Architectural Support for Capturing Emergent Locality: A Connectionist Approach

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 varying 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.

Keywords: Concurrent architectural support, coprocessor, locality, load balancing, neural networks.
Architectural Support for Capturing Emergent Locality:
A Connectionist Approach

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.

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

1 Introduction

There has been considerable progress in recent years in developing memory management methods [1, 2, 3] for Distributed Memory (DM) systems [4, 5, 6, 7, 8]. However, existing memory management methods assume static or dynamic analysis performed by the system software (compilers/runtime systems) for characterizing the execution locality properties of programs. Static methods do not work well on real problems in which the nature of data locality changes drastically during program execution. Also, all dynamic runtime systems [9, 1, 10] are software solutions which incur runtime overheads. Static schemes usually require explicit directives from users for data distribution. Hence users need to pre-analyze application programs to extract locality information and this is too heavy a burden on the user. Thus, runtime schemes are a better alternative, since they perform data distribution automatically with minimal user involvement.

Existing runtime schemes can be made more efficient if the locality information can be extracted concurrent with the program execution to minimize the software overheads. There is hence an urgent need for a hardware-based architectural support mechanism that can aid load-balancing and memory management by automatically extracting and utilizing locality information from the application program. We propose such a solution, designed to be implemented as a coprocessor, that snoops on the memory bus to track changes in data locality characteristics of the concurrently executing application program.

At a high level, locality extraction can be viewed as a problem analogous to clustering or feature extraction as discussed in Section 5. In recent years, several artificial neural network learning algorithms have been proposed for such tasks; their main appeal is in being hardware implementable. Existing neural network hardware chips, for instance, contain thousands of neurons with complex interconnections [11]. This motivates our proposal of a neural network solution to the distributed memory management problem.

The results presented in this paper demonstrate, for the first time to our knowledge, the successful online extraction of sufficient locality information from processor reference-streams. The approach can easily be used for programs with static load and locality characteristics as well as programs with dynamically varying load and locality features. 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. The hardware (coprocessor-based) approach should significantly reduce the performance overheads.

The rest of the paper is organized as follows. 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 Atomic Memory Units (AMUs) residing in the Computational Metric Space (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 and Section 7 present overviews of the architectural model and the execution-driven simulation used. Section 8 presents the applications used to test our methodology and the results of the experiments. Section 9 presents conclusions.

2 Problem Statement

The problem of computational locality is addressed via the following three problems that constitute the focus of this paper:

- **Geographical Misses:** Different processes may require concurrent access to the same data that cannot be simultaneously present in multiple memories. This leads to geographical misses arising out of residency requirement conflicts. We focus on such misses since they dominate costs when compared to capacity and conflict misses.

- **Online Locality Extraction:** Traditionally, computational locality is either programmed by users exploiting domain knowledge [12, 13], or extracted by a compiler or runtime system based on source code analysis [14, 1, 15, 2]. Often, geographical misses are mitigated using software cacheing (e.g., with loop reordering) and data replication at the user-level [16, 9]. Our approach extracts locality information online by snooping on memory references. Such information is extracted automatically and is expressed in the form of the "utility" (defined later) of data elements in ongoing local computations.

- **Programming Complexity:** Many systems rely on users to build data distribution directives explicitly into application programs, using domain knowledge. Also, to take advantage of main memory paging policies and cache line replacement policies, users must choose data abstraction granularities in accordance with the granularities of such machine-specific policies. Hence, programming complexity is significantly increased as the user has to: (a) Factor multiple architectural granularities into the design of data abstractions; (b) Factor memory replacement policies into construction of programs for maintaining geographical locality; and (c) Explicitly consider issues of data layout in distributed memory architectures.

3 Solution Overview

Our solution system is based on the following definitions and assumptions:

- The system is assumed to support a global virtual address space.

- A task is defined to consist of a specific code image used to update a specific instance of a user data abstraction.

- Applications are assumed to largely follow the “Owner Computes” rule.

- A processor assigned the task of executing the code-image for updating an instance of a user data abstraction is defined as the “owner” of the task.

Our solution to ease programming complexity automates geographical locality maintenance at the architectural level using the following:

1. An abstraction called an “Atomic Memory Unit” (AMU) defining the granularity of data allocation and migration.
Figure 1: Memory Model: User data abstractions are mapped into Atomic Memory Units in various local memory modules which support a global address space.

2. An architectural level framework called the “Computational Metric Space” (CMS), within which spatio-temporal correlations among references to AMUs are captured.

3. A hardware-realizable clustering scheme operating at processor speeds for:
   - Mapping local AMU references into the CMS, and
   - Extracting spatio-temporal correlations among AMUs in the CMS.

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

4 Memory Model

This section provides details of the memory model assumed in our system, in terms of the Atomic Memory Units and their correspondence with the Computational Metric Space.

4.1 AMU Formulation

The machine-specific unit of data migration is abstracted here as the “Atomic Memory Unit” (AMU). For example, AMUs could refer to cache lines or pages. Each AMU is indivisible, contiguous and has a unique system-wide virtual address. Each instance of a user data abstraction is inscribed and aligned into one or more contiguous AMUs. Wordwise processor references are \textit{parsed} into the AMU addresses containing the word.

Figure 1 shows the association between three different user data abstractions, the local modules of the distributed memory and the virtual address space. Each square in the local memory grids corresponds to exactly one AMU. An instance of \textit{MyStruct} maps exactly into AMU1 whereas two words are left unused in mapping an instance of \textit{MyStruct1} into AMU2. Both AMU1 and AMU2 map into one AMU while the mapping of \textit{MyStruct2} into AMU3 requires two contiguous AMUs.

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

1. Address;

2. Type; identifying user-abstraction instance contained (such as \textit{MyStruct} in Figure 1).
3. Owner, as defined earlier;

4. Utility, quantifying geographical locality (defined in Section 5.2); and

5. Work, recording the workload in terms of clock cycles required (for executing associated code-image (see assumptions in Section 3 to update AMU1).

The total descriptor storage overhead is 12 bytes per AMU, assuming that the Type, Owner, Utility and 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 Distributed Shared Memory systems (and cache based memory systems in general) such as KSR1 [6], FLASH [17] and DASH [4]. 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.

4.2 Computational Metric Space

The Computational Metric Space (CMS) is a 3-dimensional space, formulated at each processor in the system, containing AMUs referenced by the local processor. The axes of the CMS are the AMU Address, Utility and Owner values as illustrated in Figure 3. The figure depicts the CMS at processor #1 in a distributed memory system formed from the reference stream of this processor. AMUs owned (see Section 3) by different processors are mapped into separate processor planes in a local CMS by virtue of their AMU Owner values. The two-dimensional shaded spatial envelopes, encompassing the AMUs within each processor plane, are referred to as locality subspaces. The locality subspaces capture the evolving spatial affinities of AMUs referenced locally and represent the local working set.

Three other variables per processor are maintained for mapping the AMUs into the CMS, as illustrated in Figure 2. These are the MinAddr, MaxAddr and the MaxProc variables. 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 the total number of processors in the system.

---

1 The notion of an AMU code-image pair is analogous to Activation Frames in [10]. However, the AMUs have a fixed granularity unlike the data associated with Frames.
Figure 3: Memory Model: CMS Details
A processor-memory system is shown. Squares in local memory grids corresponds to AMUs. Reference stream of processor #1 illustrates: (a) Local Reference to AMU1 and AMU2 and (b) Remote reference to AMU3 (resident in Memory Module #P). The CMS at processor #1 illustrates mapping of AMUs into Processor Planes in the CMS. The AR, OR and UR axes of CMS are defined in Figure 2. AST networks map emergent locality subspaces and learn the correlation strength between them. Refer text for further details.

The actual mapping of the AMUs to their corresponding coordinates in the CMS is illustrated in the lower part of Figure 2. The coordinates constitute 3-dimensional input vectors to be fed into the neural network (discussed in Section 5), for locality subspace (re)formulations and AMU Utility computations.

5 Problem Solution: AST Network
The ongoing local computations change the AMU Utility, Work and Owner values thereby altering the mapping of AMUs in the CMS. This dynamically moves the locality subspace cluster (i.e. their location) and varies their organization (i.e. size, composition and number) Hence, it can be seen that the locality subspace clusters directly capture spatial locality properties. The relative\(^2\) frequency with which local processor AMU references map to a locality subspace cluster quantifies the temporal locality of AMUs mapped to that cluster. Hence, the problems of capturing spatial and temporal locality information are equivalent, respectively, to the problems of capturing: (a) the changing CMS cluster distributions and, (b) the relative frequency with which AMUs in locality subspaces are referenced.

We have developed a new "Adaptive Spatio-Temporal" (AST) neural network for capturing such locality information as extensions of Kohonen's Self Organizing Map (SOM [18]) and Fritzke's GCS network [19]. The AST network has a two-dimensional multi-triangular topology, where each topological vertex is a neuron\(^3\) as illustrated in Figure 4[A]. Each neuron is associated with a three dimensional "weight vector" that corresponds to a point in the CMS as shown in Figure 4[D]. Topologically neighboring neurons have similar weight vectors.

\(^2\)relative to other locality subspace clusters in the CMS
\(^3\)To avoid confusion with parallel computing terminology, we use "neuron" to denote each processing unit of the network.
5.1 Tracking Spatial Locality

The AST network operates in two modes viz., a training mode and a production mode. In the training mode, the network learns a snapshot of the CMS from a set of training samples derived from past AMU references. For each AMU vector sample, the "winner" neuron with weight vector nearest the sample in the CMS is determined. The sample is said to be "clustered" by the "activated" winner and the Euclidean distance between the sample and winner defines the "Clustering Error". The weight vectors of the winner and its topological neighbors are updated, to reduce the Clustering Error.

This iterative (i.e. clustering and weight update) process terminates when weight vectors most likely represent the locality subspace cluster-centers. At this point, the planar AST network is deemed to have learned the CMS snapshot. In our studies, we required the AST clusters to at least distinguish between AMUs on different processor planes in the CMS. The spatial distance between these planes is \(1/P\), where \(P\) is the number of processors in the system. This provides the termination criterion for the training mode viz., that the network average Clustering Error over all training samples is less than \(1/P\).

In the production mode, the AST network processes AMU vectors obtained from AMU references captured via snooping. Weight adjustments in this mode refine the training mode snapshot to dynamically track the locality subspace clusters. When the Clustering Error in this mode exceeds \(1/P\) the network is retrained as elaborated above.

In addition, each neuron possesses a Signal Receptivity Meter which records its activation frequency. Locality subspace movements and reorganization change the activation frequencies and the average Clustering Error. The AST network adapts to increasing average Clustering Error by the addition of new neurons to increase CMS-snapshot resolution. Similarly, superfluous neurons with lowered activation frequencies are deleted to coarsen AST snapshot resolution in CMS areas of low input probability. The addition and deletion of a neuron are illustrated in Figure 4[B] and Figure 4[C] respectively. Thus the continual weight adjustments coupled with neuron addition and deletion serve the purpose of capturing the dynamically changing CMS distributions.
5.2 Tracking Temporal Locality

The changing temporal locality properties are tracked only during the production mode of operation. Successive AMU input vectors, in the production mode, activate two (could be the same) neurons. Each lateral weight $W_{ij}$ (see Figure 4[E]) records the frequency with which neurons $i$ and $j$ were successive winners.

The first order correlation of neuron $i$'s activations with the activations of neuron $j$ normalized with respect to all correlations predicated upon $j$'s activations is given by: $P_{ij} = W_{ij} / \sum_k(W_{kj})$. In other words, $P_{ij}$ is the conditional probability of the evolution of locality subspace cluster $i$ with respect to the evolution of $j$. The \textit{a priori} probability of the evolution of cluster $i$ is simply its relative frequency of activation (say, $F_i$). By the theorem of total probability, the total probability, $P_i$, of the evolution of the locality subspace cluster $i$ is: \( \sum_j (P_{ij} \times F_j) \).

In other words, $F_i$ represents an \textit{a priori} frequency-based probability that the locality subspace $i$ contains AMUs belonging to the local working set. Also, $P_{ij}$ can be viewed as the conditional probability that references to AMUs in cluster $i$ are correlated with references to AMUs in cluster $j$. Thus, $P_i$ is naturally interpreted as the probability that AMUs in the locality subspace $i$ are needed, \textit{given the other locality subspace clusters}, in the ongoing local computation. $P_i$ thus connotes the utility of the AMUs in cluster $i$ in the ongoing local computation and is taken to be Utility descriptor value for all AMUs in cluster $i$.

6 Architectural Support: Model

The neural network is to be implemented as an on-chip coprocessor for each processor in the system, as illustrated in Figure 5. The coprocessor executes concurrently with the main processor, and can be implemented using “synaptic parallelism” with $\Sigma \Pi \Sigma$ pipelined functional units as in [11].

1. \textbf{Training Mode Dynamics:} The following steps are repeated until the network forms a reasonably stable map of the CMS distribution from the training vector set, triggering the production mode:

- Construct training vectors until Training Buffer is filled. Descriptors are mapped by functional unit labeled (A) in Figure 5. This requires $T_{rev}$ cycles.
<table>
<thead>
<tr>
<th>AST Network Activities</th>
<th>Co-Processor Execution Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training Vector Generation ($T_{TV}$)</td>
<td>7 KSR1 cycles</td>
</tr>
<tr>
<td>Snoop Vector Generation ($T_{SV}$)</td>
<td>62 KSR1 cycles</td>
</tr>
<tr>
<td>Input Vector Quantization ($T_{Quant}$)</td>
<td>10 KSR1 cycles</td>
</tr>
<tr>
<td>Lateral Weights Update ($T_{AST}$)</td>
<td>34 KSR1 cycles</td>
</tr>
<tr>
<td>Neuron Addition ($T_{Add}$)</td>
<td>100 KSR1 cycles</td>
</tr>
<tr>
<td>Neuron Deletion ($T_{Del}$)</td>
<td>200 KSR1 cycles</td>
</tr>
</tbody>
</table>

Table 1: AST Network Simulation Timing Parameters

- Pipeline inputs from the Training Buffer into the network for clustering and weight adaptations, requiring $T_{Quant}$ cycles.
- Add a neuron if required, requiring $T_{Add}$ cycles.
- Delete a neuron if required, requiring $T_{Del}$ cycles.

Note that the application process is never halted; the coprocessor is trained using available data while the application continues to be executed.

2. Production Mode Dynamics: Repeat the following steps until Clustering Error $> 1/P$, re-initiating the training mode:

- Map the latest AMU reference into CMS and store vector in the FIFO Input Buffer. Requires $T_{SV}$ cycles.
- Feed oldest Input Buffer vector into the network, for clustering and weight adaptation. Requires $T_{Quant}$ cycles.
- Adapt lateral weight between neurons. Generate and store a new Utility value for current AMU. Requires $T_{AST}$ cycles.
- Delete a neuron if necessary, requiring $T_{Del}$ cycles.

The timings for each step above are given in Table 1, and have been derived from simulations on the KSR1 and results in [20, 6, 11].

7 Architectural Support: Simulation

An execution driven simulation was implemented on a KSR1, to provide a proof-of-concept for the proposed approach. The AMU descriptors on each processor were implemented as part of the simulation software because the system tables could not 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. Each application thread had a local hash table maintained by the corresponding AST network simulation.

In the simulation, each application thread runs concurrently with a “network thread” spawned on a dedicated processor. Each application thread’s references to AMUs were buffered along with the inter-reference times and communicated to the corresponding AST thread, which regulated its execution as per Table 1 and the inter-reference times.

An AST thread periodically checks load-levels of its application thread and that at the owners of AMUs referenced locally. If the local load is less and the local Utility is greater, ownership is acquired by the local thread and its Owner descriptor is updated. Thus, task transfers attempt to balance changing load and locality gradients in the system via a task PULL mechanism.
In order to perform load and AMU Utility comparisons, each AST thread needs access to the hash tables of other AST threads. The need for such global access to every local hash table prompted the choice of a shared memory system for implementing the simulation. The global address space semantics inherent in shared memory systems facilitates the implementation and hence the KSRI [6] was a useful simulation platform.

8 Experimental Results

The simulations primarily evaluate how well the proposed architectural support achieves computational load balance in the face of an initial skewed load distribution, while maintaining or improving computational locality characteristics. The applications chosen represent the range of locality and load balance flux normally encountered in parallel programs. The unstructured mesh application exemplifies applications with static locality characteristics, the N-body application typifies slowly changing localities for fixed dataset sizes while the WaTor application exhibits rapidly varying localities among slowly varying dataset sizes. The execution performances for each application are presented first. The performance of the AST network is described in Section 8.2.

8.1 Application Performance

We assume an SPMD programming model and each application contains an explicit time-stepping loop within which AMUs (i.e., user data abstraction instances) are updated. The same loop code is used for the update, so that tasks in our system correspond to individual AMUs. Therefore, load balancing amounts to equalizing the time spent in the update loop over all tasks in the local partitions. Task migration corresponds to a simple AMU ownership transfer, since the loop code used for updates is the same at all sites.

The initial data distribution for all AST network coprocessor runs was skewed using 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 \( a = b = \frac{\text{num}_\text{datapoints}}{\sum_{i=0}^{\text{num}_\text{procs}} i} \). Finally, individual thread execution times in the runs were used to measure the load balance achieved in the runs.

8.1.1 Unstructured Mesh Performance

The Eulerian solver simulates airflow dynamics over an airfoil discretized using an unstructured mesh [1, 21]. In the base run, the edges and vertices were randomly but equally allocation. The edge distribution for the AST network coprocessor runs was skewed as mentioned above. The dataset for each run consisted of 2800 vertices and 17,377 edges, and each simulation was run for 50 iterations.

The bar chart in Figure 6 presents the individual thread execution times for an 8 processor AST network run. The initial skewed load distribution results in the step-like thread execution timings at the end of the first iteration. Computational load balance is achieved around the ninth iteration. These observations also hold true for simulations with 4, 16 and 24 processors.

The locality achieved and maintained in the AST runs is measured as follows. Firstly, edges in each local partition were periodically examined. In traversing these edges, the number of times \( N_e \) that vertices were encountered was recorded. We also recorded the sum \( N_v \) of the number of edges emanating from all vertices encountered during the edge list traversal. The ratio \( N_v / N_e \) serves as a metric for partitional contiguity, measuring the quality of the computational locality achieved by our system.

The neural network coprocessor runs demonstrate increasingly better locality characteristics as seen in Table 2. The partitional contiguity degrades slightly with increases in the number of processors used. The reason is that the load gradient produced by the arithmetic progression is less pronounced as the number of processors is increased. Consequently, fewer task migrations are required to correct the initial load gradient, leading to fewer chances to improve geographical locality. However, the locality properties always show an improvement over the initial conditions.
Figure 6: 8-Processor Unstructured Mesh: thread execution timings.

<table>
<thead>
<tr>
<th>Processors</th>
<th>Initial $N_c/N_v$</th>
<th>Final $N_c/N_v$ (50th Iteration)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 Processors</td>
<td>0.162570</td>
<td>0.369557</td>
</tr>
<tr>
<td>8 Processors</td>
<td>0.158141</td>
<td>0.242234</td>
</tr>
<tr>
<td>16 Processors</td>
<td>0.179298</td>
<td>0.198347</td>
</tr>
<tr>
<td>24 Processors</td>
<td>0.150277</td>
<td>0.171877</td>
</tr>
</tbody>
</table>

Table 2: Unstructured Mesh locality measurements for AST neural network simulations.
Table 3: N-Body Simulation Sizes and Parameters

<table>
<thead>
<tr>
<th>NPROC</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>NBODY n</td>
<td>4096</td>
<td>8192</td>
<td>16384</td>
<td>16384</td>
</tr>
<tr>
<td>THETA $\theta$</td>
<td>0.9</td>
<td>0.9</td>
<td>0.9</td>
<td>0.9</td>
</tr>
<tr>
<td>TIME $\Delta t$</td>
<td>0.025</td>
<td>0.025</td>
<td>0.025</td>
<td>0.025</td>
</tr>
</tbody>
</table>

Figure 7: Barnes-Hut 8 processor AST neural network run: thread execution timings

8.1.2 Barnes-Hut Performance

This application performs an N-body simulation of stars in interacting galaxies, using a hierarchical algorithm. The application domain is highly non-uniform and manifests changing locality. Hence, the base run uses the CostZones [12] scheme to periodically repartition the bodies. The initial body distribution in the AST runs was skewed as above. The dataset size (number of bodies) is static. As suggested in [12], the simulation sizes and parameters used are presented in Table 3. Further details and references for this application may be found in [12, 22, 23].

The bar chart in Figure 7 shows skewed initial thread execution times for an 8 processor AST coprocessor run. Computational load balance is achieved by the third iteration and is maintained very well over the entire run. These observations hold true for the 4, 16 and 24 processor runs as well. Task transfers (if any) aim at enhancing locality while maintaining or improving the load balance. This can be seen in the dip in the thread execution timings at the tenth iteration.

Also, the variable locality properties are captured better by in the AST coprocessor run than via the Costzone scheme. This results in superior execution timings of the AST coprocessors runs over the Costzone runs as shown in Table 4. A reason for this could be the imbalanced Costzones partitions as shown in Figure 8. This figure indicates that one thread (thread #8) takes almost twice as much time in completing its computations as the other threads.

8.1.3 WaTor Performances

The ecological domain of WaTor [24, 25] is a toroidal world of water inhabited by sharks and minnows (collectively referred to as fishes). The near-neighbor interactions among fishes and the varying dataset sizes occasion dramatic locality changes. Earlier, other researchers [26] had encountered problems in implementing locality-conscious load balancing for a similar application (Radioisoty). They suggested per-processor task queues with dynamic task stealing
<table>
<thead>
<tr>
<th>Run Size</th>
<th>AST Run Timing (sec)</th>
<th>Costzone Run Timing (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 Processes</td>
<td>81.531049</td>
<td>129.470271</td>
</tr>
<tr>
<td>16 Processors</td>
<td>98.942039</td>
<td>104.776103</td>
</tr>
<tr>
<td>24 Processors</td>
<td>65.029432</td>
<td>106.128676</td>
</tr>
<tr>
<td>32 Processors</td>
<td>60.083568</td>
<td>78.317244</td>
</tr>
</tbody>
</table>

Table 4: Barnes-Hut Execution Timings Comparison

Figure 8: Barnes-Hut 8 Processor CostZones Run :: Thread Execution Timings
Figure 9: WaTor 8 Processor AST Run :: Thread Execution Timings

<table>
<thead>
<tr>
<th>Run Size</th>
<th>AST (Minnow Loop)</th>
<th>CQ (Minnow Loop)</th>
<th>AST (Shark Loop)</th>
<th>CQ (Shark Loop)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 Processors</td>
<td>23.695952 sec</td>
<td>25.314861 sec</td>
<td>4.670540 sec</td>
<td>4.126487 sec</td>
</tr>
<tr>
<td>8 Processors</td>
<td>11.132383 sec</td>
<td>13.082895 sec</td>
<td>1.948729 sec</td>
<td>2.173711 sec</td>
</tr>
<tr>
<td>16 Processors</td>
<td>6.146037 sec</td>
<td>9.006953 sec</td>
<td>1.166132 sec</td>
<td>1.528530 sec</td>
</tr>
<tr>
<td>24 Processors</td>
<td>4.446091 sec</td>
<td>8.155250 sec</td>
<td>0.874231 sec</td>
<td>1.430807 sec</td>
</tr>
</tbody>
</table>

Table 5: WaTor Execution Timings (AST and Central Task Queue (CQ) Runs)

as a viable solution. The base WaTor runs use a central task queue to achieve the same effect. The initial shark and minnow partitions for the AST co-processor runs were skewed as described earlier. The simulation was run for 15 timesteps. The WaTor simulation parameters included a 200 x 75 ocean grid with 5000 sharks and 1563 minnows.

Figure 9 shows the thread timings for an 8 processor WaTor run using the AST coprocessor. The execution timings shown are only for the Minnow update loop, as this dominated (75% to 90% of) the total execution time. This is evident in Table 5 presenting the execution timings for AST coprocessor runs and the Central Queue (CQ) runs. 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.

A reasonable measure of the execution locality is the sum of the Euclidean distances separating successive fishes in the local partitions. Figure 10 illustrates that the AST coprocessor runs achieve increasingly better execution locality. In each of the AST runs, the inter-task distances decrease from an initial high of 75 to level off at about 40. This was not true in the CQ runs, wherein the average Euclidean distances stayed consistently at about 75. This indicates that the local partitions in the AST runs achieved increasing contiguity and compactness in spite of the increasing dataset sizes.

It is evident from Table 5 that the AST 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 AST coprocessor runs were found to be superior to the CQ runs. Hence, we argue that improved execution locality plays a significant role in the better AST coprocessor run performances; beyond what could be expected due to the central task queue bottleneck in the CQ runs.
Figure 10: WaTor AST runs Locality :: Thread Intra-Task Distances
Proc$x$::Thread$y \Rightarrow$ Locality Characteristics of Thread$y$ in an $x$ processor run

8.2 Neural Network Performance

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

8.2.1 Neural Network Sizes

AST networks of widely varying sizes were evolved, underscoring the need for a neural network with an adaptive topology. For example, the network sizes varied from 20 neurons to 200 neurons (Thread 16 in the 32 processor run) in the Barnes-Hut application, illustrated in Figure 11. We do not present AST sizes of the WaTor and Eulerian Solver runs due to limitations on space.

In this sense, the network sizes and their variations provide good indications of the locality properties of applications:

- In the Unstructured Mesh application, static locality characteristics and dataset sizes produced little fluctuation in the corresponding AST sizes.

- In the Barnes-Hut application, slowly varying locality and static dataset sizes required large networks to capture the initial CMS distribution. Subsequent slow changes in the locality subspaces are tracked by the online weight adaptations during production use and the network size shrinks, mapping only the active subspaces.

- Dramatic locality fluxes in slowly varying dataset sizes in the WaTor application produced network sizes which varied far more (in terms of range and frequency) than in either of the other two applications.

8.2.2 Sampling Efficiency

During periods in which the network is being trained, references from the concurrently executing application processor are disregarded. For evaluating the ability of the AST hardware to track the ongoing computation, the sampling efficiency of the AST network measured by the ratio

$$SampEff = \frac{\#\text{Refs. Processed}}{\text{Total Processor Refs.}}$$
is periodically computed, at intervals of 0.3 seconds of simulation clock time.

The sampling efficiency is consistently high for the AST networks evolved in all applications, ranging between 90% to 100%. This indicates that the training mode is of a relatively short duration, as seen in Figure 12. This curve shows high frequency variations caused by fairly frequent but short stints of network retraining. This effect is due to the slowly varying localities inducing slow variations in the CMS distributions. For example, at the 14 sec. point in the 32 processor run (in Figure 12), the network size (see Figure 11) increases from about 130 neurons to about 200 neurons, pointing to network retraining activity, since neuron additions occur only during retraining.

The results support the assertion that the AST network coprocessor is a viable mechanism for monitoring processor referencing activity. The AST network is able to formulate reasonably stable maps of the changing CMS distributions without undue loss in the sampling efficiency.

9 Conclusions and Discussions

The AST neural network mechanism proposed 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. The proposed network is implementable using current state-of-the-art hardware neural network implementations. Results in Section 8.2 indicate that networks converge to stable maps without undue loss in the sampling efficiency.

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 [27, 28] or locality information [14, 22]. In general, most efforts on locality conscious data remapping for load balancing require the availability of a 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 application.

Our results demonstrate the following:

- In our system, the global data dependency graph has been replaced by coarser, local subgraphs (i.e. CMS
locality subspaces) capturing locality characteristics pertinent to the local computation. The adaptive topology of the AST network automatically resolves or coarsens the subgraphs as necessary (based on Clustering Error/Activation Frequency). The results indicate that the dynamic local subgraph is a novel and viable alternative to the traditional fine-grained global data dependency graphs. Further, our results indicate that it is possible to capture locality information online by snooping on the processor reference stream. The gradually increasing locality in Table 2 and the execution times in Table 4 support this claim.

- The data utility graphs are locally generated and load transfers involve 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.

- 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.

- The scheme can be used for 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 architecture-level abstractions. The programming burden on the user is eased by successful extraction of locality and load information at this level.

References


