

# **ohmms Developers' Guide**

**Jeongnim Kim**

**ohmms Developers' Guide**  
by Jeongnim Kim

Copyright © 2004 by Jeongnim Kim

*ohmms* has been developed to provide object-oriented high-performance solutions for multi-scale materials simulations. The core components of *ohmms* are based on PETE and other numerical and parallel libraries. The most common applications of *ohmms* are atomistic simulations of crystalline and molecular systems using various model Hamiltonians to describe the interactions of constituent particles. We discuss the design and implementation of the key components of *ohmms* core libraries and applications.

# Table of Contents

<b>1. Introduction .....</b>	<b>1</b>
The guiding principles of <i>ohmms</i> development .....	1
<b>2. Development tools and libraries .....</b>	<b>3</b>
Version control: cvs .....	3
Configuration tools .....	3
autoconf/automake.....	3
cmake .....	4
Perl script .....	4
Documentation tools.....	5
doxygen.....	5
docbook.....	5
I/O libraries .....	5
<b>3. Base packages .....</b>	<b>7</b>
Packages .....	7
Expression templates .....	7
<b>4. Frequently asked questions.....</b>	<b>9</b>
General questions.....	9
Installation and compilation of <i>ohmms</i> .....	9
Application development .....	11



# Chapter 1. Introduction

## The guiding principles of *ohmms* development

*ohmms* is developed as object-oriented frameworks for materials research. The main applications built upon *ohmms* are to perform atomistic simulations to study various materials properties, e.g., interstitial-defect diffusions, the excitations in quantum dots. The secondary goal of developing *ohmms* is to provide the developers and users with simple interfaces to perform simulations and to write codes for new applications.

The design of *ohmms* has been dictated by the need not to compromise performance in high-performance computing environments and follows few simple rules. Of course, all these are learned from C++ books.

- Use ANSI standard: compilers are getting better and using standard makes the implementation cleaner and portable.
- Use generic algorithms: computer architectures are getting more complicated. Simpler, generic codes are easier for the compilers to optimize and have better chances to get better.
- Lazy computation: do not create nor calculate anything until needed.
- Lazy implementation: use better codes written by experts.

However, many compromises are made to make *ohmms* portable on common platforms. Not all compilers are created equal and many beautiful and elegant concepts in C++ often fail to work *well* on real machines due to inadequate compilers. Instead of using pure template-based implementations as desired, we mix template classes and concrete classes to reduce compiler time and to make it easier for the new developers who are not too familiar with advanced C++ to start writing codes for their applications.

We divide *ohmms* classes into two main categories:

- Base packages: mostly template classes and simple utility classes and functions
- Application packages: application-specific classes defined in namespace OHM-MMS

Developers are expected to work on application classes with the functions provided by the base classes and the application classes that are already in use. The source documentation generated by doxygen<sup>1</sup> is available here<sup>2</sup>.

## Notes

1. <http://www.doxygen.org>
2. <http://www.mcc.uiuc.edu/ohmms/docs/code/html/index.html>

*Chapter 1. Introduction*

## Chapter 2. Development tools and libraries

### Version control: cvs

### Configuration tools

*ohmms* core libraries and applications can be configured by

- autoconf/automake
- cmake
- perl script

For typical *ohmms* applications without GUI supports on generic unix/linux machines, configuring with autoconf/automake is the best choice and is recommended. On some High-performance computers (so called, Super Computers), autoconf/automake often fails to work properly. Things will improve over the time but JK found that cmake or the perl script she wrote sometimes works better and easier to change for the local environments. Especially, if OpenGL/Qt should be used, using cmake is recommended and most likely that only cmake will work.

From this point, we assume that you are in the top-directory of *ohmms* and working in a unix environment which includes Mac OS X and windows/cygwin.

#### autoconf/automake

Like many open-source libraries, start with `./configure -h` to find out what options are available and how to configure *ohmms*. We recommend out-of-source compilation to separate codes from object files and executables for the developers as described below.

#### out-of-source compilation in a `build` directory

##### 1. `mkdir`

```
[build]
```

```
mkdir build
```

create a directory to build the libraries and applications.

##### 2. `cd`

```
[build]
```

```
cd build
```

move to `build` directory.

##### 3. `./configure`

```
./configure --prefix=path-of-build [options]
```

configure with `options`. `prefix` is set to install binaries and libraries in the `build` directory. Default value of `prefix` is `/usr/local` as usual.

##### 4. `make`

build libraries and executables

5. **make**

[install]

install libraries and executables in

- build/include
- build/lib
- build/bin

**cmake**

`cmake` (Cross-platform Make)<sup>1</sup> is an open-source make system to control the software compilation process using simple platform and compiler independent configuration files. For large-scale projects that have to be ported on many platforms, like *ohmms*, using `cmake` is highly recommended. The only drawback is that `cmake` has to be installed but it is very easy to do and does not require root permission. The current versions of `cmake` is not well suited for fortran codes but there are several ways to make `cmake` work for them as implemented in *ohmms*.

**out-of-source compilation in a `build` directory with `cmake`**

1. **mkdir**

[build]

create a directory to build the libraries and applications.

2. **cd**

[build]

move to `build` directory.

3. **cmake**

[..]

run `cmake` to create Makefiles.

4. **make**

build libraries and executables in

- build/include
- build/lib
- build/bin

### Perl script

The perl script `configure.pl` is kept for historical reasons and is available only by `cvs`. Often, `configure.pl` does not reflect the revisions of the codes and should be used only for quick tests and for the platforms where `automake/autoconf` or `cmake` fail to work.

### Configuration with `configure.pl`

#### 1. `./configure.pl`

`[--arch xyz [options]]`

a build directory `bin/xyz` is created and `Makefiles` are created in there. The executable is created in `bin/xyz/ohmmsapp` directory.

#### 2. `cd`

`[bin/xyz]`

move to `bin/xyz` directory.

#### 3. `make`

run `make` to compile.

## Documentation tools

We use standard open-source tools and practices for documentations. Many thanks to the developers of `doxygen`<sup>2</sup>, `docbook`<sup>3</sup> and other useful tools.

### `doxygen`

The code documentation is created by `doxygen`.

### `docbook`

Other documentations, including this guide, are created by `docbook`.

## I/O libraries

### Notes

1. <http://www.cmake.org/HTML/Index.html>
2. <http://www.doxygen.org>
3. <http://www.docbook.org>



## Chapter 3. Base packages

\* *The classes in the base packages are independent of the physical problems and of the algorithms and can be used for any numerical simulations.*

### Packages

The typical class declaration of the classes in the base category is

```
template <class T, unsigned D>
class A {
    . . . .
};
```

where T denotes the data type (double/float/int) and D the physical dimension. The default  $T=double$  and  $D=3$  for the libraries and applications. Many classes use partial specializations with  $D=3$  for performance.

- *PETE*: Portable Expression Template Engine developed by LANL Advanced Computing Lab (included for convenience).
- *OhmmsPETE*: container classes and expression-template-capable operators generated by PETE library.
- *OhmmsData*: base classes to interface *ohmms* objects with text parsers.
- *Utilities*: utility classes and free functions, ascii parsers, generic physical concepts like species.
- *Message*: classes to handle parallelism.
- *OOMPI*: C++ class library for MPI parallel library (included for convenience).
- *EigSolver*: interfaces to numerical libraries, lapack, essl and arpack for eigen problems.
- *Lattice*: layout classes to handle supercells and spatial domains.
- *ParticleBase*: container for a particle set consisting of many particles.

### Expression templates

\* *Expression Templates is a C++ technique for passing expressions as function arguments. The expression can be inlined into the function body, which results in faster and more convenient code than C-style callback functions. This technique can also be used to evaluate vector and matrix expressions in a single pass without temporaries. In preliminary benchmark results, one compiler evaluates vector expressions at 95-99.5 percent efficiency of hand-coded C using this technique (for long vectors). The speed is 2-15 times that of a conventional C++ vector class. [Definition quoted from the abstract of Todd Veldhuizen's paper on expression templates, C++ Report, Vol. 7, No. 5, June 1995, pp26031].*

There are several c++ class libraries that implement expression templates. *ohmms* classes are built on PETE library, for its compactness and easy-to-customize features. The interfaces, member functions and operators, of the classes in *OhmmsPETE* and *ParticleBase* are *almost* compatible to those of *blitz++* library. This makes it possible for the users to switch the underlying expression template library to *blitz++*. With g++ compiler (3.2 or higher), there isn't any significant performance discrepancy between PETE, *blitz++* or *boost::ublas*. On a regular basis, the *ohmms* developers test the expression template libraries with various C++ compilers on the main platforms where their applications run and their relative performance is used to determine the default library for the expression templates.

More on expression templates and implementations can be found:

- Portable Expression Template Engine<sup>1</sup>

### *Chapter 3. Base packages*

- blitz++ at <http://www.oonumerics.org/blitz/index.html> <sup>2</sup>
- Basic Linear Algebra<sup>3</sup>

### **Notes**

1. <http://acts.nersc.gov/pete/main.html>
2. <http://www.oonumerics.org/blitz/index.html>
3. <http://www.boost.org/libs/numeric/ublas/doc/index.htm>

## Chapter 4. Frequently asked questions

### General questions.

#### 1. Why is ohmms developed and what is it?

*ohmms* is developed to do research: developing new algorithms in the electronic structure methods and applying computational materials simulation methods to study microscopic properties of complex materials. *ohmms* is not a full-scale framework like POOMA or equivalents for general scientific applications. Rather, it is a specialized framework for materials simulations written by physicists with a great interest in new software technology. The authors try to take advantage of many great things C++ and high-performance computing libraries provide. At the same time, they take a very practical approach to make the software run on as many platforms as possible.

#### 2. Why not use POOMA<sup>1</sup> for scientific simulations?

The predecessor of *ohmms* was written based on *pooma r1* but due to its license terms and fundamental differences between the two versions of *pooma*, the program had to be abandoned. Using POOMA is still a viable option for *ohmms* developers and for those who are experienced in advanced C++ computing, working with POOMA is highly recommended. *ohmms* should be considered as a poor man's version of *pooma* and is written for those who are not "C++ experts" but the scientists who appreciate what C++ can do for them and the impatient.

## Installation and compilation of *ohmms*

### 1. General questions

#### 1.1. What is the minimum requirement for autoconf/automake?

Autoconf macros for MPI, blas and lapack require autoconf 2.50 or higher. Unfortunately, portable tools are not 100% portable. On cygwin, if you have both stable and develop versions of autoconf, the configuration will fail. An easy way to fix the problem (not the best probably) is to uninstall autoconf-stable (2.13.x). Check out cygwin<sup>2</sup> site.

#### 1.2. What is the configuration file included by -DHAVE\_CONFIG\_H?

When configuration is successful, a file `src/ohmms-config.h` is created in the build directory.

#### 1.3. What is `src/OhmmsApp/AppConfig.h` for?

Many features of *ohmms* applications are not available by default. On the other hand, you may not like to use the default features. This configuration file is created to enable/disable the application-level features that are compiled for an executable. If you want to build your executable only with EAM potential, disable everything else. The default features are determined by the developers, because they have to or like to use them. Note that the content of `Makefile` depends on the configuration option `andsrc/OhmmsApp/AppConfig.h`

#### 1.4. Why can't I just do **configure** like many other packages? Why do I need to specify `c++compiler`?

It is mostly to tune compiler flags and to allow users to choose different compilers for different applications. Although `gcc` will work on any unix platform, the de-

velopers find that vendor-provided compilers tend to perform better, especially when the performance of the applications is limited by the implementations of blas/lapack and the optimization capabilities of c/f77 compilers. For instance on typical linux machines, compiling with gcc is recommended for classical simulations (pure c++ codes), while compiling with intel compilers is recommended for quantum simulations.

1.5. I'd like to use intel c and fortran compilers but it looks like gcc and g77 are selected.

configure script tries to guess a proper set of compilers based on CXX given by option `--with-cxx=CXX`. However, it is a good idea to set the compiler flags for `c` `CC=c-compiler` and `f77` `F77=f77-compiler` to make sure that the intended compilers are used. Mixing compilers on linux, g++ and intel c/f77 compilers, can improve the performance but may require changing Makefile that is generated automatically.

## 2. Configuration options

2.1. The code does not seem to run fast enough with gcc. What can be done, if any?

First step is to add `--enable-optimize` or change `CXXFLAGS="flags"`. Especially with gcc 3.2 and higher, the performance will be greatly improved. Indeed, for the production runs, always use `--enable-optimize[=yes]`. Occasionally, the compiler flags JK uses can add too much stress to the compilers (have seen compilation taking a very long time on linux). You can always go into `src/Makefile` and tone down the options a bit.

2.2. How can I turn on MPI capability?

You have to have `--enable-mpi`. In addition, you need to set few options or environment variables. If your system supports mpi compilers, which are typically shell scripts, e.g., `mpiCC` for mpich and `mpCC` for AIX, set `MPICXX=mpiCC|mpCC` to the compiler. E.g., `configure --with-cxx=icc --enable-mpi MPICXX=mpiCC MPICC=mpicc` will try to configure with intel compilers and default MPICH library. If your system has mpi header files and libraries in the standard places, probably you do not need to do anything. However, if the files cannot be found in your paths, you need to provide the paths explicitly. Note that enabling MPI does not mean you are using data-distributed mpi unless your scripts have specific instructions.

### Warning

Since autoconf uses c compiler to check mpi libraries, both `MPICC` and `MPICXX` have to be set either as environment variables or as options for the **configure**.

2.3. How can I turn on OpenMP capability?

In order to enable OpenMP, add `--enable-openmp`. After checking the compilers and their options, the script will decide if the OpenMP parallelization can be enabled. Currently, the authors have tested OpenMP on SGI Origin, IA64 running linux with intel compilers, IBM P4 with AIX compilers, DEC Cluster with DEC compilers. No platform- or compiler-dependent flags need to be provided for the systems mentioned above during configuration. However, you may need to use thread-safe c++ compiler, such as `cxx=x1c_r`. Consult HPC sites for further information.

**Warning**

Not many compilers support OpenMP constructs for C++ codes. ohmms does not just use loop parallelizations with openmp pragma. The implementation is more like mpi using OpenMP constructs. By taking advantage of strict scope resolutions of C++, one can eliminate private(variables) entirely and do many things clean and efficiently.

**3. OS-specific questions on configuration**

**3.1. configure--with-cxx=gcc** on Mac OS X (darwin) dies with an error message

**unknown configure: error: linking to Fortran libraries from C fails**

Overwrite FLIBS and LDFLAGS with configure. For instance, add to the configuration options, `--enable-shared --enable-dl --disable-static FLIBS=-lg2c LDFLAGS=-L/sw/lib.`

**Application development**

1. I wrote a code similar to `apps/simple/mdsim.cpp`. What do I do next to compile the code?

There are several ways to do this:

1. *ohmms* distribution comes with few small applications that are intended to show how to use ohmms to write your own applications (see the directory `apps`). Hacking `Makefile` (e.g., `apps/mdsim/Makefile`) created by ohmms configuration script is one way, although it is very much discouraged.
2. Use `autoconf/automake` to generate `Makefile.in` and `Makefile` for the new applications. Once you make a `Makefile.am` similar to `apps/mdsim/Makefile.am`, in the top-level directory, run `./bootstrap`. Note that you need `autoconf 2.50` or higher. This will generate all you need to compile your applications. Check out the links for gnu tools to learn about `autoconf/automake`.
3. Use `cmake` Cross-platform Make<sup>3</sup>.

**Notes**

1. <http://www.codesourcery.com/pooma>
2. <http://www.cygwin.com/faq/>
3. <http://www.cmake.org/HTML/Index.html>

*Chapter 4. Frequently asked questions*