Transient congestion avoidance in Software Distributed Shared Memory systems

J.J.Costa, T.Cortes, X.Martorell, J.Bueno, E.Ayguade
Universitat Politècnica de Catalunya - Barcelona Supercomputing Center
{jcosta,toni,xavim,jbueno,eduard}@ac.upc.es

***Abstract*—OpenMP applications executed on top of software distributed shared memory (SDSM) systems show peaks in network traffic. In these scenarios, synchronization points are used to maintain memory consistency and improve performance, and network traffic is highly increased due to data being exchanged between different nodes in the system. This behaviour generates network congestion which may limit or degrade applications performance. In this paper we present a technique to avoid these peaks by sending data producing the congestion earlier in time. Our proposal is to introduce *virtual* synchronization points between the real ones, and use them to automatically distribute the network traffic. This technique is evaluated with a synthetic benchmark, and with the classes A and B of two OpenMP codes from the NAS benchmarks (BT and CG), on top of NanosDSM, a page-based DSM implementing sequential consistency. The results show a 16% performance improvement on average over the traditional methods.**

Keywords: OpenMP; Software distributed shared memory; Pre-send; Pre-invalidation; Data forwarding; Network traffic and Congestion;

I. Introduction

OpenMP [1] and MPI[2] are the programming models best suited for parallel architectures. Even though OpenMP targets shared memory platforms and MPI the distributed ones, the ease of use of the OpenMP paradigm makes it desirable for the distributed platforms also. The main goal of a DSM system is to maintain a global shared memory consistent across the nodes in a cluster. This goal allows the execution of OpenMP parallel applications. All memory updates produced by a node in a region of code must be communicated through network messages. Usually, this communication is done at the OpenMP synchronization points (implicit and explicit barriers, locks, ...).

At these points, the network traffic increases with the number of memory changes. This traffic may require more bandwidth than the available in the network and it can reach the saturation point.

In order to help with this problem, and transfer all the data, the TCP protocol uses a 'congestion avoidance' algorithm [3], that limits the quantity of data sent, reducing bandwidth and adding some timeouts. This behaviour limits or degrades the final application performance.

The main idea in this paper is to avoid these transient congestion situations by sending the data earlier, when network is potentially idle. Our proposal is to distribute these network messages during the computation time before the congestion points.

To accomplish this goal we propose to divide the region of code between two congestion points in smaller sections, and let the OpenMP runtime detect and transfer the data that can be sent safely at the end of each section.

The *virtual synchronization points* (**VSPs**) are the proposed mechanism to divide a region. They are hints to the OpenMP runtime who will decide their semantics. These VSPs can behave like especial synchronization directives, where data that is not referenced anymore inside current region is synchronized; or they can be silently ignored.

All data transfered at a real synchronization point can be distributed through these VSPs. This distribution is done by the OpenMP runtime, who knows which data is accessed between synchronization points, and thus it can detect which data can be transfered in a safe manner at each VSP. A tight cooperation between the OpenMP and the DSM runtime is required for this detection.

II. Chopper mechanism

To maintain the memory consistency in an OpenMP application running on top of a DSM, a lot of network messages are usually generated at synchronization points. The **chopper** is the proposed mechanism to distribute these messages during the execution of the application.

This proposal has two parts: (1) an application code annotated by the programmer with *regions* and *virtual synchronization points*; and (2) an algorithm capable of distributing network traffic inside a region through its inner points at runtime.

Regions of code which suffers from network congestion needs to be identified to be able to distribute it. But, this is an easy task for parallel applications with synchronization events, because the congestion appears at these points. We can map a region to the application code between two synchronization points. At the end of this region, data with the modified memory inside the region must be sent through the network, producing a peak.

In order to avoid this single congestion point and to distribute the data, each region is divided into smaller sections with the *virtual synchronization points* added by the programmer. We call **chops** to these divisions. These points are used by the runtime to distribute all modified data inside the region, sending at each point just those data that will not be used any more.

```
!$omp parallel
!$omp start_region
!$omp do private(i)
      do i = 1, 5
        (your code here)
        !$omp chop
      enddo
!$omp stop_region
!$omp end parallel
```

Figure 1. Sample code for the creation of a parallel region in a parallel workshare where a chop is defined per each iteration.

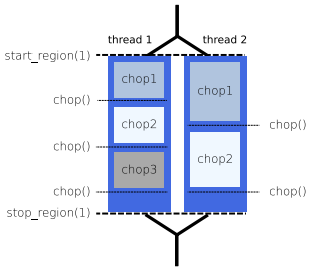


Figure 2. Graphical representation for sample code when executed with two threads.

This section explains the related concepts and the main algorithm for the chopper mechanism.

A. Region definition

A **region** is a delimited portion of code. It is defined by its limits (start and end points) and a unique identification number that is generated automatically by the compiler. Different regions should not overlap. Two operations for delimiting a region are proposed:

start_region (region_id) To mark the beginning of a new region identified by `region_id`.

stop_region (region_id) To mark the end of the region identified by `region_id`.

Figure 1 shows an example where a simple OpenMP parallel loop worksharing is enclosed between `start_region` and `stop_region` directives to define a parallel region.

B. Chop definition

A programmer can add virtual synchronization points (VSPs) inside a *region*, using the `chop()` operation. This operation divides the region into **chops**. There is no restriction in where to place the VSPs, nor their total number, but the programmer has to be aware that too many chops may produce too much overhead, and too few may have less chances to distribute the data. At runtime, a region is expressed as a sequence of one or more chops per thread, where different threads could execute different number of chops inside the same region.

The Fortran code at Figure 1, showing an example of chopper usage, creates a chop at each iteration. A graphical representation of its execution is shown in Figure 2, where two threads execute this loop with five iterations. Three chops are executed by the first thread, and two by the second, without any barrier between them.

C. Chopper algorithm

The algorithm to distribute the network traffic in a region through the virtual synchronization points has three phases. Each chop has to:

- Detect all referenced pages (read or written).
- Identify the set of pages that will not be referenced anymore until the next region: the completed pages.
- Distribute the completed pages to the next predicted region.

To control which phase must be executed, each region can be in three different states: (i) non-initialized, where the chopper is disabled; (ii) unstable, where the page accesses per chop are learnt; and (iii) stable, where the VSPs are used to distribute the completed pages.

The first time that a region is executed, the *non-initialized* state is used to avoid some page faults that are not relevant.

Afterwards, in the *unstable* state, all memory accesses inside each chop are monitored and recorded. When a region finishes, a list of completed pages is built at each inner chop. This list contains the addresses of all pages that are not referenced anymore in the remaining chops of the region.

Finally, when the sequence of regions and the completed pages have been learnt, the *stable* state is used. In this state, after executing a chop (at the VSP) the list of completed pages is used to distribute the pages to the next predicted region. The list could be empty, and then nothing is done at the VSP.

These three states are executed sequentially for all regions in the application code.

1) Detect page accesses: To detect that a page is referenced by the last time in a chop, the chopper needs to know all accessed pages at each chop inside a region. Therefore all memory references inside each chop should be monitored and recorded in a list. A tight cooperation between the OpenMP runtime and the DSM is required to detect these memory references.

This phase can have a significant overhead because it is necessary to mark pages as invalid at the beginning of each chop to avoid undetected memory references. This way any page access will generate a page fault and so it can be recorded. If this marking is not done, then the last access to the page could be detected wrong. For example, if a page has read protections before entering the chop (meaning that the page has been accessed for a read operation in the previous chop), any read operation inside the current chop will pass unnoticed, and so it can be identified as a completed page in the previous chop when it is not.

In our case, a function is registered in NanosDSM to be called whenever a page fault occurs, storing the faulted page address in a list inside the current chop.

2) Identify completed pages: When the last chop in a region finishes, a list of page addresses that can be safely distributed is built at each chop. A page can be safely distributed at a chop, if it is the last chop inside the region referencing that page. The list of completed pages at each chop is built traversing its faulted pages and adding those pages that can be safely distributed.

3) Distribute completed pages: Finally, after executing a chop in the stable state, the list of completed pages at this chop can be distributed, as they would at a synchronization point. This distribution is delegated to the DSM layer, who will be responsible to do the right thing.

In our case, when a node finishes a chop in a region, the list with the addresses of the completed pages is sent to their

```
function update_predictor( current )
    if (current is new)
        add current to predictor
    if (previous.next = current) //HIT
        previous.counter ++
    else //MISS
        if (previous.counter > 0)
            previous.counter --
        else
            previous.next := current
```

Figure 3. Pseudo-code for the region predictor update function

home nodes. All messages going to the same node are grouped into a single message to reduce latency.

After receiving this message, the DSM layer will predict the next region for each page address and it will send the pages with the right protections to the nodes that will need them at the next region. The implementation of our DSM layer is explained later at section III-A.

III. Our Environment

A. NanosDSM

NanosDSM[4] is an everything-shared SDSM for clusters, that offers a global shared address space. It is a page-based DSM implementing sequential consistency through a single-writer multiple-readers coherence protocol. Each page has a home node responsible to manage its coherence protocol.

In order to improve performance, NanosDSM offers the following features: (i) *pre-send*, to send data pages to other nodes that will request them for reading; (ii) *pre-invalidate*, to invalidate local copies of pages that will be requested by another node for writing; and (iii) *cooperation with the runtime*, allowing to register functions that will be executed whenever a page fault happens using an upcall mechanism.

NanosDSM code has been modified to solve chopper requests. When a list of page addresses is received from the chopper layer, all pages should be pre-sent/pre-invalidated according to the protections that will be required at the next predicted region. Therefore, the following features have been added to the DSM layer: (i) a next region predictor, to predict which is the next region after a given one; and (ii) a page state per region calculation, to know which nodes have a page and with which protections.

1) Next Region Predictor: In order to predict the region that goes after a given one, the sequence of regions must be detected. A simple 1-level predictor is used, it stores all regions and for each region it has the next region that follows it and a counter to ensure its validity.

Each time a new region starts, the predictor is notified. The region is inserted into the predictor if it was not present; and the predictor is updated if the saved prediction for the previous region matches (or not) the new one. The pseudocode is shown at Figure 3.

2) Page state per region: NanosDSM page fault requests have been augmented with the region identifier where the page fault has occured, this way the home node can record the state of all faulted pages inside each region. This state includes the nodes that have the page and its protections. This information is needed to run the pre-send/pre-invalidate mechanisms.

B. Nanos OpenMP Runtime

In our environment, OpenMP applications are parallelized using the Nanos Mercurium Compiler [5]. This compiler understands OpenMP directives embedded in traditional Fortran codes and generates parallel code. In the parallel code, the directives have triggered a series of transformations: parallel regions and parallel loop bodies have been encapsulated in functions for an easy creation of the parallelism. Extra code has been generated to spawn parallelism and for each thread to decide the amount of work to do from a parallel loop. Additional calls have been added to implement barriers, critical sections, etc. And variables have been privatized as indicated in the directives.

Nthlib [6] is our runtime library supporting this kind of parallel code. NthLib has been modified to implement the chopper mechanism.

IV. Methodology

In order to check the feasibility of our mechanism, three OpenMP benchmarks are used: a synthetic benchmark and two OpenMP codes from the NAS parallel benchmarks [7](BT, CG). All benchmarks have been executed on top of NanosDSM extended with the *chopper* technique.

The benchmarks are executed with 2, 4 and 8 nodes (one thread per node) in our testbed. An execution of both benchmarks without any OpenMP directive is used as baseline.

A. Benchmarks

Three versions are being evaluated for all benchmarks:

- **Original** This is the original OpenMP code.
- **Presend** This is the original version where calls to start the pre-send/pre-invalidation mechanisms have been added at the end of each parallel loop.
- **Chopper** Presend version has been extended with the chopper mechanism. A region has been defined for each loop workshare, and some of these loops have been *chopped*, adding new virtual synchronization points.

B. Synthetic benchmark

The synthetic benchmark is a simple application that iterates over two parallel loops accessing an array big enough to saturate the network (about 4000 memory pages). The first parallel loop traverses an array in sequential order writing all positions. The second loop reads all the updated positions from the first loop in reverse order to generate a different data placement, and ensure that data is not local to the node. The code is shown in Figure 4. A function *calculate* is inserted in the code to simulate an specific computation time. In our case we have used a value so that the time it takes to process all elements in a page, is the same as the time to resolve a page fault. This way the speedup can be easily modeled, and enough computation time is ensured for the chopper. Its

```
      do it = 1, niter
!$omp  parallel do
!$omp& default(shared)
!$omp& private(i)
         do i = 1, N
            v(i) = v(i) + 1
            call calculate(i)
         enddo

!$omp  parallel do
!$omp& default(shared)
!$omp& private(i,d)
         do i = N, 1, -1
            d = d + v(i)
            call calculate(i)
         enddo
       enddo
```

Figure 4. Main loop for the synthetic benchmark.

```
      do it = 1, niter
!$omp  parallel do
!$omp& default(shared)
!$omp& private(i)
         do i = 1, N
            v(i) = v(i) + 1
            call calculate(i)
            !$omp  chop
         enddo

!$omp  parallel do
!$omp& default(shared)
!$omp& private(i,d)
         do i = N, 1, -1
            d = d + v(i)
            call calculate(i)
            !$omp  chop
         enddo
       enddo
```

Figure 5. Main loop for the synthetic benchmark annotated with the chopper.

chopper version, where a chop is executed at each iteration, is shown in Figure 5.

C. BT Benchmark

The NAS BT benchmark [7] is a solver kernel for a typical problem on computational fluid dynamics codes (CFD). It updates a 3-dimensional array of points successively in the `x-`, `y-`, and `z-`direction solving a system of equations per planar grid point. The algorithm iterates through five basic functions: i) compute the right hand side matrix (rhs), solve the equations in the ii) `x-`, iii) `y-`, iv) and `z-`direction, and finally v) accumulate the results.

These 5 functions contains 15 parallel loops in the version used (NPB 3.3). The `rhs` function has 11 parallel loops, and the remaining functions have one parallel loop each. All loops are parallelized using the outermost dimension (`z`) except five of them, where the second outermost dimension (`y`) is used due to their data dependencies.

The loops basically modify eight large shared structures: *us*, *vs*, *ws*, *qs*, *square*, *rho_i*, *u* and *rhs*. Six 3-dimensional and two 4-dimensional matrices. The interesting part of this benchmark is how these structures are read or written at each loop. Most of these variables are written once, at the *rhs1* and *add* loops, and read at the remaining loops. The unique exception is the structure *rhs*, which is written in almost all of them.

D. CG Benchmark

The NAS CG Benchmark is a kernel to estimate the largest eigenvalue of a symmetric sparse matrix with a random pattern of nonzeros using a conjugate gradient method. The number of rows (na) of the sparse matrix and the number of non-zero elements per row (nz) determines the problem size of the benchmark class. We use the classes A and B as distributed in the NAS benchmarks suite for our experiments.

E. Testbed

The platform used to execute these benchmarks is a cluster with 8 nodes connected through a Full-Duplex Gigabit Ethernet network. Each node has two Power PC 970MP processors at 2.3 GHz and 4GB of memory. Even we could use a maximum of 4 threads per node, just one thread will be used for all benchmarks, because the congestion problem already appers with this number. In case more threads were used, the problem would be worse, due to the fact that more threads would reduce the computation time, and so less time would be left for the communication.

V. Performance results

A. Gigabit Bandwidth

To show the congestion problem of the network we built a client/server code that shows the bandwidth variation when sending messages of 4096 bytes in function of the number of destination nodes used. This behaviour will correspond to the worst case in a synchronization point, where all modified pages are sent to other nodes. The code sends a fixed number of messages with a fixed size to all running clients and waits for an acknowledgment from each one.

The measured network bandwidth for the Gigabit interface is shown in Figure 6. With 2 nodes, the bandwidth increases steadily till a maximum bandwidth of 150MB/s is achieved, where it remains stable. But, when more than 2 nodes are used, even the bandwidth increases steadily with small number of messages, it arrives to a point where a maximum bandwidth of 170MB/s is achieved, and then it falls down quickly, going to a minimum of 25MB/s. This saturation point appears when sending 16–32 messages.

B. Synthetic Benchmark

Speedup for the synthetic benchmark, and its presend and chopper versions are shown in Figure 7. The sequential time for this benchmark is 53.8 seconds.

The overhead of this benchmark is directly related to the number of memory accesses. This means that when the number of nodes increases, the number of memory accesses per node diminishes, and consequently the execution time decreases.

For the presend version, when a parallel loop finishes, all the pages needed by the next loop are distributed, and the execution time is better than the original one, as expected because the number of page faults is reduced.

When the chopper version is used we can see an improvement in the execution time, but the total number of page faults remains the same. This means that the presend version saturates the network, and limits the final performance. On the other hand, the chopper, is able to advance the data before the synchronization point, and so it avoids the congestion.

1) Chop Overhead: The overhead of a chop at the presend phase is relatively small. In the 8 nodes case, a chop takes 100us to handle 80 pages. This means an overhead smaller than 1% of the computation time, this small overhead is explained because it only needs to build a message with the list of finished page addresses and send it to the DSM layer.

The overhead to identify the page references and to detect the pages completed at each chop has a higher overhead, but it is negligible due to the fact that it is executed just once.

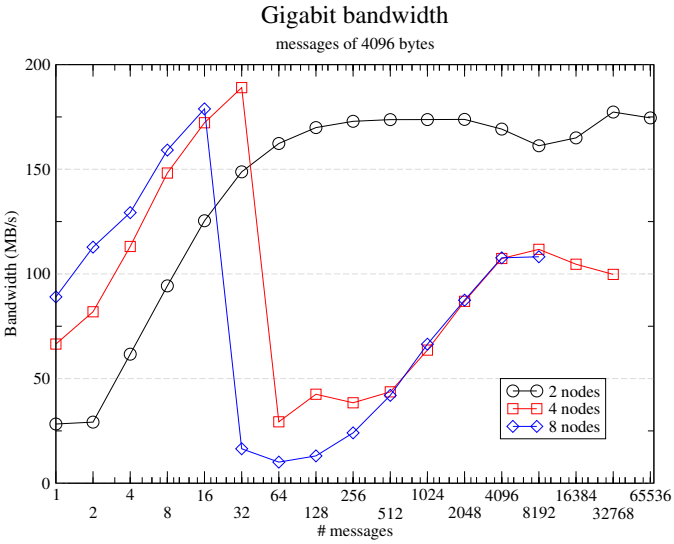


Figure 6. Bandwidth usage when sending 4096 bytes messages between 2, 4 and 8 nodes.

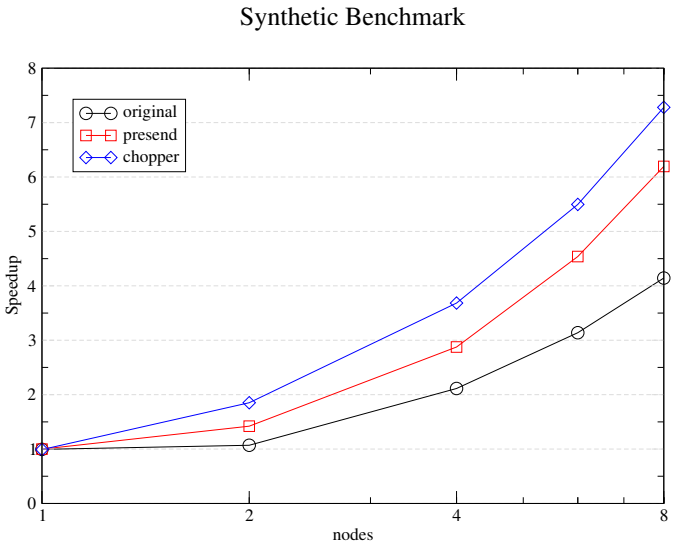


Figure 7. Synthetic benchmark speedup for different versions

C. BT Benchmark

In order to execute the 3.3 version of this benchmark in our environment we need to modify the code slightly, because our runtime lacks an implementation for the *threadprivate* directive. The solution has been to modify all the variables declared as threadprivate to be globally shared, and privatizing them each time that they are used.

To test our mechanism we used the classes A and B of this benchmark. The class defines the size N of the working matrices of NxNxN elements (64 and 102, respectively).

In class A, without any parallelism, a thread takes 46.88 seconds to execute 20 iterations of the main loop.

Speedup for this class is shown in Figure 8a. Original version has poor scalability. It has some slowdown with 2 nodes and a speedup of 1.9 is obtained when using 8 nodes. Presend technique achieves better results, speedup of 1.5 for 2 nodes and 2.3 for 8 is achieved. Even better speedups for all nodes are obtained when the chopper is used, 1.7 and 2.7 with 2 and 8 nodes respectively, meaning a performance improvement of about 16%. But this speedup trend seems to change slightly compared with the other versions, in fact a superior speedup could be expected with 8 nodes. As we will see later, this change is due to the lack of enough computation time to get profit from the chopper.

When the bigger class B is used (see Figure 8b), the original version behaves in a similar fashion to what has been observed before, but with a slightly better scalability. In contrast, the presend version is not able to improve the performance any further and just improves the scalability at 8 nodes with a speedup of 2.5. But finally, the chopper is able to get a better performance (similar to the obtained with the smaller class).

To understand the poor scalability of this benchmark, we will break down the execution time for each of its loops of the smaller class.

Figure 9 shows the total execution time per loop for the different versions of the BT benchmark class A when executed with 1, 2, 4 and 8 nodes (1 thread each). It includes time doing computation, solving page faults and waiting for any synchronization. There is an additional version called *Ideal* that shows the best execution time achievable per loop, where time executing page faults have been removed from original version.

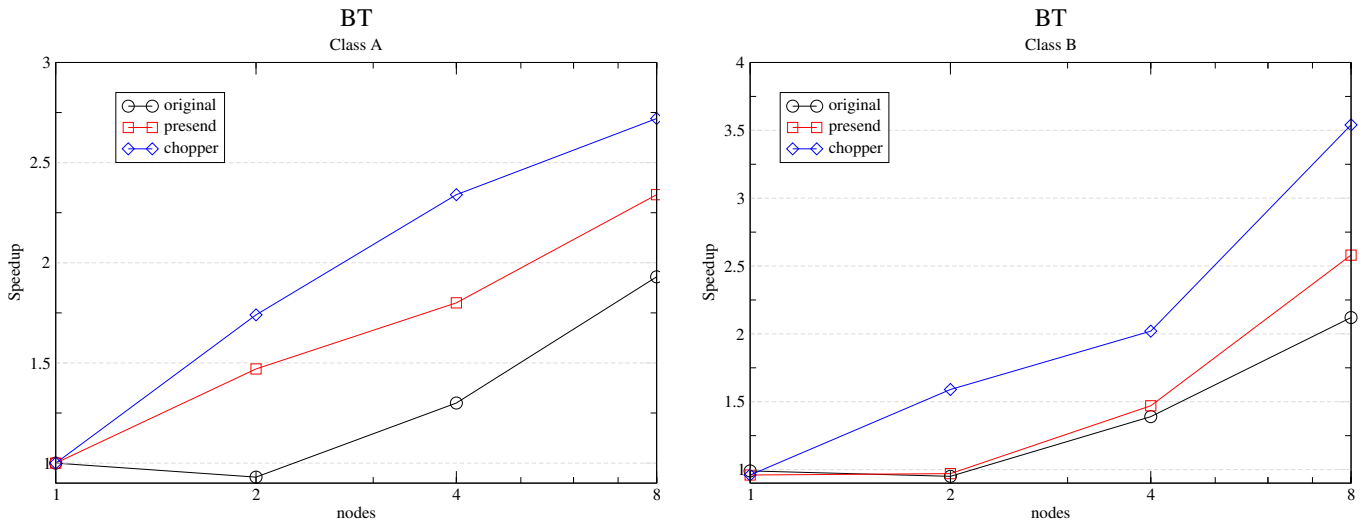
1) Original version: Looking at the sequential execution time (Figure 9a) we see that the solvers (`x`, `y` and `zsolve`) takes the major part of the total time (84%). When executed in parallel, two of these solvers achieve good speedup (`xsolve` and `ysolve`), but the `zsolve` and `add` loops increases its execution time a lot with two nodes, although their execution time decreases when the number of nodes increases. These alone can explain the poor scalability of this benchmark.

We find similar behaviours for loops at the RHS computation, but their impact in the final result is smaller. Just the `rhs3-4` loops achieve good speedup. For the remaining 9 loops, we find two behaviours. On one hand, there are three loops (`rhs5`, `rhs8` and `rhs11`) that takes more or less the same time independent of the number of threads used to execute them. On the other, there are eight loops (`rhs1-2`, `rhs6-7`, and `rhs9-10`) that exhibit the same behaviour explained for the `zsolve` loop.

Comparing original with ideal versions we see that more than half of time is due to page fault handling.

As said in section IV-C, the major part of loops are parallelized with the same dimension, meaning that all data structures in them are accessed with a local copy. But there are some loops (`rhs6-7`, `rhs9-10` and `zsolve`) that change this dimension. As expected, all data placement changes in these loops, producing a huge number of page faults to recover pages from other nodes. This happens each time that the parallelization index changes (`rhs8`, `rhs11` and `add`).

Finally, the `rhs1` loop writes temporal data structures that



(a) Class A (b) Class B

Figure 8. Speedup for the different BT classes.

are read later by other loops and so a lot of write page faults arise to invalidate all the extra copies.

2) Presend version: To avoid these page faults the pre-send/pre-invalidation technique is applied at each parallel loop. As expected, the solver loops (`xsolve` and `ysolve`) maintain its execution time and scalability.

The `zsolve` loop reduces its execution time for all nodes, but this improvement decreases with the number of nodes. This behaviour also appears in the `add` loop.

In general, the RHS phase, improves its execution time, but there are two visible behaviours. On one hand, there are a big portion of loops that behaves like the ideal version (`rhs3-rhs11`), improving their execution time with the number of nodes. But, on the other, there are a couple of loops (`rhs1-2`) that even their execution time improves, they get worse scalability than the original version.

3) Chopper version: Finally, to solve the presented problems at `zsolve` and `add` loops, we try to send data earlier by adding the chopper at the previous loops (`ysolve` and `zsolve`). A chop is used at each iteration. Like the synthetic benchmark, the number of page faults remains the same as in the presend case (Table I summarizes the number of page faults for all loops when executed with 8 nodes).

The execution time of `zsolve` and `add` has been reduced, but when the number of nodes increases, the improvement of the chopper is less notable, and more similar to the presend version. This is normal and an expected situation, due to the fact that the chopper uses the computation time of the current parallel loop to send data. When the number of threads increases, by the Amhdal's Law, the work to be done by each thread is smaller, meaning that there are less time to execute the chops. In the worst case, the chopper will behave like the presend version with a little bit of overhead when trying to execute an empty chop.

Table I
NUMBER OF PAGE FAULTS IN THE BT BENCHMARK FOR EACH LOOP WHEN EXECUTED WITH 8 NODES.

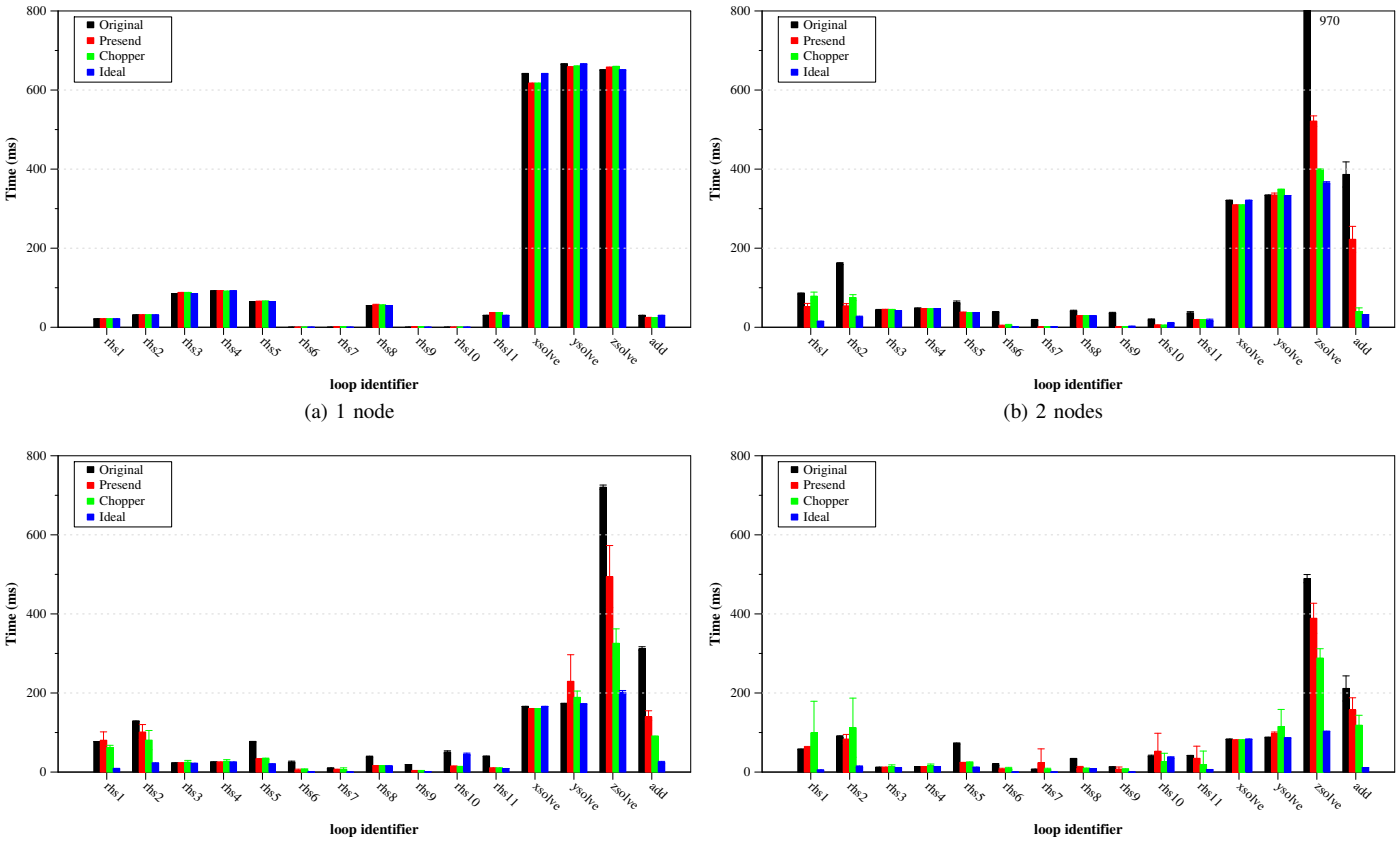
	original	**presend**	**chopper**
rhs1	280	103	103
rhs2	295	4	5
rhs3	131	1	1
rhs4	2	0	0
rhs5	94	33	32
rhs6	31	4	4
rhs7	18	4	4
rhs8	282	3	3
rhs9	38	4	4
rhs10	13	4	4
rhs11	161	3	4
xsolve	0	0	0
ysolve	4	4	4
zsolve	1177	264	269
add	593	137	139

D. CG Benchmark

A sequential execution of class A of this benchmark takes 12.82 seconds. Figure 10a shows the speedup for the different versions of this benchmark class. This class has very few computation per thread, and so the overhead of the page fault handling is difficult to compensate. As the figure shows, the speedup is quite poor. The maximum speedup achieved by the original version is just 1.3 with 8 nodes, basically due to the small computation available.

Applying the presend mechanism, a 30% better speedup is achieved, but again there is a peak at 8 nodes with an speedup of nearly two.

A small improvement is observed over the presend version (about 7%) when the chopper is used at the biggest loop.



(a) 1 node

(b) 2 nodes

(c) 4 nodes

(d) 8 nodes

Figure 9. Execution times per loop for different BT versions for 1, 2, 4 and 8 nodes.

The most interesting effect comes from the bigger class B, which takes 1399.71 seconds to execute sequentially. It has a super-linear behaviour for small number of nodes, but when the number of nodes increases, the curve has a peak at 4 nodes, and falls down to an speedup of 2.5 with 8 nodes.

The presend is not very useful here, because there are a lot of pages to be pre-sent, but the time between regions is insufficient to forward all data, concentrating all communication in a single point and saturating the network.

In contrast, when a chop is called at each iteration for the main loops, the chopper is able to maintain a perfect speedup till 8 nodes, because the communication is distributed through the computation time and the congestion problem is avoided.

These results are similar to the presented ones by [8], where the source code is reordered by the compiler to overlap the computation with the communication.

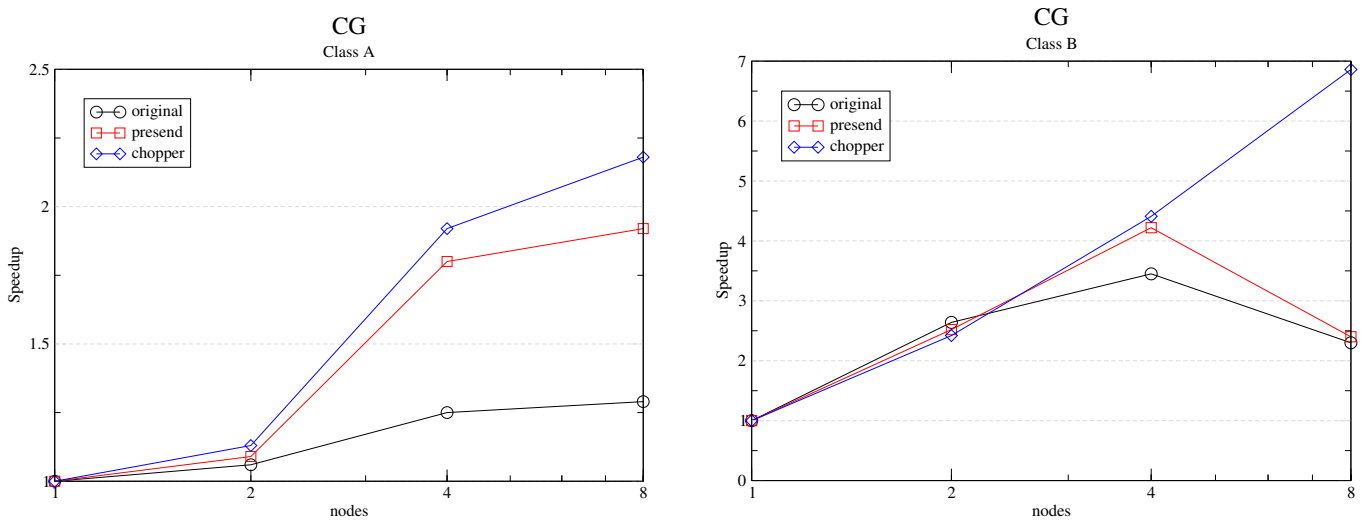
VI. Related Work

Since the DSM idea was presented by Li [9], a lot of DSM systems have appeared. The main difference between them being the platform used or the consistency protocol implemented. DSM systems implementing a relaxed consistency use synchronization points to send network messages to maintain memory consistency [10], [11], [12]. On the other hand, DSM systems implementing sequential consistency usually access the network in a page basis and so they do not need the synchronization points; usually, techniques like presend or data-forwarding [4], [13] are applied around synchronization points to improve performance.

As far as we know, nobody else has been done any research on distributing data delivery across time to avoid network congestion. But this topic has been extensively researched in the network area, like the TCP congestion algorithm [3], where a congestion window with the acknowledged messages is used to slowdown the client. The work presented here is somewhat related but instead of slowing down the client, we identify earlier points in time where data could be sent.

In fact, the OpenMP specification also offers an explicit synchronization point to the user: the *flush* operation [14]. This operation enforces consistency between a thread's temporary view and global memory. In contrast, our VSPs affects specific data (nor the whole memory), and data is affected if, and only if, it is not modified afterwards inside the current region.

A similar idea that associates data to a region of code dynamically can be seen with the Scope Consistency model [15], where they use the term *scope* to refer to a region a code. The main differences are the complexity of their interface for explicit scopes (which can produce an incorrect code if improperly used) and the fact that our proposal is implemented on top of a sequential consistency model.



(a) Class A (b) Class B

Figure 10. Speedup for the different CG classes.

VII. Conclusions

This paper shows the network congestion situation that arises when OpenMP applications, with a high volume of memory accesses, are executed on top of a page-based DSM. The congestion appears at OpenMP synchronization points, where the DSM exchanges messages to 1) maintain the global shared-memory consistent state and/or 2) to presend data to improve performance. This congestion, due to the limited bandwidth of the network, limits or degrades the performance of the applications.

This congestion can be avoided if a better temporal distribution for this messages is used. This paper proposes a technique to send the data earlier, distributing the data to be sent through various points between synchronization points, achieving a better overlapping of computation with communication. The technique is based on new virtual synchronization points added by the programmer in the application code; the runtime uses them to dynamically decide what/when data can be safely exchanged to minimize network congestion.

A synthetic microbenchmark and two NAS benchmarks (BT and CG) are used to demonstrate, on top of a DSM implementing sequential consistency, the potential of the technique proposed in this paper. Results show that the network congestion situation is reduced with the application performance increased by 16% on average. As expected, results also show that the proposed technique is limited by the computation time available to distribute all network data.

Acknowledgments

This work has been supported by the Spanish Ministry of Education (TIN2007-60625), by the Generalitat de Catalunya (2009-SGR-980), by the European Union (under the HiPEAC2 Network of Excelence FP7/ICT 217068) and by IBM through the CAS program. We would also like to thank the Barcelona Supercomputing Center for the use of their machines.

References

[1] "OpenMP Language Specification v2.5," November 2004. [Online]. Available: www.openmp.org

[2] Message passing interface web page. [Online]. Available: http://www-unix.mcs.anl.gov/mpi/

[3] M. Allman, V. Paxson, W. Stevens *et al.*, "TCP congestion control," RFC 2581, April 1999. [Online]. Available: http://rfc-editor.org/cgi-bin/rfcdoctype.pl?loc=RFC&letsgo=2581&type=http&file_format=pdf

[4] J. J. Costa, T. Cortes, X. Martorell, E. Ayguade, and J. Labarta, "Running OpenMP applications efficiently on an everything-shared SDSM," *J. Parallel Distrib. Comput.*, vol. 66, no. 5, pp. 647–658, 2006.

[5] J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta, "Nanos Mercurium: a Research Compiler for OpenMP," in *Sixth European Workshop on OpenMP*, 2004.

[6] X. Martorell, E. Ayguade, N. Navarro, J. Corbalan, M. Gonzalez, and J. Labarta, "Thread Fork/Join Techniques for Multi-level Parallelism Exploitation in NUMA Multiprocessors," in *13th International Conference on Supercomputing (ICS'99)*, june 1999.

[7] H. Jin, M. Frumkin, and J. Yan, "The OpenMP Implementation of the NAS Parallel Benchmarks and its Performance," NASA Ames Research Center, Tech. Rep. NAS-99-011, October 1999.

[8] A. Basumallik, S.-J. Min, and R. Eigenmann, "Programming Distributed Memory Sytems Using OpenMP," in *IPDPS*. IEEE, 2007, pp. 1–8.

[9] K. Li, "Shared virtual memory on loosely coupled multiprocessors," Ph.D. dissertation, Yale University, New Haven, CT, USA, 1986.

[10] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "TreadMarks: Shared Memory Computing on Networks of Workstations," *IEEE Computer*, vol. 29, no. 2, pp. 18–28, Feb. 1996.

[11] W. E. Speight and M. Burtscher, "Delphi: Prediction-based Page Prefetching to Improve the Performance of Shared Virtual Memory Systems," in *PDPTA '02: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*. CSREA Press, 2002, pp. 49–55.

[12] M. Sato, H. Harada, and Y. Ishikawa, "OpenMP Compiler for a Software Distributed Shared Memory System SCASH," in *WOMPAT2000*, July 2000.

[13] F. Mueller, "Adaptive DSM-Behavior via Speculative Data Distribution," in *IPPS/SPDP Workshops*, ser. Lecture Notes in Computer Science, vol. 1586. Springer, 1999, pp. 553–567.

[14] J. Hoeflinger and B. De Supinski, "The openmp memory model," *Lecture Notes in Computer Science*, vol. 4315, p. 167, 2008.

[15] L. Iftode, J. P. Singh, and K. Li, "Scope Consistency: A Bridge between Release Consistency and Entry Consistency," *Theory Comput. Syst.*, vol. 31, no. 4, pp. 451–473, 1998. [Online]. Available: http://www.springerlink.com/content/fm3ahf20p8p9p8e4/fulltext.pdf