*Joint NCC & IIS Bull.*, Comp. Science, 7 (1997), 1–19


# Group-oriented computation model for distributed shared memory systems

Mikhail Dorojevets

This paper describes a new group-oriented distributed shared memory (GDSM) computation model and its hardware support for improving performance in large-scale shared memory multiprocessors. The GDSM model is based on a concept of groups as basic computational units representing sets of parallel threads that cooperate and communicate by sharing an address space to solve large problems. These threads are assumed to run in parallel on separate processors in a globally-distributed shared-memory system. We consider parallel loops with inter-iteration data and control dependencies natural computational groups in scientific applications. The goal of introducing the group-oriented computation model is to 1) expose a space (logical group) dimension in the processes of creating and executing a parallel program on global heterogeneous systems and 2) exploit locality and predictability in the patterns of data sharing and processor communication by relaxing the memory release consistency model and providing multi-protocol communication within groups. This paper shows how GDSM features can be integrated into existing cache memory systems to tolerate remote memory access latency. An example of using the group-oriented computation model for parallel calculation of Fast Fourier Transform (FFT) is given.

## 1. Introduction

High performance in large scale shared memory systems requires low latency for memory accesses. Many architectural methods help hide distributed shared memory (DSM) latencies: multiple contexts [Smi78, Dor84, HaF88, ACC90], relaxed consistency models [DSB86, AdH90, GLL90, BZS93], coherent caches [ALK90, LLG90, LLJ92], and memory prefetching [LYL87, GGV90, MoG91, DDS94, DDS95]. Studies of performance of both dynamically and statically scheduled processors [LYL87, GGV90, GGH91, GHG91, GGH92] have shown the merits and limitations of each technique. These studies form a basis for the Netputer research to find a viable architectural framework to speed parallel program execution in DSM systems.

Group-oriented distributed shared memory (GDSM) is a new group computing model for scalable DSM systems. Hardware based on this model can exploit compiler knowledge about expected reference patterns for groups of processors using descriptor-controlled cache-to-cache communication with

low processor overhead. Each GDSM sharing descriptor is built from compiler-supplied information but allows runtime specification of which remote processors should receive data shared by the local processor. Shared data can be sent before they are actually needed. To determine destinations on-the-fly, memory system hardware performs built-in add operations on group processor vectors. Extra processor instructions are needed only if a compiler-supplied sharing protocol must be changed at runtime. Group processing and conventional non-group computation are both supported.

The GDSM method combines benefits of coherent caches, prefetching, and memory-based interprocessor communication with flexible runtime selection of data sharing recipients. It avoids almost all slow software control. Other benefits of the GDSM model come from safely using a more relaxed model of memory consistency within a group than between groups.

## 2. The group-oriented DSM model

The GDSM model was created to provide a systematic way to overlap memory access delays efficiently with useful computations in parallel numeric applications running on globally-distributed shared memory systems. It is assumed that such a system is heterogeneous with some its elements connected tighter than others. The latter means that different groups of processors in the system can have communication links with very different latency and bandwidth.

The group-oriented shared memory provides a framework to improve load balance and decrease communication and synchronization latency in executing parallel programs on such heterogeneous shared memory systems. In scientific programs, isolated accesses to shared variables are relatively rare. Most occur within loops with inter-iteration dependencies that have statically-predictable reference patterns.

The goals for introducing the group-oriented computation model are to 1) exploit locality in the network topology in executing a parallel program on heterogeneous distributed systems and 2) exploit locality and predictability in patterns of data sharing and processor communication.

Parallelization process for the GDSM model goes through decomposition the computation of a sequential loop into a collection of tasks consisting of parallel iterations with further assignment of these tasks to parallel threads. Groups of processors can execute these loop iterations in parallel if they honor all inter-iteration dependencies. The group-oriented computation model reflects itself in a way by which the partitioning and mapping steps are to be done. It provides a notion of groups as computational objects describing tightly-coupled computation within loops. A group is a collection of parallel threads sharing the same address space and acting together in some

specified way. The number of parallel threads created by a compiler/programmer within a group to execute loop iterations determines the desired size of some tightly-connected section of the distributed system needed to execute this group. The operating systems map all threads of the group to physical processors within the section at run time. Such group mapping improves load balance especially if the compiler can map the data shared by these threads to the same section of the distributed system.

Each group is characterized by two parameters: its data sharing policy and its set of member processors. Initial definitions of group parameters are based upon specifications from either a compiler or a user. Parameters can be updated at runtime. Multiple groups can exist and execute simultaneously on separate subsets of processing nodes in a distributed system.

GDSM speeds program execution within groups by imposing fewer restrictions on event ordering within a group than between groups, and by providing group communication protocols supported directly by hardware for sharing data rapidly within a group. The first improvement results from group release consistency (GRC), an enhancement of release consistency (RC) [GLL90]. Besides allowing write buffering and pipelining like RC, GRC reduces times for write operations executed inside groups by not waiting for completion acknowledgments from processors outside the group.

To a conventional invalidation-based protocol with many readers or one writer, GC adds replication and migration to deliver data, before they are requested, to any subset of processors in a group. To know where to send runtime data, hardware-visible descriptors created at compile time specify group members and sharing protocols. Group communication messages pass data blocks of cache-line size among group caches linked by a low-latency network. Each node has one group cache on its memory bus, as shown in Figure 1.

## A. Group release consistency

Group-oriented distributed shared memory is based upon a new group release consistency (GRC) model. Like other relaxed consistency models, weak [DSB86], release [GLL90], and entry consistency [BZS93], GRC enforces consistency on sets of operations done within critical sections of parallel programs.

Like RC, GRC requires the compiler to use special acquire and release operations to access shared data, namely **load** and **unload** (group) **descriptor**. In any RC model, acquire is a read operation that gains permission to access a set of data, and release is a write operation that relinquishes such permission. GRC uses the rules of RC with one small change: it treats them locally, applied only to all processors in one group, not to the whole system. Rule two of the release-consistent DSM [GLL90] applied to

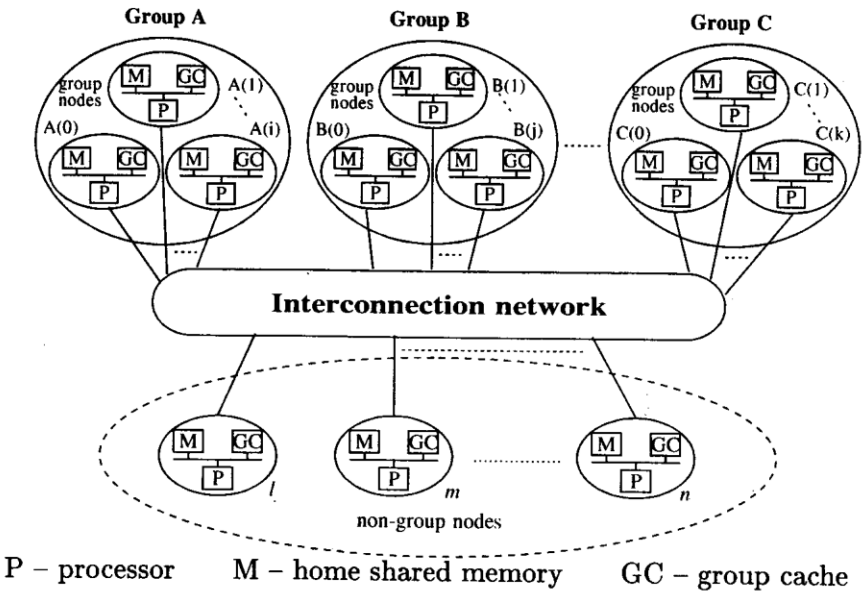


**Figure 1.** Multigroup computation in a group-oriented DSM system

a group release (**unload**) becomes: *Before a group release operation is allowed to be performed, all reads and writes previously started by all processors* **inside the local group** *must be performed.*

The small change makes a significant difference between global RC and local GRC in completing write operations. Both RC and GRC assume that until a write operation has been performed, all requests from remote processors to read (share) the value must be rejected or delayed. For traditional global RC synchronization, after writing a new shared value, a processor must receive update or invalidate completion acknowledgments from all processors with copies, including distant (non-group) ones, before honoring a read or write request for the new value. With group-local GRC, a processor can share a new value immediately after receiving acknowledgments from all processors within its group. Using GRC, a request from a processor in the current group to read local data written by a peer processor in the group can be serviced earlier than a read request for remote data written by a processor outside the group. No processor outside the currently active group can access data written within it until group execution ends.

The GRC model allows write operations to complete within a group of processors that interact frequently by sharing data, even while some processors not belonging to the group have yet-valid older copies of the variable. To erase these copies, "external" invalidations are sent to processors outside the group. They officially complete when their acknowledgments return to

the processor that has written the new value. Old external copies may exist where invalidations have not yet reached or when acknowledgments of invalidations have not yet been received. A GRC-consistent system must distinguish intra-group and external memory access events, including inter-processor invalidation, update, and acknowledgment messages.

## B. Multiprotocol group communication

Conceptually, group membership is organized as an indexed vector with one bit per processor. In fact, processor identity in a group is specified by two values: the number of processors in the group (**GPN**); and its home index (0 <=**H-index**<GPN), or bit position in the logical vector. Network interface hardware for each processor keeps a runtime map table (MT) pairing the H- index and GDSM system physical address for each processor in the currently active group. Processors in a group have the same MT and GPN, but distinct H-indices. Membership of a group cannot change after creation.

Three group communication data sharing protocols can be used within groups:

1. A conventional write-invalidate protocol allows multiple readers or a single writer per cache line. After a processor writes a new value for a shared (Copied) memory block, only its cache has a copy.
2. A replication update protocol writes a new shared value to local cache, then immediately multicasts it to all other processors in a group.
3. A migration protocol sends a value newly written by a producer processor to a single destination processor and invalidates any copies in producer caches. If the producer is also the destination, migration acts like the conventional invalidate protocol.

The single-destination migration protocol handles common "ping-pong" cases where two processors alternately read and write a shared variable. It is also generally useful in guarding data protected by synchronization variables that provide mutually exclusive access to critical variables. Data can migrate from processor to processor as critical regions are entered and exited, but must not be replicated.

The migration protocol uses a home-relative M-index to specify which processor in the group will receive values written by the current processor. The destination processor number results from hardware addition or subtraction of sender H-index and M-index values, modulo group size. Runtime hardware can increment, decrement, and rotate M-index values. Operations based on H- indices let all processors within a group use the same descriptor, but send their data to different members, if needed.

Replication and migration protocols can provide other processors with data they may need in the future. These shared data are assumed to be

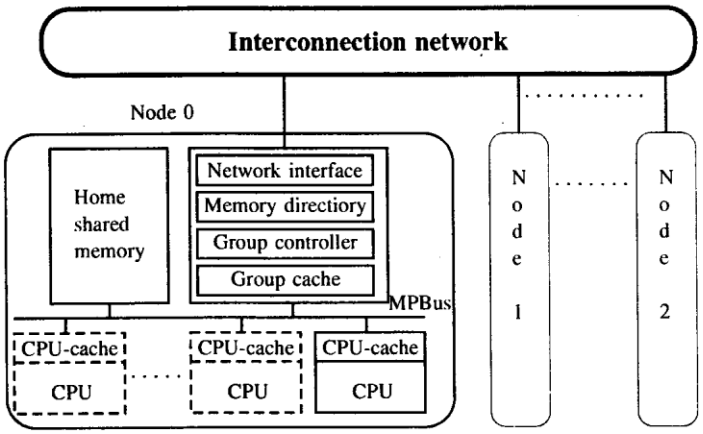


**Figure 2.** The Netputer multiprocessor distributed shared memory system

loaded only in the second-level (group) caches for network nodes. For all protocols, a read request fetches data only to the local processor.

## 3. The netputer GDSM architecture

The group-oriented distributed shared memory (GDSM) model is supported in planned hardware for Netputer, a large-scale parallel network system. Netputer consists of many processing nodes connected by a high-bandwidth low-latency network, as in Figure 2. Netputer physical memory is distributed among the nodes, but each processor can access all of memory.

Each Netputer node contains a processor with on-chip write-through caches for instructions and data, its portion of globally shared memory with the corresponding part of the memory directory, a second-level write-back group cache, and a group controller that supports GRC-coherent memory through a runtime line tagging mechanism, cache-to-cache communication and a write-buffer. Each processing node can be a symmetric, bus-based multiprocessor with shared memory. For simplicity, most of this paper talks of only one processor per node. The first-level on-chip cache is a subset of the second-level cache. The second-level cache also holds all cache lines received from remote processors. The group controller monitors the local processor memory bus to detect all writes and to maintain shared data consistency. It also provides the interface between the network and local processors. A write buffer several words depth holds parameters for each on-going write request until each block is owned exclusively by the local processor and has its cache changed to complete the write. The processor itself does not wait for writes to be completed.

Netputer provides hardware support for group processing. It has hardware-recognizable group descriptors, operations on them, and a scalable coherent (GRC) memory interface controlled by a distributed directory and tagged caches.

## A. Architectural support for group-oriented computation

Group descriptors (GDs) are synchronization variables to provide group access to shared data within critical regions. Each encapsulates all information about members of one group and their data sharing protocol. The compiler loads group descriptors into protected pages. Read or write accesses to these pages via TLB cause traps which call system library procedures such as **enter_group** and **exit_group**, which jointly form the group manager for a Netputer. In executing **enter_group** to request **group_acquire** for a particular lock, the group manager grants the request if there is no competition for the lock. To create a new group, the manager allocates as many processors as needed for the group and provides each with a unique group number, a home processor pointer and a map between logical and physical processor numbers. The enter_group procedure completes by prefetching descriptors for the new group into the group cache of each processor node in the group.

The group controller for each processing node has a current group register (CGR), given in Table 1.

**Table 1.** CGR (current group register) in group controller

| Field | Description of operation |
|---|---|
| GID | unique group identifier (used to tag group cache lines) |
| WPR | current group write protocol (how share all new data) |
| GPN | number of processors in the group |
| ROF | read-only flag (1-Rd, 0-Rd/Write) (for read requests) |
| M-index | migration index (new processor id minus old) [Migrate] |
| H-index | home index (bit of this processor in id vector for group) |

A processor switches into the group execution mode after fetching a group descriptor into its CGR by a **loadCGR** (GD-address) (**group_acquire**) operation. When its CGR is loaded, group cache hardware applies the group sharing protocol specified in CGR to all writes generated by this processor for variables shared within the group. Each processor can control sharing policies by executing runtime CGR-operations, such as **setCGRrpl**, as shown in Table 2.

**Table 2.** Netputer group memory access operations

| Group operation (param) | Description of operation |
|---|---|
| **loadCGR**(GD.mem.adr)<br>**unloadCGR**(GD.mem.adr) | **group_acquire** shared data operation<br>**group_release** shared data operation |
| **setCGRcnv,**<br>**setCGR rpl**<br>**setCGR mga, mgs** | **write protocol**=Conventional<br>**write protocol**=Replication<br>**write protocol**=Migrate to H ± M-ind |
| **CGR++/−** *[Allowed*<br>**CGR<</>>** *only*<br>**inc/decCGR**(val) *for*<br>**lft/rgtCGR**(val) *migrate]* | **increment/decrement** CGR **M-index** 1<br>**rotate left/right** CGR **M-index** 1<br>**increment/decrement** or **rotate**<br>**left/right M-index** by val |
| {**group, global, local, processor**}_**sync** | **wait for** completion of all **memory** accesses for a given type |
| **read/writeCGR**(R) | **fetch/save CGR** from/to register R |
| **read/writeMT**(MT.adr, mem.adr) | **fetch/save Map Table** contents from/to memory |

A **group_sync** operation, used before a **group_release**, stalls a processor until all memory accesses started by group processors have been performed in the group. A **global_sync** stalls until all accesses from the group have been performed globally. A **local_sync** stalls a processor until all its own outstanding operations complete in the group. A **processor_sync** stalls a processor until its outstanding operations have been performed globally.

A processor finishes group execution by performing an **unloadCGR**(GD.address) (**global_release**) operation that frees it from the group in the current group register. This operation completes only after all accesses from this group have been performed globally. Once the operation is completed, the group controller for the processor which issued the **loadCGR**(GD-address) (**acquire**) operation sends a **writeCGR**(GD-address) request to memory. The write request to the protected GD-variable causes a trap which calls an **exit_group** system procedure which unlocks access to the critical region.

## B. Memory directory and cache coherence protocols

Each processing node has a local part of the memory directory to service all accesses to distributed shared memory residing on the node. Netputer memory uses tagged directories based on the pointer cache directory scheme [Lil91, Lil93] and a group-oriented invalidation-based ownership protocol. This scheme maintains a node pointer of $\log_2 p$ bits for each address tag of log2m bits, where p and m are the number of processors and memory blocks of cache line size in the whole DSM system. When multiple processor caches have the same memory block, entries with the same address tag are

allocated as distinct pointers in the directory. The degree of associativity of the cache directory limits the number of processors sharing one line. If too many processors try to share a block, an older pointer to the block is chosen at random to be eliminated after its cached copy of the shared block is invalidated.

A read request completes when the requesting processor receives data (D-Reply). Writes send many invalidate requests to other processors. Until all requests are acknowledged, a write access is not officially done, and remote processor requests to read (share) the value must be rejected or delayed. For Netputer consistency, each D-Reply to a memory access contains a count (nACKs) of other requests issued as part of the access that processors must acknowledge before the access is complete.

Netputer supports three protocols: a **conventional** invalidation protocol for group and non- group sharing; plus **replication** and **migration** protocols, for interprocessor sharing in a group. A pointer with Dirty-state, left by **conventional-** or **migration-write**, means a modified copy of the now-stale memory block is cached by the processor specified in the p-field. Copied-state, after a **data-read** or **replication-write**, means that an unmodified copy of the block in memory is cached by the specified processor. To perform a **replication-write** in a group, the **directory** for the address stores the data in memory, sends invalidations to the processors in the p-fields of all valid pointers to the block, and allocates a lone Copied-state pointer. The pointer specifies the processor issuing the **write** as the new owner of the block. If a block newly written in any way was previously copied by a **replication-write**, cache copies may exist without explicit memory directory pointers. This indirect pointer strategy can greatly reduce the count of memory directory pointers needed when sharing by using one replication-write in a group instead of many conventional reads from individual processors.

Before loading a new value into a cache block, stale values are invalidated. An invalidation to an old owner normally causes it to invalidate all its copies **replicated** without directory pointers. However, to avoid old invalidations of new values, if **replication** is in the same group, the old owner lets the group controller for the new owner handle all invalidates and updates within the group.

One other optimization reduces memory directory size. A read-only flag in each group descriptor (Table 2), accompanies each read request to memory. Netputer memory directories do not allocate new pointers for accesses to data flagged as read-only. Several studies [AgG88, EgK88, WeG89] show that programs heavily access read-only blocks. One [LiY93] shows how sharing analyses by compilers can reduce tagged directory size by allocating only pointers needed to keep coherence.

## C. Group cache control

In Netputer, caches are kept coherent at runtime by hidden exchanges of requests between memory directories and group controllers. Possible states for lines in first-level (processor) caches are: **INValid**, **EXclusive** and **SHared**. Second-level (group) cache lines have those three states, plus **Group-Shared** (**GShared**) for replications. The group controller in each node executes group operations, manages the local group cache, and keeps the first-level and group caches consistent.

As Figure 3 shows, each line in a Netputer group cache has four fields: state, group tag, block address and data parts. A cache line in **EXclusive** state implies that a block has been modified; this cache holds the only copy. **SHared** state marks a cache line when the line is exactly like the same as the block in memory and there are probably its clean copies in other caches. **GShared** state marks a cache line modified by a **replicate-write**; it may be shared by other processors without directory pointers. When the line was last replicated to the group caches for the processors, memory was updated to hold the correct value for the block. The one processor with a cache entry in **GShared** state is the current group owner of this block.

| **State** bits: | **GrouptagN$_c$**: | **Memory** block address **A$_c$**: | **Data**: |
|---|---|---|---|
| 2 | 14 | $\log_2 m$ | 64-256 |

States: **INV**alid, **EX**clusive, **SH**ared, Group-Shared (or **GS**hared)

**Figure 3.** Fields for one cache line in the group cache

After a **replication-write**, only the owner processor has a pointer in any memory directory even though there are many other cache copies. The strategy of not registering some memory block copies within a group requires group cache hardware for each owner to manage sharing of replicated blocks. Netputer group cache control mechanisms guarantee that the directory together with the group caches can find and invalidate all cache copies of a block, registered in a memory directory or not.

At any time only one group has parameters loaded into each CGR and network interface and is active for a given set of processors. After the threads composing a group finish executing, processors can be allocated to a new group of threads. The final **unloadCGR**(GD-adr) ending group execution followed by the initial **loadCGR**(GD-adr) operation for a new group defines a "group boundary". When a group starts, on-chip and group caches may have lines in any state (**INV, EX, SH, GS**) left from older group executions. Processors executing the current group of threads and receiving invalidations from the memory directory cannot find and invalidate

unregistered replicated copies produced within other groups. Each processor has a processor membership table (MT) for only one group at a time.

There are many ways to invalidate unregistered copies. A poor method is to force group controllers to flush local caches to memory at group boundaries. Valid cache lines registered in a directory are removed too. By losing temporal locality between groups, this poor solution would cause many cache misses at the start of each group. A more selective solution is to provide a **GShared** cache line state to mark the owner copy of each line loaded by a **replication-write**. All other (unregistered) copies within the group are in **SHared** state. Registered lines in cache after a load (**read**) are also in **SHared** state. Only one copy from each set of replicated lines needs to be in **GShared** state and registered; the rest can easily be found given the first. Before crossing a group boundary, processors can scan their group caches for all **GShared** lines. All processors in the group must be sent invalidations for unregistered (**SHared**) copies of all **GShared** lines. This second solution preserves temporal locality across group boundaries, but uses sequential scans and invalidations that can greatly increase execution times for programs with many groups of short duration threads.

Netputer uses a third, better method that avoids sequential scans when crossing group boundaries. Besides a **GShared** state, each line in a Netputer group cache has a **group tag** telling which group was active when it was loaded. The group software manager guarantees unique identifiers for all group descriptors in one program. When group numbers may need to be reused, the manager forces nodes to flush caches to memory to ensure that no tagged lines remain. Group tag number zero is reserved for non-group execution. The use of group tags makes all lines produced by **replication-writes** in one group, and only these writes, automatically become stale after the group finishes. All other group cache lines, either **read** into cache from memory or produced by **conventional-** and **migration-writes**, cross group boundaries. Lines in conventional first-level on-chip processor caches are not affected by group boundaries.

To ignore stale lines at runtime, Netputer group cache hardware repeatedly compares the group tag $\mathbf{N_c}$ of cache lines to the group number $N_g$ from the processor current group register (CGR). It also gives current tag $N_g$ to each newly created line. After a processor performs a **replication-write** to a block, its group cache has a new line with state **GShared** and tag $\mathbf{N_c} = \mathbf{N_g}$. All other group caches have copies of the line with $\mathbf{N_c} = \mathbf{N_g}$, but **SHared** state. Only **replication-write** creates non-zero tags. All other read and write operations, executed both in and out of a group, produce group cache lines with tag $\mathbf{Nc} = 0$. The group cache operations involving $\mathbf{N_c}$ and $\mathbf{N_g}$ can easily be implemented by hardware. The rule for determining if a group cache has a hit at a cache line with address $\mathbf{A_c}$ and tag $\mathbf{N_c}$ for an access seeking address $\mathbf{A}$ from group $\mathbf{N_g}$ is:

if$((A == A_c)\&\&((N_g bit - ORN_c) == N_g))$ then the group cache line matches, yielding a hit.

# 4. An example using groups for parallel FFT

This section shows how group communication can speed execution of a large application code, parallel linear Fast Fourier Transform (FFT) [Pea68]. Figure 4 gives the heart of Netputer group- oriented parallel FFT.

```
 /* Parallel linear FFT by P CPUs on array A of N complex data */
 /* Transform data in A into Fourier coefficient sums in place */
int N, P, S;                    /* number of data, CPUs, & passes: S= log2(N) */
int ID, otshx, SZ, ZR;                      /* CPU id, outsharing distance in A */
int isp;                                    /* result pair separation, *2 = size of segments */
int j, k, lc;                               /* loops: FFT pass(1..S), index in left segment */
complex A[N], AR[2,N];                      /* local inputs, shared partial results */
complex T, U, W;                            /* local variables */
main()                                      /* each CPU shares results one way per pass */
{loadCGR(GD);                               /* start execution of process group GD */
 P = CGR.GPN;                               /* number of CPUs in group */
 ID = CGR.Hndx;                             /* home CPU running this code */
 SZ = N/P;                                  /* size of segment of A done by each CPU */
 ZR = ID*SZ;                                /* index of first A,AR result pair for this CPU */
 CGR.Mndx=1<<(log2(P)-1);                   /*CPUs share P/2 data apart */
 setCGRmga;
 for (k=ZR; k<ZR+SZ; k++) AR[(k+N/2) mod N] = A[k];
 for (j=1; j<=S; j++){                      /* MAIN LOOP: sum FFT in A,AR.*/
  group_sync;                               /* complete all group migrations for pass-1 */
  isp = 1<<(S-j-1);                         /* isp = 2^(S-j-1), result offset, spans angle PI */
  U = complex(1.0, 0.0);                    /* angle 0, for initial e^i-angle = 1 */
  W = complex(cos(PI/isp), sin(PI/isp));    /* interdata e^i-angle */
  CGR.Mndx>>1;                              /* halve M-index, CPUs get closer */
  if ((ZR div isp) mod 2)                   /* will new AR sums be ..*/
     { M = setCGRmga; otshx = +isp }        /* shared right or right */
     else { M = setCGRmgs; otshx = -isp }/* left in this pass? */
  for (k=ZR; k<ZR+min(isp,SZ); k++){        /* FFT KERNEL sums */
   for (lc=k; lc<ZR+SZ; lc+=2*isp){         /* Did last AR go right? */
   if ((ZR div (2*isp)) mod 2) T = (AR[(j mod 2), lc]-A[lc])*U
                          else T = (A[lc]-AR[(j mod 2), lc])*U;
     A[lc] = A[lc]+AR[(j mod 2),lc];        /* Update part sum */
     M:: AR[((j+1) mod 2),lc+otshx] = T;    /* Sum for next pass.*/
  }
  U=U*W;                                    /* e^i-angle for next point in segment */
  }                                         /* end of FFT KERNEL */
}                               /* Done, no sync: last pass all local. Odd-index sums in AR.*/
 for (lc=ZR+1; lc<ZR+SZ; k+=2)
  A[lc] = AR[lc+(S mod 2)*N];               /* put final AR sums in odd A's*/
 unloadCGR                      /* await global completion of all group writes */
 }                              /* FFT end: Fourier coefs for freqs in order in A on P CPUs */
```

**Figure 4.** Netputer code for computational heart of 1-D FFT.

This application code performs a linear FFT for **P** processors and **N** complex-valued data points, with **P** and **N** being exact powers of 2. For

each inner loop (**lc**) within an outer pass (**j**), each CPU merges a partial sum **AR**[**lc**] produced by a possibly remote CPU and one local one **A**[**lc**] to form two complex sums and send one **AR** to another CPU. Each CPU figures **SZ** result pairs (**A, AR**) in each mid-level (**k**) loop. To avoid **AR** results for the next pass overwriting **AR** inputs to this pass, **AR** contains two values per index, one written in odd passes and one in even. For simplicity, the initial 0 or 1 index for **AR** are ignored in Figure 3. This code sums the input data in their original order by using a second array **AR** as well as initial input data array **A**. Before pass 1, each CPU migrates a copy of its local segment of data in **A** to **AR** on the one other CPU. Since the migration distance is exactly P/2 CPUs and destination indices are calculated modulo group size **P**, additive migrations — **setCGRmga** — can be used to prefetch initial **AR** data inputs shared both right and left **N/2** data positions. After the last pass, successive Fourier coefficients are in **A**[**2n**], **AR**[**2n+1**] pairs on each CPU. Odd-indexed **AR**[**2n+1**] are moved to **A**[**2n+1**] to leave all in **A** at the end. The group-oriented code uses migration-writes to share each new partial sum with at most one other processor as soon as it is calculated. After $\log_2$ P passes, the **Mndx** index offset is zero in its low bits; new **AR** values stay on the CPU where they are calculated. Only ten **bold** lines must be changed to convert a more standard demand fetch version of this code into a much faster Netputer version.

## 5. Related work

The new group-oriented model for DSM systems allows an efficient hybrid of software and hardware mechanisms to provide global shared memory. Many investigations of weak consistency models and coherent caches have influenced this work.

Group release consistency (GRC) is a local relaxation in groups of classic release consistency RC [GLL90]. GRC lets interprocessor sharing accesses in groups finish earlier than does global RC.

The Stanford DASH [LLJ92] demonstrates how mechanisms needed to support cache coherence in DSM systems can be implemented with a protocol-engine associated with each private cache and each memory module. DASH uses a distributed directory invalidation protocol between processor clusters [LLG90] and snooping within. It supports RC by counters and fence operations. DASH uses a (p+1)-bit full memory directory [CeF78]. DASH supports invalidation and two update mechanisms: 1) Update-write: newly produced data are rapidly sent to all processors with a cached block copy; and 2) Deliver: after completing a sequence of writes into its cache using invalidation, a processor issues a deliver instruction to specify which clusters should receive a copy of the new block.

The Paradigm multiprocessor [CGB91] uses full-bit vectors to maintain coherence between and within processor clusters. Besides an invalidation ownership protocol, Paradigm memory passes messages to allow cache lines for shared memory to be message buffers. It supports message exchanges by two additional bits in each cache directory entry and one special bus operation, Notify. A Notify issued for a given line sets its state to request_notification and its P-bit vector bit to 1. A later writeback for a request_notification line interrupts every processor registered as a receiver of the line.

MIT Alewife [ALK90] has multithreaded nodes within a mesh. It has a LimitLess directory[CKA91] with hardware for 4 pointers per address tag and software interrupts if multiple shared copies or a (p+1)-bit full directory is needed.

Netputer directory methods allow multiple readers to share a block, depending on both processor activities and types of memory requests. In non-group execution, at most S processors can share any block, where S is memory directory associativity. S is much smaller than the number (P) of processors in a Netputer. By using group caches and controllers, Netputer memory directory supports a full (p+1)-bit scheme in hardware with neither a full (p+1)-bit vector per memory block nor software interrupts. Full physical processor addressing is achieved via a single full logical-to-physical CPU map table per group controller. Multiple processors in a node share one group cache and controller. Netputer hardware supports full processor sharing only for memory blocks replication-written during group execution.

At runtime, Paradigm forms lists of processors requesting a block; Netputer uses compiler analyses of group interactions to specify which processor(s) must be sent new data in advance. There are only two ways to send data within a group: to all processors or to one. Having only replicate (all) or migrate (one) variants simplifies Netputer cache maintenance. It avoids using slow interrupts, as Paradigm does. Netputer replication-write operations use one list of destination processors per group, not per separate memory access as DASH deliver operations do.

Dahlgren et al. [DDS94] show that a basic directory-based write-invalidate protocol augmented by simple extensions including a migratory sharing optimization and a competitive-update mechanism can eliminate a substantial part of the memory access penalty while keeping the hardware complexity of the memory controller moderate. These results have influenced our choice of the Netputer protocols to be implemented for the intra-group computation.

The Netputer descriptor mechanism allowing logical processor addressing and run-time selective sharing for data migrations was inspired by the inter-thread communication mechanism implemented in the Russian MARS-M computer [VGD87, DoW92], a tightly-coupled multiprocessor system with

multithreaded processors and shared memory. MARS-M threads communicate with each other through logical channels (hardware pipes), and channel hardware maps logical to physical channel numbers at runtime. However, MARS-M channels are not mapped into shared memory space.

Software cache-coherent systems [ChV89, MiB89] first used the idea of marking cache lines with tags (called version numbers [ChV89] and timestamps [MiB89]) to determine whether these lines are up to date when they are referenced. Another hybrid approach [ChV91] uses a directory only to ensure that all cached blocks are updated with the correct state at the parallel- task boundary, and to avoid dynamic invalidations. The directory monitors memory references generated by a program and dynamically updates its state to specify which caches contain which memory blocks, and whether they are modified. At each parallel task boundary, a processor sequentially scans its cache and invalidates cache entries that the stored directory information specifies should be invalidated.

Netputer needs neither separate memories to hold tags nor dedicated instructions to use tags as software coherent schemes do. Netputer relies on a software system manager only once during runtime, just to provide a unique number for each group. Executing a group, Netputer uses the current group number to mark lines produced by replication-write operations. In contrast to the hybrid approach [ChV91], Netputer processors do not use sequential scans at group boundaries. However, a weakness of Netputer tagging versus compiler-directed mechanisms is that to provide correctness, Netputer treats all replication-produced cache lines as stale whenever it exits their group.

Netputer shares many goals with Munin [BCZ90, BCZ90b] that uses software to implement the release consistency model. Munin relies on annotations by programmers to declare synchronizers and other shared variables. The main difference between Munin and Netputer, besides implementation, is that Munin is variable-oriented and Netputer is parallel-task or group- oriented. Netputer also distinguishes synchronizers (group descriptors) from other variables and uses either compiler or programmer assistance to define sharing groups and protocols.

Another Netputer goal, not otherwise discussed in this paper, is to optimize sharing along highly likely program paths, not just in sharing individual variables as Munin does. Usually Netputer protocols specified by descriptors apply to groups of memory references, not just to individual ones. Protocols can be changed at runtime. If necessary, a programmer or compiler may specify different protocols to access the same variable along different paths within the same or other groups.

To decrease the hardware costs and provide flexibility of hardwired memory-protocol engines like the DASH one [LLJ92], the Stanford FLASH [KOH94] and the Wisconsin Typhoon [RLW94] designs have provided ded-

icated programmable processors to emulate memory protocols by software handlers. These designs allow users to access these communication processors directly. Much (perhaps even all) of the functionality of the GDSM coherence protocol can be implemented using these dedicated communication processors. Thus, we plan to analyze advantages and disadvantages of using special programmable dedicated processors rather than a pure hardwired support proposed in this paper to support the group-oriented model of computation.

The current status of the Netputer project is the building of simulation models to refine our design choices. We plan to simulate the memory behavior expected from complex parallel systems that are executing realistic application codes, and to explore compiler analysis techniques to determine how much critical information about processor and memory usage patterns we can easily extract before execution.

## 6. Conclusions

This paper describes new architectural features for distributed shared memory (DSM) systems optimized for joint computations by groups of logically-related processes. It shows how Netputer features can be integrated into existing cache memory systems to speed parallel scientific computations, especially on heterogeneous distributed systems with thousands of processors. Novel features in this approach include: group release consistency, fast ways at runtime to modify compiler optimizations of data sharing embedded in software-modifiable write protocol designators, practical ways to provide update-based data replication to speed computations in huge systems, indirect pointers to replicated data to reduce need for explicit pointers in memory directories, efficient use of tagged caches for fast group context swaps, and ways to chain memory systems together for fast hardware controlled distribution of shared data.

## References

[ACC90] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, B. Smith, *The Tera computer system*, Proc. of Intern. Conf. on Supercomputing, 1990, 1–6.

[AdH90] S. Adve, M. Hill, *Weak ordering – a new definition*, Proc. of Intern. Sympos. on Computer Architecture, 1990, 2–14.

[AgG88] A. Agarwal, A. Gupta, *Memory-reference characteristics of multiprocessor applications under MACH*, ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems, ACM, New York, 1988, 215–225.

[ALK90] A. Agarwal, B.-H. Lim, D. Kranz, J. Kubiatovicz, *APRIL: A processor architecture for multiprocessing*, Proc. of 17th Intern. Sympos. on Comput. Architecture, 1990, 104–114.

[BCZ90] J.K. Bennett, J.B. Carter, W. Zwaenepoel, *Adaptive software cache management for distributed shared memory architectures*, Proc. of Intern. Sympos. on Computer Architecture, 1990, 125–134.

[BCZ90b] J.K. Bennett, J.B. Carter, W. Zwaenepoel, *Munin: Distributed shared memory based on type-specific memory coherence*, Proc. of Conf. on the Principles and Practice of Parallel Programming, 1990, 168–176.

[BZS93] B.N. Bershad, M.J. Zekauskas, W.A. Sawdon, *The Midway distributed shared memory system*, Proc. IEEE COMPCON Conf., 1993, 528–537.

[CeF78] L. Censier, P. Feautrier, *A new solution to coherence problems in multicache systems*, IEEE Trans. on Computers, **C(27)**, 1978, 1112–1118.

[CGB91] D.R. Cheriton, H.A. Goosen, P.D. Boyle, *Paradigm: A Highly Scalable shared- memory multiprocessor*, IEEE Computer, **24**, No. 2, Feb 1991, 33–46.

[ChV89] H. Cheong, A.V. Veidenbaum, *A version control approach to cache coherence*, Proc. of ACM Intern. Conf. on Supercomputing, ACM, New York, 1989, 322–330.

[ChV91] Y.-C. Chen, A.V. Veidenbaum, *A software coherence scheme with the assistance of directories*, Proc. of ACM Intern. Conf. on Supercomputing, 1991, 284–294.

[CKA91] D. L. Chaiken, J. Kubiatovicz, A. Agarwal, *LimitLESS Directories: A scalable cache coherence scheme*, Proc. of 4th Intern. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV), 1991, 224–234.

[Dor84] M.N. Dorojevets, *The MARS-M control processor*, Theoretical and applied problems in parallel processing, Computing Center of SD of Russian Academy of Sciences, Novosibirsk, 1984, 150–160.

[DoW92] M.N. Dorozhevets, P. Wolcott, *The El'brus-3 and MARS-M: Recent Advances in Russian High-performance Computing*, Proc. of Journal of Supercomputing, **6**, 1992, 5–48.

[DSB86] M. Dubois, C. Scheurich, F. Briggs, *Memory access buffering in multiprocessors*, Proc. of 13th Intern. Sympos. on Computer Architecture, 1986, 434–442.

[DDS94] F. Dahlgren, M. Dubois, P. Stenstrom, *Combined performance gains of simple protocol extensions*, Proc. of 21st Intern. Sympos. on Computer Architecture, 1994, 187–197.

[DDS95] F. Dahlgren, M. Dubois, P. Stenstrom, *Sequential hardware prefetching in shared- memory multiprocessors*, IEEE Transactions on Parallel and Distributed Systems, **6**, No. 7, July 1995, 733–746.

[EgK88] S.J. Eggers, R.H. Katz, *A characterization of sharing in parallel programs and its application to coherency protocol evaluation*, Proc. of Intern. Sympos. on Computer Architecture, 1988, 373–382.

[GGH91] K. Gharachorloo, A. Gupta, J. Hennessy, *Performance evaluation of memory consistency models for shared-memory multiprocessors*, Proc. of Intern. Sympos. on Computer Architecture, 1991, 245–257.

[GGH92] K. Gharachorloo, A. Gupta, J. Hennessy, *Hiding memory latency using dynamic scheduling in shared-memory multiprocessors*, Proc. of Intern. Sympos. on Computer Architecture, 1992, 22–33.

[GGV90] E. Gornish, E. Granston, A. Veidenbaum, *Compiler-directed data prefetching in multiprocessors with memory hierarchies*, Proc. of Intern. Conf. on Supercomputing, 1990, 354–368.

[GHG91] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, W.-D. Weber, *Comparative evaluation of latency reducing and tolerating techniques*, Proc. of Intern. Sympos. on Computer Architecture, 1991, 254–263.

[GLL90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, J. Hennessy, *Memory consistency and event ordering in scalable shared-memory multiprocessors*, Proc. of Intern. Sympos. on Computer Architecture, 1990, 15–28.

[HaF88] R.H. Halstead, Jr., T. Fujita, *MASA: A multithreaded processor architecture for parallel symbolic computing*, Proc. of Intern. Sympos. on Computer Architecture, 1988, 443–451.

[KOH94] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, J. Hennessy, *The Stanford FLASH multiprocessor*, Proc. of 21st Intern. Sympos. on Computer Architecture, 1994, 302–313.

[LYL87] R.L. Lee, P.-C. Yew, D.H. Lawrie, *Data prefetching in shared memory multiprocessors*, Proc. of Intern. Conf. on Parallel Processing, 1987, 28–31.

[LLG90] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, J. Hennessy, *The directory-based cache coherence protocol for the DASH multiprocessor*, Proc. of 17th Intern. Sympos. on Computer Architecture, 1990, 148–159.

[LLJ92] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, J. Hennessy, *The DASH prototype: implementation and performance*, Proc. of 17th Intern. Sympos. on Computer Architecture, 1992, 92–103.

[Lil91] D.J. Lilja, *Processor parallelism considerations and memory latency reduction in shared memory multiprocessors*, Center for Supercomputing Research and Development, Univ. of Illinois, Urbana, 1991, Rep. No. 1136.

[Lil93] D.J. Lilja, *Cache coherence in large-scale shared-memory multiprocessors: issues and comparisons*, ACM Computing Surveys, **25**, No. 3, 1993, 303–338.

[MiB89] S.L. Min, J.-L. Baer, *A timestamp-based cache coherence scheme*, Proc. of Intern. Conf. on Parallel Processing , **I, Architecture**, 1989, 22–32.

[MoG91] T. Mowry, A. Gupta, *Tolerating latency through software-controlled prefetching in shared-memory multiprocessors*, Journal of Parallel and Distributed Computing, **12**, No. 3, 1991, 87–106.

[Pea68] M.C. Pease, *An adaptation of Fast Fourier Transform for parallel processing*, J. ACM, 1968, 252–264.

[RLW94] S. Reinhardt, J. Larus, D. Wood, *Tempest and Typhoon: User-level shared memory*, Proc. of 21st Intern. Sympos. on Computer Architecture, 1994, 325–336.

[Smi78] B.J. Smith, *A pipelined, shared resource MIMD computer*, Proc. of Intern. Conf. on Parallel Processing, 1978, 6–8.

[VGD87] Yu. Vishnevsky, G. Grishin, M. Dorojevets, N. Chertenkov, P. Shedko, *The MARS-M heterogeneous multiprocessor system*, Proc. of 1st USSR Conf. on Problems in Designing and Using Supercomputers and Supersystems, Minsk, **1**, 1987, 30–31.

[WeG89] W.-D. Weber, A. Gupta, *Analysis of cache invalidation patterns in multiprocessors*, Proc. of Intern. Conf. on Architectural Support for Programming Languages and Operating Systems, 1989, 243–256.