1 Installation

TPO++ currently supports the following platforms and compilers:

<table>
<thead>
<tr>
<th>Platform</th>
<th>Compiler</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>egcs</td>
<td></td>
</tr>
<tr>
<td>egcs&gt;2.93.4</td>
<td>KAI C++</td>
<td></td>
</tr>
<tr>
<td>Solaris 2.7</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Linux</td>
<td>-</td>
<td>√</td>
</tr>
<tr>
<td>Unicos/Cray</td>
<td>-</td>
<td>√</td>
</tr>
</tbody>
</table>

Get TPO++ from ftp://www-ti.informatik.uni-tuebingen.de/ tpo/tpo.tar.gz and unpack the distribution with gzip -dc tpo.tar.gz | tar xfv -. Change into the TPO directory and run ./configure. configure recognizes several options, described below:

--with-kcpp Enables KAI C++ compiler

--with-comm=<comm> Choose the communication mechanism. <comm> can be set to mpi or socket. If you are using MPICH, set the environment variable MPICHHOME to the base directory of your MPICH installation. Alternatively, you can provide --with-mpich=<path to mpich> to configure. This also implies --with-comm=mpi.

--enable-debug Enables TPO++ debugging modes

--enable-optimize Enables optimization.

After configuration type make to build the TPO++ system. Note that TPO++ requires GNU make (version 3.76.1 has been tested).

The source of TPO++ is contained in the directory src. TPO++ provides a collection of tests contained in test. You can run the tests with cd test; make test. More example applications can be found in examples. TPO++ documentation is contained in doc.

2 Running TPO++ applications

The way application can be started is different between the underlying implementations of TPO++.

2.1 Running the MPI version

TPO++ applications based on the MPI version are started using the usual MPI startup method for the target architecture. Usually, this is:

mpirun -np <N> executable [options]
2.2 Running the Socket version

Since the socket implementation is experimental, there is currently no startup script provided. The executables have to be started on all hosts manually. Startup is defined by several environment variables. For all hosts set:

- **NETLIB_MASTER_PORT** Network port used by the coordinator
- **NETLIB_WORLD_SIZE** Total number of hosts, including coordinator

*Only* for slave hosts, you have to provide the hostname of the coordinator in **NETLIB_MASTER_HOST**.

3 Getting started

Since TPO++ provides an interface syntactically and semantically close to MPI, we assume the reader to be familiar with basic MPI concepts, including communicator, ranks, tag etc. More advanced MPI concepts provided in TPO++ will be explained in more detail.

The usual hello world example in TPO++ can be found below:

```cpp
#include <iostream>
#include <tpo++.H>

int main(int argc, char *argv[]) {
    TPO::init(&argc, &argv);

    cout << "Hello world from host " << TPO::CommWorld.rank() << endl;
}
```

All objects, functions and global variables of TPO++ are encapsulated in a namespace TPO. To use them, you have to provide a prefix TPO::. In the example above, the global initialization function TPO::init is used to start up the message-passing environment. Each host prints “hello world” message and its own communication rank, using the `rank()` method of the global object TPO::CommWorld. An alternative would be to state using namespace TPO to import all TPO++ symbols into the default namespace. For clarity, all our examples use the explicit TPO:: prefix.

4 Sending and receiving

The communicator class of TPO++ provides a couple of methods for sending and receiving data. The most simplest way to communicate is using standard mode blocking send and receives calls. The following simple example shows, how to communicate a C++ basic type in TPO++:
#include <tpo++.H>
int main(int argc, char *argv[]) {
    TPO::init(&argc, &argv);

    if (TPO::CommWorld.rank()==0) {
        double data=3.14;
        TPO::CommWorld.send(data, 1);
    } else {
        double data=0.0;
        TPO::CommWorld.recv(data);
    }
}

On this example, the sender only provides the data to be sent and the rank of
the target host. An optional message tag is omitted and defaults to 0. The full
syntax of the send method is:

class TPO::Communicator {
    template <class T>
    void send(const T& buf, const Rank& dest, const Tag& tag = 0);
};

It works on arbitrary types, provided that TPO++ knows how to send them.
Sending of user-defined types is more elaborated in section 4.4.
The receive method receives by default from any sender and messages with any
tag, so the only argument provided is the data to be received. Further, all receive
function return a object of type TPO::Status containing additional information
about the received message, like the actual sender and tag. In the example the
return status is ignored. Section 4.1 provides details on the status class. The full
syntax of the receive method is:

class TPO::Communicator {
    template <class T>
    Status recv(T& buf,
        const Rank& source = Rank::ANY_SOURCE,
        const Tag& tag = Tag::ANY_TAG);
};

The next example shows, how to send and receive Standard Template Library
containers with TPO++:

#include <tpo++.H>
int main(int argc, char *argv[]) {

TPO::init(&argc, &argv);

if (TPO::CommWorld.rank()==0) {
    vector<double> VD(10);
    fill(VD.begin(), VD.end(), 3.14);
    TPO::CommWorld.send(VD.begin(), VD.end(), 1);
} else {
    vector<double> VD(10);
    TPO::CommWorld.recv(VD.begin(), VD.end());
}

In this example, we send a vector of double from host 0 to host 1. For transmitting a STL container, both, sender and receiver have to provide appropriate iterator pointing to the data to be sent or received. On the sender side, this allows to send subranges of containers using the same send call. On the receiver side, the application must provide enough space to store the received data, and specifies the a target range. This again allows to receive at an arbitrary offset in the container. The requirement to provide space is consistent with MPI semantics. If the receiver runs out of space, the result of the receive operation is undefined, consistent with STL semantics (f.ex. copy).

Currently TPO++ allows to send and receive STL lists, vectors, deques and stacks (?).

A – possible expensive – way to receive data without knowing its size in advance is the use of inserters. Inserters are special iterators allocating the space needed for insertion on demand. TPO++ provides a special net_back_insert_iterator, which allows to receive into an empty container and allocates the space needed. The following example shows this approach:

#include <tpo++.H>
int main(int argc, char *argv[]) {
    vector<
    fill(VD.begin(), VD.end(), 3.14);
    TPO::CommWorld.send(VD.begin(), VD.end(), 1);
} else {
    vector<double> VD;
    TPO::net_back_insert_iterator<vector<double> > result(VD);
    TPO::CommWorld.recv(result);
    }

A – possible expensive – way to receive data without knowing its size in advance is the use of inserters. Inserters are special iterators allocating the space needed for insertion on demand. TPO++ provides a special net_back_insert_iterator, which allows to receive into an empty container and allocates the space needed. The following example shows this approach:
4.1 Status objects

All TPO++ receive methods of the communicator classes return an object of type TPO::Status which provides additional information about the message received. Status objects can be queried about the actual senders rank, the message tag, possible errors and the number of bytes contained in the message.

```cpp
class TPO::Status {
  int count() const; // return the number of bytes received
  Rank source() const; // source rank of the message
  Tag tag() const;    // tag of the message
  int error() const;  // error code of the message
};
```

4.2 Send modes

The MPI standard defines 4 different send modes, normal mode, synchronous mode, buffered mode and ready mode. TPO++ provides the same modes in the communicator class. The communicator method send corresponds to normal mode. For the other modes, TPO++ provides the methods ssend, bsend and rsend, having the same semantics like the MPI calls. To send in buffered mode, the user previously has to provide a user-space buffer to TPO++. To do this, the class TPO::Buffer can be used to create a buffer, which can be made available for TPO++ using Buffer_attach and removed using Buffer_detach. The next example shows how to send in buffered mode:

```cpp
TPO::Buffer my_buffer(1024);

if (TPO::CommWorld.rank()==0) { // send in buffered mode
  TPO::Buffer_attach(my_buffer);
  TPO::CommWorld.bsend(some_date, 1);
  TPO::Buffer_detach();
} else {
  TPO::CommWorld.recv(some_data);
}
```

4.3 Asynchronous communication

4.4 User-defined types

In the general case, to enable a user-defined type for transmission, it must derive from the abstract base class TPO::Message.

```cpp
class TPO::Message {
```
TPO::Message defines the pure virtual methods serialize and deserialize which must be implemented by the user-defined class. They are responsible for sending and receiving the user-defined type, respectively. For both methods, the TPO++ library provides an argument of type TPO::Message_data which is used to marshall the objects data. TPO::Message_data provides insert() and extract() methods, which can be supplied all legal datatypes including containers and other user-defined types. serialize() calls repeatedly insert() to prepare the objects member for transmission, and, symmetrically, deserialize() uses extract() to receive the objects member data. An example may illustrate this:

```cpp
// Dynamic user-defined object
class my_complex : public TPO::Message {
public:
  my_complex() : im(0.0), re(0.0) {}

  void serialize(Message_data& m) const {
    m.insert(im);
    m.insert(re);
  }

  void deserialize(Message_data& m) {
    m.extract(im);
    m.extract(re);
  }

private:
  double re;
  double im;
};
TPO_MARSHALL_DYNAMIC(my_complex);
```

### 4.5 Optimizing the transmission of user-defined types

In the general case, TPO++ cannot make any assumptions about the datatype to be transmitted. The layout in memory can be arbitrary and the number of data elements represented by the type can vary. To optimize communication, the user is allowed to give optimization hints to the TPO++ library. Care must be taken, that the contraints stated by an optimization hint for a type apply to that type, else the result of transmitting the type can be arbitrary, including program failure, since, for reasons of efficiency, TPO++ makes no runtime checks.
TPO++ differs between types having a linear memory layout and types representing data to be transmitted which is contained in non-contiguous memory locations. It further makes a distinction between dynamic data, i.e. types representing data which can be different on every transmission and static data, i.e. types representing constant data during object lifetime.

The most general case corresponds to a marshalling category of \texttt{TPO::MARSHALL\_DYNAMIC} and works for all types, but incurring a possible performance loss for static or trivial types.

The user can tag a type with \texttt{TPO::MARSHALL} to give TPO++ a hint, that all objects of this type occupy the same amount of memory.

The user can tag a type with \texttt{TPO::TRIVIAL} to give TPO++ a hint, that an object of this type has constant in linear memory layout. This special optimization should only be used for fine-tuning application performance, since it breaks the type-safety.

### 4.6 User-defined containers

\texttt{TPO\_CONTAINER}.

### 5 Communicators and groups

Similar to MPI, TPO++ support different communicators, corresponding to different communication layers. The communication done on different communicators does not interact in any way. On startup, TPO++ provides the predefined communicator \texttt{TPO::CommWorld} containing all hosts. The number of hosts and the rank of the current process in a communicator can be queried by its \texttt{size()} and \texttt{rank()} methods.

```cpp
class TPO::Communicator {
    TPO::Rank rank() const;
    int size const;
};
```

The user can create new communicators by declaring variables of type \texttt{Communicator}.

A newly created communicator is empty, if provided no arguments, are can be created from an existing communicator using \textit{duplication}, by providing the communicator as an argument.

```cpp
Communicator my_comm_1; // creates an empty communicator
Communicator my_comm_2(CommWorld); // creates a duplicate of CommWorld
```

The hosts associated with a given communicator can be modified using \textit{group calls}. To do this, a communicator provides a method \texttt{group()} to extract the group of hosts associated with it.
class TPO::Communicator {
    TPO::Group group() const;
};

The group objects, i.e. the hosts represented by the group, can be modified on different ways. Like communicators, groups allow to query its size and the rank of the current host in a group using the size() and rank() methods.

class TPO::Group {
    Rank rank() const;
    int size() const;
};

Groups can be compared. ELABORATE ON THIS. IMPLEMENT THIS BETTER.
Standard set operations, like intersection, difference and merge can be used to create new group from existing groups. They are provided as overloaded operators + for union, * for intersection and - for difference.

class TPO::Group {
    Group operator+(const Group&) const;
    Group operator*(const Group&) const;
    Group operator-(const Group&) const;
};

The next example show, how a new group, representing the hosts in the intersection of two given groups can be created:

    TPO::Group group1, group2;
    TPO::Group new_group;

    new_group = group1 * group2;

Moreover, groups can be compared.
If a group has been modified, the application can create a new communicator for the processes defined in the group using the creation constructor of communicators:

class TPO::Communicator {
    Communicator(const TPO::Communicator&, const TPO::Group&);
};

Communicator my_comm_3(CommWorld, my_group);

This creates a new communicator containg the specified hosts and is a collective call over all hosts in the old communicator. The Group argument must be identical on all calling hosts, and the hosts contained in it must be a subset of the group of hosts associated with the old communicator.
5.1 Error handling

6 Collective communication

TPO++ implements the complete set of MPI 1.2 collective communication. Collective communication can be grouped in the following categories:

- Basic operations: Broadcast and barriers.
- Data exchange operations: Scatter and gather operations.
- Reduction operations: Reductions and scans.

6.1 Basic operations

To use a barrier, all hosts of a given communicator have to call its method `barrier()`.

```cpp
class TPO::Communicator {
    void barrier();
};
```

6.2 Data exchange operations

6.3 Reduction operations

Reduction operations are based on STL-like functional operators. A functional operator defines the operation to execute in its `operator()`. Like MPI user-defined reduction functions, this operator is binary, i.e. takes two inputs to combine and returns the result in its first argument. The functional operators also define a static boolean field `commute` which is true, if the operation can be executed in arbitrary order. An example for a `sum` operators is given below:

```cpp
// user-defined operator for reduce/scan
template <class T>
class sum {
public:
    void operator() (T& inout, const T& in) {
        inout += in;
    }
    static bool commute;
};

template <class T>
bool sum<T>::commute = true;
```
Unlike functional operators used in the STL, it makes no sense to define operators which rely on an internal state, because the operator may be executed on different (unknown) hosts.

A Debugging TPO++

To enable the debugging code, recompile TPO++ with configure option --enable-debug. TPO++ recognizes a couple of debugging switches in debugging mode:

- `tpo:debug-container` Debug transmission of containers
- `tpo:debug-basic-types` Debug transmission of basic types
- `tpo:debug-debug-receive` Debug receive code
- `tpo:debug-debug-send` Debug send code
- `tpo:debug-debug-mds` Debug internal message buffers
Index

Asynchronous communication, 6
Blocking communication, 3
buffered mode, see Send modes
Collective communication, 10
Basic operations, 10
Data exchange operations, 10
Reduction operations, 10
Communicators, 8
Error handling, 10
Groups, 8
Installation of TPO++, 2
normal mode, see Send modes
Point-to-point-communication, 3
ready mode, see Send modes
Reduction operations, 10
Running TPO++ applications, 2
MPI version, 2
Socket version, 3
Send modes, 6
buffered mode, 6
normal mode, 6
ready mode, 6
synchronous mode, 6
Standard Template Library, 4
STL, see Standard Template Library
Supported platforms, 2
synchronous mode, see Send modes
TPO::Communicator, 8
TPO::Communicator::barrier(), 10
TPO::Communicator::Communicator(), 9
TPO::Communicator::group(), 8
TPO::Communicator::rank(), 8
TPO::Communicator::recv, 4
TPO::Communicator::send, 4
TPO::Communicator::size(), 8
TPO::CommWorld, 3
TPO::Group, 8
TPO::Group::operator*( ), 9
TPO::Group::operator+( ), 9
TPO::Group::operator( ), 9
TPO::Group::rank( ), 9
TPO::Group::size( ), 9
TPO::init( ), 3
TPO::Message, 6
TPO::Status, 6
User-defined types, 6