19th International Parallel and Distributed Processing Symposium (IPDPS), 10th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems (DPDNS'05), April 4-8, Denver CO, USA, 2005, p. 278b

Fault-tolerant Parallel Applications with Dynamic Parallel Schedules

Sebastian Gerlach, Roger D. Hersch
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland
School of Computer and Communication Sciences

Abstract

Commodity computer clusters are often composed of hundreds of computing nodes. These generally off-the-shelf systems are not designed for high reliability. Node failures therefore drive the MTBF of such clusters to unacceptable levels. The software frameworks used for running parallel applications need to be fault-tolerant in order to ensure continued execution despite node failures. We propose an extension to the flow graph based Dynamic Parallel Schedules (DPS) development framework that allows non-trivial parallel applications to pursue their execution despite node failures. The proposed fault-tolerance mechanism relies on a set of backup threads located in the volatile storage of alternate nodes. These backup threads are kept up to date by duplication of the transmitted data objects and periodical checkpointing of thread states. In case 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 corresponding valid re-execution order is automatically deduced from the data flow graph of the DPS application. Multiple simultaneous failures can be tolerated, provided that for each thread either the active thread or its corresponding backup thread survives. For threads that do not store a local state, an optimized mechanism eliminates the need for duplicate data object transmissions. The overhead induced by the fault tolerance mechanism consists mainly of duplicate data object transmissions that can, for compute bound applications, be carried out in parallel with ongoing computations. The increase in execution time due to fault tolerance therefore remains relatively low. It depends on the communication to computation ratio and on the parallel program's efficiency.

Keywords: *Parallel computing, clusters of workstations, parallel schedules, graceful degradation, fault tolerance, checkpointing, message logging*

1. Introduction

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 [11].

Checkpoint based approaches store the current state of computation to stable storage. *Coordinated* checkpointing on all participating nodes [22] 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 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 [23].

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 [11]. 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 [14][7]. *Optimistic* logging begins processing messages without waiting for a successful write to stable storage [21]. 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 [12].

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 [10][1]. When parallel applications

[1] Most fault-tolerant parallel MPI systems rely on coordinated checkpointing [20][1][8]. Further fault-tolerant implementations of MPI rely on uncoordinated checkpointing either with pessimistic message logging [3][16][5] or with pessimistic sender based message logging [6].

are described using high-level approaches, additional information about the structure of the application is available. For example, task graphs [9], Calypso [2] and Chime [18] make use of such information for recovering and resuming computation after a failure.

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 [17].

In the present paper, we describe recovery mechanisms relying on the flow graph structure of applications created with the Dynamic Parallel Schedules (DPS) framework [13]. The parallel program's flow graph greatly simplifies message logging, checkpointing and the recovery process in case of failures. We limit our considerations to fail-stop failures [19].

2. The Dynamic Parallel Schedules framework

DPS applications are defined as directed acyclic graphs of operations. The fundamental operation types are *leaf*, *split*, *merge* and *stream* [13]. 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.

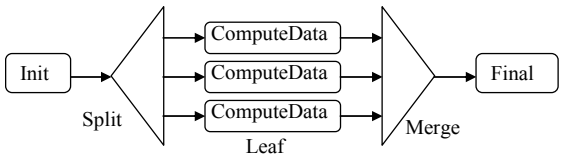

Figure 1. Flow graph describing data distribution (split), parallel processing, and collection of results (merge)

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. ComputeData in Figure 1). 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, i.e. the developer can provide 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.

During execution, operation instances are identified with a hierarchical *operation identifier* that is transferred along with the data objects. These identifiers allow DPS to identify subgraphs within the flow graph. A new identifier is added to the hierarchy for every traversed split operation, and removed again in the corresponding merge. Figure 2 shows the identifiers generated for two nested split-merge pairs.

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 neighborhood-dependent computation. Two thread collections are created. The first, *Main*, handles the global split and merge operations, and contains only a single thread. The second, *Compute*, handles the parallel computation, and contains one thread for each compute node.

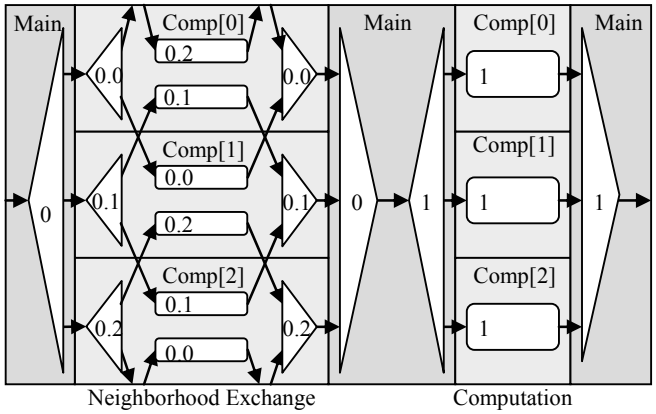

Figure 2. A flow graph and its associated thread collections (the communications between Comp[0] and Comp[2] wrap around). The numbers inside the operations are the operation identifiers.

A thread in DPS 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. For example, in a matrix computation, matrix parts may be stored within each thread of a thread collection. 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.

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 2 can easily be specified by using relative thread indices.

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.

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

The DPS flow graph with its associated thread collections is a powerful tool for implementing fault-tolerance. Whereas implementations of fault-tolerance for low level message passing libraries have little or no information about the application, high-level frameworks provide information about the applications execution patterns.

Every operation that is executed within the DPS framework has two data input sources: the incoming data object(s), and the local thread state data. Likewise, it has two outputs: the outgoing data object(s), and any modifications made to the local thread state data. Operations are assumed to be deterministic, i.e. for a given initial thread state and set of incoming data objects, they will always produce the same output.

The execution order of operations in a parallel schedule is defined by the flow graph, and constrained by the split/merge operation pairs. Within a split-merge operation pair, the outgoing branches of the split operation may be executed in any order. Let us consider the flow graph shown in Figure 3a. By assuming that the first split operation generates two data objects, and that the second split operation splits each of them into three data objects, we obtain the operation executions shown in Figure 3b. The execution order of the operations is only constrained by the flow graph's directed edges. For instance, two valid execution orders for the graph are {1,2a,3a1,3a2,3a3,4a,2b,3b1,3b2,3b3,4b,5} and {1,2b,3b3,2a, 3b2,3a1,3b1,3a3,4b,3a2,4a,5}. Because of the asynchronous execution model of DPS and of external factors such as network load or operating system scheduling, the effective execution order cannot be predicted.

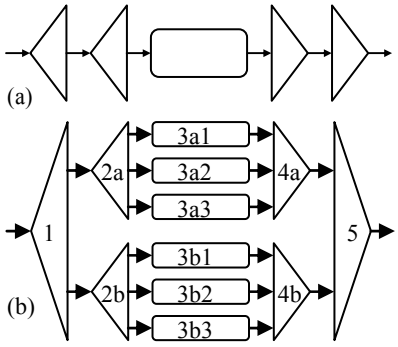

Figure 3. (a) Simple flow graph and (b) operations that are executed when running the parallel schedule (each arrow represents one data object).

However, in case of node failure, we may, thanks to a data object numbering mechanism reflecting the flow graph's execution constraints, reconstruct a failed node state on a backup node by ensuring a valid execution order (section 4). The reconstruction of a thread state lost due to a node failure has similar properties as pessimistic logging [11]: incoming data objects are consumed during re-execution according to a valid execution order and duplicata of outgoing data objects generated during re-execution have no effect on other nodes (silent re-execution property [5]).

DPS detects node failures by monitoring communications. When a node is not able to communicate with another node, that node is considered to be failed[1]. The TCP/IP network layer used by DPS reports these failures as disconnections. On a failure, DPS starts a master election process among the participating nodes. The master then polls all nodes to discover which node has become unavailable. It updates the thread collections used by the application by removing all threads that were located on the failed node and by converting their backup threads into active threads (section 4). The updated thread collections are distributed onto all nodes. During the thread collection update process, execution of operations is not suspended.

We have implemented two recovery mechanisms: the first for the general case and the second optimized for handling graph segments whose operations are executed within a stateless DPS thread, i.e. a DPS thread without local state. Both types of graph segments often coexist, since applications may use several distinct thread collections for executing their various subtasks, for example by separating pure computation tasks from tasks that require access to a distributed data structure stored in local thread states. By examining the threads associated to the flow graph, DPS can automatically apply the appropriate recovery mechanism depending on whether a thread is stateless or not[2].

4. General recovery scheme

In the general case, we have to be able to reconstruct the local thread states of all threads that are located on the failed node. The local thread state can only be modified by an operation associated with this thread. The inputs of an operation can only be its input data object and the local thread state. A new thread state T_2 is obtained by applying the operation UserOp to the input data object D_1 and the current state T_1.

$$T_2 = \text{UserOp}\ (\ T_1, D_1\ )$$

Our recovery mechanism relies on a set of backup threads. At least one backup thread is created for every active thread in a thread collection. These backup threads are mapped onto nodes that differ from the nodes running the active threads, for example by rotating the thread indices (Figure 4). When a data object is sent to an operation on a given thread, it is also sent to the backup thread.

This type of mapping provides partial support for simultaneous multiple node failures, since as long as for each thread either the active thread or the backup thread survives, the application can pursue its execution. In order to support successive failures, the backup threads that were located on a failed node or that have

[1] DPS assumes that the underlying network will not produce partial disconnections, i.e. that a failure will never allow a node to communicate only with some of its peers.

[2] DPS considers a thread to have a state if it is defined as a custom class within the application's implementation.

been consumed when replacing a failed active thread are reconstructed on another node with the mechanism described in section 5.

When a data object is posted to an operation on a given thread, the active thread performs the operation, and the backup thread only stores a copy of the data object. When the node with the active thread fails, the backup thread has all the data objects needed for reconstructing the latest state.

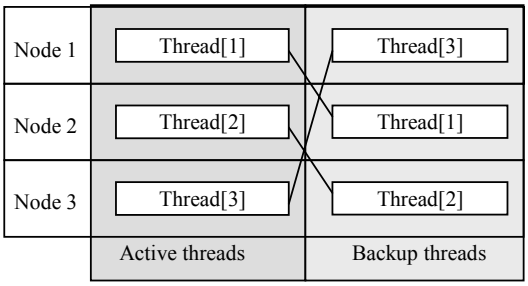

Figure 4. Mapping of active and backup threads onto processing nodes

Since DPS operates asynchronously and the data objects sent to the backup threads originate from multiple nodes, the data objects can potentially arrive into the backup thread queues in an order that does not necessarily correspond to a valid execution order. In order to ensure a successful recovery, it may be necessary to reorder the data objects into a valid execution order as described in section 3.

According to the flow graph, any data object leaving an operation depends on all data objects having previously entered this operation. In order to reflect this dependency, each operation assigns a sequence number to all the data objects it generates. The sequence number of an outgoing data object is larger than the sequence number of all previously received data objects. We compute the sequence number as follows:

$$SeqOut_{n+1} = \mathbf{max}(SeqIn, SeqOut_n) + 1$$

SeqIn is the collection of sequence numbers of all input data objects that have been received by the operation and $SeqOut_n$ is the sequence number of the previously sent object (if any). This mechanism ensures that the sequence numbers steadily increase. However it does not produce globally unique sequence numbers, since two separate branches of the graph may produce the same numbers. Nevertheless, it does accurately reflect the execution order constraints induced by the DPS flow graph. Figure 5 shows the sequence numbers generated for a neighborhood exchange followed by a computation.

When recovering from a failure and initiating a re-execution on the backup thread's node, the data objects present in the backup thread's queue are sorted according to their sequence numbers. The backup thread is subsequently marked as active, causing execution of operations to resume with a valid execution order. Since the stored data objects contain the operation identifiers, split and merge operations occur with the same combinations of data objects as the original failed execution.

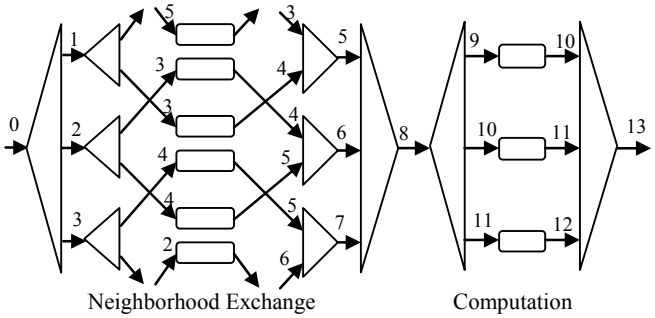

Figure 5. Sequence numbers associated with the transmitted data objects for a neighborhood-dependent computation.

Since operations are re-executed during the reconstruction process, they will also repost their output data objects. If a data object is sent to an operation on a thread where that operation has already been successfully executed on a previous instance of the same data object (same sequence number and same operation identifier), that data object is discarded (silent re-execution property).

The proposed recovery scheme induces a communication overhead in running applications, since every data object is sent twice, once to the active thread and once to the backup thread. On applications that are not communication bound, part of this overhead can be hidden, since network transfers are partially overlapped with computations. There is also a memory overhead induced by the backup threads, which contain both a previous thread state and the pending data object queue.

5. Checkpointing

In order to shorten the reconstruction time of a failed node, one may copy the state of the active threads onto the corresponding backup threads. Such copies are also needed in order to regenerate new backup threads when the previous backup threads have been consumed. Each DPS thread has three components that must be conserved for successful reconstruction: the local thread state, the queue of data objects that wait for processing, and the state of active (but possibly suspended) operations within that thread.

The data structures used for checkpointing and message logging are very simple, since no additional elements such as dependency graphs need to be created. All the necessary information such as the sequence numbers and the operation identifiers are already present within the data objects. When starting the checkpointing process, DPS first waits for the currently executing operation on the active thread to end or to be suspended (for example when waiting for its next input data object). The checkpoint is then sent to the backup thread. It is composed of the current 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.

Since copying the current state also removes part of the pending data object queue on the backup thread, it limits 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. The effective overhead induced by stopping a thread in order to checkpoint can therefore be kept very low.

Since the reconstruction time after a failure is equal to the time elapsed since the last checkpoint, periodic checkpointing is essential for quick reconstruction.

6. Recovery without local thread state

A DPS graph segment whose operations do not use local thread state data is similar to a simple task farm, where computation tasks are fully represented by their input data objects. The computation nodes are perfectly interchangeable, and no state needs to be restored. For graph segments located within stateless threads, we eliminate the need for backup threads by logging the data objects on the sender node's volatile storage.

When a node fails, all tasks that were executing or queued on the failed node need to be resent to other nodes. Within a task farm model, resending tasks to other nodes is delegated to the master node of the task farm. Since DPS does not specify a master node, resending of tasks is delegated to the nodes executing the split-merge operation pair. This makes sense since the split operation is responsible for the creation and distribution of tasks and the merge operation can monitor which tasks have already been successfully completed.

In order to implement the task redistribution mechanism, DPS keeps for every split operation a copy of every sent data object (representing a task) in the split operation's local thread state. The data objects sent by the split operation are then processed by a pipeline of one or more stateless leaf operations, and the results are returned to the merge operation. When the merge operation receives the data object, it notifies the split operation that the processing operations associated with this data object have been carried out successfully. DPS subsequently removes the copy of the sent data object from the local thread state.

When a failure is detected, the thread collection is reconstructed by removing the failed nodes, i.e. by reducing the number of available threads within the thread collection. The split operation resends all its stored data objects. Since the thread collection has changed, these data objects are routed to valid computation nodes. All data objects are sent, since their route beyond the first operation is not known to the split operation. An example of such a case is shown in Figure 6.

After a failure, additional threads can be added to the thread collection so as to provide support for subsequent failures. In order to ensure the survival of a parallel schedule, at least the outermost split-merge pair must be located on threads with backup threads. Split-merge pairs having backup threads can, in case of failure, be recovered using the general recovery scheme and therefore always launch the re-execution of the enclosed operations.

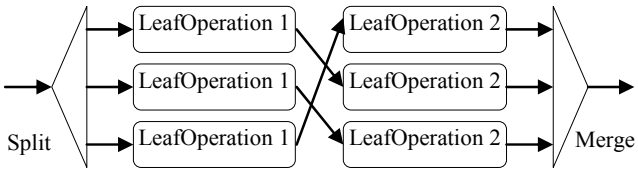

Figure 6. Example of complex routing within a split-merge operation pair

Recovery without local state induces a memory overhead. During normal execution, the nodes that execute split operations must store the data objects until the corresponding notification from the merge operation is received. This overhead can be limited by using the DPS flow control mechanism, which limits the number of data objects that are simultaneously in circulation [13]. Recovery without local state data removes the need for transmitting duplicate data objects. It also supports a larger number of simultaneous failures within a thread collection, since failed executions can be re-executed as long as at least one thread in the thread collection remains valid.

7. Performance evaluation

In order to evaluate the performance overheads introduced by the fault tolerance mechanisms, we have developed a simple analytical model for predicting them. We tested the model by predicting and measuring the overheads for a parallel implementation of Conway's Game of Life. The game of life program has a non-trivial parallel structure which is similar to many iterative finite difference computational problems [15].

The world data structure is evenly distributed between the nodes, each node holding a horizontal band of the world. Each computation requires knowledge of the state of lines of cells held on neighboring nodes. A simple approach consists in first exchanging borders, and after a global synchronization, computing the future state of the world. The corresponding DPS flow graph is illustrated in Figure 7.

The computation of the future state of the center of the part of the world stored on a node can be carried out without knowledge of any cell lines located on the neighboring nodes. We can perform this computation in parallel with the border exchange. A new flow graph (Figure 8) can thus be constructed, by keeping most of the operations as they were in the previous graph. This improved flow graph enables DPS to hide most of the communication overhead incurred by the border exchange.

The performance overhead prediction model only takes into account the additional network communications induced by fault

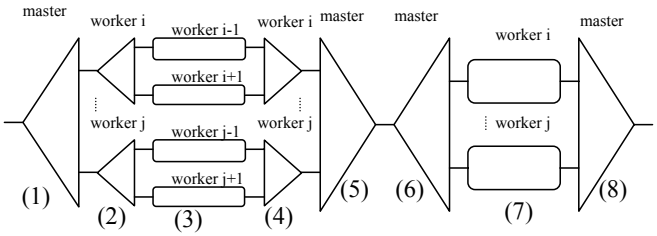

Figure 7. Simple flow graph for the parallel game of life (unfolded view): (1) split to worker nodes; (2) split border transfer request to neighboring nodes; (3) neighbors send the borders; (4) collect borders; (5) global synchronization to ensure that all borders have been exchanged; (6) split computation requests; (7) compute next state of world; (8) synchronize end of current iteration.

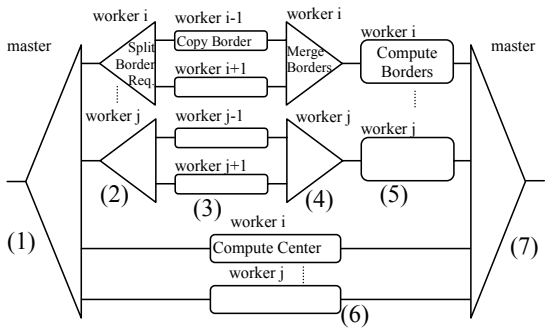

Figure 8. Improved flow graph for the parallel game of life (unfolded view): (1) split to worker nodes; (2) split border transfer request to neighboring nodes; (3) neighbors send the borders; (4) collect borders; (5) compute next state of borders; (6) compute next state of center; (7) synchronize end of current iteration.

tolerance, since the overhead consists mainly of duplicate messages sent to backup threads and of thread state exchanges between active and backup threads. The following factors need to be considered: the application's communication to computation ratio[1], the network bandwidth, the relative CPU load for network transfers, and finally the size of the additional network transfers due to fault tolerance. For compute bound applications, the processing time required for the redundant network transfers (duplicate data objects sent to backup nodes) is added to the program execution time observed when fault tolerance is disabled. In communication bound applications, the full network transfer time of duplicate data objects is added. Duplicate data object transfers that take place at the end of the flow graph, such as values returned to the final merge function, can be ignored, since program execution will terminate before the duplicate data objects have been sent to their backup threads.

We measure two overheads: the overhead induced by duplicate posting of data objects when fault tolerance is activated, and the overheads induced by checkpointing. These overheads are computed by comparing the execution times of the application with fault tolerance enabled and disabled.

Figure 9 illustrates timelines for the execution of one iteration of the game of life on a single node with and without fault tolerance. Separate timelines indicate incoming data object communications, computations, and outgoing data object communications. The additional communications induced by the fault tolerance scheme are carried out in parallel with the computation of the center cells and do therefore not significantly affect the parallel program's overall execution time.

When using large world sizes, the game of life application is compute-bound and has a low communication to computation ratio. In order to demonstrate the influence of the communication to computation ratio, we create a variant of the application by artificially increasing the message size of the border exchange operation. Each node sends an additional 1 MB of data to each of its neighbors, inducing a higher parallel communication load.

The performance measurements were taken on a network of Sun Ultra 10 workstations (360MHz, 256MB RAM) with a Fast Ethernet interconnect (100 Mb/s). Performing full-duplex network transfers at maximum speed (9 MB/s in both directions) consumes 50% of the CPU of the Sun workstations.

Figure 10 illustrates the overhead in parallel application execution time with the additional border communications. Without additional border communications, the communication to computation ratio is always lower than 0.3% and the corresponding overheads are too low to be measurable. With the additional border communications, Figure 10 shows that the overhead induced by fault tolerance is roughly proportional to the application's communication to computation ratio. The overheads closely match the predictions computed by using the previously described model.

Figure 11 shows that fault tolerance has only a small impact on speedup. The overheads induced by fault tolerance are similar

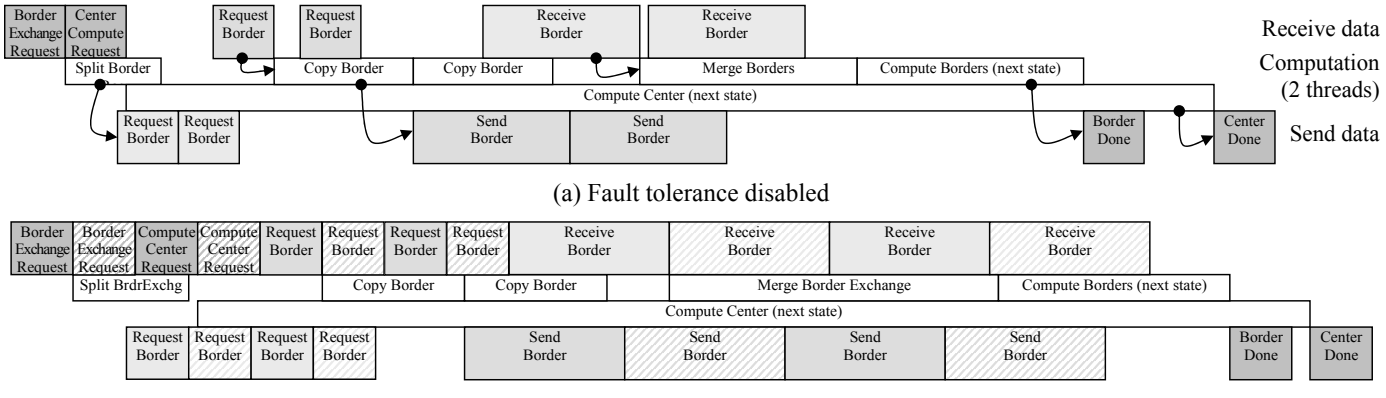

Figure 9. Timelines for a worker node running the improved graph of the game of life, with (a) fault tolerance disabled and (b) fault tolerance enabled (illustrative example, does not reflect real time intervals).

[1] The communication to computation ratio is defined as the ratio between pure communication time, i.e. the sum of all data object transfer times, and pure computation time, i.e. the time to execute the application on a single node computer without any data transfers.

for both the normal and improved graphs since the additional network load is identical.

A further overhead induced by fault tolerance is the checkpointing time. The overheads induced by checkpointing can be predicted using the same model as the message duplication overheads, since the checkpoints are transferred to the backup threads during program execution. The only additional overhead is caused by the necessity to lock the individual threads in order to take valid checkpoint images of their states. The overheads for checkpointing were also measured on the game of life sample application, using a 4000x4000 world size and the improved graph. The size of a checkpoint is 32 MB.

The checkpointing overhead (Figure 12) shrinks with the number of iterations per checkpoint. The relative overhead is nearly independent of the number of nodes, since state replication is carried out in parallel, and both the size of the replicated state and the program execution time are inversely proportional to the number of nodes.

8. 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 reexecuting 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

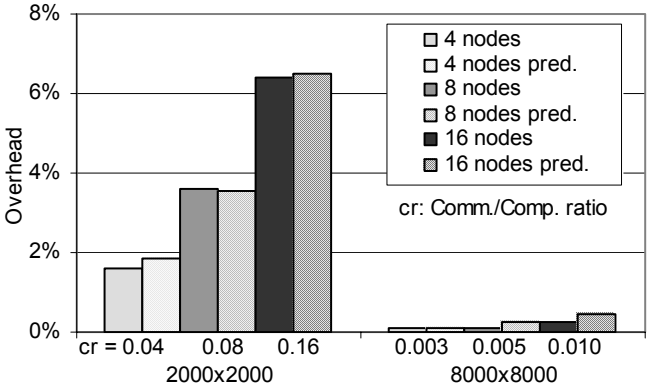

Figure 10. Measured and predicted performance overhead of the Game of Life when fault tolerance is enabled (improved graph, 1MB additional border communications)

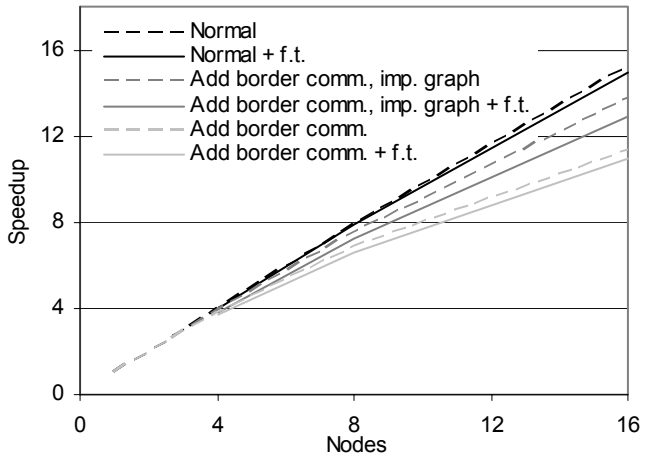

Figure 11. Speedup of the Game of Life, world size 2000x2000, showing normal execution and execution with additional border communications, with and without fault tolerance enabled (f.t.)

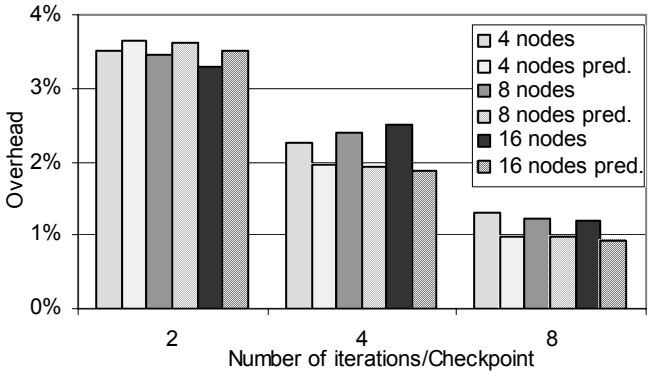

Figure 12. Measured and predicted performance overhead of the Game of Life for checkpointing, world size 4000x4000.

split-merge pair are recoverable with the general purpose recovery mechanism.

In the future, we intend to explore techniques for storing data objects at their source rather than on backup threads. This will lead to more complex reconstruction mechanisms, but may avoid network overheads, which is especially important for data intensive applications.

We also intend to automate the checkpointing operations by allowing the DPS framework to decide when checkpointing should be carried out by monitoring the applications flow graph. The resulting fault tolerance scheme may then become fully transparent for the application developer.

The DPS software is available on the Web under the GPL license at http://dps.epfl.ch. The version supporting graceful degradation will soon become available.

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. Skjellum, 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] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fédak, C. Germain, T. Hérault, P. Lemarinier, O. Lodygensky, F. Magniette, V. N´eri, A. Selikhov, *MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes*, High Performance Networking and Computing (SC2002), Baltimore, USA, November 2002

[6] A. Bouteiller, F. Cappello, T. Hérault, G. Krawezick, P. Lemarinier, F. Magniette, *MPICH-V2, a fault-tolerant MPI for volatile nodes based on pessimistic sender based message logging*, High Performance Networking and Computing (SC2003), Phoenix, USA, November 2003

[7] S. Chakravorty, L.V. Kale, *A fault tolerant protocol for massively parallel systems*, 18th International Parallel and Distributed Processing Symposium (IPDPS'04), pp. 212-219, April 2004

[8] Y. Chen, K. Li, J.S. Planck, *Clip: A checkpointing tool for message-passing parallel programs*, High Performance Networking and Computing (SC97), IEEE/ACM, November 1997

[9] 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

[10] 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

[11] 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

[12] 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

[13] S. Gerlach, R.D. Hersch, *DPS - Dynamic Parallel Schedules*, 17th International Parallel and Distributed Processing Symposium (IPDPS'03), Nice, France, pp. 15-24, April 2003

[14] 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

[15] V. Kumar, A. Grama, A. Gupta, G. Karypis, *Introduction to Parallel Computing*, Benjamin Cummings Publishing Company, Chapter 11, Solving sparse systems of linear equations, pp. 407-489, 1993

[16] S. Louca, N. Neophytou, A. Lachanas, P. Evripidou, *MPI-FT: Portable fault tolerance scheme for MPI*, Parallel Processing Letters, Vol.10, No.4, World Scientific Publishing Company, pp. 371-382, 2000

[17] 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

[18] S. Sardesai, *Chime: A Versatile Distributed Parallel Processing Environment*, PhD thesis, Arizona State University, 1997

[19] R.D. Schlichting, F.B. Schneider, *Fail-stop processors: An approach to designing fault-tolerant computing systems*, ACM Transactions on Computer Systems Vol. 1, No. 3, pp. 222–238, 1983

[20] G. Stellner, *CoCheck: Checkpointing and Process Migration for MPI,* Proc. 10th International Parallel Processing Symposium, April 1996

[21] R. Strom, S. Yemini, *Optimistic recovery in distributed systems*, ACM Transactions on Computer Systems, Vol. 3, No. 3, pp. 204–226, 1985

[22] Y. Tamir, C.H. Sequin, *Error recovery in multicomputers using global checkpoints*, Proceedings of the International Conference on Parallel Processing, pp. 32–41, 1984

[23] 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