Object Oriented Parallel Programming with C++

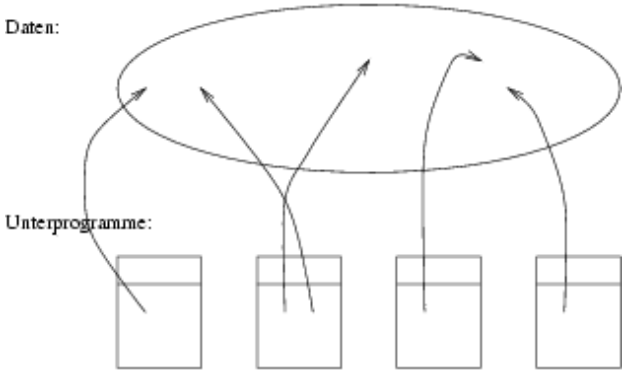
Matthias Müller

University of Stuttgart
High Performance Computing Center Stuttgart (HLRS)
www.hlrs.de

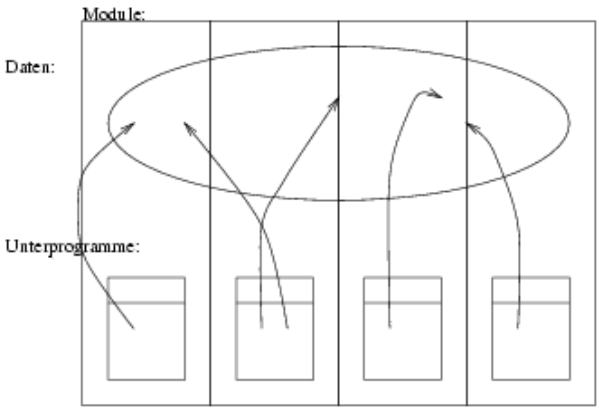
Outline

- Why Object Oriented Programming?
- Some Language Philosophy
- Performance issues
- Case Studies:
  - Short Range Molecular Dynamics
  - CFD
  - Particles in liquid
- Conclusion

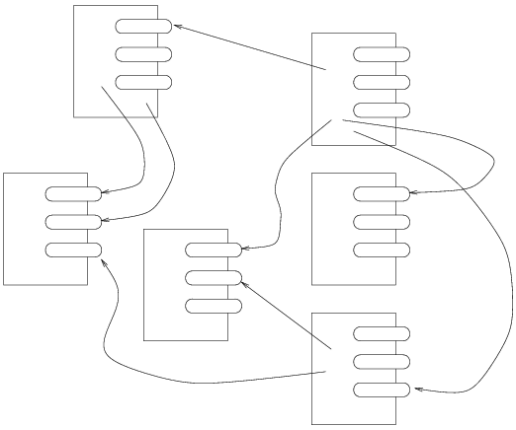
Topology of a Classical Programming Language



Topology of a Programming Language with Modules



Topology of an Object Oriented Language



Possible Benefits of Object Oriented Programming

- Better maintainability of programs
- More frequent code re-use
- More efficient software development in groups
- Higher adaptability of software to new demands
- ...

This benefits are especially important for scientific environments with an average student “lifetime” between one and five years.

Possible Drawbacks of Object Oriented Programming

- Overhead of interface and object definitions will compensate benefits for small programs.
- Object oriented programming may not be suitable for scientific programming.
- Problem oriented languages might be better than multi-purpose object oriented languages.
- A wrong design will result in a costly re-design of the program instead of a quick hack.
- Abstraction may introduce performance penalty.

Language Philosophy

- OO languages to choose from:
  - Smalltalk
  - Eiffel
  - Ada
  - C++
  - Java
  - Fortran 2000?
- Java and C++ are widespread
- Only C++ is available and supported by vendor on Supercomputers like NEC SX, Hitachi SR8000, ...
- It is possible to write Object Oriented programs in “classical” languages, but this requires great self discipline and some times results in reduced performance

History of C++

- 1985 first commercial release of C++ by AT&T
- 1989 beginning of standardization process
- 1. September 1998 ISO/IEC 14882 language standard.

- The standard is quite new.
- Some language features were added quite late.
- The standard includes the definition of a huge standard library. Which requires very recent language features.
- Some compilers still do not implement a sufficient subset of the standard (SUN CC, IBM xlC).
- Many compilers do not include a complete standard library (Cray CC)
- The last two points may change daily.

C++ Language Design

- C++ is a multi-paradigm language
  - Procedural programming:
    - **block structures**
    - **functions**
  - Modular programming:
    - **namespaces**
    - **file scope of identifiers**
  - Object Oriented Programming
    - **classes**
    - **inheritance**
    - **polymorphism**
    - **overloading of functions**
  - Generic Programming
    - **templates**

C++ Language Design

- With a few minor exceptions C++ is a better C:
  This allows a smooth migration from C to C++.
- C++ was designed to have optimal run-time efficiency:
  - you don't pay what you don't use
  - polymorphism:
    - **C++ uses static polymorphism wherever possible. This is necessary to avoid indirect function calls.**
    - **Run time overhead is under the control of the user.**
    - **Availability of templates enables programs with flat inheritance hierarchy, or no inheritance at all.**
  - templates
    - **do not introduce any run-time overhead**
    - **allows to write generic programs for user and built-in types together with operator overloading**
  - no safety checks at run-time

C++ Language Design

- Memory is under the control of user
  - location of variables (heap, stack) according to lifetime of object
  - user decides whether default initialization is required
  - no garbage collector, de-allocation is explicit in destructor
  - user can provide his own allocation strategy
    - **overloading of `new` and `delete`**
    - **users can provide allocators to containers in STL**

  The performance of many scientific programs is dominated by memory access!

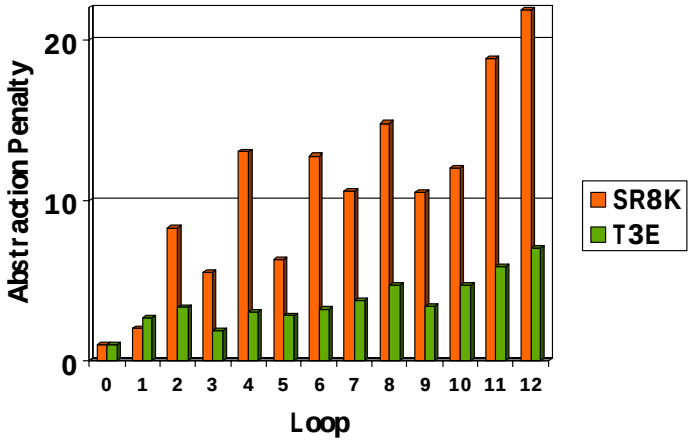
Performance and C++: Stepanov Abstraction Benchmark

- Well known C++ compiler benchmark (used by KAI and gcc).
- Adds 2000 doubles in an array 25000 times.
- 13 different loops that add more and more abstraction.
- A perfect compiler should generate the same code for all loops.
- Performance is given in MFlops and relative to loop 0 (Fortran style loop)
- Overall compiler quality is given as ***abstraction penalty:*** Geometric mean of performance of the loop 1-12 relative to loop 0. **It claims to represent the factor you will be punished by the compiler if you use C++ data abstraction features.**

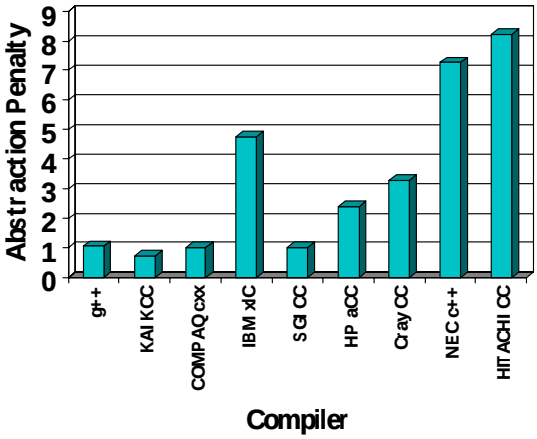
Stepanov benchmark: description of loops

Nr.	Description
0	Fortran style loop
1-12	STL like accumulators with plus function obj.
1,3,5,7,9,11	doubles
2,4,8,10,12	Doubles wrapped in a class
1,2	Regular pointers
3,4	Pointers wrapped in a class
5,6	Pointers wrapped in a reverse-iterator
7,8	Wrapped Pointers wrapped in adapt.
9,10	Wrapped Pointers double wrapped in adapt.
11,12	Double wrapped Ptrs. Double wrapped in class

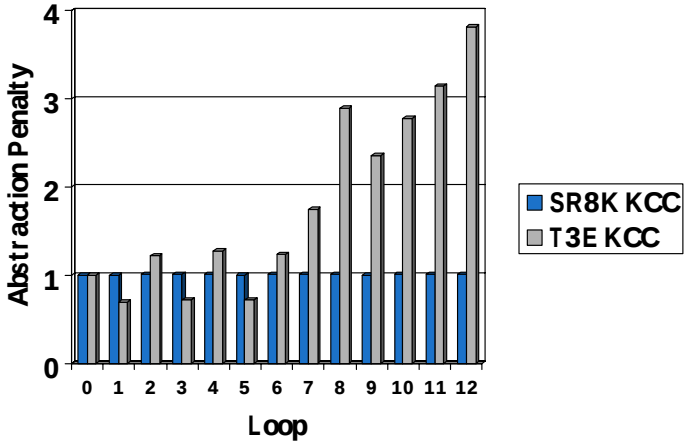
Results of Stepanov Benchmark on SR8000 and T3E



Results of Stepanov Benchmark for Different Platforms



Results of Stepanov Benchmark on SR8000 and T3E



Availability of KCC at HLRS

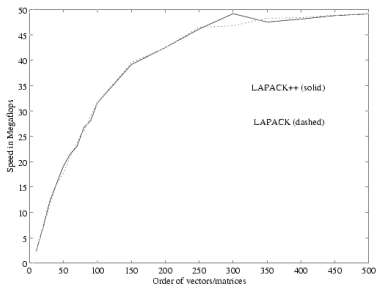
- Most Workstation Platforms:
  - Compaq True64, HP HP-UX, IBM AIX, Linux (Redhat, i386), SGI Irix, SUN Solaris
  - Windows NT
- Some Supercomputer Platforms:
  - Cray T3E, Hitachi SR2201, Hitachi SR8000
- KCC is installed on T3E, SR2201 and SR8000
  (for T3E “module load KCC”, for others in PATH)
- Floating License for University Stuttgart
  2 licenses are available
  Download the software from `http://www.kai.com`
  `(LM_LICENSE_FILE=7244@servint1.rus.uni-stuttgart.de)`
- Information:
  `http://www.hlrs.de/organization/par/services/tools/compiler/kcc.html`

C++ Performance Issues

- Multi Language Programming
- Inlining
- Aliasing
- Temporary Objects
- Expression Templates

C++ and Performance: Multi Language Programming

- Because C++ allows control of memory layout one possible approach is to use C++ for interfaces and complex pre- and post-processing and use Fortran for the numerical expensive part.
- This approach is successfully taken e.g. by Lapack++



- Only suitable if basic data structures are simple, like dense matrices.

C++ and Performance: Inlining

- Inlining should eliminate the overhead of function calls
- Classification of functions:
  - Small functions
    - **computing time is dominated by function call**

```
double& operator[](int i){
   return data[i];
}
```

    - **Inlining is easy**
  - Medium size functions
    - **Significant overhead of function call**
    - **Medium complexity of calculation**
    - **Inlining is difficult**
  - Large size functions
    - **Overhead of function call is negligible**
    - **Large and complex calculations**
    - **Inlining does not matter, or is even a handicap due to code size**

C++ and Performance: Inlining

- For a highly optimizing compiler there should be no “medium sized functions”.
- But: reasons that can inhibit inlining
  - no source code available: provide definition in header file
  - local variables:
    write

```
return data[i];
```

    instead of

```
double rvalue=data[i];
return rvalue;
```

  - function has block scopes (loop, if-then-block):
    blame your compiler writer
- some compilers only inline if inline specifier is provided by user

C++ and Performance: Inlining

- Virtual functions may inhibit inlining, because they are implemented with an indirect function call via the virtual function table

```
class A{
   virtual f();
};
A myA;
A* p=&myA;
myA.f(); //might be inlined
p->f(); // will not be inlined
```

- Use static polymorphism (non virtual functions, templates) instead of dynamic polymorphism wherever possible
- If you really need virtual functions, they will probably be as fast or faster as any hand written code

C++ and Performance: Aliasing

- C++ inherited the alias problems from C
- Example

```
void rank1update(Matrix& A, Vector& x){
   for(int i=0; i<A.rows(); i++)
      for(int j=0; j<A.cols(); j++)
         A(i,j)=x(i)*x(j);
}
```

- Compiler does not know that x is not part of A
- `restrict` qualifier may help, but is not part of C++ standard. Some compilers (KCC, Cray, ..) support this keyword.
- with this qualifier the user gives the promise that there is no alias
- in Fortran this promise is part of the language definition
- you may use a macro to eliminate `restrict`

C++ and Performance: Problem of Temporary Objects

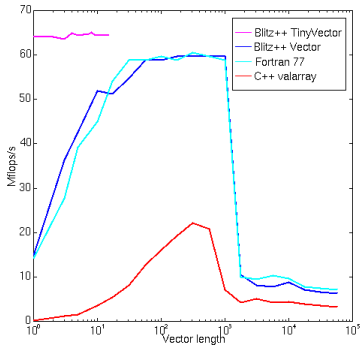
- C++ allows to write expressive code like:
```
TinyVec v1,v2,v3;
v1=v1+v2+v3;
```
- A classical code will look like:
```
class TinyVec{
   public:
   TinyVector operator+(const TinyVec& rhs){
      TinyVec rvalue=*this;
      rvalue+=rhs;
      return rvalue;
   }
   const TinyVec& operator=(...);
   const TinyVec& operator+=(...);
};
```


C++ and Performance: Problem of Temporary Objects

- This will generate code similar to:
```
TinyVec v1,v2,v3;
TinyVec tmp1,tmp2;
tmp1=operator_plus(v1,v2);
tmp2=operator_plus(tmp1,v3);
v1=tmp2;
```
- Possible solutions
  - good compilers will inline all calls and eliminate temporary objects.
  - careful introduction of special return types and overloaded functions will eliminate the need of temporaries (complicated)
  - Advanced techniques like template expressions will in addition perform loop unrolling etc. (complicated)
  - Use a library that implements above optimization techniques.
- First or last approach is recommended

C++ and Performance: Expression Templates

- Idea: use template techniques to make calculations and loop unrolling at compile time.
- Example: **blitz++** library `http://www.oonumerics.org/blitz`
- `Performance of daxpy operation: y=y+a*x`



C++ and MPI: Template Problem

- Problem: in generic template code your write code for type T. The type can represent anything from double to some user-defined type. As soon as you use external libraries you need to write something like MPI_DOUBLE if T is a double, and maybe my_particle_mpi_type if T is a user type.
- This problem is not limited to MPI but occurs in all situations, where you need to map a type to some other information (string, upper limit of this type, etc...)

- Solution: traits
- This solution is also used in `numerical_limits` provided in header <limits>.

C++ and MPI: Generic programming with traits

- Provide template class:

```
template<class T>
class PTM_MPI_traits {
public:
  static inline MPI_Datatype datatype(void);
};
```

- Specialize for every type you need:

```
template<>
class PTM_MPI_traits<double>{
public:
  static inline MPI_Datatype datatype(void){return
MPI_DOUBLE;};
};
```

- Use the traits in MPI Calls:

```
MPI_Send(...,PTM_MPI_traits<T>::datatype(),...);
```

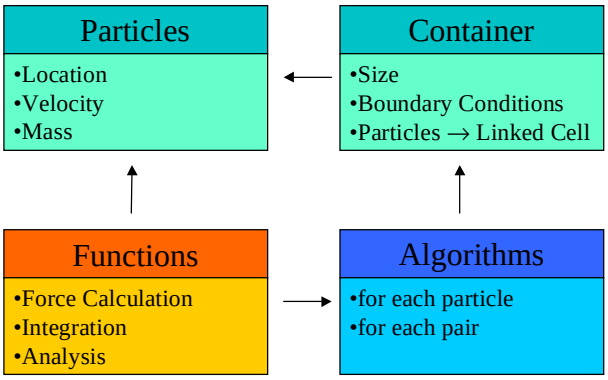

Case Studies

- Molecular Dynamics
- CFD Code
- Sedimentation of Particles in Liquid

Molecular Dynamics

- Different particle types
  - spheres
  - ellipsoids
  - polygons , ...
- Different forces
  - Lennard-Jones
  - Contact Forces
- Different Integrators
  - Verlet, velocity Verlet
  - Leap Frog
  - Predictor Corrector
- Versatile data analysis

Molecular Dynamics



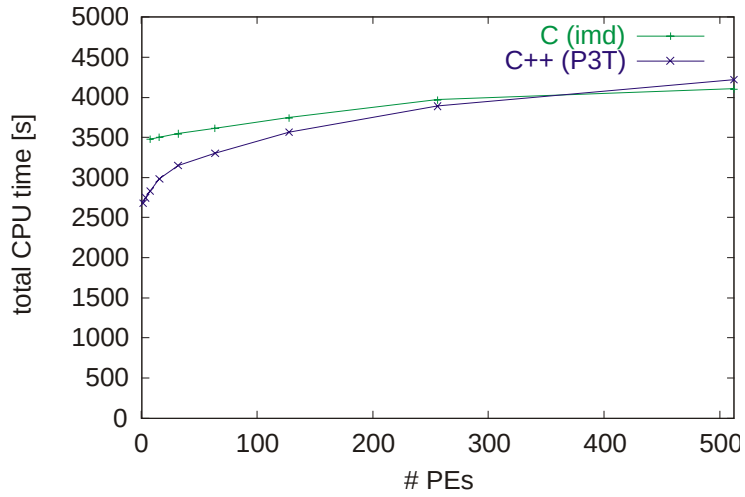
Main loop

```
double dt=0.1;                              // time step
PartContLC<Particle,3>  box(ll,ur,bound_cond,cut_off);
Int_vv_start<Particle>  int_start(dt);   // integration
Int_vv_finish<Particle> int_finish(dt);

while( t < maxT ){
   // velocity verlet integration fist part
   for_each_particle(box.begin(),box.end(),int_start);
   // force calculation
   box.update();
   box.for_each_pair(myForce);
   // velocity verlet integration second part
   for_each_particle(box.begin(),box.end(),int_finish);
   t+=dt;
}
```


Forces

```
template<class PARTICLE>
class Force{
public:
   inline void operator()(PARTICLE& p1, PARTICLE& p2){
        Vector force;
        // calculation of f goes here
        p1.f += force;
        p2.f -= force; // actio = reactio
   }
};

Force<myParticle> myForce;
myParticle p1,p2;
myForce(p1,p2);
```

Performance in Comparison to C

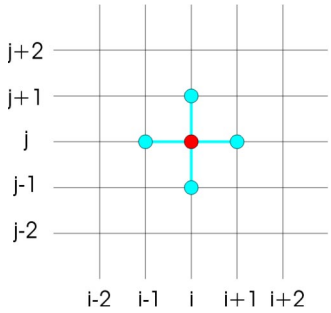


Lenard Jones mit 442368 Teilchen

Partial Differential Equations

Partial Diff. Eq.:

- local information
- rectangular grid
- n-dimensional



Example: Poisson-Equation

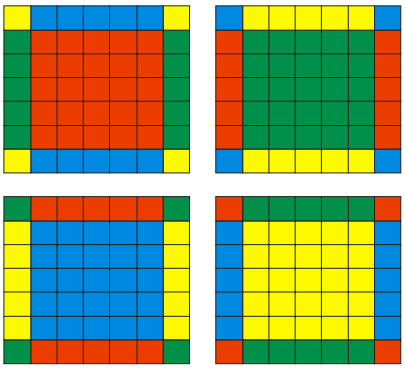
$$\nabla^2 p = f(\mathbf{x})$$

discretized:

$$\frac{1}{h^2}\big(p_{i-1,j,k} + p_{i+1,j,k} + p_{i,j-1,k} + p_{i,j+1,k} + p_{i,j,k-1} + p_{i,j,k+1} - 6p_{i,j,k} \big) = f_{i,j,k}$$

n-dimensional parallel Grid

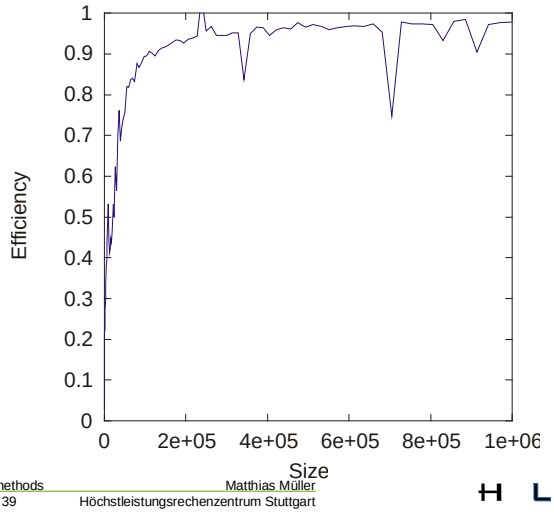
```
const int        dimension=2;
const IntVector  cpu_layout(2,2);
const int        n_shadow=1;
const intVector  boundary_cd(1,1);
const intVector  size(10,10);

// define PE layout
PT_Parallel PEs(dimension,cpu_layout,
                boundary_cd);

// create field
PT_PArray<double> p(PEs, size,
                    n_shadows);

// initialize values
init_field(p);
// update boundary conditions
p.update_bc();
```



Gauss-Seidel Smoother

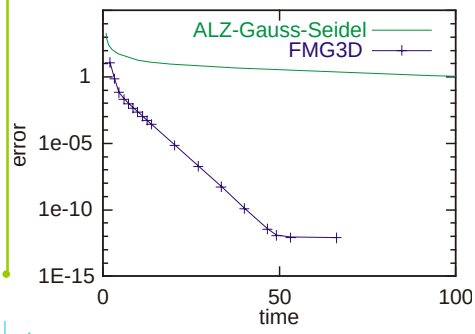
```
struct Poisson_Node { double p, double f };

PT_PArray<Poisson_Node>   p( .....);
GS_operator<Poisson_Node> GaussSeidel(h);

while ( error > limit ){
  // apply boundary conditions
  p.update_bc();
  // apply Gauss Seidel operator
  //    alternatingly linewise
  apply_alw(p,p.begin(),p.end(),GaussSeidel);
}
```

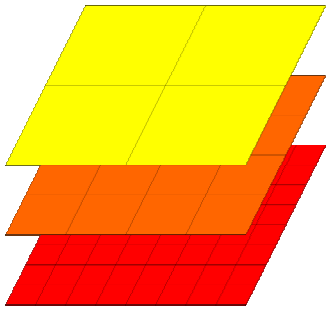
Parallel Efficiency



Multigrid Solver

The use of a multigrid solver improves the poor scaling behavior of the iterative Gauss Seidel method.



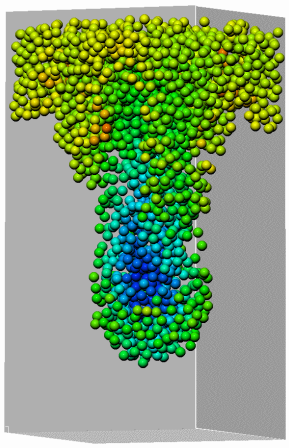


Multigrid Solver

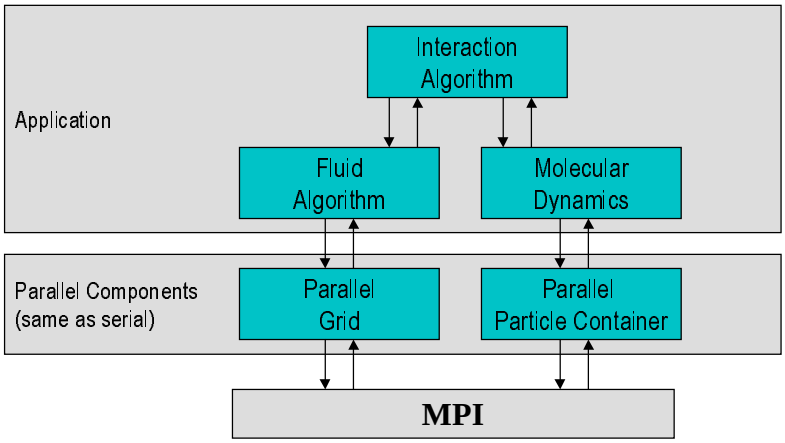
```
const IntVector   cpu_layout(2,4,2);
const IntVector   boundary_cd(1,0,1);
const int         dimension=3;
const int         n_shadow=1;
const IntVector   field_size(257,513,257);

PT_Parallel       PE_layout(Dimension,cpu_layout,boundary_cd);
PT_PArray<double> pressure(PE_layout,field_size,n_shadows);
PT_PArray<double> div_f(PE_layout,field_size,n_shadows);
PT_Poisson        poisson(PE_layout,field_size,n_shadows);

// ... compute rhs (div_f) and solve Poisson equation
poisson.rhs() = div_f;
poisson.solve();
pressure      = poisson.x();
ofstream  save("restart.dat");
save << pressure;
```


Sedimentation of Particles in Liquid

- Relevant system size: 10^5-10^8 particles
- Numerical treatment is expensive
- Treatment of subproblems
  - Molecular Dynamics
    - **Forces**
  - Partial Differential Equations
    - **Navier-Stokes equation with moderate Reynolds numbers**
    - **parallel / serial data structures**
    - **Multigrid Solver for Poisson Equation**
    - **moving boundaries (explicit force density)**



Structure of Particles in Liquid Simulation Program



Conclusion

- Object Oriented programming in C++ is possible without performance penalty
- Separation of algorithms and data results in greater flexibility
- Encapsulation of parallelism should be one goal of design