Parallel Particle Framework Project As-Built Documentation

Martin Casado
Northern Arizona University
3515 N. Andes Dr.
Flagstaff, AZ 86004

Noel Keen
Lawrence Livermore National Laboratory
7000 East Ave. L-39
Livermore, CA 94550

April 11, 2000

Contents

List of Figures

1 Project Overview

There is currently no standard method for distributing radiation transport, Monte Carlo physics code across a large distributed network. The lack of a standard leads to code that is hard to maintain, may be inefficient and time consuming to develop. In addition, parallelization is a complex topic and can require excessive training costs to bring developers up to speed on efficient implementation. We attempted to address this problem by developing the Parallel Particle Framework. The Parallel Particle Framework (PPF) is designed to aid developers of radiation transport Monte Carlo physics packages by providing an intuitive interface for parallelization of the package across a distributed platform. The philosophy behind the design of the PPF was to create a UNIX based library that is platform independent, easy to integrate into an existing package, provides a simple, intuitive interface and supports loadable extensions. This document details installation, use, design and maintenance of the PPF. However, it should be noted that the PPF is a work in progress and functionality will be added and changed within the library. Please consult the documentation distributed with the PPF for the most current updates and changes.

2 Installation

2.1 System Requirements

Currently the PPF has only been tested on the LC Tera Clusters (DEC Unix), ASCII Blue (AIX 4.3.2) and Linux 2.2.10. Supported compilers are **KCC** and **gcc 2.95**. The PPF should compile under other Unix environments supporting either KCC or gcc 2.95.

The PPF requires a number of external packages for compilation, including:

- PACT: PACT(Portable Application Code Toolkit) is a set of libraries and utilities to facilitate the development of portable software systems. It has a bias toward scientific and engineering applications especially in terms of the data handling services.

  PACT can be downloaded at:
  http://www.llnl.gov/def_sci/pact/

and supports most Unix environments.

- mpich: MPICH is a portable implementation of the full MPI specification for a wide variety of parallel computing environments, including workstation clusters and massively parallel processors.

  MPICH can be downloaded at:
  http://www.mcs.anl.gov/mpich/download.html

- MesaGL: Mesa is a 3-D graphics library with an API which is very similar to that of OpenGL. MesaGL is only required for compilation of the visualization module.

  MesaGL can be downloaded at:
  http://www.ssec.wisc.edu/ brianp/Mesa.htm

Polymesh code which is included with the PPF package currently uses non-standard template constructs which aren't compatible with older compilers. For problems with compiler compatibility, we suggest installing or upgrading gcc version 2.95 or greater.

2.2 Quick Installation Overview

This quick start assumes that you have correctly installed all of the packages mentioned in the system requirements sections and you have a supported compiler. If your system is correctly set up you should be able to install the PPF by performing the following steps.

- Download the ppf.tar.gz file
- **gunzip ppf.tar.gz**
- **tar -xvf ppf.tar**
- **cd ppf**
- **./configure**

- Edit the makefile
  - edit line 48 to point to the correct compiler
  - Set macros **MPICH_INC MPICH_LIB PACT_INC PACT_LIB** to point to your MPICH include directory, MPICH libraries, pact include directory and pact libraries respectively.
  - Note: You can find all areas in makefile that need editing by searching for the word CHANGE which is placed before each section that may need to be modified.
- Write and save modified makefile
- Cross fingers
- From the PPF base directory (which you should already be in) simply type **make make run** To run the example problem.

2.3 Detailed Installation

If, for some reason, the PPF didn't install correctly under the quick installation instructions, read the following section for a more detailed look at the installation procedure. If the following instructions are still not adequate in getting the PPF installed please contact Martin Casado, mc8@dana.ucc.nau.edu or Noel Keen ndk@llnl.gov for further assistance.

The PPF package is distributed as a source tree in a gzipped tar archive. Since the source package is gzipped, only binary transmission means are suitable for acquiring the package. If you are downloading the package from an ftp site be sure that the transmission is in binary mode or you will most likely, have problems gunzipping it.

Once you have successfully downloaded the package use the following commands to uncompress and un-archive it. Be sure that you un-tar the package in the directory you want to house the PPF directory! Also be sure that you have permission to write to the directory you are attempting to install it in!

```
martin/ppfbackup> gunzip ppf_march_30_99.tar.gz
martin/ppfbackup> tar -xvf ppf_march_30_99.tar
****Lots of stuff here****
martin/ppfbackup>
```

Once the source is successfully extracted onto the computer, you should check and make sure you have all the necessary libraries. Check your system to see if MPICH and pact are installed (and MesaGL if you plan on compiling the visualization module). If you can't find either of these packages, download them from the addresses posted above in the requirements section. We leave installation instructions of the packages to their documentation. You will also want to check your compiler version, this can be done with **gcc –version** (assuming you are using gcc). If you have a version previous to 2.95 you are going to have to upgrade.

Once you have verified that all the correct packages are installed on your system, change directories to the ppf base directory, **cd ppf** from the directory that you un-tarred it in, and run the following command **./configure**. The configure script should run through your computer checking for installed components and then create a Makefile. If the configure script does not finish for some reason you will have to cd to EVERY directory within the PPF package and copy **Makefile.in** to Makefile. This problem has not been encountered yet however.

If the configure script has run correctly, or you manually changed all makefile.in's edit the Makefile in the PPF base directory. You will need to change lines 48, to point to your compiler. If you are using g++ be sure to keep the -Wno-non-template-friend -ftemplate-depth-30 flags.

```
C++ = /usr/local/bin/g++ -Wno-non-template-friend -ftemplate-depth-30
```

Secondly be sure to change the following lines, starting at line 54

```
################# Change Values Here ###########################

# HERE: change this to be the MPICH include directory on your computer
MPICH_INC = -I/usr/lib/mpich/include

# HERE: change this to be you MPICH lib directory
MPICH_LIB =  -L/usr/lib -L/usr/local/mpich-1.1.2shm/lib

# HERE: change this to be the include directory for PACT
# Pact if available for download at
# http://www.llnl.gov/def_sci/pact/pact_homepage.html
PACT_INC = -I/home/martin/pact/dev/lnx-2.0.33/include \
        -I/usr/local/apps/kull/include
```

```
# HERE: change this to be your PACT lib directory
PACT_LIB = -L/home/martin/pact/dev/lnx-2.0.33/lib \
        -L/usr/local/apps/kull/lib

################# Done with system specific changes ########
```

To point to the correct include and library directories for both PACT and MPICH.

With the correct changes added to the makefile you should be able to run the **make** command in the base directory (the same one as the makefile) and the PPF should begin compiling. Your screen should look similar to this:

```
martin/ppf> make
/usr/local/bin/g++ -Wno-non-template-friend -ftemplate-depth-30
 -DPOLYMESH -DUSE_MPI  -I./geom/PolyMesh -I./auxiliary/DS++
 -I./auxiliary/Utilities -I./ -I./infrastructure
-I./infrastructure/Include -I/usr/lib/mpich/include
-I/home/martin/pact/dev/lnx-2.0.33/include
-I/usr/local/apps/kull/include  -o geom/PolyMesh/PolyMesh.o -c
 geom/PolyMesh/PolyMesh.cc

/usr/local/bin/g++ -Wno-non-template-friend -ftemplate-depth-30
 -DPOLYMESH -DUSE_MPI  -I./geom/PolyMesh -I./auxiliary/DS++
 -I./auxiliary/Utilities -I./ -I./infrastructure
 -I./infrastructure/Include -I/usr/lib/mpich/include
-I/home/martin/pact/dev/lnx-2.0.33/include
-I/usr/local/apps/kull/include  -o geom/PolyMesh/PartialMesh.o
-c geom/PolyMesh/PartialMesh.cc
****much more stuff here***
```

Depending on the system the PPF could take well over 10 minutes to compile.

To test out the PPF, type **make run** in the base directory of the PPF. This assumes that your have **mpirun** in your path. If you get something that starts out looking like the following and doesn't crash, then congratulations, you have successfully installed the PPF.

```
martin/ppf> make run
```

```
mpirun -p4pg host.list ppf
 rank0 total numberOfNeighbors for: SZones=1 SFaces=6 SNodes=8
RZones1 RFaces6 RNodes4
Performing simple mesh communication checks.
 rank0 total numberOfNeighbors for: SZones=1 SFaces=6 SNodes=4
RZones1 RFaces6 RNodes8
Performing simple mesh communication checks.
```

3 Users Guide

This section is intended to provide an general overview of the PPF API. Depending on the application there are a number of methods in which the PPF library can be used to parallelize the package. The best method in approaching the use of the PPF is to go over the API description, and then follow the examples provided in the PPF package.
Please note that the PPF makes the following assumptions:

- Particles do not interact with each other
- Particles are completely independent. Solutions can be achieved by running a single particle at a time and summing the results.
- When package is run in parallel, the correct number of domains is entered at the command line. For example, if the problem is the 4 domain tophat problem the package should be called as follows:

  ```
  martin/ppf>mpirun -p4pg myhostlist -np 4 ppfTopHat
  ```

  If there is not a one to one correspondence between the number of domains and the processes the problem will not run.
- If the mesh is handled similarly to the *p*pfdrivermesh class provided with the PPF as an example, the partial meshes (each domain) should be labeled as follows: **problemname1.pdb problemname2.pdb problemname3.pdb etc.**.

3.1 API Overview

The only class which the developer should have to be concerned with is ppfInterface. The entire PPF functionality is handled through this class. The ppfInterface.hh header file provides a good description of all interface methods and is included here for a reference:

```
/*****************************************************
* ppfInterface.cc
*
* Martin Casado
*
* ppfInterface is the COMPLETE interface to the
* entire PPF package.  ppfInterface is the facade
* class from the Facade Pattern(see Design Patterns;
* Gramma, Helm, Johnson Vlissades)
* The responsibility of the ppfInterface is to dole out
* requests to the appropriate sub-packages
*
* Note that ppfInterface is, as well, a Singleton class
* which is described in Design Patterns referenced above
*
*****************************************************/

#ifndef _PPF_INTERFACE_HH_
#define _PPF_INTERFACE_HH_

//stl headers
#include <vector>

//PPF specific headers
#include "ppfLogic.hh"
#include "./geom/PolyMesh/CommunicateMesh.hh"
#include "ppfStatistics.hh"

#include "ppfLogic.cc"

template <class type>
class ppfInterface
```

```
{

private:

   //-------------------private methods----------------------------
   //hide the constructor so that those unreliable programmer
   //types must call our Create method
   ppfInterface(int, bool);

   //-------------------private member variables--------------------
   //pointer to hold single instance of class, must be set to zero on
   //corresponding .cc file
   static ppfInterface *instance;

   ppfLogic<type>* ppf_logic;
   ppfStatistics*  ppf_stats;

public:

   //-------------------public methods-----------------------

   //////////////////////////////////////////////////////////////
   // static Create method
   // used to return a single instance of ppfInterface, this
   // ensures that we do not have multiple instances of
   // ppfInterface lousing things up
   //
   // expects the number of domains as an argument
   //////////////////////////////////////////////////////////////

   static ppfInterface* Create(int,  bool);

   /////////////////////////////////////////////////////////////
   // hasFrozenParticles
   //
   // returns a 1 if ppf still has frozen particles, otherwise
   // returns a 0
   /////////////////////////////////////////////////////////////
```

```
int hasFrozenParticles();

//destructor
~ppfInterface();
///////////////////////////////////////////////////////////////
// submitFrozenParticles
//
// Add particles to PPF's frozen particle vector, PPF will make
// the decisions on whether or not to send right away and handle
// all of the buffering issues
//
////////////////////////////////////////////////////////////////
void submitFrozenParticles(vector<type>& );

/////////////////////////////////////////////////////////////////
// handleFrozenParticles
//
// this function gives the PPF the choice to read/send particles
// at the given moment, or wait if there are not enough particles
// to justify the overhead
/////////////////////////////////////////////////////////////////
void handleFrozenParticles();

/////////////////////////////////////////////////////////////////////
// flushParticles
//
// send the particles no matter what, even if there is only 1 particle
// waiting to be sent!
/////////////////////////////////////////////////////////////////////
void flushFrozenParticles();

////////////////////////////////////////////////////////////////////////
// recieveParticles
//
// receive the particles which are being sent from other domains
// to this domain.
////////////////////////////////////////////////////////////////////////
int recieveParticles(vector<type>&);
```

```
    ////////////////////////////////////////////////////////////////////////
    // registerCommunicationMesh
    //
    // pass a pointer from the communication mesh to the ppf so it can
    // determine where to send the particles
    ////////////////////////////////////////////////////////////////////////
    void registerCommunicationMesh(CommunicateMesh<Dim3>*);


    //////////////////////////////////////////////////////////////////////
    // getTotalParticlesHandled
    //
    // returns the total number of particles seen by ppf.
    //////////////////////////////////////////////////////////////////////
    int getTotalParticlesHandled();

    //////////////////////////////////////////////////////////////////////
    // getTotalParticlesSent
    //
    // returns the total number of particles passed to other domains
    //////////////////////////////////////////////////////////////////////
    int getTotalParticlesSent();

    //////////////////////////////////////////////////////////////////////
    // getTotalParticlesRejected
    //
    // returns the total number of particles rejected by ppf
    //////////////////////////////////////////////////////////////////////
    int getTotalParticlesRejected();

};

#endif
```

The methods crucial to the functionality of the PPF are described below.

- **template < class type>**
  ppfInterface<type>* ppfInterface<type>::Create(int domains,bool

stats)
This static create method is the only way to construct a PPF object. The two arguments which it accepts are, the number of processes for the run and whether or not to keep statistics during the run. Statistics will slow down the run though the overhead is marginal. Correct usage of the *Create* method is as follows:

```
ppfInterface *thePPF;
thePPF = ppfInterface<Particle>::Create(
                  ppf_mesh->getNumProcs(),true);
```

- **void ppfInterface<type> ::registerCommunicationMesh(CommunicateMesh<Dim3>* comm_mesh_in)**
  The PPF needs a pointer to the communication mesh to get important information, such as the correct sides to send and receive particles from. You MUST register the Communication mesh before trying to send or receive particles. Correct usage of the *registerCommunicationMesh* method is as follows:

```
thePPF->registerCommunicationMesh(
             ptrToCommunicationMesh);
```

- **template <class type> void ppfInterface<type>::flushFrozenParticles()**
  The PPF allows two methods of sending particles from the current domain, flushing and handling. Flushing the particles sends the particles to their destination domain regardless of how many are waiting to be sent. This can be highly inefficient if only one particle is sent at a time. Correct usage of the *flushFrozenParticles* method is as follows:

```
thePPF->flushFrozenParticles();
```

- **template <class type> void ppfInterface<type>::handleFrozenParticles()**
  This method is used the same as *flushFrozenParticles* except that it will only send if there are 10[1] or more particles to be sent. The usage is identical to that of *flushFrozenParticles*.

[1] This value can be changed within ppfLogic.hh

- **template <class type>**
  int ppfInterface<type> ::recieveParticles(vector<type> & v)
  Every time this method is called, vector v gets filled with all particles which were sent from other domains to the current domain. This currently is NOT a blocking call but this may change in the future. Correct usage of the *recieveParticles* method is as follows:

  ```
  thePPF->recieveParticles(receivedParticles);
  ```

- **template <class type> int ppfInterface<type>::hasFrozenParticles()**
  This method performs a global reduce on all the domains to determine whether there are any particles still bouncing around the system or not. This method is helpful in determining whether or not a timestep is finished. Correct usage of the *hasFrozenParticles* method is as follows:

  ```
  while(thePPF->hasFrozenParticles())
  {
  \\timestep body here
  }
  ```

- **template <class type>**
  void ppfInterface<type>::submitFrozenParticles(vector<type> & vect)
  Once the physics package has decided which particles are exiting the current domain to another domain, they are passed to this method to be sent to other domains using either *flushFrozenParticles* or *handleFrozenParticles*. Correct usage of the *submitFrozenParticles* method is as follows:

  ```
  thePPF->submitFrozenParticles(frozenParticleList);
  ```

3.2 Usage

The best way to understand how the PPF can fit into a physics package is to understand the example physics problem provided with the PPF package. The **ppfmain.cc** file provides a sample driver file for a radiation transport physics problem. Classes *ppfdriverphysics* and *ppfdrivermesh* provide physics functionality and Mesh interfacing for example problem. These two classes can be copied and modified to suit any physics package as necessary.

The main driver file is well commented and provides adequate example of the PPF integrated with a physics package. The ppfmain.cc is included for reference:

```
/////////////////////////////////////////////////////
//ppfmain.cc
// Martin Casado
// sample driver file for physics package using ppf
////////////////////////////////////////////////////
#include "ppfdrivermesh.hh"
#include "ppfdriverphysics.hh"
#include "ppfInterface.hh"
#include "Particle.hh"

int main(int argc, char** argv){
  ppfdrivermesh* ppf_mesh; //define mesh interface
  ppfdriverphysics* ppf_physics; //define physics
  ppfInterface<Particle>* thePPF;
  vector<Particle> receivedParticles;
  //used to recieve particles from other domains

  //instantiate mesh to run Su_Olson_Mesh problem
  ppf_mesh = new ppfdrivermesh(argc, argv,
                             "Su_Olson_Mesh", 2/);
  cout<<"ppf_mesh is initialized"<<endl;
  ppf_physics = new ppfdriverphysics(
                            ppf_mesh->getRank());

  //create initial set of particles
  ppf_physics->GenerateInitialParticles(
  *(ppf_mesh->getMesh()),ppf_mesh->getRank());

  //create thePPF and initialize
  thePPF = ppfInterface<Particle>::Create(
                             ppf_mesh->getNumProcs(),
```

```
                              true);
  //register communication mesh with the PPF
  thePPF->registerCommunicationMesh(
                    ppf_mesh->getCommMesh());

//--------------main loop ------------------------
  int iteration=0;
  int Max_iteration = 1;
 cout<<"Entering main loop"<<endl;
  while(iteration < Max_iteration)
  {
    cout  << "begin iteration number: " << iteration
          << "number of Particles = " i
          << ppf_physics->getNumParticles()
          << endl;

    //have the physics generate new particles as needed
    ppf_physics->GenerateParticles();

    //move the current list of particles throught he mesh
    ppf_physics->AdvanceParticles(*(ppf_mesh->getMesh()),
                                 ppf_mesh->getRank());

    //have the physics submit the list of frozen
   // particles to the PPF to be sent
    thePPF->submitFrozenParticles(
                       ppf_physics->getFrozenParticles());
    ppf_physics->clearFrozenParticles();

    //while global reduce says there is still particles left...
    while(thePPF->hasFrozenParticles())
    {
      cout<<"LOOP"<<endl;
      //pass particles here!!!!!!!!!!!!!!
      thePPF->flushFrozenParticles();

      //Use this to synchronize all domains
      //this will be handled by recieveParticles in the future
```

```
#ifdef USE_MPI
      MPI_Barrier(MPI_COMM_WORLD);
#endif

      //get particles here!!!!!!!!!!!!!
      thePPF->recieveParticles(receivedParticles);

      //add particles here!!!!!!
      ppf_physics->   //run new particles
      ppf_physics->AdvanceParticles(*(ppf_mesh->getMesh()),
                                    ppf_mesh->getRank());

      //submit new particles
      thePPF->submitFrozenParticles(
                    ppf_physics->getFrozenParticles());
      ppf_physics->clearFrozenParticles();
    }

    iteration++;
  }

#ifdef USE_MPI
  MPI_Barrier(MPI_COMM_WORLD);
#endif

  cout<<" Driver Finished, PPF has "
      <<thePPF->hasFrozenParticles()
      <<" frozen particles"<<endl;

  ppf_mesh->cleanExit();
  delete ppf_mesh;
  delete ppf_physics;
  delete thePPF;
  return 0;
}
```

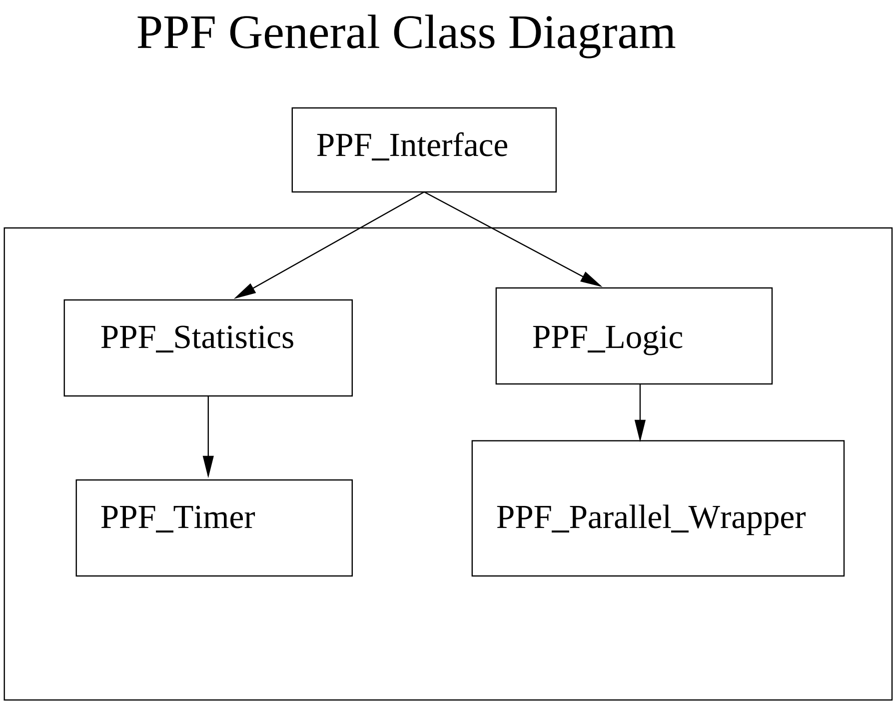

Figure 1: Block diagram of class relationships within PPF.

4 Architecture

The PPF package makes use of the facade pattern[2] to simplify the interface to the library. All calls to the PPF are made through the *ppfInterface* class. Figure 1 shows the basic class relationships within the main PPF architecture. Notice that the *ppfInterface* class is the equivalent of the facade class and acts as the entire interface to the PPF package. It is likely in future implementations that the *ppfStatistics* and the *ppfLogic* classes will have references to each other.

The programmer should not have to concern herself with the inter-workings or the relationships of the classes in the PPF library. Figure 2 depicts the programmers view of the PPF which should be strictly from the perspective of the *ppfInterface* class.

At the time of the writing of this document, the module loader and mod-

[2]Design Patterns,

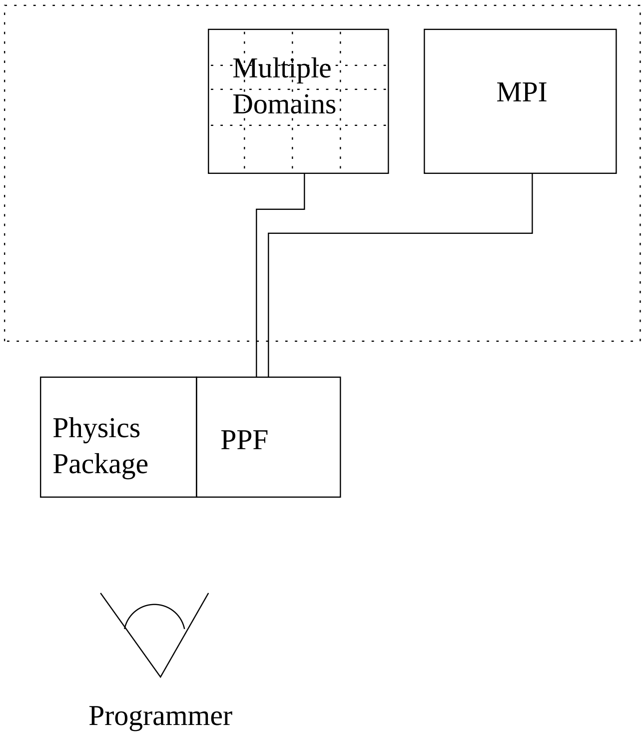

Figure 2: Functional view of the PPF from the eye of the programmer.

ule classes are not architecturally stable as a standard interface is still in development. Not until further versions will the authors support use of the module loader and modules.

One final comment as to the architecture of the PPF is that the *ppfInterface* class is a singleton[3] class which will only allow one instance at a time. This is to protect programmers from attempting to create multiple instances which will most likely compromise the correctness of the problem being run, if the problem runs at all. As mentioned previously, *ppfInterface* can only be instantiated through its static *Create* method to limit the number of instances to one. The *Create* method uses the **new** operator to allocated memory and therefore the PPF instance MUST be deallocated upon exiting the program or it will result in a memory leak.

5 Maintenance

5.1 Corrective Maintenance

Currently the PPF uses non blocking I/O on the receives however this does not intuitively lend to integration in a typical radiation transport particle package for reasons described in the send and receive analysis paper found in the documentation section of the project web-site[4] . Hence, future implementations will either use completely blocking receives or have the option of using a blocking receive.

Currently there are no known major bugs. The PPF base has been tested on two problems, **Su Olson** and the **Top Hat** problem and migrated from the former which is a two domain problem to the latter which is a four domain without any troubles. At the time of the writing of this paper, the PPF has not been tested on any problems larger than four domains, however the package should be able to handle unlimited domains in the same manner.

The module interface is still rather problematic in is not presently supported. It is recommended that until a version with stable module support is released, users should not attempt to use the current modules.

The biggest problem with the project is migration between platforms largely due to dependencies on external packages (i.e. PACT, MPICH) and

[3]Design Patterns, Gamme, ERich. Addison Wesley. 1995

[4]http://www.cse.nau.edu/Design/D4P/EGR486/CSE/99-Projects/Multi/documents.html

compiler compatibilities. Currently the PPF is only supported on Linux 2.2.10 and above using gcc 2.95 and above, the LC Tera cluster and ASCII Blue[5]. All modifications to the PPF should be done with the use of autoconf to keep the project as platform independent as possible.

5.2 Perfective Maintenance

The PPF project is still incomplete and there is quit a bit of functionality that will be integrated with the PPF package as the project progresses. A major design goal is to keep the fundamental package small so that it isn't unnecessarily complicated and wont add a lot of overhead to the project that it is being used in. In light of this, extraneous functionality that is not part of the core message passing is planned to be done in modules. A list of the components to be added to the PPF either in the base package or modules is as follows.

- **General Domain Decomposition** - Transport problems usually relay on a finite number of sources for particles which are not distributed evenly throughout the project description. This leads to an unevenness in the workload between processors. In the worst case one node might do all the work and the rest lay idle. The PPF is planned to have constructs which will deal with load balancing by distributing particles throughout the cluster, or replicating domains that are doing the majority of the work. The actual interface to the programmer has yet to be determined.

- **Shared Memory Parallelization** - Most distributed systems that the PPF will be run on have smp nodes of four to twelve processors. Using MPI across a shared memory model is a valid way of breaking up the work but it does not take advantage of shared memory for quicker message passing nor does it limit the number of problem instances to one. Ideally, the PPF would parallelize the problem using MPI over the distribute platform while handling shared memory parallelization via pthreads or OpenMP.

- Modules

[5]We would really appreciate if other successful ports are emailed to us so we can include them in the official release package

  - **Graphical Statistics** - Module which once loaded and run displays real time statistics of number of particles per node, message traffic between nodes etc. The complete functionality of this module has not been decided.
  - **Visualization** - This module provides a real time display of particles within the problem. Currently the module is in rudimentary form, but it does create images via OpenGL that show particle behavior within the problem.
  - **Snapshot** - This module will enable developers to take snap shots of the problem during execution and save them to disk to be visualized at a later time. Run-time visualization is costly and in effort to avoid a loss in performance, the developer can view particle behavior after the problem has been run.

- **Fault tolerance** - It is not unlikely for a node to go down on massively parallel supercomputers such as ASCII Blue. It is the goal of the developers of the PPF to be able to detect downed nodes and to correct the problem without compromising data. If a node fails, the PPF will notify the Physics package and hand back the particles which would have been lost, or in worst case, notify the Physics package as to the number of particles lost.

5.3 Adaptive Maintenance

All future implementations of the PPF are expected to be fully compatible with the current version. Additions will be covered in documentation and should not change current calls within the interface, however additional methods may be added. Special care should be taken when porting the PPF between platforms. All ports that will be included in the official release will not impinge on the performance or compatibility of platforms previously supported. Therefore, all changes to adapt to a new platform must be done within the autoconf system.

Any changes to the PPF that affects performance or decisions in the actual message passing will be handled in the *ppfLogic* class or a class that works in conjunction with ppfLogic. However this should have no bearing on the user since all the functionality in the PPF is accessed through *ppfInterface* which will not change.

All changes in the mechanics of the parallelization will be made within the *ppfParallelWrapper* class. Currently there are no plans to change from using MPI as the parallel mechanism, however if this should happen, only the *ppfParallelWrapper* class will be effected. This abstraction frees the user of concern with the parallel mechanisms employed. It is possible as well for there to be a release of a PVM version of the PPF for researchers who do not have access to a machine with MPI or prefer to use PVM.

6 Postmortem

Design and implementation of the PPF has been an incredible insight for me into the realm of large scale development. During the course of the two semester project the PPF helped me to understand the strengths and weaknesses of my approach of the design process. This section of the the paper provides a reflection of my experiences during the design and development process of the PPF. There are a number of lessons I have learned from the outcomes of decisions I made during the design of the PPF that will change the way I approach projects in the future. A key understanding of what went right and what went wrong during the course of the project can be applied to future projects so the same mistakes are not made twice and successful approaches are reused.

6.1 What Went Right

Overall that project, in my opinion, was a success. The outcome is a functional library which adheres to the basic requirements that the sponsor put forth at the beginning of the semester. However, how functional a package is, is not the only means to judge the success of a project. Maintainability is a big issue with projects that are destined to be partially completed by the project's end, as the PPF was. A hacked piece of software is of little to no use to the sponsor of the project and is a waste of resources. As the PPF library stands at the end of the project there is a strong and reliable foundation that can be easily built upon in discrete components. This section will describe successes in the design and development process of the PPF.

- **Use of the Iterative Method** The iterative method proved to be a good choice for the development of the PPF library. One of the major advantages provided by the iterative method is that it enables continual testing from the onset of the project. This is particularly important in the domain of parallel computing. Parallel code by nature is tricky and the paradigm used by MPI requires a functioning program too execute.

  Another advantage of using the iterative method is that the integrity of the architecture is tested early on in the project. As components of the system are replaced from skeletal code to functioning entities it becomes apparent where architectural problems will arise. Within a waterfall approach the architecture of a given component within the software structure is not tested until that component is tested.

From the onset of the project it was apparent that the PPF could not possibly be completed in its entirety. The goal was to produce a functioning base library that is solid and is an intuitive base for future additions. The iterative method adds to the maintainability of the code because it inherently demonstrates the architecture taking out the guesswork for future developers. This allowed me to add skeletal code for functionality that I had no intention of implementing, but can easily be swapped into the project at a later time.

The final perk of using the iterative method that I would like to mention is that it forces a modular design. The PPF was designed and developed fully with the object oriented paradigm in mind. The only exception was the MPI code which followed the first releases of the MPI standard due to support issues. Since the iterative method requires that the project be broken up into discrete components, this method fit well with our object oriented approach to the PPF.

In the future when confronted with the decision to choose a design method I will likely stick with the iterative method if the project has the following characteristics:

- It may not be completed.
- IT follows an object oriented paradigm.
- Maintainability is a big issue.
- The architecture is straight-forward and obvious at the onset of the project.

I have not had enough experience with other design methods to contrast them with the iterative.

- **Previous Experience** My experience working on similar problems to the PPF was probably my greatest asset to the success of the project. Going into senior design I had adequate experience working on the massively parallel Livermore machines. The absence of this experience would have hindered me greatly since there is a steep learning curve to becoming familiar with these complex systems.

During the summer of 1999 I worked on parallelizing an Implicit Monte Carlo package (IMC) which is a particle transport package. This implementation specific parallelization lead to my understanding of how to approach the problem generally. This basic understanding was key in the design of the PPF. Had I not had experience working with actual parallelization of radiation trnsport packages, there is little chance I would have understood the subtleties of the problem.

The contacts I created at Livermore were, as well, a great advantage over the project. During design of the PPF I would contact the physicists which are potential clients, to check whether the interface was intuitive and the functionality was germain to their projects. This feedback greatly shaped the overall design of the project.

- **Communication with Sponsor** Noel was an excellent sponsor and source of information throughout the project. Noel received his bachelors and masters in nuclear engineering and has experience at both Los Alamos National Labs and Livermore National Lab. Noel was key to specking out design and interface issues and provided me with quite a bit of material to help me get started. It is safe to say that without Noel's attention, the project would not have ended on such a successful note.

  Noel was always available via email or telephone and answered my inquiries promptly. I have a collection of more than one hundred and twenty emails to and from Noel that document our communication throughout the project. Noel, as well, provided insights into porting the PolyMesh code which proved extremely difficult for me.

  In future projects I will be careful to make sure that sponsors provide the development team with the attention that Noel did because that is key to a projects success. Ensuring the sponsors active role will require establishing weekly (or more frequent) meeting times, email availability and the sponsor's expertise. If the sponsor is not knowledgeable in the area of the project, then it would be a good idea to establish an outside contact with someone who is.

- **Use of facade pattern** Design patterns are well researched methods for designing object oriented software. The facade pattern uses a "facade class" which is a single interface class to a more complex package.

For example, within the PPF there are multiple packages abstracted out due to responsibility, however the facade class *ppfInterface* provides a single interface.

The main reasoning to using the facade pattern is for simplicity. The physicists who will be using the PPF are, for the most part, not computer scientists and therefore the intricacies of the functionality should be transparent to the developers. In the realm of library development, intuitive and simple interfacing is a high priority, second only to functionality[6]. In the realm of object oriented design, the facade pattern is a powerful tool for simplifying and encapsulating the interface.

- **Encapsulation by Responsibility** Encapsulation by responsibility allows the architecture to be structured in such a way that the basic functionality of a component can be changed without affecting the entire project. A good example of this would be the PPF's *ppfParallelWrapper* class. The responsibility of this class is to encapsulate all the low level, implementation specific parallelization calls (the calls to MPI in the case of the current release). If the developers of the PPF in the future decided to change the particle receiving method to flush out particle buffers when it is called, this could be implemented with no affect on the rest of the package. The only affect of added functionality to the PPF would be updating the interface within the *ppfInterface* class.

- **Use of autoconf** The world of Unix is a highly confusing, non-standardized place to develop in. Each flavor of Unix has its own quirks that are sure to be troublesome to a multi-platform project of any complexity. One method of dealing with these differences in Unices is autoconf. Autoconf is a shell script which checks the characteristics of the machine it is being called on. The following list gives an portion of numerous characteristics checked for by autoconf:

  - which compiler to use
  - byte ordering of the system
  - which system calls are supported

[6]However it can be argued that if the interface is poor then the functionality is useless making the interface the higher priority of the two.

- which libraries are installed on the machine
- how the compiler handles certain code snippets
- etc.

From the information that it gets from the system, autoconf generates a custom Makefile and a custom header file. If the system does not have required software to support the compilation, autoconf notifies the user in an understandable format, not an esoteric message produced by the compiler or linker. The programmer can use the header file created by autoconf to format her code to work with the current environment. Autoconf is a necessary tool for multi-platform projects on Unix, and should be used for convenience to the user.

6.2 What Went Wrong

As with any project there were a number of blunders that resulted from inexperience, wrong decisions and blatant mistakes. Fortunately for the PPF, these were minor and relatively sparse. The majority of the problems encountered served as excellent learning experiences and have shaped the way I will approach projects in the future. The following list details what went wrong during the course of the PPF project.

- **Not Enough Knowledge of Project Before Design** Though I had experience working with radiation transport problems going into the project , I still managed to assume more than I actually new. This resulted in me constructing a design without actually getting my hands dirty with the code first. While the architecture seems to be solid, I believe I would have had a better grasp on how to approach the problem if I had spent a week working on "throw away code" to learn the intricacies involved in coding such a project. In the future I will attempt to spend a few days trying to tackle the problem without thinking through a full design. This is to serve for pedagogical purposes only, all code developed during this learning period will not be used in the final project. After dabbling with the code, I will use what knowledge I gained to help in creating a strong design.

- **Freeware Tools** While the use of freeware tools for development and documentation have their obvious benefits, I learned that there are two major drawbacks; support and a steep learning curve. Since there is no commercial backing to open source tools, there is not much incentive for the developers to answer questions regarding usage of them. Furthermore, I found there to be many more bugs (mostly undocumented) in open source software than in commercial. A second disadvantage was the steep learning curve. While on the windows platform an entire project including development, design and documentation and be done using Visual C++ and Microsoft Word this is not the case with non-commercial software on Unix. I had to use a different tool for nearly each task which upped the training before starting on the project.

In retro-spect, under the time constraints, it probably would have been a better idea to develop in a more integrated environment. With that said, now that I have gained the precursory knowledge needed to use these tools I plan to use them in the future. The Unix environment provide a very powerful development environment, and if the developer is comfortable with basic Unix development tools there is no need to use a commercial package.

- **Schedule** It was pretty obvious from the get go that the schedule I drafted up was not going to be a good indication of the time it would take to complete projects. Design and requirements took less time than expected, however porting and system issues dominated the beginning of the second semester and weren't even mentioned in the schedule.

  I believe that accurate prediction of the project schedule comes with experience. This is the largest project I have worked on alone and there was no chance for parallelism to adjust for changes in the project schedule. In future projects I will use experiences such as this one to draw from in estimating project times.

- **Testing** The major problem I faced in the testing arena was that I did not have the expertise to determine whether that package I had written was providing the correct answers in real packages. I could test the raw computer science of the library; absence of memory leaks, no data lost during sends, statistics where accurate. However, the correctness of the package cannot be determined without running it on real physics problems and verifying the results. It would turn out that I would do as much testing as I could on integrity of the code from a computer science standpoint and hand the package to Noel for verification with real physics problems.

6.3 Concluding Remarks

If I were to start over on the PPF project knowing what I know now I would approach it differently, however this is testimony to that fact that I have learned a great deal in my experience designing and implementing it. Given a fresh start I would get familiar with the code before embarking on the design, pay closer attention to the tools I was planning on using, and ensure there were valid testing methods (and determine what those were). All in all, however, I believe the architecture and implementation of the PPF are well done and a good base for future developments.

Over the past three ears I have done all large design projects within a team environment and working on my own was a real eye opener. Usually in a team environment you distribute tasks based on competence of the team members in that area. Traditionally I have been involved with high level design and coding only, I was never chosen to handle systems, documentation nor web page development. This experience has allowed me to demonstrate to myself my ability to function in all areas of the design process. I would suggest to students serious about the field of software engineering to attempt a large project individually. This will give that student a much better estimate of her abilities when placed in a team environment and it will make her a more valuable teammate due to a basic understanding in all areas of the design process.