ABSTRACT

Utility computing is becoming a popular way of exploiting the potential of computational grids. In utility computing, users are provided with computational power in a transparent manner similar to the way in which electrical utilities supply power to their customers. To take full advantage of utility computing, an application needs to be mobile; that is, it needs to be able to migrate between heterogeneous computing platforms while it is executing. At present, there are few high-performance computing applications of this sort, and re-engineering legacy codes to be mobile can take enormous effort.

In this paper, we describe the PC3 system, which converts C/MPI programs into mobile programs almost transparently. This system is based on application-level checkpointing, and it uses type information to translate a checkpoint taken on one platform for restart on a different platform. Because C is not a strongly typed language, the system uses a checker that analyzes programs and flags potential problems that may prevent such a translation. To our knowledge, the PC3 system is the first system to provide all of these features.

Experimental results show that our system is efficient. In particular, the overheads for checkpointing sequential programs are comparable to the overheads of using the Condor system for taking system-level checkpoints. Furthermore, we have successfully restarted 64-processor checkpoints, taken on the Lemiuex system at the Pittsburgh Supercomputing Center, on Windows clusters at the Cornell Theory Center, and we have found that the book-keeping overhead of our system is negligible compared to the time required to move checkpoint data between sites on the Internet.

1. INTRODUCTION

Utility computing is becoming a popular way of exploiting the potential of computational grids. Instead of submitting jobs for execution on a particular computer as is done currently, users submit jobs to an agent that (i) finds the necessary resources somewhere in a federation of high-performance computers and high-bandwidth networks, (ii) schedules and monitors the execution of the job at that site, and (iii) migrates the job during its execution to different sites in the federation if it is advantageous to exploit changing resource availability. Although skeptics abound, many companies are investing in this model of grid computing.

To take full advantage of utility computing, an application needs to be mobile; that is, it needs to be able to migrate between platforms while it is executing, as shown in Figure 1. Although applications execute on a single site at any given time, a mobile application will be able to migrate to a different site if more resources become available there. Such applications are also more resilient to hardware faults since they can be restarted on a new site if the site they were executing on suffers a hardware failure.

At present, there are few high-performance computing applications of this sort. However, there are large numbers of existing codes written for traditional parallel machines that might be good candidates for using emerging grid systems. How can these static applications be transformed into mobile applications? One approach would be to transform these applications by hand. Although this has been done for some codes such as the Cactus library [14], transforming codes manually is expensive. A more promising approach is to develop software tools to perform this transformation with minimal manual input. Criteria for success include the following.

- **Transparency**: A software tool that can transform programs more or less automatically to make them mobile would dramatically lower the cost of entry into the grid. We call this property transparency.
In this paper, we describe a system called PC³ (for Portable Cornell Checkpointing Compiler) that converts parallel C programs which use the MPI communication library into mobile programs for computational grids. Our system is based on three key ideas.

- **Application-level checkpointing**: We use a pre-compiler to instrument a C program so that it can save and restore its own state without relying on operating system or hardware support. Unlike system-level checkpointing systems like Condor [20], which are tied to particular architectures and operating systems, application-level checkpointing provides an approach for making programs self-checkpointing and self-restarting.

- **C Checker**: On a given architecture, our system can correctly restart all C programs that conform to the C specification[12]. Further, for a subset of C programs that are “well-typed” (defined in more detail in Section 2), our system can restart checkpoints on a different architecture. In our system, a C program checker analyzes the program and flags potential problems that may prevent the generation of portable checkpoints. If a program passes all the checks, it is guaranteed that the state of the program can be saved and restored in a portable way. Otherwise, non-portable checkpoints are generated.

- **Coordination layer**: To save the state of a parallel program while it is executing, it is necessary to coordinate the state-saving by the different processes. If the program is written in a bulk-synchronous manner, the state of the computation can be saved at global barriers. However, some programs such as many mesh programs self-checkpointing and self-restarting.

The problem of making applications mobile is obviously easier if all platforms are identical (same kind of processors, same number of processors, same operating system etc.). In reality, one rarely finds identical parallel platforms at different sites, so mobility between heterogenous platforms is desirable.

**Performance**: Users who expect a high level of performance on one platform are unlikely to sacrifice performance for mobility. Although there may be cases where this tradeoff is appropriate, applications should retain their level of performance when they are made mobile.

In this paper, we describe a system called PC³ (for Portable Cornell Checkpointing Compiler) that converts parallel C programs which use the MPI communication library into mobile programs for computational grids.

Our system is based on three key ideas.

- **Application-level checkpointing**: We use a pre-compiler to instrument a C program so that it can save and restore its own state without relying on operating system or hardware support. Unlike system-level checkpointing systems like Condor [20], which are tied to particular architectures and operating systems, application-level checkpointing provides an approach for making programs self-checkpointing and self-restarting.

- **C Checker**: On a given architecture, our system can correctly restart all C programs that conform to the C specification[12]. Further, for a subset of C programs that are “well-typed” (defined in more detail in Section 2), our system can restart checkpoints on a different architecture. In our system, a C program checker analyzes the program and flags potential problems that may prevent the generation of portable checkpoints. If a program passes all the checks, it is guaranteed that the state of the program can be saved and restored in a portable way. Otherwise, non-portable checkpoints are generated.

- **Coordination layer**: To save the state of a parallel program while it is executing, it is necessary to coordinate the state-saving by the different processes. If the program is written in a bulk-synchronous manner, the state of the computation can be saved at global barriers. However, some programs such as many mesh programs self-checkpointing and self-restarting.

Unfortunately, general C programs violate these properties. C is a low-level language that exposes the representation of objects to the programmer. Moreover, it is not strongly typed and the static type information cannot be used reliably to translate between different representations of a given type. On the other hand, there is obviously a subset of C programs that satisfy the conditions given above. Therefore, it is necessary to analyze an input program to determine if these two conditions are met.

The problem of deciding whether a general C program satisfies these conditions is of course undecidable. Therefore, we must settle for a conservative approach that identifies some but not all such programs. For example, one solution is to prohibit the programmer from using any construct that might lead to a violation of these conditions. However, this

![Figure 2: PC³ System Architecture](image_url)
solution is too draconian - for example, it rules out the use of the sizeof construct since it reveals information about the representation of types, but this would rule out any C program that performs heap allocation.

Another concern is the time required to execute the decision procedure. A system that uses a powerful inter-procedural alias and pointer analysis might be able to certify more programs than one that uses purely syntactic rules, but this increased precision comes at the expense of increased compilation time.

To make the right trade-off between accuracy of analysis and compilation time, we were guided by the fact that most computational science programs, unlike systems programs, are written so that they can run on most architectures without modification. In particular, our system uses a stronger version of the first condition described above: the type of every data object is fixed when it is created, and the type must be known to our system.

Our system does not do any flow analysis but relies instead on local checks, as described below. Our system raises both errors and warnings. An error is raised when the system cannot generate the information required to translate the checkpoint. In our system, this happens only if the system is unable to determine the type of an object created by a heap allocation routine. A warning is raised when the system detects a potential portability problem, but is still able to generate enough information to perform the translation. In this case, it is up to the programmer to determine if a problem actually exists.

1. Heap allocation: Since we require that the types of all objects be determined at the time they are created, we restrict the use of malloc to certain idioms. In particular, the malloc in the first assignment statement below does not match a permitted idiom and will trigger an error. The malloc's in the second and third assignments show permissible idioms.

   ```c
   p=malloc(100);
   p=(T *)malloc(4*sizeof(T));
   p=(T *)malloc(sizeof(e));
   // e is an expression of type T
   // or an array of elements of type T
   ```

2. Unions: To avoid having to perform flow analysis to track the type of data in a union, our system replaces unions with structures. This is legal if the original C program used unions in a field-disjoint manner. Since C permits the programmer to write data of one type into the union and then read data of another type, this transformation can change the output of the program. Therefore, we raise a warning for all unions. The programmer needs to verify that unions are used in a field-disjoint way or else rewrite the code so that they are.

3. sizeof: The result of the operator is representation dependent. Consider the following program fragment:

   ```c
   x=sizeof(T);
   checkpoint();
   if (x=sizeof(T))
   {
     ...
   }
   else
   {
     ...
   }
   ```

On restart on a different architecture, the invariant that the value of x be equal to sizeof(T) may no longer hold. Our analysis only permits the sizeof operator in limited circumstances such as an argument to malloc as shown above. All other occurrences trigger warnings. A more elaborate solution is to perform flow analysis to ensure that no object in a checkpoint depends on the result of a sizeof operation, but we have not found the need for this.

4. Type Casts: We only permit upcasts and downcasts in accordance with the prefix sub-typing hierarchy [23] (intuitively, this means that appending more fields to a structure type creates a sub-type). All other casts are flagged as warnings since their result can be representation-dependent. Further, we raise a warning for downcasts since they may lead to non-portability. This happens when the static type of the object pointed to (in the cast declaration) is a subtype of the dynamic type of the object actually pointed to and the pointer obtained from the cast is subsequently dereferenced. An example illustrating pointer casts and the warnings our analysis generates is the following.

```c
struct S1 {
  char x;
  int y;
}
struct S2 {
  char c;
  int z;
  float w;
}
int *ip;
struct S1 *p;
struct S2 *q;
...
p=(struct S1 *)q;
  // Upcast, Portable
q=(struct S2 *)p;
  // Downcast, Flag warning
  // May be non-portable
ip=(int *) q;
  // Non-portable cast, Flag warning
```

5. Pointer arithmetic: Arbitrary pointer arithmetic can lead to pointers that cannot be translated on recovery. Our system can only translate null pointers or pointers that point to some object. This is also a serious limitation of all C pointer translation algorithms discussed for portable checkpointing in the literature. In the presence of pointer arithmetic, we may obtain
3. PORTABLE CHECKPOINTING

To save the state of a sequential application in a portable way, we must solve two problems.

1. How do we restart the application from its last checkpoint?
2. How do we use type information to translate application data in the checkpoint?

3.1 Restarting the application from its checkpoint

The first problem arises from the fact that we are doing application-level checkpointing, so we cannot save the program counter as part of the checkpoint. The solution to this problem is well-known and we use the solution described in [4]. It involves instrumenting the application so that it maintains a data structure that tracks function calls and returns. This data structure is stored as part of the checkpoint. The application is also transformed so that on restart, the application reinvokes the sequence of function calls active at the checkpoint, thereby rebuilding stack frames and restarting execution at the point in the program just after the checkpoint.

3.2 Type representations

For each type in the program, our system maintains information about its representation in a structure called its type metric [26]. Our pre-compiler automatically generates the code to generate the type metrics for all the types of a program. This code is portable and it is executed to obtain the type metrics on that architecture. The type metrics for the C types are described below.

- Arithmetic Types: These include the integer and floating-point types. Most modern architectures have standardized representations such as 2’s complement for integers and IEEE single and double precision floating-point for float and double respectively. Since, we also record the endianness of the architecture, the size turns out to be sufficient as a metric. If an architecture uses its own proprietary format, we resort to more specialized metrics.

- Pointer Types: Pointers in C are machine addresses and C implementations usually provide an integer type that has enough bits to hold a machine address. Hence, the type metric for pointers is similar to that of integers.

- Structure Types: C lays out the fields of a structure in the order they appear in the declaration of the type with an optional padding in between fields. Given two adjacent fields of a structure and a platform, the padding between the fields is fixed and is determined solely by the type of the structure. However, the padding might be different for different platforms. Type metric for structures includes the number of fields and for each field, its byte offset from the beginning of the structure and its type-metric (defined recursively). Bitfields can be handled as a special case by specifying the offset in bits.

- Union Types: Currently, we replace all unions with structures and so we do not need a metric for unions.

- Array Types: In C, the elements of an array are laid out contiguously. Hence, the type metric for arrays consists of the type metric of an element of the array and the number of elements.

3.3 Checkpoint and Recovery Mechanisms

Since our input language guarantees that we have complete type information, the data of the application can be

```c
double a[20];
int x;
double *startp,*endp, *ptr;

startp=a;
endp=a+20; //FLAG WARNING
//points to 1-past last element of a

ptr=startp;
checkpoint();
while (ptr<endp) {
    ...
    //compute
    ptr++; // FLAG WARNING
}
```

In this example, the pointer `endp` cannot be accurately translated. Since, it does not lie in the range of any object, it will be regarded as a dangling pointer and will be set to NULL. This would lead to incorrect execution on restart. In response to the warning, the programmer needs to rewrite his code to remove the problem. One way would be to set `endp` to `a + 19` and adjust the condition accordingly.

In addition, as described in [23], in the presence of upcasts and downcasts, pointer arithmetic can violate type information. Let us consider a pointer obtained from an upcast leading to the static type of the object pointed to by the pointer being a subtype of the type of the object actually pointed to. If such a pointer were incremented and then dereferenced, the value obtained would in general expose the representation. However, arithmetic in the presence of casts does not conform with the C specification and so we do not attempt to check for it.

6. Variable arguments: Our system does not currently provide support for application routines that accept a variable number of arguments. Because of this, our system will issue a warning when vararg constructs are encountered. As a special case, we permit calls to system routines that accept a variable number of arguments and that do not affect the integrity of the checkpoint (e.g., `printf`).

```c
double a[20];
int x;
```
regarded as a set of typed objects. Objects may be further
classified as global objects, stack objects or heap objects
depending on where they are located in the address space.

Our system maintains a set of descriptors, one per object.
The descriptor of an object contains the address of the object \((ADDR_o)\), its type metric \((TYP_o)\) and its size \((SIZ_o)\).
A descriptor is added to the set when the object comes into
existence and removed when it is destroyed. In our system,
the set of object descriptors is distributed among the follow-
ing three structures.

- **SVD**: Set of Stack Variable Descriptors organized as a
  stack into which descriptors are pushed and popped as
  the corresponding variables enter and leave the scope
during execution.

- **GVD**: Set of Global Variable Descriptors organized
  as a list into which the descriptors for all the global
  variables are inserted when the program starts up. All
  the descriptors are removed at program termination.

- **HOD**: Set of heap object descriptors organized as a
  Red Black Tree (with the object address serving as the
  key). Descriptors are inserted on allocation operations
  such as `malloc()` and removed when deallocated by
  `free()`.

When a checkpoint is taken, all the objects are stored in
binary mode along with the SVD, GVD and HOD.

On restart, we construct the SVD and the GVD for the
target architecture as part of the execution till the check-
point location. We load the SVD, GVD and the HOD for
the old architecture from the checkpoint. From the HOD
loaded from the checkpoint, we can obtain the number of
objects on the heap and the type metric and size of each
object on the old architecture. We use this information to
recompute the sizes and metrics on the new architecture.
Then, we allocate space for the objects on the new archi-
tecture and construct the new HOD. At this point, we can
start loading each object from the checkpoint as per its de-
scriptor on the old architecture and translating it to the new
architecture as described in Section 3.4.

Because we retain the binary contents of the objects, we
were able to smoothly integrate an alternate recovery mecha-
nism that can only restart on the same architecture. This
mechanism works for all programs that conform to the C
specification. Thus, our system can provide portable migra-
tion for a subset of C programs and non-portable migration
for the rest.

### 3.4 State Translation

On restart on the target architecture, our system knows
for each object its descriptor on the source and target ar-
chitecture. For integer and floating point types, the trans-
lation achieves exact value equivalence. The two descrip-
tors contain all the information the system needs to do the
translation. We check that the range of the type on the
target architecture can contain the value as this could be
an issue in migrating from a 64 bit architecture to a 32 bit
architecture. For the benchmarks used in this paper, we
have not run into this problem. We can have easily have
a per-program migration policy that disables heterogenous
migration of applications that critically depend on the value
ranges for a particular architecture.

For pointer types, the system needs to locate the corre-
sponding object on the new architecture and set its value
to the address of this object. For an object \(o\), let \(ADDR_o\),
\(TYP_o\) and \(SIZ_o\) be the values of the descriptor fields corre-
sponding to the source architecture and let \(ADDR_t\), \(TYP_t\)
and \(SIZ_t\) be the respective values for the target architec-
ture. When a pointer \(p\) needs to be translated, our system
searches the set of object descriptors on the source archi-
tecture to find the object it points to. If for some object
\(o\), \(ADDR_o\) matches the value of \(p\), then the value of \(p\)
on the new architecture is \(ADDR_t\). If the value \(p\) lies within
the range of some object \(o\) which could for instance be a
structure or an array \((ADDR_o < p < ADDR_o + SIZ_o)\).
A recursive search of its type information \(TYP_o\) is used to
infer which sub-object \(p\) points to. The translated value
of \(p\) is then equal to \(ADDR_t\) incremented by the offset of
the corresponding sub-object as obtained from \(TYP_t\). If no
matching object is found, the pointer is set to null.

Structure and array types are translated in a recursive
manner by traversing the type information until we reach
an integer, floating point or pointer type which is handled
as discussed in the preceding paragraphs.

### 3.5 Efficiency Considerations

Since the translation of a pointer needs a search of the ob-
ject descriptor set, it is crucial that we devise efficient data
structures for performing this search. Before the transla-
tion phase, we sort the entries of SVD and the GVD by
address and determine the address range of the globals, the
stack and the heap regions. The HOD is already sorted as it
was serialized from a red-black tree. The sort has a cost of
\(O(N\log N)\) where \(N\) is the total number of objects. When
a pointer needs to be translated, we determine from its ad-
dress which region it points to and do a binary search on
the corresponding SVD, GVD or the HOD. For each object,
searching takes \(O(\log N)\) and so the total cost of pointer
translation is \(O(N\log N)\). It turns out that this scales well
and for large pointer-dense address spaces, we see small over-
heads. Moreover, we incur this cost only on restart after a
migration.

### 4. HANDLING MPI

To checkpoint MPI programs, we must co-ordinate state
saving by different MPI processes. In addition, the check-
point must capture the state of the MPI library, even though
our system does not have access to the code of this library.
Both these functions are accomplished by the coordina-
tion layer shown in Figure 2. By design, this layer sits be-
tween the application and the MPI library, and intercepts
all calls from the instrumented application program to the
MPI library. This design permits us to implement the co-
ordination protocol without modifying the underlying MPI
library, which promotes modularity and eliminates the need
for access to MPI library code, which is proprietary on some
systems. Further, it allows us to migrate easily from one
MPI implementation to another.

#### 4.1 Protocols

The coordination layer supports two protocols, and the
application programmer needs to select the one that is most
suitable for his application.

##### 4.1.1 Barrier Coordination Protocol
The barrier protocol is suitable for programs that are written using the Bulk Synchronous Programming (BSP) model. In the BSP model, programs are divided into supersteps. Each superstep consists of a computation phase followed by a global communication phase followed by a synchronous barrier. All communication initiated within a superstep completes before the completion of the synchronous barrier. For such programs, the barrier provides an opportunity to place a potential checkpoint location.

This protocol is initiated by a coordinator which is typically the process with rank 0. When the coordinator arrives at its potential checkpoint location, it makes a decision about whether a checkpoint should be taken. It broadcasts its decision in a message to all the other processes. Based on the decision of the coordinator either all processes take a checkpoint at that barrier or they do not.

Note that since the broadcast introduces coordination, it is not necessary to have an actual barrier in the MPI program. The barrier protocol is applicable to more programs than those which strictly adhere to the BSP model. It is sufficient if the program is SPMD with the repeating superstep structure and that the potential checkpoint location marks a spot in the superstep at which all communication calls issued previously have completed.

### 4.1.2 Coordinated Non-blocking Protocol

To handle other programming styles that replace barriers with distributed work-queues and data synchronization [2, 7, 19, 22], the co-ordination layer supports the co-ordinated non-blocking protocol described in [4]. This protocol extends to general MPI programs and enables an MPI program to take a checkpoint without global barrier synchronization.

For lack of space, we do not discuss the details of the non-blocking checkpointing protocol but give only a high-level overview. To understand the issues, consider Figure 3. Program execution can be divided into a succession of *epochs* where an epoch is the period between two successive global checkpoints (recovery lines). Epochs are labelled with integers in this figure. In general, messages can be divided into three categories: *intra-epoch messages*, which are sent and received in the same epoch, *late messages*, which are sent in one epoch and received in a later epoch, and *early messages*, which are sent in one epoch and received in an earlier epoch.

This protocol records as part of a global checkpoint the state of each checkpoint as well as a log of all messages that cross the recovery line. On recovery, for each message send call, the sender uses the log to detect if the message is an early message and if so suppresses sending it since the receiver had already received it in the previous epoch. Further, for each message receive call, the receiver checks to see if it corresponds to a late message. If so, the receiver reads its contents from the log as the sender will not send the message after restart since it belonged to the previous epoch. In order to support this, the log records the data of all late messages. These ideas for point-to-point messages are extended to handle collective communication [5].

### 4.2 Portable MPI State Capture

The MPI state of the application consists of two components.

- **In-flight messages**: In-flight messages are a concern only for the coordinated non-blocking protocol. These consist of the late messages which need to be replayed from the log. Early messages are suppressed and their contents do not appear in the log.
- **Opaque objects**: The MPI library includes a number of opaque objects such as requests, statusus, datatypes, communications, etc. The types of these objects are tied to the MPI implementation and are not portable. On restart, we need to ensure that these objects are correctly restored i.e. they represent the state of the communication as it was before the checkpoint.

The logging mechanism described in [27] can be easily extended to capture in-flight messages in a portable manner. We assume that the datatypes that make up the message are those indicated in the MPI communication call. Since the atoms of all datatypes are the elementary datatypes which map to the basic types for C, the data ultimately gets translated using the translation mechanisms for C objects.

To restore opaque objects, our system maintains a log of all operations performed on them. On restart, the log is used to recreate the objects and replay the operations so they get restored correctly. The logging mechanism was described in [27] and can be easily extended for portable checkpointing. We focus on two features that need special attention in the portable context.

#### Derived Datatypes

To support transfer of non-contiguous buffers, MPI supports derived datatypes that allow the user to specify data at non-contiguous locations. New derived datatypes are constructed out of old datatypes (which may in turn be derived datatypes) using constructors. These constructors specify the layout of the old datatypes within the new datatype using a set of displacements from a base address and are non-recursive.

```
typ = < typ1, disp1 >, < typ2, disp2 >, ..., < typn, dispn >
```

In this specification, *typ* is the new derived datatype, *typ1*, *typ2*, ..., *typn* are the old datatypes and *disp1*, *disp2*, ..., *dispn* are the corresponding displacements. The main portability concern is the ability to restore the displacements correctly on the new architecture. MPI provides different flavors of constructors. Depending on the constructor, displacements can be specified in terms of bytes or the number of elements of the old type.

The simplest constructor is *MPI_Type_contiguous* that lays out the elements contiguously and so has no notion of displacements. *MPI_Type_vector* and *MPI_Type_indexed* are composed of elements of a single datatype and specify the displacement in terms of the number of elements of this
constituent datatype. From the description of these constructors in the MPI specification[13], it is reasonable to assume that these derived datatypes would mark sections of an array of elements having the constituent datatype. Hence, the element offsets specified in the datatype constructor would remain the same on the new architecture.

The remaining constructors MPI_Type_indexed and MPI_Type_located specify the displacements of the constituents in bytes. These displacements need to be recomputed. We describe source-to-source transformations that maintain enough logging information to permit translation of these byte displacements. These transformations can be completely automated by our pre-compiler, but we have not implemented them yet.

All displacement computations in MPI are performed on the special integer type MPI_Aint. There are three operations that MPI specifies for creation and manipulation of objects of this type. First, the MPI_Address function is used to transfer an address to a location of type MPI_Aint. Second, the MPI_Type_extent function is used to transfer an extent to such a location. Extents in MPI are analogous to the size that the sizeof operator returns. Third, subtraction of one MPI_Aint location from another can be used to transfer a displacement to a MPI_Aint location. We will not permit arbitrary arithmetic on MPI_Aint locations. A typical example with the three operations is the following.

```c
struct {
    char x;
    int y;
} v;
MPI_Datatype d;
MPI_Aint addr1,addr2,disp, ext;

MPI_Address(&v.x, &addr1);
    // addr1 contains address of v.x
MPI_Address(&v.y, &addr2);
    // addr2 contains address of v.y
disp=addr2-addr1;
    // disp contains displacement of
        // v.y from v.x
MPI_Type_extent(d,&ext);
    // ext contains extent of datatype d
```

Next, we describe a mechanism to log MPI displacement manipulations. We replace the MPI_Aint type with a structure of type PCCC_MPI_Aint. The structure contains a `value` field which corresponds to the original MPI_Aint location and a `usage` field which indicates whether the original location contained an address, a displacement or an extent. It also contains additional fields that maintain logging information for reconstruction on recovery. The source-to-source transformations and the logging that occurs for the above example is illustrated below:

```c
struct {
    char x;
    int y;
} v;
PCCC_MPI_Datatype d;
PCCC_MPI_Aint addr1,addr2, disp, ext;

MPI_Address(&v.x, &addr1.value);
```

5. EXPERIMENTS

To evaluate the performance of our system, we used the NAS parallel benchmarks [21] and the benchmarks art, equake, mcf and vpr from SPEC CPU2000 [8]. We used f2c to convert the benchmarks that were written in FORTRAN to C before instrumenting them with our pre-compiler. Checkpoints were taken within the top level loop in these benchmarks.

5.1 Pre-compiler Warnings

Table 1 shows the number of warnings and errors generated by the checker. No errors or warnings were raised for the C versions of the NAS benchmarks. Although they use both `sizeof` as well as casts to `char *` in calls to I/O routines, our checker filters these out since they do not affect the portability of checkpoints.

In the SPEC suite, mcf and vpr generated a number of warnings. On inspection, we found two pointer arithmetic

<table>
<thead>
<tr>
<th>App</th>
<th>alloc</th>
<th>ptr arith</th>
<th>down</th>
<th>ptr-int</th>
<th>sizeof</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAS</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>art</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>equake</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mcf</td>
<td>0</td>
<td>45</td>
<td>17</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>vpr</td>
<td>11</td>
<td>43</td>
<td>2</td>
<td>0</td>
<td>17</td>
</tr>
</tbody>
</table>

Table 1: Precompiler Error and Warning Statistics

```c
addr1.usage=IS_ADDRESS;
addr1.address=&v.x;

MPI_Address(kv.y, &addr2.value);
addr2.usage=IS_ADDRESS;
addr2.address=&v.y;

disp.value=addr2.value-addr1.value;
disp.usage=IS_DISPLACEMENT;
disp.source=addr1.address;
disp.target=addr2.address;

MPI_Type_extent(d,&ext.value);
ext.usage=IS_EXTENT;
ext.datatype=d;
```

On restart, the contents of the usage field guides the reconstruction. If the value is an address, the pointer translation algorithm from Section 3.4 is used to translate the address field and the value is recomputed using the `MPI_Address` function. If the value is a displacement, the pointers corresponding to the `source` and `target` fields are translated and the new displacement obtained by subtraction. Finally, if the value is an extent, the `MPI_Type_extent` function is used to recompute the extent from the `datatype` field.

Pack and Unpack. Here the sender packs successive objects (potentially having different datatypes) into a contiguous buffer and sends the buffer to the receiver. The receiver receives the data and unpacks the objects. We cannot support the unpack operation on recovery on a different architecture since the packing format may be different. The solution to this problem is to implement our own pack and unpack routines that use a portable format for the data.
Table 2: Checkpoint sizes on X86-LINUX-GNU and USPARC-SOLARIS-SUNWSPRO

<table>
<thead>
<tr>
<th>App</th>
<th>X86ckpt size(MB)</th>
<th>USPARCckpt size(MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bt.A</td>
<td>354.2</td>
<td>354.2</td>
</tr>
<tr>
<td>cg.A</td>
<td>62.5</td>
<td>89</td>
</tr>
<tr>
<td>ep.A</td>
<td>1.13</td>
<td>1.13</td>
</tr>
<tr>
<td>ft.A</td>
<td>469.8</td>
<td>469.8</td>
</tr>
<tr>
<td>is.A</td>
<td>151.0</td>
<td>151.0</td>
</tr>
<tr>
<td>lu.A</td>
<td>50.8</td>
<td>50.8</td>
</tr>
<tr>
<td>mg.A</td>
<td>456.9</td>
<td>456.9</td>
</tr>
<tr>
<td>sp.A</td>
<td>96</td>
<td>96</td>
</tr>
<tr>
<td>art</td>
<td>3.6</td>
<td>3.8</td>
</tr>
<tr>
<td>equake</td>
<td>49.6</td>
<td>60.3</td>
</tr>
<tr>
<td>mcf</td>
<td>99.5</td>
<td>199.0</td>
</tr>
<tr>
<td>vpr</td>
<td>37.6</td>
<td>42.7</td>
</tr>
</tbody>
</table>

operations in mcf and one in vpr that created pointers that were one-past-the-end and one-befor-the-beginning of arrays. Furthermore, in mcf, there were four problematic pointer-integer casts that were obfuscating the type information in the checkpoint. We manually modified the code to eliminate these constructs. The remaining warnings did not require any modifications to the code. Finally, in vpr, there are memory allocation routines that call malloc without specifying the types of the allocated objects. This benchmark used allocation wrapper routines that internally called malloc but did not export the type to the malloc calls. We modified the benchmark to call malloc directly with the associated types at each call site.

5.2 Uniprocessor Results

We conducted experiments on the following platforms.

- X86-LINUX-GNU: Intel Pentium 4 Processor(32 bit, little endian), 3.4GHz, 2GB RAM, 1MB L2 cache, running Red Hat Linux Enterprise with the GNU C Compiler
- X86-WIN-INTEL: Same architecture as X86-LINUX-GNU but running Windows XP Pro with the Intel C compiler
- USPARC-SOLARIS-SUNWSPRO: UltraSPARC IIIi(64 bit, big endian), 1060 MHz, running Solaris 9 with the SUNWSPRO C compiler
- ALPHA-TRU64-HP: Quad-processor 750 Compaq Alpha server ES45 (64 bit, little endian), 1-GHz, 4GB RAM, running TRU64 Unix with the HP C compiler

Note that these are representative of the challenges posed by heterogeneity such as 32 vs 64 bit, little vs big endian, Unix vs. Windows and different compilers. For lack of space, we present only a subset of the measurements we made in our experiments.

Table 2 shows the sizes of checkpoints for different applications on X86-LINUX-GNU and USPARC-SOLARIS-SUNWSPRO. It can be seen that ft and mg save the most data.

We consider the execution times for the following configurations

- Baseline: original code(converted by f2c if written in Fortran),
- 0ckpt: code instrumented for checkpointing but no checkpoints taken,
- 1ckpt: instrumented code as in 0ckpt but taking a single checkpoint roughly in the middle of execution, and
- 1ckpt + restart: code as in 1ckpt but aborting the program after it has taken a checkpoint and restarting it from the checkpoint and running it to completion.

The overhead of instrumentation can be determined by subtracting the times for Baseline and 0ckpt. The overhead of taking a checkpoint can be inferred by subtracting the time for 1ckpt from the time for 0ckpt. The time to restart from a checkpoint can be obtained by subtracting the time for 1ckpt + restart from that for 1ckpt.

For lack of space, we show only the results for X86-LINUX-GNU, and USPARC-SOLARIS-SUNWSPRO in Figures 4, and 5 respectively. On X86-LINUX-GNU, we compare the performance of our system with Condor, which is the most commonly used system-level checkpoint on this platform. Condor does not support checkpointing on the other platforms. Since our approach is based on application-level checkpointing, we were able to checkpoint and restart on the other platforms with minimal effort. For most of the applications and architectures, the failure-free overhead is negligible and less than 1%. On X86-LINUX-GNU, we see an overhead of 5% and 10% for cp and vpr respectively. On USPARC-SOLARIS-SUNWSPRO, we see overheads of 7% and 9% on art and vpr respectively. Some of this overhead appears to arise from the inability of the back-end compiler to optimize the code after it has been instrumented by our system.

The checkpoint and restart overhead can be estimated from the bar for 1ckpt and 1ckpt + restart in Figures 4 and 5. As can be seen, these overheads are comparable to those of the Condor system. Furthermore, our system allows restart on a different architecture. In Figure 6, we show the time taken to restart and run to completion on X86-LINUX-GNU, starting from checkpoints taken on different platforms. It can be seen that the overheads of translation are low, and that the cost of restarting from a checkpoint taken on a different architecture is not significantly higher than the cost of restarting from a checkpoint taken on the same architecture.

Table 3 shows the time spent in the recovery routines. In this experiments, checkpoints were taken on the Sun UltraSPARC and restarted on the X86-LINUX-GNU system. The cost of moving data between machines is not shown in this table, but it dominates the translation and recovery costs since the bandwidth we obtained using the department NFS was roughly 3.7 Mb/sec.

5.3 Parallel Results

Since the SPEC codes are sequential, we used only the NAS parallel benchmarks in the parallel experiments. We used the barrier checkpointing protocol in all the experiments. The platforms we used were the following.

- LEMIEUX: 3000 node Lemiux cluster at the Pittsburgh Supercomputing Center running TRU64 Unix. Lemiux comprises quad-processor 750 Compaq Alpha server ES45 nodes running at 1-GHz with 4GB
Figure 4: Execution Times on X86-LINUX-GCC

Figure 5: Execution Times on USPARC-SOLARIS-SUNWSPRO

Figure 6: Time to Complete after restarting on X86-LINUX-GNU from checkpoints taken on X86-LINUX-GNU(native), X86-WIN-INTEL, USPARC-SOLARIS-SUNWSPRO and ALPHA-TRU64-HP

Table 3: Timing of recovery routines on X86-LINUX-GNU. Recovery from checkpoint taken on USPARC-SOLARIS-SUNWSPRO

<table>
<thead>
<tr>
<th>App</th>
<th>ckpt size(MB)</th>
<th>restore pc(s)</th>
<th>restore data(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bt.A</td>
<td>354.2</td>
<td>0.15</td>
<td>15.2</td>
</tr>
<tr>
<td>cg.A</td>
<td>89</td>
<td>0.001</td>
<td>0.43</td>
</tr>
<tr>
<td>ep.A</td>
<td>1.13</td>
<td>0.001</td>
<td>0.21</td>
</tr>
<tr>
<td>ft.A</td>
<td>469.8</td>
<td>0.001</td>
<td>11</td>
</tr>
<tr>
<td>is.A</td>
<td>151.0</td>
<td>0.02</td>
<td>4.8</td>
</tr>
<tr>
<td>lu.A</td>
<td>50.8</td>
<td>0.05</td>
<td>2.1</td>
</tr>
<tr>
<td>mg.A</td>
<td>456.9</td>
<td>0.15</td>
<td>10.5</td>
</tr>
<tr>
<td>sp.A</td>
<td>96</td>
<td>0.05</td>
<td>4.3</td>
</tr>
<tr>
<td>art</td>
<td>3.8</td>
<td>0.04</td>
<td>0.09</td>
</tr>
<tr>
<td>equake</td>
<td>60.3</td>
<td>1.25</td>
<td>0.9</td>
</tr>
<tr>
<td>mcf</td>
<td>199.0</td>
<td>0.018</td>
<td>6.8</td>
</tr>
<tr>
<td>vpr</td>
<td>42.7</td>
<td>0.6</td>
<td>1.05</td>
</tr>
</tbody>
</table>

Table 4 shows the sizes of checkpoints on the Lemieux system for different numbers of processors. The sizes of checkpoints on the VELOCITY cluster are approximately the same for all codes other than CG. For CG, the checkpoint sizes per processor on VELOCITY are 122MB, 89.9MB and 29.6MB for 4, 16 and 64 processors respectively.

We used the MPI versions of the NPB benchmarks to evaluate our system on 4, 16, 64 and 256 processor jobs. On Lemieux, we used all four processors on a node, while on Velocity, we used only one processor per node because of memory limitations. As a consequence, we could not run the 256 processor job on VELOCITY.

To test the porting of parallel jobs between different platforms, we restarted Lemieux checkpoints on the Windows RAM/node and connected via a Quadrics interconnection network. We used the native HP C compiler on Lemieux. The native MPI implementation on this platform is based upon MPICH and uses ELAN as the low-level communication library across nodes.

- VELOCITY : 128 node Velocity II cluster at the Cornell Theory Center running Windows XP. Each node is a Dual Processor 2.4GHz P4 Xeon with 2 GB RAM/Node, 72GB Disk/Node and 512KB Cache/Processor(SMP). The nodes are connected via Force10 Gigabit Ethernet. We used the GNU C compiler on this platform. The native MPI used was MPIPro Version 1.7.0.
Table 4: Checkpoint sizes in MB per processor for 4, 16, 64 and 256 processor runs on Lemieux. The corresponding problem classes are indicated in brackets.

<table>
<thead>
<tr>
<th>App</th>
<th>4PROCS</th>
<th>16PROCS</th>
<th>64PROCS</th>
<th>256PROCS</th>
</tr>
</thead>
<tbody>
<tr>
<td>bt</td>
<td>82.8(A)</td>
<td>272.1(C)</td>
<td>76.5(C)</td>
<td>268.0(D)</td>
</tr>
<tr>
<td>cg</td>
<td>173.7(B)</td>
<td>128.1(C)</td>
<td>42.5(C)</td>
<td>255.9(D)</td>
</tr>
<tr>
<td>ep</td>
<td>1.1(B)</td>
<td>1.1(C)</td>
<td>1.1(D)</td>
<td>1.1(D)</td>
</tr>
<tr>
<td>ft</td>
<td>117.4(A)</td>
<td>469.8(C)</td>
<td>117.5(C)</td>
<td>469.8(D)</td>
</tr>
<tr>
<td>is</td>
<td>151.0(B)</td>
<td>151.0(C)</td>
<td>37.8(C)</td>
<td>18.9(C)</td>
</tr>
<tr>
<td>lu</td>
<td>51.5(B)</td>
<td>53.8(C)</td>
<td>16.5(C)</td>
<td>59.2(D)</td>
</tr>
<tr>
<td>mg</td>
<td>120.1(B)</td>
<td>233.0(C)</td>
<td>456.9(C)</td>
<td>120.1(D)</td>
</tr>
<tr>
<td>sp</td>
<td>106.9(B)</td>
<td>114.0(C)</td>
<td>40.1(C)</td>
<td>139.6(D)</td>
</tr>
</tbody>
</table>

Figure 7: 64 processor execution times on Lemieux

Figure 8: 256 processor execution times on Lemieux

Figure 9: 16 processor execution times on VELOCITY

Figure 10: 64 processor execution times on VELOCITY

Figure 11: Time to restart and complete on VELOCITY from checkpoints taken on VELOCITY(native) and LEMIEUX(portable) on 16 processors. Our results show that the major bottleneck in this style of utility computing is the cost of transferring data between sites. To reduce this cost, it is imperative to reduce the amount of saved state, and to find faster solutions for transferring data between platforms.

6. RELATED WORK

Alvisi et al [9] is a good survey of work on checkpointing and related techniques like message-logging. Most checkpointing systems in the literature use system-level checkpointing (SLC), so they do not provide a solution to the problem of moving state between heterogenous platforms. The most commonly used checkpointing system in grid environments is Condor [20], which takes system-level checkpoints of sequential programs. Examples of other system-level checkpointing systems are Libckpt[25] and Starfish[1].

Previous systems for portable checkpointing for C programs include Porch [26], April [11], HPCM [6], TUI [28] and MigThread [16]. These systems use type-information similar to PC3 although the details of their implementation.
are different. Of these, Porch, April, HPCM and MigThread use an automated pre-compiler approach. The TUI system relies on modifications to the compiler to provide type information. Unlike PC3, they do not automatically detect programs that they cannot handle. In particular, an application that conforms to the C specification may successfully be compiled by these systems, but may fail to recover correctly. MigThread uses a combination of flow analysis and runtime checks to track the accuracy of the included type information with regard to unions and pointer-integer casts. However, it does not check for other portability violations that C program execution can incur as described in Section 3. Further, these systems do not handle MPI programs.

Systems like Cyclone [17] and CCured [23] have been designed to provide memory safety guarantees for variations of the C language. The problem of portable checkpoints is different from memory safety. Memory safety addresses potentially hostile system code and attempts to catch out of specification behaviour such as accesses outside array bounds, null-pointer dereferences and jumps to non-existent locations. CCured deals with the legacy C applications by adding runtime checks that focus primarily on pointer dereference violations. It is not concerned with preserving type information for non-pointer data. Cyclone is a type-safe variant of C that includes an enhanced type-system, exceptions and region-based memory management.

SRS [29] by Vadhiyar and Dongarra is a checkpointing infrastructure that uses the Internet Backplane Protocol [24] for storing and migrating checkpoints. Its application-level checkpointer allows the programmer to manually specify the data that needs to be shared as well as its distribution. On recovery, the system uses this information to recover the program’s state and redistribute the data on a potentially different number of processors. The Dome system [3] is a C++ library of data-parallel objects that supports checkpointing and restart using application level checkpointing. All objects that need to be checkpointed are defined as special Dome objects capable of saving their state in portable XDR format to a checkpoint file. Legion [18] is another system that uses an object-oriented framework. Legion provides naming and location services so that applications programs can be written without any awareness of where objects reside. To allow objects to be migrated between heterogeneous resources, Legion requires that every object provide save_state() and restore_state() methods. The legacy applications that we are targeting are not written using these frameworks.

Finally, there are several ongoing efforts to build more reliable implementations of MPI, which are orthogonal to our work. LA-MPI [15] tolerates link failures by using multiple communication paths, retransmission etc. to increase the reliability of communication channels. FT-MPI [10] is an implementation of MPI that can detect processor and communication failures, and inform the application.

7. CONCLUSIONS

In this paper, we have described the PC3 system for migrating C MPI programs in computational grids. The PC3 pre-compiler automatically transforms the input program to save its own state making its use transparent to the application programmer. It can portably checkpoint a subset of C and includes a checker that flags program constructs that may cause portability problems. The system uses type information to save the state of the MPI processes in a portable manner, and uses either barrier or non-blocking protocols for coordinating the MPI state. We also describe source-to-source transformations for checkpointing MPI derived datatypes. Our experimental results indicate that the overheads of the system are less than 2% on average for the benchmarks used in the paper.

Our checker is purely syntactic, so it may raise false warnings. In ongoing work, we are implementing a more powerful checking mechanism. We believe that more advanced static analyses can reduce the number of reported errors and warnings substantially. We are also considering inserting low-overhead runtime checks to do complete certification of the portability of the application. We currently do not check that calls to libraries do not violate the type information of the program. We enforce a restricted, well-typed interface to some routines in the C Standard Library such as malloc. We would like to extend this mechanism to handle interfaces to other libraries.

The sizes of the MPI checkpoints can get very large as the problem sizes and the number of processors are increased (as high as 100GB). The major bottleneck to mobility is in transferring these large checkpoints over the network. One way to reduce the checkpoint size is to save only the live objects. Further, some objects may be recomputed cheaply on recovery and so need not be included in the checkpoint. We are investigating static analyses that can identify such objects. We are also interested in technological solutions for improving the transfer bandwidth.

Finally, to improve the utilization of grid systems, it would be useful to be able to restart on a different number of processors. We are using over-decomposition as a mechanism to distribute the computation into more fine-grained threads. On restart, we plan to re-distribute the per-thread portable checkpoints on the target configuration to ensure good load-balancing.

8. REFERENCES


