

# A Cluster Implementation for the Parallel Programming Language SequenceL

Per Andersen, Daniel Cooke, Nelson Ruston, Julian Russbach  
Department of Computer Science, Texas Tech University  
2500 Broadway, Lubbock Texas 79409

*Abstract - SequenceL is a concise, high-level language with a simple semantic that provides for the automatic derivation of many iterative and all parallel control structures. The semantic repeatedly applies a "Normalize-Transpose-Distribute" operation to functions and operators until base cases are discovered. Base cases include the grounding of variables and the application of built-in operators to operands of appropriate types. The automatic derivation of many iterative and all parallel control structures suggests that the language is a good candidate for execution on a parallel computer. This paper presents an implementation of the SequenceL Tuple Space/Thin Evaluator architecture on a distributed memory multicomputer.*

Keywords: SequenceL, Cluster, Functional Language, Parallel Programming

## 1.0 Introduction

We have already shown in an earlier paper [8] that SequenceL is a language that discovers parallelisms in a program from the nature of its execution cycle. The development of a prototype compiler utilizing a multithreaded approach for a shared memory system (SMS) was the first attempt to realize what we understood theoretically as a new parallel programming language approach for parallel programming development [2]. Results from this early work were significant in that it led to the development of a new execution model for SequenceL, the SequenceL Normalize-Transpose-Distribute (NTD) execution model. The research presented in this paper is based on this execution model and applying it to a distributed memory multicomputer. Although the first SequenceL compiler was developed for a Shared Memory multiprocessor [2, 3] we recognized that in order to gain wide spread acceptance and to have the greatest impact on parallel programmers a compiler or execution system for a Distributed Memory System (DMS) multicomputer was required. Low cost commodity based parallel computers such as Beowulf [5] clusters were the target system for this new SequenceL compiler and/or execution environment. This paper presents the research done to date on the development of a cluster-based architecture for executing SequenceL code.

The paper is organized as follows; Section 2 gives a brief outline of the language, section 3 presents the SequenceL Normalize-Transpose-Distribute (NTD) execution model, section 4 presents the Tuple Space/Thin Evaluator architecture and section 5 provides information on the results of experiments with this architecture and thoughts on the future direction of this work.

## 2.0 SequenceL, the Language

A number of papers have been presented on SequenceL [6, 7] only a brief description of the language will be given here. The language most similar to SequenceL is parallel Haskell. [11, 12] A fundamental difference, however, is that parallel Haskell requires the programmer to direct the decomposition of aggregated data structures. SequenceL [6, 7, 8] does not require the programmer to direct the decomposition of data structures, but instead performs the decompositions automatically through a series of simplification steps. In these steps, many iterative and parallel control structures are uncovered automatically as the evaluation of a problem solution evolves. We feel that implicit parallel programming

languages have the advantage over explicit parallel programming languages when used by non-expert “parallel” programmers and it is this community that we are targeting with SequenceL.

SequenceL is a functional language that operates on one data type – a sequence. SequenceL was conceived as a programming language that provided a unique programming paradigm where all data and control operations are contained in sequences. SequenceL functions are expressed as operations on sequences or in terms of other functions that operate on sequences. In turn all computations yield sequences or functions that operate on sequences and all terminating SequenceL programs return a sequence.

The following is a simple example of a SequenceL operation that squares the addition of two sequences.

$$[ ([1, 2, 3] + [4, 5, 6]) ^ 2 ] = [25, 49, 81]$$

Aside from its orthogonal approach as a language SequenceL exhibits many other qualities. It has been described as a language for processing nonscalars [6], expressing solutions as abstract data products, and finding all implicit control and data parallelisms within a program [8].

Nonscalars refer to sequences of length greater than 1 e.g. [1,2,3,4,5], whereas scalars refer to sequences of length 1 e.g. [45] (sometimes called “singletons”). When presented with operations between scalars and nonscalars SequenceL behaves as follows:

<b>Scalar operator Scalar</b>	→ <b>Scalar</b>
<b>Scalar operator Nonscalar</b>	→ <b>Nonscalar</b>
<b>Nonscalar operator Nonscalar</b>	→ <b>Nonscalar</b>

SequenceL handles operations on nonscalars of different length by its normalize function. For example, given the sequences [1, 2, 3, 4, 5], [3] SequenceL normalizes the length of the smaller sequence to the length of the larger i.e. [1, 2, 3, 4, 5], [3, 3, 3, 3, 3]. In this fashion SequenceL can compute results of nonscalars of different lengths:

$$\begin{aligned} & [1, 2, 3, 4, 5] * [3] \\ & = [1, 2, 3, 4, 5] * [3, 3, 3, 3, 3] \\ & = [3, 6, 9, 12, 15] \end{aligned}$$

Notice the repetition of each sequence to the normalized length of the longest sequence. SequenceL can also normalize nested sequences within nonscalars:

$$[4, [5, 6]], [7, 8, 9, 10, 11]$$

normalizes to:

$$[4, [5, 6], 4, [5,6], 4], [7, 8, 9, 10, 11]$$

Since the sequence [5, 6] is considered an individual element of the left sequence it is repeated as a single element when normalized. The flexibility of normalization allows SequenceL to solve operations between nonscalars of any length without returning an error.

SequenceL enables the programmer to declare intuitively the composition of the solution as one would think or say the problem

```
mmrow: [s] * [[s]] -> [[s]]
mmrow(a,b) ::= dp(a,transpose(b))
```

```
dp: [s] * [s] -> s
dp(x,y) ::= sum(x * y)
```

This can be read as - matrix multiply of two matrices “a” and “b” is the dot product of a matrix “a” with the transpose of another matrix “b”, where the dot product of two matrices is the sum of the product of each of their rows. The function declaration  $[s] * [[s]] \rightarrow [[s]]$  indicates that the first argument is to be treated by the NTD operation by one less dimension than the second argument. In this case the second argument is a matrix therefore the first argument, which is also a matrix is evaluated by row with respect to the second argument when NTD is applied [9].

### 3.0 Normalize-Transpose-Distribute Execution Model

The NTD execution model for SequenceL was developed from the work done to develop a multiprocessor SequenceL compiler, this compiler generated C/Pthread code from SequenceL source code. The resultant C/Pthread code was compiled using the IRIX C compiler and executed on a 56 processor SGI Origin2000 [2]. The steps of the NTD cycle form a more explicit procedure for parallel execution. NTD works as follows; When SequenceL encounters an operation on a set of sequences the length of each sequence is checked. If the sequences are of different length, the length of the smaller sequences is normalized to the length of the longer sequence. The sequences are then transposed such that the *ith* element of each sequence is paired with the *ith* element of every other sequence. There are now *i* unique subsequences in the sequence. The operator is then normalized to length *i* and transposed among all *i* subsequences. Following this each pair can be distributed and evaluated in parallel. Below shows the steps of the process, prefix notation is used for ease of demonstration:

```

      +([2,4,6,8,10,12], [1,3,5])
N     +([2,4,6,8,10,12], [1,3,5,1,3,5])
T     +([[2,1], [4,3], [6,5], [8,1], [10,3], [12,5]])
N     + + + + + ([[2,1], [4,3], [6,5], [8,1], [10,3], [12,5]])
T     [+([2,1]), +([4,3]), +([6,5]), +([8,1]), +([10,3]), +([12, 5])]
D     [+([2,1]), +([4,3]), +([6,5]), +([8,1]), +([10,3]), +([12, 5])]

```

Normalize-Transpose is repeated on nested sequences until all terms can undergo no further normalize-transpose operations. The term is then ready for distribution. The last line in the above example trace contains a parent term,

$$+([2,1]), +([4,3]), +([6,5]), +([8,1]), +([10,3]), +([12, 5])]$$

which contains 6 SequenceL child terms.

The final step, when all terms are ready to distribute is to distribute the child terms and evaluate each in parallel. Accept for a child term’s position in its parent term it is independent of all the other child terms. Therefore the first child term  $+([2,1])$  can be evaluated independent of the other five child terms. Therefore the terms, although fine-grained, could be distributed and evaluated in parallel, with the result of evaluating a child term replacing that term in its position in the parent term. This example is a simplification of the process, but is a good overview of the NTD process. When complex nesting and functions are added the NTD process can become fairly complex although its complexity is hidden from the programmers.

### 4.0 SequenceL Execution Environment

The design of the multicomputer parallel execution environment for SequenceL is based on Tuple Spaces, a shared memory model that does not require physical shared memory [4]. David Gelernter at the Yale Linda Group pioneered the concept of a Tuple Space when they proposed the language Linda [10]. SequenceL tuples are SequenceL terms, such as the 6 terms shown in the last example above, that can be evaluated by “distributed thin evaluators” which consume tuples from Tuple Space. These evaluators are thin in the sense that they only perform NTD’s, ground function bodies, and, in “base” cases, evaluate the grounded terms that cannot be further decomposed. A SequenceL evaluator is placed on each node participating in the SequenceL execution environment. Figure 1 is a high-level representation of the SequenceL architecture described in this paper.

Figure 2 is an example of a SequenceL expression being evaluated using the Tuple Space/Thin Evaluator architecture shown in Figure 1. The SequenceL expression is a matrix multiplication of two matrices,  $((7, 2), (8, 3))$  and  $((9, 4), (1, 5))$ . An “advocate” processor performs the initial NTD and places resultant tuples, to be evaluated, in the Tuple Space. Notice, in step 1, the advocate processor performs an NTD on the *mmrow* function. The function *mmrow*’s first argument is declared as one dimensional, therefore the two constituent rows making up the matrix in the first argument result in a normalization operation being applied to the second matrix resulting in two copies of the second matrix. The transpose that follows yields  $mmrow((7,2), ((9,4),(1,5)))$   $mmrow((8,3), ((9,4),(1,5)))$ . The distribute operation places header information *AP(1)* and *AP(2)* in each of the two tuples indicating the position and processor to which results will be returned and then places the tuples in Tuple Space. SequenceL thin evaluators exist on all processors and are in constant search of work. Seeing the two tuples, evaluators on other processors (processors 1 and 2 in this example) quickly remove the tuples, ground the *mmrow* functions, perform the specified *transposes*, and then each processor places a tuple back into the Tuple Space (steps 2 and 3). Processors 3 and 4 now grab the work from Tuple Space, perform NTD’s on the respective *dp* functions and each place two tuples in Tuple Space, with appropriate header information. (Steps 4 and 5 in Figure 2). In step 6, other evaluators consume the tuples from the tuple space, perform additional NTD’s, and ultimately perform the arithmetic leading to the result tuples being placed in Tuple Space in step 7. Notice the multiplications could also have been performed in parallel – this is not shown to preserve space in this paper. In step 8, the advocate collects the resultant tuples based upon the header information and assembles and evaluates the final result. This example illustrates the SequenceL NTD process as it relates to the Tuple Space/Thin Evaluator architecture; in reality the amount of computational work in this example is not large enough to warrant placing tuples in Tuple Space. In the actual implementation of the architecture a workload sharing mechanism determines when tuples and how many tuples should be placed in Tuple Space.

The Tuple Space/Thin Evaluator abstraction was evaluated through the implementation of a SequenceL execution environment on a Cluster of Workstations (COW). The hardware arrangement for the MPI-based version was a 9 node COW with Rocks Cluster installed [13]. A SequenceL thin evaluator was written in C for execution on the nodes of the COW. SequenceL code was parsed and translated into an intermediate form suitable for storage in a hash table. Each hash of the hash table represented a SequenceL tuple, which could be evaluated by a thin evaluator or distributed in parallel. The idea of the hash table was to unfold the SequenceL code into hashes that could be arbitrarily examined, distributed as tuples and evaluated in parallel. In addition to carrying SequenceL terms for evaluation the tuples also carried information on how to return the evaluated term back to the advocate processor as explained in the example above.

A single node was chosen as the initiator with respect to evaluating a SequenceL program. When this initial node generated enough tuples the workload sharing mechanism would begin distributing the tuples to idle processors. The workload balancing mechanism was implemented in an attempt minimize communication overhead and maximize computational time. The workload balancing mechanism is based on a simple busy/idle processor mechanism [1]. A SequenceL tuple stack is used to keep track of which SequenceL terms in the hash table are ready for evaluation. Designating a node as “busy” makes it eligible for sharing some of its tuples, a node designated as “idle” means it was available to receive tuples from busy nodes. Between a node’s busy and idle state is a node state that can be best described, as a “working” state which means the node has tuples to evaluate but not enough to share. The communication protocol for keeping tracking of busy and idle nodes and communicating that information to all nodes in the COW was inspired by the token ring protocol. For the SequenceL Tuple Space/Thin Evaluator architecture a token travels around a logical ring visiting each node in the COW in turn. The token contains information on the current workload state of each node in the COW. When a node receives the token it updates the token with its current load, if a node holding the token is designated as busy and sees another node in the ring is designated as idle, the busy node will send load directly to that node. When a node is not engaged in sending load to another machine or token-based operations, it will compute parallel tasks using its SequenceL thin evaluator. This allows serial execution of parallelisms by default and periodic dynamic load balancing when needed on the network. The nodes of the network are considered autonomous –

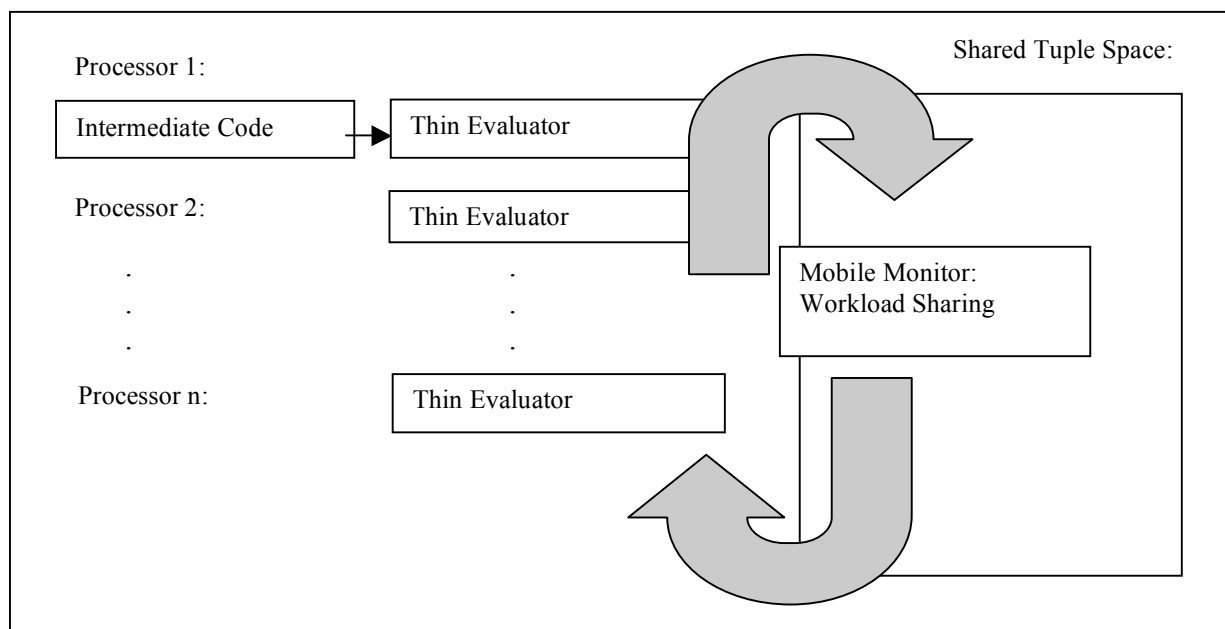


Figure 1: SequenceL Architecture

decisions are made independently and all communication is peer-to-peer. There is no active arbitrator or server thus bottlenecking is not an issue and the token passing provides synchronicity between machines. Nodes that are neither busy nor idle can chose to ignore the token, when this happened a timeout occurs and the next node in the token ring is selected to receive the token. All communication is asynchronous (i.e. non-blocking) except when sending buffer size information between nodes so as to anticipate the size of a large message. The communication code and computational code is multi-threaded, a node can be sending or receiving the token, concurrent with computation. Atomicity between communication and computation was only enforced when packing, unpacking, pushing, and popping to and from the SequenceL tuple stack when workload sharing was taking place.

## 5.0 Conclusions

An implementation of a SequenceL Tuple Space/Thin Evaluator concept on a distributed memory was realized by the research reported in this paper. As a result of this implementation effort a number of problems were uncovered; the most significant problem encountered was the large percentage of the execution time that was spent in data serialization of SequenceL tuples as a result of workload sharing. The data serialization overhead is associated with the time required to move the SequenceL tuples out of the hash table and into an MPI message for transfer to idle nodes. A second problem was also uncovered also related to the workload sharing mechanism. Threshold setting are used to determining whether a node is busy, idle or working the amount of work a node has is estimated from the SequenceL expressions contained in tuples to be evaluated. This workload estimate is done dynamically during runtime as opposed to statically at compile time. Projecting the work before the expression is evaluated at runtime is a difficult task to accomplish for anything but the simplest expression. Once expressions become complex, such as deeply nested or recursive SequenceL expression, reasonably accurate workload estimates become expensive to compute in terms of overhead. For the Tuple Space/Thin Evaluator architecture discussed in this paper the runtime workload estimates were restricted to the built-in operators and involved only lightly nested sequences. The capability to do reasonable runtime workload estimates of NTD operations involving complex SequenceL expressions and the resultant parallelization of these functions has been left for future research. With the successful demonstration of the Tuple Space/Thin Evaluator concept, the next immediate task is the development of a new SequenceL internal data representation that preserves the fundamental SequenceL evaluation paradigm NTD, while at the same time providing for a more efficient mapping between the SequenceL tuple format and the MPI message buffer. The latest language developments can be found in the latest SequenceL paper [9].

$mmrow: [s] * [[s]] \rightarrow [[s]]$   
 $mmrow(a,b) ::= dp(a,transpose(b))$   
 $dp: [s] * [s] \rightarrow s$   
 $dp(x,y) ::= sum(x * y)$

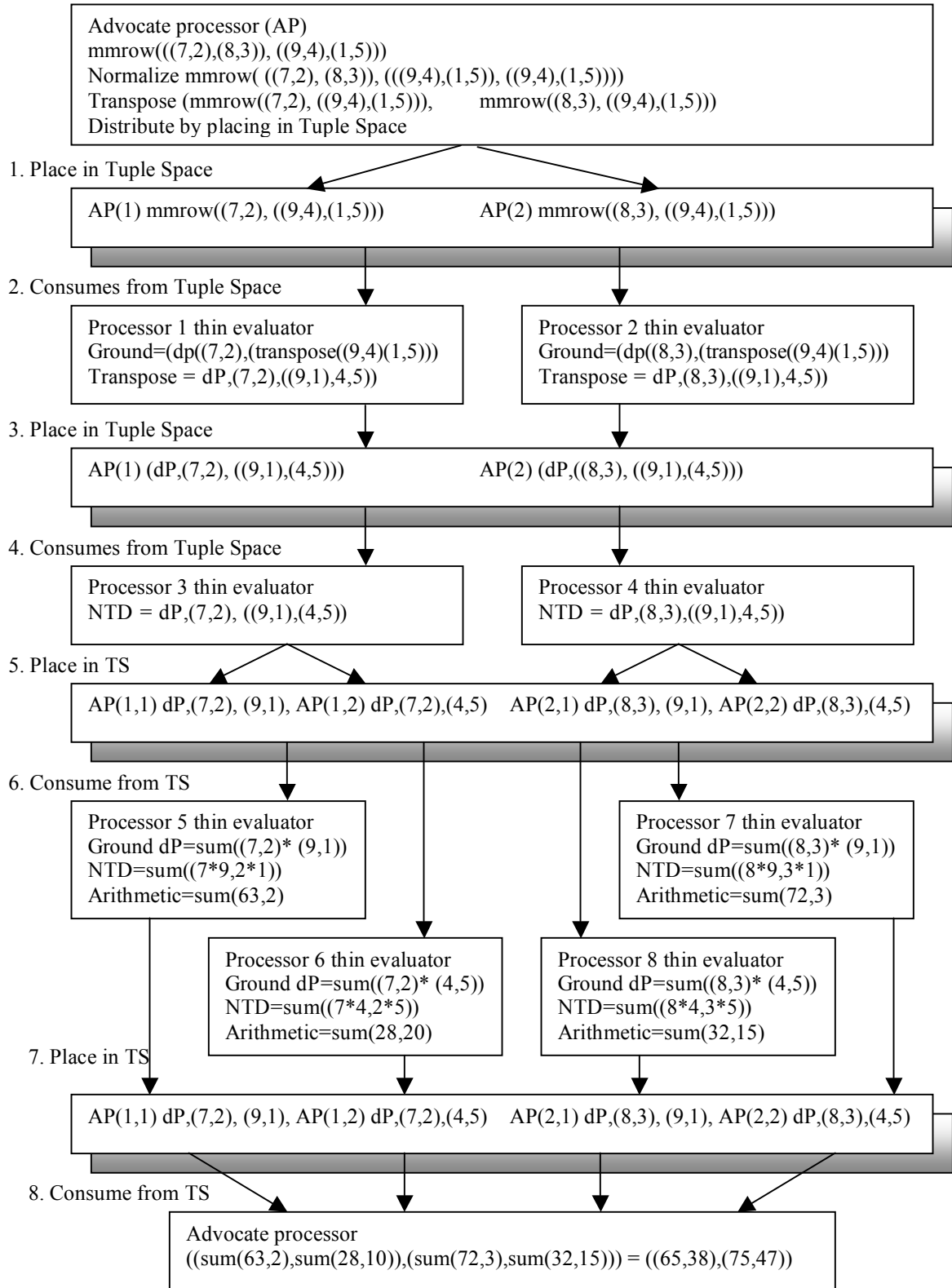


Figure 2. SequenceL Tuple Space/Thin Evaluator Execution Model

## 6.0 References

- [1] Per Andersen and John K. Antonio, "Implementation and Utilization of a Heterogeneous Multicomputer for the Study of Load Balancing Strategies," 7th IEEE Symposium on High-Performance Distributed Computing, Chicago, July 1998.
- [2] Per Andersen, "A Parallel SequenceL Compiler," PhD Thesis, Texas Tech University 2002.
- [3] Per Andersen and Daniel E. Cooke, "Assessment of SequenceL as a High-Level Parallel Programming Language," 15th International Conference on Parallel and Distributed Computing, November 3-5, 2003 Marina del Rey, CA, USA.
- [4] Henri E. Bal, Jennifer G. Steiner, Andrew S. Tanenbaum, "Programming Languages for Distributed Computing Systems," ACM Computing Surveys, Vol 21(3), September 1989, ACM Press NY NY. pp 261-322.
- [5] The Beowulf Cluster Site. <http://www.beowulf.org>, June 2005.
- [6] Daniel E. Cooke, "An Introduction to SEQUENCEL: A language to experiment with Nonscalar Constructs," Software Practice and Experience, Vol. 26(11), November 1996, pp. 1205-1246.
- [7] D. Cooke, "SequenceL Provides a Different way to View Programming," *Computer Languages* 24 (1998) 1-32.
- [8] Daniel E. Cooke, Per Andersen, "Automatic Parallel Control Structures in SequenceL," Software – Practice & Experience, Volume 30 Issue 14, pp. 1541-1570, 2000.
- [9] Daniel Cooke, Nelson Rushton, Per Andersen, Changming Ma, Adem Ozyavas, "Normalize, Transpose, and Distribute: An Automatic Approach for Handling Nonscalars," ACM Transactions on Programming Languages and Systems, [in progress].
- [10] David Gelernter, "Generative Communication in Linda," ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 7 Issue 1 ACM Press, January 1985.
- [11] Loidl, H-W. Rubio, F. Scaife, N. Hammond, K. Horiguchi, S. Klusik, U. Loogen, R. Michaelson, G.J. Pena, R. Priebe, S. Trinder, P.W., "Comparing Parallel Functional Languages: Programming and Performance," Higher-Order and Symbolic Computation 16 (3): 203-251, September 2003.
- [12] R.S. Nikhil and Arvind, "Implicit Parallel Programming in pH," Morgan Kaufmann, San Francisco, 2001.
- [13] Rocks Cluster. <http://www.rockscluster.org>, accessed Feb 18th 2006.