A Proposal for PORTS1

The PORTS working group took the first step toward a common run-time specification for compilers by developing the PORTS0 thread interface. This document proposes a plan for developing a PORTS1 interface for communication, remote procedure call, clocks, and event logging.

The strategy used to develop the PORTS0 thread interface was successful because:

- It started with the best standard available, pthreads.
- A simple subset of the standard that could be implemented on a wide variety of architectures was selected.
- Where pthreads lacked needed functionality, the PORTS group defined a new set of function interfaces.

The PORTS group should follow this same pattern while designing PORTS1.

• Basic Definitions:

The PORTS group should agree on some basic data structures and definitions for discussing the machine model. The group has discussed these constructs several times in the past. Below is a proposed formalization of those constructs.

```
// ******************************************************
// ******************************************************
// BASIC MACHINE DEFINITIONS:
// ******************************************************
// ******************************************************

// A basic machine address, with no type information.
typedef void* LocalPointer;

// An OwnerContext uniquely describes an address space.
class OwnerContext {
 public:
  int nodenumber;
  int contextnumber;
  // Comparing two OwnerContexts is done element-wise, and determined
  // if the two contexts refer to the same address space.
  int operator==(OwnerContext C) {
    return ((C.nodenumber == nodenumber) &&
            (C.contextnumber == contextnumber));
  }
};

// Pointer to any physical memory location (no type info)
class GlobalPointer {
 public:
  LocalPointer toLocalPointer() { return P };
  LocalPointer P;
  OwnerContext O;
};
```

- **Communication**

There are three basic machine communication paradigms reflected in popular commercial machines: shared memory, distributed memory with message passing, and distributed memory with network DMA. These three communication architectures should not be cast into a single universal model. Rather, a compiler designed to generate code for all three types of machines should understand these models and generate the most efficient code for communication between nodes. The compiler should be able to access the lowest standard communication protocol for a given architecture. The next three sections propose a PORTS1 interface for basic communication.

- **Shared Memory**

The PORTS0 interface solved many issues for compilers using shared memory by providing interfaces for mutual exclusion, synchronization, and reentrant library functions. However, allocation of strictly shared and local memory was overlooked. On some shared memory machines, shared memory pages must be specially allocated, and they often have a different cost associated with access. Moreover, strictly local memory may use a different caching strategy. PORTS1 should include the following function interfaces:

```
ports1_SharedMalloc();
ports1_SharedFree();
ports1_LocalMalloc();
ports1_LocalFree();
```

Of course on machines where there is only one interface for allocating and freeing memory, these interfaces will simply be macros to `malloc()` and `free()` respectively. These function interfaces are also needed for machines communicating with network DMA (see below).

- **Message Passing**

The high performance computing community has worked hard to develop a standard for message passing on distributed memory machines. Their effort, MPI, should form the basis for PORTS1 message passing -- just like pthreads formed the basis for PORTS0. For example, IBM's MPI on the SP2 has a very efficient implementation; the various MPI versions of `SEND()` and `RECV()` have been tuned specifically for the SP2 communication switch. Compilers capable of scheduling communication in advance with matched send/recv pairs must be permitted **direct** access to those primitives for optimal performance. The tremendous efforts of the MPI community should be reflected in the PORTS1 interface.

Like with pthreads, only a subset of the immense functionality of MPI will probably be needed for PORTS1. PORTS should assemble a small working group to identify the communication primitives from MPI that should be included in a minimal PORTS1 definition. Since MPI is not currently designed to send messages between user-level threads, the working group should also examine what constructs, like those found in the ICASE *Chant* package, could be provided in a future PORTS2 interface. Such an investigation should also explore possible connections between thread scheduling and

message arrival. The **PORTS1 MPI** working group should circulate their proposed subset of MPI through the ports mailing list. At the next meeting of PORTS, the final list of functions should be decided.

• Network DMA

Network DMA, or *put* and *get* for distributed memory is provided by the Cray T3D and Meiko CS2. CMU also has an experimental low-latency put/get for the Intel Paragon. Other vendors are likely to provide put/get interfaces in the future. However, there is no clear standard for put/get functionality. PORTS1 must propose a generalized put/get interface until a standard gains wide-spread acceptance. Using the definition of *GlobalPointer* provided above, the following functions provide a network DMA interface for PORTS1:

```
// ******************************************************
// ******************************************************
// BASIC NETWORK DMA FUNCTIONS:
// ******************************************************
// ******************************************************

// ******************************************************
// PUT bytes into location designated by globalptr
int ports1_StoreStart(GlobalPointer dst, LocalPointer *src,
                      int length, int *handle);

// ******************************************************
// GET bytes from location designated by globalptr
int ports1_FetchStart(LocalPointer *dst, GlobalPointer src,
                      int length, int *handle);

// ******************************************************
// Check if a "handle" has completed (non-blocking)
int ports1_Probe(int *handle);

// ******************************************************
// Wait (continue polling) on a "handle" until
// it has completed (blocking)
// Implementation suggestion:  This should translate into
// getting moved out of the thread ready queue until the DMA
// completes
void ports_Wait(int *handle);
```

A small group of interested individuals should review the interface suggested above, and circulate a draft for **PORTS1 network DMA**. At the next meeting of PORTS, the final list of functions should be decided.

• Remote Procedure Call

PORTS1 needs an *active message* or *remote procedure call* function interface. Such an interface would provide the minimum functionality necessary to implement asynchronous behavior. While it has been suggested that MPI might provide active message routines in the future, the PORTS group cannot wait. PORTS1 should develop a simple, compact pair of routines for asynchronous communication. The implementations of an RPC-like

mechanism for shared memory, message passing, and network DMA machines will be very different. Some machines will use an event polling loop, while others will use interrupt driven events.

It is assumed that the argument buffer passed to the remote machine may be copied several times during transit. This is due to the basic asynchronous nature of RPC-style events, and is a fundamental cost overhead. However, since the proposed communication interfaces (shared memory, message passing, and network DMA) described above do not require extra buffer copies during transit, long argument buffers should be passed by reference, with a global pointer. Then the handler for the remote context can fetch the argument buffer directly, and without additional overhead.

Using the definition of *OwnerContext* provided above, one possible function interface is shown below.

```
// ******************************************************
// ******************************************************
// BASIC RPC FUNCTIONALITY:
// ******************************************************
// ******************************************************

// ******************************************************
// Request that a handler be executed in the given OwnerContext
int ports1_RemoteAction(OwnerContext C, int type,
                        LocalAddress *buffer,
                        int length, int *handle);

// ******************************************************
// The function that the run-time system executes after
// receiving a RemoteAction request from another context.
void ports1_RemoteActionHandler(OwnerContext asker,
              int type, LocalAddress *buffer, int length);
```

Like the working groups described above, a group of people interested in the functionality and possible implementations of RPC-style communication should be formed. They should form a draft document and circulate the proposal via the mailing list. The function interfaces should be finalized at the next meeting of PORTS.

- **Clocks**

While MPI does provide some clock functions, an interface for clocks that shared memory and network DMA machines can use is required. A simple clock interface should be included in PORTS1. Since high resolution synchronized clocks are vital to performance monitoring, a PORTS1 clock implementation should use the best resolution clocks available on the hardware.

```
// ******************************************************
// ******************************************************
// BASIC CLOCK FUNCTIONS:
// ******************************************************
// ******************************************************
```

```
// *******************************************************
// return a current wall-clock time (in secs.) as a double.
// No correlation to real TOD
double ports1_WallClock();

// *******************************************************
double ports1_UserClock();

// *******************************************************
// Resolution of the user timer in seconds
double ports1_UserTick();

// *******************************************************
// Resolution of the wall timer in seconds
double ports1_WallTick();
```

- **Event Logging**

Basic tracing and event logging should not be an afterthought - tacked onto a run-time specification at the end. Rather, PORTS1 should define basic tracing and profiling functions that can be quickly implemented and used throughout the run-time system in order to amass information about its behavior.

```
// *******************************************************
// *******************************************************
// BASIC TRACING AND PROFILING:
// *******************************************************
// *******************************************************

// Initialize tracing; must be called once per (phys) node
void  ports1_EvInit(char* filename);

// trace event + parameter
void  ports1_Event(long int eventident,  long int parameter);

// terminate tracing; flush event buffer if necessary */
void  ports1_EvClose();

// flush event buffer on request
void  ports1_EvFlush();
```

- **Other Functions**

PORTS1 will also require basic functions to describe and allocate processor and context resources at run time. Those functions will be added to the PORTS1 interface as needed to implement the interfaces described above.