# Domain Specific Language for high performance computing

Ryuki Hiwada s1280076 Supervised by Naohito Nakasato

## Abstract

## 1 Introduction

### 1.1 Background

Particle simulations have been executed in various fields such as astrophysics and fluid dynamics. For example, N-body simulations simulate the dynamical evolution due to gravity between planets, and models of fluid motion models simulate water. Particle simulations are computationally demanding with respect to the number of particles. As an example, we will discuss the case of an N-body simulation. In this simulation, most of the computational time is occupied by the calculation of the gravitational interaction. equation (2.2.1) is the equation for the gravitational interaction.

$$a_i = \sum_{j \neq i}^{N} G m_j \frac{\mathbf{x}_j - \mathbf{x}_i}{\left(\left|\mathbf{x}_j - \mathbf{x}_i\right| + \epsilon^2\right)^{3/2}} \tag{1}$$

$a_i$, $\mathbf{x}_i$ $m_j$,$\mathbf{x}_j$,$N$,$G$,$\epsilon$ are the acceleration and position of particle i, the mass of particle j, the number of position particles, respectively, and softening parameters to prevent gravitational constant divergence. From equation (2.2.1),time complexity is proportional to the number of particles N,

N-body simulations often require large numbers of particles. For example, a globular cluster contains about $10^4$ $10^5$ stars. Therefore, very long computation time is required. parallel computing is generally used to speed up the process.

The computation of interparticle interactions has high data parallelism because the computation of particles is independent of each other. Therefore, Parallelism by SIMD(Single Instruction/Multiple Data ), one of Flynn's taxonomy [1] , is suitable. SIMD means processing multiple data for a single instruction. However, to perform SIMD, code must be written for the computer architecture.

The goal of this study is to allow users to run particle interaction calculations in parallel without having to be aware of parallelization. We developed a Domain Specific Language(DSL )called Pyker. DSL is a programming language for solving specific problems. Pyker generates parallelized code with SIMD instructions for the calculation of interactions from the definition of variables and the description of formulas. The evaluation method for Pyker is to compare the performance of the automatic SIMD performed by the C compiler with that of the SIMD by Pyker. For this purpose, we measured the execution time of the code that does not adapt the SIMD instructions generated by Pyker for each of the three interaction calculations. The execution time of the code that does not adapt SIMD instructions generated from Pyker and the code that does adapt SIMD instructions are measured and compared.

In this paper, section 2 explains how Pyker defines the interparticle interaction equation, section 3 describes the process of code generation by Pyker, and section 4 presents the results of the performance evaluation experiments.

## 2 Definition of the interparticle interaction in Pyker

### 2.1 interparticle interaction

we explain the needed for information to the calculate interparticle interaction with reference to the actual code. gravity.cpp is the C++ source code that performs the calculation in equation (2.2.1).

Listing 1: skeleton code

```
1
2   for (int i = 0; i < n; i++) {
3     double ax = 0.0;
4     double ay = 0.0;
5     double az = 0.0;
6
7     for (int j = 0; j < n; j++) {
8         double dx = double(xj[j][0] -
              xi[i][0]);
9         double dy = double(xj[j][1] -
              xi[i][1]);
10        double dz = double(xj[j][2] -
              xi[i][2]);
11        double r2 = dx * dx + dy * dy +
               dz * dz + eps2;
12        double ri2 = 1.0f / sqrt(r2);
13
14        double mr = mj[j] * ri2 * ri2 *
               ri2;
15
16        ax += double(mr * dx);
17        ay += double(mr * dy);
18        az += double(mr * dz);
19    }
20
```

```
    acci[i][0] = ax;
    acci[i][1] = ay;
    acci[i][2] = az;
}
```

We will explain how this code corresponds to the expression (2.2.1). acci,xi,mj,xj,eps2 correspond to $a_i$, $\mathbf{x}_i$ $m_j$,$\mathbf{x}_j$,$N$,$\epsilon$ respectively. The $G$ is not in the code because it can be calculated after the calculation of the interparticle interactions.

From this code, we can see that the variables to store the results, the variables for particle i and particle j respectively, and the variables for $N$ and $G$ can be calculated after the calculation of the particle-particle interaction. The other variables have different processing and data structures.

Therefore, there are three categories of information in a variable.

- Whether the variable is a particle i, a particle j, a variable that stores the result, or any other variable
- The type information of the variable.
- Whether the variable is a vector or not

With the definition of this variable and a description of the mathematical equation using the variable, a code to calculate the interparticle interaction can be generated. We explain the language specification in the next section.

## 2.2 language specification

There are two parts in Pyker. variable definition part and interaction description part. First, we explain about the variable definition part.

### 2.2.1 variable definition part

Variable definitions are shown below in regular expressions.

$$\text{"}(EPI|EPJ|FORCE)?\ \ (vec3\ <)?(F64)\ >?\ \backslash w+\text{"}$$

EPI and EPJ represent the variables for particles i and j, respectively, and FORCE indicates that they are variables that store the results. Other variables are not written.

Vector are denoted by vec3.If variable is a vector, the variable type is defined in '<>'.

Variable type is 'F64' for double-precision.In Pyker, Variable type is only 'F64'.

Finally, the name of the variable is written.

次に相互作用記述の説明をする。相互作用の記述に使える演算は以下の図に示す。

```
+, -, /, *, **, sqrt(), =, +=
```

+, −, ∗, / はそれぞれ 加算、減算、乗算、徐算を表す。∗∗ はべき乗を表し、sqrt() は平方根を表す。∗ の項がどちらもベクトルの時、∗∗ の底がベクトル、指数が 2 の場合にそれぞれ内積の計算と判断される。= は一時変数を定義する際に使用する。一時変数は定義する際に型を定義する必要はなく、すでに定義されている変数の型とベクトルから推論を行い決定する。+ = は FORCE の変数に計算結果を加算することを示す。原則、要素がベクトルの場合は、ベクトルの要素ごとに演算を行う。記述に使えるオブジェクトは、変数定義を行った変数とその記述より前に定義された一時変数と浮動小数である。

以下にの計算を Pyker 記述した例を示す。

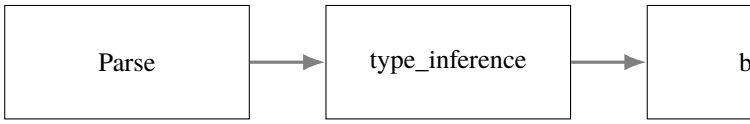
Listing 2: gravity_interparticle

```
EPI vec3<F64> ri
EPJ vec3<F64> rj
EPJ F64 mass
EPJ F64 eps2
FORCE vec3<F64> ai
rij = ri - rj
r2 = rij * rij + eps2
r_inv = 1.0 / sqrt(r2)
r2_inv = r_inv * r_inv
mr_inv = mass * r_inv
mr3_inv = r2_inv * mr_inv
ai += mr3_inv * rij
```

次に、これらの記述から SIMD 命令を行えるコードを生成する説明をする。

# 3 Parser of Pyker

Pyker では、このセクションでは、まずどのようにして SIMD による並列化を行うか説明し、その後 Pyker の記述の読み込みからコード生成までの処理内容と流れを説明します。




### 3.0.1 AVX2 for parallelization

To perform SIMD operations, just like with regular computations, data is first loaded from memory, computed based on that data, and the results are stored. However, in the case of SIMD, computations are performed using SIMD-specific registers, so the methods of loading, computing, and storing differ. When loading, multiple elements that will undergo the same instruction are stored in a SIMD-specific register. These are then used to perform computations simultaneously, and the results are stored for each.

For instance, consider a calculation like C = X + Y, where elements of double-precision floating arrays x and y are computed . When using AVX2 for SIMD, since

the registers are 256 bits, four elements from x and y are loaded each. These are then added simultaneously, and the results are stored in the array C. When loading into SIMD registers, if it's a 1-dimensional array, contiguous memory access is sufficient, but for data structures like Array of Structures (AoS )shown in Figure a., elements need to be loaded individually.

For example, when considering the 3-dimensional coordinates of particles as double-precision floats in pos[n][3](where n is the total number of particles, and pos[i ][0], pos[i][1], pos[i][2]represent the x, y, z coordinates of the i-th particle, respectively), and calculating the difference in x coordinates between particles i and j, it becomes dx = pos[i][0]- pos[j][0]. However, for parallelization with SIMD, desiring dx0 = pos[i][0]- pos[j ][0], dx1 = pos[i+1][0]- pos[j][0], dx2 = pos[i+2][0]- pos[j][0], dx3 = pos[i+3][0]- pos[j][0], the data are packed as (pos[i][0], pos[i+1][0], pos[i+2][0], pos[i+3][0]), (pos[j][0], pos[j][0], pos[j][0], pos[j][0), (dx0, dx1, dx2, dx3)and subtraction is performed simultaneously. However, since data needs to be loaded onto SIMD registers all at once, the gather instruction is used to load data for particle i. The gather instruction is used to read non-contiguous data elements from memory addresses by specifying addresses. An explanation of the gather instruction is shown in Figure x. Data loaded in this way are computed simultaneously using SIMD instructions.

```
dx0 = pos[i][0] - pos[j][0]
dx1 = pos[i + 1][0] - pos[j][0]
dx2 = pos[i + 2][0] - pos[j][0]
dx3 = pos[i + 3][0] - pos[j][0]
```

```
(x1, x2, x3, x4) + (y1, y2, y3, y)
```

When storing computation results, addresses may not be contiguous. In such cases, there is an instruction called scatter, which is the counterpart to gather, but it is not supported in AVX2, so data are stored using pointers individually. In this way, parallelization is achieved with SIMD. Moreover, these instructions can be explicitly handled in high-level programming languages like C and C++ using intrinsic functions. To handle intrinsic functions in C and C++, include the file immintrin.h, which is available as a standard in the language. An example of C++ code that calculates the difference in x coordinates, simd_tmp.cpp, is shown. Pyker generates C++ code capable of executing SIMD instructions like simd_tmp.cpp from the described formulas. The next section will explain the Pyker.

## 3.1 Pyker

In this DSL, code is generated by writing and compiling two definitions: the definition of variables and the definition of interaction formulas. First, let us explain the variable definition.

In this DSL, variables are classified into classes: EPI, EPJ, FORCE, and others. EPI represents particles that receive interactions, while EPJ represents particles that provide interactions. FORCE holds the results of the interaction calculations, and other variables include softening parameters and the like. To perform calculations using these variables, in C++, it would be similar to 4.

Listing 3: skeleton code

```
int kernel(double xi[][3], double xj
    [][3], double ai[][3], double eps2
    , int n){
  // (1).preprocess
  for(int i = 0;i < n;i += 4) { //
      (2)Increment the number of
      parallels
    // (3).load EPI
    // (4).Initialization of tmporary
        force
    for(int j = 0;j < n;j += 1) {
        // (5) load EPJ
        // (6) calculate interparticle
            interactions
    }
    // (7). Store calculation result in
        the FORCE
  }
}
```

(1) involves preparatory steps such as defining variables necessary for the calculation. (2) involves loading the variables of EPI. (3) is about initializing variables that temporarily hold the results of the interaction calculation. (4) involves performing the interaction calculation and saving the results in the primary variable of force. (5) involves storing the results in the FORCE variables. To generate this code, in variable definition, information about the class of the variable, its type, and its dimension is necessary. Therefore, in pyker, variable definitions are written as follows. . The first column, if it is a class, writes the name of the class and its member variable name. The second column explicitly writes the dimensions of the vector if there are any, specifying 3 or 4 dimensions, and the type is either 32-bit or 64-bit floating-point. The third column becomes the name of the variable handled on pykg. For example, "EPI.pos vec3<F64> xi" would be a member variable of EPI named pos, a 3-dimensional 64-bit floating-point variable named xi. The definition of interaction formulas is written to accumulate the results in the FORCE variable using the defined variables. Primary variables necessary for the mathematical description can be newly defined using previously defined variables. Available operations include basic arithmetic, sqrt, and the power symbol "**". The result is stored in the FORCE variable

using '='.For instance, to generate code that c alculates the gravitational interaction formula shown in Figure 1, it would be written in this DSL as figure1.pyker.

Listing 4: hoge

```
EPI.pos vec3<F64> xi
EPJ.pos vec3<F64> xj
EPJ.m F64 mass
FORCE.acc vec3<F64> ai
F64 eps2
F64 g
dr = xj - xi
ai = g * mass * dr / sqrt(dr ** 2 +
    eps2) ** 3
```

## 3.2 Implementation

The process of generating code is illustrated in the following flowchart. In Parsing, the information of the variables defined in the DSL code is converted into a hash with the variable names as keys. Formulas are converted into syntax trees using the sympify() method of Sympy, and a list of these trees is obtained. Next, Common Subexpression Elimination (CSE) is performed on the syntax trees of the converted formulas. CSE is the process of reducing the number of operations by pre-calculating common subexpressions in formulas and using their results in subsequent calculations. In type inference, the types of the primary variables in the formulas are inferred. Afterward, all syntax trees are converted into binary trees to align them with SIMD operations. Finally, the code for calculating interactions is generated by traversing the processed syntax trees. The following sections describe how the implementation was carried out in the order of Parsing, CSE, type inference, conversion to binary trees, and Code generation.

### 3.2.1 Parsing

In Parsing, the process is divided between variable definition and formula handling. In variable definition, objects with class, vector, type, and variable name are created respectively. A hash with the variable name as the key is then made (name_variable_map). Formulas use the sympify() method from the Sympy library to convert the code read as a string into a syntax tree of formulas, obtaining a list of these syntax trees (expr_list). Next, Common Subexpression Elimination (CSE) is performed.

### 3.2.2 CSE

### 3.2.3 type inference

### 3.2.4 convert to binary trees

### 3.2.5 code generatation of non SIMD

### 3.2.6 code generatation of SIMD

# 4 Experiments

## 4.1 gravity interparticle

Listing 5: Nbody-kernel.pyker

```
EPI vec3<F64> ri
EPJ vec3<F64> rj
EPJ F64 mass
EPJ F64 eps2
FORCE vec3<F64> ai
rij = ri - rj
r2 = rij * rij + eps2
r_inv = 1.0 / sqrt(r2)
r2_inv = r_inv * r_inv
mr_inv = mass * r_inv
mr3_inv = r2_inv * mr_inv
ai = mr3_inv * rij
```

Listing 6: Nbody-kernel.pikg

```
EPI F64vec ri:r
EPJ F64vec rj:r
EPJ F64 mj:m
EPJ F64 eps2:eps
FORCE F64vec ai:acc
rij = ri - rj
r2 = rij * rij + eps2
r_inv = rsqrt(r2)
r2_inv = r_inv * r_inv
mr_inv = mj * r_inv
mr3_inv = r2_inv * mr_inv
ai += mr3_inv * rij
```

non-simd
g++ -O3

| N | Pyker | PIKG |
|---|---|---|
| 25000 | 6.725156 | 4.444766 |
| 10000 | 0.738918sec | 0.517324sec |

Listing 7: Nbody-kernel.pikg

```
g++ -O3 -I ~/school/PIKG/PIKG/inc -mavx2 -mfma gravity_interparticle.cpp
```

| N | Pyker(sec) | PIKG(sec) |
|---|---|---|
| 50000 | 5.119071 | 3.333270 |
| 25000 | 1.218129 | 1.148511 |
| 10000 | 0.143777 | 0.101594 |
| 1000 | 0.003521 | 0.002168 |

## 4.2 LennardJones

Listing 8: LennardJones-kernel.pyker

```
EPI F64 rix
EPI F64 riy
EPI F64 riz
EPJ F64 rjx
EPJ F64 rjy
EPJ F64 rjz
EPJ F64 eps

FORCE F64 fx
FORCE F64 fy
FORCE F64 fz
FORCE F64 p
dx = rix - rjx
dy = riy - rjy
dz = riz - rjz
r2 = dx * dx + dy * dy + dz * dz + eps
r2i = 1.0 / r2
r6i = r2i * r2i * r2i
f = (48.0* r6i - 24.0) * r6i * r2i
fx = f * dx
fy = f * dy
fz = f * dz
p = 4.0 * (r6i) * (r6i - 1.0)
```

Listing 9: LennardJones-kernel.pyker

```
EPI F64 rix:rx
EPI F64 riy:ry
EPI F64 riz:rz

EPJ F64 rjx:rx
EPJ F64 rjy:ry
EPJ F64 rjz:rz
EPJ F64 eps2:eps

FORCE F64 fx:fx
FORCE F64 fy:fy
FORCE F64 fz:fz
FORCE F64 p:p

dx = rix - rjx
dy = riy - rjy
dz = riz - rjz
r2 = dx * dx + dy * dy + dz * dz + eps2
r2i = 1.0 / r2
r6i = r2i * r2i * r2i
f = (48.0 * r6i - 24.0) * r6i * r2i
fx += f * dx
fy += f * dy
fz += f * dz
p += 4.0 * r6i*(r6i - 1.0)
```

| | N | Pyker | PIKG |
|---|---|---|---|
| non-simd | 50000 | 40.736862 | 11.656278 |
| | 25000 | 10.662050 | 3.521253 |
| | 10000 | 0.985475 | 0.442775 |

| | N | Pyker | PIKG |
|---|---|---|---|
| simd | 50000 | 3.702871 | 2.838773 |
| | 25000 | 0.838122 | 0.9904373 |
| | 10000 | 0.134388 | 0.215270 |

# 5 Conclusion

# 6 Acknowledgement

# References

# References

[1] M.J. Flynn, “Some computer organizations and their effectiveness,” IEEE transactions on computers, vol.100, no.9, pp.948–960, 1972.