Imprecise Exceptions in Distributed Parallel Components

Kostadin Damevski and Steven Parker

School of Computing, University of Utah, Salt Lake City UT 84112, USA

Abstract. Modern microprocessors have sacrificed the exactness of exceptions for improved performance long ago. This is a side effect of reordering instructions so that the microprocessor can execute instructions which were not to be executed due to an exception. By throwing more circuits at the problem, microprocessors are designed so that they are able to roll back to the instruction causing the exception. However, some microprocessors, like the HP Alpha, do not roll back and impose a paradigm of inaccurate exceptions. This decision can reduce circuit complexity and increase speed. We propose a similar method of handling exceptions in a component environment that achieves high performance by sacrificing exception accuracy when dealing with parallel Single Program Multiple Data (SPMD) components. The particular domain this design is intended for is high performance computing, which requires maximum resource use and efficiency. A performance-centric way to handle exceptions is explained as well as additional methodology to enforce exception strictness if required.

1 Introduction

The component software abstraction is one that has been leveraged in multiple computing environments. One such environment is that of high performance scientific computing (see [1]). High performance computing is an environment that, more so than others, emphasizes optimizing the performance of the application. In order to achieve the maximum possible performance and leverage distributed computing potential, the execution of an application may be performed in parallel by multiple parallel computing threads. Typical parallel algorithms divide the problem space in some fashion among the available computing threads, which work in concert to solve the problem, sometimes communicating part of the developing solution to each other. These algorithms often employ the Single Program Multiple Data (SPMD) programming paradigm.

In a component environment, multiple computing threads can be coupled to form one component called an SPMD parallel component. This collection of computing threads is intended to work collaboratively to solve some problem. In order to provide interaction between SPMD parallel components in a distributed environment, collective method invocations are used from a proxy[1] to a parallel

[1] Proxies are objects used to represent other objects. All requests to the proxy are forwarded to the represented object. They are used frequently in distributed mid-

component. Collective invocations are defined as ones which require the involvement of each computing thread. That is, all callee computing threads receive an invocation for the same method and all caller threads make an invocation. Therefore, a method invocation launches all computing threads that proceed to execute the same method on separate pieces of data (as the SPMD paradigm suggests). The movement and separation of the data are problems that have been discussed in literature [3, 4, 6, 2].

Parallel components have already been introduced by multiple component frameworks intended for high performance applications. However, these frameworks differ slightly in the way parallel component interaction is defined. For instance, the Parallel Remote Method Invocation (PRMI) mechanism of the PARDIS framework is indirect so that a parallel component's invocation to another parallel component is serialized before it is sent over the wire and deserialized afterwards. That is, the data between components is transferred through a single communication link [Figure 1 (left)]. The authors of PARDIS later acknowledged the performance penalty of the serialization [7] and provided direct, multiport component interaction [Figure 1 (right)]. The improved performance of the direct method, in this case, was to be expected as this method provides better utilization of the network and imposes less synchronization on the computing threads. Our system, SCIRun2, leveraged the previous work in this field and was built with collective invocations that provide direct communication between each parallel computing thread (as it is needed).

Another important aspect of collective invocations is the level of synchronization imposed in their implementation by the underlying system. In order to provide a guarantee of collectiveness, systems often enforce a barrier on the collective invocation. This unarguably provides better error reporting, but at a very significant cost of execution speed, which is unacceptable for many high performance computing applications. Because of this, SCIRun2 and other equivalent systems extend unsynchronized behavior as much as possible. The imprecise exceptions described here, only make sense in a direct and mostly unsynchronized component framework. In fact, their main goal is to loosen the synchronization reigns that regular implementation of exceptions would require.

2 Problem Description and related work

This direct mode of invocation and communication introduces our nemesis when dealing with exceptions. Specifically, what if only one computing thread on the invoked (callee) component throws an exception? This exception will be propagated quickly to one of its proxies on the invoking (caller) component, which would cause one of the invoking threads to throw an exception. The rest of the invoking threads are not aware of the exception as they do not participate in the invocation on the excepting thread. The result is that the invoking component has an incoherent state, possibly destroying the collective behavior of the caller.

dleware in order to invoke a method on a component. In this context, proxies are synonymous to caller components.

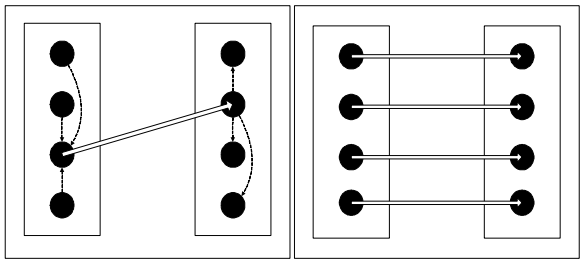

Fig. 1. Serialized and multiport (direct) collective invocation of parallel components

The user is not in control of which invoking thread has received the exception and is left with a mess to clean up. There is little useful that could come out of this situation. One possibility is to do away with exceptions altogether. Another is to provide the user with a necessary framework to propagate the exception to all the processes of the invoking component. The latter approach has an additional problem: due to the communication lag and the general unsynchronized nature of the processes making the invocation, the exceptions may not arrive exactly during the method invocation. We call these imprecise exceptions. Appropriate thread synchronization can be in place to mitigate this problem; however synchronization will always hurt the performance of the application. Instead we introduce imprecise exceptions, allowing the programmer to trade off accuracy of the exception and performance.

One category of exceptions that CORBA supports are those that represent various network and system failures. These are termed system exceptions. Conversely, user purpose exceptions are explicitly defined in the IDL via a throws clause in each method. While CORBA's user exceptions occur only on the server side, system exceptions can occur in both client and server. An exception hierarchy of classes also exists. The base classes impose methods on the derived classes which report useful information about the exception, such as whether or not method completed executing or not. However, exception inheritance of any sort is not allowed within the CORBA IDL.

Imprecise exceptions have been suggested for the Haskell functional programming language [5]. This is a language that achieves high performance and flexibility through lazy evaluation of expressions. Adding imprecise exceptions to Haskell provides the stack unwinding error reporting mechanism of exceptions, without sacrificing rich transformations or other strengths of the language. Some of our ideas are similar to the ideas presented by Peyton-Jones et al. [5], although the domain is quite different.

3 Design

This work will provide the utility of component exceptions in a distributed SPMD component setting, while attempting not to sacrifice performance. In

doing this we ought to emphasize that very little additional overhead is imposed in cases in which an exception does not occur. If an exception does occur, it is thrown and the application incurs a performance penalty (as it should).

Let us consider an example that enables us to propose a design for imprecise parallel component exceptions. In discussing this design, we will use the following interface (written in Scientific Interface Definition Language (SIDL) [8]):

```
interface SomeThrowSomeNo {
   void exceptionOnemethod() throws exceptionOne;
   void exceptionTwomethod() throws exceptionTwo;
   void noexceptionmethod();
}
```

The SomeThrowSomeNo interface contains two methods that throw exceptions and one method that does not. The *noexceptionmethod* method has no implementation. The two exception-throwing methods have the following implementation:

```
void exceptionOnemethod() {
   if (rank == 1) throw exceptionOne;
}
void exceptionTwomethod() {
   if (rank == 2) throw exceptionTwo;
}
```

For the purposes of this discussion, we consider two SPMD parallel components of three parallel threads each; one component will be the caller and the other the callee. We assume that a proxy has already been created by the caller component. Each thread of the caller component has a parallel proxy and is able to invoke a method on any callee thread. However, the collective interaction dictates that all callee threads should be invoked only once. Therefore, we have a one-to-one correspondence in the collective invocation between the threads of the parallel component. For the sake of clarity let's assume that the caller thread whose rank is zero invokes the method on the zero ranked callee thread, rank one invokes on rank one, etc.

3.1 Reporting Imprecise Exceptions

Upon the execution of this SPMD code:

```
stsn-proxy->exceptionOnemethod();
stsn-proxy->noexceptionmethod();
```

The caller thread ranked one receives an exception message back from its invocation. Threads zero and two have continued execution to some other part of the code. We are concerned that threads zero and two eventually receive information about the exception that occurred. Thread one has the responsibility

of informing the other threads of the exception in *exceptionOnemethod*. It does so through an asynchronous message to each of the other threads. After this, thread one can throw the C++ equivalent of *exceptionOne* and goes into its appropriate catch block (if it exists).

All remote method invocation stubs contain code to check for the existence of asynchronous messages notifying of exceptions in other threads. In the scenario we discussed, we assume the threads are perfectly synchronized (i.e. they execute the same instruction at the same time) and an asynchronous message travels at lightning speed. Upon the invocation of *noexceptionmethod*, threads zero and two will instantaneously detect the existence of *exceptionOne* and will throw it within their context. At this time all threads would have excepted and reached a steady state.

Asynchronous messages provide the appropriate behavior since they do not impose synchronization of the processes in order to communicate the exception and because reads for these messages come by at an extremely small cost. Performing a check for an exception message within all remote invocations may be unnecessary at times. At other times, it would be overzealous. Overall, we feel that providing the check through each proxy use strikes a perfect balance in most cases. The cost of checking for an asynchronous message is extremely small (typically checking a global flag) and negligible even if there are many frequent uses of the proxy. However, if the opposite is true and proxy use is very infrequent it could decrease the granularity of the exception checks to an unacceptable limit. We discuss this problem in the next section. The extreme example of this case is when there are no proxy uses after an exception and no opportunity to report this exception. This problem is considered in Section 5.

3.2 getException method

We realize that it is unfair to assume lightning speed message delivery as we did before. Therefore, if the threads are not synchronized, we should be prepared to see *exceptionOne* thrown sometime later than the invocation of the *noexceptionmethod*. In many cases this is not acceptable. Executing *noexceptionmethod* may depend on the successful finish of *exceptionOnemethod*.

If the programmer is unable to count on exceptions to occur within a specific invocation, an exception can crop up too late or too early and provide the wrong behavior. In order to allow the programmer to control where exceptions occur in the code, we provide a *getException* method which is invoked on the proxy (similar to Peyton-Jones et al.'s use of the same method). Once we invoke *getException* we guarantee that all exceptions that have occurred have been propagated to this thread. The only real way to ensure this in a parallel and distributed environment is to synchronize the caller threads. That is, this method requires a barrier and as such imposes a performance penalty. Despite this, it is useful for it provides the programmer with a wider range of uses for exceptions.

4 Corner Cases

Although the design presented above is straightforward, the complex interaction between two parallel components requires that a number of non-trivial scenarios be analyzed.

Consider what would happen if the implementation of *exceptionOnemethod* were something like this:

```
void exceptionOnemethod() { throw exceptionOne; }
```

Collective invocation to this method would result with each caller thread receiving a separate exception. When this occurs, threads should ignore the asynchronous exception messages reporting exceptionOne in order not to throw the same exception multiple times. To do this, we keep an exception log which makes sure that we are aware of exception we have already produced. In doing this we have to make sure that we keep enough information to help us determine the right course of action. We use a combination of tags to uniquely identify each proxy invocation. The guarantee that all collective methods are invoked by all computing threads can be relied on to uniquely tag each collective invocation without imposing additional communication.

Let us consider the following example where method invocations are executed by the caller component using the SomeThrowSomeNo interface:

```
stsn-proxy->exceptionOnemethod();
stsn-proxy->exceptionTwomethod();
```

Both *exceptionOne* and *exceptionTwo* have been thrown in this scenario. Caller thread one and two respectively are the first ones to receive each exception. Let us suppose that they both contact thread zero at about the same time so that thread zero has received messages about two distinct exceptions. Now thread zero has a dilemma about which exception to report: *exceptionOne* or *exceptionTwo*. Here are some possible ways to resolve this problem:

• Allow non-determinism. This allows for either exception to be thrown. It is probably an improvement over no exceptions being thrown, however it is definitely not a clean solution as the collective nature of the caller is not preserved, and the application may except differently in each run. This may confuse the programmer and result with unwanted behavior.

• Always report the user exception before the system or vice versa. This may prove to be effective in some situations. However, this choice would work poorly when the confusion is between two user and two system exceptions.

• Report the first method's exception first. Using the information tags that we collect per each proxy invocation, we can easily make this determination. This would provide consistent behavior and error reporting which would be of use to a programmer. We opt for this solution.

This scenario also requires that thread rank two comes to its senses and throws *exceptionOne* instead of *exceptionTwo*. In order to provide this behavior,

we should synchronize and determine the correct exception between all the exceptions that have been witnessed among all of the threads. This includes each thread that comes in direct contact with an exception. It should not rush and throw an exception before it makes sure that it is the correct exception to throw. In order to perform this, our implementation modifies the tournament barrier algorithm to determine the best exception to be thrown and propagate the result to all the threads. This algorithm is also applied in the implementation of the *getException* method, since similar exception election is needed there as well. Detailed discussion of the tournament barrier and other barrier types can be found in [9].

One situation not captured by our running example is one where a collective invocation of the same method yields two different exceptions caught by separate caller threads. This is a situation that is not likely to occur often. We propose a random yet consistent choice of which exception to throw. The exception reporting mechanism will use a consistent way of choosing to throw the same exception across different runs or compilations and in all of the computing threads. In our system exceptions are related to exception IDs assigned by the compiler roughly representing the order of the exceptions in the throws clause. The lower exception ID is chosen to resolve this confusion. This provides consistent behavior to all runs of the code compiled by our IDL compiler. As every compiler and parallel component architecture could make this determination differently (we do not propose that everyone uses exception IDs), we fully expect that a different exception may be raised in the same situation in a different environment. This is something that the parallel component programmer should be prepared to face. Another possibility would be to prefer the exception type thrown by the lowest (or highest) rank.

The exception reporting mechanism relies on the checks performed by the proxy upon each method invocation. If no method invocation existed in the executing code we could fail to report an exception. In order to remedy this, we could impose a *getException* call upon the proxy destructor. This would guarantee that exceptions are reported and is a viable solution. However, if proxies are created and deleted often, we would significantly constrain the performance of the application by synchronizing the computing threads upon each invocation to *getException*. Another possibility is to save the actual deletion of the proxy until the very end of the program. Synchronization at the end of a program will not impact performance. Our suggestion is to perform a *getException* synchronization within the Object Request Broker[2] (ORB) destructor. The ORB's deallocation by the destructor typically occurs when main finishes on a particular machine.

An imprecise exception can come before, during, or after the excepting method invocation in caller threads that do not directly receive an exception. So far, we have mostly discussed exceptions arriving after the collective method invocation to the exception method is finished. We have not yet considered the case of them

[2] Object Request Broker exists on each node and manages requests among components. An ORB should be deallocated when all component interactions on one node cease.

arriving before. Exceptions arriving before the invocation are very awkward to face. In essence, we have been given a glimpse of the future and we respond to this sight even though we have not yet reached this point of execution. The best alternative is to rule this case out altogether using the proxy infrastructure we have already described. This comes at almost no extra cost and does not impact performance. We can hold off and ignore the existence of a specific exception from an asynchronous message until we encounter the method which has produced it. When we reach this method we do not invoke it but throw the exception at that time.

5 Conclusions

We have presented a paradigm of imprecise exceptions for parallel distributed components. Based on the fact that less synchronization improves performance, we claim that this approach saves us from a drastic execution speed penalty with exceptions. We examined the difficult scenarios that are possible with imprecise exceptions. We also provided a way to synchronize imprecise exceptions so that they provide the same behavior as regular exceptions. This imprecise exception mechanism is targeted at a component-based distributed Problem-Solving Environment, but could be applied to a variety of scenarios involving multiple communicating SPMD parallel programs.

6 Results

The performance improvement of our design is somewhat self-validating. We provide an exception handling method that allows a greater degree of asynchrony, which in turn betters performance. In spite of this, we implemented a simple example and provided some runtimes for it in order to show that our concepts work and to exemplify how much of a performance increase we can achieve.

The use scenario we chose involves a parallel query tool sending queries to a parallel data server. We implemented these as parallel components and tested various modes of exception use: total imprecise exceptions, use of the *getException* method after each query, and use of regular (precise) exceptions. Assuming the time to process the query is close to zero, here are the results of this test (between two Pentium4 class machines connected by Gigabit Ethernet):

The results were largely what we expected. The performance increase of the imprecise exceptions scales as the degree of parallelism increases. It is important to note that the imprecise exceptions we measured do not make use of the *getException* method. This total asynchrony is unrealistic in practice; therefore the difference in performance would be slightly less exaggerated and peaking at the numbers we have shown above.

Table 1. Average time for a single query (averaged for 10000 queries).

Instances of Query Tool	Imprecise Exceptions	getException()	Precise Exceptions
1	658.39	675.89	672.91
2	978.55	1487.46	1595.33
3	1924.08	3482.97	4238.06
4	1705.58	6882.35	8899.77

Times are in microseconds.

7 Acknowledgements

The authors would like to acknowledge Matthew Flatt for initially pointing us to Haskell exceptions. This work was funded by the Department of Energy Center for Component Technology for Terascale Simulation Software (CCTTSS) and by NSF ACI-0113829.

References

1. R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a Common Component Architecture for High-Performance Scientific Computing. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, 1999.
2. F. Bertrand, Y. Yuan, K. Chiu, and R. Bramley. An approach to parallel MxN communication. In *Proceedings of the 3rd Los Alamos Computer Science Institute (LACSI) Symposium*, October 2003.
3. K. Damevski and S. Parker. Parallel remote method invocation and m-by-n data redistribution. In *Proceedings of the 3rd Los Alamos Computer Science Institute (LACSI) Symposium*, October 2003.
4. G. A. Geist, J. A. Kohl, and P. M. Papadopoulos. CUMULVS: Providing fault-tolerance, visualization and steering of parallel applications. In *Environment and Tools for Parallel Scientific Computing Workshop*, Domaine de Faverges-de-la-Tour, Lyon, France, August 1996.
5. S.L. Peyton Jones, A. Reid, F. Henderson, C.A.R. Hoare, and S. Marlow. A semantics for imprecise exceptions. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 25–36, 1999.
6. K. Keahey, P. K. Fasel, and S. M. Mniszewski. PAWS: Collective invocations and data transfers. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computation*, July 2001.
7. K. Keahey and D. Gannon. Developing and evaluating abstractions for distributed supercomputing. *Journal of Cluster Computing, special issue on High Performance Distributed Computing*, 1(1), 1998.
8. S. Kohn, G. Kumfert, J. Painter, and C. Ribbens. Divorcing language dependencies from a scientific software library. In *Proceedings of the 10th SIAM Conference on Parallel Processing*, Portsmouth, VA, March 2001.
9. J.M. Mellor-Crummey and M.L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.