Modernizing the C++ Interface to MPI

Prabhanjan Kambadur

Open Systems Lab
Indiana University
Bloomington, USA

EuroPVM/MPI'06

Bonn, Germany

18th September 2006

Motivation

```
std::list<std::list<int> > lst;
mpi::Send(lst, dest, tag, comm);
```

- "Generic Programming" techniques
  - Emphasize *re-usability* and *efficiency*
  - Non-Intrusive
  - No runtime penalties for primitive types
- Previous efforts relied on object-oriented techniques
  - Intrusive
  - Runtime penalties
  - OOMPI and MPI++

Motivation - contd

```
std::list<std::list<int> > ls;

int num_lists = ls.size();
MPI_Send(&num_lists, 1, MPI_INT, dest, tag, comm);

std::list<std::list<int> >::iterator out_iter = ls.begin();

while (out_iter != ls.end()) {
    std::vector<int> buffer(out_iter->begin(), out_iter->end());
    MPI_Send((void*)&buffer.front(),out_iter->size(), MPI_INT, dest, tag, comm);
    ++out_iter;
}
```

Goals

- Support modern C++ idioms
  - Generic containers through templates
  - Operator overloading, functors
  - Iterators
- Eliminate redundancies
  - Default function parameters
  - Deducible arguments
  - References
- No runtime penalty for primitive types

Eliminating Redundancies

```
Send(&data,1,MPI_INT,dest,tag,MPI_COMM_WORLD);
```

- Use References
- Count can be deduced
- Type can be deduced
- Default values for parameters

Eliminating Redundancies

Send(data,1,MPI_INT,dest,tag,MPI_COMM_WORLD);

- Use References
- Count can be deduced
- Type can be deduced
- Default values for parameters

Eliminating Redundancies

Send(data,MPI_INT,dest,tag,MPI_COMM_WORLD);

- Use References
- Count can be deduced
- Type can be deduced
- Default values for parameters

Eliminating Redundancies

```
Send(data,dest,tag,MPI_COMM_WORLD);
```

- Use References
- Count can be deduced
- Type can be deduced
- Default values for parameters

Eliminating Redundancies

Send(data,dest,tag);

- Use References
- Count can be deduced
- Type can be deduced
- Default values for parameters

Supporting STL Containers

```
using namespace std;

list<list<int> > ls;
// intialize ls

int num_lists = ls.size();
MPI_Send(&num_lists, 1, MPI_INT, dest, tag, comm);

list<list<int> >::iterator out_iter = ls.begin();
while(out_iter != ls.end()){
   vector<int> buffer(out_iter->begin(), out_iter->end());
   MPI_Send((void*)&buffer.front(), out_iter->size(),
                    MPI_INT, dest, tag, comm);
   ++out_iter;
}
```

Supporting STL Containers

```
using namespace std;

list<list<int> > ls;
// intialize ls

int num_lists = ls.size();
MPI_Send(&num_lists, 1, MPI_INT, dest, tag, comm);

list<list<int> >::iterator out_iter = ls.begin();
while(out_iter != ls.end()){
   vector<int> buffer(out_iter->begin(), out_iter->end());
   MPI_Send((void*)&buffer.front(), out_iter->size(),
                    MPI_INT, dest, tag, comm);
   ++out_iter;
}
```

Supporting STL Containers

```
using namespace std;

list<list<int> > ls;
// intialize ls

int num_lists = ls.size();
MPI_Send(&num_lists, 1, MPI_INT, dest, tag, comm);

list<list<int> >::iterator out_iter = ls.begin();
while(out_iter != ls.end()){
   vector<int> buffer(out_iter->begin(), out_iter->end());
   MPI_Send((void*)&buffer.front(), out_iter->size(),
                    MPI_INT, dest, tag, comm);
   ++out_iter;
}
```

Supporting STL Containers

```
using namespace std;

list<list<int> > ls;
// intialize ls

int num_lists = ls.size();
MPI_Send(&num_lists, 1, MPI_INT, dest, tag, comm);

list<list<int> >::iterator out_iter = ls.begin();
while(out_iter != ls.end()){
   vector<int> buffer(out_iter->begin(), out_iter->end());
   MPI_Send((void*)&buffer.front(), out_iter->size(),
                    MPI_INT, dest, tag, comm);
   ++out_iter;
}

Send(ls, dest, tag);
```

Extending Support for User-defined Types

```
struct employee_record {
   std:: string   employee_name;
   int  employee_id;
   std:: list  <std:: string > address;
};

//  Sender: Rank 0
employee_record data;
mpi::Send(data, 1, msg_tag)

//  Receiver: Rank 1
employee_record data;
mpi::Recv(data, 0, msg_tag)
```

Goals

- Minimal effort
- Non-intrusive

User-defined Types

```
template<class Archiver>
void serialize (Archiver & ar,  employee_record & rec, const unsigned int version) {
    ar & rec.employee_name & rec.employee_id & rec.address;
}
```

- Boost Serialization Library (BSL)
  - Simple interface
  - Non-Intrusive
  - Built-in support for STL containers and iterators
  - Customizable **Archiver**

Supporting Functors

```
string common_prefix(const string& s1, const string& s2) {
  if (s1.size() <= s2.size())
    return string(s1.begin(), mismatch(s1.begin(), s1.end(), s2.begin()).first);
  else
      return common_prefix(s2, s1);
}

string global_common_prefix = mpi::Allreduce(my_string, &common_prefix);
```

- Reduction operations on
  - STL and user-defined types
  - STL and user-defined functors

```
int data[n];
mpi::Send(data, data+10, dest, tag);
 -> Call MPI_Send Directly

struct gps{ int x; int y;}
// Create the MPI_Datatype
gps data;
mpi::Send(data, dest, tag);
 -> Use the MPI_Datatype to call MPI_Send directly

std::list<int> data;
mpi::Send(data, dest, tag);
 -> Serialize data and then, call MPI_Send
```

- Choose the best implementation for each type
  - Compile-time
  - Based on type-properties

```
int data[n];
mpi::Send(data, data+10, dest, tag);
 -> Call MPI_Send Directly

struct gps{ int x; int y;}
// Create the MPI_Datatype
gps data;
mpi::Send(data, dest, tag);
 -> Use the MPI_Datatype to call MPI_Send directly

std::list<int> data;
mpi::Send(data, dest, tag);
 -> Serialize data and then, call MPI_Send
```

- Choose the best implementation for each type
  - Compile-time
  - Based on type-properties

```
int data[n];
mpi::Send(data, data+10, dest, tag);
 -> Call MPI_Send Directly

struct gps{ int x; int y;}
// Create the MPI_Datatype
gps data;
mpi::Send(data, dest, tag);
 -> Use the MPI_Datatype to call MPI_Send directly

std::list<int> data;
mpi::Send(data, dest, tag);
 -> Serialize data and then, call MPI_Send
```

- Choose the best implementation for each type
  - Compile-time
  - Based on type-properties

```
int data[n];
mpi::Send(data, data+10, dest, tag);
 -> Call MPI_Send Directly
```

```
struct gps{ int x; int y;}
// Create the MPI_Datatype
gps data;
mpi::Send(data, dest, tag);
 -> Use the MPI_Datatype to call MPI_Send directly
```

```
std::list<int> data;
mpi::Send(data, dest, tag);
 -> Serialize data and then, call MPI_Send
```

- Choose the best implementation for each type
  - Compile-time
  - Based on type-properties

Implementation Strategies
Function Specialization

```
int data[n];
mpi::Send(data, data+10, dest, tag);
 -> Call MPI_Send Directly

struct gps{ int x; int y;}
// Create the MPI_Datatype
gps data;
mpi::Send(data, dest, tag);
 -> Use the MPI_Datatype to call MPI_Send directly

std::list<int> data;
mpi::Send(data, dest, tag);
 -> Serialize data and then, call MPI_Send
```

- Choose the best implementation for each type
  - Compile-time
  - Based on type-properties

```
int data[n];
mpi::Send(data, data+10, dest, tag);
 -> Call MPI_Send Directly

struct gps{ int x; int y;}
// Create the MPI_Datatype
gps data;
mpi::Send(data, dest, tag);
 -> Use the MPI_Datatype to call MPI_Send directly

std::list<int> data;
mpi::Send(data, dest, tag);
 -> Serialize data and then, call MPI_Send
```

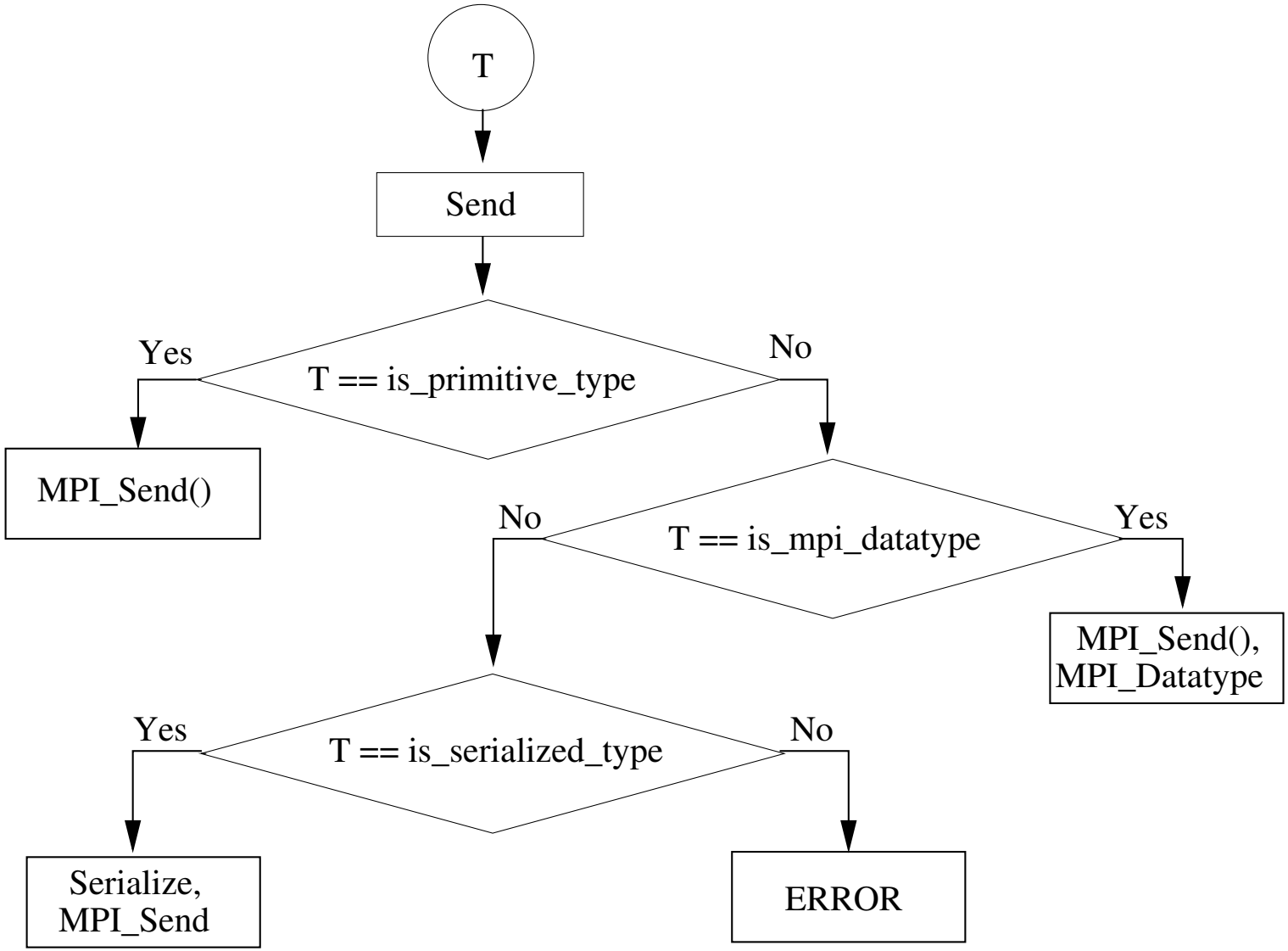
- Choose the best implementation for each type
  - Compile-time
  - Based on type-properties

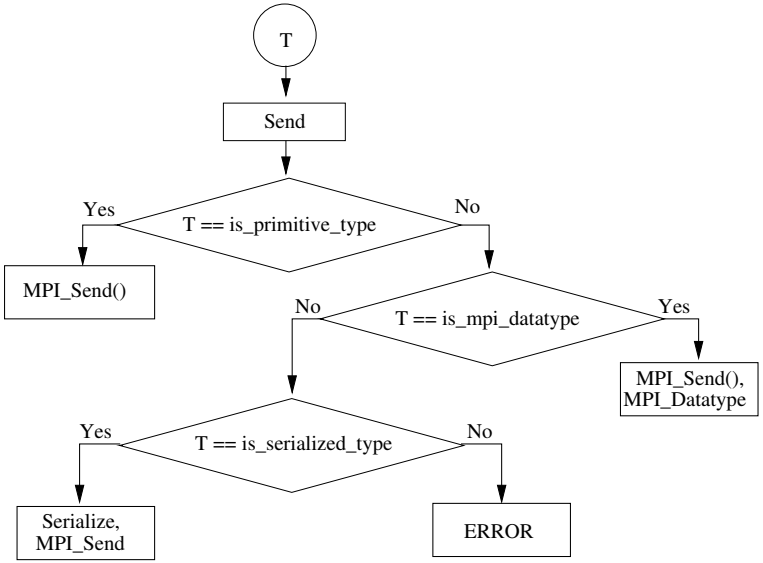
```
int data[n];
mpi::Send(data, data+10, dest, tag);
 -> Call MPI_Send Directly

struct gps{ int x; int y;}
// Create the MPI_Datatype
gps data;
mpi::Send(data, dest, tag);
 -> Use the MPI_Datatype to call MPI_Send directly

std::list<int> data;
mpi::Send(data, dest, tag);
 -> Serialize data and then, call MPI_Send
```

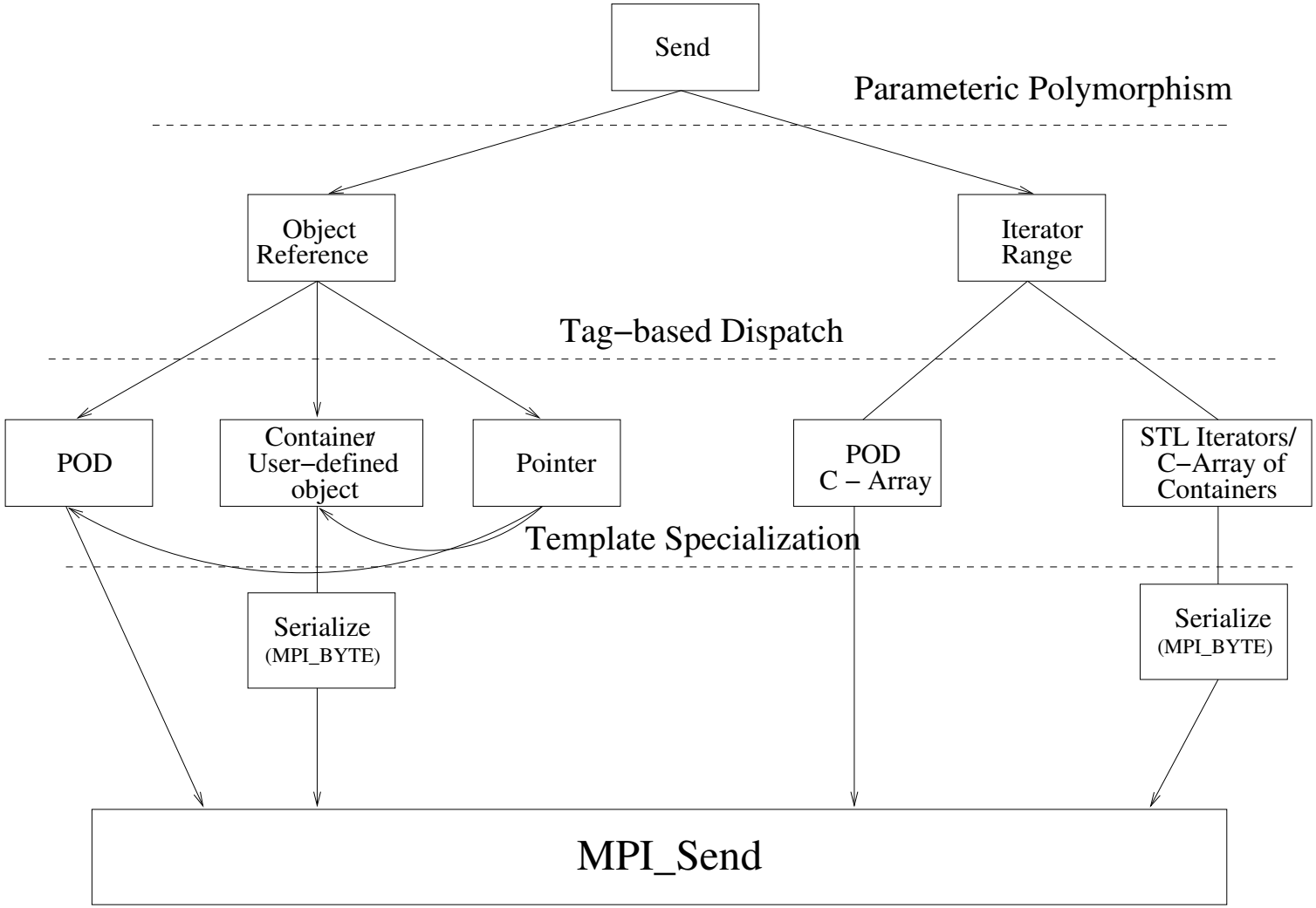
- Choose the best implementation for each type
  - Compile-time
  - Based on type-properties


T
Send
T == is_primitive_type
Yes
No
MPI_Send()
T == is_mpi_datatype
No
Yes
MPI_Send(),
MPI_Datatype
T == is_serialized_type
Yes
No
Serialize,
MPI_Send
ERROR



- Attach type-properties
- Query type-properties
- Non-intrusive

Different pathways for mpi::Send



Performance Results - NetPIPE

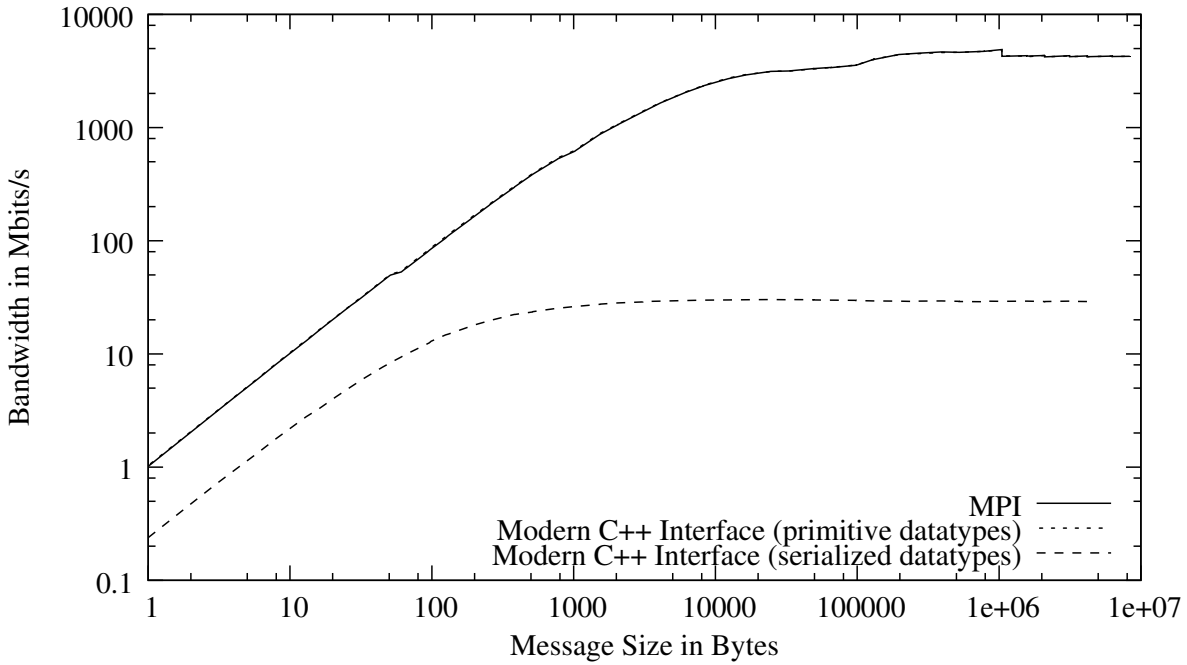


Figure: NetPIPE numbers for our modernized C++ interface to MPI and MPI.

- Dual processor AMD Opteron, 4GB RAM, 1MB cache
- Mellanox Infiniband cards
- RedHat Enterprise Linux (2.6.9-22.0.1.ELsmp), GCC 3.4.4, Open MPI 1.0.1

Conclusions

Our Modern C++ Interface to MPI provides

- Natural C++ syntax for message-passing
- Seamless support for user-defined data types
  - Encouraged by the STL
- Zero abstraction penalty for primitive data types
  - Serialized types incur performance penalty

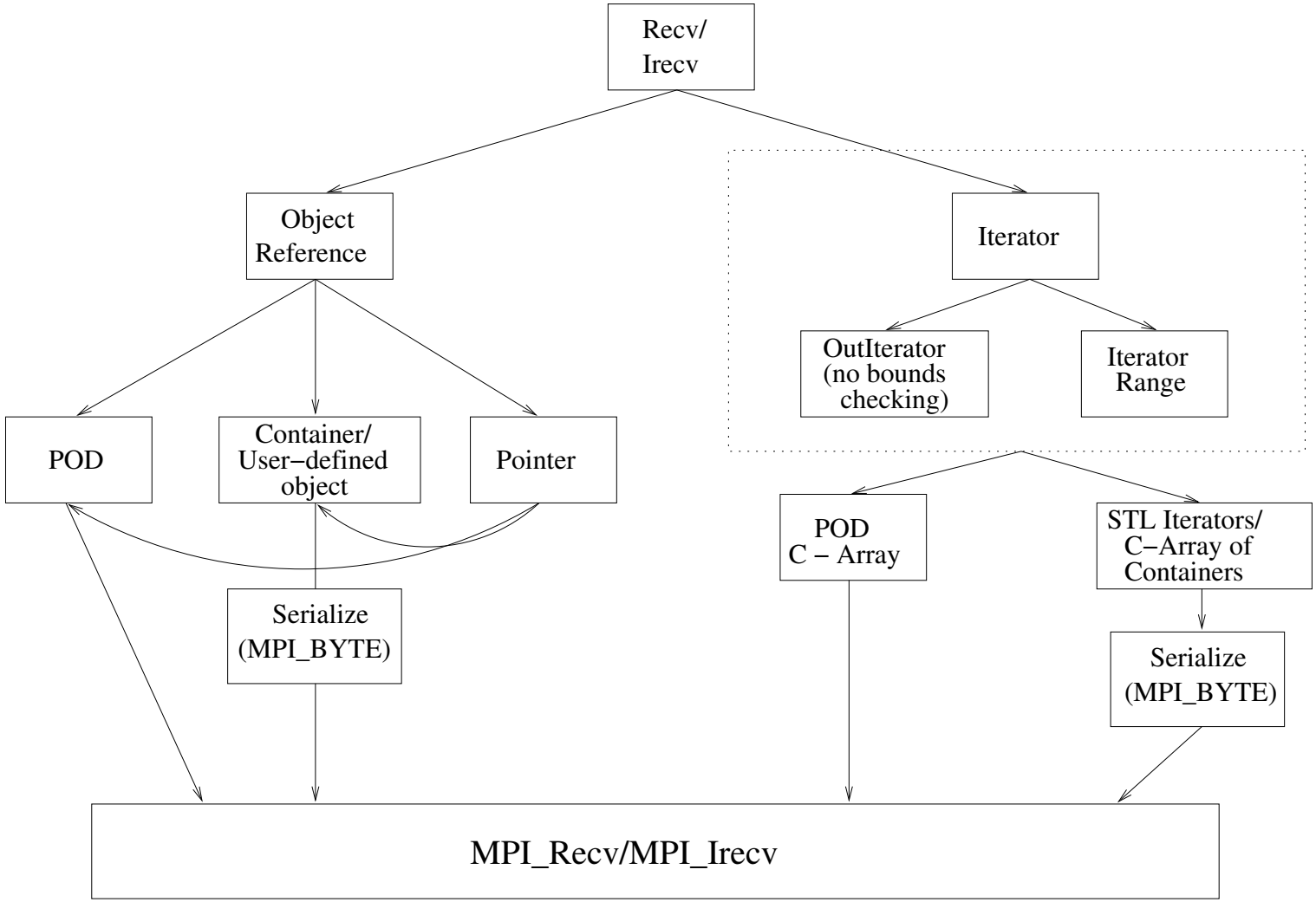
Future Work

- Improve performance of non-blocking operations
- Collective operations on user-defined functors

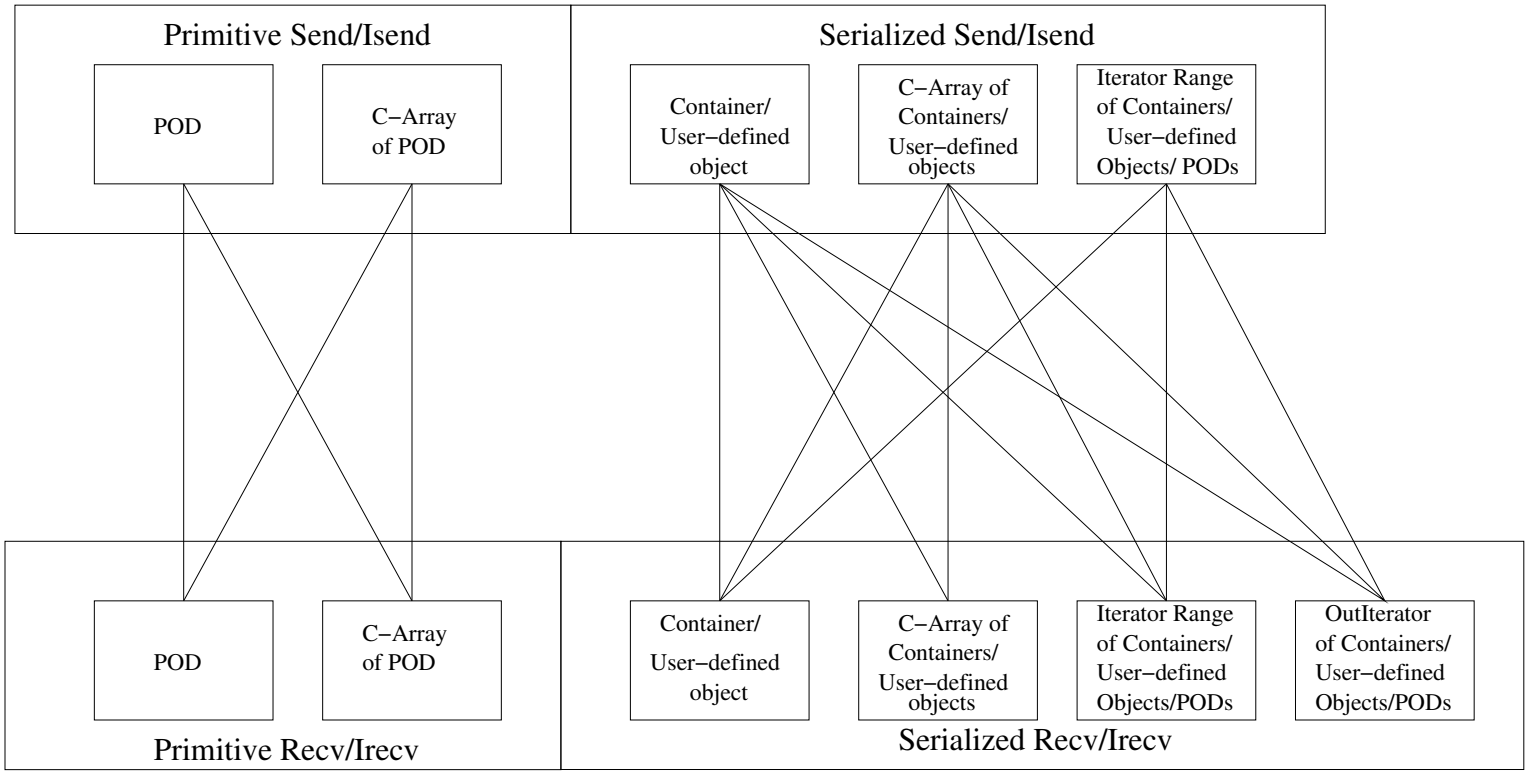
Questions

THANK YOU

Different pathways for mpi::Recv



Matching logic



Type-traits - an example

```
struct primitive_tag {};
struct serialized_tag {};
struct container_tag : serialized_tag {};

template <typename T>
struct object_traits {
    typedef serialized_tag object_category;
};

template <>
struct object_traits<int> {
    typedef primitive_tag object_category;
    static inline MPI_Datatype get_mpi_type() { return MPI_INT; }
};

template <typename T, typename Alloc>
struct object_traits<std::vector<T, Alloc> >{
    typedef container_tag object_category;
};
```

Function Specialization using tag-based dispatch

```
template <typename T>
int Send(const T& var, int dest, tag tag, MPI_Comm comm) {
  typedef typename object_traits<T>::object_category object_category;
  return send_impl(var, dest, tag, object_category());
}

template <typename T>
int send_impl(const T& var, int dest, tag tag,
              MPI_Comm comm, primitive_tag) {
  return MPI_Send ((void*)var, 1, object_traits<T>::get_mpi_type(), dest, tag, comm);
}

template <typename T>
int send_impl(const T& var, int dest, tag tag,
              MPI_Comm comm, serialized_tag) {
  // Serialize var and then MPI_Send
}
```