Department of

Computer Science

A Comparison of Explicit and Implicit Programming Styles for Distributed Memory Multiprocessors

Matthew Haines and Wim Bohm

Technical Report CS-93-104

March 30, 1993

Colorado State University

A Comparison of Explicit and Implicit Programming Styles for Distributed Memory Multiprocessors

Matthew Haines[*] Wim Böhm[†]

Computer Science Department
Colorado State University
Fort Collins, CO 80523

March 25, 1993

Abstract

Task and data management in distributed memory multiprocessors can be expressed explicitly in the programming language or can be provided implicitly by the compiler or runtime system. In this paper we compare three programming paradigms for distributed memory multiprocessors: implicit task and data management using the functional programming language Sisal, explicit task management and implicit data management using C combined with the virtual addressing runtime system VISA, and explicit task and data management using C with message passing primitives. We measure, in both time and space, the programming effort and performance of each of these paradigms. We show that the implicit programming style offers good performance for our benchmarks, but that even higher levels of performance can be obtained at the cost of lowering the programming abstraction, resulting in more complex and machine-dependent programs.

[*]Supported in part by a grant from Sandia National Laboratories
[†]Supported in part by NSF grant MIP-9113268

1 Introduction

Large-scale distributed memory multiprocessors represent the current state of the art in high-performance computer architecture. Programming these machines requires the management of both parallel tasks and distributed data, which is often done explicitly using language constructs for spawning and synchronizing tasks, and for message passing. The resulting programs are difficult and time-consuming to write, and contain a large amount of machine dependent housekeeping code not germane to the specification of the problem. An alternative approach is to employ a software system to provide implicit management for tasks and/or data.

This paper introduces the design of a runtime system, called *VISA* [6], for implicit memory management on a distributed memory multiprocessor. The compiler or programmer is provided with a shared memory abstraction, and a set of primitives for allocating and accessing shared data structures within a virtual address space. Data structures are allocated using a variety of *data decompositions* specified by a set of predefined or user-defined mapping functions. We compare the merits of three programming styles: Sisal with VISA, explicit parallel C with VISA, and explicit parallel C with message passing. In the Sisal with VISA case the compiler inserts the appropriate VISA calls, whereas in the C with VISA case the programmer does. It would be interesting to compare the performance of these approaches to a parallelizing compiler for sequential C or Fortran, but such a compiler is not available on the nCUBE/2. Using two relatively simple problems, Successive Over-Relaxation and Lawrence Livermore Loop #7, we measure the programming effort in terms of program length and programming time for these paradigms. These measures are clearly subjective, but given the state-of-the-art of parallel software engineering, the best we can provide. We measure the execution time and storage use of our programs. In the panel discussion at *Supercomputing 92* [2], one of the authors claimed that it is considerably easier to write parallel programs for distributed memory multiprocessors in an implicit style rather than in an explicit style, and that the implicit style does not need to suffer from an overwhelming loss of efficiency. This paper quantifies these claims.

Sisal is a functional language that supports data types and operations for scientific computation [10]. The Sisal compiler consists of three parts: a frontend, a backend, and a runtime system. The frontend translates the source program into intermediate dependence graph form. The backend optimizes the intermediate representation and generates native C code. The *runtime system* provides the Sisal compiler with two main abstractions: task management and memory management. We are working on a runtime system that provides support for both abstractions in a distributed memory environment, and in [4] we introduced the design and initial performance of the distributed task management abstraction.

In Section 2 provides an overview of VISA, and the design and implementation of the supporting system. Section 3 describes the benchmarks used in evaluating the three programming methods, and a description of the programming effort of each approach. Section 4 provides the performance of each of the programs and an analysis of the results. Section 5 provides a brief description of related research projects, and we conclude in Section 6.

2 The Design and Implementation of VISA

VISA is a distributed memory runtime system that provides a single addressing space and general data decomposition functions to a programmer or compiler. The compiler augments a parallel program with VISA primitives for allocating and accessing the data structures to be kept in the single addressing space. Any variables not placed in the VISA space are unaffected by the system. The augmented program is then compiled using the native language compiler of choice, and linked with the VISA library to create the object program, which can then be executed on a distributed memory multiprocessor.

2.1 Message Passing

All message passing required for accessing remote values is handled *implicitly* by the VISA system through the use of a *message passing abstraction*, supporting both synchronous (blocking) and asynchronous (non-blocking) operations. Since these operations are provided by most host operating systems for distributed memory multiprocessors, VISA can be easily ported to other distributed memory multiprocessors by modifying the message passing abstraction to make the proper native calls.

Specifically, the abstraction supports a non-blocking send abstraction (*WriteMsg*), a blocking receive abstraction (*ReadMsg*), and an asynchronous receive abstraction using interrupts and an interrupt handling routine (*MsgInterruptHandler*). Asynchronous message reception requires polling at some level to determine when a message arrives and take appropriate action. Most systems, including the nCUBE/2, provide hardware polling for incoming messages, resulting in a hardware trap that is caught by the operating system, and then passed into the user-level in the form of an interrupt. The interrupt causes a VISA message interrupt handler to deal with the message. If the interrupt handler is allowed to be invoked at any arbitrary time during the computation, it cannot modify the global state of the computation. Therefore, either the interrupt handler must be selectively disabled during the times when global data structures are accessed, or it must be prevented from modifying global data structures. The former option requires the placement of expensive system calls for enabling and disabling interrupts around all global data structure accesses, which can be costly and error-prone. Therefore, the VISA system employs the latter option: Any message requiring a global modification is enqueued onto a message list for handling outside of the scope of the interrupt handler.

2.2 Data Distribution

As depicted in Figure 1, the VISA address space is allocated in part of the local memory of each participating node. This creates two types of addressing space for each participating node in the system: a shared *virtual* addressing space that spans all of the processors, and a *local* address space for data visible only to the local processor. Each data structure allocated to the VISA

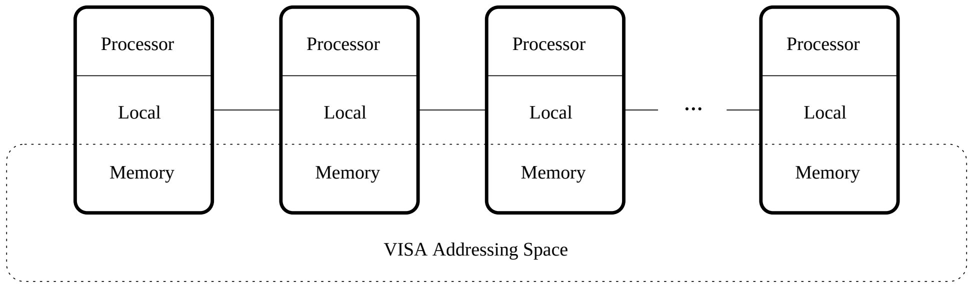

Figure 1: The VISA Addressing Space

Mapping Function	Blocksize	Start PE	Replicate
scalar_map	n	map_arg	No
replicate_map	n	P_{id}	Yes
block_map	n/p	map_arg	No
variable_block_map	map_arg	P_0	No
interleave_map	1	map_arg	No

Table 1: Control Parameter Settings for Various 1D Mapping Functions

space receives a contiguous set of *virtual* addresses shared among the nodes, which are mapped onto *physical* addresses from each node.

Data distribution determines how the physical storage for a global data structure is to be divided among the participating nodes. The goal is to divide the data structure among the nodes so as to minimize the number remote references caused by the distribution. This means that the distribution of data must be tied to the access pattern of the parallel computation, and therefore data distribution needs to be flexible to support a wide variety of access patterns. For VISA, data distribution is accomplished by dividing a data structure into a set of blocks, where each block contains *blocksize* elements. The blocks are then allocated to the physical memories of the nodes in round-robin fashion. To facilitate a variety of distribution schemes, we assign a set of *control parameters* to each data structure that define the blocksize (`blocksize`), the node to which the first block is assigned (`start_node`), and the processor stride at which the blocks are distributed (`stride`). A fourth control parameter specifies whether or not a data structure is to be replicated. Table 1 details these parameter settings for several one-dimensional mapping functions, where the *map_arg* is passed in from the allocation routine, typically specifying the starting node. The stride is always 1 for these one-dimensional mapping functions, but varies for some of the multi-dimensional mapping functions. VISA provides support for multi-dimensional data structures, but we restrict ourselves to one-dimensional data structures in this paper.

Specification of the distribution function is accomplished by passing the name of a mapping function, such as defined in Table 1, to the VISA memory allocation routine, *visa_malloc()*.

Field	Function
`visa_base`	The range of global virtual addresses
`local_base`	The range of local physical addresses for locally-owned blocks
`optimized_base`	The range of optimized virtual addresses
`nelems`	The number of elements
`size`	The size of each element
`blocksize`	The blocksize (elements per block) used for distribution
`start_node`	The node ID on which to begin distributing the blocks
`stride`	The stride at which to distribute the blocks
`replicate`	A boolean to determine if this data structure is replicated
`table_index`	The index into the range_map table for this entry
`next`	A utility pointer

Figure 2: Description of a `range_map` entry

Although a wide range of common mapping functions are pre-defined by the VISA system, it is possible for the user to define a new mapping function, such that the mapping function establishes the desired values of the control parameters.

2.3 General Address Translation

Address translation is the process of obtaining the physical address of a datum given its virtual address. For a distributed memory multiprocessor, a physical address consists of the tuple *(node, pa)*, where *node* is a node designator and *pa* is the physical address within that node. Since VISA employs a block-based addressing scheme, where the blocksize, starting node, and stride may all vary, it is necessary to store these control parameters, along with other information about each data structure, in a descriptor called a *range_map entry*. The entire VISA space is therefore described by the collection of these entries, called the *range_map table*. The term "range" refers to the fact that, since all data structures are assigned contiguous addresses in both virtual and physical spaces, the range (low, high) is sufficient to represent all of the addresses within a data structure. To ensure local access of the range_map entries, the range_map table is replicated.

In addition to the control parameters, each range_map entry (see Figure 2) contains three address ranges for each data structure:

- The *visa_base* represents the range of global virtual (VISA) addresses for this data structure.
- The *local_base* represents the range of local physical addresses of the blocks that are allocated locally for this data structure.

- The *optimized_base* represents the optimized range of global addresses, as explained in Section 2.4.

Address translation proceeds as follows:

- The range_map entry for the desired data structure is fetched by the *find_rm()* routine, which is exposed to the compiler so that the range_map entry for a data structure that is to be accessed many times need only be fetched once.

- From a *virtual address*, the relative element position within the data structure (`element`), the block containing the desired element (`block`), and the offset of the element within this block (`block_offset`) are computed:

  ```
  element = address - low_range
  block = element / blocksize
  block_offset = element mod blocksize
  ```

- Now the node which possesses the block (`node`), the relative block number within that node (`node_block`), and the relative offset of the actual datum within the node (`rel_offset`) are computed, where `P` is the number of participating nodes:

  ```
  node = (start_node + (block * stride)) mod P
  node_block = block / P
  rel_offset = node_block * blocksize + block_offset
  ```

  `node_block` is necessary to accommodate more than one block from the same data structure being assigned to the same node, such as where there are more blocks then nodes.

- If the access is local (i.e. `node` is equal to the local node designator) the `rel_offset` is incremented by the *local_base* from the range_map entry to produce the actual offset in local physical memory:

  ```
  offset = rel_offset + local_base.
  ```

  If the access is remote, a message is sent to the specified node, requesting that the desired datum be fetched and returned.

An alternative to this address translation scheme is to have a *fixed* blocksize, start_pe, and stride for every data structure. Address translation calculations can then proceed directly from the virtual address bits. We have implemented this fixed addressing scheme and found that although the actual translation process is faster, the fixed control parameters often cause mis-alignment with the parallel loops which access the data structures, resulting in an excessive number of remote references and severely degraded overall performance of the application [5]. Thus we have found it more effective to provide flexible decompositions using the variable control parameters, and to eliminate address translation for local references, which we refer to as *optimized* address translation.

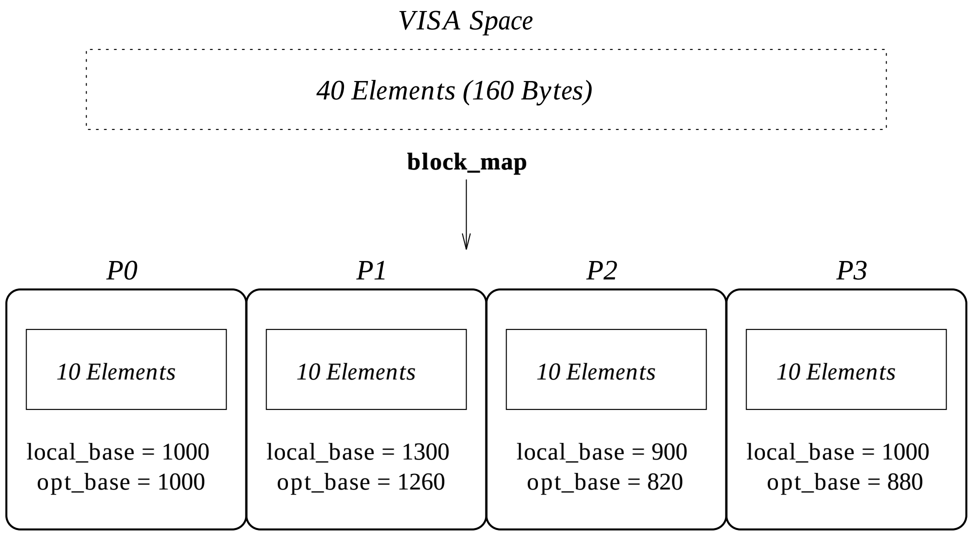

Figure 3: Sample VISA data structure with computed optimized_base values

2.4 Optimized Address Translation

In order to eliminate address translation for local VISA accesses, we introduce a new function, called *visa_opt*, which re-writes the virtual base address with the structure's *optimized* base address, and establishes a pair of "water mark" registers to hold the low and high values of the range corresponding to the local_base. The optimized_base is the local_base minus the offset necessary to generate a global address that will result in a local access. For example, suppose an array of 40 integers (4 bytes each) is allocated using *block_map* among 4 nodes, as depicted in Figure 3, where the local_base values are different for each node, which is possible since each node manages its local memory independently of the other nodes. Each processor would allocate local storage for *blocksize = 10* elements (40 bytes), and set the local_base accordingly. If, for example, the third node wishes to optimize the base address for this structure, then the optimized value is the local_base minus 20 elements (80 bytes), corresponding to the two blocks of 10 elements each that proceed it in the distribution. Once the base address for a structure has been optimized, any further access to this structure, represented as some offset from the base, will be checked against the low and high water marks. If the computed address falls within the water marks, then the access can proceed without translation, otherwise the address is passed along to the VISA access routines for general address translation and proper remote handling. Special macros are defined to perform the water mark checks, so that the total overhead for a local access has been reduced to the time required for three comparisons.

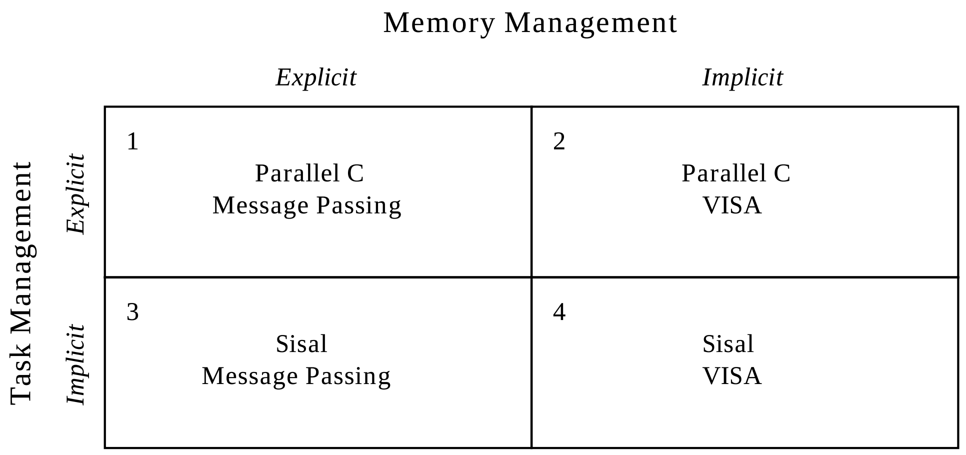

Figure 4: Parallel Programming Style Combinations

3 Benchmarks

A parallel program executing on a distributed memory multiprocessor must address two issues, either explicitly or implicitly:

1. Task management. Parallel execution is achieved by *dividing* the portions of code which may be executed in parallel into parallel tasks, *distributing* the tasks among the participating nodes for parallel execution, and *synchronizing* their results so that the computation remains determinate.
2. Memory management. Global data structures need to be *distributed* among the participating nodes in such a way as to minimize the number of remote references generated by the execution of the parallel tasks. Once a distribution is agreed upon, the program must identify those references that fall outside of the local distribution (i.e. remote), and communicate the request to the node which contains the value.

Given these two orthogonal programming issues, either of which may be handled explicitly or implicitly, there are four possible parallel programming style combinations, as depicted in Figure 4:

1. Explicit task management using parallel C and explicit memory management using message passing primitives. Similar to assembly language, this style represents the lowest level of abstraction, but the possibility for the highest level of performance.
2. Explicit task management using parallel C and implicit memory management provided by the VISA runtime system. This style alleviates the programmer from the details of a distributed memory system and explicit message passing.

3. Implicit task management using Sisal and explicit memory management using message passing primitives. This represents a machine-dependent Sisal compiler that has been given the ability to generate explicit distributed memory code, much like the distributed memory Fortran compilers [7, 15]. However, such a modification to the compiler has not been undertaken, and thus we cannot expand on this style in our analysis.

4. Implicit task management using Sisal and implicit memory management provided by the VISA runtime system. This represents the opposite end of the programming effort spectrum from explicit parallel C with message passing.

To measure the relative merits of each style, in terms of programming effort and execution speed, we encode two applications in the three programming styles (1, 2, and 4) specified above. In selecting our benchmarks, we wanted programs that utilized one-dimensional data structures using relatively simple access patterns that could exploit the explicit memory management style and would be simple enough to encode using explicit parallel C with message passing. We selected two codes, where each code is designed to highlight the effectiveness of either task or memory management techniques. The programs are:

- *Lawrence Livermore Loop #7.* This program creates an array A from an input array B and constants R, T, C_1, and C_2, where A_i is defined as: $A_i = B_i + R * C_1 + R^2 * C_2 + T * B_{i+3} + T * R * B_{i+2} + T * R^2 * B_{i+1} + T^2 * B_{i+6} + T^2 * R * B_{i+5} + T^2 * R^2 * B_{i+4}$. With very little task management required, this problem highlights the differences between the implicit and explicit memory management styles.

- *Successive Over-Relaxation (SOR).* This problem performs a "smoothing" operation on an array by iterating over the array and computing each new A_i as the average of the previous iteration's A_{i-1}, A_i, and $Ai + 1$. The access pattern is fixed over all of the iterations, and the array is distributed among the nodes in equal-sized blocks, matching the distribution of the parallel (inner) loop to minimize the remote references. The iteration loop in this program provides a method of controlling the amount synchronization required, thus highlighting the differences between implicit and explicit task management.

Both of these programs were encoded using the three programming styles as follows:

- *Sisal with VISA.* Both codes were transformed into Sisal directly from their mathematical descriptions. The code only specifies *what* is to be computed, not *how* the computations are to proceed. The result is a machine-independent specification of the problem that runs on any machine Sisal supports.

- *Explicit parallel C with VISA.* Moving into explicit task management, the codes have to specify how the parallel loop is to divided among the workers, and how explicit synchronization is to be performed. Memory management is handled by the VISA system, however, for the Livermore Loop #7 code, special registers were employed to cache the values of the B array so that multiple remote references to retrieve the same value were eliminated.

	LLNL Loop #7			*SOR*		
Measure	SISAL	C+VISA	C+MP	SISAL	C+VISA	C+MP
Lines of code	25	163	338	24	184	459
Time to encode (hrs)	0.25	2.5	9.5	0.25	3.0	11.0

Table 2: Comparison of programming effort, in both time and space

- *Explicit parallel C with message passing.* Moving away from the VISA system, the explicit task management code is augmented with explicit message passing designed to optimize the number of remote references required and perform all remote references before the computation loop is initiated (*pre-fetching*). Also, the communication model is changed from an interrupt-driven request/reply model used in VISA to a synchronous read/write model so that the overhead of the interrupt handler can be avoided. This allows the computation (inner) loop to run completely without remote references. Special buffers are used to hold the pre-fetched values, and synchronous communication phases are necessary to avoid deadlock. The distribution of data among the processors is also explicitly stated, and altering this distribution would require re-coding both the explicit communication and computation phases.

4 Results and Analysis

We compare the relative merits of each programming style using two metrics: programming effort and performance. Table 2 displays the programming effort in terms of lines of code that the user is responsible for writing, and approximate time it took us to code and debug each of the programs, where, as in all of our tables, SISAL represents the Sisal codes, C+VISA represents the explicit parallel C with VISA codes, and C+MP represents the explicit parallel C with message passing codes. The claim that implicit parallel languages ease the task of programming distributed memory multiprocessors is clearly supported by these numbers. We acknowledge that these measurements are subjective as to the overall programming effort, however, they do paint a realistic picture of the relative difficulties of these programming styles. As we move from Sisal to explicit C with VISA, and to explicit C with message passing, the code becomes increasingly more complex, requiring increasingly more lines of code, and becoming more machine-dependent. The question, then, is whether increased performance justifies the additional programming effort.

Table 3 gives the execution results for LLNL Loop #7, where a constant blocksize of 65536 (2^{16}) double-precision elements is used and *Array Size* represents the total size of the A and B arrays, Sp_1 represents the speedup in going from Sisal to C with VISA (T_{SISAL}/T_{C+VISA}), and Sp_2 represents the speedup in going from C with VISA to C with message passing (T_{C+VISA}/T_{C+MP}). In order to highlight the performance gain achieved by explicit memory management, the blocksize, or number of array elements per processor, was kept constant at 65,536 (2^{16}) double-precision

PEs	Array Size	SISAL	C+VISA		C+MP	
		Time (s)	Time (s)	Sp_1	Time (s)	Sp_2
1	65536	1.8002	1.3232	1.36	0.7462	1.77
2	131072	1.8699	1.3868	1.35	0.7479	1.85
4	262144	1.9307	1.3983	1.38	0.7493	1.86
8	524288	1.9322	1.3922	1.39	0.7518	1.85
16	1048576	2.0143	1.3959	1.44	0.7569	1.84
32	2097152	2.2006	1.4029	1.57	0.7673	1.83
64	4194304	2.5794	1.4173	1.81	0.7882	1.80
Ave.				1.47		1.83

Table 3: Execution times for LLNL Loop #7

PEs	Blocksize	Ratio	SISAL	C+VISA		C+MP	
			Time (s)	Time (s)	Sp_1	Time (s)	Sp_2
1	65536	.002	114.7980	119.6738	0.96	51.9780	2.30
2	32768	.004	58.2668	60.8672	0.96	41.1032	1.48
4	16384	.008	30.2806	30.4173	0.99	21.0470	1.46
8	8192	.016	15.5127	15.4519	1.00	10.6547	1.45
16	4096	.032	9.1281	8.1962	1.11	5.5524	1.48
32	2048	.063	7.2312	5.0998	1.42	3.1722	1.61
64	1024	.125	8.8509	4.4798	1.97	2.6409	1.69
Ave.					1.20		1.64

Table 4: Execution times for SOR

elements. The data reveals that an average speedup of 1.47 is achieved when going from Sisal to explicit C with VISA, which is due to the memory caching optimization rather than the explicit control of tasks. Additionally, an average speedup of 1.83 is achieved when moving from explicit C with VISA to explicit C with message passing, demonstrating the overhead of the VISA system and the effectiveness of the pre-fetching optimization. In terms of space requirements, Sisal uses the minimum: two arrays of size n, one for A and one for B. Explicit C with VISA allocates an additional 7 double-precision locations per array to cache the values of B_i through B_{i+6} so that they need only be retrieved once. Explicit C with message passing also allocates an additional block of 7 elements to store the values of B that reside on the neighboring node.

Table 4 gives the execution results for SOR, where a constant array size of 65536 (2^{16}) double-precision elements and 128 iterations is used. In order to highlight the performance gain achieved by explicit task management, the array size is held constant, causing the blocksize to decrease and the ratio of iterations to blocksize to increase as the number of processors increases. This ratio represents the increasing emphasis being placed on task management. In moving from Sisal to explicit C with VISA, there is an average speedup of 1.20, which starts as a performance decrease and gains as the ratio of iterations to blocksize increases, placing greater emphasis of task management on the total execution time. This initial loss in performance is due to the ability of the Sisal compiler to generate code that is highly optimized, which sometimes outperforms normal hand-coded C. However, this small gain is quickly lost as the complex Sisal task management system is outperformed by the hand-coded C task management. In moving from explicit C with VISA to explicit C with message passing, there is an average speedup of 1.64, again representing the overhead of VISA and the effectiveness of pre-fetching all remote references. The single processor time of explicit C with message passing shows the enormous overhead of synchronization that this problem creates, which is not as visible in the other two approaches due to the overhead of VISA. In terms of space requirements, explicit C with VISA uses the minimal two arrays of size n, one for the previous iteration and one for the current iteration, and pointers are swapped at the end of each iteration. The Sisal compiler also recognizes this optimization, but generates the two swap arrays only after generating an array to hold the initial values, resulting in a space overhead of n elements. The explicit C with message passing uses only the two necessary arrays, but allocates an additional two elements per processor to hold the pre-fetched remote values from neighboring nodes.

5 Related Research

The most common alternatives to programming distributed memory multiprocessors using an explicit parallel language with message passing are distributed memory language compilers, such as FortranD [7], Kali [8], and Superb [15]. These systems offer the advantage of implicit management for both tasks and memory, and allow the programmer to use a familiar programming paradigm: sequential shared memory. Although these systems have had success in implementing some applications, there are several problems that have kept them from wide-spread use:

- Parallelizing a sequentially written program requires extensive dependence analysis that can be hampered with common imperative programming phenomena such as aliasing. Also, symbolic subscript terms with unknown values, coupled subscripts, and nonzero or nonunity coefficients of loop indices often make dependence analysis impossible for even the most sophisticated parallelizing compilers [13].

- Due to the complexity of these compilers and the difficulties in porting them to new machines, their availability is limited to only of few of the currently available distributed memory multiprocessor systems. As stated earlier, such a compiler is not commercially available for the nCUBE/2.

- Though parallelizing/vectorizing compilers have proven to be successful for some applications on shared memory multiprocessors and vector processors with shared memory, they are largely unproven for distributed memory multiprocessors. Also, the way in which data distribution is controlled and the amount of programmer interaction varies widely from system to system, which can make porting an application from one DMMP compiler to another a non-trivial task.

- Programmers have long been aware that the language design has a significant impact on how easily an algorithm can be transformed into working code [11]. Even the so-called "general purpose" languages are recognized as being suited for certain problem solving approaches. The transformation process is more tedious and error prone when the conceptual models supported by the language relate only peripherally to the problem-solving model of the programmer. Unfortunately, though the compilation ideas for these compilers are applicable for a wide range of languages, almost all of these systems offer the same programming language, drastically restricting the choice of languages for distributed memory machines.

By utilizing a strict functional language, we can ease many of the dependence analysis problems for a compiler, such as aliasing and subscript analysis. Also, our runtime-based approach to providing a shared memory paradigm has the advantage of being language independent, offering the possibility of being used by any shared memory compiler, and leaving the programmer with more freedom to choose the best language to match the application, and offering a consistent approach to data distribution and access. However, runtime address translation can be expensive if the translation is not optimized out, and the compiler does need to be modified to generate the appropriate VISA primitives. Strict functional programming languages can also be restrictive in terms of expressibility, sometimes requiring complex and convoluted code to perform simple tasks.

Another area of research that offers a language-independent shared memory paradigm is Distributed Shared Memory [1, 9, 12]. However, the inability to couple parallel tasks tightly with the distribution of data, controlled implicitly by the operating system, can result in misalignment, causing excessive message passing. Also since the granularity of sharing data in these systems is often very large (typically a page), contention, or *false sharing* can occur, in which two unrelated data items exist on the same sharable unit, prohibiting simultaneous access. Since the sharable unit in VISA is an individual data structure, false sharing does not occur.

6 Conclusions

We have introduced the design and implementation of a runtime-based approach to providing a shared memory paradigm and implicit memory management for a distributed memory multiprocessor. Using this runtime system, we have explored the advantages and disadvantages of explicit and implicit programming styles for both task management and memory management.

Sisal with VISA provides implicit management of both tasks and data, and offers reasonable performance while alleviating the programmer from the implementation details of an architecture.

The result is efficient machine-independent code that is portable among a wide range of architectures [3]. Furthermore, since the current Sisal compiler is unaware of distributed memory and costs associated with accessing remote data, we expect a performance gain when such information is exploited by the compiler [14].

Explicit parallel C with VISA offers the ability to increase the performance of an application, but at the cost of increased size, programming effort, and machine-dependence. For our simple programs, an average speedup of 1.34 over Sisal is achieved, but at the cost of increasing the code size by an average factor of 7, and increasing the time required to encode and debug the programs by an average factor of 11.

Explicit parallel C with explicit message passing offers the ability to exploit the problem and machine details to obtain the highest performance for a particular machine. For our programs, average speedups of 1.74 over C with VISA, and 2.34 over Sisal are achieved. Once again, this increase in performance is obtained at the cost of increasing program sizes by an average factor of 2 over explicit C with VISA, and by a an average factor of 15 over Sisal, while increasing the time required to encode and debug the programs by an average factor of 4 over explicit C with VISA, and by an average factor of 40 over Sisal.

The results show that although implicit parallel programming can offer reasonable performance, it is possible to increase the performance by taking explicit control over task management or data management. It is the decision of the applications programmer as to whether the increase in performance warrants the increase in programming effort when moving from implicit to explicit programming styles, but the option should nonetheless be available.

Distributed memory multiprocessors represent today's most powerful computer systems, yet efficient support for high-level abstractions lags. We must make a concerted effort to alleviate the programmer from the details of programming distributed memory multiprocessors, but not at the expense of performance. Clearly this is a challenging goal.

References

[1] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Shared memory for distributed memory multiprocessors. Technical Report Rice COMP TR89-91, Rice University, April 1989.

[2] Wim Böhm, J.C. Browne, David Forslund, Andre Goforth, Ken Kennedy, and James McGraw. Politically incorrect languages for supercomputers – a panel discussion. In *Proceedings of Supercomputing 92*, pages 704–706. IEEE, November 1992.

[3] David Cann. Retire Fortran? A debate rekindled. *Communications of the ACM*, 35(8):81–89, August 1992.

[4] Matthew Haines and Wim Böhm. Thread management in a distributed memory implmentation of sisal. In *Proceedings of the Dataflow Workshop, International Symposium on Computer Architecture*, May 1992. To Appear.

[5] Matthew Haines and Wim Böhm. Task management, virtual shared memory, and multithreading in a distributed memory implementation of sisal. In *Proceedings of Parallel Architectures and Languages Europe*, June 1993. To Appear.

[6] Matthew Haines and Wim Böhm. The VISA user's guide. Technical Report CS-93-102, Colorado State University, Fort Collins, CO, February 1993.

[7] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[8] C. Koelbel and P. Mehrotra. Compiling global name-Space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Computing*, 2(4):440–451, October 1991.

[9] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Yale University, September 1986.

[10] J. R. McGraw, S. K. Skedzielewski, S. J. Allan, R. R. Oldehoeft, J. Glauert, C. Kirkham, W. Noyce, and R. Thomas. SISAL: Streams and iteration in a single assignment language: Reference manual version 1.2. Manual M-146, Rev. 1, Lawrence Livermore National Laboratory, Livermore, CA, March 1985.

[11] Cherri M. Pancake and Donna Bergmark. Do parallel languages respond to the needs of scientific programmers. *IEEE Computer*, 23(12):13–24, December 1990.

[12] Umakishore Ramachandran, Mustaque Ahamad, and M. Yousef A. Khalidi. Unifying synchronization and data transfer in maintaining coherence of distributed shared memory. Technical Report GIT-CS-88/23, Georgia Institute of Technology, June 1988.

[13] Zhiyu Shen, Zhiyuan Li, and Pen-Chung Yew. An emperical study of Fortran programs for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):356–364, July 1990.

[14] R. Wolski and J. Feo. An extended data flow model for program partitioning on NUMA architectures. In *Proceedings of the Second Sisal User Conference*, October 1992.

[15] H. Zima, H. Bast, and M. Gerndt. Superb: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1986.

Appendix A: VISA Functions

- *Allocation*
  - V_ADDRESS **visa_malloc** (int *nelems*, int *size*, map_function *map*, int *map_arg*)
    This function allocates a block of VISA space (*nelems* * *size* bytes), which will be distributed according to *map*, and returns a pointer to the start of the allocated space. A range_map entry is also created and distributed among the nodes, and local space is allocated, according to the map, to store the data structure.
- *Deallocation*
  - void **visa_free** (V_ADDRESS *address*)
    This function returns the given portion of VISA space to the free pool, removes the corresponding range_map entry from each of the range_map tables, and deallocates the local storage used for storing the structure.
- *Access*
  - range_map_type * **find_rm** (V_ADDRESS *address*)
    Return a pointer to the range_map entry corresponding to the given VISA address. This pointer is then passed into each of the access routines as an argument so that the fetch does not have to be done for each access.
  - char **visa_get_c** (V_ADDRESS *address*, range_map_type **rm*)
    int **visa_get_i** (V_ADDRESS *address*, range_map_type **rm*)
    float **visa_get_f** (V_ADDRESS *address*, range_map_type **rm*)
    double **visa_get_d** (V_ADDRESS *address*, range_map_type **rm*)
    These functions return the desired value from the given VISA address. If the range_map entry *rm* is not defined, then the corresponding range_map entry for this structure will be fetched, which is true for all of the access functions.
  - void **visa_get_m** (POINTER *data*, int *size*, V_ADDRESS *address*, range_map_type **rm*)
    This function copies the block of data starting at the given VISA address and for a length of *size* into the local address pointed to by *data*.
  - void **visa_put_c** (char *value*, V_ADDRESS *address*, range_map_type **rm*)
    void **visa_put_i** (int *value*, V_ADDRESS *address*, range_map_type **rm*)
    void **visa_put_f** (float *value*, V_ADDRESS *address*, range_map_type **rm*)
    void **visa_put_d** (double *value*, V_ADDRESS *address*, range_map_type **rm*)
    These functions place *value* into the given VISA address location.
  - void **visa_put_m** (POINTER *data*, int *size*, V_ADDRESS *address*, range_map_type **rm*)
    This function copies the local data block of size *size* and pointed to by *data* into the given VISA address location.
  - void **visa_update_c** (uchar *red*, char *value*, V_ADDRESS *address*, range_map_type **rm*)
    void **visa_update_i** (uchar *red*, int *value*, V_ADDRESS *address*, range_map_type **rm*)
    void **visa_update_f** (uchar *red*, float *value*, V_ADDRESS *address*, range_map_type **rm*)
    void **visa_update_d** (uchar *red*, double *value*, V_ADDRESS *address*, range_map_type **rm*)
    These functions update the value stored in the given VISA address with *value*, according to the reduction *red*. Currently supported reductions include *V_SUM* and *V_PRODUCT*.