Fault-Tolerant Parallel Applications with Dynamic Parallel Schedules: A Programmer's Perspective

Sebastian Gerlach, Basile Schaeli, and Roger D. Hersch

Ecole Polytechnique Fédérale de Lausanne (EPFL),
School of Computer and Communication Sciences,
Station 14, 1015 Ecublens, Switzerland
dps@epfl.ch

Abstract. Dynamic Parallel Schedules (DPS) is a flow graph based framework for developing parallel applications on clusters of workstations. The DPS flow graph execution model enables automatic pipelined parallel execution of applications. DPS supports graceful degradation of parallel applications in case of node failures. The fault-tolerance mechanism relies on a set of backup threads stored in the volatile storage of alternate nodes that are kept up to date by both duplicating transmitted data objects and performing periodical checkpointing. The current state of a failed node can be reconstructed on its backup threads by re-executing the application since the last checkpoint. A valid execution order is automatically deduced from the flow graph. The addition of fault-tolerance to a DPS application requires only minor changes to the application's source code. The present contribution focuses on the development of fault-tolerant parallel applications with DPS from a programmer's perspective.

1 Introduction and Related Work

Clusters of commodity workstations are rapidly growing in size and complexity as computation power requirements increase. The large number of computing nodes incorporated within a cluster dramatically increases the likelihood of node failures during program executions. Therefore, ongoing research focuses on graceful degradation and the continuation of program execution despite individual node failures.

In the context of message-passing systems, two major classes of recovery schemes have been proposed: checkpoint-based and message log-based recovery [8].

Checkpoint based approaches store the current state of computation to stable storage. Coordinated checkpointing on all participating nodes [16] may be achieved by stopping in an ordered manner all computations and communications, and performing a two-phase commit in order to create a consistent distributed checkpoint. Checkpointing can also be performed independently on

J. Kohlas, B. Meyer, and A. Schiper (Eds.): Dependable Systems, LNCS 4028, pp. 195–210, 2006.

all participating nodes (uncoordinated checkpointing). This removes the performance bottlenecks induced by the global synchronization required for coordinated checkpoints and allows checkpointing at convenient times, for example when the data size associated with a checkpoint is very small. Several checkpoints need to be stored on each node, and a consistent state from which to restart has to be found when a failure occurs [4]. In unfavorable situations, the recovery can lead to the domino effect, where no consistent checkpoint other than the initial state can be found. In order to eliminate the domino effect, additional constraints on checkpointing sequences need to be introduced, for example based on the applications' communication patterns [17].

Message logging approaches store in addition to checkpoints all the messages flowing through the system. The logged messages allow bringing a node to any given state by re-executing its application code with the corresponding sequence of logged input messages. Three types of message logging are usually considered: pessimistic, optimistic and causal. Pessimistic logging logs every received message to stable storage before processing it. This ensures that the log is always up to date, but incurs a performance penalty due to the blocking logging operation. The penalty can be reduced by using specific storage hardware, or by using sender-based message logging [13][5]. Optimistic logging begins processing messages without waiting for a successful write to stable storage [15]. The overhead of pessimistic logging is removed, but several messages might be lost in case of failures. When the system is restarted it must roll back to a previous consistent state on all nodes. Finally, causal logging also provides low overhead and limits the backtracking that has to be performed during recovery. It does however require the construction of an antecedence graph for messages, and requires a rather complex recovery scheme [9].

These mechanisms make no assumptions about the internal structure of the applications other than the use of message passing for communications between processes. They are thus very well suited for applications written with general-purpose message passing libraries such as MPI [7]. When parallel applications are described using high-level approaches, additional information about the structure of the application is available. For example, task graphs [6] or Calypso [2] make use of such information for recovering and resuming computation after a failure. In order to keep the fault-tolerance mechanism efficient, the application developer often needs to provide specific hints or to use certain constructs within the application. These modifications are required for example in order to allow an application to be restarted from a stored checkpoint.

Fault-tolerance schemes also vary in the assumptions they make about the number and nature of failures that can be recovered. Placing additional limitations on the recoverable cases may enable significant optimizations when compared to the general case. For example, if the system has never more than one failure at a time, stable storage can be replaced with transfers to neighboring nodes [14]. Such a scheme has the advantage of allowing the application to recover without having to fetch data from the stable storage of the failed node.

Dynamic Parallel Schedules (DPS) is a high-level framework for developing parallel applications [10]. The DPS framework supports fault-tolerance using a combined message logging and checkpointing approach [11]. The fault-tolerance mechanism uses the parallel application's structure exposed in the application's high-level description in order to hide most of the complexity of fault-tolerance from the application developer. However, the support for fault-tolerance is not fully transparent; the developer needs to take some specific requirements into account. In the present paper, we describe the implications of developing fault-tolerant applications from a developer's perspective.

2 The Dynamic Parallel Schedules Framework

DPS applications are defined as directed acyclic graphs of operations. The fundamental types of operations are leaf, split, merge and stream operations. The inputs and outputs of the operations are strongly typed data objects. Figure 1 illustrates the flow graph of a simple parallel application, describing the asynchronous flow of data between operations.

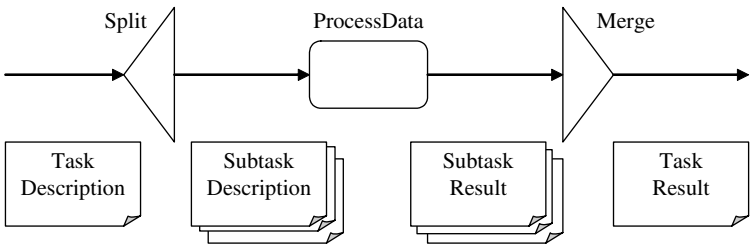

Fig. 1. Flow graph describing data distribution (split), parallel processing, and collection of results (merge)

The split operations are used to divide the incoming data objects into smaller objects representing subtasks. These subtasks are subsequently sent to the next operations specified by the flow graph (e.g. *ProcessData*). The leaf operations process the incoming data objects, and produce one output data object for each input data object. The merge operations are used to collect the results into a single large output object. Once all the results corresponding to the data objects originally sent by a split operation have been collected, the larger resulting data object is sent out. Successive data objects arriving at the entry of a split operation yield successive new instances of the split-merge operation pair.

The stream operations combine a merge operation with a subsequent split operation. Instead of waiting for the merge operation to receive all its data objects before allowing the subsequent split operation to send new data objects, the stream operation can stream out new data objects based on groups of incoming data objects. Stream operations allow programmers to finely tune their processing pipeline and therefore to ensure a maximal utilization of the underlying hardware.

All operations, including split and merge operations are extensible constructs where the developer provides his own code to control how processing requests or data are distributed, and how processed sub-results are merged into one result. The data objects circulating in the flow graph may contain any combination of simple types or complex types such as arrays or lists. The following source code shows a typical implementation for a split operation within DPS, where a task is split into smaller parts.

```
class Split : public dps::SplitOperation
  <SplitInDataObject, SplitOutDataObject> // Data object types
{
  IDENTIFY(Split)
public:
  // This method is called when the input data object is received
  void execute(SplitInDataObject *in)
  {
    // Split task into NB_PARTS small parts
    for(Int32 splitIndex=0;splitIndex<NB_PARTS;splitIndex++)
    {
      SplitOutDataObject *sot=new SplitOutDataObject();

      // Fill the output data object with meaningful data

      postDataObject(sot);
    }
  }
};
```

Other operations are implemented by deriving them from other base classes depending on their functionality, such as *dps::LeafOperation* or *dps:: MergeOperation*.

Operations within a flow graph are carried out within threads grouped in thread collections. Figure 2 illustrates the distribution of flow graph elements into thread collections for a simple compute farm application. Two thread collections are created. The first, *MasterThread*, handles the global split and merge operations, and contains only a single thread. The second, *WorkerThreads*, handles the parallel computation, and contains one thread for each compute node.

A DPS thread is a logical construct representing an execution environment for a set of operations. In data parallel applications, data is stored within threads that are distributed across the available compute nodes. Threads are implemented as standard C++ objects. Figure 3 shows an example of a grid-based data structure distributed on 3 threads. Each thread stores additional data in order to enable neighborhood dependent computations. DPS threads are mapped to operating system threads, although not necessarily in a one-to-one relationship. For instance several DPS threads residing on a single processor node may share a single operating system thread.

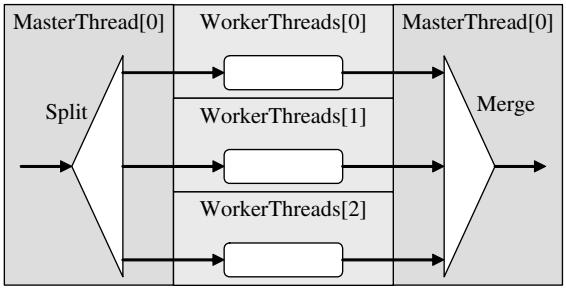

Fig. 2. A flow graph and its associated thread collections

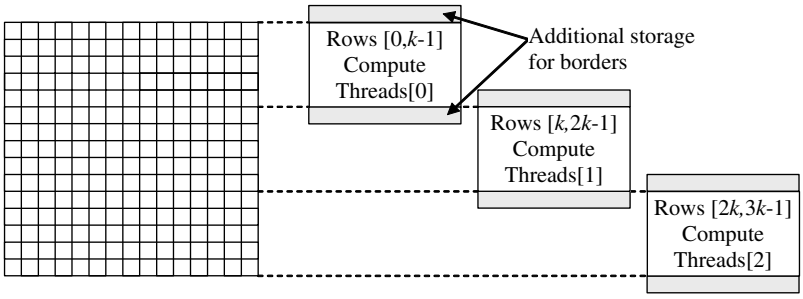

Fig. 3. Distribution of a grid-based data structure on 3 threads, each thread also storing copies of its neighboring grid lines (borders)

The selection of the thread within a thread collection on which an operation is to be executed is accomplished by evaluating at runtime a user defined routing function attached to the corresponding directed edge of the flow graph. Communication patterns such as the neighborhood exchanges illustrated in Figure 4 required for updating a distributed data structure (Figure 3) can easily be specified by using relative thread indices. The first part of the flow graph ensures that all nodes have sufficient neighborhood information available, and the second part performs the computation on all nodes. The intermediate synchronization ensures that the global state remains consistent.

By transferring data objects as soon as they are computed, and maintaining queues of arriving data objects, execution of DPS applications is fully pipelined and asynchronous. Data object queues are associated with the thread that contains the operations that will consume them. This macro data flow behavior enables automatic overlapping of communications and computations. In order to limit the size of the data object queues stored in threads, DPS provides a flow control mechanism that can be used to limit the number of data objects in circulation between a split operation and the corresponding merge operation. The flow control mechanism suspends the split operation until the processed data objects have been received by the corresponding merge operation.

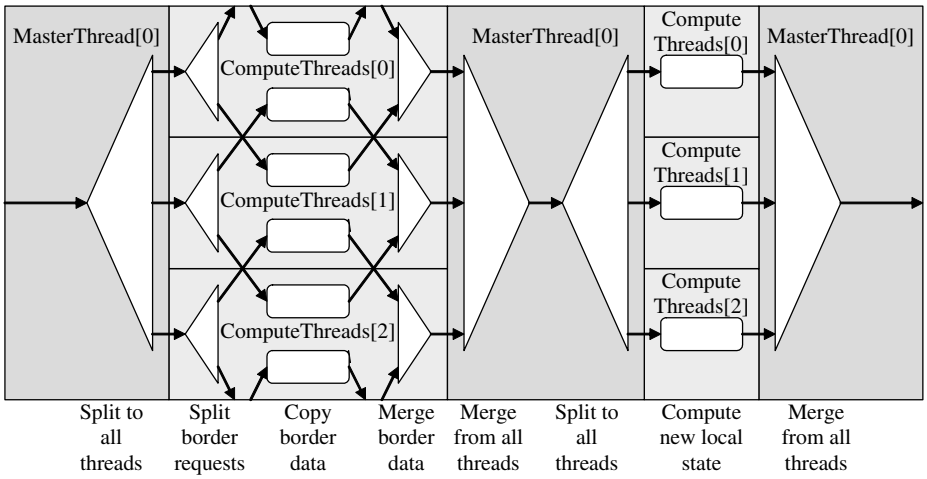

Fig. 4. A flow graph for one iteration of an iterative neighborhood-dependant parallel computation

The flow graph together with its collections of threads and its routing functions forms a parallel schedule. A parallel schedule describes a fine to medium-grained parallel application. Its operations represent the small subtasks that are executed in a pipeline-parallel manner according to the flow graph. The DPS communication layer, hidden from the application programmer, relies on TCP sockets, and uses an optimized data serialization scheme that minimizes memory copies.

3 Fault-Tolerance in DPS

DPS provides a fault-tolerance mechanism that allows applications to continue execution despite node failures. The fault-tolerance mechanism is implemented by providing a scheme for the recovery of flow graph program execution segments located on a failed node. The scheme is composed of two distinct recovery mechanisms. The first general purpose mechanism enables the reconstruction of the state of a thread upon node failure. The second specialized mechanism is an optimization for threads that do not store any local state information.

DPS detects node failures by monitoring communications. A node is considered to be failed when it is not able to communicate with another node. The TCP/IP network layer used by DPS reports failures when communications fail or disconnections occur.

3.1 General Purpose Recovery Mechanism

The general purpose mechanism relies on a set of backup threads that are mapped onto an alternate set of nodes as illustrated in Figure 5. When a data

object is sent to an operation on a given thread, a copy is also sent to the backup thread. Upon occurrence of a failure, the current state of the threads that were on the failed node is reconstructed on the backup threads by re-executing operations. The valid execution sequence of operations is automatically deduced from the flow graph of the corresponding DPS application by applying a simple data object numbering scheme.

Operations are assumed to be deterministic, i.e. for a given initial thread state and a set of incoming data objects, they will always produce the same output. This assumption is necessary to ensure that the reconstruction on a backup thread will yield a state identical to the state that was present on the failed thread.

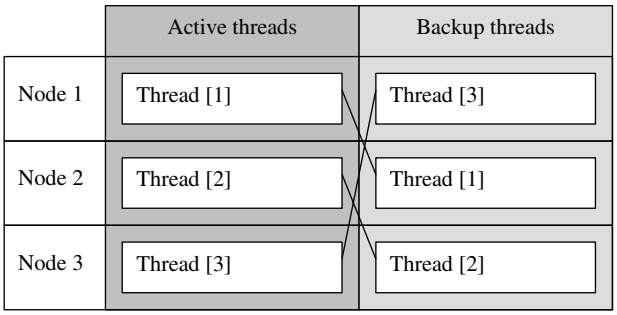

Fig. 5. Mapping of a thread collection with backup threads

In order to shorten the reconstruction time of a failed node, one may replicate the state of the active threads onto the corresponding backup threads (periodic checkpointing). When the backup thread is subsequently used for reconstructing the thread state after a failure of the active thread, reconstruction is initiated from the replicated state rather than from the initial state. Each DPS thread has three components that must be conserved for successful reconstruction: the current local thread state, the queue of data objects that wait for processing, and the state of suspended operations within that thread. Since replicating the current state also removes part of the pending data object queue on the backup thread, it reduces the memory requirements on the backup nodes. This checkpointing operation can be carried out asynchronously and independently on all individual threads. Independent checkpointing of individual threads enables the compute nodes to remain potentially busy during the checkpointing process by executing operations attached to other threads mapped to the same node. The effective overhead induced by stopping a thread in order to checkpoint can therefore be kept very low.

Since both active and backup threads are stored in the volatile storage of the processing nodes, only a single copy of the thread is left after a failure. In order to ensure that the application can survive successive failures, it is necessary to rapidly create a new backup thread for the remaining copy. The new backup

thread is created by checkpointing the surviving thread copy immediately after activation, in order to minimize the time during which the application is fragile. This general-purpose fault-tolerance mechanism allows the computation to continue as long as for each thread within every thread collection either the active thread or its backup thread remains valid.

3.2 Recovery for Threads Without Local State Data

For threads that do not store any local state data (*stateless* threads), the recovery mechanism can be simplified. If the general purpose mechanism would be used, the backup threads would store only the duplicated data objects. It is therefore more efficient not to send out the duplicate data objects, but rather to keep them on the sender node. Since the operations running on stateless threads do not use any local state, these operations can be executed on any thread. If a stateless thread fails, it is removed from the thread collection. The sender node resends the data objects to another thread in the collection. The execution of the application can continue as long as at least one thread remains valid within the stateless thread collection.

The flow graph provides information about the runtime execution patterns of applications, allowing the framework to transparently select the appropriate recovery mechanism for the graph segments. For compute bound applications, the fault-tolerance overheads during normal program execution remain low thanks to the asynchronous communications that occur in parallel with computations. A detailed description of both fault-tolerance mechanisms and the associated performance overheads can be found in [11].

4 Implementing Fault-Tolerance

The following sections focus on the elements that must be taken care of by the developer of a fault-tolerant application. As an example, we use two applications: a compute farm application, where a master node distributes computation tasks onto worker nodes, and a complex application with a distributed state that is updated iteratively. These applications use the flow graphs illustrated in Figures 2 and 4 respectively. The DPS fault-tolerance mechanism is presented in two steps. The first step aims at adding fault-tolerance, allowing the application to survive multiple failures. The second step ensures an efficient reconstruction process by enabling checkpointing.

4.1 Simple Compute Farm Applications

A fault-tolerant compute farm application needs to be able to survive two types of failures: the failure of a worker node, and the failure of the master node. Since the worker threads do not store any local data, these threads can be handled by the specialized sender-based stateless thread recovery mechanism provided by DPS. Since this mechanism simply redistributes the unprocessed worker tasks to the surviving worker threads, no changes are required in the source code

implementation of the application. Therefore, when the application is running, any node other than the one running the master thread can fail at any time. As long as one worker node remains active, the program execution is unaffected.

Fault-tolerance on the master thread is important, since this thread is running the split and merge operations. At least one backup thread needs to be added to the mapping of the thread collection *MasterThread*. This will allow the master thread to be reconstructed on other nodes participating in the computation, ensuring successful completion if the initial master thread fails. The backup thread is simply created by adding a list of valid backup nodes to the mapping of the master thread collection:

```
masterThread.addThread("node1+node2+node3");
```

In this example, the master thread is located on *node1* and its backup thread on *node2*. The third node *node3* will take over the role as backup if either of the other nodes fails in order to ensure support for multiple subsequent failures.

On a master node failure, the split operation is restarted from the beginning, and all processing requests are sent again. The routing function does not necessarily return constant results for a given data object when the total number of threads varies, some data objects will get routed to different nodes on re-execution, and part of the computation may possibly be performed again. Those data objects that are resent to the same nodes will be caught by a mechanism for eliminating duplicate data objects [11]. This additional reconstruction overhead can be reduced by periodically checkpointing the main thread, i.e. by replicating its current state to the backup thread as described in section 5.

4.2 Applications Storing a Distributed State

Applications that store local data within their computation threads need backup threads. For example, let us consider an application using a thread collection *computeThreads*, containing three computation threads mapped onto nodes *node1*, *node2* and *node3*. Each thread needs to have at least one backup thread. In order to ensure that the thread collection can survive failures until a single node is left, we use all nodes as backups for each thread, creating a round-robin mapping as shown in Figure 6. The proposed mapping can be obtained with the following mapping string:

```
computeThreads.addThread
  ("node1+node2+node3 node2+node3+node1 node3+node1+node2");
```

This mapping ensures that any two nodes may fail without preventing the application from completing successfully. The thread mapping strings (*"node1+node2+node3 node2+node3+node1 node3+node1+node2"*) with round robin mapping of backup threads may be generated automatically by the DPS framework [12]. In order to ensure acceptable reconstruction times, it is again necessary to perform periodic checkpointing.

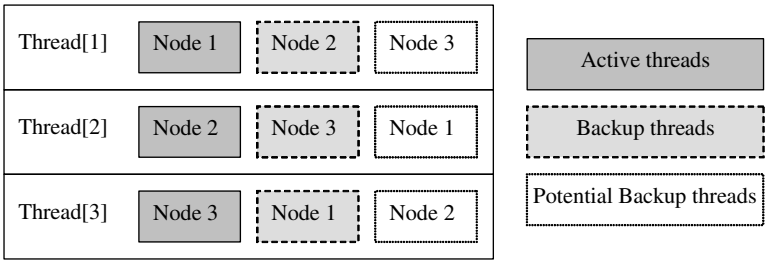

Fig. 6. Mapping of a thread collection with backup threads

5 Checkpointing Support

In order to provide support for checkpointing, long-running operations that may be suspended (split and merge operations) need some minor modifications, so as to allow them to be restarted from other points than from the beginning. Since operations are simple C++ functions, and since the language does not provide a simple means to checkpoint the current state of a function and to restart it later, the application needs to help the framework in order to obtain the desired functionality. The required modifications are independent of the general application structure, since the changes affect only individual operations. For the following discussion, we use the split and merge operations of the compute farm application (Figure 2) as example.

The first functionality that the application needs to provide is the ability to checkpoint the current state of the operation. For the split operation previously shown in section 2, the loop counter *splitIndex* needs to be made serializable, allowing the operation to be recreated from a checkpoint. Since DPS provides an automatic serialization mechanism for data objects, we reuse this mechanism for operations. Therefore, the split operation uses the automatic serialization syntax for its data members as follows:

```
class Split : public dps::SplitOperation
  <SplitInDataObject, SplitOutDataObject, MasterThread>
{
  CLASSDEF(Split)
    BASECLASS(dps::OperationBase)
  MEMBERS
    ITEM(Int32,splitIndex) // Current loop counter
  CLASSEND;
```

The second functionality that the application needs to provide is the ability to restart the operation from a saved checkpoint. DPS uses the input data object parameter of the function to distinguish between a normal call to the operation and a restarted call. When the operation is initially called during normal execution, it receives a valid non-NULL input data object. However, when it is being

restarted from a checkpoint after a failure, no input data object is passed as parameter (i.e. the input data object pointer is NULL). This particular case is used to skip the initialization of the internal variables, since they have already been set up by the checkpoint:

```
public:
  // Split operation
  void execute(SplitInDataObject *in)
  {
    // If the input data object is NULL, the operation is
    // being restarted from a checkpoint. Otherwise, we need to
    // initialize our variables.
    if(in)
      splitIndex=0;
```

The content of the loop itself is kept unchanged. The update of the loop counter has been moved to a position before the call to *postDataObject*, since it is at this point that a checkpoint is taken when a checkpoint is requested. The loop condition can be checked with a *while* statement.

```
    // Loop until all output data objects have been generated
    while(splitIndex<NB_PARTS)
    {
      SplitOutDataObject *sot=new SplitOutDataObject(splitIndex);
      splitIndex++;

      postDataObject(sot);
    }
  }
```

Finally, the application needs to call the checkpointing function for the main thread collection. Since checkpointing is fully asynchronous within the DPS framework, this can be done anywhere. In the present example, we add the checkpointing request within the main loop of the Split operation. Three checkpoints are requested, one for every 25% of output data objects posted. We introduce an additional member variable *next* that indicates at which point the next checkpoint is due. This variable is checked within the loop, and checkpoints are requested accordingly. This variable also needs to be serializable like the loop counter.

```
    // Loop until all output data objects have been generated
    while(splitIndex<NB_PARTS)
    {
      // Do some periodic checkpointing in Split
      if(splitIndex>next)
      {
        next+=NB_PARTS/4;
```

```
        // This is an asynchronous call, the checkpoint will be
        // taken shortly after.
        getController()->getThreadCollection<MasterThread>
                                        ("master").checkpoint();
      }

      SplitOutDataObject *sot=new SplitOutDataObject(splitIndex);
      splitIndex++;

      postDataObject(sot);
    }
```

Calling the *checkpoint* function does not immediately create a checkpoint, but informs the framework that a checkpoint should be taken as soon as possible. Since all the threads within a thread collection are independent, they are checkpointed individually. The checkpointing process is started as soon as the currently executing operation on the current thread ends or is suspended (for example when waiting for its next input data object). When no operation is running on a thread, its state is guaranteed to be consistent. The checkpoint is then sent to the backup thread. The checkpoint is composed of the current local state of the active thread, the list of currently suspended operations as well as the list of all the data objects that have been processed since the last update. The new state replaces the previous state stored on the backup thread, and the listed data objects are removed from the backup thread's data object queue. When the checkpointing process is complete, execution resumes normally on the thread. In the above example, the checkpoint is taken on the call to *postDataObject* immediately following the call to *checkpoint*.

When checkpointing is used on this type of application, it is important to enable flow control, in order to ensure that the split operation does not post all subtasks at once. If flow control is disabled, all the checkpoints are taken at the same time after termination of the execution of the split function, making the complete process useless. With flow control enabled, the checkpoints are taken as expected, since the split operation is periodically suspended while waiting for the merge operation to catch up.

The Merge operation needs similar changes in order to ensure that the current output data object state is correctly preserved when checkpointing. The following source code describes the original merge operation before adding code for fault-tolerance:

```
class Merge : public dps::MergeOperation
  <MergeInDataObject, MergeOutDataObject >
{
  void execute(MergeInDataObject *in)
  {
    // Create output data object
    MergeOutDataObject *output=new MergeOutDataObject();
```

```
    // Wait until all the computation results have been received
    do
    {
      // Add the result contained in the input data object 'in'
      // to the output data object
    }
    while((in=waitForNextDataObject())!=NULL);

    postDataObject(output);
  }
}
```

The local state of the operation is entirely contained in the output data object, which is updated for each incoming data object in a *while* loop. Therefore, in order to enable restarting, the output data object needs to be stored within the merge operation class. In the DPS framework, the *dps::SingleRef* class is used to store a serializable pointer.

```
class Merge : public dps::MergeOperation
  <MergeInDataObject, MergeOutDataObject >
{
  CLASSDEF(Merge)
    BASECLASS(dps::OperationBase)
  MEMBERS
    // The output data object
    ITEM(dps::SingleRef<MergeOutDataObject>,output)
  CLASSEND;
```

Just like the Split operation, the Merge operation uses the state of the input data object for initialization of the output data object. The loop within the merge operation is unchanged compared with the non fault-tolerant code, since the local state is already updated before calling *waitForNextDataObject*. In the fault-tolerant case, the last operation of the flow graph is responsible for storing the result of the parallel computation, rather than posting a data object to the caller of the parallel schedule. This is necessary to ensure that the parallel application terminates even when the original master node that initiated the execution of the parallel schedule is dead. Since the merge operation is the last operation in the flow graph, the operation ends with a call to *endSession* in the DPS controller, which causes the application to terminate. Since the application terminates from within the merge operation, the output data object is never posted.

```
  void execute(MergeInDataObject *in)
  {
    // If the operation is not being restarted, initialize the
    // output data object
```

```
    if(in!=NULL)
      output=new MergeOutDataObject();

    // Wait until all the computation results have been received
    do
    {
      // Add the result from the input data object 'in' to the
      // output data object if in is not NULL
    }
    while((in=waitForNextDataObject())!=NULL);

    // Store computation result before terminating application

    getController()->endSession(true);
  }
```

5.1 Serializing Thread States

For applications that store a local thread state, it is necessary to ensure that the local thread state can be copied correctly within the checkpointing process. This is achieved by using the DPS serialization mechanism. Consider the following thread with local data:

```
struct ComputeThread
{
  int data; // Single integer stored in thread
};
```

The thread is simply converted to the serializable form as follows:

```
struct ComputeThread
{
  CLASSDEF(ComputeThread)
  MEMBERS
    ITEM(int,data)  // Single integer stored in thread
  CLASSEND;
};
```

6 Conclusions and Future Work

DPS is a novel high-level environment for developing parallel applications specified as executable flow graphs. The DPS framework provides dynamic handling of resources, in particular the ability to specify the mapping of threads to nodes at runtime, and to modify this mapping during program execution. Flow graphs and updatable thread mappings are the foundation on which we build fault-tolerance.

We implement fault-tolerance by providing a hybrid recovery scheme using two compatible mechanisms for the recovery of flow graph program execution segments located on a failed node. The first general purpose mechanism relies on duplicate data objects sent to backup nodes in order to enable the reconstruction of the state of a thread upon node failure. Backup threads are kept up to date by periodical checkpointing of thread states. Upon occurrence of a failure, the current state of the threads that were on the failed node is reconstructed on the backup threads by re-executing operations. The valid execution sequence of operations is automatically deduced from the flow graph of the corresponding DPS application by applying a simple sender-based data object numbering scheme. A second specialized sender-based mechanism is used for operations that do not depend on local state information, such as graph segments comprising simple compute farms. Since no state needs to be reconstructed in case of failures, the duplicate communications are avoided. The flow graph provides information about the runtime execution patterns of applications, allowing the framework to transparently select the appropriate recovery mechanism for the graph segments. For compute bound applications, the fault-tolerance overheads during normal program execution remain low thanks to the DPS asynchronous communications that occur in parallel with computations.

The general-purpose fault-tolerance mechanism allows computation to continue as long as for each thread within every thread collection either the active thread or its backup thread remains valid. The optional compatible stateless recovery mechanism requires that at least one thread remains valid within every stateless thread collection, and that the threads hosting the surrounding split-merge pair are recoverable with the general purpose recovery mechanism.

The fault-tolerance mechanisms are not fully transparent to the application developer. However, only minor changes need to be made to the application in order to enable fault-tolerance. The required changes are due to limitations of the C++ language. Some aspects, such as checkpointing requests, are currently left to the programmer. These requests could also be performed automatically by the framework by monitoring the applications flow graph. The resulting fault tolerance scheme may then become more transparent to the application developer.

The complete DPS software package is available on the Web under the GPL license at http://dps.epfl.ch. The complete source code for the applications presented in this paper can also be found at this address.

Acknowledgements

This project has been partially funded by the Hasler Foundation, project DICS-1845.

References

1. A. Agbaria, R. Friedman, Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations, 8th International Symposium on High Performance Distributed Computing (HPDC-8'99), IEEE CS Press, August 1999

2. A. Baratloo, P. Dasgupta, Z.M. Kedem, Calypso: A Novel Software System for Fault-Tolerant Parallel Procssing on Distributed Platforms, Proc. International Symposium on High-Performance Distributed Computing, pp. 122-129, 1995
3. R. Batchu, J. Neelamegam, Z. Cui, M. Beddhua, A. Skjel-lum, Y. Dandass, M. Apte, MPI/FT: Architecture and taxonomies for fault-tolerant, message-passing middleware for performance-portable parallel computing, 1st IEEE International Symposium of Cluster Computing and the Grid, Melbourne, Australia, 2001
4. B. Bhargava, S.R. Lian, Independent Checkpointing and Concurrent Rollback for Recovery - an Optimistic Approach, Proc. IEEE Symposium on Reliable Distributed Systems, pp. 3-12, 1988
5. S. Chakravorty, L.V. Kale, A fault tolerant protocol for massively parallel systems, 18th International Parallel and Dis-tributed Processing Symposium (IPDPS'04), pp. 212-219, April 2004
6. D. Das, P. Dasgupta, P.P. Das, A New Method for Transparent Fault Tolerance of Distributed Programs on a Network of Workstations Using Alternative Schedules, Proc. Conf. on Algorithms and Architectures for Parallel Processing (ICAPP'97), pp. 479-486, 1997
7. J. Dongarra, S. Otto, M. Snir, D. Walker, A message passing standard for MPP and Workstations, Communications of the ACM Vol. 39, No. 7, pp. 84-90, 1996
8. E.N. Elnozahy, L. Alvisi, Y.M. Wang, D.B. Johnson, A Survey of Rollback-Recovery Protocols in Message-Passing Systems, ACM Computing Surveys, Vol. 34, No. 3, pp. 375-408, September 2002
9. E.N. Elnozahy, W. Zwaenepoel, Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback and Fast Output Commit, IEEE Transactions on Computers, Vol. 41 No. 5, pp. 526-531, May 1992
10. S. Gerlach, R.D. Hersch, DPS - Dynamic Parallel Schedules, International Parallel and Distributed Processing Symposium (IPDPS'03), pp. 15-24, April 2003
11. S. Gerlach, R.D. Hersch, Fault-tolerant Parallel Applications with Dynamic Parallel Schedules, International Parallel and Distributed Processing Symposium (IPDPS'05), p. 278b, April 2005
12. S. Gerlach, DPS online documentation, http://dps.epfl.ch
13. D.B. Johnson, W. Zwaenepoel, Sender based message logging, Digest of Papers, FTCS-17, Proc. 17th Annual International Symposium on Fault-Tolerant Computing, pp. 14-19, 1987
14. J.S. Plank, Y. Kim, J.J. Dongarra, Algorithm-Based Diskless Checkpointing for Fault Tolerant Matrix Operations, FTCS-25, Proc. 25th Annual International Symposium on Fault-Tolerant Computing, pp. 351-360, 1995
15. R. Strom, S. Yemini, Optimistic recovery in distributed systems, ACM Transactions on Computer Systems, Vol. 3, No. 3, pp. 204-226, 1985
16. Y. Tamir, C.H. Sequin, Error recovery in multicomputers using global checkpoints, Proceedings of the International Conference on Parallel Processing, pp. 32-41, 1984
17. Y.M. Wang, W.K. Fuchs, Lazy Checkpoint Coordination for Bounding Rollback Propagation, Proc. 12th Symposium on Reliable Distributed Systems, pp. 78-85, October 1993