Locality-Conscious Load Balancing:

Connectionist Architectural Support*

Chaitanya Tumuluri, Alok N. Choudhary

121 Link Hall, ECE Department

Syracuse University, Syracuse NY 13244

(tumuluri, choudhar)@cat.syr.edu

Phone: (315) 443-4280

Fax: (315) 443-2583

Chilukuri K. Mohan

2-120 CST, School of CIS

Syracuse University, Syracuse NY 13244

ckmohan@syr.edu

Phone: (315) 443-2322

Fax: (315) 443-1122

Abstract

Traditionally, in distributed memory architectures, locality maintenance and load balancing are seen as user level activities involving compiler and runtime system support in software. Such software solutions require an explicit phase of execution, requiring the application to suspend its activities. This paper presents *the first* (to our knowledge) architecture-level scheme for extracting locality **concurrent** with the application execution. An artificial neural network coprocessor is used for dynamically monitoring processor reference streams to learn *temporally emergent* utilities of data elements in ongoing local computations. This facilitates use of kernel-level load balancing schemes thus, easing the user programming burden. The kernel-level scheme migrates data to processor memories evincing higher utilities during load-balancing. The performance of an execution-driven simulation evaluating the proposed coprocessor is presented for three applications. The applications chosen represent the range of load and locality fluxes encountered in parallel programs, with **(a)** static locality and load characteristics, **(b)** slowly varying localities for fixed dataset sizes and **(c)** rapidly fluctuating localities among slowly varying dataset sizes. The performance results indicate the viability and success of the coprocessor in concurrently extracting locality for use in load balancing activities.

*This research was supported in part by an NSF Young Investigator Award CCR-9357840.

Keywords: Concurrent architecural support, coprocessor, locality, load balancing, artificial neural networks.

1 Introduction

The goal of memory management is to optimize the cost of memory operations engendered by applications in order to minimize the application turnaround time. All strategies for accomplishing this task rely on the experimentally supported locality of execution principle [1] which tells us that computations normally go through distinct *phases* of execution characterized by a preponderance of references to a small subset of memory. If the current contents of memory were to be called its *state*, most strategies for *locality maintenance* aim at maximizing the match between the current subset and the state of the fastest memory. In **D**istributed **M**emory (DM) systems, costs of *remote* memory accesses outweigh the costs of accesses to *any* local memory (except disks) in the hierarchy, and therefore, it is more important to satisfy references *locally*. This adds to the complexity of maintaining locality in application execution on DM architectures.

Traditionally, in DM architectures, locality maintenance was seen as a *user level* strategy because user level scheduling policies (e.g. a runtime-system library) would have access to data layout information and data access patterns not available to a kernel (OS) level policy [2]. It was further argued that user level scheduling decisions, dependent on data granularity and frequency of synchronization, were more common than kernel level decisions. This significantly increases the programming complexity on DM architectures. In contrast, our effort is geared towards extracting locality information from processor reference-streams, which could be used by kernel level scheduling policies for maintaining load balanced execution and locality. This has the advantage of reducing the current DM system programming complexity.

This paper presents the architectural support needed for a kernel-level locality-conscious, load-balancing scheme. The architectural support consists of:

1. Memory abstractions defining granularity of load/locality maintenance at the architectural level
2. A scheme for mapping user-level data abstractions into the architecture-level memory abstractions
3. A clustering mechanism to extract locality information among the memory abstractions

For the first time, to our knowledge, results in this paper demonstrate successful extraction of sufficient locality information for the purposes of maintaining locality and load balanced execution via online memory reference stream monitoring. The applicability of the system is broad enough to encompass programs with diverse load and locality characteristics, ranging from programs with static load and locality to programs with dynamically varying load and locality features. Also, as a consequence of implementation at the architectural level, our system runs concurrently with the application and does not require a separate phase of execution (halting the application execution) as in most software based solutions [3, 4].

Section 2 states the problems being attacked and Section 3 presents an overview of the solution to these problems. In Section 4, memory is abstracted as consisting of indivisible **A**tomic **M**emory **U**nits (AMUs) mapped to a **C**omputational **M**etric **S**pace (CMS). The notion of locality is mapped into concepts of locality subspaces of the CMS, containing actively referenced AMUs. Section 5 presents details of the clustering mechanism used. Section 6 presents overviews of the architectural model and the execution-driven simulation used. Section 7 details the application testbed for validating the proposed methodology. Section 8 presents the results of our experiments and Section 9 presents conclusions.

2 Problem Formulation

Computation from an architectural perspective is the sequence of accesses to memory and the corresponding changes to the memory state. *Computational locality* is then a quantitative characterization of the interaction between the memory and the computation. Degradations in locality occur due to *capacity misses* arising from finite memory sizes, *conflict misses* arising from the replacement policies for finite memories and finally, *geographical misses* arising from use of physically distributed memories. Our efforts are focussed on the decision problem of on-the-fly data/task migration for mitigating geographical misses addressing the following three problems:

- *Geographical Misses*: In DM architectures, *concurrent* processes may require access to the same data at the same time. Since data cannot be present simultaneously in geographically separated memories, this leads to a *geographical miss* arising out of residency requirement conflicts. In our study we ignore capacity and conflict misses and focus solely on the problem of maintaining geographical locality, since geographical misses dominate the costs.

- *Online Locality Extraction*: Traditionally, computational locality has been extracted by the user in coding the application using domain knowledge [5, 2], or by a compiler/runtime system which relies on application code analysis [6, 3]. Our approach is the first, to our knowledge, to extract locality information dynamically at the architectural level using computational memory references. We extract the locally *emergent cohesive properties* of references to **A**tomic **M**emory **U**nits (AMUs), which are units of data migration among processor memories (e.g., cache blocks or collections thereof). Cohesiveness is quantified by the **Utility** of these AMUs in the local working set.

- *Programming Complexity*: User-level attempts at maintaining geographical locality extract spatio-temporal correlations among references to data using a-priori domain knowledge and careful analysis of user-defined data abstractions such as arrays or aggregate datatypes. The correlation information enables programs to layout data in memory for minimizing geographical misses. The granularity of locality maintenance in memory *influences* granularities of user data abstractions, implicitly defining granularities of user level schemes. Main memory paging policies and cache line replacement policies are examples of locality maintenance policies in memory. Thus, programming complexity on DM machines is significantly increased since the user has to:

    1. Factor **multiple** architectural granularities into the design of data abstractions.
    2. Factor memory replacement policies into *construction* of programs for maintaining *geographical* locality.
    3. Explicitly or implicitly consider issues of data layout in DM architectures.

3 Problem Solution: Overview

Our system assumes that concurrent tasks have a specific code image used to update a specific instance of a user data abstraction. Traditionally, geographical misses in computing the update are commonly mitigated using software caching (with loop reordering for e.g.) and data replication at the *user-level* [7, 8], and/or using a runtime system requiring an *explicit phase* of *global* data (re)distribution as in [3, 9, 4]. Our solution to ease programming complexity *automates* geographical locality maintenance at the architectural level using the following:

1. An atomic granularity of data allocation and migration (AMU) and a scheme for mapping user data abstractions into AMUs

2. An architectural level framework called the **C**omputational **M**etric **S**pace (CMS), within which spatial and temporal correlations of references to AMUs are captured.

3. A hardware-realizable clustering scheme operating at processor speeds for:

   - Mapping local references into AMUs in the CMS, and
   - Extracting spatio-temporal correlations among AMUs.

4. A kernel-level scheme for migrating AMU for maintaining geographical locality and computational load balance.

4 Memory Model

This section provides details of the memory model assumed in our system. First the notion of Atomic Memory Units is developed. This is followed by details of AMU descriptors used to track emergent computational characteristics. The tracking process occurs in the Computational Metric Space, details of which are provided in Section 4.3. Finally, the mapping of AMUs into the CMS is presented in Section 4.4.

4.1 AMU Formulation

We abstract memory as comprised of indivisible and contiguous **A**tomic **M**emory **U**nits (AMUs, e.g., cache lines or pages) which is also the atomic unit of automatic data migration in the system. Our system assumes a global **V**irtual **A**ddress **S**pace (VAS) in which each AMU and its constituent words have a unique system-wide address. Further, each *instance* of a user data abstraction is inscribed and aligned into one or more contiguous AMUs. One or more AMUs encapsulating each instance of a user data abstraction is termed **D**ata-abstraction **I**nstance (DI). Wordwise processor references are *parsed* into the AMU addresses containing the word.

These concepts are illustrated in Figure 1 showing the mapping of three different user data abstractions into their DIs in the local modules of the distributed memory. The figure illustrates a **P** processor system with a VAS of 2^N words of memory which has been broken up into the constituent **5 word AMU**s. Each square in the local memory grids corresponds to exactly one AMU. One instance of *MyStruct* maps exactly into the DI labelled AMU1 and one instance of *MyStruct1* into the DI labeled AMU2 leaves two words

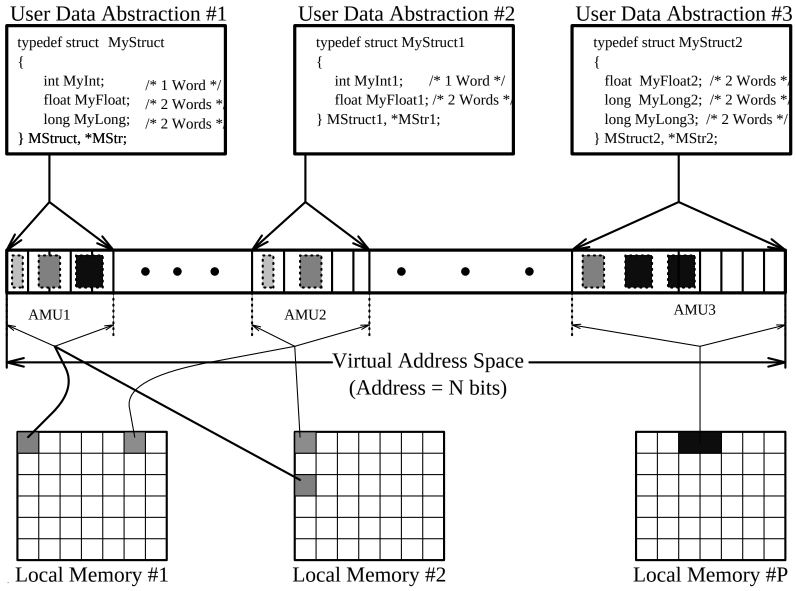

Figure 1: Memory Model: AMU Details; *Int*eger, *float*ing point number and *long* integer mapping into the words in the DIs are shaded differently

unused. Both AMU1 and AMU2 map into one AMU while the mapping of *MyStruct2* into AMU3 requires *two* contiguous AMUs.

4.2 AMU Descriptors in Memory

Each AMU is characterized by five descriptors as illustrated in Figure 2.

1. Individual AMUs are distinguished by their addresses constituting the *AMU Tag* descriptor.

2. The second descriptor, *AMU-DI Type* distinguishes references to DIs in the processor reference stream. We assume language, compiler and system support, allowing users to tag data-abstractions (such as *MyStruct* in Figure 1). Also, DIs of different user data abstractions have distinct *AMU-DI Type* values.

3. It is necessary to track ownership of each DI, using the *AMU Owner* descriptor, to effect task transfers for maintaining geographical locality and computational load balance.

4. The geographical locality of individual DIs, quantified by a measure called *Utility*, is recorded in the *AMU Utility* descriptor.

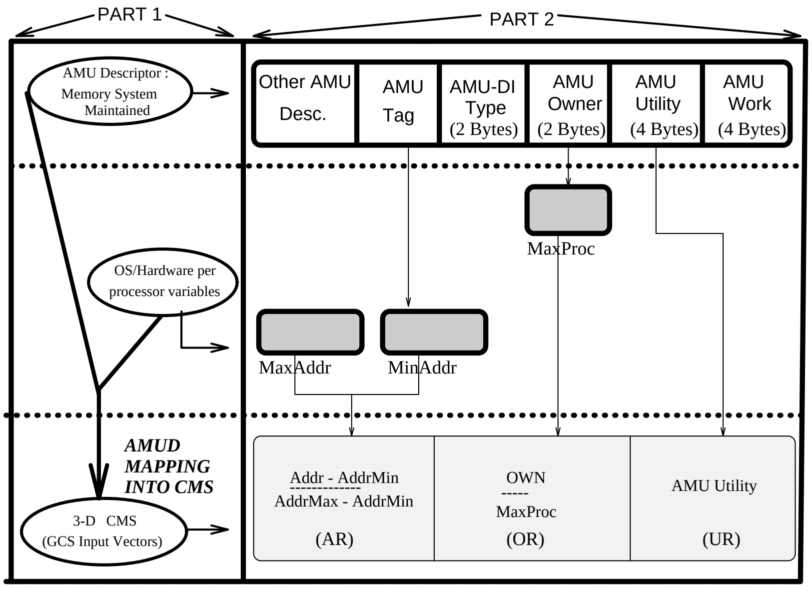

Figure 2: Memory Model: AMU Descriptors and Mapping to CMS

5. The *AMU Work* descriptor records the workload associated with an AMU. This is measured by the number of *clock cycles* required to execute the associated code-image.

The total descriptor storage overhead is 12 bytes per AMU, assuming that the *AMU-DI Type*, *AMU Owner*, *AMU Utility* and *AMU Work* descriptors require 2 bytes (2^{16} different data abstractions), 2 bytes (2^{16} Processors), 4 bytes (floating point number) and 4 bytes (floating point number) respectively. The overhead is reasonable compared with the descriptor field sizes in modern DSM multiprocessors (and cache based memory systems in general) such as KSR1 [10], FLASH [11] and DASH [12] architectures, the overhead is reasonable. In the KSR1, for example, if the AMU were to be a *subpage* of size 128 bytes, 12 bytes represents less than a 10% overhead.

4.3 CMS Formulation

The CMS is a 3-dimensional space whose axes are the *AMU Tag*, *AMU Utility* and *AMU Owner* values, ensuring unique mappings of each AMU into this space. The space is illustrated in Figure 3 showing three Processor Planes (PP), each containing DIs owned by three different processors. The two-dimensional shaded spatial envelopes, encompassing the DIs within each *PP*, are referred to as *locality subspaces*. The inset in the right hand corner of Figure 3 shows the *Combined PP* of the local processor. Future references to *locality subspaces* will mean the locality subspaces in the local Combined PP, unless otherwise specified.

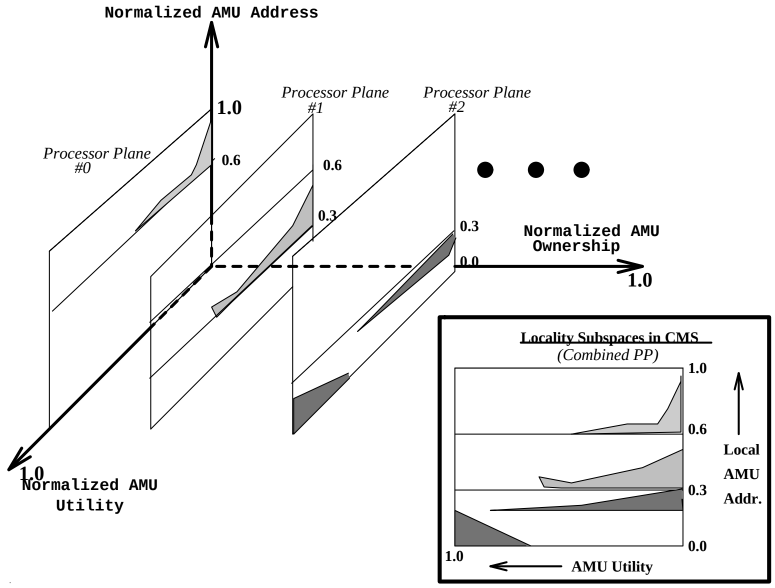

Figure 3: Memory Model: CMS Details

The locality subspaces capture the current working set of the local processor. The dimensional ranges of all axes in Figure 3 are normalized to lie in the range $[0, 1]$.

4.4 Mapping AMUs into the CMS

Apart from the AMU descriptors, three other variables are maintained for mapping the AMUs into the CMS, as illustrated in Figure 2. These are the *MinAddr*, *MaxAddr* and the *MaxProc* variables, maintained for for each processor. If the addresses generated by a processor are taken to be long integers, then the lowest address generated to date is stored in the variable *MinAddr*. The *MaxAddr* long integer variable similarly stores the highest address generated by the processor. Finally, the *MaxProc* variable is simply the total number of processors in the system. These three variables are used to effect the normalizations mentioned in Section 4.3 as part of the mapping.

The actual mapping of the AMUs to their corresponding coordinates in the CMS is illustrated in the last row of Part 2 of Figure 2. The coordinates constitute 3-dimensional input vectors to be fed into the STGCS neural network (discussed in Section 5), for locality subspace (re)formulations and *AMU Utility* computations. We assume existence of special hardware to generate the 3-d input vectors according to this mapping scheme. Finally, we postulate a special store (**Desc_STORE**) instruction to update the *AMU Utility* descriptor in local memory with values from the STGCS clustering mechanism.

5 Problem Solution: STGCS Network

AMUs map uniquely into the CMS producing a CMS **S**patial (CMSS) distribution consisting of locality subspaces. A blowup of the locality subspaces, in Figure 5, reveals clusters of AMUs labeled *C1* through *C9*. Changes in *AMU Utility*, *AMU Work* and *AMU Owner* values, however, alter the AMU mapping into the CMS leading to dynamically emergent cluster size, composition, number and location. The CMSS distributions at different time instants are thus *emergent characterizations* of the ongoing computation. We use an Unsupervised **A**rtificial **N**eural **N**etworks (ANN) called **G**rowing **C**ell **S**tructures (GCS) [13], to form spatial maps of the emergent CMSS distributions, thus tracking emergent computational locality. The architectural adaptivity (i.e. shrinkage and growth in size) of the network was an advantage in dealing with the time-varying CMSS distributions. Since GCS does not explicitly deal with time-varying distributions, we enhanced the GCS ANN to create the **S**patio-**T**emporal GCS (STGCS) network for this purpose.

5.1 STGCS Architecture and Dynamics

We always use a 2-dimensional network (forming planar maps of the CMSS distribution) and the initial network topology is a 2-dimensional triangle. The basic network topology labeled [**A**], the topology labeled [**B**] after addition of neuron *N6* and the topology labeled [**C**], after the deletion of neuron *N2* are illustrated in Figure 4. Three-dimensional input vectors are fed to all neurons simultaneously via weighted input lines labeled W^I in sub-figure labeled [**D**]. For further details, refer to [13]. Among other extensions to create the STGCS network, every neuron is connected to every other neuron via weighted, directed lateral connections (W in sub-figure [**E**]), henceforth denoted by W_{ij}, indicating a connection from neuron j to neuron i. Each neuron has an associated **S**ignal **R**eceptivity **M**eter (SRM) recording the relative (to other neurons) frequency of hits to this neuron (sub-figure [**E**]). Also, every neuron is endowed with memory in the form of a FIFO buffer storing the last **5** remote DI input vectors which mapped to it. This is used in periodically assessing which remote DIs should be transferred to the local processor's ownership for maintaining locality-conscious computational load balance.

Input vectors to the network are the 3-dimensional coordinates of the AMU locations in the CMS. Each input vector is fed simultaneously to all the neurons via weighted input lines (W^I)and *quantized* (i.e. mapped) into that neuron whose input weights best match it. The STGCS network operates in two

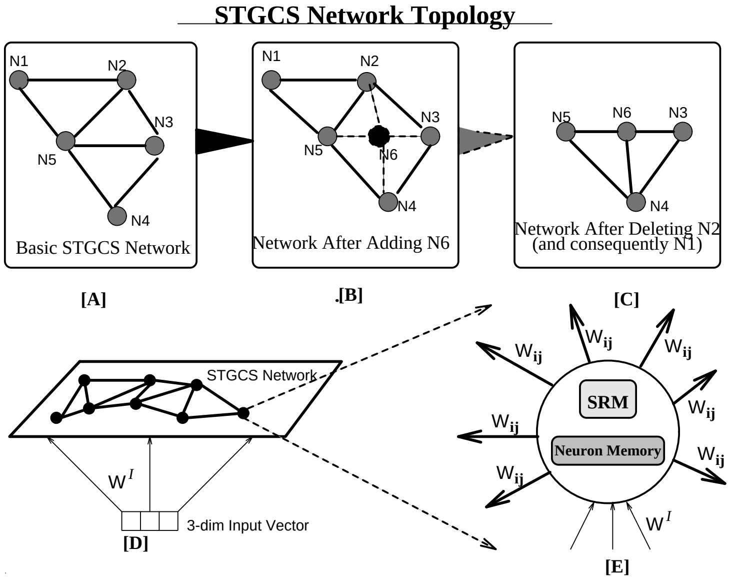

Figure 4: STGCS Topology and Neuron Internal Details

modes viz., a training mode and a production mode. In the training phase, the network maps the CMSS distribution from a set of training samples. One quarter of them are extracted from the AMUs in the reference stream, while the remaining are constructed by randomly sampling AMUs in local memory. The training phase positions neurons as best as possible at cluster centers (i.e. match W^I with cluster center coordinates in Figure 5) so the network topology forms a dimensionally reduced (i.e. 2-d) map of the 3-d CMSS distribution.

In the production mode, the W^I adjustments track the dynamically emergent CMSS distributions. The W_{ij}, implemented as counters, represent the number of times that a hit occurred to neuron (i.e. cluster or locality subspace) j followed by a hit to neuron i, during the production phase. Such a predecessor neuron j and successor neuron i are said to be *mutually correlated* and, the W_{ij} *continuously* quantify the frequency of occurrence of the successor-predecessor relationship. In production use, the network tracks the emergent cluster correlations via the W_{ij} and these values are used in computing the **total probability of mutual correlation of a neuron with respect to all other neurons in the network**. The total probability of a neuron is the *AMU Utility* value for all AMUs mapped to it.

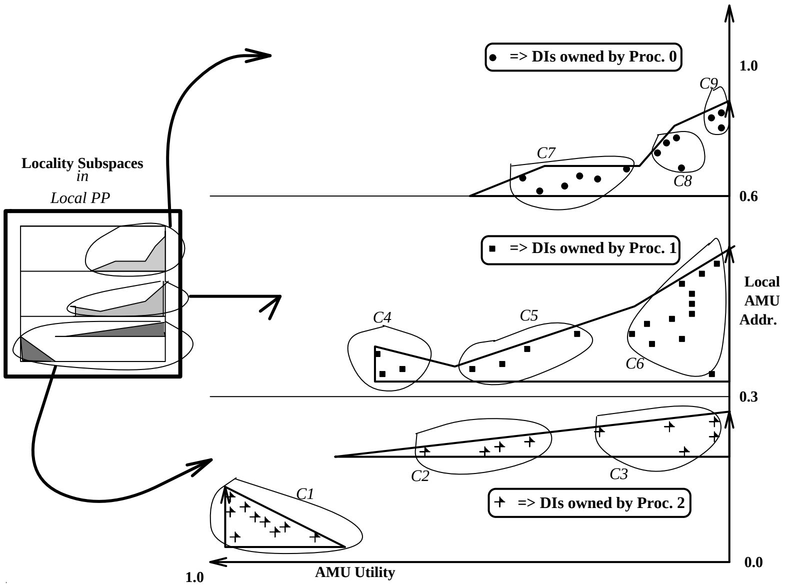

Figure 5: Memory Model: CMS Locality Subspaces and DI Clusters

6 Architectural Support: Model and Simulation

This section is divided into two parts. Section 6.1 provides details of the architectural implementation model that was assumed in this study. It assumes a generic local processor and its local memory in a DSM multiprocessor architecture and describes a possible implementation of our system. Section 6.2 presents details of an execution driven simulation implemented to validate the model.

6.1 Architectural Level Model

We postulate the use of an on-chip STGCS coprocessor unit per processing node in the system as illustrated in Figure 6. The coprocessor is a functional unit executing concurrently with the main processor, which could be implemented using synaptic parallelism with $\Sigma\Pi\Sigma$ pipelined functional units as in [14]. The coprocessor functionality is briefly described with respect to Figure 6 as follows.

1. **Training Phase Dynamics**: The following steps are repeated until the network forms a reasonably stable map of the CMSS distribution from the training vector set. At that time, the network enters the production phase.

    - Transfer last quarter of vectors in the STGCS Input (FIFO) Buffer into the STGCS Training Buffer

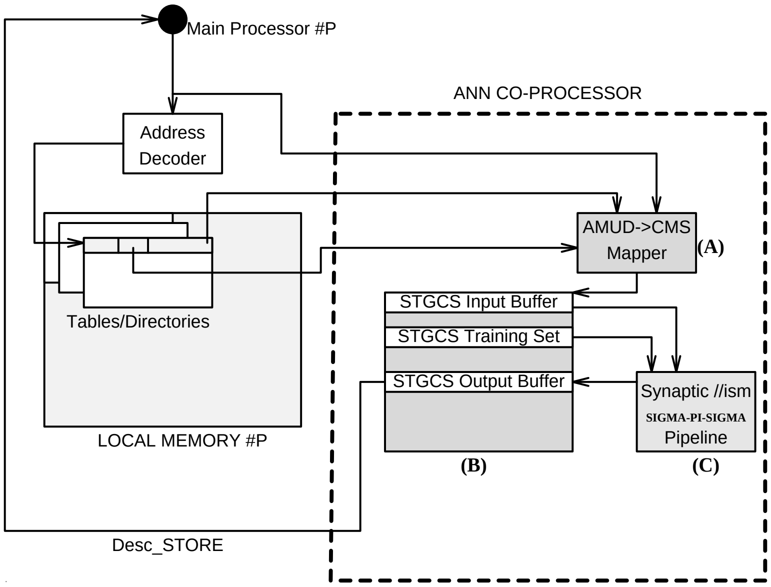

Figure 6: Architectural Support: Coprocessor Overview, **(A)** = AMUD-CMS Mapping Logic, **(B)** = CoProcessor Buffers, **(C)** = STGCS Processor Unit

- Randomly sample local memory Page Table/Directory for AMU descriptors until remaining three-quarters of the STGCS Training Buffer is filled. Descriptor are mapped by functional unit labeled (A) in Figure 6. This requires T_{TVect} cycles.
- Pipeline inputs from the STGCS Training Buffer into the STGCS network for quantization and Input Weight adaptations which requires T_{Quant} cycles.
- Add a neuron if required. This needs T_{Add} cycles.
- Delete a neuron if required. This requires T_{Del} cycles.

2. **Production Phase Dynamics**: Repeat the following steps until the network map of the emergent CMSS distribution is detected to be poor. At that time the network reverts back to the training phase.

   - Map the latest main processor AMU reference into the CMS. Store vector in the FIFO STGCS Input Buffer. This requires T_{SVect} cycles.
   - Feed the oldest vector from the STGCS Input Buffer into the network, for quantization and Input Weight adaptation. This requires T_{Quant} cycles.

STGCS Activities	Co-Processor Execution Speed
Training Vector Generation (T_{TVect})	7 KSR1 cycles
Snoop Vector Generation (T_{SVect})	62 KSR1 cycles
Input Vector Quantization (T_{Quant})	10 KSR1 cycles
Lateral Weights Update (T_{IM})	34 KSR1 cycles
Neuron Addition (T_{Add})	100 KSR1 cycles
Neuron Deletion (T_{Del})	200 KSR1 cycles

Table 1: STGCS Simulation Timing Parameters

- Adapt lateral weight between neurons to which the current and previous AMU vectors were mapped and generate new *AMU Utility* value for current AMU. Issue a *Desc_STORE* instruction on this value. This requires T_{IM} cycles.
- Delete a neuron if required which requires T_{Del} cycles.

The *Desc_STORE* instruction could be incorporated as part of the external I/O instruction set and such instructions could be inserted into the instruction stream of the main processor. Further, the timings for each step above is given in Table 1, and have been derived from simulations on the KSR1 and results in [15, 10, 14].

6.2 Simulation Details

An execution driven simulation was implemented in order to validate the ideas presented. The AMU descriptors on each processor were necessarily implemented as part of the simulation software because the system tables couldn't be accessed. The descriptors were stored in a hashed bucket table with 1024 buckets that expanded dynamically and was indexed by the last 10 bits of the AMU address of DIs. Each application thread had a local hash table maintained by the corresponding part of the STGCS simulation.

6.2.1 Online Locality Extraction

The problem of online locality extraction warranted an implementation simulating concurrent execution of the STGCS coprocessor and the main application processor. It was decided to spawn an STGCS thread for each application thread, and dedicate a processor for its concurrent execution with the application thread. Thus the simulations required twice as many processors as the original application run. Each application thread's references to AMUs were buffered along with the inter-reference times and communicated to the corresponding STGCS thread, which regulated its actions as per Table 1 and the inter-reference times.

6.2.2 Task Migration Rule

STGCS threads periodically check load-levels of their application threads and the remote DIs referenced in the recent past captured in the STGCS neuron memory. The load-levels are compared locally and at the current owner of each DI. If the local load is less and the local *AMU Utility* is greater, the DI ownership is acquired by the local thread and its *AMU Owner* descriptor updated. Thus, task transfers try to balance *emergent* load and locality gradients in the system via a task *PULL* mechanism.

In order to perform load and *AMU Utility* comparisons, each STGCS thread needs access to the hash tables of other STGCS threads. The need for such global access to every local hash table prompted the choice of a shared memory system for implementing the simulation. It could have been implemented on a message passing architecture, but remote accesses to hash tables is far more difficult to implement using the unique local address spaces in message passing systems. The global address space semantics inherent in shared memory systems facilitated the implementation better and hence the KSR1 [10], was chosen as our simulation platform.

7 Application Testbeds

This section discusses the application testbed chosen for evaluating our proposed system. Three applications were used: **(1)** the Barnes-Hut algorithm for a Monte-Carlo N-body simulation of interacting galaxies, **(2)** the Unstructured Mesh kernel for an Eulerian solver, and **(3)** "WaTor", an ecological simulation of sharks and minnows in an ocean. The three applications represent the range of execution locality and load characteristics normally encountered in parallel programs, ranging from irregular and static (Eulerian Solver) to dynamic and adaptive (Barnes-Hut and WaTor).

7.1 Programming Model Assumptions

We assume an *SPMD* programming model. In all applications we use, there is an explicit time-stepping loop within which each DI is updated. The same code-image (code text within the loop) is used to update each DI so that *tasks* within our system corresponded to individual DIs. Therefore, balancing the computational load is equivalent to equalizing the time spent in traversing the loop for all the DIs in local partitions. Task migration corresponds to a relocation of the site where the DI update occurs since the code-image used for updates is the same at all sites. The site or processor where the DI is scheduled to be updated is considered to **own** the DI.

7.2 Eulerian Solver: Unstructured Mesh

The first application in the testbed is an unstructured Eulerian solver, which simulates the airflow dynamics over an airfoil [3, 16]. The airfoil is discretized and represented as an unstructured mesh and the code structure is simply a loop over all the mesh edges for updating the vertices of the mesh. The mesh is static leading to fixed locality characteristics. The application is loosely synchronous, irregular, static, and uses the initial partitioning of data for all iterations of the main computational loop. The base run partitioning was a random but equal allocation of the mesh edges and vertices among the threads. The initial edge distribution for the corresponding STGCS co-processor run was skewed. The distribution was an arithmetic progression of the form $a + i \times b$ where a is the initial value, i is the processor/thread-id and b is the increment in the progression such that $a = b = (\#OfEdges/\sum_{i=0}^{nproc} i)$, where $nproc$ is the number of processors in the run. The distribution of vertices was equal but random among the threads. The dataset for each of the 4, 8, 16 and 24 processor runs consisted of 2800 vertices and 17,377 edges. The simulation was run for 50 iterations of the time-stepping loop.

The individual thread execution times served as the measure of the locality-conscious computational load balance achieved. We also explicitly measure the locality achieved and maintained by the our system. Firstly edges in each local partition were periodically noted. Then, in traversing these edges, the number of times (N_r) that vertices were encountered was recorded. Thirdly, we record the sum (N_v) of the number of edges emanating from all vertices encountered during the edge list traversal. Then, the ratio N_r/N_v served as a metric for partitional contiguity, measuring the quality of the computational locality achieved by our system.

7.3 N-Body Simulation: Barnes-Hut Algorithm

The application performs an N-body simulation of stars in interacting galaxies, using a hierarchical algorithm. The SPLASH version of this application [5] was adapted and ported to the KSR1. This application is loosely synchronous, irregular and adaptive. It has two distinct *activity* phases within the main computational loop called the TrackTime loop, viz., a Load/Data Partitioning phase and a Computational phase.

NPROC	4	8	16	24
NBODY n	4096	8192	16384	16384
THETA θ	0.9	0.9	0.9	0.9
TIME $\triangle t$	0.025	0.025	0.025	0.025

Table 2: **N-Body Simulation Sizes and Parameters**

The scheme used to partition the bodies among the various threads is the **CostZones** scheme outlined in [5]. The physical application domain is highly non-uniform and manifests irregular, dynamically changing inter-particle interactions and hence memory access patterns. The dataset size (number of bodies) is static. Further details and references for this application may be found in [5, 17, 18].

The individual thread execution times in each run were used to the measure of the locality-conscious computational load balance achieved. The base run using the *CostZones* scheme was compared with a run utilizing the STGCS coprocessor simulation. As suggested in [5], the parameters for the runs are presented in Table 2. As a test of the load balancing abilities of our system, the runs with the STGCS co-processor always started with a skewed initial load distribution as in the Eulerian Solver application.

7.4 Ecological Simulation: WaTor

The third and final application in the testbed is an ecological simulation called WaTor [19, 20]. The ecological domain is a toroidal (regular, rectangular mesh with periodic boundaries) world of water inhabited by sharks and minnows (collectively referred to as fishes). The main data structure in the program is a single linked list of fishes, each element of which is either a shark or a minnow. The rules for fish breeding operations are the same as in [19]. However, fish movements are a little more sophisticated: sharks and minnows are given directional preferences for movement based on the conditions at their neighboring (to a prespecified distance) grid points.

The *near-neighbor* interactions among fishes due to discrete fish movements occasion far more dramatic locality changes than the continuous field dynamics of interaction and continuous body movements in the Barnes-Hut simulation. Additionally, locality fluxes also arise due to varying population sizes, facilitated greatly by our goal-directed fish movement dynamics. WaTor is loosely synchronous, irregular, adaptive and dynamically produces variations in the dataset sizes. Similar execution characteristics are found in the *Radiosity* code [21], where researchers found it difficult to implement locality-conscious load balancing due to runtime increases in dataset sizes. Dynamic task stealing between per-processor task queues was suggested [21] for load balancing. In the base WaTor simulation, used for performance comparisons with our scheme, a single central task queue was used to provide the same effect. Both the initial shark and minnow distributions for the STGCS co-processor run were skewed using the same arithmetic progression described in Section 7.2. The simulation was run for 15 timesteps.

As before, individual thread execution times measure the locality-conscious load balance achieved in the runs. An explicit measure of the locality achieved and maintained is taken to be the sum of the Euclidean distances separating successive fishes in the local partition in each iteration. The WaTor simulation parameters included a 200×75 ocean grid with 5000 sharks and 1563 minnows. The minnow and shark breeding ages were 3 and 10 iterations respectively and the shark starvation age was set at 3 iterations. Finally the radius of interaction for determining the movement preferences was set at 5 grid points.

8 Results

The primary goal of the simulations was to evaluate how well the proposed system was able to achieve computational load balance in the face of an initial skewed load distribution. In achieving load balance, the computational locality maintained or achieved was another performance measure examined. The three applications chosen were representative of the range of locality and load balance fluxes normally encountered in parallel programs. The unstructured mesh simulation exemplifies applications with static locality characteristics, the N-body simulation typifies slowly changing localities for fixed dataset sizes while WaTor produced rapidly fluctuating localities among slowly varying dataset sizes. First, Section 8.1 presents the application execution performances. Subsequently, the performance of the STGCS network is described in Section 8.2.

8.1 Application Performance

The performance of each application is measured in terms of execution times and application-specific locality measures outlined in Section 7.

8.1.1 Unstructured Mesh Performances

The bar chart in Figure 7 presents the individual thread execution times for an 8 processor Unstructured Mesh run using the STGCS coprocessor scheme. The initial skewed distribution of processing load among

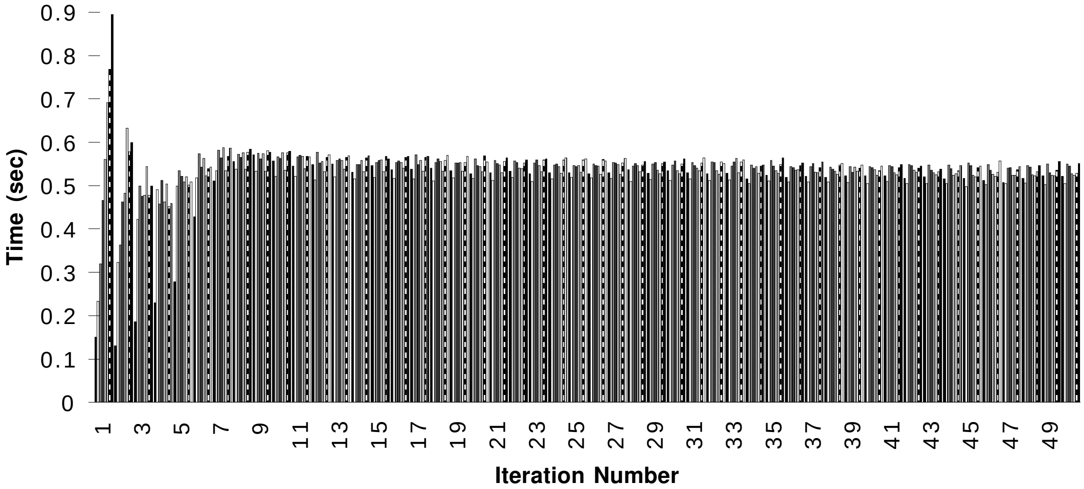

Figure 7: Unstructured Mesh 8 Processor STGCS Run :: Thread Execution Timings

	Initial N_r/N_v (1st Iteration)	Final N_r/N_v (50th Iteration)
4 Processors	0.162570	0.369557
8 Processors	0.158141	0.242234
16 Processors	0.179298	0.198347
24 Processors	0.150277	0.171877

Table 3: Unstructured Mesh Locality Measurements for STGCS

the application threads results in the step-like thread execution timings at the end of the first iteration. Computational load balance is achieved around the tenth iteration. These observations also hold true for simulations with 4, 16 and 24 processors. Note that there is no explicit separate load balancing or data partitioning phase, unlike that required in a software solution.

The system demonstrates successively better locality characteristics within local load partitions using the STGCS coprocessor scheme. These results are presented in Table 3. As the number of processors used in the simulation is increased, the partitional contiguity of each thread's load partition undergoes a slight degradation. The reason is that the load gradient produced by the arithmetic progression is less pronounced as the number of processors is increased, for a fixed dataset size. Consequently, fewer task migrations are required to correct the initial load gradient, leading to fewer chances to improve geographical locality. However, the locality measure (N_r/N_v) of the system always shows an improvement over the initial values.

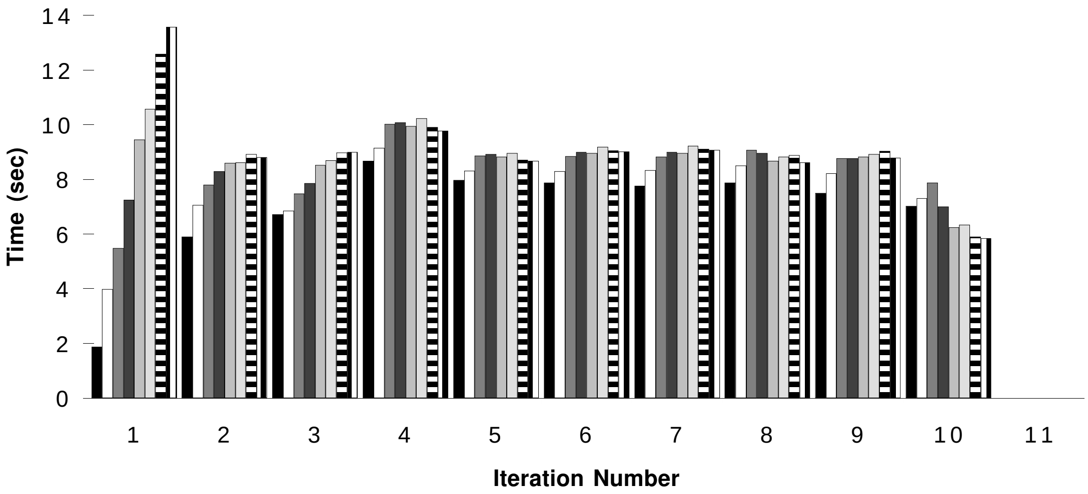

Figure 8: Barnes Hut 8 Processor STGCS Run :: Thread Execution Timings

8.1.2 Barnes Hut Performances

The bar chart in Figure 8 and Figure 9 presents the individual thread execution times for an 8 and a 32 processor run, using the STGCS coprocessor scheme for achieving locality and load balance, again beginning with an initial skewed distribution of processing load among the application threads. Computational load balance is within the first three iterations. Subsequently, the load balance achieved in the run is maintained very well over the entire run. These observations hold true for 16 and 24 processor runs as well and are not presented due to space limitations.

Once load balance has been reasonably achieved, subsequent task transfers (if any) aim at enhancing locality while maintaining load balance. This can be seen in the dip in the thread execution timings at the tenth iteration. Also, the slowly varying locality is captured well by the STGCS mechanism resulting in the improved execution timings presented in Table 4. One reason for the large discrepancies in the execution timings was the load imbalances inherent in the Costzones scheme. This is illustrated in Figure 10 charting the individual thread execution timings for an 8 processor Costzone run, wherein one thread (thread #8) takes almost twice as much time in completing its computations as other threads.

Note that our goal here is to use a known scheme to compare and evaluate the performance of our proposed architectural support system. We consider the results a success even if the performance is

Run Size	STGCS Run Timing (sec)	Costzone Run Timing (sec)
8 Processors	81.531049	129.470271
16 Processors	98.942039	104.776103
24 Processors	65.029432	106.128676
32 Processors	60.083558	78.317244

Table 4: Barnes Hut Execution Timings (STGCS and Costzones)

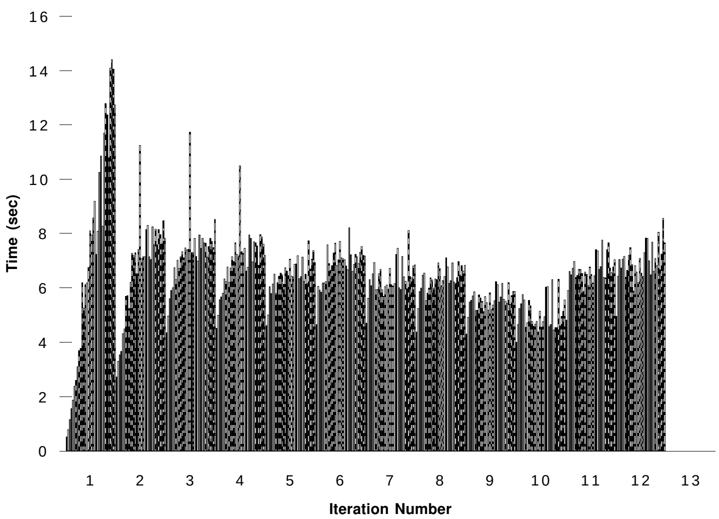

Figure 9: Barnes Hut 32 Processor STGCS Run :: Thread Execution Timings

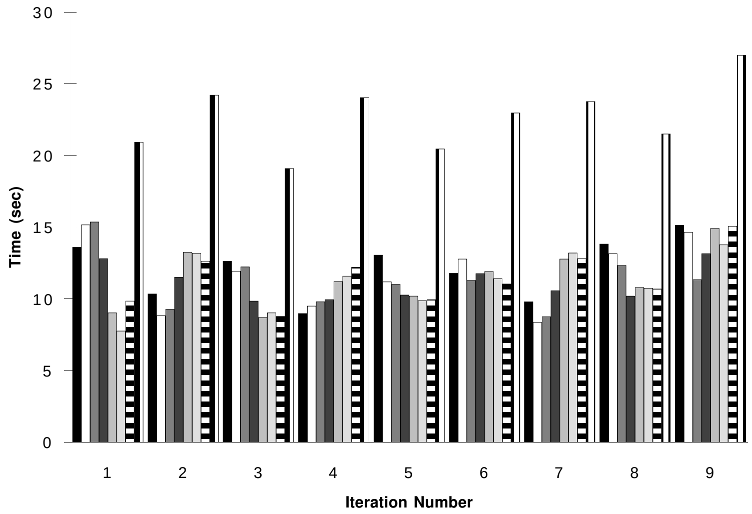

Figure 10: Barnes Hut 8 Processor CostZones Run :: Thread Execution Timings

comparable with established software schemes, such as CostZones, which require an explicit separate load balancing phase. Our focus is not on the evaluation and critique of the established schemes themselves.

8.1.3 WaTor Performances

Figure 11 and Figure 12 show the individual thread execution timings for an 8 and 24 processor WaTor run using the STGCS coprocessor mechanism. The execution timings shown are only for the Minnow update loop, as this was found to be responsible for 75% to 90% of the total execution time. This is evident from Table 5 presenting the execution timings for runs using the STGCS coprocessor mechanism and the **C**entral task **Q**ueue (CQ) load balancing mechanism. The Shark Update loop timings are therefore not described here. The execution timings increase as the system evolves, due to growth in the dataset sizes; load balance is maintained in spite of this phenomenon.

It is evident from Table 5 that the STGCS coprocessor runs scale well in performance. This may be because the central task queue acts as a bottle-neck and degrades the performance. However, the locality characteristics of the runs presented in Figure 13 and Figure 13 illustrate that the STGCS coprocessor runs achieve increasingly better execution locality. This was not true in the CQ runs, wherein the euclidean distances between successive fishes updated stayed consistently at about 75 (see Figure 14). However, in each of the STGCS runs, the inter-task distances decrease from an initial high of 75 to level off at about 40

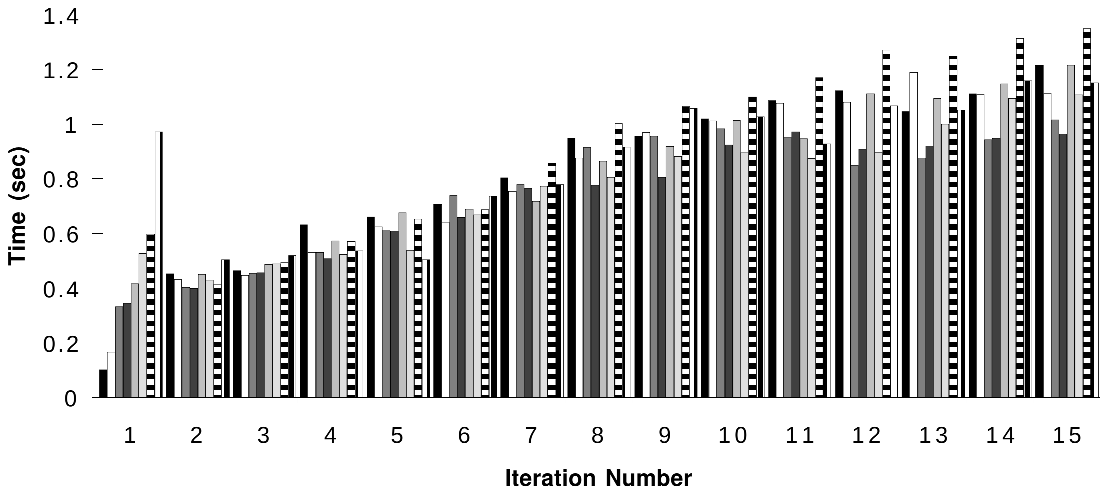

Figure 11: WaTor 8 Processor STGCS Run :: Thread Execution Timings

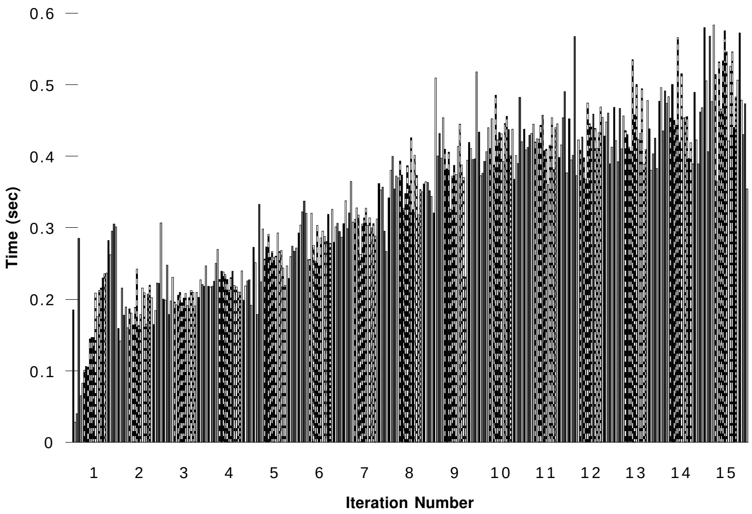

Figure 12: WaTor 24 Processor STGCS Run :: Thread Execution Timings

Run Size	STGCS (Minnow Loop)	CQ (Minnow Loop)	STGCS (Shark Loop)	CQ (Shark Loop)
4 Processors	23.695952 sec	25.314861 sec	4.679540 sec	4.126487 sec
8 Processors	11.132383 sec	13.082895 sec	1.948729 sec	2.173711 sec
16 Processors	6.146037 sec	9.000953 sec	1.166132 sec	1.528530 sec
24 Processors	4.446091 sec	8.155250 sec	0.874231 sec	1.430807 sec

Table 5: WaTor Execution Timings (STGCS and **C**entral Task **Q**ueue (CQ) Runs)

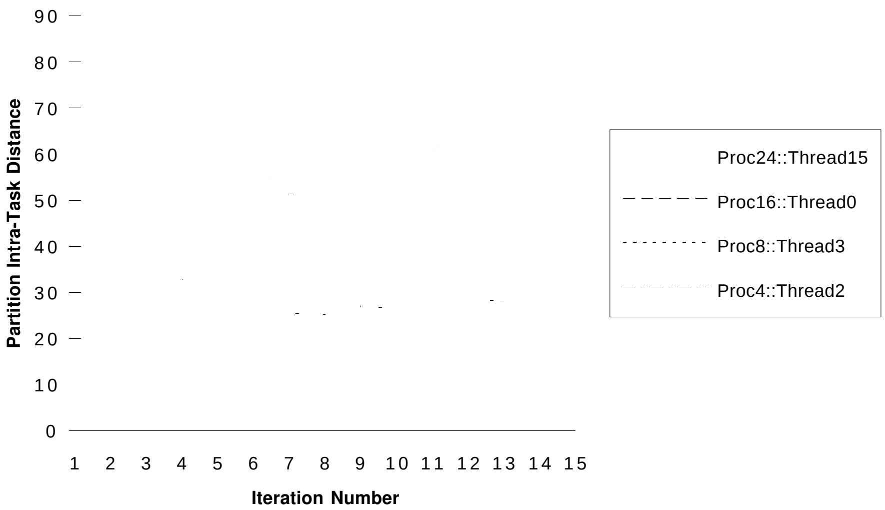

Figure 13: WaTor STGCS runs Locality :: Thread Intra-Task Distances
Proc$x$::**Thread**y $\Rightarrow$ Locality Characteristics of Thread#y in an x processor run

as in Figure 13. This indicates that the local partitions in the STGCS runs achieved increasing contiguity and compactness *in spite of the increasing dataset sizes.* Hence, we conclude that locality maintenance does play a role in the better performance beyond what could be expected due to the the central task queue bottleneck in the CQ runs.

8.2 STGCS Network Performance

In this section, we examine the network sizes evolved and the sampling efficiency of the STGCS networks in the three applications.

8.2.1 STGCS Network Sizes

STGCS networks of widely varying sizes were evolved, underscoring the need for a neural network whose architecture can be adapted automatically during the execution depending on the application. For example, the network sizes varied from 50 neurons (Thread 5 in the 8 processor run) to 8 neurons (Thread 8 in the 16

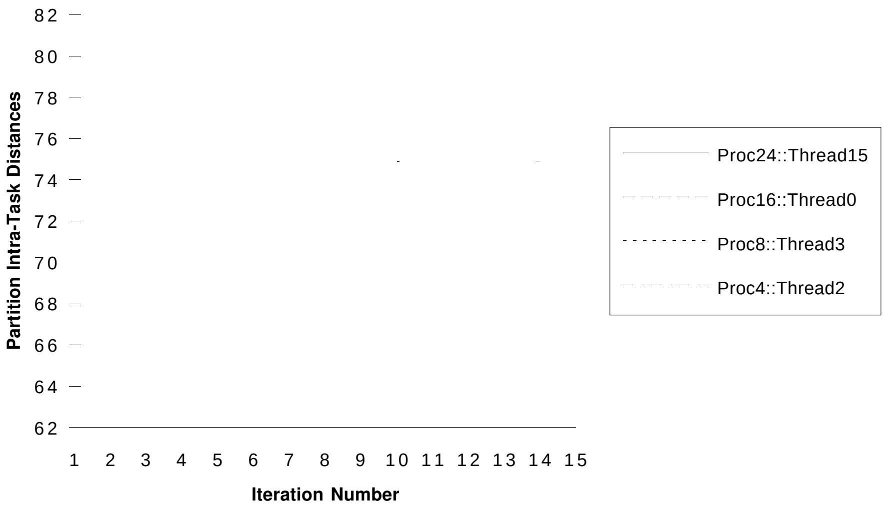

Figure 14: WaTor CQ runs Locality :: Thread Intra-Task Distances
Proc$x$::**Thread**y $\Rightarrow$ Locality Characteristics of Thread#y in an x processor run

processor run) in the Unstructured Mesh application, illustrated in Figure 15. But, both the Barnes-Hut and the WaTor runs show great variations in network sizes ranging between 200 and 45 neurons (thread 16 in 32 processor Barnes Hut run), as seen in Figure 16 and Figure 17.

In this sense, the network sizes and variations in sizes provide a good indication of the locality fluxes inherent in the applications:

- Static locality characteristics and static dataset sizes in the Unstructured Mesh application produced little, if any, fluctuations in the corresponding STGCS network sizes.
- Adiabatically varying locality characteristics with static dataset sizes in the Barnes Hut application, needed large networks to map the initial CMSS distribution. Subsequent adiabatic changes in the locality subspaces are tracked by the online weight adaptations during production use and the network size shrinks to map only the active subspaces.
- Dramatic locality fluxes with adiabatically varying dataset sizes in the WaTor application produces network sizes which vary far more than in either of the other two applications. This is clearly in evidence in the multiple peaks in the 24 processor and 16 processor WaTor runs as opposed to the single peak in the Barnes Hut runs.

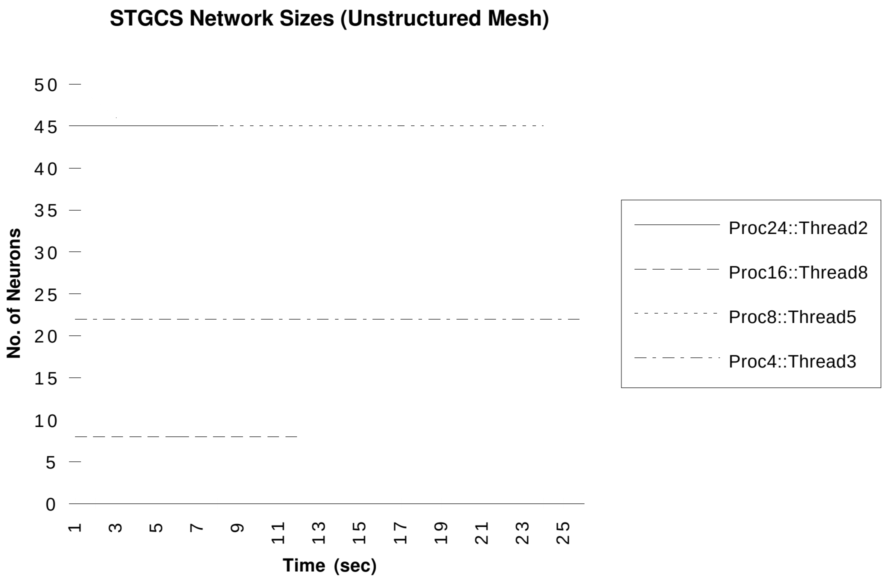

Figure 15: Unstructured Mesh :: STGCS Network Sizes
Proc$x$::**Thread**y $\Rightarrow$ Locality Characteristics of Thread#y in an x processor run

These network sizes (of the order of 200 neurons) are small by ANN hardware implementation capabilities (of the order of thousands of neurons) and hence are a feasible technology.

8.2.2 STGCS Sampling Efficiency

During the STGCS training phases interspersed between the production phases, references from the co-executing application processor are disregarded. For evaluating the ability of the STGCS hardware to track the ongoing computation, the sampling efficiency of the STGCS network measured by the ratio

$$SampEff = \frac{\#Refs.Processed}{TotalProcessorRefs.}$$

is computed every 0.3 seconds of simulation clock time.

The sampling efficiency is consistently high in all three applications, ranging between **90%** to **100%**. This indicates that the training phases, during which the references from the processor were not quantized, are of relatively short durations. This is seen in Figure 18 for example. Due to constraints on space, the WaTor and Unstructured Mesh STGCS sampling efficiency curves are not presented. The graph for the WaTor simulation is similar to the Barnes-Hut simulation, while the static locality characteristics of the Unstructured Mesh application induce little variations in the sampling efficiency.

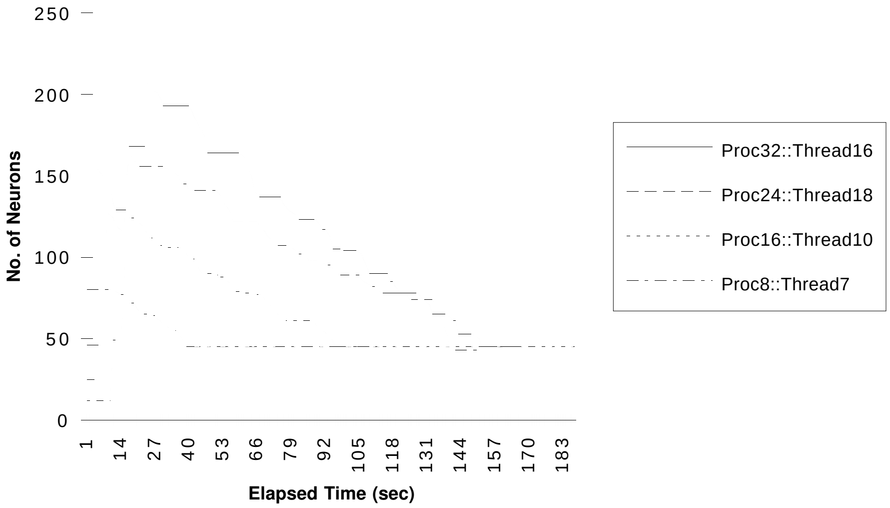

Figure 16: Barnes-Hut :: STGCS Network Sizes
Proc$x$::**Thread**y $\Rightarrow$ Locality Characteristics of Thread#y in an x processor run

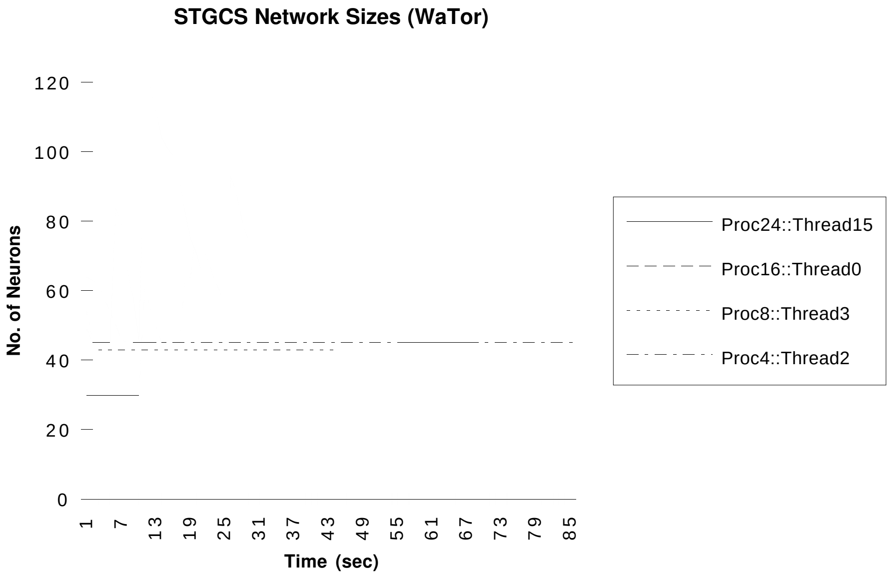

Figure 17: WaTor :: STGCS Network Sizes
Proc$x$::**Thread**y $\Rightarrow$ Locality Characteristics of Thread#y in an x processor run

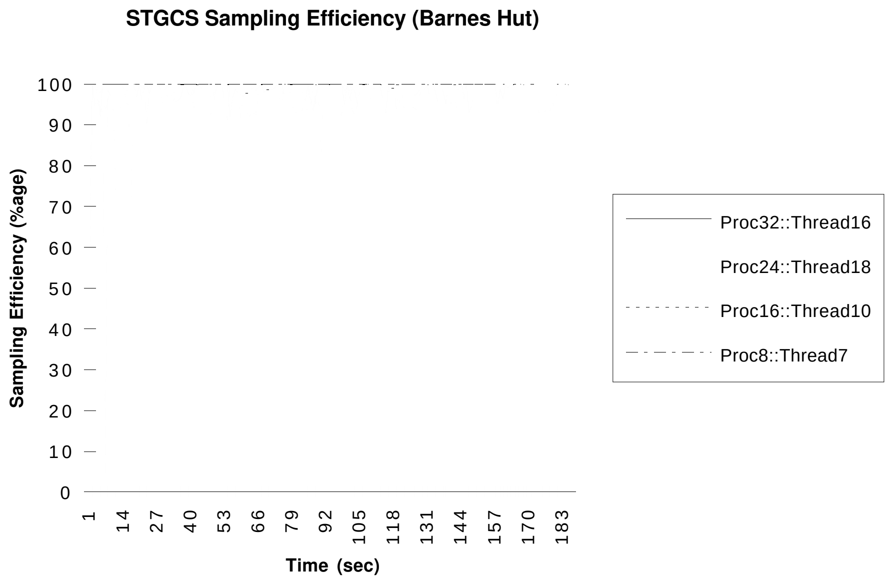

Figure 18: Barnes Hut :: STGCS Sampling Efficiency
Proc$x$::**Thread**y $\Rightarrow$ Locality Characteristics of Thread#y in an x processor run

In the Barnes Hut application the amount of computation per reference is larger and the larger inter-reference times consequently allow the network to achieve 100% sampling efficiencies (Figure 18). This curve evinces high frequency variations caused by fairly frequent but short stints of network retraining. This effect is due to the slowly varying locality fluxes inducing slow variations in the emergent CMSS distributions over time. For example, at the *14 sec.* point in the 32 processor run (Figure 18), the network size (Figure 16 increases from about 130 neurons to about 200 neurons, pointing to network retraining activity (since neuron additions occur only during retraining).

Overall, the results support the assertion that use of an STGCS coprocessor (for the operation timings assumed in Table 1) is a viable technique for monitoring processor referencing activity and the network is able to formulate reasonably stable maps of the emergent CMSS distributions without undue loss in the sampling efficiency.

9 Conclusions and Discussions

The main results of our efforts are summarized first. The STGCS clustering mechanism for extracting locality information dynamically from the incoming reference stream, has been shown to be both feasible and successful. Load balancing and execution locality is achieved over a wide range of applications. In the Barnes Hut application for example, the results show a remarkable improvement in performance over the Costzones scheme. Results in Section 8.2 indicate that the network sizes that evolve show the need for an architecturally adaptive network which is easily within the grasp of state-of-the-art hardware neural network implementations routinely dealing with network sizes of the order of thousands of neurons. Results in Section 8.2 indicate that networks converge to stable maps without undue loss in the sampling efficiency as well.

One of the recurrent assumptions in research into parallel and distributed systems, is the difficulty of achieving computational load balance without the use of *a priori* domain knowledge, such as precedence constraints [22, 23] or locality information [6, 17]. In general, most efforts on locality conscious data remapping for load balancing require the availability of a (dynamically generated or otherwise) global data dependency graph. All such software schemes need an explicit phase for the remapping-system execution, where the application execution is halted. Further, most efforts have not explored dynamic and adaptive applications such as the Barnes-Hut or the WaTor simulation explored in the current work.

In our system, the globally generated data dependency graph has been replaced with the locally captured data utility graph. The data utility graph is in effect a coarser subgraph of the data dependency graph, with data elements lumped into locality subspaces. It should be noted that the architectural adaptability of the STGCS network enables resolution of the subspaces to generate a more detailed picture if the quantization error is high, i.e. diverse elements occur within the subspaces. The results of the experiments conducted indicate that the coarse-grained local data utility graph is a novel and viable alternative to the traditional fine-grained global data dependency graphs for capturing locality information.

Secondly, our results also indicate that it is possible to capture such locality information on-the-fly from an examination of the modified processor reference streams (modified to capture references to user-tagged AMUs). The gradually increasing locality in Table 3, the execution times in Table 4 and the comparatively better locality in the WaTor application (refer Figure 13) support this claim.

Thirdly, the data utility graphs are locally generated and load transfer involves only the current owner and the processor requesting the data/task ownership. Thus, our scheme implements an incremental, decentralized and distributed paradigm for locality-conscious load balancing, which promises to be scalable.

Fourthly, the data utility graphs are generated concurrently with the main application execution as opposed to other schemes which require the application to halt execution. This is a direct consequence of our focus on extracting locality concurrent with the application execution at the architectural level.

Fifthly, and significantly, the scheme has been shown to work with a wide range of applications including dynamic, irregular and adaptive problems.

Finally, at a more abstract level, load-balancing and locality maintenance are often seen as top-down problems in the sense that the **user** maps domain knowledge about domain-inspired data abstractions into the corresponding execution locality characteristics about architectural level data abstractions. By contrast, our results show excellent prospects for a bottom-up approach wherein locality/load-balance are seen as ultimately concerned with architectural-level abstractions. The programming burden on the user is eased by successful extraction of locality and load information at this level.

References

[1] P. J. Denning, "Working Sets past and present," *IEEE Transactions on Software Engineering*, vol. SE-6, pp. 64–84, January 1980.

[2] E. P. Markatos and T. J. LeBlanc, "Load Balancing vs. Locality Management in Shared-Memory Multiprocessors," in *Proc. 1992 International Conference on Parallel Processing*, pp. I:258–267, August 1992.

[3] R. Das, R. Ponnusamy, J. Saltz, and D. Mavriplis, "Distributed Memory Compiler Methods for Irregular Problems - Data Copy Reuse and Runtime Partitioning," Tech. Rep. 91-73, ICASE, September 1991.

[4] A. Zaafrani and M. R. Ito, "Partitioning the Global Space for Distributed Memories," in *Proc. SUPERCOMPUTING '93*, pp. 327–335, 1993.

[5] J. P. Singh, W.-D. Weber, and A. Gupta, "SPLASH: Stanford Parallel Applications for Shared Memory," *Computer Architecture News*, vol. 20, pp. 5–44, March 1992.

[6] D. M. Nichol and J. H. Saltz, "Dynamic Remapping of Parallel Computations with Varying Resource Demands," *IEEE Transactions on Computers*, vol. 37, pp. 1073–1083, September 1988.

[7] A. Krishnamurthy, D. E. Culler, A. Dusseau, S. C. Goldstein, S. Lumetta, T. von Eicken, and K. Yelick, "Parallel Programming in Split-C," in *Proc. SUPERCOMPUTING '93*, pp. 262–273, 1993.

[8] A. Malony, B. Mohr, P. Beckman, D. Gannon, S. Yang, F. Bodin, and S. Kesavan, "Implementing a Parallel C++ Runtime System for Scalable Parallel Systems," in *Proc. SUPERCOMPUTING '93*, pp. 588–597, 1993.

[9] B. Chapman, P. Mehrotra, H. Moritsch, and H. Zima, "Dynamic Data Distributions in VIENNA FORTRAN," in *Proc. SUPERCOMPUTING '93*, pp. 284–293, 1993.

[10] Kendall Square Research, *KSR/Series Principles of Operation*, March 1994.

[11] J. Kuskin, D. Ofelt, M. Heinrich, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy, "The Stanford FLASH Multiprocessor," in *Computer Architecture News, ISCA '21 Proceedings*, pp. 302–313, April 1994.

[12] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam, "The Stanford DASH Multiprocessor," *IEEE Computer*, pp. 63–79, March 1992.

[13] B. M. Fritzke, "Growing Cell Structures - A Self-organizing Network for Unsupervised and Supervised Learning," *Neural Networks*, vol. 7, no. 9, pp. 1441–1460, 1993.

[14] H. Speckmann, P. Thole, and W. Rosenstiel, "A COprocessor for KOhonen's Selforganizing Map (COKOS)," in *Proc. 1993 International Joint Conference on Neural Networks*, pp. 1951–1954, 1993.

[15] R. H. Saavedra, R. S. Gaines, and M. J. Carlton, "Micro Benchmark Analysis of the KSR1," in *Proc. of SUPERCOMPUTING '93*, pp. 202–213, September 1993.

[16] D. J. Mavriplis, R. Das, R. E. Vermeland, and J. Saltz, "Implementation of a Parallel Euler Solver on Shared and Distributed Memory Architectures," in *Proc. SUPERCOMPUTING '92*, pp. 132–141, November 1992.

[17] J. E. Barnes and P. Hut, "A Hierarchical O(NlogN) Force Calculation Algorithm," *Nature*, vol. 324, no. 4, pp. 446–449, 1986.

[18] J. K. Salmon, *Parallel Hierarchical N-Body Methods.* Phd thesis, California Institute of Technology, December 1990.

[19] A. K. Dewdney, "Computer Recreations," *Scientific American*, pp. 14–22, December 1984.

[20] I. Angus, G. Fox, J. Kim, and D. Walker, *Solving Problems on Concurrent Processors*, vol. II. Englewood Cliffs, N.J.: Prentice Hall, 1988.

[21] J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. L. Hennessy, "Load Balancing and Data Locality in Hierarchical N-body Methods," Tech. Rep. CSL-TR-92-506, Stanford University, February 1992.

[22] D. Towsley, C. G. Rommel, and J. Stankovic, "Analysis of Fork-Join Program Response Times on Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, pp. 286–303, July 1990.

[23] F. C. Lin and R. M. Keller, "The Gradient Model Load Balancing Method," *IEEE Transactions on Software Engineering*, January 1987.