments;
7     struct tnode* child[8];
8 };
```

Figure 6: Structure `tnode` in the template

in the template store and are explained in the following subsections.

4.2.1. Parallel CPU (OpenMP and MPI) tree code templates

The tree code used for PNBsolver is adapted from [19]. We developed parallel versions of the tree code in MPI and OpenMP. More details about the speedup, error and execution time are included in Section 6. To complete a CPU tree code template, we need: 1) Taylor coefficients, 2) Direct implementation, and 3) Tree code parameters such as MAXPARNODE, ORDER (taylor coefficient order) and THETA (MAC). PNBsolver sets default values for all of these parameters, but this can be customized by the user as shown later in Figure 10.

Providing Taylor coefficients: The parameter value TAYLOR is used to find the Taylor coefficients. The PNBsolver code generator reads the file specified as the parameter value of TAYLOR and looks for a function with signature `comptcoeff (struct tnode *t)`, where `tnode` is a structure representing the tree node. In this function, the position of an interacting body can also be accessed. The declaration of structure `tnode` is shown in Figure 6. The values of the Taylor coefficients should be set in a 3D array of size ORDER. In addition to the `comptcoeff` function, users can add two more functions: `setup()` and `teardown()`, for declaring values that might be required for the computation. These two functions are executed only once before and after the tree creation, while the `comptcoeff` function is executed for every target that satisfies THETA. All of these functions can access the user-specified parameters and some global parameters which include NUMPARS (number of bodies). If PNBsolver is executed with a TAYLOR parameter, it will look for a file specified as the value. If PNBsolver cannot find the file in the current directory, a file having the above three function signatures will be created. PNBsolver uses Taylor coefficients mentioned in [19] if TAYLOR is not specified and it is found effective for Force and Potential calculations.

4.2.2. Parallel GPU template for direct computation

For the GPU implementation using the direct summation, we have used two optimization techniques: 1) Tiling (partitioning of the computation domain into smaller tiles) with shared memory, and 2) Loop unrolling (replacing a loop with similar independent statements). The effect of these optimizations for Coulomb Potential is shown in Figure 7. In the figure, the direct version is the case when the kernel code generated was executed as a CUDA kernel without
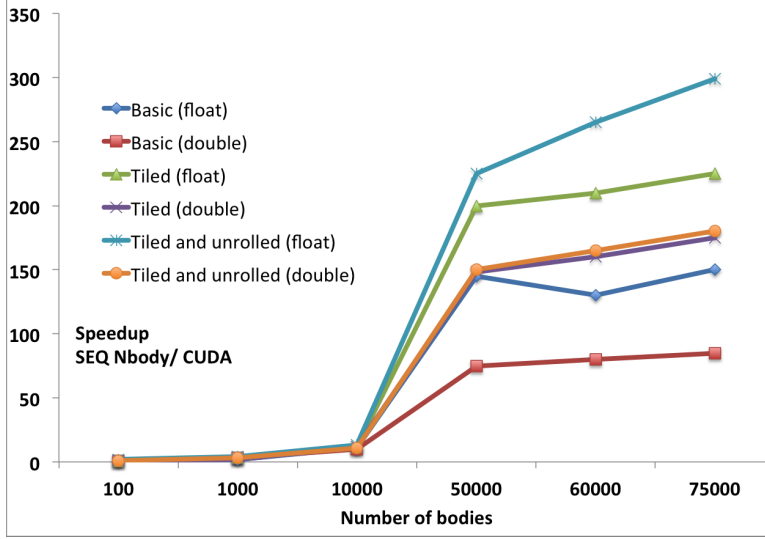
Figure 7: Effect of optimization techniques (Tiling and unrolling) for Coulomb potential

any modifications on a Tesla M2070. All kernels in the GPU template can be executed for `double` and `float` mode. In the figure, Speedup is defined as the ratio of execution time of sequential implementation of Coulomb potential on a CPU to that of the CUDA implementation. Every reading is an average value of at least three of the same executions.

4.2.3. Parallel GPU tree code template

The implementation we use for GPU tree code is adapted from [29]. Our current implementation only supports zeroth ORDER; hence, there is no need to specify Taylor coefficients for tree code implementation.

4.3. Modes of operation

There are three modes of operation for PNBsolver: 1) ACCURATE, 2) AVERAGE, and 3) FAST. The execution time of the problem decreases from ACCURATE to FAST and error decreases from FAST to ACCURATE. This is achieved by varying the parameters ORDER and THETA. The effect of these two parameters for two problems executed for a PNBsolver OpenMP solution for a fixed size is shown in Figures 8 and 9. As seen from the figure, as the ORDER increases, error is less but the program takes longer to finish execution. In the case of THETA, the program finishes faster for higher values of THETA, but has increased errors due to lowered threshold for using particle-cluster interaction. Unless specified, the THETA value is set to 0.5 as a default.

PNBsolver uses these tuning parameters to allow programmers to control the error and execution of generated programs. The parameter values set through a mode can be overridden by specifying the parameter in the `generate` clause.
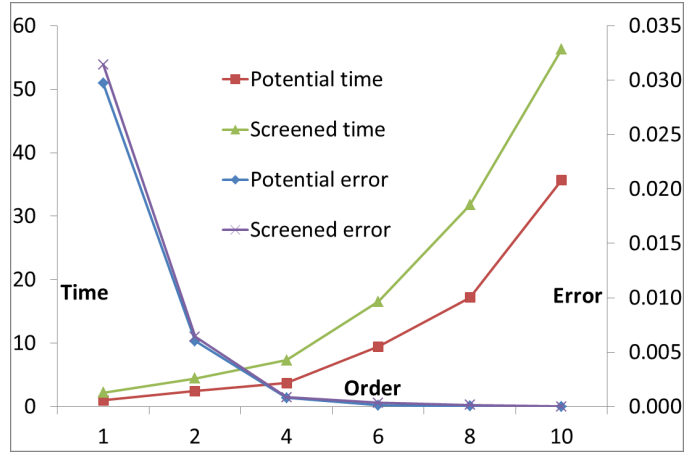
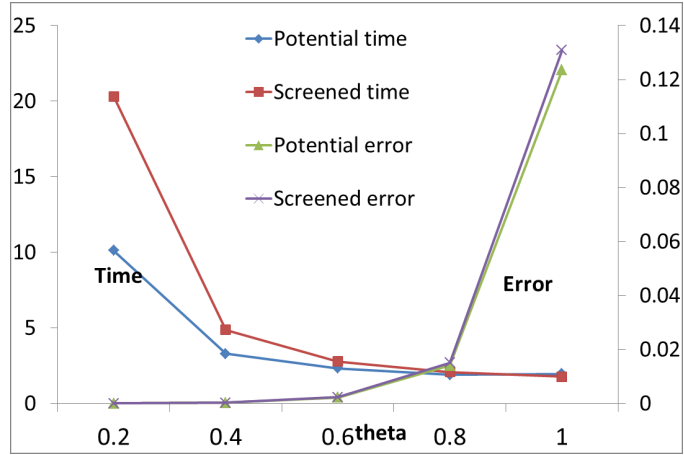Figure 8: Effect of ORDER in error and time

Figure 9: Effect of THETA in error and time

Table 2: Three modes of operation and corresponding tuning parameters

Platform	Mode	Algorithm	Parameters
GPU (CUDA)	ACCURATE	DIRECT	DOUBLE
	AVERAGE	DIRECT	FLOAT
	FAST	TREE	
CPU (OpenMP & MPI)	ACCURATE	TREE	ORDER=6
	AVERAGE	TREE	ORDER=4
	FAST	TREE	ORDER=1

The value of relevant parameters for the three modes are shown in Table 2. As

shown in the table, GPU implementations (currently only supported in CUDA) in the ACCURATE mode gives the most accurate results using direct computation with `double` precision for all computation variables, the AVERAGE mode also uses direct computation, but with `float` precision for all variables. Hence, the AVERAGE mode gives faster execution time than ACCURATE mode, but less accurate results. In the FAST mode, CUDA implementations make use of the Tree code algorithm to give the fastest execution time, but even less accurate results compared to the AVERAGE mode. In CPU implementations (MPI and OpenMP), the Tree code algorithm is used in all three cases, such that the speed and accuracy are controlled by tuning the ORDER variable. When order is set at a higher value, the algorithm provides accurate results, but execution time is longer, as shown in Figure 8.

5. PNBsolver and handwritten code comparison

In this section, we compare the execution time of two kernels generated by PNBsolver with that of handwritten code from expert programmers.

5.1. Yukawa potential (screened Coulomb potential) calculation

The electrostatic Yukawa energy potential at the ith charge Y_i, due to the N mutually interacting charged particles with coordinates R and partial charges $Q_i(i = 1, N)$, is calculated as shown in Equation 6.

$$Y_i = Q_i \sum_{j=1, j!=i}^{N} \frac{Q_j e^{-|R_i - R_j|}}{|R_i - R_j|} \qquad (6)$$

The PNBsolver representation for Yukawa interactions is shown in Figure 10. The code is generated for an MPI version with parameters: TAYLOR and MAXPARNODE (line 20). Since the code is generated for AVERAGE mode, from Table 2, ORDER=4 and default value of THETA is 0.5. It is assumed that the Taylor coefficient implementation is available in a file "comptt.c"

Comparison: We compared the execution time of MPI code generated by PNBsolver to to Li et al. [23]. To make the comparison as accurate as possible, 1) we set the same parameter values for the version in test, 2) used a FORTRAN template instead of the C template, and 3) MPI code generated by PNBsolver is executed with one process. The execution times were recorded for the size varying from 1000 (1k) to 500k. The execution times are shown in Figure 11, where FORTRAN screened [23] is the version in test (FORTRAN Screened (Li) in figure) and FORTRAN screened (PNBsolver) is the generated version. As seen in Figure 11, generated code executed nearly as fast as the sequential hand written code.

5.2. Gravitational acceleration

The equation to calculate gravitational acceleration is similar to Equation 5 without the M_i in the outer loop. The PNBsolver code for Force is shown in

16

```
1 kernel Yukawa in R
2
3 // Kernel declarations
4   scalar Y
5   scalar Q
6   constant KAPPA=1.0
7
8   read "R_1,R_2,R_3,Q", "data.dat"
9
10  //Actual computation
11  Y=Q SUM(Q*exp(-KAPPA*R)/R)
12
13  // Write Yukawa potential to file
14  write "Y", "out.dat"
15
16 endkernel
17
18 generate MPI AVERAGE Yukawa
19 //Parameters
20   [TAYLOR=comptt.c, MAXPARNODE=100].
```

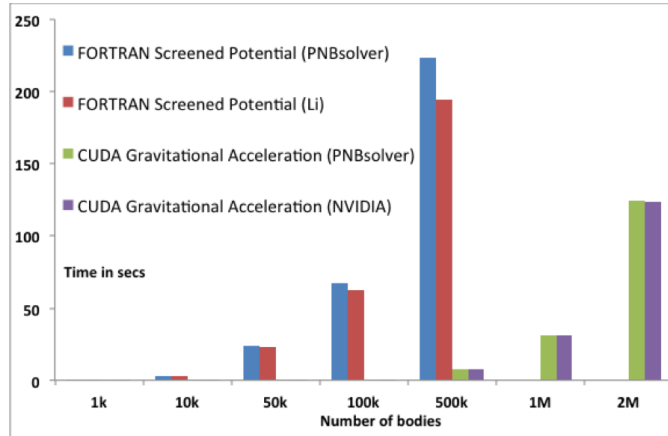Figure 10: Screened (Yukawa) potential kernel in space R using PNBsolver

Figure 11: Execution time comparison of hand-written code with generated code

Figure 1. We compared the PNBsolver acceleration with an acceleration implementation from the NVIDIA CUDA installation package[6]. We slightly modified the NVIDIA code to make more precise comparisons. For all GPU plots, the execution time includes: 1) Time to allocate memory in GPU, 2) Time to transfer memory from CPU to GPU, 3) GPU execution, and 4) Copying results back to CPU. We added code to pass the input values to the GPU before execution and also code to read back after execution. However, we used the `nbody_kernel.cu` file, which performs the actual implementation for comparison.

Comparison: We compared the execution time of the NVIDIA implemen-

---

[6]CUDA SDK, http://developer.nvidia.com/cuda-cc-sdk-code-samples

```
1  kernel Potential in R
2
3    scalar Y, Q
4
5    read "R_1,R_2,R_3,Q", "data.dat"
6
7    //Actual computation
8    V=Q SUM(Q/R)
9
10   // Write Yukawa potential to file
11   write "V", "out.dat"
12
13 endkernel
14
15 generate OMP AVERAGE Potential.
```

Figure 12: Coulomb potential kernel in space R using PNBsolver

tation of gravitational acceleration with the generated code of PNBsolver. The number of bodies were varied from 1k to 2M. For PNBsolver, the CUDA direct summation was used with single point precision. The execution times of both of these versions are shown in Figure 11. As seen from the figure, PNBsolver can generate code that is as fast as the CUDA implementation.

6. More N-body problems

In this section, we explain the speedup and the errors (numerical deviation from the expected results) of a few N-body problems in three different architectures: 1) Shared memory (OpenMP), 2) Distributed memory (MPI), and 3) GPU (CUDA). The analysis of OpenMP and MPI programs is given in the following subsection and CUDA programs is provided in subsection 6.2.

6.1. Analysis of OpenMP and MPI programs

For the OpenMP and MPI program analysis, in addition to the Force (Equation 5) and Yukawa potential (Equation 6), we generated code for the Coulomb potential which is calculated as shown in Equation 7. The PNBsolver representation of the Coulomb potential is shown in Figure 12.

$$V_i = Q_i \sum_{j=1, j \neq i}^{N} \frac{Q_j}{|R_i - R_j|} \tag{7}$$

6.1.1. Execution environment

The MPI and OpenMP programs were executed in a cluster with eight nodes. The programs read positions and charges from a file of 5 million records. We modified the generated code to include a direct implementation on every program to verify the accuracy of the implementation. The programs were executed for a number of bodies ranging from 5k to 500k.
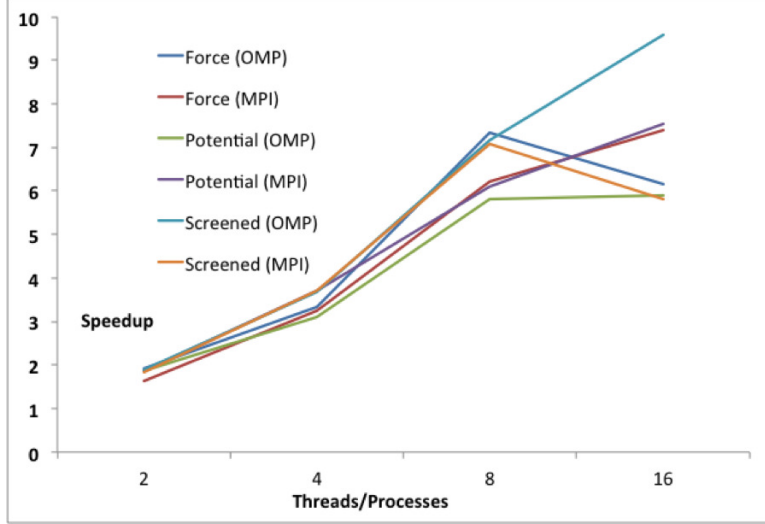
Figure 13: Speedup of tree code programs with sequential tree code

6.1.2. Speed up of MPI and OpenMP programs

To evaluate our parallel implementations of the tree code algorithms in OpenMP and MPI platforms, the speedup (ratio of the sequential tree code execution time to that of the parallel tree code), as shown in Figure 13, is plotted against the number of threads for OpenMP and processes for MPI programs. As shown in the graph, our parallel implementation gave a linear speedup until eight (number of nodes in the cluster) threads/processes. For more computation intensive equations, executing with more threads or processes than the available resources can slow down the program to a considerable rate as in the case of Screened Coulomb potential in Figure 13. In Figure 14, the speedup is plotted against the number of bodies, where relative speedup is defined as the ratio of execution time of a sequential program to that of the tree code program. In this case, super linear speedup is observed with the tree code algorithm.

6.2. Analysis of CUDA programs

For CUDA analysis, we included the expression shown in Equation 8 [24] for computing vortex sheet motion in 3D flow. The PNBsolver file is shown in Figure 15.

$$v_i = -\sum_{j=1, j \neq i}^{N} \frac{W_j}{(|R_i - R_j|^2 + \delta^2)^3} \tag{8}$$

6.2.1. Execution environment

We executed the generated code on a Tesla M2070 GPU card. The generated code was modified to verify the results for accuracy with the direct implemen-
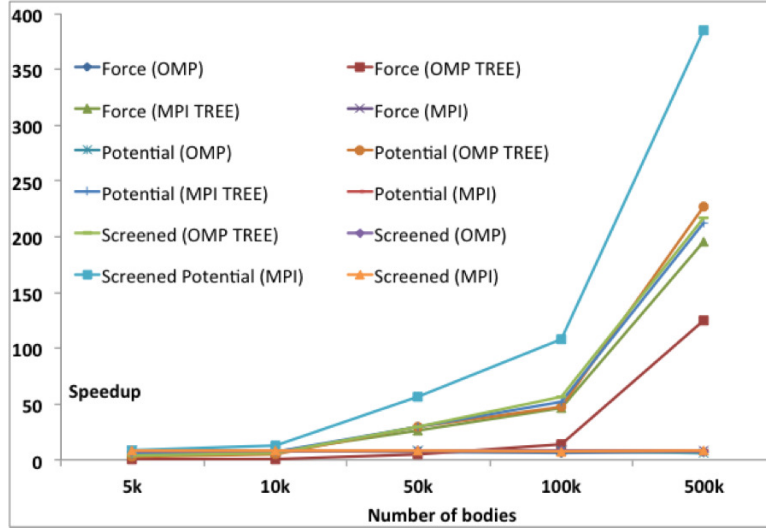
Figure 14: Speedup of tree code programs with sequential direct code

```
1 kernel Vortex in R
2
3    scalar W
4    vector V
5    constant DELATSQ=0.00125
6
7    read "R_1,R_2,R_3,W", "data.dat"
8
9    V= SUM(W/((R_*R_+DELTASQ)(R_*R_+DELTASQ) (R_*R_+DELTASQ)))
10
11   write "V_1,V_2,V_3", "out.dat"
12
13 endkernel
14 generate CUDA AVERAGE Vortex.
```

Figure 15: Coulomb potential kernel in space R using PNBsolver

tation. Similar to the OpenMP and MPI implementations, the generated code read from a file of five million records and programs were executed from a number of bodies ranging from 10K to 5M values.

6.2.2. Speedup of CUDA programs

The OpenMP direct implementation of the Coulomb potential for five million bodies executed with eight threads took almost two days to finish execution, while the CUDA FAST version finished in less than a few seconds. For plotting, we defined relative speedup as the ratio of execution time of CUDA direct double or float to that of CUDA tree code. The speedups for the four equations are shown in Figure 16 and relative errors recorded in the computation are shown in Figure 17. A relative error is defined as the ratio of the highest absolute error to the highest value in direct summation. As shown in Figures 16 and
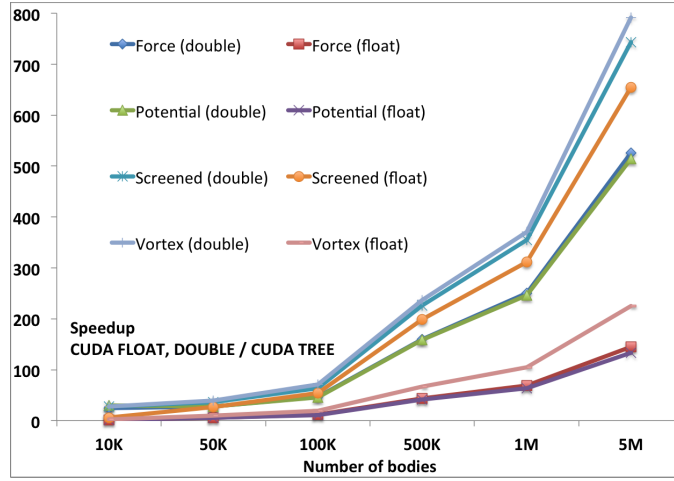
20

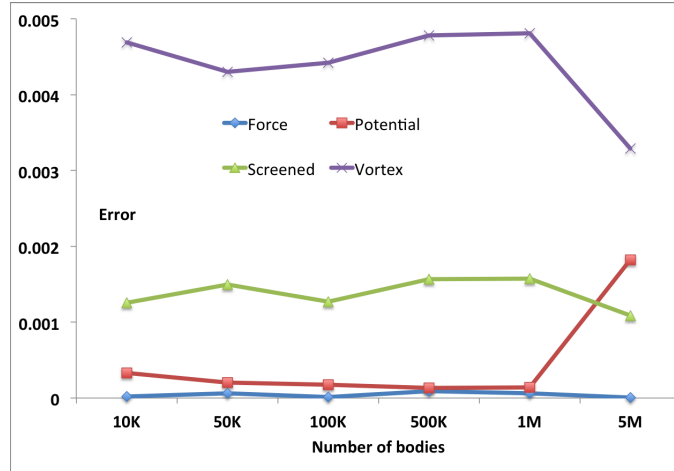Figure 16: Speedup of CUDA tree code with direct CUDA

Figure 17: Error in CUDA tree code programs

17, PNBsolver can generate CUDA programs that are fast and more accurate. From Figure 16, it can be seen that the speedup of float is less than double. This is because the execution time of float is less compared to double. Therefore the speedup with the tree code is less.

7. Related works

We have classified the related works into three categories, as explained in the following subsections.

21

7.1. N-body solutions

A major advance in N-body simulation was the introduction of the GRAPE (Gravity PiPE) series of special-purpose computers [31]. But recent works show that comparable speedup can be achieved through GPUs [32]. AMBER [33] is a package of computer programs written in FORTRAN to simulate the structural and energetic properties of molecules. AMBER used a non-bonded cutoff sphere for applying approximation in N-body energy equations. AMBER 4.0 adds parallelism to modules of the program. NAMD [34] is a parallel simulation program written in C++ and used to simulate the behavior of bio-molecular systems. NAMD2 uses Converse (a portable runtime framework) to support interoperability between different parallel paradigms. FMM-Yukawa [35] is a FORTRAN program package for fast evaluation of the screened Coulomb interaction of N particles. Programs can execute in linear time for nearly uniform particle distributions. Compared to PNBsolver, these programs have targeted a specific field and to a specific N-body problem.

MATLAB[7] is a high performance language for technical computing where solutions and problems are expressed in mathematical notations. Being a general tool, it is highly improbable to have the expressiveness or performance as compared to the domain-specific PNBsolver, which can be extended to support more algorithms by adding templates and allow programmers to fine-tune these algorithms based on their specific problem.

7.2. Tree code algorithms

Cheng et. al. [36] used an adaptive quad tree structure to develop an adaptive fast solver for a modified Helmholtz equation in two dimensions. Ossmani et. al. [37] used a tree based data structure to evaluate efficiency of a multi-scale hybrid grid-particle vortex method. Krasny et. al. [19] used a tree code algorithm to compute non-bonded particle-cluster and cluster-cluster interaction. They also used an oct-tree data structure and implemented different tree traversing techniques specific to different boundary conditions. Krasny et. al. [20] implemented the tree code in Cartesian coordinates and used a recurrence relation to compute the Taylor coefficient for evaluating sums of multi-quadric radial basis function (RBF). Xu et. al. [21] used a tree code algorithm for calculating polarized Coulomb interaction of an N-body system. Li et. al. [23] used the algorithm for evaluating electrostatic potential of screened Coulomb interaction using a new recurrence relation. Baczewski et. al. [38] presented an Accelerated Cartesian Expansion (ACE) method to evaluate periodic Helmholtz, Coulomb and Yukawa potential using $\mathcal{O}(N)$ operations and $\mathcal{O}(N)$ storage using different approximations in both tree building and tree traversing. All of these works represent different variations of the original tree code algorithm. Because PNBsolver is more focused on the approach than to a specific algorithm, PNBsolver can support any of these.

---

[7]MATLAB, http://www.mathworks.com

7.3. Modeling in parallel programming

There have been several modeling efforts in solving HPC applications [39, 40]. Liszt [5] is a DSL for constructing mesh-based solvers that can generate MPI, pthreads, and CUDA code. Similarly, SPIRAL [7] generates high performance code of Digital Signal Processing (DSP) transforms. Popular parallel paradigms such as MapReduce are available in cloud environments through similar approaches [41]. CODE [42] and GASPARD [43] are two graphical programming environments unlike PNBsolver, which is a modeling tool. PPModel [44, 45] is a graphical modeling tool that helps programmers to identify parallel sections in a program and re-target to a different platform. PPModel is a general tool and hence can only transform the code to the target platform without any optimizations.

8. Future work and Conclusion

PNBsolver has been implemented to show that our two-stage modeling approach can be applied to parallel programs. In the implementation, tree code algorithms are generated for the specified order for MPI/OpenMP programs, and zero order for CUDA programs. Extending a GPU program also for the specified order is a future direction of work. Another possible direction would be adding more templates for supporting other platforms and paradigms like OpenCL.

N-body problems involve computing interactions of multiple bodies in a 3D space. The existing solutions work on a specific problem, specific algorithm, and specific platform. In this paper, we introduce a domain-specific language called PNBsolver, which can express the computations in an N-body problem in a manner that is oblivious to their implementation, platform, and algorithmic details without compromising the execution time. PNBsolver can be executed in three modes: 1) FAST, 2) AVERAGE, and 3) ACCURATE modes in three popular parallel programming paradigms (CUDA, OpenMP and MPI). Modes are defined based on the target platform (e.g., ACCURATE mode on a GPU uses a double precision algorithm in CUDA, and the ACCURATE mode on a CPU uses a 10-order tree code algorithm). The speedup and errors were analyzed for four commonly seen N-body interactions. We compared the execution time of the generated code with that of handwritten code by expert programmers to show that the execution time is not compromised. PNBsolver was successfully applied to five N-body problems and their corresponding speedups and errors were plotted.

23

References

[1] L. F. Menabrea, "Sketch of the analytic engine invented by charles babbage," *Bibliothéque Universelle de Genéve*, no. 82, September 1842.

[2] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[3] A. R. Brodtkorb and T. R. Hagen, "A comparison of three commodity-level parallel architectures: Multi-core cpu, cell be and gpu," in *Proceedings of the International Conference on Mathematical Methods for Curves and Surfaces*, Tønsberg, Norway, July 2008, pp. 70–80.

[4] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., 2001.

[5] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan, "Liszt: a domain specific language for building portable mesh-based pde solvers," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Seattle, Washington, 2011, pp. 9:1–9:12.

[6] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley, "An overview of the trilinos project," *ACM Transactions on Mathematical Software*, vol. 31, no. 3, pp. 397–423, 2005.

[7] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE, special issue on Program Generation, Optimization, and Adaptation*, vol. 93, no. 2, pp. 232–275, 2005.

[8] D. Schmidt, "Model-driven engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25–32, 2006.

[9] A. Lédeczi, A. Bakay, M. Maróti, P. Völgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai, "Composing domain-specific design environments," *IEEE Computer*, vol. 34, no. 11, pp. 44–51, 2001.

[10] J. Sprinkle, M. Mernik, J.-P. Tolvanen, and D. Spinellis, "Guest Editors' Introduction: What kinds of nails need a Domain-Specific hammer?" *IEEE Software*, vol. 26, no. 4, pp. 15–18, 2009.

[11] C. Atkinson and T. Kuhne, "Model-driven development: A metamodeling foundation," *IEEE Software*, vol. 20, no. 5, pp. 36–41, 2003.

[12] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Computing Surveys*, vol. 37, no. 4, pp. 316–344, 2005.

[13] F. Diacu, "The solution of the n-body problem," *The Mathematical Intelligencer*, vol. 18, no. 3, pp. 66–70, Nov 1996.

[14] K. M. Watson, "Multiple scattering and the many-body problem - applications to photomeson production in complex nuclei," *Phys. Rev.*, vol. 89, no. 3, pp. 575–587, Feb 1953.

[15] R. Jastrow, "Many-body problem with strong forces," *Phys. Rev.*, vol. 98, no. 5, pp. 1479–1484, Jun 1955.

[16] M. Sasai and P. G. Wolynes, "Stochastic gene expression as a many-body problem," *Proceedings of the National Academy of Science, PNAS*, vol. 100, no. 5, pp. 2374–2379, Mar 2003.

[17] J. Carlson, J. Kogut, and V. R. Pandharipande, "Quark model for baryons based on quantum chromodynamics," *Phys. Rev. D*, vol. 27, no. 1, pp. 233–243, Jan 1983.

[18] J. Barnes and P. Hut, "A hierarchical o(n log n) force-calculation algorithm," *Nature*, vol. 324, pp. 446–449, Dec 1986.

[19] R. Krasny and Z.-H. Duan, "Treecode algorithms for computing nonbonded particle interactions," in *Computational Methods for Macromolecules: Challenges and Applications*, T. Schlick and H. H. Gan, Eds. Springer Berlin Heidelberg, 2002, pp. 359–380.

[20] R. Krasny and L. Wang, "Fast evaluation of multiquadric rbf sums by a cartesian treecode," *SIAM Journal on Scientific Computing*, vol. 33, no. 5, pp. 2341–2355, Sep 2011.

[21] Z. Xu, "Treecode algorithm for pairwise electrostatic interactions with solvent-solute polarization," *Phys. Rev. E*, vol. 81, no. 2, p. 020902, Feb 2010.

[22] J. K. Salmon and M. S. Warren, "Skeletons from the Treecode Closet," *Journal of Computational Physics*, vol. 111, pp. 136–155, 1994.

[23] P. Li, H. Johnston, and R. Krasny, "A cartesian treecode for screened coulomb interactions," *Journal of Computational Physics*, vol. 228, no. 10, pp. 3858 – 3868, Jun 2009.

[24] K. Lindsay and R. Krasny, "A particle method and adaptive treecode for vortex sheet motion in three-dimensional flow," *Journal of Computational Physics*, vol. 172, no. 2, pp. 879–907, September 2001.

[25] "Johnston tree code," http://www.math.umass.edu/~johnston/, [Online; accessed 04-May-2012].

[26] "Barnes tree code," http://www.ifa.hawaii.edu/~barnes/treecode/, [Online; accessed 04-May-2012].

[27] "Elucidation tree code," http://www.prism.gatech.edu/~gth716h/BNtree/, [Online; accessed 04-May-2012].

[28] D. Liu, Z.-H. Duan, R. Krasny, and J. Zhu, "Parallel implementation of the treecode ewald method," in *Proceedings of the International Parallel and Distributed Processing Symposium*, Santa Fe, NM, April 2004.

[29] K. P. Martin Burtscher, "An efficient cuda implementation of the tree-based barnes hut n-body algorithm," in *Chapter 6 in GPU Computing Gems Emerald Edition*, W. mei W. Hwu, Ed. Elsevier Inc., 2011, pp. 75–92.

[30] J. Bdorf, E. Gaburov, and S. P. Zwart, "Bonsai: A GPU Tree-Code," 2012.

[31] J. Makino and M. Taiji, *Scientific Simulations with Special-Purpose Computers - the GRAPE Systems*, 1st ed. New York, NY: John Wiley, 1998.

[32] R. G. Belleman, J. Bédorf, and S. F. Portegies Zwart, "High performance direct gravitational N-body simulations on graphics processing units II: An implementation in CUDA," *New Astronomy*, vol. 13, no. 2, pp. 103–112, 2008.

[33] D. A. Pearlman, J. W. Case, David A. Caldwell, W. S. Ross, T. E. Cheatham, S. DeBolt, D. Ferguson, G. Seibel, and P. Kollman, "AMBER, a package of computer programs for applying molecular mechanics, normal mode analysis, molecular dynamics and free energy calculations to simulate the structural and energetic properties of molecules," *Computer Physics Communications*, vol. 91, no. 13, pp. 1 – 41, Sep 1995.

[34] L. Kal, R. Skeel, M. Bhandarkar, R. Brunner, A. Gursoy, N. Krawetz, J. Phillips, A. Shinozaki, K. Varadarajan, and K. Schulten, "Namd2: Greater scalability for parallel molecular dynamics," *Journal of Computational Physics*, vol. 151, no. 1, pp. 283 – 312, May 1999.

[35] J. Huang, J. Jia, and B. Zhang, "Fmm-yukawa: An adaptive fast multipole method for screened coulomb interactions," *Computer Physics Communications*, vol. 180, no. 11, pp. 2331 – 2338, Nov 2009.

[36] H. Cheng, J. Huang, and T. J. Leiterman, "An adaptive fast solver for the modified helmholtz equation in two dimensions," *Journal of Computational Physics*, vol. 211, no. 2, pp. 616 – 637, Jan 2006.

[37] M. E. Ossmani and P. Poncet, "Efficiency of multiscale hybrid grid-particle vortex methods," *Multiscale modeling & simulation*, vol. 8, no. 5, pp. 1671 – 1690, Sep 2010.

[38] A. D. Baczewski and B. Shanker, "An o(n) method for rapidly computing periodic potentials using accelerated cartesian expansions," no. arXiv:1107.3069, Jul 2011.

[39] R. Arora, P. Bangalore, and M. Mernik, "Raising the level of abstraction for developing message passing applications," *Journal of Supercomputing*, vol. 59, no. 2, pp. 1079–1100, 2012.

[40] ——, "A technique for non-invasive application-level checkpointing," *Journal of Supercomputing*, vol. 57, no. 3, pp. 227–255, 2011.

[41] F. Jacob, A. Wagner, P. Bahri, S. Vrbsky, and J. Gray, "Simplifying the development and deployment of MapReduce algorithms," *Journal of Next-Generation Computing*, vol. 2, no. 2, pp. 123–142, 2011.

[42] J. Browne, M. Azam, and S. Sobek, "CODE: A unified approach to parallel programming," *IEEE Software*, vol. 6, no. 4, pp. 10–18, 1989.

[43] F. Devin, P. Boulet, J.-L. Dekeyser, and P. Marquet, "GASPARD: A visual parallel programming environment," in *Proceedings of the International Conference on Parallel Computing in Electrical Engineering*, Warsaw, Poland, September 2002, p. 145.

[44] F. Jacob, J. Gray, J. C. Carver, M. Mernik, and P. Bangalore, "PPModel: a modeling tool for source code maintenance and optimization of parallel programs," *The Journal of Supercomputing*, vol. 62, no. 3, pp. 1560–1582, 2012.

[45] F. Jacob, Y. Sun, J. Gray, and P. Bangalore, "A platform-independent tool for modeling parallel programs," in *Proceedings of the ACM Southeast Regional Conference*, Kennesaw, GA, March 2011, pp. 138–143.